O'REILLY®

Second Edition

Java Generics and Collections

Fundamentals and Recommended Practices

Early Release

RAW & UNEDITED

Maurice Naftalin
& Philip Wadler

Java Generics and Collections

Fundamentals and Recommended Practices

SECOND EDITION

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Maurice Naftalin and Philip Wadler

Beijing • Boston • Farnham • Sebastopol • Tokyo

Java Generics and Collections

by Maurice Naftalin and Philip Wadler

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Brian Guerin

Development Editor: Sara Hunter

Production Editor: Aleeya Rahman

Interior Designer: David Futato

Illustrator: Kate Dullea

December 2023: Second Edition

Revision History for the Early Release

- 2023-04-25: First Release
- 2023-06-30: Second Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098136727 for release details.

the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-13666-6

Dedication

We dedicate this book to Joyce Naftalin, Lionel Naftalin, Adam Wadler, and Leora Wadler

—Maurice Naftalin and Philip Wadler

Chapter 1. Subtyping and Wildcards

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

Now that we've covered the basics, we can start to cover more advanced features of generics, such as subtyping and wildcards. In this section, we'll review how subtyping works and we'll see how wildcards let you use subtyping in connection with generics. The Java Collections Framework will serve as the source of our examples; see Part 2 for detail on specific features of the Collections API.

Subtyping and the Substitution Principle

Subtyping is a key feature of object-oriented languages such as Java. In Java, one type is a *subtype* of another if they are related by an `extends` or `implements` clause. Here are some examples:

`Integer`	is a subtype of	`Number`
`Double`	is a subtype of	`Number`

`ArrayList<E>`	is a subtype of	`List<E>`
`List<E>`	is a subtype of	`Collection<E>`
`Collection<E>`	is a subtype of	`Iterable<E>`

Subtyping is transitive, meaning that if one type is a subtype of a second, and the second is a subtype of a third, then the first is a subtype of the third. So, from the last two lines in the preceding list, it follows that `List<E>` is a subtype of `Iterable<E>`. If one type is a subtype of another, we also say that the second is a *supertype* of the first. Every reference type is a subtype of `Object`, and `Object` is a supertype of every reference type. We also define the subtype relationship so that every type is a subtype of itself.

The most important property of subtyping is summarized in the Substitution Principle [1]

> *Substitution Principle: wherever a value of type `T` is expected, you can provide instead a value of a subtype of `T`.*

For example, a variable of a given type may be assigned a value of any subtype of that type, and a method with a parameter of a given type may be invoked with an argument of any subtype of that type.

Consider the interface `Collection<E>`. One of its methods is add, which takes a parameter of type `E`:

```
interface Collection<E> {
  public boolean add(E elt);
  ...
}
```

According to the Substitution Principle, if we have a collection of numbers we may add an integer or a double to it, because `Integer` and `Double` are subtypes of `Number`.

```
List<Number> nums = new ArrayList<>();
nums.add(2);
```

```
nums.add(0.25);
assert nums.equals(List.of(2, 0.25));
```

Here, subtyping is used in two ways for each method call. The first call is permitted because `nums` has the type `List<Number>`, which is a subtype of `Collection<Number>`, and `2` has the type `Integer` (thanks to boxing), which is a subtype of `Number`. The second call is similarly permitted. In both calls, the `E` in `List<E>` is taken to be `Number`.

It may seem reasonable to expect that since `Integer` is a subtype of `Number`, it follows that `List<Integer>` is a subtype of `List<Number>`. But this is *not* the case, because, if it were, the Substitution Principle would rapidly get us into trouble. To see why it is not safe to assign a value of type `List<Integer>` to a variable of type `List<Number>`, consider the following code:

```
List<Integer> ints = new ArrayList<>();
List<Number> nums = ints;         // ❶ can't be allowed
nums.add(3.14);
```

This code assigns the variable `ints` to point to a list of `Integer`, and then—if the Substitution Principle were to require ❶ to succeed—assigns `nums` to point to the *same* list; in that case, the call in the following line—which is clearly legal—would add a `Double` to this list, meaning that the variable `ints`, typed as a `List<Integer>` would be pointing at a list containing a `Double`, and Java's type system would be broken. Obviously, this can't be allowed; the solution is to outlaw assignment ❶. If `List<Integer>` is prevented from being a subtype of `List<Number>`, then the Substitution Principle does not apply and ❶ can be reported as a compile error.

What about the reverse? Can we take `List<Number>` to be a subtype of `List<Integer>`? No, that doesn't work either, as shown by the following code:

```
List<Number> nums = new ArrayList<>();
nums.add(3.14);
List<Integer> ints = nums;        // ❷ can't be allowed
```

where allowing ❷ leads to the same problem—once again, a variable typed as `List<Integer>` is pointing to a list containing a `Double`. So `List<Number>` is not a subtype of `List<Integer>`, and since we've already seen that `List<Integer>` is not a subtype of `List<Number>`, all we have is the trivial case where `List<Integer>` is a subtype of itself. (We also have the more intuitive relationship that `List<Integer>` is a subtype of `Collection<Integer>`.)

Arrays behave quite differently; with them, `Integer[]` *is* a subtype of `Number[]`. We will compare the treatment of lists and arrays later (see "Arrays").

Sometimes we would like lists to behave more like arrays, in that we want to accept not only a list with elements of a given type, but also a list with elements of any subtype of a given type. For this purpose, we use *wildcards*.

Wildcards with extends

Another method in the `Collection` interface is `addAll`, which adds all members of one collection to another:

```
interface Collection<E> {
  ...
  public boolean addAll(Collection<? extends E> c);
  ...
}
```

Clearly, given a collection of elements of type `E`, we should be able to add all members of another collection with elements of type `E`. The quizzical phrase "`? extends E`" means that it is also OK to add all members of a collection with elements of any type that is a *subtype* of `E`. The question mark is called a *wildcard*; it stands for some as-yet-unknown subtype of `E`.

Here is an example. We create an empty list of numbers, and add to it first a list of integers and then a list of doubles:

```
List<Number> nums = new ArrayList<>();
List<Integer> ints = List.of(1, 2);
List<Double> dbls = List.of(1.0, 0.5);
```

```
nums.addAll(ints);
nums.addAll(dbls);
assert nums.equals(List.of(1, 2, 1.0, 0.5));
```

The first call is permitted because `nums` has type `List<Number>`, which is a subtype of `Collection<Number>`, and `ints` has type `List<Integer>`, which is a subtype of `Collection<Integer>`, which is in turn a subtype of `Collection<? extends Number>`. The second call is similarly permitted. In both calls, `E` is taken to be `Number`. If the parameter declaration for `addAll` had been written without the wildcard, then the calls to add lists of integers and doubles to a list of numbers would not have been permitted; you would only have been able to add a list that was explicitly declared to be a list of numbers.

We can also use wildcards when declaring variables. Here is a variant of the introductory example of the preceding section, changed by adding a wildcard to the second line:

```
List<Integer> ints = new ArrayList<>();
List<? extends Number> nums = ints;  // ❸ now legal
nums.add(3.14);                      // ❹ can't be allowed
```

Previously, line ❸ caused a compile-time error (because `List<Integer>` is not a subtype of `List<Number>`), but line ❹ was fine (because a double is a number, so you can add a double to a `List<Number>`). Now line ❸ is fine (because `List<Integer>` is a subtype of `List<? extends Number>`), but line ❹ has to be a compile error: you cannot add a double to a `List<? extends Number>`, since you don't know the type represented by the wildcard; all you know is that it is some subtype of `Number`. Otherwise, as before, the last line would result in the variable `ints`, typed as a `List<Integer>`, pointing to a list containing a double.

In general, if a structure contains elements with a type of the form "`? extends E`", we can get elements out of the structure, but we cannot put elements into the structure. To put elements into the structure we need another kind of wildcard, as explained in the next section.

Wildcards with super

Here is a method, from the convenience class `Collections`, that copies the elements from a source list into a destination list:

```
public static <T> void copy(List<? super T> dst, List<? extends T>
src) {
  for (int i = 0; i < src.size(); i++) {
    dst.set(i, src.get(i));
  }
}
```

The quizzical phrase "`? super T`" means that the destination list may have elements of any type that is a *supertype* of `T`, just as the source list may have elements of any type that is a *subtype* of `T`.

Here is a sample call.

```
List<Object> objs = Stream.of(2, 3.14,
"four").collect(Collectors.toList());
List<Integer> ints = List.of(5, 6);
Collections.copy(objs, ints);
assert objs.equals(List.of(5, 6, "four"));
```

As with any generic method, the type parameter may be inferred or may be given explicitly. In this case, there are four possible choices, all of which type-check and all of which have the same effect:

```
Collections.copy(objs, ints);
Collections.<Object>copy(objs, ints);
Collections.<Number>copy(objs, ints);
Collections.<Integer>copy(objs, ints);
```

The first call leaves the type parameter implicit; it is taken to be `Integer`, since that is the most specific choice that works. In the third line, it is taken to be `Number`. The call is permitted because `objs` has type `List<Object>`, which is a subtype of `List<? super Number>` (since `Object` is a supertype of `Number`, as required by the `super` wildcard) and `ints` has type `List<Integer>`, which is a subtype of `List<? extends Number>` (since `Integer` is a subtype of `Number`, as required by the `extends`

wildcard).

We could also declare the method with several possible *signatures* (a method signature is the combination of the method identifier, its type parameters, and the types and order of its parameters).

```
public static <T> void copy(List<T> dst, List<T> src)
public static <T> void copy(List<T> dst, List<? extends T> src)
public static <T> void copy(List<? super T> dst, List<T> src)
public static <T> void copy(List<? super T> dst, List<? extends T>
src)
```

The first of these is too restrictive, as it only permits calls when the destination and source have exactly the same type. The remaining three are equivalent for calls that use implicit type parameters, but differ for explicit type parameters. For the example calls above, the second signature works only when the type parameter is `Object`, the third signature works only when the type parameter is `Integer`, and the last signature works (as we have seen) for all three type parameters—i.e., `Object`, `Number`, and `Integer`. When writing a method signature, always use wildcards where you can, since this permits the widest range of calls.

The Get and Put Principle

It may be good practice to insert wildcards whenever possible, but how do you decide *which* wildcard to use? Where should you use `extends`, where should you use `super`, and where is it inappropriate to use a wildcard at all?

Fortunately, a simple principle determines which is appropriate.

> *The Get and Put Principle: use an `extends` wildcard when you only get values out of a structure, use a `super` wildcard when you only put values into a structure, and don't use a wildcard when you both get and put.*

In *Effective Java* ([Bloch17]), Joshua Bloch gives this principle the mnemonic PECS (producer extends, consumer super). We already saw this principle at work in the signature of the `copy` method:

```
public static <T> void copy(List<? super T> dst, List<? extends T>
```

```
src)
```

The method gets values out of the source `src`, so it is declared with an `extends` wildcard, and it puts values into the destination `dst`, so it is declared with a `super` wildcard.

Whenever you use an iterator or a stream, you are getting values out of a structure, so you must use an `extends` wildcard. Here is the iterator version of a method that takes a collection of numbers, converts each to a double, and sums them up:

```
public static double sum(Collection<? extends Number> nums) {
  double s = 0.0;
  for (Number num : nums) s += num.doubleValue();
  return s;
}
```

The stream equivalent doesn't declare a `Number` variable explicitly, but the type constraint on the argument is exactly the same:

```
public static double sum(Collection<? extends Number> nums) {
   return nums.stream().mapToDouble(Number::doubleValue).sum();
}
```

Since these methods use `extends`, the following calls are all legal:

```
List<Integer> ints = List.of(1,2,3);
assert sum(ints) == 6.0;

List<Double> doubles = List.of(2.5,3.5);
assert sum(doubles) == 6.0;

List<Number> nums = List.of(1,2,2.5,3.5);
assert sum(nums) == 9.0;
```

The first two calls would not be legal if `extends` was not used.

Whenever you use the `add` method, you're putting values into a structure, so you should use a `super` wildcard. Here is a method that takes a collection of numbers and an integer `n`, and puts the first `n` integers, starting from zero, into the collection:

```
public static void count(Collection<? super Integer> ints, int n) {
  for (int i = 0; i < n; i++) ints.add(i);
}
```

(There is no simple stream-based equivalent to this method.) Since the parameter to this method uses `super`, the following calls are all legal:

```
List<Integer> ints1 = new ArrayList<>();
count(ints1, 5);
assert ints1.equals(List.of(0, 1, 2, 3, 4));

List<Number> nums1 = new ArrayList<>();
count(nums1, 5); nums1.add(5.0);
assert nums1.equals(List.of(0, 1, 2, 3, 4, 5.0));

List<Object> objs1 = new ArrayList<>();
count(objs1, 5); objs1.add("five");
assert objs1.equals(List.of(0, 1, 2, 3, 4, "five"));
```

The last two calls would not be legal if `super` was not used.

Whenever you both put values into and get values out of the same structure, you should not use a wildcard.

```
public static double sumCount(Collection<Number> nums, int n) {
  count(nums, n);
  return sum(nums);
}
```

The collection is passed to both `sum` and `count`, so its element type must both extend `Number` (as `sum` requires) and be a supertype of `Integer` (as `count` requires). The only two classes that satisfy both of these constraints are `Number` and `Integer`; we have picked the first of these. Here is a sample call:

```
List<Number> nums2 = new ArrayList<>();
double sum = sumCount(nums2,5);
assert sum == 10;
```

Since there is no wildcard, the argument must be a collection of `Number`.

If you don’t like having to choose between `Number` and `Integer`, it might occur to you that if Java let you write a wildcard with both `extends` and

`super`, you would not need to choose. For instance, we could write the following:

```
double sumCount(Collection<? extends Number super Integer> coll, int
n)
// not legal Java!
```

Then we could call `sumCount` on either a collection of numbers or a collection of integers. But Java *doesn't* permit this. The only reason for outlawing it is simplicity, and conceivably Java might support such notation in the future. But, for now, if you need to both get and put then don't use wildcards.

The Get and Put Principle also works the other way around. If an `extends` wildcard is present, pretty much all you will be able to do is get but not put values of that type; and if a `super` wildcard is present, pretty much all you will be able to do is put but not get values of that type. For example, consider the following code, which uses a list declared with an `extends` wildcard:

```
List<Integer> ints = new ArrayList<>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
double dbl = sum(nums);     // ok
nums.add(3.14);             // compile-time error
```

The call to `sum` is fine, because it gets values from the list, but the call to `add` is not, because it puts a value into the list. This is just as well, since otherwise we could add a double to a list of integers!

Conversely, consider the following code, which uses a list declared with a `super` wildcard:

```
List<Object> objs = new ArrayList<>();
objs.add(1);
objs.add("two");
List<? super Integer> ints = objs;
ints.add(3);              // ok
double dbl = sum(ints);  // compile-time error
```

Now the call to `add` is fine, because it puts a value into the list, but the call to `sum` is not, because it gets a value from the list. This is just as well, because the

sum of a list containing a string makes no sense!

The exception proves the rule, as the saying goes, and indeed each of these rules has an exception. The one value that you can put into a type declared with an `extends` wildcard is `null`, the only value that belongs to every reference type:

```
List<Integer> ints = new ArrayList<>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
nums.add(null);    // ok
assert nums.equals(Arrays.asList(1, 2, null));
```

Similarly, the only values that you can get from a type declared with a `super` wildcard are those of type `Object`, which is the supertype of every reference type:

```
List<Object> objs = List.of(1,"two");
List<? super Integer> ints = objs;
String str = "";
for (Object obj : ints) str += obj.toString();
assert str.equals("1two");
```

You may find it helpful to think of `? extends T` as being an unknown type in an interval bounded by the type of `null` below and by `T` above (where the type of `null` is a subtype of every reference type). Similarly, you may think of `? super T` as being an unknown type in an interval bounded by `T` below and by `Object` above (see Figure 1-1).

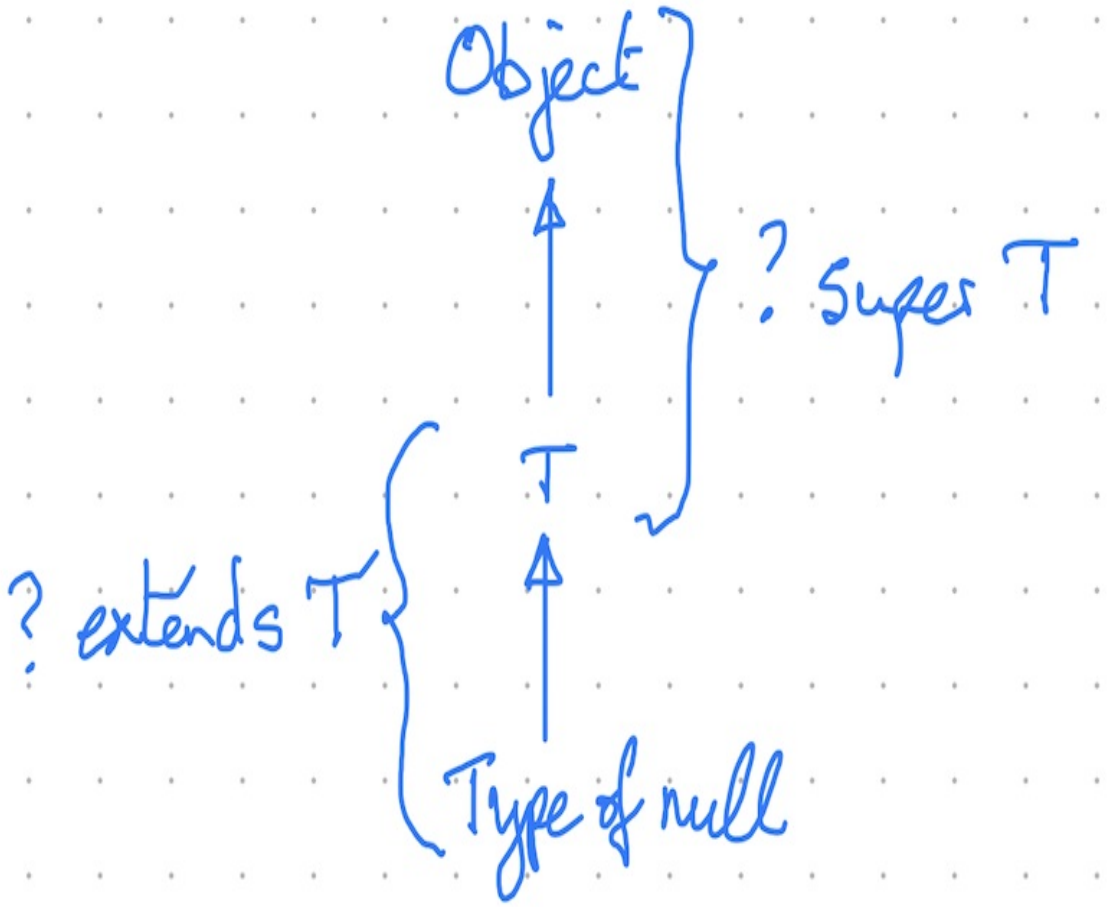

Figure 1-1. Bounded types: extends and super

It is tempting to think that an `extends` wildcard ensures immutability or at least unmodifiability (see "Immutability and Unmodifiability"), but it does not. As we saw earlier, given a list of type `List<? extends Number>`, you can still add `null` values to the list. You can also remove list elements (using `remove`, `removeAll`, or `retainAll`) or permute the list (using `swap`, `sort`, or `shuffle` in the convenience class `Collections`; see "Changing the Order of List Elements"). The problems and advantages of achieving unmodifiability or immutability are explored in "Immutability and Unmodifiability" and further in [Link to Come].

Because `String` is final and can have no subtypes, you might expect that `List<String>` is the same type as `List<? extends String>`. But in fact the former is a subtype of the latter, not the same type, as can be seen by an application of our principles. The Substitution Principle tells us it is a subtype,

because it is fine to pass a value of the former type where the latter is expected. The Get and Put Principle tells us that it is not the same type, because we can add a string to a value of the former type but not the latter.

Arrays

It is instructive to compare the treatment of lists and arrays in Java, keeping in mind the Substitution Principle and the Get and Put Principle.

In Java, array subtyping is *covariant*, meaning that type `S[]` is considered to be a subtype of `T[]` whenever `S` is a subtype of `T`. Consider the following code fragment, which allocates an array of integers, assigns it to a variable typed as a `Number` array, and then attempts to store a double in the array:

```
Integer[] ints = {0};
Number[] nums = ints;
nums[0] = 3.14;  // array store exception
```

Something is wrong with this program since, if it were to succeed, the variable `ints`, typed as an `Integer[]`, would be pointing at an array containing a `Double`, and once again Java's type system would be broken. Where is the problem? Since `Integer[]` is considered a subtype of `Number[]`, according to the Substitution Principle the assignment on the second line must be legal. Instead, the problem is caught at run time on the third line. When an array is allocated (as on the first line), it is tagged with its reified type (a run-time representation of its component type, in this case, `Integer`), and every time a value is stored in the array (as on the third line), an array store exception is raised if the reified type is not compatible with the assigned value (in this case, a double cannot be stored into an array of `Integer`).

In contrast, the subtyping relation for generics is *invariant*, meaning that type `List<S>` is *not* considered to be a subtype of `List<T>`, except in the trivial case where `S` and `T` are identical. Here is a code fragment analogous to the preceding one, with lists replacing arrays:

```
List<Integer> ints = Arrays.asList(0);
List<Number> nums = ints;  // compile-time error
nums.set(0, 3.14);
```

Since `List<Integer>` is not considered to be a subtype of `List<Number>`, the problem is detected on the second line, not the third, and it is detected at compile time, not run time.

Wildcards reintroduce covariant subtyping for generics, in that type `List<S>` *is* considered to be a subtype of `List<? extends T>` when `S` is a subtype of `T`. Here is a third variant of the fragment:

```
List<Integer> ints = Arrays.asList(0);
List<? extends Number> nums = ints;
nums.set(0, 3.14);  // compile-time error
```

As with arrays, the third line is in error, but, in contrast to arrays, the problem is detected at compile time, not run time. The assignment violates the Get and Put Principle, because you cannot put a value into a type declared with an `extends` wildcard.

Wildcards also introduce *contravariant* subtyping for generics, in that type `List<S>` is considered to be a *subtype* of `List<? super T>` when `S` is a *supertype* of `T` (as opposed to a subtype). Arrays do not support contravariant subtyping. For instance, recall that the method `count` accepted a parameter of type `Collection<? super Integer>` and filled it with integers. There is no equivalent way to do this with an array, since Java does not permit you to write `(? super Integer)[]`.

Detecting problems at compile time rather than at run time brings two advantages, one minor and one major. The minor advantage is that it is more efficient. The system does not need to carry around a description of the element type at run time, and the system does not need to check against this description every time an assignment into an array is performed. The major advantage is that a common family of errors is detected by the compiler. This improves every aspect of the program's life cycle: coding, debugging, testing, and maintenance are all made easier, quicker, and less expensive.

Apart from the fact that typing errors are caught earlier, there are many other reasons to prefer collection classes to arrays. Collections are more flexible than arrays; the only operations supported on arrays are to get or set a component, and the number of elements they can contain is fixed. Collections support many

additional operations, including testing for containment, adding and removing elements, and combining two collections. Collections may be lists (where order is significant and elements may be repeated, and further operations are available), sets (where order is not significant and elements may not be repeated), or queues (predominantly used in workflow situations), and a number of representations are available, including arrays, linked lists, trees, and hash tables. Specialised collections are available for situations requiring efficient concurrent access. Finally, although this is not an inherent advantage of collections over arrays, the convenience class `Collections` offers operations to rotate or shuffle a list, to find the maximum of a collection, to make a collection unmodifiable or synchronized, and others—many more than are provided by the corresponding convenience class `Arrays`.

Nonetheless, there are situations in which arrays are preferable to collections. In particular, arrays of primitive types are much more efficient than either arrays or collections of reference types, since they don't involve boxing, and they use less memory with much better spatial locality (see "Performance"). Further, assignments into a primitive array need not check for an array store exception, because arrays of primitive type do not have subtypes. These advantages may lead you to replace collections with arrays in performance-critical code sections. As always, you should measure comparative performance to justify such a design, bearing in mind that future compiler and library design improvements may change the relative performance balance. Finally, in some cases arrays may be preferable for reasons of compatibility.

To summarize, it is better to detect errors at compile time rather than run time, but Java arrays are forced to detect certain errors at run time by the decision to make array subtyping covariant. Was this a good decision? Before the advent of generics, it was absolutely necessary. For instance, look at the following methods, which are used to sort any array or to fill an array with a given value:

```
public static void sort(Object[] a);
public static void fill(Object[] a, Object val);
```

Thanks to covariance, these methods can be used to sort or fill arrays of any reference type. Without covariance and without generics, there would have been no way to declare methods that apply for all types. With generics, however,

covariant arrays became unnecessary: these methods could now have the following signatures, directly stating that they work for all types:

```
public static <T> void sort(T[] a);
public static <T> void fill(T[] a, T val);
```

In some sense, covariant arrays are an artifact of the lack of generics in earlier versions of Java. Once you have generics, covariant arrays are probably the wrong design choice, and the only reason for retaining them is backward compatibility.

Sections ???–??? discuss inconvenient interactions between generics and arrays. For many purposes, it may be sensible to consider arrays as an obsolete type. We return to this point in [Link to Come].

Wildcards Versus Type Parameters

The `contains` method checks whether a collection contains a given object, and its generalization, `containsAll`, checks whether a collection contains every element of another collection. This section presents two alternative approaches to giving generic signatures for these methods. The first approach uses wildcards and is the one used in the Java Collections Framework. The second approach uses type parameters.

Wildcards Here are the types that the methods have in Java with generics:

```
interface Collection<E> {
  ...
  public boolean contains(Object o);
  public boolean containsAll(Collection<?> c);
  ...
}
```

The first method does not use generics at all! The second method is our first sight of an important abbreviation. The type `Collection<?>` stands for:

```
Collection<? extends Object>
```

Extending `Object` is one of the most common uses of wildcards, so it makes

sense to provide a short form for writing it.

These methods let us test for membership and containment:

```
Object obj = "one";
List<Object> objs = List.of("one", 2, 3.5, 4);
List<Integer> ints = List.of(2, 4);
assert objs.contains(obj);
assert objs.containsAll(ints);
assert ! ints.contains(obj);
assert ! ints.containsAll(objs);
```

The given list of objects contains both the string `"one"` and the given list of integers, but the given list of integers does not contain the string `"one"`, nor does it contain the given list of objects.

The tests `ints.contains(obj)` and `ints.containsAll(objs)` might seem silly. Of course, a list of integers won't contain an arbitrary object, such as the string `"one"`. But it is permitted because sometimes such tests might succeed:

```
Object obj = 1;
List<Object> objs = List.of(1, 3);
List<Integer> ints = List.of(1, 2, 3, 4);
assert ints.contains(obj);
assert ints.containsAll(objs);
```

In this case, the object may be contained in the list of integers because it happens to be an integer, and the list of objects may be contained within the list of integers because every object in the list happens to be an integer.

Type Parameters You might reasonably choose an alternative design for collections—a design in which you can only test containment for subtypes of the element type:

```
interface MyCollection<E> {  // alternative design
  ...
  public boolean contains(E o);
  public boolean containsAll(Collection<? extends E> c);
  ...
}
```

Say we have a class `MyList` that implements `MyCollection`. Now the tests are legal only one way around:

```
Object obj = "one";
MyList<Object> objs = MyList.of("one", 2, 3.5, 4);
MyList<Integer> ints = MyList.of(2, 4);
assert objs.contains(obj);
assert objs.containsAll(ints);
assert ! ints.contains(obj);             // compile-time error
assert ! ints.containsAll(objs);         // compile-time error
```

The last two tests are illegal, because the type declarations now require that we can only test whether a list contains an element of a subtype of that list. So we can check whether a list of objects contains a list of integers, but not the other way around.

The library designers chose wildcards over type parameters in the case of `contains`, `containsAll`, and other methods in `Collection`, `List`, and `Map` that test for, or remove values from, the collection. There are arguments both for and against this choice: we will explore them in ???.

Wildcard Capture

When a generic method is invoked, the type parameter may be chosen to match the unknown type represented by a wildcard. This is called *wildcard capture*.

Consider the method `reverse` in the convenience class `java.util.Collections`, which accepts a list of any type and reverses it. It can be given either of the following two signatures, which are equivalent:

```
public static void reverse(List<?> list);
public static <T> void reverse(List<T> list);
```

The wildcard signature is slightly shorter and clearer, and is the one used in the library.

If you use the second signature, it is easy to implement the method:

```
public static<T> void reverse(List<T> list) {
  List<T> tmp = new ArrayList<>(list);
```

```
    for (int i = 0; i < list.size(); i++) {
      list.set(i, tmp.get(list.size()-i-1));
    }
  }
```

This copies the argument into a temporary list, and then writes from the copy back into the original in reverse order.

If you try to use the first signature with a similar method body, it won't work:

```
  public static void reverse(List<?> list) {
    List<Object> tmp = new ArrayList<>(list);
    for (int i = 0; i < list.size(); i++) {
      list.set(i, tmp.get(list.size()-i-1));  // compile-time error
    }
  }
```

Now it is not legal to write from the copy back into the original, because we are trying to write from a list of objects into a list of unknown type. Replacing `List<Object>` with `List<?>` won't fix the problem, because now we have two lists with (possibly different) unknown element types.

Instead, you can implement the method with the first signature by implementing a private method with the second signature, and calling the second from the first:

```
  public static void reverse(List<?> list) { rev(list); }
  private static <T> void rev(List<T> list) {
    List<T> tmp = new ArrayList<>(list);
    for (int i = 0; i < list.size(); i++) {
      list.set(i, tmp.get(list.size()-i-1));
    }
  }
```

Here we say that the type variable `T` has *captured* the wildcard. This is a generally useful technique when dealing with wildcards, and it is worth knowing.

Another reason to know about wildcard capture is that it can show up in error messages, even if you don't use the above technique. In general, each occurrence of a wildcard is taken to stand for some unknown type. If the compiler prints an error message containing this type, it is referred to as `capture of ?`. For instance, with the OpenJDK compiler at Java 19, the incorrect version of

`reverse` generates the following error message:

```
Capture.java:8: error: incompatible types: Object cannot be converted
to CAP#1
            list.set(i, tmp.get(list.size()-i-1));  // compile-time
error
                                 ^
  where CAP#1 is a fresh type-variable: CAP#1 extends Object from
capture of ?
```

`CAP#1` is the name that the compiler has given to the type of the elements of `list`. If there is more than one distinct wildcard, each represents a different unknown type, so they will be assigned different names—even if the type associated with each is the same, for example `capture of ?` or, in the case of a bounded wildcard, `capture of ? extends Number`.

Restrictions on Wildcards

Wildcards may not appear at the top level in class instance creation expressions (`new`), in explicit type parameters in generic method calls, or in supertypes (`extends` and `implements`).

Instance Creation In a class instance creation expression, if the type is a parameterized type, then none of the type parameters may be wildcards. For example, the following are illegal:

```
List<?> list = new ArrayList<?>();  // compile-time error
Map<String, ? extends Number> map
  = new HashMap<String, ? extends Number>();  // compile-time error
```

This is usually not a hardship. The Get and Put Principle tells us that if a structure contains a wildcard, we should only get values out of it (if it is an `extends` wildcard) or only put values into it (if it is a `super` wildcard). For a structure to be useful, we must do both. Therefore, we usually create a structure at a precise type, even if we use wildcard types to put values into or get values from the structure, as in the following example:

```
List<Number> nums = new ArrayList<Number>();
List<? super Number> sink = nums;
```

```
List<? extends Number> source = nums;
for (int i=0; i<4; i++) sink.add(i);
int sum = nums.stream().mapToInt(Number::intValue).sum();
assert sum == 6;
```

Here wildcards appear in the second and third lines, but not in the first line that creates the list.

Only top-level parameters in instance creation are prohibited from containing wildcards. Nested wildcards are permitted. Hence, the following is legal:

```
List<List<?>> lists = new ArrayList<List<?>>();
lists.add(List.of(1,2,3));
lists.add(List.of("four","five"));
assert lists.equals(List.of(List.of(1, 2, 3), List.of("four",
"five")));
assert lists.getFirst().getFirst().toString().equals("1");
```

Even though the list of lists is created at a wildcard type, each individual list within it has a specific type: the first is a list of integers and the second is a list of strings. The wildcard type prohibits us from extracting elements from the inner lists as any type other than `Object`, but since that is the type used by `toString`, this code is well typed.

One way to remember the restriction is that the relationship between wildcards and ordinary types is similar to the relationship between interfaces and classes—wildcards and interfaces are more general, ordinary types and classes are more specific, and instance creation requires the more specific information. Consider the following three statements:

```
List<?> list = new ArrayList<Object>();   // ok
List<?> list = new List<Object>()  // compile-time error
List<?> list = new ArrayList<?>()  // compile-time error
```

The first is legal; the second is illegal because an instance creation expression requires a class, not an interface; and the third is illegal because an instance creation expression requires an ordinary type, not a wildcard.

You might wonder why this restriction is necessary. The Java designers had in mind that every wildcard type is shorthand for some ordinary type, so they believed that ultimately every object should be created with an ordinary type. It

is not clear whether this restriction is necessary, but it is unlikely to be a problem. (We tried hard to contrive a situation in which it was a problem, and we failed!)

Generic Method Calls If a generic method call includes explicit type parameters, those type parameters must not be wildcards. For example, say we have the following generic method:

```
class Lists {
  public static <T> List<T> factory() { return new ArrayList<T>(); }
}
```

You may choose for the type parameters to be inferred, or you may pass an explicit type parameter. Both of the following are legal:

```
List<?> list = Lists.factory();
List<?> list = Lists.<Object>factory();
```

If an explicit type parameter is passed, it must not be a wildcard:

```
List<?> list = Lists.<?>factory();  // compile-time error
```

As before, nested wildcards are permitted:

```
List<List<?>> list = Lists.<List<?>>factory();  // ok
```

The motivation for this restriction is similar to the previous one. Again, it is not clear whether it is necessary, but it is unlikely to be a problem.

Supertypes When a class instance is created, it invokes the constructor for its supertype. Hence, any restriction that applies to instance creation must also apply to supertypes. In a class declaration, if the supertype or any superinterface has type parameters, these types must not be wildcards.

For example, this declaration is illegal:

```
class AnyList extends ArrayList<?> {...} // compile-time error
```

And so is this:

```
class AnotherList implements List<?> {...} // compile-time error
```

But, as before, nested wildcards are permitted:

```
class NestedList extends ArrayList<List<?>> {...} // ok
```

The motivation for this restriction is similar to the previous two. As before, it is not clear whether it is necessary, but it is unlikely to be a problem.

---

[1] Liskov87] provides a definition of behavioral subtyping in terms of the ability to substitute values of one type for values of another without any change in program behavior. Applying this definition in reverse gives the design principle often called the *Liskov Substitution Principle*. We discuss its application to the Collections Framework in Chapter 5 and [Link to Come].

Chapter 2. Comparison and Bounds

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

Now that we have the basics, let's look at some more advanced uses of generics. This chapter describes the interfaces `Comparable<T>` and `Comparator<T>`, which are used to support comparison on elements. These interfaces are useful if, for instance, you want to find the maximum element of a collection or to sort a list. Along the way, we will introduce bounds on type variables, an important feature of generics that is particularly useful in combination with the `Comparable<T>` interface.

Comparable<T>

The interface `Comparable<T>` declares a single instance method for comparing one object with another:

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

The `compareTo` method returns an integer value that is negative, zero, or

positive depending upon whether the receiver—this object—is less than, equal to, or greater than the argument. When a class implements `Comparable`, the ordering specified by this interface is called the *natural ordering* for that class.

Typically, an object belonging to a class can only be compared with an object belonging to the same class. For instance, `Integer` implements `Comparable<Integer>`:

```
Integer int0 = 0;
Integer int1 = 1;
assert int0.compareTo(int1) < 0;
```

The comparison returns a negative number, since `0` precedes `1` under numerical ordering. Similarly, `String` implements `Comparable<String>`:

```
String str0 = "zero";
String str1 = "one";
assert str0.compareTo(str1) > 0;
```

This comparison returns a positive number, since `"zero"` follows `"one"` under alphabetic ordering.

The type parameter to the interface allows nonsensical comparisons to be caught at compile time:

```
Integer i = 0;
String s = "one";
assert i.compareTo(s) < 0; // compile-time error
```

You can compare an integer with an integer or a string with a string, but attempting to compare an integer with a string is a compile-time error.

Comparison is not supported between arbitrary numerical types:

```
Number m = Integer.valueOf(2);
Number n = Double.valueOf(3.14);
assert m.compareTo(n) < 0; // compile-time error
```

Here the comparison is illegal, because the `Number` class does not implement the `Comparable` interface.

Comparison differs from equality in that it does not accept a `null` argument. If `x` is not `null`, `x.equals(null)` must return `false`, while `x.compareTo(null)` must throw a `NullPointerException`.

We adapt standard idioms for comparison, writing `x.compareTo(y) < 0` instead of `x < y`, and writing `x.compareTo(y) <= 0` instead of `x <= y`.

Contract for Comparable The contract for the `Comparable<T>` interface specifies three properties. The properties are defined using the sign function, which is defined such that `sgn(x)` returns -1, 0, or 1, depending on whether `x` is negative, zero, or positive.

First, comparison is anti-symmetric. Reversing the order of arguments reverses the result:

```
sgn(x.compareTo(y)) == -sgn(y.compareTo(x))
```

This generalizes the property for numbers: `x < y` if and only if `y > x`. It is also required that `x.compareTo(y)` raises an exception if and only if `y.compareTo(x)` raises an exception.

Taking `x` and `y` to be the same gives us `sgn(x.compareTo(x)) == -sgn(x.compareTo(x))`. It follows that

```
x.compareTo(x) == 0
```

so comparison is reflexive—that is, every value compares as the same as itself.

Second, comparison is transitive. If one value is smaller than a second, and the second is smaller than a third, then the first is smaller than the third:

```
if  x.compareTo(y) < 0 and  y.compareTo(z) < 0  then  x.compareTo(z) <
0
```

This generalizes the property for numbers: if `x < y` and `y < z` then `x < z`.

Third, comparison is a congruence. If two values compare as the same then they compare the same way with any third value:

```
if  x.compareTo(y) == 0 then  sgn(x.compareTo(z)) ==
```

```
sgn(y.compareTo(z))
```

This generalizes the property for numbers: if `x == y` then `x < z` if and only if `y < z`. Presumably, it is also required that if `x.compareTo(y) == 0` then `x.compareTo(z)` raises an exception if and only if `y.compareTo(z)` raises an exception, although this is not explicitly stated.

Consistent with Equals The contract for `Comparable` strongly recommends that the `compareTo` method should be *consistent with equals*—that is, that two objects should satisfy the `equals` method if and only if they compare as the same:

```
x.equals(y)  if and only if  x.compareTo(y) == 0
```

Notice that this is a recommendation, not a mandatory part of the contract. But in fact it is the normal case: most classes having a natural ordering do comply with this recommendation. If you are designing a class that implements `Comparable`, you should not ignore it without good reason. The best-known example in the platform library that contravenes this recommendation is `java.math.BigDecimal`: two instances of this class representing the same value but with different precisions, for example 4.0 and 4.00, will compare as the same but not satisfy the `equals` method. One consequence of this design choice is that values of such a class cannot be reliably stored in an internally ordered collection like `NavigableSet` or `NavigableMap`, as we will see in Chapter 8. A discussion of possible reasons for choosing inconsistency with `equals`, and the wider consequences of this choice, will be found in [Link to Come].

Look Out for This! It's worth pointing out a subtlety in the definition of comparison. Suppose that you have a series of `Event` objects, each described by its name and the number of milliseconds at which it occurred before or after the present moment. Here is one way to define the natural order for `Event` to be the same as the timing order:

```
record Event(String name, int millisecs) implements Comparable<Event>
{
    public int compareTo(Event other) {
        return this.millisecs < other.millisecs ? -1
```

```
                : this.millisecs == other.millisecs ? 0
                : 1;
    }
}
```

The conditional expression returns -1, 0, or 1 depending on whether the receiver is less than, equal to, or greater than the argument. At first sight, you might think that the following code would work just as well, since `compareTo` is permitted to return any negative integer if the receiver is less than the argument, and any positive integer if it is greater:

```
record Event(String name, int millisecs) implements Comparable<Event>
{
    public int compareTo(Event other) {
        // bad implementation — don't do this
        return this.millisecs - other.millisecs;
    }
}
```

But if the two values being compared have opposite signs, calculating the difference between them may result in overflow—that is, a value outside the range that can be stored in an integer, between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`. In “Comparator<T>” we will see how the factory methods of `Comparator` provide concise ways of comparing two objects without encountering this problem.

Maximum of a Collection

In this section, we show how to use the `Comparable` interface to find the maximum element in a collection.We begin with a simplified version. The signature of the version actually found in the Collections Framework is more complicated, and later we will see why.

Here is the code to find the maximum element in a nonempty collection, from the class `Collections`:

```
public static <T extends Comparable<T>> T max(Collection<T> coll) {
    Iterator<? extends T> i = coll.iterator();
    T candidate = i.next();
    while (i.hasNext()) {
```

```
            T next = i.next();
            if (next.compareTo(candidate) > 0)
                candidate = next;
        }
        return candidate;
    }
```

We first saw generic methods that declare new type variables in the signature in [Link to Come]. For instance, the method `asList` takes an array of type `E[]` and returns a result of type `List<E>`, and does so for *any* type `E`. Here we have a generic method that declares a *bound* on the type variable. The method `max` takes a collection of type `Collection<T>` and returns a `T`, and it does this for *any* type `T` such that `T` is a subtype of `Comparable<T>`.

The highlighted phrase in angle brackets at the beginning of the type signature declares the type variable `T`, and we say that `T` is *bounded* by `Comparable<T>`. As with wildcards, bounds for type variables are always indicated by the keyword `extends`, even when the bound is an interface rather than a class, as is the case here. Unlike wildcards, type variables can only be bounded using `extends`, not `super`.

In this case, the bound is *recursive*, in that the bound on `T` itself depends upon `T`. It is even possible to have mutually recursive bounds, such as:

```
<T extends C<T,U>, U extends D<T,U>>
```

An example of mutually recursive bounds appears in [Link to Come].

The method body first obtains an iterator over the collection, then calls the `next` method to select the first element as a candidate for the maximum; the specification of this method allows it to throw a `NoSuchElementException` if it is supplied with an empty collection. It then compares the candidate with each element in the collection, setting the candidate to the element when the element is larger.

> The code shown above, from the current (Java 20) version of the JDK, was written before Java 8 introduced streams and static methods on interfaces, which permit a more concise and probably more efficient version, with the

same signature but returning a value of type `Optional<T>` to allow for the case of an empty collection:

```
public static <T extends Comparable<T>> Optional<T>
max(Collection<T> coll) {
    return coll.stream().max(Comparator.naturalOrder());
}
```

We will explore the features of `Comparator` that make this possible in “Comparator<T>”.

When calling the method, `T` may be chosen to be `Integer` (since `Integer` implements `Comparable<Integer>`) or `String` (since `String` implements `Comparable<String>`):

```
List<Integer> ints = Arrays.asList(0,1,2);
assert Collections.max(ints) == 2;

List<String> strs = Arrays.asList("zero","one","two");
assert Collections.max(strs).equals("zero");
```

But we may not choose `T` to be `Number` (since `Number` does not implement `Comparable`):

```
List<Number> nums = Arrays.asList(0,1,2,3.14);
assert Collections.max(nums) == 3.14; // compile-time error
```

As expected, here the call to `max` is illegal.

Declarations for methods should be as general as possible to maximize utility. If you can replace a type parameter with a wildcard then you should do so. We can improve the declaration of `max` by replacing:

```
<T extends Comparable<T>> T max(Collection<T> coll)
```

with:

```
<T extends Comparable<? super T>> T max(Collection<? extends T> coll)
```

Following the Get and Put Principle, we use `extends` with `Collection` because we *get* values of type `T` from the collection, and we use `super` with `Comparable` because we *put* value of type `T` into the `compareTo` method. In the next section, we'll see an example that would not type-check if the `super` clause above was omitted.

If you look at the signature of this method in the Java library, you will see something that looks even worse than the preceding code:

```
<T extends Object & Comparable<? super T>> T max(Collection<? extends
T> coll)
```

The highlighted bound is there for backward compatibility, as we will explain at the end of "Multiple Bounds".

A Fruity Example

The `Comparable<T>` interface gives fine control over what can and cannot be compared. Say that we have a `Fruit` class with subclasses `Apple` and `Orange`. Depending on how we set things up, we may *prohibit* comparison of apples with oranges or we may *permit* such comparison.

Example 2-1 prohibits comparison of apples with oranges. Here are the three classes it declares:

```
class Fruit {...}
class Apple extends Fruit implements Comparable<Apple> {...}
class Orange extends Fruit implements Comparable<Orange> {...}
```

Each fruit has a name and a size, and two fruits are equal if they have the same name and the same size. Since we have overridden `equals`, we have also overridden `hashCode`, to ensure that equal objects have the same hash code (see Hash Tables for an explanation of the importance of this). Apples are compared by comparing their sizes, and so are oranges. Since `Apple` implements `Comparable<Apple>`, it is clear that you can compare apples with apples, but not with oranges. The test code builds three lists, one of apples, one of oranges, and one containing mixed fruits. We may find the maximum of

the first two lists, but attempting to find the maximum of the mixed list signals an error at compile time.

Example 2-2 permits comparison of apples with oranges. Compare these three class declarations with those given previously (all differences between Example 2-1 and Example 2-2 are highlighted):

```
class Fruit implements Comparable<Fruit> {...}
class Apple extends Fruit {...}
class Orange extends Fruit {...}
```

As before, each fruit has a name and a size, and two fruits are equal if they have the same name and the same size. Now, since `Fruit` implements `Comparable<Fruit>`, any two fruits may be compared by comparing their sizes. So the test code can find the maximum of all three lists, including the one that mixes apples with oranges.

Recall that at the end of the previous section we extended the type signature of `max` to use `super`:

```
<T extends Comparable<? super T>> T max(Collection<? extends T> coll)
```

Example 2-2 shows why this wildcard is needed. If we want to compare two oranges, we take T in the preceding code to be `Orange`:

```
Orange extends Comparable<? super Orange>
```

And this is true because both of the following hold:

```
Orange extends Comparable<Fruit>  and  Fruit super Orange
```

Without the super wildcard, finding the maximum of a `List<Orange>` would be illegal, even though finding the maximum of a `List<Fruit>` is permitted.

Also note that the natural ordering used here is not consistent with equals (see "Comparable<T>"). Two fruits with different names but the same size compare as the same, but they are not equal.

Comparator<T>

Sometimes we want to compare objects that do not implement the `Comparable` interface, or to compare objects using a different ordering from the one specified by that interface. The ordering provided by the `Comparable` interface is called the *natural ordering*, so the `Comparator` interface provides, so to speak, an unnatural ordering.

We specify additional orderings using the `Comparator` interface, which declares two abstract methods:

```
interface Comparator<T> {
  public int compare(T o1, T o2);
  public boolean equals(Object obj);
}
```

The `compare` method returns a value that is negative, zero, or positive depending upon whether the first object is less than, equal to, or greater than the second object—just as with `compareTo`. (The `equals` method is the one familiar from class `Object`; it is included in the interface to remind implementors that equal comparators must have `compare` methods that impose the same ordering.)

Example 2-1. Prohibiting comparison of apples with oranges

```
abstract class Fruit {
  protected String name;
  protected int size;
  protected Fruit(String name, int size) {
    this.name = name; this.size = size;
  }
  public boolean equals(Object o) {
    if (o instanceof Fruit) {
      Fruit that = (Fruit)o;
      return this.name.equals(that.name) && this.size == that.size;
    } else return false;
  }
  public int hashCode() {
    return Objects.hash(name,size);
  }
  protected int compareTo(Fruit that) {
    return Integer.compare(this.size, that.size);
  }
}
```

```
class Apple extends Fruit implements Comparable<Apple> {
  public Apple(int size) { super("Apple", size); }
  public int compareTo(Apple a) { return super.compareTo(a); }
}
class Orange extends Fruit implements Comparable<Orange> {
  public Orange(int size) { super("Orange", size); }
  public int compareTo(Orange o) { return super.compareTo(o); }
}
class Test {
  public static void main(String[] args) {

    Apple a1 = new Apple(1); Apple a2 = new Apple(2);
    Orange o3 = new Orange(3); Orange o4 = new Orange(4);

    List<Apple> apples = Arrays.asList(a1,a2);
    assert Collections.max(apples).equals(a2);

    List<Orange> oranges = Arrays.asList(o3,o4);
    assert Collections.max(oranges).equals(o4);

    List<Fruit> mixed = List.of(a1,o3);
    Collections.max(mixed);       // compile-time error
  }
}
```

Example 2-2. Permitting comparison of apples with oranges

```
abstract class Fruit implements Comparable<Fruit> {
  protected String name;
  protected int size;
  protected Fruit(String name, int size) {
    this.name = name; this.size = size;
  }
  public boolean equals(Object o) {
    if (o instanceof Fruit) {
      Fruit that = (Fruit)o;
      return this.name.equals(that.name) && this.size == that.size;
    } else return false;
  }
  public int hashCode() {
    return Objects.hash(name,size);
  }
  public int compareTo(Fruit that) {
    return Integer.compare(this.size, that.size);
  }
}
class Apple extends Fruit {
  public Apple(int size) { super("Apple", size); }
}
class Orange extends Fruit {
```

```
  public Orange(int size) { super("Orange", size); }
}
class Test {
  public static void main(String[] args) {

    Apple a1 = new Apple(1); Apple a2 = new Apple(2);
    Orange o3 = new Orange(3); Orange o4 = new Orange(4);

    List<Apple> apples = Arrays.asList(a1,a2);
    assert Collections.max(apples).equals(a2);

    List<Orange> oranges = Arrays.asList(o3,o4);
    assert Collections.max(oranges).equals(o4);

    List<Fruit> mixed = List.of(a1,o3);
    assert Collections.max(mixed).equals(o3); // ok
  }
}
```

Here is a comparator that considers the shorter of two strings to be smaller. Only if two strings have the same length are they compared using the natural (alphabetic) ordering.

```
  Comparator<String> sizeOrder =
    new Comparator<>() {
      public int compare(String s1, String s2) {
        return
          s1.length() < s2.length() ? -1 :
          s1.length() > s2.length() ? 1 :
          s1.compareTo(s2) ;
        }
  };
```

And here is an example of its use:

```
  assert "two".compareTo("three") > 0;
  assert sizeOrder.compare("two","three") < 0;
```

In the natural alphabetic ordering, `"two"` is greater than `"three"`, whereas in the size ordering it is smaller.

The Java libraries always provide a choice between `Comparable` and `Comparator`. For every generic method with a type variable bounded by `Comparable`, there is another generic method with an additional argument of

type `Comparator`. For instance, corresponding to:

```
public static <T extends Comparable<? super T>>
  T max(Collection<? extends T> coll)
```

we also have:

```
public static <T>
  T max(Collection<? extends T> coll, Comparator<? super T> comp)
```

There are similar methods to find the minimum. For example, here is how to find the maximum and minimum of a list using the natural ordering and using the size ordering:

```
Collection<String> strings = Arrays.asList("from","aaa","to","zzz");
assert Collections.max(strings).equals("zzz");
assert Collections.min(strings).equals("aaa");
assert Collections.max(strings,sizeOrder).equals("from");
assert Collections.min(strings,sizeOrder).equals("to");
```

The string `"from"` is the maximum using the size ordering because it is longest, and `"to"` is minimum because it is shortest.

Here, slightly simplified, is the code in the class `Collection` for the version of `max` that takes a `Comparator`.

```
public static <T extends Comparable<T>> T max(Collection<T> coll,
Comparator<? super T> comp) {
    Iterator<? extends T> i = coll.iterator();
    T candidate = i.next();
    while (i.hasNext()) {
        T next = i.next();
        if (comp.compare(next, candidate) > 0)
            candidate = next;
    }
    return candidate;
}
```

Compared to the previous version, the only change is that where before we wrote `next.compareTo(candidate)`, now we write `comp.compare(next, candidate)`.

It is easy to define a comparator that provides the natural ordering. This is a slightly simplified version of the code for the static method `naturalOrder` on the `Comparator` interface:

```
public static <T extends Comparable<? super T>> Comparator<T>
naturalOrder() {
  return new Comparator<>(){
    public int compare(T o1, T o2) { return o1.compareTo(o2); }
  };
}
```

Using this, we can define the version of `max` that uses the natural ordering in terms of the version that uses a given comparator. This is the overload of `max` that was used in the stream version of the `Collections` method in “Comparable<T>”:

```
public static <T extends Comparable<? super T>> T max(Collection<?
extends T> coll) {
  return Collections.max(coll, Comparator.naturalOrder());
}
```

In everyday practice, you will probably create `Comparator` instances using one of the static or default methods that the interface exposes, or a combination of them. Let’s explore how they work on a simple record type:

```
record Person(String name, int age) {}
```

Probably the most commonly used methods are `comparing` and its variants, which accept a function that extracts a key to use in comparing two objects. For example, if we want to sort a list of `Person` objects by name, we can use this `Comparator`:

```
Comparator<Person> compareByName = Comparator.comparing(Person::name);
```

The method `comparing` takes a function (represented by the functional interface `java.util.function.Function`[1] —typically a getter—from the type to be sorted to a sort key. It then returns a comparator that, when it is applied to two objects of the type to be sorted, will return a value corresponding

to the natural ordering on their respective sort keys. For example, if we define a list of `Person` objects:

```
Person a32 = new Person("Alice", 32);
Person b23 = new Person("Bob", 23);
List<Person> l = new ArrayList<>(List.of(a32, b23));
```

we can write

```
l.sort(compareByName);
assert l.equals(List.of(a32,b23));
```

Even somewhat simplified, the declaration of `Comparator.comparing` in the JDK looks very difficult:

```
public static <T, U extends Comparable<? super U>> Comparator<T>
    comparing(Function<? super T, ? extends U> keyExtractor) {
        return (c1, c2) ->
            keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

In fact, we have met most of the difficulties already. `T` is the type of the class to be compared, and `U` is the type of the sort key. As we saw at the end of "Maximum of a Collection", the most generally useful bound on the type of a comparable is `U extends Comparable<? super U>`: in other words, if any supertype of `U` defines `compareTo`, `U` will inherit that definition. The type signature of the `Function` interface can be explained by the Substitution Principle: a function that can be applied to some supertype of `T` can certainly be applied to a `T` value; conversely, if it returns a subtype of `U`, that will be certainly be usable where a `U` is required.

The intimidating appearance of this declaration contrasts with the ease of using the method that it declares. This contrast is typical of Java generics: the API designer needs to think very carefully about how to make it as usable and general as possible. If the design is successful, client code developers will be highly appreciative—if they stop to think about it, that is, since the best-designed APIs are distinguished by their unobtrusiveness.

The `Function` parameter of the method `comparing` can only be applied to

an object of a reference type. So for primitive types, we need specialized versions of `comparing`. For example:

```
Comparator<Person> compareByAge =
Comparator.comparingInt(Person::age);
l.sort(compareByAge);
assert l.equals(List.of(b23,a32));
```

Earlier, in "Comparable<T>", we saw two solutions to the problem of comparing two `Event` objects, one clumsy and one defective. We can use `comparingInt` to define a version of `compareTo` that is more readable and less error-prone than the first version, but without the overflow problem of the second:

```
record Event(String name, int millisecs) implements Comparable<Event>
{
    public int compareTo(Event other) {
        return
Comparator.comparingInt(Event::millisecs).compare(this,other);
    }
}
```

The ordering of a comparator can be conveniently reversed using the instance method `reversed`:

```
l.sort(compareByAge.reversed());
assert l.equals(List.of(a32,b23));
```

Sorting often requires multiple levels: when the result of sorting on one key produces groups of results, with the results in each group having the same value of the sort key. Each group may then need to be sorted by a secondary key. `Comparator` supports this with the instance method `comparingAndThen`, which creates a comparator that compares its arguments first using the receiver (this comparator) then, if they are equal, using the comparator supplied as the argument. To show this in action, we add a new `Person` object with the same name as one pre-existing object and the same age as the other.

```
Person a23 = new Person("Alice", 23);
l.add(a23);
l.sort(compareByName.thenComparing(compareByAge));
```

```
assert l.equals(List.of(a23,a32,b23));
```

The `Comparator` API is designed so that these methods are composable. For example, supposing that we want to sort a list of `Person` objects in reverse order of age and then, in the case of tied results, by reverse order of name. That is easy to express:

```
l.sort(compareByAge.reversed().thenComparing(compareByName.reversed())
);
assert l.equals(List.of(a32,b23,a23));
```

Unlike `Comparable::compareTo`, the `compare` method of `Comparator` can optionally permit comparison of `null` arguments. The static methods `nullsFirst` and `nullsLast` take a `Comparator` and return a null-friendly comparator that treats non-null values the same as the comparator supplied as the argument, but considers null to be less—or, respectively, greater—than any non-null value:

```
l.add(null);
l.sort(Comparator.nullsFirst(compareByAge));
assert l.equals(Arrays.asList(null,b23,a23,a32));
```

As a final example of comparators, here is a method that takes a comparator on elements and returns a comparator on lists of elements:

```
public static <E>
  Comparator<List<E>> listComparator(final Comparator<? super E> comp)
{
  return new Comparator<>() {
    public int compare(List<E> list1, List<E> list2) {
      int n1 = list1.size();
      int n2 = list2.size();
      for (int i = 0; i < Math.min(n1,n2); i++) {
        int k = comp.compare(list1.get(i), list2.get(i));
        if (k != 0) return k;
      }
      return Integer.compare(n1, n2);
    }
  };
}
```

The loop compares corresponding elements of the two lists, and terminates when corresponding elements are found that are not equal (in which case, the list with the smaller element is considered smaller) or when the end of either list is reached (in which case, the shorter list is considered smaller). This is the usual ordering for lists; if we convert a string to a list of characters, it gives the usual ordering on strings.

Enumerated Types

Java 5 includes support for enumerated types: types whose values consist of a fixed set of constants. Here are two simple examples:

```
enum Season { WINTER, SPRING, SUMMER, FALL }
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
SUNDAY }
```

Each enumerated type declaration can be expanded into a corresponding class in a stylized way. The corresponding class is designed so that it has exactly one instance for each of the enumerated constants, bound to a suitable static final variable. For example, the first of the `enum` declarations above expands into a class called `Season`. Exactly four instances of this class exist, bound to four static final variables with the names `WINTER`, `SPRING`, `SUMMER`, and `FALL`. There is no way to create any further instances of `Season`.

Foremost amongst the many useful features of Java enums is their type safety: for example, you can't assign a `Weekday` value to a variable of type `Season`, or compare two values of these different types. This type safety is enforced by the declaration of the enumerated types, as in Example 2-4. Each class corresponding to an enumerated type is a subclass of `java.lang.Enum`, declared as in Example 2-3[2].

The class `Enum` provides properties that every enumerated type needs: its name and its ordinal—that is, its position in the enumeration sequence. Also, by implementing `Comparable`, it ensures that its derived classes will also be `Comparable`. But the requirement for type safety means that it is not enough that, for example, `Season implements Comparable`; on its own, that would still permit comparison of a `Season` with a `Weekday`. What is

necessary is that `Season` should implement `Comparable<Season>`.

This takes us some distance towards understanding the declaration of `Enum`:

```
class Enum<E extends Enum<E>> implements Comparable<E>
```

You may find this frightening at first sight—your authors certainly did! But don't panic. Matching things up shows how a class derived from `Enum` will have the property that we need. From what we already know, we have that

```
class Season extends ... implements Comparable<Season>
```

where `...` is some parameterization of `Enum`. If that parameterization is `Enum<Season>`, we get

```
class Season extends Enum<Season>                 // 1
class Enum<Season> implements Comparable<Season>  // 2
```

and substituting //1 into //2 gives

```
class Enum<Season extends Enum<Season>> implements Comparable<Season>
```

Of course, `Enum` needs to work on any enumerated type, not only `Season`, so its declaration in the class library is

```
class Enum<E extends Enum<E>> implements Comparable<E>
```

Example 2-3. Base class for enumerated types

```
public abstract class Enum<E extends Enum<E>> implements Comparable<E> {
  private final String name;
  private final int ordinal;
  protected Enum(String name, int ordinal) {
    this.name = name; this.ordinal = ordinal;
  }
  public final String name() { return name; }
  public final int ordinal() { return ordinal; }
  public String toString() { return name; }
  public final int compareTo(E o) {
    return ordinal - ((Enum<?>)o).ordinal;
  }
}
```

Example 2-4. Class corresponding to an enumerated type

```
/*
   corresponds to enum Season { WINTER, SPRING, SUMMER, FALL }
*/
final class Season extends Enum<Season> {
  private Season(String name, int ordinal) { super(name,ordinal); }
  public static final Season WINTER = new Season("WINTER",0);
  public static final Season SPRING = new Season("SPRING",1);
  public static final Season SUMMER = new Season("SUMMER",2);
  public static final Season FALL   = new Season("FALL",3);
  private static final Season[] VALUES = { WINTER, SPRING, SUMMER, FALL
};
  public static Season[] values() { return VALUES.clone(); }
  public static Season valueOf(String name) {
    for (Season e : VALUES) if (e.name().equals(name)) return e;
    throw new IllegalArgumentException();
  }
}
```

Recursively bounded types like `T extends Comparable<T>` and `E extends Enum<E>` occur when we want to constrain a parameter to range only over the subtypes of a particular type. For example, in any subtype `T` of the base `Enum` class, the `compareTo` method can only be applied to an argument of type `T`, not to any other subtype of `Enum`: the definition provides a way of naming the type itself in the body of the class. (The term “self-type” is sometimes used for this in discussions of recursively bounded types.)

The rest of the definitions are straightforward. The base class `Enum` defines two fields, a string `name` and an integer `ordinal`, that are possessed by every instance of an enumerated type; the fields are final because once they are initialized, their value never changes. The constructor for the class is protected, to ensure that it is used only within subclasses of this class. Each enumeration class makes the constructor private, to ensure that it is used only to create the enumerated constants. For instance, the `Season` class has a private constructor that is invoked exactly four times in order to initialize the final variables `WINTER`, `SPRING`, `SUMMER`, and `FALL`.

The base class defines accessor methods for the `name` and `ordinal` fields. The `toString` method returns the name, and the `compareTo` method just returns the difference of the ordinals for the two enumerated values. (Unlike the definition of `Event` in “Comparable<T>”, this is safe because, since the

ordinals are always positive, there is no possibility of overflow.) Hence, constants have the same ordering as their ordinals—for example, `WINTER` precedes `SUMMER`.

Lastly, there are two static methods in every class that corresponds to an enumerated type. The `values` method returns an array of all the constants of the type. It returns a (shallow) clone of the internal array. Cloning is vital to ensure that the client cannot alter the internal array. Note that you don't need a cast when calling the `clone` method, because cloning for arrays now takes advantage of covariant return types (see "Covariant Overriding"). The `valueOf` method takes a string and returns the corresponding constant, found by searching the internal array. It returns an `IllegalArgumentException` if the string does not name a value of the enumeration.

Multiple Bounds

We have seen many examples where a type variable or wildcard is bounded by a single class or interface. In rare situations, it may be desirable to have multiple bounds, and we show how to do so here.

To demonstrate, we use three interfaces from the Java library. The `Readable` interface has a `read` method to read into a buffer from a character source, the `Appendable` interface has an `append` method to copy from a buffer into a target capable of receiving characters, and the `Closeable` interface has a `close` method to close a source or target. Possible sources and targets include character files, buffers, streams, and so on.

For maximum flexibility, we might want to write a `copy` method that accepts a `Supplier` of any source that implements both `Readable` and `Closeable`, and a `Supplier` of any target that implements both `Appendable` and `Closeable`. The method will copy the contents of the source to the target. Unfortunately, character readers and writers typically cannot be opened without potentially throwing `IOException`, so we need a specialized supplier interface:

```
interface IoeThrowingSupplier<S> {
    S get() throws IOException;
```

```
}
```

The method `copy` can now be declared:

```
public static <S extends Readable & Closeable,
               T extends Appendable & Closeable>
    void copy(IoeThrowingSupplier<S> src, IoeThrowingSupplier<T> tgt,
int size)
                                                        throws
IOException {
    try (S s = src.get(); T t = tgt.get()) {
        CharBuffer buf = CharBuffer.allocate(size);
        int i = s.read(buf);
        while (i >= 0) {
            buf.flip(); // prepare buffer for writing
            t.append(buf);
            buf.clear(); // prepare buffer for reading
            i = s.read(buf);
        }
    }
}
```

The first line of this method specifies that `S` ranges over any type that implements both `Readable` and `Closeable`, and that `T` ranges over any type that implements `Appendable` and `Closeable`. When multiple bounds on a type variable appear, they are separated by ampersands.

The try-with-resources statement evaluates the supplied lambdas, which open and return a character source and a character target. It then repeatedly reads from the source into a buffer and appends from the buffer into a target. When the source is empty, the try block is exited, closing both the source and the target. As an example, this method may be called with a `FileReader` and a `FileWriter` as source and target:

```
int size = 32;
Files.writeString(Path.of("file.in"), "hello world");
copy(() -> new FileReader("file.in"),
     () -> new FileWriter("file.out"), size);
assert Files.readString(Path.of("File.out")).equals("hello world");
```

Other possible sources include `FilterReader`, `PipedReader`, and `StringReader`, and other possible targets include `FilterWriter`,

`PipedWriter`, and `PrintStream`. But you could not use `StringBuffer` as a target, since it implements `Appendable` but not `Closeable`.

If you are familiar with the `java.io` library, you may have spotted that all classes that implement both `Readable` and `Closeable` are subclasses of `Reader`, and almost all classes that implement `Appendable` and `Closeable` are subclasses of `Writer`. So you might wonder why we don't simplify the method signature like this:

```
public static void copy(Reader src, Writer trg, int size)
```

This will indeed admit most of the same classes, but not all of them. For instance, `PrintStream` implements `Appendable` and `Closeable` but is not a subclass of `Writer`. Furthermore, you can't rule out the possibility that some programmer using your code might have his or her own custom class that, say, implements `Readable` and `Closeable` but is not a subclass of `Reader`.

When multiple bounds appear, the first bound is used for erasure. We saw this used earlier in "Maximum of a Collection":

```
public static <T extends Object & Comparable<? super T>>
  T max(Collection<? extends T> coll)
```

Without the highlighted text, the erased type signature for `max` would have `Comparable` as the return type, whereas in legacy libraries the return type is `Object`. Maintaining compatibility with legacy libraries is further discussed in [Link to Come] and [Link to Come].

Bridges

As we mentioned earlier, generics are implemented by erasure: the compiled representation of code written with generics is almost exactly the same as that written without. However, in the case of a parameterized supertype—for example, an interface such as `Comparable<T>`—erasure may require additional methods to be inserted by the compiler; these additional methods are called *bridges*.

Example 2-5 shows the `Comparable` interface and a simple `Point` record implementing that interface. The natural order on `Point`s is defined by comparing their distances from the origin (or, more precisely, the squares of their distances from the origin: it amounts to the same thing). The declaration of `compareTo` in `Point` overrides that in `Comparable`, because the signatures match.

Example 2-5. Generic code implementing a parameterized interface

```
interface Comparable<T> {
    public int compareTo(T other);
}
record Point(double x, double y) implements Comparable<Point> {
    public int compareTo(Point p) {
        return Double.compare(this.x * this.x + this.y * this.y,
                              p.x * p.x + p.y * p.y);
    }
}
```

Erasure changes the situation, however, as shown in Example 2-6.

Example 2-6. Parameterized interface and implementation after erasure

```
interface Comparable {
    public int compareTo(Object other);
}
record Point(double x, double y) implements Comparable {  // fails
compilation
    public int compareTo(Point p) {
        return Double.compare(this.x * this.x + this.y * this.y,
                              p.x * p.x + p.y * p.y);
    }
}
```

The signature of `compareTo` in the interface has changed, and to restore overriding the compiler must add an extra method to the implementation:

```
  public int compareTo(Object other) {
       return compareTo((Point)other);
  }
```

You can confirm the existence of this bridge using reflection:

```
  Map<Class<?>, Boolean> typeToBridge =
  (Arrays.stream(Point.class.getMethods())
      .filter(m -> m.getName().equals("compareTo"))
```

```
    .collect(Collectors.toMap(m -> m.getParameterTypes()[0],
Method::isBridge)));
assert typeToBridge.size() == 2;
assert ! typeToBridge.get(Point.class);    // compareTo(Point) is not
a bridge method
assert typeToBridge.get(Object.class);     // compareTo(Object) is a
bridge method
```

And the entire declaration of the method can be seen by decompiling the class file, for example with the open source decompiler FernFlower, which produces this output:

```
....
// $FF: synthetic method
// $FF: bridge method
public int compareTo(Object var1) {
    return this.compareTo((Point)var1);
}
....
```

Bridge methods are required whenever a class or interface implements or extends a parameterised supertype by instantiating its type parameter[3].

Bridges can play an important role when converting legacy code to use generics; see [Link to Come].

Covariant Overriding

At the same time that generics were introduced, Java started supporting covariant method overriding. This feature is not directly related to generics, but we discuss it here because it is worth knowing, and because it is implemented using a bridging technique like that described in the previous section.

In early versions of Java, one method could override another only if the signatures and the return types matched exactly. With covariant overriding, the signatures must still match (after erasure), but the requirement to match the return types is relaxed so that now the return type of the overriding method need only be a subtype of the return type of the overridden method.

The `clone` method of class `Object` illustrates the advantages of covariant overriding:

```
class Object {
    ...
    protected Object clone() { ... }
}
```

Without covariant overriding, any class that overrides `clone` has to give the overriding method exactly the same return type, namely `Object`. For example, here is a `Point` record that does this:

```
record Point(double x, double y)  {
    public Object clone() { return new Point(x,y); }
}
```

Here, even though `clone` always returns a `Point`, without covariance the old rules required it to have the return type `Object`. This was annoying, since every invocation of `clone` had to cast its result.

```
Point p = new Point(1,2);
Point q = (Point)p.clone();
```

With covariant overriding, it is possible to give the `clone` method a return type that is more to the point, as it were:

```
record Point(double x, double y) {
    public Point clone() { return new Point(x,y); }
}
```

Now we may clone without a cast:

```
Point p = new Point(1,2);
Point q = p.clone();
```

Covariant overriding is implemented using the bridging technique described in the previous section. As before, you can see the bridge using decompilation or by applying reflection. Here is code that finds all methods with the name `clone` in the class `Point`, then maps the return type of each overload to the result of calling the `Method` method `isBridge`:

```
Map<Class<?>,Boolean> returnToBridge =
(Arrays.stream(Point.class.getMethods())
```

```
        .filter(m -> m.getName().equals("clone"))
        .collect(Collectors.toMap(Method::getReturnType,
Method::isBridge)));
assert returnToBridge.size() == 2;
assert returnToBridge.get(Object.class);     // Object clone(Point) is
a bridge method
assert ! returnToBridge.get(Point.class);    // Point clone(Point) is
not a bridge method
```

Here the bridging technique exploits the fact that in a class file two methods of the same class may have the same argument signature, even though this is not permitted in Java source. The bridge method simply calls the first method.

---

1 See *https://www.lambdafaq.org/what-is-a-functional-interface/*

2 The code for Example 2-3 does not correspond on all points with the source in the Java library, and most of the code corresponding to Example 2-4 is actually synthesized by the Java compiler, but these examples are functionally equivalent to the code that is actually executed.

3 Strictly speaking, this is true only if erasure changes the signature of any of the supertype‘s methods.

Chapter 3. Declarations

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

This chapter discusses how to declare a generic class. It describes constructors, static members, and nested classes, and it fills in some details of how erasure works.

Constructors

In a generic class, type parameters appear in the header that declares the class, but not in the constructor:

```
class Pair<T,U> {
  private final T first;
  private final U second;
  public Pair(T first, U second) {
        this.first = first;
        this.second = second;
  }
  public T getFirst() { return first; }
  public U getSecond() { return second; }
}
```

The type parameters `T` and `U` are declared at the beginning of the class, not in the constructor. However, actual type parameters are passed to the constructor

whenever it is invoked:

```
Pair<String, Integer> pair1 = new Pair<String, Integer>("one",2);
assert pair1.getFirst().equals("one") && pair1.getSecond() == 2;
```

Look Out for This! A common mistake is to forget the type parameters when invoking the constructor:

```
Pair<String, Integer> pair2 = new Pair("one",2);
```

This mistake produces a warning, but not an error. It is taken to be legal, because `Pair` is treated as a *raw type*, a type containing no parametric type information but which can be converted to the corresponding parameterized type, generating only an unchecked warning (see [Link to Come], which explains raw types and how the `-Xlint:unchecked` flag can help you spot errors of this kind).

Records can be parameterised in just the same way as regular classes:

```
record Pair<T,U>(T first, U second) {}
```

Static Members

Compilation by erasure means that at run time parameterised types are replaced by the corresponding raw type: for example, the interfaces `List<Integer>`, `List<String>`, and `List<List<String>>` are all implemented by a single interface, namely `List`:

```
List<Integer> ints = Arrays.asList(1,2,3);
List<String> strings = Arrays.asList("one","two");
assert ints.getClass() == strings.getClass();
```

We see here that the class object associated with a list of integers at run time is the same as the one associated with a list of strings.

One consequence is that static members of a generic class are shared across all instantiations of that class, including instantiations at different types. Static members of a class cannot refer to the type parameter of a generic class, and when accessing a static member the class name should not be parameterized.

For example, here is a class, `Cell<T>`, in which each cell has an integer identifier and a value of type `T`:

```
class Cell<T> {
    private final int id;
    private final T value;
    private final static AtomicInteger count = new AtomicInteger();
    private static int nextId() { return count.getAndIncrement(); }
    public Cell(T value) {
        this.value = value;
        id = nextId();
    }
    public T getValue() { return value; }
    public int getId() { return id; }
    public static int getCount() { return count.get(); }
}
```

A static field, `count`, is used to allocate a distinct identifier to each cell. For the count, an `AtomicInteger` is used to ensure that unique identifiers are generated even under concurrent access. The static `getCount` method returns the current count.

Here is code that allocates a cell containing a string and a cell containing an integer, which are allocated the identifiers `0` and `1`, respectively:

```
Cell<String> a = new Cell<String>("one");
Cell<Integer> b = new Cell<Integer>(2);
assert a.getId() == 0 && b.getId() == 1 && Cell.getCount() == 2;
```

Static members are shared across all instantiations of a class, so the same count is incremented when allocating either a string or an integer cell.

Because static members are independent of any type parameters, we are not permitted to follow the class name with type parameters when accessing a static member:

```
Cell.getCount();          // ok
Cell<Integer>.getCount(); // compile-time error
Cell<?>.getCount();       // compile-time error
```

The count is static, so it is a property of the class as a whole, not any particular instance.

For the same reason, you can't refer to a type parameter anywhere in the declaration of a static member. Here is a second version of `Cell`, which attempts to use a static variable to keep a list of all values stored in any cell:

```
class Cell2<T> {
  private final T value;
  private static List<T> values = new ArrayList<T>(); // illegal
  public Cell2(T value) {
        this.value=value;
        values.add(value);
  }
  public T getValue() { return value; }
  public static List<T> getValues() { return values; } // illegal
}
```

Since the class may be used with different type parameters at different places, it makes no sense to refer to `T` in the declaration of the static field `values` or the static method `getValues`, and these lines are reported as errors at compile time. If we want a list of all values kept in cells, then we need to use a list of objects, as in the following variant:

```
class Cell2<T> {
  private final T value;
  private static List<Object> values = new ArrayList<Object>(); // ok
  public Cell2(T value) {
        this.value=value;
        values.add(value);
  }
  public T getValue() { return value; }
  public static List<Object> getValues() { return values; } // ok
}
```

This code compiles and runs with no difficulty:

```
Cell2<String> a = new Cell2<String>("one");
Cell2<Integer> b = new Cell2<Integer>(2);
assert Cell2.getValues().equals(List.of("one",2));
```

Nested Classes

Java permits nesting one class inside another. If the outer class has type parameters and the inner class is a member class—that is, not static—then type

parameters of the outer class are visible within the inner class.

Example 3-1 shows a class implementing collections as a singly-linked list. This class extends `java.util.AbstractCollection`, so it only needs to define the methods `size`, `add`, and `iterator` (see [Link to Come]). It contains an inner class, `Node`, for the list nodes, and an anonymous inner class implementing `Iterator<E>`. The type parameter `E` is in scope within both of these classes.

Example 3-1. Type parameters are in scope for member classes

```
class LinkedCollection<E> extends AbstractCollection<E> {
  private class Node {
    private E element;
    private Node next = null;
    private Node(E elt) { element = elt; }
  }
  private Node first = new Node(null);
  private Node last = first;
  private int size = 0;
  public LinkedCollection() {}
  public LinkedCollection(Collection<? extends E> c) { addAll(c); }
  public int size() { return size; }
  public boolean add(E elt) {
    last.next = new Node(elt); last = last.next; size++;
    return true;
  }
  public Iterator<E> iterator() {
    return new Iterator<E>() {
      private Node current = first;
      public boolean hasNext() {
        return current.next != null;
      }
      public E next() {
        if (current.next != null) {
          current = current.next;
          return current.element;
        } else throw new NoSuchElementException();
      }
      public void remove() {
        throw new UnsupportedOperationException();
      }
    };
  }
}
```

For contrast, Example 3-2 shows a similar implementation, but this time the

inner `Node` class is static, and so the type parameter `E` is *not* in scope for this class. Instead, the inner class is declared with its own type parameter, `T`. Where the previous version referred to `Node`, the new version refers to `Node<E>`. The anonymous iterator class in the preceding example has also been replaced by a static inner class, again with its own type parameter.

If the node classes had been made public rather than private, you would refer to the node class in the first example as `LinkedCollection<E>.Node`, whereas you would refer to the node class in the second example as `LinkedCollection.Node<E>`.

Example 3-2. Type parameters are not in scope for static inner classes

```
class LinkedCollection<E> extends AbstractCollection<E> {
  private static class Node<T> {
    private T element;
    private Node<T> next = null;
    private Node(T elt) { element = elt; }
  }
  private Node<E> first = new Node<E>(null);
  private Node<E> last = first;
  private int size = 0;
  public LinkedCollection() {}
  public LinkedCollection(Collection<? extends E> c) { addAll(c); }
  public int size() { return size; }
  public boolean add(E elt) {
    last.next = new Node<E>(elt); last = last.next; size++;
    return true;
  }
  private static class LinkedIterator<T> implements Iterator<T> {
    private Node<T> current;
    public LinkedIterator(Node<T> first) { current = first; }
    public boolean hasNext() {
      return current.next != null;
    }
    public T next() {
      if (current.next != null) {
        current = current.next;
        return current.element;
      } else throw new NoSuchElementException();
    }
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
  public Iterator<E> iterator() {
    return new LinkedIterator<E>(first);
```

```
  }
}
```

Of the two alternatives described here, the second is preferable. Member classes are implemented by including a reference to the enclosing instance, since they may, in general, access components of that instance. Static inner classes are usually both simpler and more efficient.

How Erasure Works

The erasure of a type is defined as follows: drop all type parameters from parameterized types, and replace any type variable with the erasure of its bound, or with `Object` if it has no bound, or with the erasure of the leftmost bound if it has multiple bounds. Here are some examples:

- The erasure of `List<Integer>`, `List<String>`, and `List<List<String>>` is `List`.
- The erasure of `List<Integer>[]` is `List[]`.
- The erasure of `List` is itself, similarly for any raw type .
- The erasure of `int` is itself, similarly for any primitive type.
- The erasure of `Integer` is itself, similarly for any type without type parameters.
- The erasure of `T` in the definition of `toList` (see [Link to Come]) is `Object`, because `T` has no bound.
- The erasure of `T` in the definition of `max` (see "Maximum of a Collection") is `Comparable`, because `T` has bound `Comparable<? super T>`.
- The erasure of `T` in the final definition of `max` (see "Multiple Bounds") is `Object`, because `T` has bound `Object & Comparable<T>` and we take the erasure of the leftmost bound.
- The erasures of `S` and `T` in the definition of `copy` (see "Multiple Bounds") are `Readable and Appendable`, because `S` has bound `Readable & Closeable` and `T` has bound Appendable & Closeable.

- The erasure of `LinkedCollection<E>.Node` or `LinkedCollection.Node<E>` (see "Nested Classes") is `LinkedCollection.Node`.

In Java, two methods of the same class cannot have the same signature—that is, the same name and parameter types. Since generics are implemented by erasure, it also follows that two distinct methods cannot have signatures with the same erasure. A class cannot overload two methods whose signatures have the same erasure, and a class cannot implement two interfaces that have the same erasure.

For example, here is a class with two convenience methods. One adds together every integer in a list of integers, and the other concatenates together every string in a list of strings:

```
class Overloaded {
  public static int sum(List<Integer> ints) {
    int sum = 0;
    for (int i : ints) sum += i;
    return sum;
  }
  public static String sum(List<String> strings) {
    StringBuffer sum = new StringBuffer();
    for (String s : strings) sum.append(s);
    return sum.toString();
  }
}
```

Here are the erasures of the declarations of the two methods:

```
int sum(List)
String sum(List)
```

But it is the signatures alone, not the return types, that allow the Java compiler to distinguish different method overloads. In this case the erasures of the signatures of both methods are identical:

```
sum(List)
```

So a name clash is reported at compile time.

For another example, here is a bad version of the `Integer` class, that tries to

make it possible to compare an integer with either an integer or a long:

```
class Integer implements Comparable<Integer>, Comparable<Long> {
    // compile-time error, cannot implement two interfaces with same
erasure
    private final int value;
    ...
    public int compareTo(Integer i) {
        return (value < i.value) ? -1 : (value == i.value) ? 0 : 1;
    }
    public int compareTo(Long l) {
        return (value < l.intValue()) ? -1 : (value == l.intValue()) ?
0 : 1;
    }
    ...
}
```

If this were supported, it would, in general, require a complex and confusing definition of bridge methods (see “Bridges”). The simplest and most understandable option by far is to ban this case.

Chapter 4. The Main Interfaces of the Java Collections Framework

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

A *collection* is an object that provides access to a group of objects, allowing them to be processed in a uniform way. A *collections framework* provides a uniform view of a set of collection types specifying and implementing common data structures, following consistent design rules so that they can work together. Figure 4-1 shows the main interfaces of the Java Collections Framework, together with one other—`Iterable`—which is outside the Framework. `Iterable` (see "Iterable and Iterators") defines the contract that a class—any class, not only one of those in the Framework—must implement in order to be used as the target for an "enhanced for statement", usually called a *foreach* statement.

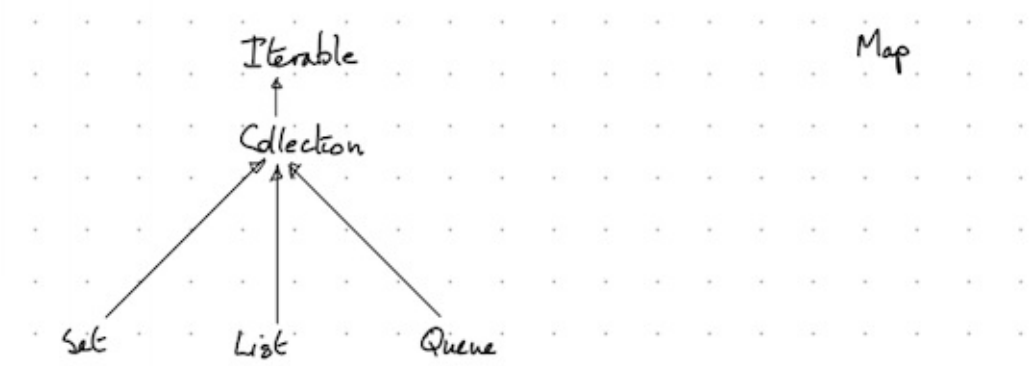

Figure 4-1. The main interfaces of the Java Collections Framework

The Framework interfaces in Figure 4-1 have the following purposes:

`Collection`

`Collection` exposes the core functionality required of any collection other than a `Map`. Its methods support managing elements by adding or removing single or multiple elements, checking membership of a single or multiple values, and inspecting and exporting elements. It has no direct concrete implementations; the concrete collection classes all implement one of its subinterfaces as well.

`Set`

A `Set` is a collection in which order is not significant and contains no duplicates. It adds no operations to `Collection`, but the contracts for its element-adding operations specify that they will not create duplicates.

`List`

A `List` is a collection in which order is significant and which accommodates duplicate elements. It adds operations supporting indexed access.

`Queue`

A `Queue` is a collection whose additional operations accept elements at one end, its *tail*, for processing, and yield them up at the other end, its *head*, in the order in which they are to be processed.

`Map`

A `Map` is a collection which uses key-value associations to store and retrieve elements. The keys of a `Map` form a `Set`, so they follow the rule that there is no duplication and ordering is not significant. The operations of `Map` allow it to be maintained by addition and removal of single or multiple key-value associations, but these are ancillary to the main use of `Map`s, which is to find the values that correspond to given keys.

Using the Different Collection Types

A simple example, a word cloud generator, will illustrate the purpose of some of these types. Word cloud generators normally ignore the order of the words in their input, so any of the word clouds in Figure 4-2 could be produced from the `Set` created by

```
Set.of("Larry", "Curly", "Moe")
```

Figure 4-2. Word clouds generated from `Set` data

Although the position of words in a cloud is randomly determined, their size depends on the number of repetitions of each in the input to the generator. So, since `Set`s cannot store duplicate elements, they are not actually a suitable data structure for representing the contents of a word cloud. For a collection type that does accommodates duplicates, we must turn to `List`. The word cloud generated by

```
List.of("Larry", "Larry", Curly", "Moe")
```

will be like one of those in Figure 4-3, reflecting the fact that the string `"Larry"` occurs twice as often as the other two. (Again, the random order of the words in Figure 4-3 is not because the order has been lost from the collection but because word cloud generators ignore the order of their input.)

Figure 4-3. Word clouds generated from `List` data

Since any `Set` can be represented by a `List`--one without duplicates and whose order is ignored—why have a separate data type at all? The reason is that the operations of `Set` make it much easier to guarantee element uniqueness, when that is what we actually want, and being able to disregard order can allow operations to be much more efficient.

In our word cloud example, a better representation of the input list would be a `Map` associating each word with its frequency:

```
Map.of("Larry", 2, "Curly", 1, "Moe", 1)
```

You can think of a `Map` as a table, one of those shown in Figure 4-4:

Key	Value
“Larry”	2
“Curly”	1
“Moe”	1

Key	Value
“Curly”	1
“Larry”	2
“Moe”	1

Key	Value
“Moe”	1
“Larry”	2
“Curly”	1

Figure 4-4. Word cloud data represented as a `Map`

The three tables in Figure 4-4 represent the same `Map`, because the table rows—that is, the key-value pairs that make up the `Map`--form a set, so that their order is not significant. Further, the keys also form a set, so there couldn’t be two rows

with the same key, “Larry” say.

Sequenced Collections

In the word cloud example, the order of the elements wasn’t important. Often, though, the ordering of elements is significant, and many collections, for example `List`, do preserve it—and, sometimes, impose it. Such collections are said to be *sequenced*. A sequenced collection is one with a defined order—unlike `Collection`, `Set` or `Map`--that can also be iterated in either direction, unlike `Queue`. Sequenced collections differ in how their ordering is derived: for some, like `List`, elements retain the order in which they were added, whereas for others, like `NavigableSet` (see below), the ordering is dictated by the values of the elements. These are sometimes called *externally ordered* and *internally ordered* types, respectively, reflecting the difference between an order which is arbitrarily imposed on the elements, for example by the order in which they are added, and an order that is an inherent property of the elements themselves, for example by alphabetic ordering on strings.

A new interface, `SequencedCollection`, was introduced in Java ?? to unify the sequenced collections, of both kinds. Figure 4-5 adds the sequenced interfaces of the Framework to those of Figure 4-1.

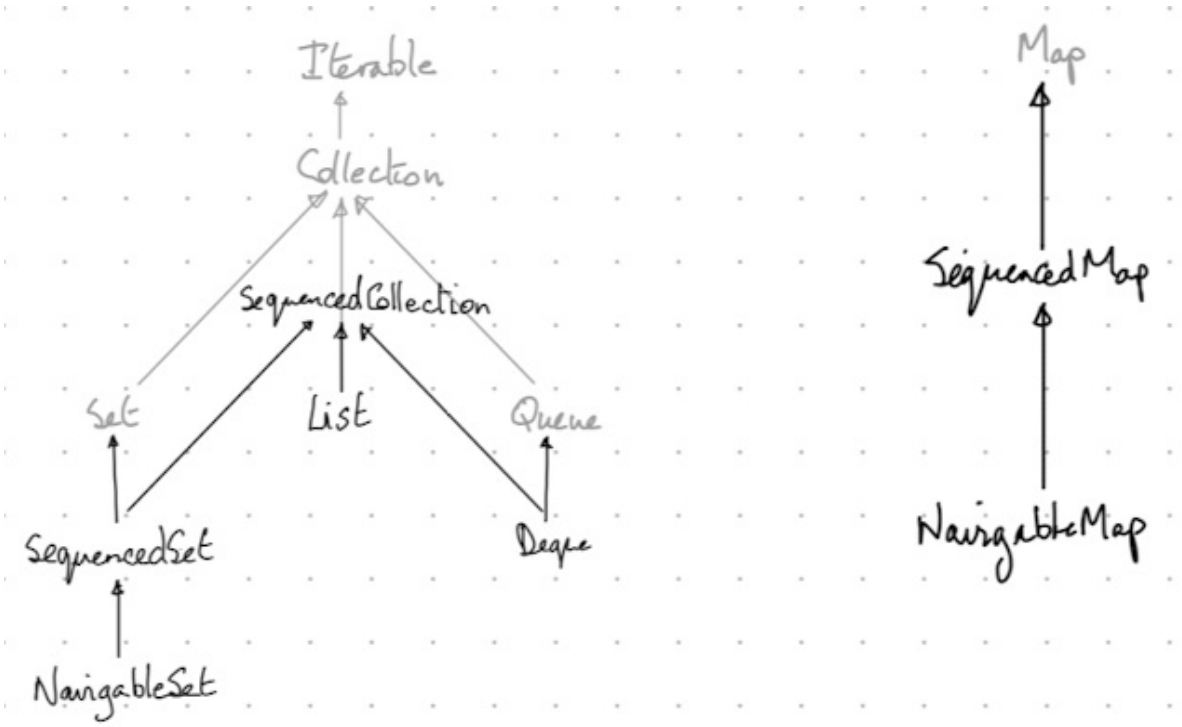

Figure 4-5. The sequenced interfaces of the Java Collections Framework

`SequencedCollection`

`SequencedCollection` provides a reversed view (see §11.6), together with the operations that all sequenced collections must support on the first and last elements of the collection: access, addition, and removal.

`SequencedSet` and `NavigableSet`

A `SequencedSet` is an externally or internally ordered `Set` that also exposes the methods of `SequencedCollection`. A `NavigableSet` is an internally ordered `SequencedSet` that therefore also automatically sorts its elements, and also provides additional methods to find elements adjacent to a target value.

`Deque`

A `Deque` is a double-ended queue that can both accept and yield up elements at either end.

`SequencedMap` and `NavigableMap`

A `SequencedMap` is a `Map` whose keys form a `SequencedSet`. A

`NavigableMap` is a `SequencedMap` whose keys form a `NavigableSet`, so that its entries are automatically sorted by the key ordering, and its methods can find keys and key-value pairs adjacent to a target key value.

Chapters Chapter 6 through Chapter 11 will concentrate on each of the Collections Framework interfaces in turn. First, though, in Chapter 5, we need to cover some preliminary ideas which run through the entire Framework design.

Chapter 5. Preliminaries

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

In this chapter, we will take time to discuss the concepts underlying the framework, before we get into the detail of the collections themselves.

Iterable and Iterators

An iterator is an object that implements the interface `Iterator`:

```
public Iterator<E> {
  boolean hasNext();   // return true if the iteration has further
elements
  E next();            // return the next element in the iteration
  void remove();       // remove the last element returned by the
iterator
}
```

The purpose of iterators is to provide a uniform way of accessing collection elements sequentially, so whatever kind of collection you are dealing with, and however it is implemented, you always know how to process its elements in turn. Iterators are often not used directly, because in many situations the *foreach* statement is more convenient: compare the code to print every element of a collection `coll` using *foreach*:

```
for (Object o : coll) {
  System.out.println(o);
}
```

with code explicitly using an iterator:

```
for (Iterator itr = coll.iterator() ; itr.hasNext() ; ) {
  System.out.println(itr.next());
}
```

The target of a *foreach* statement can be an array, or any class that implements the interface `Iterable`:

```
public Iterable<T> {
  Iterator<T> iterator();   // return an iterator over elements of
type T
}
```

The `Collection` interface extends `Iterable`, so any set, list, or queue can be the target of *foreach*. If you write your own implementation of `Iterable`, that too can be used with *foreach*. ??? shows an example. A `Counter` object is initialized with a count of `Integer` objects; its iterator returns these in ascending order in response to calls of `next`. Now `Counter` objects can be the target of a *foreach* statement:

```
int total = 0;
for (int i : new Counter(3)) {
   total += i;
}
assert total == 6;
```

In practice, it is unusual to implement `Iterable` directly in this way, as *foreach* is most commonly used with arrays and the standard collections classes. Direct use of `Iterators`, rather than a *foreach* statement, is necessary mainly when you want to make a *structural* change to a collection—broadly speaking, adding or removing elements—in the course of iteration. As we just saw, the `Iterator` interface exposes only a method for removal of collection elements, though its subinterface `ListIterator`, available to `List` implementations, also provides methods to add and replace elements.

At one time or another, you may have made the mistake of trying, during iteration, to make a structural change directly—that is, rather than using one of the iterator methods. If the collection that you were iterating over was one of the general-purpose `Collection` implementations —`ArrayList`, `HashMap`, and so on—you may have been puzzled by seeing `ConcurrentModificationException` thrown from single-threaded code. The iterators of these collections throw this exception whenever they detect that the collection from which they were derived has been structurally modified—except by the iterator itself, of course.

The motivation for this behavior is that a structural change made during iteration is possibly evidence that another thread is accessing the collection; structural changes that you wish the iterating thread to make should be made using the iterator. Allowing another thread to access a non-thread-safe collection will probably result in a failure some time later, when it will be difficult to diagnose. To avoid this problem the general-purpose Collections Framework iterators are *fail-fast*: their methods check for any structural modifications made since the last iterator method call, throwing `ConcurrentModificationException` if they detect one. Although this restriction rules out some sound programs, it rules out many more unsound ones.

```
class Counter implements Iterable<Integer> {
  private int count;
  public Counter(int count) { this.count = count; }
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      private int i = 0;
      public boolean hasNext() { return i < count; }
      public Integer next() { i++; return i; }
      public void remove(){ throw new UnsupportedOperationException();
}
    };
  }
}
```

Now `Counter` objects can be the target of a *foreach* statement:

```
int total = 0;
for (int i : new Counter(3)) {
   total += i;
}
```

```
assert total == 6;
```

In practice, it is unusual to implement `Iterable` directly in this way, as *foreach* is most commonly used with arrays and the standard collections classes. Direct use of `Iterators`, rather than a *foreach* statement, is necessary mainly when you want to make a *structural* change to a collection—broadly speaking, adding or removing elements—in the course of iteration. As we just saw, the `Iterator` interface exposes only a method for removal of collection elements, though its subinterface `ListIterator`, available to `List` implementations, also provides methods to add and replace elements.

At one time or another, you may have made the mistake of trying, during iteration, to make a structural change directly—that is, rather than using one of the iterator methods. If the collection that you were iterating over was one of the general-purpose `Collection` implementations —`ArrayList`, `HashMap`, and so on—you may have been puzzled by seeing `ConcurrentModificationException` thrown from single-threaded code. The iterators of these collections throw this exception whenever they detect that the collection from which they were derived has been structurally modified—except by the iterator itself, of course.

The motivation for this behavior is that a structural change made during iteration is possibly evidence that another thread is accessing the collection; structural changes that you wish the iterating thread to make should be made using the iterator. Allowing another thread to access a non-thread-safe collection will probably result in a failure some time later, when it will be difficult to diagnose. To avoid this problem the general-purpose Collections Framework iterators are *fail-fast*: their methods check for any structural modifications made since the last iterator method call, throwing `ConcurrentModificationException` if they detect one. Although this restriction rules out some sound programs, it rules out many more unsound ones.

The concurrent collections have other strategies for handling concurrent modification, such as weakly consistent iterators. We discuss them in more detail in “Collections and Thread Safety”.

Implementations

We have looked briefly at the interfaces of the Collections Framework, which define the behavior that we can expect of each collection. But as we mentioned in the introduction to this chapter, there are several ways of implementing each of these interfaces. Why doesn't the Framework just use the best implementation for each interface? That would certainly make life simpler—too simple, in fact, to be anything like life really is. If an implementation is a greyhound for some operations, Murphy's Law tells us that it will be a tortoise for others. Because there is no "best" implementation of any of the interfaces, you have to make a tradeoff, judging which operations are used most frequently in your application and choosing the implementation that optimizes those operations.

The three main kinds of operations that most collection interfaces require are insertion and removal of elements by position, retrieval of elements by content, and iteration over the collection elements. The implementations provide many variations on these operations, but the main differences among them can be discussed in terms of how they carry out these three. In this section, we'll briefly survey the four main structures used as the basis of the implementations and later, as we need them, we will look at each in more detail. The four structures are:

Arrays

> These are the structures familiar from the Java language—and just about every other programming language since Fortran. Because arrays are implemented directly in hardware, they have the properties of random-access memory: very fast for accessing elements by position and for iterating over them, but slower for inserting and removing elements at arbitrary positions (because that may require adjusting the position of other elements). Arrays are used in the Collections Framework as the backing structure for `ArrayList`, `CopyOnWriteArrayList`, `EnumSet` and `EnumMap`, and for many of the `Queue` and `Deque` implementations. They also form an important part of the mechanism for implementing hash tables (discussed shortly).

Linear linked lists

As the name implies, these consist of chains of linked cells. Each cell contains a reference to data and a reference to the next cell in the list (and, in some implementations, the previous cell). Linked lists perform quite differently from arrays: accessing elements by position is slow, because you have to follow the reference chain from the start of the list, but insertion and removal operations can be performed in constant time by rearranging the cell references. Linked lists are the primary backing structure used for the classes `ConcurrentLinkedQueue`, `LinkedBlockingQueue`, and `LinkedList`.

Other linked data structures

Linked structures are particularly suitable for representing nonlinear types like trees and skip lists, especially if they need to be rearranged as new elements are added. Such structures provide an inexpensive way of maintaining sorted order in their data, allowing fast searching by content. Trees are the backing structures for `TreeSet` and `TreeMap`. Skip lists are used in `SkipListSet` and `SkipListMap`. Priority heaps, used in the implementation of `PriorityQueue` and `PriorityBlockingQueue`, are tree-related structures.

Hash tables

These provide a way of storing elements indexed on their content rather than on an integer-valued index, as with lists. In contrast to arrays and linked lists, hash tables provide no support for accessing elements by position, but access by content is usally very fast, as are insertion and removal. Hash tables are the backing structure for many `Set` and `Map` implementations, including `HashSet` and `LinkedHashSet` together with the corresponding maps `HashMap` and `LinkedHashMap`, as well as `WeakHashMap`, `IdentityHashMap` and `ConcurrentHashMap`.

The content-based indexing of hashed collections depends on two `Object` methods, `hashCode` and `equals`, which are applied in succession to position an element for insertion or to locate it for retrieval. So it is in the contracts for these methods, specifically for `Object.hashCode`, that we find the

requirements that hashed collections place on their relationship. The crucial requirement, sometimes overlooked—with disastrous consequences—by Java programmers, is that if two objects are equal according to the `equals` method, then calling `hashCode` on each must produce the same result. If `hashCode` depends on an instance field—or any other data about an object—that is not used by the `equals` method, then the first stage of a retrieval operation will very likely be misdirected. One way in which this can occur is if you do not override `Object.hashCode` at all; the value that it returns in this case will be implementation-dependent (in OpenJDK it is usually randomly generated), but is in any case highly unlikely to be the same for two different instances.

The toy class `Person` is a case in point:

```
class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public boolean equals(Object o) {
        return o instanceof Person p && name.equals(p.name);
    }
}
```

This class should override `hashCode` so that it depends only on the same field, *i.e.* `name`, that `equals` depends on. But it does not; as a result, hashed collections will not work correctly with it:

```
Set<Person> people = new HashSet<>();
people.add(new Person("Alice"));
assert ! people.contains(new Person("Alice"));
```

Views

In the Collections Framework, a *view* of a data structure provides a way of working with it as though it had been transformed in some way—either into a differently-organised structure of the same type, or a structure of another type completely. The most obvious examples of this, in a general sense, are the backing structures discussed in the last section. But even though replacing an element in an `ArrayList`, for example, is implemented by replacing an

element in the underlying array, we don't normally refer to an `ArrayList` as a view of an array. That's because `ArrayList` is more than just a view; it's capable of insulating its user from the limitations of arrays: for example, you frequently want to add elements to a list. Arrays can't support that, but `ArrayList` hides that limitation, if necessary by transferring all its elements to a new and larger backing array. So an `ArrayList` is much more than a simple view of a single array, and that term isn't usually used to describe it.

Sometimes, however, a simple view is what is needed. For example, suppose we want to test for the presence of a particular object in an array. One obvious way to do this is to iterate over the array elements, testing each for equality with the search target. An alternative would be to use the `contains` method of the `List` interface to do that work instead. An array is not a `List`, though, so how can a `List` method be useful in handling it? We might well want to avoid the overhead of creating a new `ArrayList` object, physically copying all the elements of the array into a new collection. In this situation, a better answer is to get a `List` *view* of the array—an object that "looks like" a `List`, but implements all its operations directly on the underlying array. The method `asList` of the utility class `Arrays` provides such a view. The simple view that it returns supports some `List` operations, such as `contains`, and methods like `get` and `set` which access or replace the array elements, but it won't allow you to make *structural modifications*, like adding or removing elements, which aren't supported by the underlying array.

The data "of" the view actually resides in the underlying structure, so changes made to that structure are immediately visible in the view, and vice versa. For example, the following code compiles and runs without errors:

```
Integer[] arr = {1,2,3};
var list = Arrays.asList(arr);
list.set(0,3);
assert arr[0] == 3;
arr[2] = 0;
assert list.get(2) == 0;
```

The Collections API exposes many methods returning views—for example, the keys of a `Map` can be viewed as a `Set`, as can its entries; collections can be viewed as unmodifiable, and so on. Each of these views has different rules

dictating which modifications they will allow: some allow certain structural modifications, some allow only non-structural modifications, and some are fully unmodifiable. So the interfaces that these views implement have some of their operations labelled *optional*: this is perhaps the most controversial aspect of the Java Collections Framework design. We'll discuss it further in the section "Contracts" and in Chapter [Link to Come]. The views themselves will be discussed in subsequent chapters, each one in the context of its backing collection.

Performance

The usefulness of a collections library depends on the impact of the performance of its collections as part of a working system. Unfortunately, this is very difficult both to predict in theory and to assess in practice. Many factors contribute to it, including:

- how often any of the collection's operations are executed
- which operations are executed most frequently
- the time cost of each of the operations that are executed
- how much garbage each produces, and what overhead is incurred in collecting it
- the *locality* properties (see "Memory") of the collection
- how much parallelism is involved, both at instruction and thread level

The study of how these factors combine to affect the speed of a real-life system belongs to the subject of performance tuning, of which the most important rule is often quoted in the form provided by Donald Knuth ([Knuth74]): "premature optimization is the root of all evil." The explanation of this remark is twofold: firstly, for many programs, performance is just not a critical issue. If a program is rarely executed or already uses few resources, optimization is a waste of effort, and indeed may well be harmful. Secondly, even for performance-critical programs, assessing *which* part is critical normally requires accurate measurement: in the same paper, Knuth added "It is often a mistake to make *a*

priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail."

This carries an important implication about comparing the performance of different collections. For example, `CopyOnWriteArrayList` provides highly efficient concurrent read operations, at the cost of very expensive writes. So to use it in a system that requires highly performant concurrent access to a `List` you have to have confidence, gained by measurement if necessary, that read operations greatly outnumber writes.

Memory

Traditionally, an algorithm was assessed on the basis of its use of two resources: time and space. With the forty-year exponential decrease in the cost of memory from the 1970s onwards, space complexity became less important, and the focus sharpened on time complexity (see "Instruction Count and the *O*-notation"). But modern trends in both software and hardware systems architecture have complicated the picture by reintroducing memory concerns as a major focus. The software concern is relatively straightforward: for Java programs, memory temporarily allocated in the course of program execution must be reclaimed by the garbage collector, and garbage collection is an expensive overhead to the operation of any program.

The effects of modern trends in hardware design require more explanation. For four decades, from the 1970s onwards, processor speeds rose exponentially, far outstripping the speed increase of every other component. These include memory and the memory bus, the components that together are responsible for keeping processors supplied with data and instructions to work on. The different grades of memory available conform to the same rule that governs storage technologies of all kinds, including on-board memory, disk drives of all kinds, and even tape (still in use for archival storage): the cost of storage is inversely correlated with the speed of the medium. The faster the medium can store and retrieve data, the greater the cost per byte of capacity. This leads designers to create *memory hierarchies*, with large amounts of cheap storage, like disk and even tape, at the bottom, and small amounts of expensive static RAM located on-chip, physically close to the processor, at the top. Figure Figure 5-1 shows a

highly simplified view of the top part—the on-board and on-chip components—of the hierarchy. On-chip memory is organised in caches called Level 1, Level 2, and (sometimes) Level 3, each one slower and larger (and less power-hungry) than its predecessor.

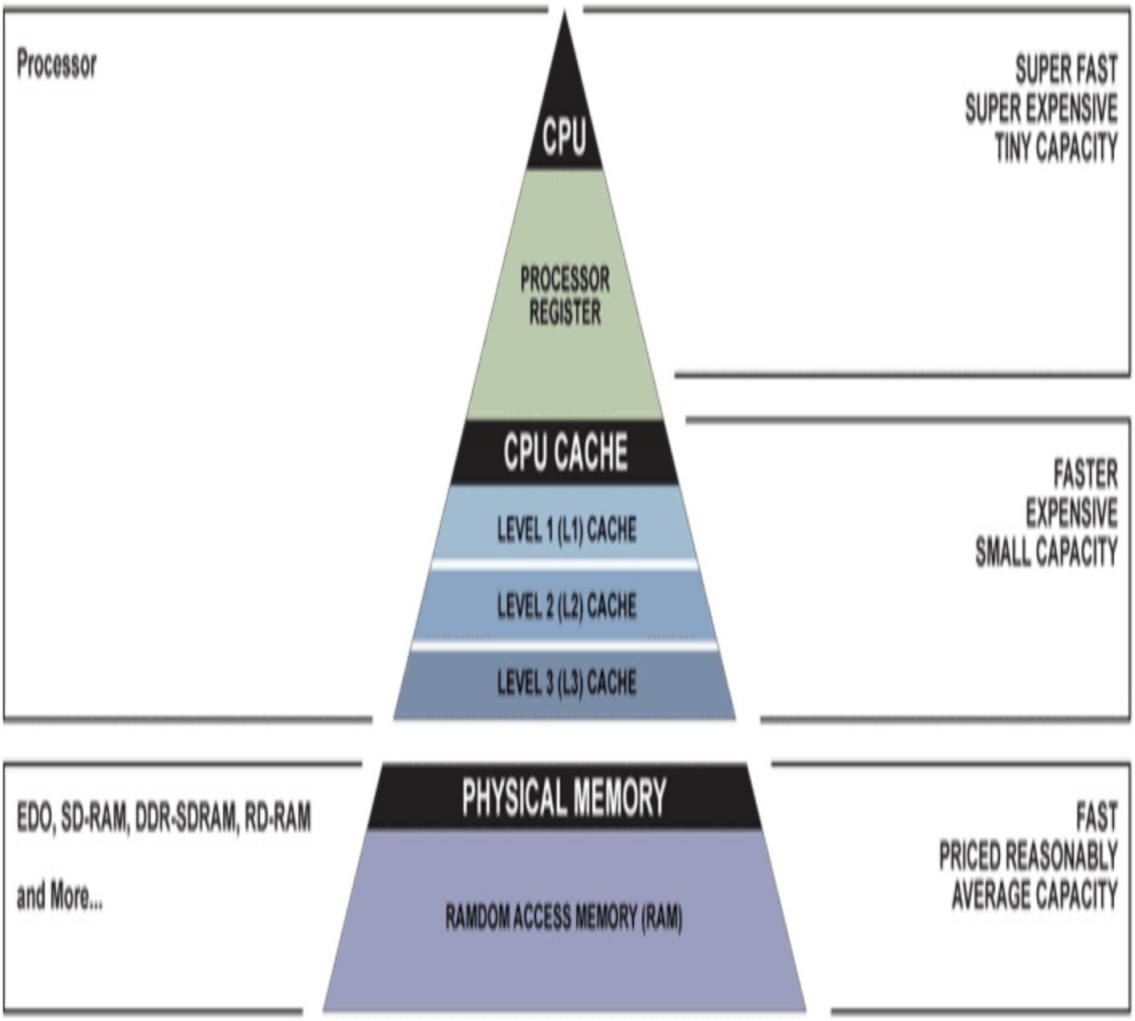

Figure 5-1. The memory hierarchy

If a data item is not available at one level, lower levels are searched until it is found. If the level is a cache, this is called a *cache miss*. Data in caches is organised in units called *cache lines*, commonly 64 bytes. If a failure to find one byte of data causes a cache miss, an entire cache line must be discarded in order to load that byte from a lower level in the hierarchy, along with the 63 bytes adjacent to it in memory. Such expensive cache misses will occur less often for programs that conform to the principle of *spatial locality*, which says that

programs tend to reuse data and instructions near to those that they have used recently. Two programs, each of which execute the same number of instructions, may have very different execution times if one exhibits spatial locality and the other does not.

Object-oriented programs often exhibit poor spatial locality, because objects can be located anywhere in memory, but some are worse than others. For example, a program iterating over an array of primitives will show good spatial locality, and also a predictable pattern of data access, enabling predictive retrieval of data leading to reduction of cache miss overheads. Iterating over an array of reference types will be worse, because although access to the array itself is predictable and localized, access to the referenced objects is not. A linked structure is worse still; memory access is in general not predictable or localized at all, so cache misses will typically be frequent and expensive. Every further level of indirection increases the likelihood of cache misses and, even more expensively, of page faults.

Instruction Count and the *O*-notation

Although, as we observed in the previous section, memory usage can no longer be separated cleanly from execution time, there is still merit in trying to make a separate estimate of the latter. Traditionally, execution time was assumed to be proportional to the count of CPU operations needed to complete a program. This assumption is also subject to some qualifications, which we'll discuss at the end of this section. First, though, how is the execution count estimated? Detailed analysis can be complex. A relatively simple example is provided in Donald Knuth's classic book Sorting and Searching ([Knuth98]), where the worst-case execution time for a multiple list insertion sort program on Knuth's notional MIX machine is derived as

$$3.5N^2 + 24.5N + 4M + 2$$

where N is the number of elements being sorted and M is the number of lists.

As a shorthand way of describing algorithm efficiency, this isn't very convenient. Clearly we need a broader brush for general use. The one most commonly used is the *O-notation* (pronounced "big-oh notation"). The *O*-

notation is a way of describing the performance of an algorithm in an abstract way, without the detail required to predict the precise performance of a particular program running on a particular machine. the main reason for using it is that it gives us a way of describing how the execution time for an algorithm depends on the size of its data set, provided the data set is large enough. For example, in the previous expression the first two terms are comparable for low values of N; in fact, for $N < 8$, the second term is larger. But as N grows, the first term increasingly dominates the expression and, by the time it reaches 100, the first term is 15 times as large as the second one. Using a very broad brush, we say that the worst case for this algorithm takes time $O(N^2)$. We don't care too much about the coefficient because that doesn't make any difference to the single most important question we want to ask about any algorithm: what happens to the execution count when the data size increases—say, when it doubles? For the worst-case insertion sort, the answer is that the execution count goes up fourfold. That makes $O(N^2)$ pretty bad—worse than any we will meet in practical use in this book.

Table 5-1. Some common Big-O classes

Time	Common name	Effect on the execution count if N is doubled	Example algorithms
$O(1)$	Constant	Unchanged	Insertion into a hash table ("Set Implementations")
$O(\log N)$	Logarithmic	Increased by a constant amount	Insertion into a tree ("TreeSet")
$O(N)$	Linear	Doubled	Linear search
$O(N \log N)$		Doubled plus an amount proportional to N	Merge sort ("Changing the Order of List

			Elements”)
$O(N^2)$	Quadratic	Increased fourfold	Bubble sort

Table 5-1 shows some commonly found execution counts, together with examples of algorithms to which they apply. For example, many other execution counts are possible, including some that are much worse than those in the Figure. Many important problems can be solved only by algorithms that take $O(2^N)$—for these, when N doubles, the execution count is squared! For all but the smallest data sets, such algorithms are infeasibly slow.

Sometimes we have to think about situations in which the cost of an operation varies with the state of the data structure. For example, adding an element to the end of an `ArrayList` can normally be done in constant time, unless the `ArrayList` has reached its capacity. In that case, a new and larger array must be allocated, and the contents of the old array transferred into it. The cost of this operation is linear in the number of elements in the array, but it happens relatively rarely. In situations like this, we calculate the *amortized cost* of the operation—that is, the total cost of performing it n times divided by n, taken to the limit as n becomes arbitrarily large. In the case of adding an element to an `ArrayList`, the total cost for N elements is $O(N)$, so the amortized cost is $O(1)$.

Big-O analysis of execution counts is often very useful for a broad-brush comparisons of the running times of programs and, for that reason, each interface chapter of this book includes big-O analysis of common operations for each implementation. All the same, there are reasons for caution in treating O-numbers as a proxy for running times. They include:

- the assumption in determining the dominant term is that the data size will always be sufficiently large to outweigh constant factors and multipliers; for example, $O(N + c)$ is taken to be $O(N)$ for constant c, on the assumption that in situations where it matters, $N >> c$. This may not be true for very large factors;
- machine instructions do not all have the same execution time; for example,

inserting an element at the first position of an `ArrayList` has complexity $O(N)$, because every element must be moved to a position one place higher in the array. But if it is possible for that operation to be executed with a single hardware instruction, then the actual execution time may compare favorably with the same operation on a `LinkedList`, even though there it has complexity of $O(1)$;

- parallelism complicates the picture still further. With instruction-level parallelism, instructions are processed in stages, with the different stages of successive instructions being executed simultaneously. So the elapsed execution time of a program is normally much less than the summed execution times of its individual instructions. The same applies, on a larger scale to thread-level parallelism, in which different threads can process their tasks simultaneously except when they are in contention for some resource.

In conclusion, it is worth repeating the value of theoretical performance analysis, including *O*-numbers: it can provide valuable guidance at the design stage. You would never choose an $O(N^2)$ algorithm over an $O(N)$ one for a large data set. But if you have a running system with a performance problem, there is no substitute for accurate measurement to compare different candidate implementations.

Immutability and Unmodifiability

Functional programming is an attractive style; at their best, functional programs are elegant and demonstrably sound, much more so than object-oriented programs. For this reason, some of the features of functional languages that provide these advantages have been gradually adopted into Java, starting with generics, continuing with streams and lambdas (see “Lambdas and Streams”), and culminating with pattern-matching, at the time of this writing still a work in progress. One important feature of the functional style is that its data structures are *immutable*—that is, their state cannot be modified after their creation. Immutability confers a number of advantages on a program:

- Immutable objects are thread-safe
- Immutable objects are perfectly encapsulated, so removing the need for

defensive copying

- Immutability guarantees stable lookup in keyed and ordered collections (see "Contracts")
- Immutability reduces the number of states that a program can be in, making it simpler, clearer, and easier to understand and reason about

Realising these advantages in a Java program is difficult, however. For an object to be truly immutable, its entire object graph—that is, the object itself and everything it refers to, directly or indirectly—must not observably change after construction. Obviously, immutable graph components like wrapper objects and strings present no problem, but for mutable components the often difficult (and rarely achieved) requirement is that the graph must have guaranteed exclusive access to them.

This has led to alternative conflicting terminologies for describing immutability of collections:

- Frameworks like Guava and Eclipse collections refer to immutability of an entire object graph as *deep immutability*. They refer to a collection that refuses modification at the first level—that is, an attempt to add, remove, or replace an element—as *shallow immutability*, or often just *immutability*.
- The Java collections documentation uses *immutability* to mean immutability of the entire object graph. What the other frameworks refer to as "shallow immutability", the Java documentation—and this book—call *unmodifiability*.

Unmodifiability is a kind of partial immutability that doesn't fully confer any of the advantages of immutability listed above, so you might well question its usefulness. But it does confer real advantages:

- Collections of immutable objects, including wrapper objects and strings, are not uncommon, and unmodifiable collections of these provide the full benefits of immutability.
- Even partial immutability reduces the number of program states that you have to consider when understanding a program and reasoning about its correctness.

- Because unmodifiable sets and maps can be backed by arrays instead of hashed structures, they can provide very significant space savings.

Chapter [Link to Come] discusses further the situations in which you would choose to use unmodifiable collections.

Contracts

In reading about software design, you are likely to come across the term *contract*, often without any accompanying explanation. In fact, software engineering gives this term a meaning that is very close to what people usually understand a contract to be. In everyday usage, a contract defines what two parties can expect of each other—their obligations to each other in some transaction. If a contract specifies the service that a supplier is offering to a client, the supplier's obligations are obvious. But the client, too, may have obligations—including, but not only, the obligation to pay—and failing to meet them will automatically release the supplier from their obligations as well. For example, airlines' conditions of carriage—for the class of tickets that your authors can afford, anyway—release them from the obligation to carry passengers who have failed to turn up on time. This allows the airlines to plan their service on the assumption that all the passengers they are carrying are punctual; they do not have to incur extra work to accommodate clients who have not fulfilled their side of the contract.

Contracts work the same way in software. If the contract for a method lays down preconditions on its arguments (*i.e.* the obligations that a client must fulfill), the method is required to return its contracted results only when those preconditions are fulfilled. For example, binary search (see "Finding Specific Values in a List") is a fast algorithm to find a key within an ordered list, and it fails if you apply it to an unordered list. So the contract for `Collections.binarySearch` can say, "if the list is unsorted, the results are undefined", and the implementer of binary search is free to write code which, given an unordered list, returns random results, throws an exception, or even enters an infinite loop. In practice, this situation is relatively rare in the contracts of the core API because, instead of restricting input validity, they mostly allow for error states in the preconditions and specify the exceptions that the method must throw if it gets bad input. This

design is appropriate for general libraries such as the Collections Framework, which will be very heavily used in widely varying situations by programmers of widely varying ability. APIs of less general libraries usually avoid it, because it restricts the flexibility of the supplier unnecessarily. In principle, all that a client should need to know is how to keep to its side of the contract; if it fails to do that, all bets are off and there should be no need to say exactly what the supplier will do.

It's good practice in Java to code to an interface rather than to a particular implementation, so as to provide maximum flexibility in choosing implementations. For that to work, what does it imply about the behavior of implementations? If your client code uses methods of the `List` interface, for example, and at run time the object doing the work is actually an `ArrayList`, you would like to know that the assumptions you have made about how `Lists` behave are true for `ArrayLists` also. So a class implementing an interface usually has to fulfill all the obligations laid down by the terms of the interface contract. (A weaker form of these obligations is already imposed by the compiler: a class claiming to implement an interface must provide concrete method definitions matching the declarations in the interface, but contracts take this further by specifying the behavior of these methods as well.)

The obligations imposed by the Collections Framework interfaces are unusual in some respects, however. A guiding principle in the design of the Framework was that it should be simple—a vital characteristic in a library that every Java programmer must learn. Frameworks that separate modifiable from unmodifiable interfaces have many more types than the Java Collections Framework, so the Framework designers decided to avoid this separation. But since some collection views (and, since Java 9, unmodifiable collections also) do not support all write operations, the interface contracts label these operations *optional*. For example, the `Set` view of a `Map`'s keys can have elements removed but not added (see Chapter 11), while other view collections can have elements neither added nor removed (e.g., the list view returned by `Arrays.asList`), or support no modification operations at all, like collections that have been wrapped in an unmodifiable wrapper (see "Unmodifiable Collections"). The interface contracts provide for implementations to throw `UnsuppportedOperationException` when

optional methods are called. This feature of the Java Collections Framework is sometimes criticised as contravening the so-called Liskov Substitution Principle (LSP), named for the pioneering computing scientist Barbara Liskov. This "principle" is not in fact an unbreakable *principle*, but a *description* of systems in which any use of a supertype, such as an interface, can be replaced by an instance of one of its subtypes, such as an implementing class, without changing the meaning of the program. Actually, the fact that the interface contracts specify write operations as optional means that the LSP does indeed describe the behavior of the views and unmodifiable collections. (Critics of the Collections Framework will dismiss this argument as hairsplitting.)

A problem with this approach is that a client programmer, passed a collection of some interface type, cannot be confident that it will support optional operations without throwing `UnsupportedOperationException`. A safe rule of thumb for handling collections is to avoid writing to them unless you know the implementing type, usually because you yourself own the collection. We'll discuss this guidance further in [Link to Come], and in the chapter [Link to Come] we'll attempt to evaluate the decisions that led to the design of the Framework as it is today.

Although until now we have only discussed functional requirements, contracts can also require performance guarantees. To fully understand the performance characteristics of a class, however, you often need to know detail about the algorithms of the implementation. In this Chapter, while we concentrate mainly on contracts and how as a client programmer you can make use of them, we also provide further implementation detail from the platform classes where it might be of interest. This can be useful in deciding between implementations, but remember that it is not stable; while contracts are binding, one of the main advantages of using them is that they allow implementations to change as better algorithms are discovered or as hardware improvements change their relative merits. And of course, if you are using another implementation, algorithm details not governed by the contract may be entirely different.

Content-based Organization

The discussion of sequenced collections ("Sequenced Collections") introduced the idea of internally ordered sequences, in which the elements are automatically

positioned according to the order of their contents, as defined by some ordering relation (see Chapter 2). These are not the only collections classes to organise their elements automatically according to their contents: hashed collections, priority queues and delay queues have the same property. In the cases of internally ordered sequences and of the queues, the content-based organization is observable via the operations of the collection; not so in the case of the hashed collections, but their organization is used when elements need to be located. What all these collections have in common is an implicit invariant: for them to operate correctly, the value of the fields used to position the elements must not change once they have been inserted into the collection. For example, consider a different version of the class `Person`, this time defined with a `hashCode` method, but one that depends on the value of the mutable field `name`:

```
class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int hashCode() {
        return name.hashCode();
    }
    public boolean equals(Object o) {
        return o instanceof Person p && name.equals(p.name);
    }
}
```

Now an object of the type `Person` can be reliably retrieved, but only if the value of `name` is unchanged. If that field is modified, the collection can no longer match the object with either the old or the new value:

```
Set<Person> people = new HashSet<>();
Person alice = new Person("Alice");
people.add(alice);
moe.setName("Bob");
assert ! people.contains(new Person("Alice"));
assert ! people.contains(new Person("Bob"));
```

To avoid problems like this, the best guideline is this: whenever you are storing

objects in a `Set`, a `Map`, or an internally ordered `Queue`, ensure that the fields used by the collection to organize its contents are immutable.

Lambdas and Streams

We saw, in "Immutability and Unmodifiability", that the advantages of the functional programming style have led object-oriented languages towards adoption of many functional language features. Following the introduction of generics in Java 5, the next big innovation came in Java 8, with the introduction of lambdas and streams. Lambdas introduced a way in which functions could be represented in Java programs. For example, the statement:

```
Function<Integer,Integer> doubleInt = x -> x * 2;
```

declares a function `doubleInt`, which takes an argument of type `Integer` and returns the `Integer` result of doubling it.

The importance of lambdas for collections and collection processing comes from their use in stream operations. Streams are a mechanism for transporting a sequence of values from a source to a destination through a series of operations, typically implemented as lambdas, each of which can transform, drop, or insert values on the way. This provides an alternative model to the traditional use of collections for aggregate data processing. A simple, if rather contrived, example illustrates the change in thinking. (Taken from the introduction to *Mastering Lambdas*, by Maurice Naftalin (Oracle Press, 2015)). The record `Point` represents a point defined by its x and y co-ordinates, with a single method `distanceFrom` that calculates its distance from another point.

```
record Point(int x, int y) {
    public double distanceFrom(Point p) { .... }
}
```

Without streams, a common model of bulk data processing is to process collections in a series of stages: a collection is iteratively processed to produce a new collection, which in turn is iteratively processed, and so on. The following code starts with a collection of `Integer` instances, applies an arbitrary transformation to produce a set of `Point` instances, and finally finds the

maximum among the distances of each `Point` from the origin.

```
Point origin = new Point(0, 0);
List<Integer> intList = Arrays.asList(1, 2, 3, 4, 5);
List<Point> pointList = new ArrayList<>();
for (Integer i : intList) {
    pointList.add(new Point(i % 3, i / 3));
}
double maxDistance = Double.MIN_VALUE;
for (Point p : pointList) {
    maxDistance = Math.max(p.distanceFrom(origin), maxDistance);
}
```

The real code for which this is a model has several disadvantages: it is very verbose; the intermediate collection `pointList` is an overhead on the operation of the program, resulting in increased garbage collection costs or even in heap space exhaustion; there is an implicit assumption, dif lt to spot, that the minimum value of an empty list is `Double.MIN_VALUE`; perhaps worst of all, the intent of the program is hard to discern, because the crucial operations are interspersed with the code for collection handling.

Here is the equivalent code, expressed as a pipeline processing the original collection `intList` through a series of transformations:

```
Point origin = new Point(0, 0);
List<Integer> intList = Arrays.asList(1, 2, 3, 4, 5);
OptionalDouble maxDistance =
   intList.stream()
      .map(i -> new Point(i % 3, i / 3))
      .mapToDouble(p -> p.distanceFrom(origin))
      .max();
```

This example shows in miniature the advantages of stream code: it is more concise and readable, uses less intermediate storage, handles an empty source gracefully, and can never attempt to mutate the source collection.

From here on in this book, where example collection code can benefit from partial conversion to the use of streams, we will show stream code as an alternative in a sidebar.

Collections and Thread Safety

When a Java program is running, it is executing one or more execution streams, or *threads*. A thread is like a lightweight process, so a program simultaneously executing several threads can be thought of as a computer running several programs simultaneously, but with one important difference: different threads can simultaneously access the same memory locations and other system resources. On machines with multiple processors, truly concurrent thread execution can be achieved by assigning a processor to each thread. If, however, there are more threads than processors—the usual case—multithreading is implemented by *time slicing*, in which a processor executes some instructions from each thread in turn before switching to the next one.

There are two good reasons for using multithreaded programs. An obvious one, in the case of multicore and multiprocessor machines, is to share the work and get it done quicker. (This reason is becoming ever more compelling as hardware designers turn increasingly to parallelism as the way of improving overall performance.) A second one is that two operations may take varying, perhaps unknown, amounts of time, and you do not want the response to one operation to await the completion of the other. This is particularly true for a graphical user interface GUI), where the response to the user clicking a button should be immediate, and should not be delayed if, say, the program happens to be running compute-intensive part of the application at the time.

Although concurrency may be essential to achieving good performance, it comes at a price. Different threads simultaneously accessing the same memory location can produce unexpected results, unless you take care to constrain their access. Consider Example 5-1, in which the class `ArrayStack` uses an array and an index to implement the interface `Stack`, which models a stack of `int` (despite the similarity of names, this example is different from [Link to Come]). For `ArrayStack` to work correctly, the variable `index` should always point at the top element of the stack, no matter how many elements are added to or removed from the stack. This is an *invariant* of the class. Now think about what can happen if two threads simultaneously attempt to push an element on to the stack. As part of the `push` method, each will execute the lines `//1` and `//2`, which are correct in a single-threaded environment but in a multi-threaded environment may break the invariant. For example, if thread A executes line `//1`, thread B

executes line `//1` and then line `//2`, and finally thread A executes line `//2`, only the value added by thread B will now be on the stack, and it will have overwritten the value added by thread A. The stack pointer, though, will have been incremented by two, so the value in the top position of the stack is whatever happened to be there before. This is called a *race condition*, and it will leave the program in an inconsistent state, likely to fail because other parts of it will depend on the invariant being true.

Example 5-1. A non-thread-safe stack implementation

```
interface Stack {
  public void push(int elt);
  public int pop();
  public boolean isEmpty();
}

Stack
class ArrayStack implements Stack{
  private final int MAX_ELEMENTS = 10;
  private int[] stack;
  private int index;
  public ArrayStack() {
    stack = new int[MAX_ELEMENTS];
    index = -1;
  }
  public void push(int elt) {
    if (index != stack.length - 1) {
      index++;                                          //1
      stack[index] = elt;                               //2
    } else {
      throw new IllegalStateException("stack overflow");
    }
  }
  public int pop() {
    if (index != -1) {
      return stack[index];
      index--;
    } else {
      throw new IllegalStateException("stack underflow");
    }
  }
  public boolean isEmpty() { return index == -1; }
}
```

The increasing importance of concurrent programming during the lifetime of Java has led to a corresponding emphasis in the collections library on flexible and efficient concurrency policies. As a user of the Java collections, you need a

basic understanding of the concurrency policies of the different collections in order to know how to choose between them and how to use them appropriately. In this section, we'll briefly outline the different ways in which the Framework collections handle concurrency, and the implications for the programmer. For a full treatment of the general theory of concurrent programming, see *Concurrent Programming in Java* ([Lea99]), and for more detail about concurrency in Java and the collections implementations, see *Java Concurrency in Practice* ([Goetz06]).

Synchronization and the Legacy Collections

Code like that in `ArrayStack` is not *thread-safe*—it works when executed by a single thread, but may break in a multi-threaded environment. Since the incorrect behavior we observed involved two threads simultaneously executing the `push` method, we could change the program to make that impossible. Using `synchronized` to modify the declaration of the `push` method will guarantee that once a thread has started to execute it, all other threads are excluded from that method until the execution is done:

```
public synchronized void push(int elt) { ... }
```

This is called *synchronizing* on a *critical section* of code, in this case the whole of the `push` method. Before a thread can execute synchronized code, it has to get the *lock* on some object—by default, as in this case, the current object. While a lock is held by one thread, another thread that tries to enter any critical section synchronized on that lock will *block*—that is, will be suspended—until it can get the lock. This synchronized version of `push` is thread-safe; in a multi-threaded environment, each thread behaves consistently with its behavior in a single-threaded environment.To safeguard the invariant and make `ArrayStack` as a whole thread-safe, the methods `pop` and `isEmpty` must also be synchronized on the same object. The method `isEmpty` doesn't write to shared data, so synchronizing it isn't required to prevent a race condition, but for a different reason. Each thread may use a separate memory cache, which means that writes by one thread may not be seen by another unless either they both take place within blocks synchronized on the same lock, or unless the variable is marked with the `volatile` keyword.

Full method synchronization was, in fact, the policy of the collection classes provided in JDK1.0: `Vector`, `Hashtable`, and their subclasses; all methods that access their instance data are synchronized. These are now regarded as legacy classes to be avoided because of the high price this policy imposes on all clients of these classes, whether they require thread safety or not. Synchronization can be very expensive: forcing threads to queue up to enter the critical section one at a time slows down the overall execution of the program, and the overhead of administering locks can be very high if they are often contended.

Java 2: Synchronized Collections and Fail-Fast Iterators

The performance cost of internal synchronization in the JDK 1.0 collections led the designers to avoid it when the Collections Framework was first introduced in JDK 1.2. Instead, the platform implementations of the interfaces `List`, `Set`, and `Map` widened the programmer's choice of concurrency policies. To provide maximum performance for single-threaded execution, the new collections provided no concurrency control at all. (More recently, the same policy change has been made for the synchronized class `StringBuffer`, which was complemented in Java 5 by its unsynchronized equivalent, `StringBuilder`.)

Along with this change came a new concurrency policy for collection iterators. In multithreaded environments, a thread which has obtained an iterator will usually continue to use it while other threads modify the original collection. So iterator behavior has to be considered as an integral part of a collection's concurrency policy. The policy of the iterators for the Java 2 collections is to *fail fast*, as described in "Iterable and Iterators": every time they access the backing collection, they check it for structural modification (which, in general, means that elements have been added or removed from the collection). If they detect structural modification, they fail immediately, throwing `ConcurrentModificationException` rather than continuing to attempt to iterate over the modified collection with unpredictable results. Note that this fail-fast behavior is provided to help find and diagnose bugs; it is not guaranteed as part of the collection contract.

The appearance of Java collections without compulsory synchronization was a welcome development. However, thread-safe collections were still required in

many situations, so the Framework provided an option to use the new collections with the old concurrency policy, by means of synchronized wrappers (see Chapter 12). These are created by calling one of the factory methods in the `Collections` class, supplying an unsynchronized collection which it will encapsulate. For example, to make a synchronized `List`, you could supply an instance of `ArrayList` to be wrapped. The wrapper implements the interface by delegating method calls to the collection you supplied, but the calls are synchronized on the wrapper object itself. Example 5-2 shows a synchronized wrapper for the interface `Stack` of Example 5-1. To get a thread-safe `Stack`, you would write:

```
Stack threadSafe = new SynchronizedArrayStack(new ArrayStack());
```

This is the preferred idiom for using synchronized wrappers; the only reference to the wrapped object is held by the wrapper, so all calls on the wrapped object will be synchronized on the same lock—that belonging to the wrapper object itself. It's important to have the synchronized wrappers available, but don't use them more than you have to, because they suffer the same performance disadvantages as the legacy collections.

Example 5-2. A synchronized wrapper for ArrayStack

```
public class SynchronizedArrayStack implements Stack {
  private final Stack stack;
  public SynchronizedArrayStack(Stack stack) {
    this.stack = stack;
  }
  public synchronized void push(int elt) { stack.push(elt); }
  public synchronized int pop() { return stack.pop(); }
  public synchronized boolean isEmpty() { return stack.isEmpty(); }
}
```

Using Synchronized Collections Safely Even a class like `SynchronizedArrayStack`, which has fully synchronized methods and is itself thread-safe, must still be used with care in a concurrent environment. For example, this client code is not thread-safe:

```
Stack stack = new SynchronizedArrayStack(new ArrayStack());
...
// don't do this in a multi-threaded environment
if (!stack.isEmpty()) {
```

```
    stack.pop();              // can throw IllegalStateException
  }
```

The exception would be raised if the last element on the stack were removed by another thread in the time between the evaluation of `isEmpty` and the execution of `pop`. This is an example of a common concurrent program bug, sometimes called *test-then-act*, in which program behavior is guided by information that in some circumstances will be out of date. To avoid it, the test and action must be executed atomically. For synchronized collections (as for the legacy collections), this must be enforced with *client-side locking*:

```
synchronized(stack) {
  if (!stack.isEmpty()) {
    stack.pop();
  }
}
```

For this technique to work reliably, the lock that the client uses to guard the atomic action should be the same one that is used by the methods of the synchronized wrapper. In this example, as in the synchronized collections, the methods of the wrapper are synchronized on the wrapper object itself. (An alternative is to confine references to the collection within a single client, which enforces its own synchronization discipline. But this strategy has limited applicability.)

Client-side locking ensures thread-safety, but at a cost: since other threads cannot use any of the collection's methods while the action is being performed, guarding a long-lasting action (say, iterating over an entire array) will have an impact on throughput. This impact can be very large if the synchronized methods are heavily used; unless your application needs a feature of the synchronized collections, such as exclusive locking, the Java 5 concurrent collections are almost always a better option.

Java 5: Concurrent Collections

Java 5 introduced thread-safe concurrent collections as part of a much larger set of concurrency utilities, including primitives—atomic variables and locks—which give the Java programmer access to relatively recent hardware

innovations for managing concurrent threads, notably *compare-and-swap* operations, explained below. The concurrent collections remove the necessity for client-side locking as described in the previous section—in fact, external synchronization is not even possible with these collections, as there is no one object which when locked will block all methods. Where operations need to be atomic—for example, inserting an element into a `Map` only if it is currently absent—the concurrent collections provide a method specified to perform atomically—in this case, `ConcurrentMap.putIfAbsent`.

If you need thread safety, the concurrent collections generally provide much better performance than synchronized collections. This is primarily because their throughput is not reduced by the need to serialize access, as is the case with the synchronized collections. Synchronized collections also suffer the overhead of managing locks, which can be high if there is much contention. These differences can lead to efficiency differences of two orders of magnitude for concurrent access by more than a few threads.

Mechanisms

> The concurrent collections achieve thread-safety by several different mechanisms. The first of these—the only one that does not use the new primitives—is *copy-on-write*. Classes that use copy-on-write store their values in an internal array, which is effectively immutable; any change to the value of the collection results in a new array being created to represent the new values. Synchronization is used by these classes, though only briefly, during the creation of a new array; because read operations do not need to be synchronized, copy-on-write collections perform well in the situations for which they are designed—those in which reads greatly predominate over writes. Copy-on-write is used by the collection classes `CopyOnWriteArrayList` and `CopyOnWriteArraySet`.

A second group of thread-safe collections relies on compare-and-swap (CAS), a fundamental improvement on traditional synchronization. To see how it works, consider a computation in which the value of a single variable is used as input to a long-running calculation whose eventual result is used to update the variable. Traditional synchronization makes the whole computation atomic, excluding any other thread from concurrently accessing the variable. This reduces opportunities

for parallel execution and hurts throughput. An algorithm based on CAS behaves differently: it makes a local copy of the variable and performs the calculation without getting exclusive access. Only when it is ready to update the variable does it call CAS, which in one atomic operation compares the variable's value with its value at the start and, if they are the same, updates it with the new value. If they are not the same, the variable must have been modified by another thread; in this situation, the CAS thread can try the whole computation again using the new value, or give up, or—in some algorithms— continue, because the interference will have actually done its work for it! Collections using CAS include `ConcurrentLinkedQueue` and `ConcurrentSkipListMap`.

The third group uses implementations of `java.util.concurrent.locks.Lock`, an interface introduced in Java 5 as a more flexible alternative to classical synchronization. A `Lock` has the same basic behavior as classical synchronization, but a thread can also acquire it under special conditions: only if the lock is not currently held, or with a timeout, or if the thread is not interrupted. Unlike synchronized code, in which an object lock is held while a code block or a method is executed, a `Lock` is held until its `unlock` method is called (making it possible for a `Lock` to be released in a different block or method from that in which it was acquired). Some of the collection classes in this group make use of these features to divide the collection into parts that can be separately locked, giving improved concurrency. For example, `LinkedBlockingQueue` has separate locks for the head and tail ends of the queue, so that elements can be added and removed in parallel. Other collections using these locks include `ConcurrentHashMap` and most of the implementations of `BlockingQueue`.

Iterators The mechanisms described above lead to iterator policies more suitable for concurrent use than fail-fast, which implicitly regards concurrent modification as a problem to be eliminated. Copy-on-write collections have *snapshot iterators*. These collections are backed by arrays which, once created, are never changed; if a value in the collection needs to be changed, a new array is created. So an iterator can read the values in one of these arrays (but never modify them) without danger of them being changed by another thread. Snapshot iterators do not throw `ConcurrentModificationException`.

Collections which rely on CAS have *weakly consistent* iterators, which reflect

some but not necessarily all of the changes that have been made to their backing collection since they were created. For example, if elements in the collection have been modified or removed before the iterator reaches them, it definitely will reflect these changes, but no such guarantee is made for insertions. Weakly consistent iterators also do not throw `ConcurrentModificationException`.

The third group described above also have weakly consistent iterators. In Java 6 this includes `DelayQueue` and `PriorityBlockingQueue`, which in Java 5 had fail-fast iterators. This means that you cannot iterate over the Java 5 version of these queues unless they are quiescent, at a time when no elements are being added or inserted; at other times you have to copy their elements into an array using `toArray` and iterate over that instead.

Chapter 6. The Collection Interface

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 12th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

The interface `Collection<E>` defines the core functionality that we expect of any collection other than a map. It provides methods in four groups: adding elements, removing them, querying the collection's contents, and making its elements available for further processing.

Adding Elements

```
boolean add(E e)                            // adds the element e
boolean addAll(Collection<? extends E> c)  // adds the contents of c
```

The contracts for these methods specify that after their execution, the collection must contain the element or elements suppied in the argument. So if a method call fails because, for example, it has attempted to add `null` to a `null`-hostile collection, it must throw an exception. But calls can succeed without changing the contents of the collection if, for example, they attempt to add an element to a set that already contains it. In this case, the result returned will be false, indicating that the collection was unchanged by the call.

To maintain the type uniformity of the collection, these methods only allow you

to add elements, or element collections, of the parametric type. This is in contrast to the methods of the next group:

Removing Elements

```
void clear()                              // removes all elements
boolean remove(Object o)                  // removes an element o
boolean removeAll(Collection<?> c)        // removes all occurrences of
the elements in c
boolean retainAll(Collection<?> c)        // removes the elements *not*
in c
boolean removeIf(Predicate<? super E> p)
                                          // removes the elements for
which p is true
```

Apart from `clear`, which empties the collection, the methods in this group, as in the first one, return a boolean result indicating whether the collection was changed by their action.

If the argument `o` to `remove` is `null`, the method will remove a single `null` from the collection if one is present. Otherwise, if an element `e` is present for which `o.equals(e)`, it removes it; if not, it leaves the collection unchanged. The behavior of removing a single element is in contrast to `removeAll`, which removes all occurrences in this collection of every matching element in the supplied collection.

In contrast to the methods for adding elements, these methods—and those of the next group—will accept as arguments elements or element collections of any type. We'll discuss this design choice later, in the section "Using the Methods of Collection".

Querying the Contents of a Collection

```
boolean contains(Object o)            // returns true if o is present
boolean containsAll(Collection<?> c)  // returns true if all elements
of c
                                      // are present in the collection
boolean isEmpty()                     // returns true if no elements
are present
int size()                            // returns the element count (or
                                      // Integer.MAX_VALUE if that is
less)
```

The decision to make `size` return `Integer.MAX_VALUE` for extremely large collections was probably taken on the assumption that collections this large—with more than two billion elements—will rarely occur. Even so, an alternative design, throwing an exception instead of returning an arbitrary value, would have the advantage of ensuring that the contract for `size` could clearly state that if it does succeed in returning a value, that value will be correct.

Making a Collection's Contents Available for Further Processing

```
Iterator<E> iterator()        // returns an Iterator over the elements
Spliterator<E> spliterator    // returns a Spliterator over the
elements
Stream<E> stream()            // returns a sequential Stream with this
as its source
Stream<E> parallelStream()    // returns a parallel Stream with this
as its source
Object[] toArray()            // copies contents to an Object[]  ❶
<T> T[] toArray(T[] t)        // copies contents to a T[] (for any T)
❷
<T> T[] toArray(IntFunction<T[]> generator)
                              // copies contents to a T[], created by
the generator ❸
```

Description for Callout 1

❸

The first two methods in this group create objects that make the collection's elements available for processing sequentially or in parallel, as described in (new preliminaries section). As that section explains, the most common use of `Spliterator`s is internally in one of the next two methods, `stream` and `parallelStream`, which support the functional-style processing of stream elements.

The last three methods, different overloads of `toArray`, all return an array containing the elements of this collection. Overload ❶ creates a new `Object[]`, while ❷ and ❸ take a `T[]`—or, respectively, a function producing a `T[]` of a given size—and return a `T[]` containing the elements of the collection. These methods are important because many APIs, principally older ones and those for which performance is especially important, expose methods that accept or return arrays.

As discussed in [Link to Come], the arguments of the last two methods are

required in order to provide the virtual machine with the reifiable type of the array. The new array will be created during the method execution, although if the array supplied as the argument to ❷ is long enough, it is used to receive the elements of the collection, overwriting its existing elements. Reusing a supplied array in this way can be more efficient, particularly if the method is being called repeatedly. But if instead you are creating a new array, a common and straightforward idiom is to supply an array of zero length:

```
Collection<String> cs = ...
String[] sa = cs.toArray(String[]::new);
```

which, in `Collection`'s default implementation of ❸, results in a call to ❷ with an argument of length 0:

```
Collection<String> cs = ...
String[] sa = cs.toArray(new String[0]);
```

If ❷ or ❸ is being called repeatedly in this way, a more efficient alternative is to declare a single empty array of the required type, that can then be used as many times as required:

```
private static final String[] EMPTY_STRING_ARRAY = new String[0];
Collection<String> cs = ...
String[] sa = cs.toArray(EMPTY_STRING_ARRAY);
```

Why is *any* type allowed for `T` in the declaration of ❷ and ❸, rather than restricting the type to `E`, the parametric type of the collection? One reason is to allow the possibility of giving the array a more specific component type than that of the collection, when the elements of the collection all happen to belong to the same subtype:

```
List<Object> l = List.of("zero","one");
String[] a = l.toArray(new String[0]);
```

Here, a list of objects happens to contain only strings, so it can be converted into a `String[]`, in an operation analogous to the `promote` method described in [Link to Come]. If the list contains an object that is not a string, the error is

caught at run time rather than compile time:

```
List<Object> l = List.of("zero","one",2);
String[] a = l.toArray(new String[0]);      // throws
ArrayStoreException (see §2.5)
```

In general, you may want to copy a collection of a given type into an array of a more specific type (for instance, copying a list of objects into an array of strings, as just shown), or of a more general type (for instance, copying a list of strings into an array of objects). You would never want to copy a collection of a given type into an array of a completely unrelated type (for instance, copying a list of integers into an array of strings is always wrong). But since there's no way to specify this constraint in Java, such errors are caught at run time rather than compile time.

One drawback of this design is that, applied to collections of wrapper types, it doesn't accommodate automatic unboxing into the corresponding array of primitives:

```
List<Integer> l = List.of(0,1,2);
int[] a = l.toArray(new int[0]);  // compile-time error
```

This is illegal because the parameter `T` in the method call must—as for any type parameter—be a reference type. The call would work if we replaced both occurrences of `int` with `Integer`, but often this won't do because, for performance or compatibility reasons, we require an array of primitive type. In such cases, there is nothing for it but to copy the array explicitly:

```
List<Integer> l = List.of(0,1,2);
int[] a = new int[l.size()];
for (int i=0; i<l.size(); i++) a[i] = l.get(i);
```

The Stream API provides a neater alternative:

```
l.stream().mapToInt(Integer::intValue).toArray()
```

Using the Methods of Collection

Let's construct a small example to illustrate the use of the collection classes. Your authors are forever trying to get organized; we'll imagine that our latest effort involves writing our own to-do manager. In this example, we'll see how to write code in the classical Collections Framework idiom, and also, in sidebars, the more modern functional style of the Stream API.

We begin by defining an interface to represent tasks, and records to represent two different kinds of tasks: writing code and making phone calls.

This is the interface that defines a task:

```
public interface Task extends Comparable<Task> {
    default int compareTo(Task t) {
        return toString().compareTo(t.toString());
    }
}
```

and the two concrete `Task` types are

```
record CodingTask(String spec) implements Task {}
record PhoneTask(String name, String number) implements Task {}
```

Tasks will need to be comparable to be used in ordered collections (such as `OrderedSet` and `OrderedMap`); the other methods that they will require —`equals`, `hashCode`, and `toString`—are automatically supplied by the record implementation. The natural ordering on tasks (see "Comparable<T>") corresponds to the ordering on their string representations and, as records, equality of two tasks is defined by the equality of their string-valued fields. Since the natural ordering on strings is consistent with equality—that is, `compareTo` returns `0` exactly when `equals` returns `true`—it follows that for tasks also, the natural order is consistent with equality.

A coding task is specified by its name, and a phone task is specified by the name and number of the person to be called. As records whose string components are all immutable, objects of these types are themselves immutable (see ??).

The string representation of a record returned by the default `toString` method always begins with the name of the record type. Since "`CodingTask`" precedes

"`PhoneTask`" in the alphabetic ordering on strings, and since tasks are ordered according to the results returned by `toString`, it follows that coding tasks come before phone tasks in the natural ordering—highly appropriate for people who much prefer coding to talking on the phone!

We also define an empty task:

```
record EmptyTask() implements Task {
    public String toString() {
        return "";
    }
}
```

Since the empty string precedes all others in the natural ordering on strings, the empty task comes before all others in the natural ordering on tasks. This task will be useful when we construct range views of sorted sets (see Getting Range Views).

```
PhoneTask mikePhone = new PhoneTask("Mike", "987 6543");
PhoneTask paulPhone = new PhoneTask("Paul", "123 4567");
CodingTask databaseCode = new CodingTask("db");
CodingTask guiCode = new CodingTask("gui");
CodingTask logicCode = new CodingTask("logic");

Collection<PhoneTask> phoneTasks = new HashSet<>();
Collection<CodingTask> codingTasks = new HashSet<>();
Collection<Task> mondayTasks = new HashSet<>();
Collection<Task> tuesdayTasks = new HashSet<>();

Collections.addAll(phoneTasks, mikePhone, paulPhone);
Collections.addAll(codingTasks, databaseCode, guiCode, logicCode);
Collections.addAll(mondayTasks, logicCode, mikePhone);
Collections.addAll(tuesdayTasks, databaseCode, guiCode, paulPhone);
```

Using the Stream API, the two blocks of code above can be condensed into a single block:

```
var phoneTasks = Stream.of(mikePhone,paulPhone).collect(toSet());
var codingTasks =
Stream.of(databaseCode,guiCode,logicCode).collect(toSet());
var mondayTasks = Stream.of(logicCode,mikePhone).collect(toSet());
var tuesdayTasks =
```

```
  Stream.of(databaseCode,guiCode,paulPhone).collect(toSet());
```

```
assert phoneTasks.equals(Set.of(mikePhone, paulPhone));
assert codingTasks.equals(Set.of(databaseCode, guiCode, logicCode));
assert mondayTasks.equals(Set.of(logicCode, mikePhone));
assert tuesdayTasks.equals(Set.of(databaseCode, guiCode, paulPhone));
```

??? shows how we can define a series of tasks to be carried out. (In a real system, of course, tasks would be most likely held in a database and retrieved from there.) As part of the retrieval process, the tasks have been organized into various categories—phone tasks, coding tasks, and so on, represented by sets, using the method `Collections.addAll` introduced in [Link to Come]. The choice of sets as the specific collection type isn't altogether arbitrary: although any collection type would have served the purpose, `Set` is unique in the Framework in having exactly the same methods as the top-level `Collection` interface that it inherits from. Now we can use the methods of `Collection` to work with these categories. The examples that follow use the methods in the order in which they were presented earlier.

Adding Elements: We can add new tasks to the schedule:

```
PhoneTask ruthPhone = new PhoneTask("Ruth", "567 1234");
mondayTasks.add(ruthPhone);
assert mondayTasks.equals(Set.of(logicCode, mikePhone, ruthPhone));
```

or we can combine schedules together:

```
Collection<Task> allTasks = new HashSet<>(mondayTasks);
allTasks.addAll(tuesdayTasks);
assert allTasks.equals(Set.of(logicCode, mikePhone, ruthPhone,
        databaseCode, guiCode, paulPhone));
```

```
  var allTasks = Stream.of(mondayTasks,tuesdayTasks)
         .flatMap(Collection::stream)
         .collect(toSet());
```

Removing Elements: When a task is completed, we can remove it from a schedule:

```
boolean wasPresent = mondayTasks.remove(mikePhone);
assert wasPresent;
assert mondayTasks.equals(Set.of(logicCode, ruthPhone))
```

and we can clear a schedule out altogether because all of its tasks have been done (a method that, in reality, we don't get to use very often):

```
mondayTasks.clear();
assert mondayTasks.equals(Collections.EMPTY_SET);
```

The removal methods also allow us to combine entire collections in various ways. For example, to see which tasks other than phone calls are scheduled for Tuesday, we can write:

```
Collection<Task> tuesdayNonphoneTasks = new HashSet<>(tuesdayTasks);
tuesdayNonphoneTasks.removeAll(phoneTasks);
assert tuesdayNonphoneTasks.equals(Set.of(databaseCode, guiCode));
```

```
var tuesdayNonPhoneTasks = tuesdayTasks.stream()
   .filter(t -> ! phoneTasks.contains(t));
```

or to see which phone calls are scheduled for that day:

```
Collection<Task> phoneTuesdayTasks = new HashSet<>(tuesdayTasks);
phoneTuesdayTasks.retainAll(phoneTasks);
assert phoneTuesdayTasks.equals(Set.of(paulPhone));
```

```
var phoneTuesdayTasks = tuesdayTasks.stream()
       .filter(phoneTasks::contains).collect(toSet());
```

This last example can be approached differently to achieve the same result:

```
Collection<PhoneTask> tuesdayPhoneTasks = new HashSet<>(phoneTasks);
tuesdayPhoneTasks.retainAll(tuesdayTasks);
assert tuesdayPhoneTasks.equals(Set.of(paulPhone));
```

```
var tuesdayPhoneTasks = phoneTasks.stream()
   .filter(tuesdayTasks::contains).collect(toSet());
```

Note that `phoneTuesdayTasks` has the type `Collection<Task>`, while `tuesdayPhoneTasks` has the more precise type `Collection<PhoneTask>`.

This last example provides an explanation of the signatures of methods in this group and the next. We have already discussed ("Wildcards Versus Type Parameters") why they take arguments of type `Object` or `Collection<?>` whereas the methods for adding to the collection restrict their arguments to its parametric type. Taking the example of `retainAll`, its contract requires the removal of the elements of this collection which do not occur in the argument collection. That doesn't justify any restriction on what the argument collection may contain; in the example just given it can contain instances of any kind of `Task`, not just `PhoneTask`. There is no reason even to restrict the argument in this way to collections of supertypes of the parametric type; we actually want the least restrictive type possible, which is `Collection<?>`. Similar reasoning applies to `remove`, `removeAll`, `contains`, and `containsAll`.

Querying the Contents of a Collection: These methods allow us to check, for example, that the operations above have worked correctly. We'll use `assert` here to make the system check our belief that we have programmed the previous operations correctly. For example the first statement will fail, throwing an `AssertionError`, if `tuesdayPhoneTasks` does not contain `paulPhone`:

```
assert tuesdayPhoneTasks.contains(paulPhone);
assert tuesdayTasks.containsAll(tuesdayPhoneTasks);
assert mondayTasks.isEmpty();
assert mondayTasks.size() == 0;
```

Making the Collection Contents Available for Further Processing: The methods in this group provide an iterator over the collection, deliver its contents into a stream, or convert it to an array.

"Iterable and Iterators" showed how most uses of iterators can be replaced by the simpler *foreach* statement, which uses them implicitly. But there are uses of iteration with which *foreach* can't help: you have to use an explicit iterator if you want to change the structure of a collection without encountering `ConcurrentModificationException`, or if you want to process two lists in parallel. For example, suppose that we decide that we don't have time for phone tasks on Tuesday. It may perhaps be tempting to use *foreach* to filter them from our task list, but that won't work for the reasons described in "Iterable and Iterators":

```
// throws ConcurrentModificationException
for (Task t : tuesdayTasks) {
  if (t instanceof PhoneTask) {
    tuesdayTasks.remove(t);
  }
}
```

Using an iterator explicitly is no improvement if you still use `Collection` methods to modify the structure:

```
// throws ConcurrentModificationException
for (Iterator<Task> it = tuesdayTasks.iterator() ; it.hasNext() ; ) {
  Task t = it.next();
  if (t instanceof PhoneTask) {
    tuesdayTasks.remove(t);
  }
}
```

But using the *iterator's* structure-changing method gives the result we want:

```
for (Iterator<Task> it = tuesdayTasks.iterator() ; it.hasNext() ; ) {
  Task t = it.next();
  if (t instanceof PhoneTask) {
    it.remove();
  }
}
```

Compared to this inelegant code, the appeal of the Stream API is obvious:

```
var tuesdayCodeTasks = tuesdayTasks.stream()
        .filter(t -> !(t instanceof PhoneTask))
        .collect(Collectors.toSet());
```

The stream version has the advantages of clarity, brevity, and the possibility of efficiently parallelizing this operation, simply by changing the call of `stream` to `parallelStream`. The performance cost of these advantages is the extra memory required to create the new collection.

A more elaborate example of the use of iterators will be shown in another chapter, although there again we'll see that the Stream API offers a more concise and readable alternative.

Implementing Collection

There are no concrete implementations of `Collection`. The class `AbstractCollection`, which partially implements it, is one of a series of skeletal implementations—including `AbstractSet`, `AbstractList`, and so on—which provide functionality common to the different concrete implementations of each interface. These skeletal implementations are available to help the designer of new implementations of the Framework interfaces. For example, `Collection` could serve as the interface for *bags* (unordered lists), and a programmer implementing bags could extend `AbstractCollection` and find most of the implementation work already done.

Collection Constructors

We will go on to look at the three main kinds of collection in the next three chapters, but we should first explain two common forms of constructor which are shared by most collection implementations. Taking `HashSet` as an example, these are:

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

The first of these creates an empty set, and the second a set that will contain the elements of any collection of the parametric type—or one of its subtypes, of course. Using this constructor has the same effect as creating an empty set with the default constructor, and then adding the contents of a collection using `addAll`. This is sometimes called a “copy constructor”, but that term should really be reserved for constructors which make a copy of an object of the same *class*, whereas constructors of the second form can take any object which implements the *interface* `Collection<? extends E>`. Joshua Bloch has suggested the term “conversion constructor”.

Not all collection classes have constructors of both forms —`ArrayBlockingQueue`, for example, cannot be created without fixing its capacity, and `SynchronousQueue` cannot hold any elements at all, so no constructor of the second form is appropriate. In addition, many collection classes have other constructors besides these two, but which ones they have depends not on the interface they implement but on the underlying implementation; these additional constructors are used to configure the implementation.

Chapter 7. The SequencedCollection Interface

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 13th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

The interface `SequencedCollection` occupies a unique place in the design space of the Framework. Rather than specifying the contract for a particular data structure, it unifies behaviors exposed by a variety of interfaces and one implementation—`List`, `NavigableSet`, `Deque`, and `LinkedHashSet`—that are spread across the type hierarchy and mostly not otherwise related. What these collections have in common is that all represent a sequence of elements whose order is semantically significant (sometimes called the *encounter order*), whether the order is internally or externally imposed (see "Sequenced Collections").

It was `Deque` (see "Deques"), a relatively late addition to the Collections Framework, that introduced the set of methods that has now been generalised to all collections with an encounter order, providing the ability to add, inspect, and remove the first and last elements of the sequence. Some of these capabilities were previously available, unevenly and under different names, in these collections: for example, `NavigableSet` supported the removal of the first and last elements but not their inspection, while `List` directly supported only addition of a last element, all other operations requiring a sequence position to

be specified (see [Link to Come]).

It would be impossible to unify the behavior of so many disparate collections without some special cases. For example, it makes no sense to add a first or last element to a `NavigableSet`, given that the position of all the elements in these collections is determined by the internal sorting algorithm. For another example, unmodifiable lists cannot support structural modifications such as addition and removal of elements. These special cases will be dealt with in the chapters covering the different collection types.

The single new method provided by `SequencedCollection` is `reversed`, which provides a reverse-ordered view (see ??) of the original collection. This makes the ability to process an ordered collection in reverse (previously only available to clients of `NavigableSet` via its `descendingSet` method) available to users of any collection with encounter order.

Adding Elements

```
void addFirst(E e)      // adds e as the first element of this
collection
void addLast(E e)       // adds e as the last element of this
collection
```

The contracts for these methods specify that after their successful execution, the collection must contain the supplied element as the first (respectively last) element in the encounter order.

Inspecting Elements

```
E getFirst()            // returns the first element of this
collection
E getLast()             // returns the last element of this collection
```

Removing Elements

```
E removeFirst()         // removes and returns the first element of
this collection
E removeLast()          // removes and returns the last element of
this collection
```

Calling an inspection or a removal method on an empty collection will normally result in a `NoSuchElementException`.

View-Generating Method

```
SequencedCollection<E> reversed()
```

Returns a reverse-ordered view of this collection. Of course, you would prefer that calling `reversed` on, for example, a `List` (an interface that inherits from `SequencedCollection`) would return a `List`, rather than a generic `SequencedCollection`, as the declaration above implies. For this reason, the four interfaces that inherit from `SequencedCollection`—`List`, `SequencedSet`, `NavigableSet`, and `Deque`—all define covariant overrides for reversed. So, for example, `SequencedSet`'s declaration of `reversed` is

```
SequencedSet<E> reversed()
```

Similar overrides are declared by the other three interfaces.

The reverse ordering affects iteration and other order-sensitive operations, including those on views of the returned view. The contract states that any successful modifications to this view must write through to the underlying collection, but that the inverse—visibility in this view of changes to the underlying collection—are implementation-dependent.

Chapter 8. Sets

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 14th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

A *set* is a collection of items that cannot contain duplicates; adding an item that is already present in the set has no effect. The `Set<E>` interface has the same methods as those of `Collection`, but it is defined separately in order to allow the contract of `add` (and `addAll`, which is defined in terms of `add`) to be changed in this way. For example, returning to the task manager, suppose that on Monday you have free time to carry out your telephone tasks. You can make the new collection of tasks for Monday by adding all your telephone tasks to the existing Monday tasks. Let `mondayTasks` and `phoneTasks` be as declared in ???. Using a set, you can write:

```
Set<Task> phoneAndMondayTasks = new HashSet<>(mondayTasks);
phoneAndMondayTasks.addAll(phoneTasks);
assert phoneAndMondayTasks.equals(Set.of(logicCode, mikePhone,
paulPhone));
```

This works because of the way that duplicate elements are handled. The task `mikePhone`, although it is in both `mondayTasks` and `phoneTasks`, appears only once in `phoneAndMondayTasks`—appropriately for this application: you definitely don't want to have to start doing tasks twice over!

This seems straightforward, but to understand exactly how different `Set`

implementations work, we need to look more closely at what defines a duplicate—that is, the *equivalence relation* for that set. The example above uses `HashSet`, for which the equivalence relation is the `equals` method: in other words, for a `HashSet` two objects are duplicates if and only if the `equals` method, called on one with the other as its argument, returns `true`. This might seem an obvious choice, but it's not the only one—although the Javadoc (at JDK 20) for `Set` and its implementations often implies that it is. Some other implementations also use `equals`, like `CopyOnWriteArraySet` and the family of classes that we're calling `UnmodifiableSet` (see "UnmodifiableSet"). But another choice, if less common, is the identity relation; a set using that relation will contain a reference to every unique object that has been added to it. One such example is `EnumSet` (see "EnumSet")—although since `Enum`s are singletons, `EnumSet` could also be said to use `equals`. Another is any set created from an `IdentityHashMap` (see "IdentityHashMap") using the `Collections` method `newSetFromMap`.

A third alternative for an equivalence relation is specified by the contract for `NavigableSet`. A `NavigableSet` (see "NavigableSet") maintains its elements in sorted order using an ordering relation provided by either its natural order or a `Comparator` (see Chapter 2). `NavigableSet` also uses that ordering relation in another way: it defines two objects as equivalent if, using it, they compare as equal, regardless of whether they satisfy the equality relation.

The fact that different sets use different equivalence relations creates no difficulties so long as the different relations are compatible (see Chapter 2)—that is, they give the same result applied to every value pair. Confusion arises, however, with incompatible relations; for example, two objects may not satisfy the equality relation even though the set's ordering relation compares them as equal. This confusion is unfortunately compounded by the Javadoc for `Set` and its implementations, which in places fails to recognise that some implementations use equivalence relations other than `equals`. Hopefully, future Javadoc versions may correct that problem.

You can avoid the confusion by understanding the consequences of using different equivalence relations for different sets. One consequence, obvious once you understand the problem, is that sets may contain duplicate elements satisfying `equals` or, conversely, that they may elide occurrences of ones that

don’t. A less obvious consequence concerns equality between sets: to determine whether two sets `A` and `B` are equal, `A` must test each member of `B` to discover whether it is equivalent to a member of `A`. If the roles are reversed, and if `A` and `B` are using different equivalence relations, the results may be different, so set equality loses symmetry.

Set Implementations

When we used the methods of `Collection` in the examples of Chapter 6, we emphasized that they would work with any implementation of `Collection`, not only `Set`, and, even staying with `Set`, not only the `Set` implementation `HashSet`. But in practice, of course, you have to decide on a concrete implementation. This Chapter surveys the different `Set` implementations provided by the Framework, which differ both in how fast they perform the basic operations of `add`, `contains`, and iteration, and in the order in which their iterators return their elements. At the end of the Chapter, we’ll summarize the comparative performance of the different implementations.

Of the `Set` implementations in the Collections Framework, four directly implement `Set` itself; the others inherit from the subinterface `SequencedSet` (see “SequencedSet”. In this section, we will look at these four: `HashSet`, `CopyOnWriteArraySet`, `EnumSet`, and the family of implementations that we’re calling `UnmodifiableSet`.

HashSet

This class is the most commonly used implementation of `Set`. As the name implies, it is implemented by a *hash table*, an array in which elements are stored at a position derived from their contents. Since hash tables store and retrieve elements by their content, they are well suited to implementing the operations of `Set` (the Collections Framework also uses them for various implementations of `Map`). For example, to implement `contains(Object o)` you would look for the element `o` and return `true` if it were found.

An element’s position in a hash table is calculated by a *hash function* of its contents. Hash functions are designed to give, as far as possible, an even spread

of results (*hash codes*) from the element values that might be stored. For an example, here is code like that used in the `String` class to calculate a hash code:

```
int hash = 0;
for (char ch : str.toCharArray()) {
  hash = hash * 31 + ch;
}
```

Traditionally, hash tables obtain a table index from the hash code by taking the remainder after division by the table length. Division is a slow operation in practice, so the Collections Framework classes use bit masking instead. Since that means it is the pattern of bits at the low end of the hash code that is significant, prime numbers (such as 31, here) are used in calculating the hash code, because multiplying by primes will not tend to shift information away from the low end, as would multiplying by a power of two, for example.

Clearly, unless your table has more locations than there are values that might be stored in it, sometimes two distinct values will hash to the same location in the hash table (this is called a *collision*). For instance, no `int`-indexed table can be large enough to store all string values without collisions.We can minimize the problem with a good hash function—one that spreads the elements out equally in the table—but, when collisions do occur, we have to have a way of keeping the colliding elements at the same table location or *bucket*. This is often done by storing them in a linked structure—a list or a tree—as shown in Figure 8-1. We will look at linked structures in more detail later, but for now it’s enough to see that elements stored at the same bucket can still be accessed, at the cost of following a chain of cell references. Figure 8-1 shows the situation resulting from running this code on the OpenJDK implementation of Java 20:

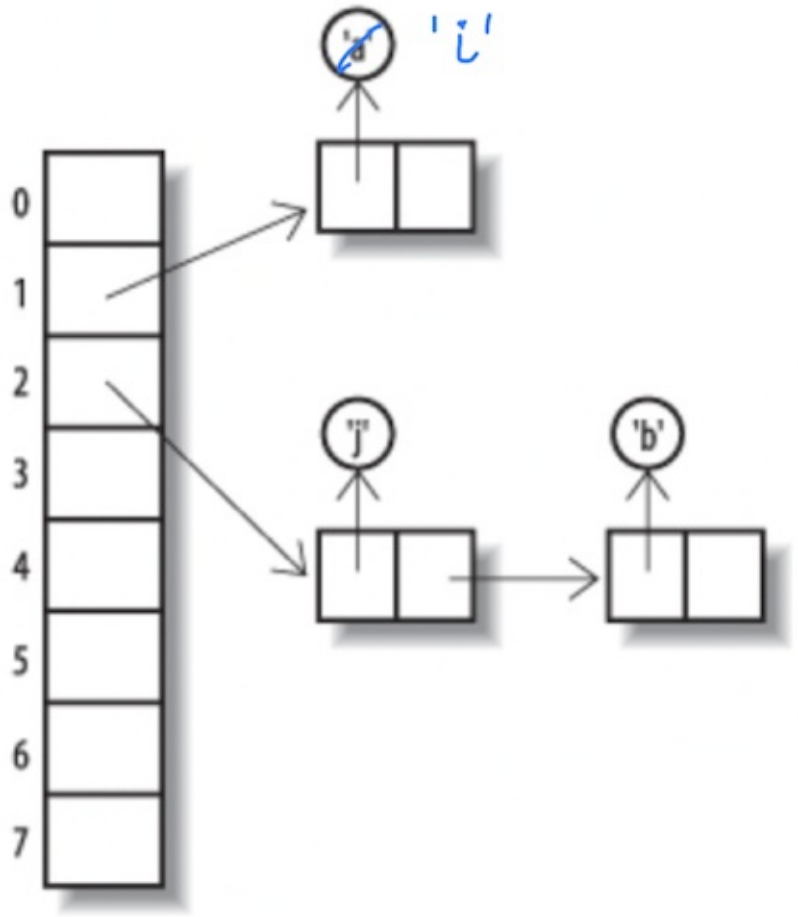

Figure 8-1. A hash table with chained overflow

```
Set<Character> s1 = new HashSet<Character>(8);
s1.add('j');
s1.add('i');
s1.add('b');
```

The index values of the table elements have been calculated by using the bottom three bits (for a table of length 8) of the hash code of each element. In this implementation, a `Character`'s hash code is just the Unicode value of the character it contains. (In practice, a hash table would be much bigger than this. Also, this diagram is simplified from the real situation; because `HashSet` is actually implemented by a specialized `HashMap` (see "HashMap"), each of the cells in the chain contains not one but two references, to a key and a value (see Chapter 11). Only the key is shown in this diagram because, when a hash table is

being used to represent a set, all values are the same—only the presence of the key is significant.)

As long as there are no collisions, the cost of inserting or retrieving an element is constant. As the hash table fills, collisions become more likely; assuming a good hash function, the probability of a collision in a lightly loaded table is proportional to its *load*, defined as the number of elements in the table divided by its capacity (the number of buckets). If a collision does take place, an overflow structure—a linked list or tree—has to be created and subsequently traversed, adding an extra cost to insertion. If the size of the hash table is fixed, performance will worsen as more elements are added and the load increases. To prevent this from happening, the table size is increased by *rehashing*—copying to a new and larger table—when the load reaches a specified threshold (its *load factor*).

Iterating over a hash table requires each bucket to be examined to see whether it is occupied and therefore costs a time proportional to the capacity of the hash table plus the number of elements it contains. Since the iterator examines each bucket in turn, the order in which elements are returned depends on their hash codes, so there is no guarantee as to the order in which the elements will be returned. In the OpenJDK implementation of Java 20, the hash table shown in Figure 8-1 yields its elements in order of ascending table index and forward traversal of the linked lists. Printing it produces the following output.

```
[i, j, b]
```

Later in this section we will look at `LinkedHashSet`, a variant of this implementation with an iterator that does return elements in their insertion order.

The chief attraction of a hash table implementation for sets is the constant-time performance (for lightly-loaded tables) of the basic operations of `add`, `remove`, `contains`, and `size`. Its main performance disadvantages are the poor performance of heavily-loaded tables, and the iteration performance: iterating through the table involves examining every bucket, so the cost includes a factor attributable to the table length, regardless of the size of the set it contains.

`HashSet` has the standard constructors that we introduced in "Collection Constructors", together with two additional constructors:

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

Both of these constructors create an empty set but allow some control over the size of the underlying table, creating one with a length of the next-largest power of 2 after the supplied capacity and, optionally, with the desired load factor. Most of the hash-based maps in the Collections Framework have similar constructors. You can use these constructors to create a table large enough to store all the elements you expect it to hold without requiring expensive resizing operations. In practice, however, they have proved confusing and difficult to use: confusing, because their `int` parameter, often misunderstood to be the expected number of entries, is in fact used to compute the table size, and difficult to use because computing the argument correctly from the expected maximum number of entries is implementation-dependent and error-prone. So Java 19 added static factory methods, which take only a parameter signifying the expected maximum number of entries. These methods are recommended as being easier to use and less subject to implementation changes. The factory method for `HashSet` is `newHashSet`:

```
static <T> HashSet<T> newHashSet(int numElements)
```

`HashSet` is unsychronized and not thread-safe; its iterators are fail-fast.

CopyOnWriteArraySet

In functional terms, `CopyOnWriteArraySet` is another straightforward implementation of the `Set` contract, but with quite different performance characteristics from `HashSet`. This class is implemented as a thin wrapper around an instance of `CopyOnWriteArrayList`, which in turn is backed by an array. The array is treated as immutable; any modification of the set results in the creation of an entirely new array. So `add` has complexity $O(n)$, as does `contains`, which has to be implemented by a linear search. Clearly you wouldn't use `CopyOnWriteArraySet` in a context where you were expecting many searches or insertions. But the array implementation means that iteration costs $O(1)$ per element—faster than `HashSet`—and it has one advantage which is really compelling in some applications: it provides thread

safety (see “Collections and Thread Safety”) without adding to the cost of read operations. This is in contrast to those collections which use locking to achieve thread safety for all operations (for example, the synchronized collections of “Synchronized Collections”). Locking operations are always a potential bottleneck in multi-threaded applications. By contrast, read operations on copy-on-write collections are implemented on the backing array, and thanks to its immutability they can be used by any thread without danger of interference from a concurrent write operation.

When would you want to use a set with these characteristics? In fairly specialized cases; one that is quite common is in the implementation of the Subject-Observer design pattern (see [Link to Come]), which requires events to be notified to a set of observers. This set may not be modified during the process of notification; with locking set implementations, read and write operations share the overhead necessary to ensure this, whereas with `CopyOnWriteArraySet` the overhead is carried entirely by write operations. This makes sense for Subject-Observer: in typical uses of this pattern, event notifications occur much more frequently than changes to the listener set.

Iterators for `CopyOnWriteArraySet` can be used only to read the set. When they are created, they are attached to the instance of the backing array being used by the set at that moment. Since no instance of this array is ever modified, the iterators’ `remove` method is not implemented. These are snapshot iterators (see “Collections and Thread Safety”); they reflect the state of the set at the time they were created, and can subsequently be traversed without any danger of interference from threads modifying the set from which they were derived.

Since there are no configuration parameters for `CopyOnWriteArraySet`, the constructors are just the standard ones discussed in “Collection Constructors”.

EnumSet

This class exists to take advantage of the efficient implementations that are possible when the number of possible elements is fixed and a unique index can be assigned to each. These two conditions hold for a set of elements of the same `Enum`; the number of keys is fixed by the constants of the enumerated type, and the `ordinal` method returns values that are guaranteed to be unique to each

constant. In addition, the values that `ordinal` returns form a compact range, starting from zero—ideal, in fact, for use as array indices or, in the standard implementation, indices of a bit vector. So `add`, `remove`, and `contains` are implemented as bit manipulations, with constant-time performance. Bit manipulation on a single word is extremely fast, and a `long` value can be used to represent `EnumSets` over `enum` types with up to 64 values. Larger `enums` can be treated in a similar way, with some overhead, using more than one word for the representation.

`EnumSet` is an abstract class that implements these different representations by means of different package-private subclasses. It hides the concrete implementation from the programmer, instead exposing factory methods that call the constructor for the appropriate subclass. The following group of static factory methods provide ways of creating `EnumSets` with different initial contents: empty, specified elements only, or all elements of the `enum`.

```
<E extends Enum<E>> EnumSet<E> of(E first, E... rest)
        // create a set initially containing the specified elements
<E extends Enum<E>> EnumSet<E> range(E from, E to)
        // create a set initially containing all of the elements in
        // the range defined by the two specified endpoints
<E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType)
        // create a set initially containing all elements in
elementType
<E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType)
        // create a set of elementType, initially empty
```

An `EnumSet` contains the reified type of its elements, which is used at run time for checking the validity of new entries. This type is supplied by the above factory methods in two different ways. The methods `of` and `range` receive at least one `enum` argument, which can be queried for its declaring class (that is, the `Enum` that it belongs to). For `allOf` and `noneOf`, which have no `enum` arguments, a class token is supplied instead.

Common cases for `EnumSet` creation are optimized by the second group of methods, which allow you to efficiently create sets with one, two, three, four, or five elements of an enumerated type.

```
<E extends Enum<E>> EnumSet<E> of(E e)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2)
```

```
<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)
```

The third set of methods allows the creation of an `EnumSet` from an existing collection:

```
<E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)
      // create an EnumSet with the same element type as s, and
      // with the same elements
<E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)
      // create an EnumSet from the elements of c, which must contain
      // at least one element
<E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> s)
      // create an EnumSet with the same element type as s,
      // containing the elements not in s
```

The collection supplied as the argument to the second version of `copyOf` must be nonempty so that the element type can be determined.

In use, `EnumSet` obeys the contract for `Set`, with the added requirement that its iterators will return their elements in their natural order (the order in which their `enum` constants are declared). It is not thread-safe, but unlike the unsynchronized general-purpose collections, its iterators are not fail-fast. They may be either snapshot or weakly consistent; to be conservative, the contract guarantees only that they will be weakly consistent (see “Collections and Thread Safety”).

UnmodifiableSet

You won’t find any reference to the implementation(s) `UnmodifiableSet<E>` of the `Set` interface in the Javadoc or in the code of the Collections Framework: it’s a name invented in this book for a family of package-private classes that client programmers can never access by name, but that are important because they provide the implementation of the unmodifiable sets obtained from the various overloads of the factory methods `Set.of` and `Set.copyOf`. These methods (which are static, like all factory methods—the keyword is omitted below for brevity) can be divided into three groups analogous to those of `EnumSet`. The first contains just an overload of the

method `of` that takes no arguments and returns an empty unmodifiable set:

```
<E> Set<E> of() // Returns an unmodifiable list containing zero
elements
```

The second group allows you to create an unmodifiable set from up to ten specified elements:

```
<E> Set<E> of(E e1) // Returns an unmodifiable set containing one
element
<E> Set<E> of(E e1, E e2) // Returns an unmodifiable set containing
two elements
<E> Set<E> of(E e1, E e2, E e3) // Returns an unmodifiable set
containing three
                                        // elements
<E> Set<E> of(E e1, E e2, E e3, E e4) // Returns an unmodifiable set
containing four
                                        // elements
<E> Set<E> of(E e1, E e2, E e3, E e4, E e5) // Returns an unmodifiable
set containing
                                        // five elements
<E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6) // Returns an
unmodifiable set
                                        // containing six elements
<E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7) // Returns an
unmodifiable
                                        // set containing seven
elements
<E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8) //
Returns an
                                        // unmodifiable set
containing eight elements
<E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9) //
Returns an
                                        // unmodifiable set
containing nine elements
<E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E
e10) // Returns an
                                        // unmodifiable set
containing ten elements
```

The third group allows you to create an unmodifiable set from a collection, an array, or a varargs-supplied list of arguments:

```
<E> Set<E> copyOf(Collection<? extends E> coll)
                                        // Returns an unmodifiable
```

```
set containing
                                                    // an arbitrary number of
elements
<E> Set<E> of(E... elements) // Returns an unmodifiable set containing
an arbitrary
                                                    // number of elements
```

The classes that make up `UnmodifiableSet` take advantage of its unmodifiability to use fixed-length arrays as the backing structures. Without the overhead of empty table buckets or linked overflow structures, these implementations require much less space than a hashed structure. Iteration is also correspondingly more efficient, with the added benefit of improved spatial locality (see “Memory”). The tradeoff for faster iteration is that containment can only be determined by a linear search, *O(n)* in complexity.

In the OpenJDK implementation, the iterators for `UnmodifiableSet` have an unusual characteristic: the order of the iteration is randomly determined for each virtual machine instance. The reason for this design is to avoid replicating an odd sort of compatibility problem with `HashSet`: in the past, developers have based tests and even production code on the apparently fixed iteration order of `HashSet`. But since the contract for `HashSet` doesn’t specify the iteration order, implementers have changed it between versions, resulting in code breaking on upgrades. The iterators for unmodifiable sets have been designed to prevent developers being lulled into this error.

SequencedSet

A `SequencedSet` is an externally or internally ordered `Set` that also exposes the methods of `SequencedCollection`. It combines the methods of these two interfaces, adding to them only in one respect: it provides a covariant override of the method `reversed` of `SequencedCollection` in order to return a value of type `SequencedSet` (see ???). It has a single direct implementation, `LinkedHashSet`, and is extended by the interface `NavigableSet` (actually by its parent interface `SortedSet`, but see the introductory remarks in “NavigableSet”).

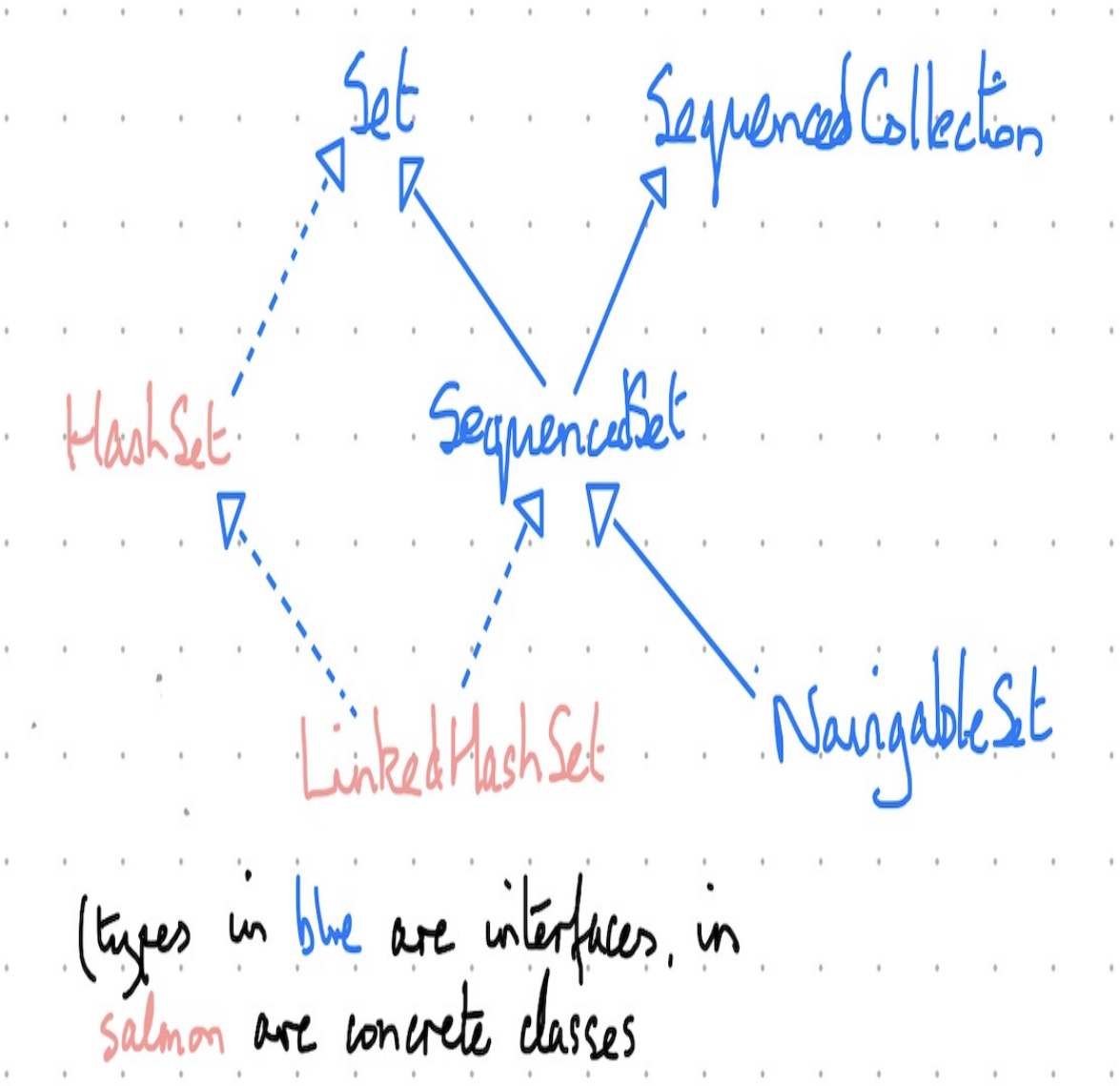

Figure 8-2. SequencedSet and Related Types

LinkedHashSet

This class inherits from `HashSet` and, further, implements `SequencedSet` by maintaining a linked list of its elements, as shown by the curved arrows in Figure 8-3. The situation in that figure would result from this code:

```
var lhs = LinkedHashSet.<Character>newLinkedHashSet(3);
Collections.addAll(lhs, 'j', 'i', 'b');
```

```
// iterators of a LinkedHashSet return their elements in insertion
order:
assert lhs.toString().equals("[j, i, b]");

// the methods of SequencedSet are all available:
assert lhs.getLast() == 'b';
assert lhs.reversed().toString().equals("[b, i, j]");
```

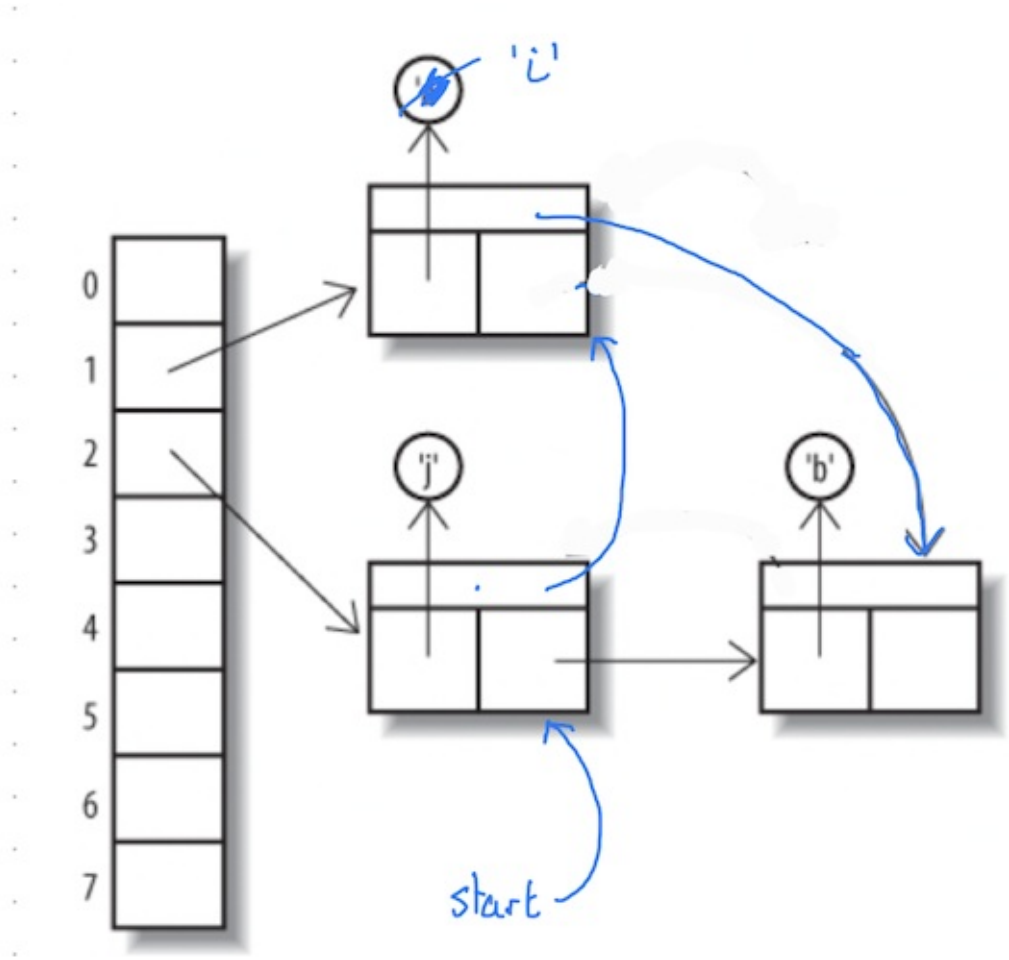

Figure 8-3. A linked hash table

The linked structure also has a useful consequence in terms of improved performance for iteration: `next` executes in constant time, as the linked list can be used to visit each element in turn. This is in contrast to `HashSet`, for which every bucket in the hash table must be visited whether it is occupied or not. In the case of a large table that is sparsely occupied that would be a significant inefficiency, but in general the overhead involved in maintaining the linked list means that you would choose `LinkedHashSet` in preference to `HashSet`

only if that situation was likely to apply in your application.

The constructors for `LinkedHashSet` provide the same facilities as those of `HashSet` for configuring the underlying hash table. And in the same way as `HashSet`, it has a modern factory method `newLinkedHashSet` that accepts as its argument the expected number of elements and sizes the table optimally.

This class is unsychronized and not thread-safe; its iterators are fail-fast.

NavigableSet

The interface `NavigableSet` adds to the `SequencedSet` contract a guarantee that its iterator will traverse the set in ascending element order, and adds further methods to find the elements adjacent to a target value. Prior to Java 6, when `NavigableSet` was introduced, the only subinterface of `Set` was an interface called `SortedSet`, which guarantees iteration order but does not expose the closest-match methods. `SortedSet` is still in the JDK—it extends `SequencedSet` and is in its turn extended by `NavigableSet`—but is no longer of much interest, since it has no direct implementations in the platform.

We can use `NavigableSet` to add some useful functionality to the to-do manager. Until now, the methods of `Collection` and `Set` have provided no help in ordering our tasks—surely one of the central requirements of a to-do manager. Example 8-1 defines a record `PriorityTask` which attaches a priority to a task. There are three priorities, `HIGH`, `MEDIUM`, and `LOW`, declared so that `HIGH` priority comes first (i.e. is lowest) in the natural ordering. To compare two `PriorityTasks`, the comparator `priorityTaskCmpr` first compares their priorities: if the priorities are unequal, the higher priority tasks comes first. If the priorities are equal, it goes on to use the natural ordering on the underlying tasks. So two objects can only compare as equal if they actually are equal—that is, the natural ordering is consistent with equals, and the problems discussed at the beginning of this Chapter don’t arise. Conversely, an example of how they could arise would be if we defined the comparator such that all tasks of equal priority compared as equal. With that definition, a `NavigableSet` could only contain at most one task of each priority.

Example 8-1. The record `PriorityTask`

```
public enum Priority { HIGH, MEDIUM, LOW }
public record PriorityTask(Task task, Priority priority)
    implements Comparable<PriorityTask> {

    static Comparator<PriorityTask> priorityTaskCmpr =
        Comparator.comparing(PriorityTask::priority)
            .thenComparing(PriorityTask::task);

    public int compareTo(PriorityTask pt) {
        return priorityTaskCmpr.compare(this,pt);
    }
}
```

The following code shows `NavigableSet` working with a set of `PriorityTasks`.

```
  NavigableSet<PriorityTask> priorityTasks = new TreeSet<PriorityTask>
  ();

  priorityTasks.add(new PriorityTask(mikePhone, Priority.MEDIUM));
  priorityTasks.add(new PriorityTask(paulPhone, Priority.HIGH));
  priorityTasks.add(new PriorityTask(databaseCode, Priority.MEDIUM));
  priorityTasks.add(new PriorityTask(guiCode, Priority.LOW));

  assert(priorityTasks.toString()).equals("""
      [PriorityTask[task=PhoneTask[name=Paul, number=123 4567],
  priority=HIGH], \
      PriorityTask[task=CodingTask[spec=db], priority=MEDIUM], \
      PriorityTask[task=PhoneTask[name=Mike, number=987 6543],
  priority=MEDIUM], \
      PriorityTask[task=CodingTask[spec=gui], priority=LOW]]""");
```

The methods defined by the `NavigableSet` interface fall into six groups:

Retrieving the Comparator

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

This method returns the set's comparator if it has been given one at construction time, or `null` if the set uses the natural ordering of its elements. The reason that the bounded type can be used for the `Comparator` parameter is because a `NavigableSet` parameterized on `E` can rely for ordering on a `Comparator` defined on any supertype of `E`. For example, recalling "A Fruity Example", a `Comparator<Fruit>` could be used for a `NavigableSet<Apple>`.

Inspecting the First and Last Elements

```
E first() // return the first element in the set
E last()  // return the last element in the set
```

If the set is empty, these operations throw `NoSuchElementException`.

Removing the First and Last Elements

```
E pollFirst() // retrieve and remove the first (lowest) element,
              // or return null if this set is empty
E pollLast()  // retrieve and remove the last (highest) element,
              // or return null if this set is empty
```

These are analogous to the methods of the same name in `Deque` (see "Deques"), and help to support the use of `NavigableSet` in applications which require queue functionality. For example, in the version of the to-do manager in this section, we could get the highest-priority task off the list, ready to be carried out, by means of this:

```
PriorityTask nextTask = priorityTasks.pollFirst();
assert nextTask.toString().equals(
     "PriorityTask[task=PhoneTask[name=Paul, number=123 4567],
priority=HIGH]");
```

Getting Range Views

An interval such as a range view can be *open*, *half-open*, or *closed*, depending on how many of its limit points it contains. For example, the range of numbers x for which $0 \le x \le 1$ is closed, because it contains both limit points 0 and 1. The ranges $0 \le x < 1$ and $0 < x \le 1$ are half-open because they contain only one of the limit points, and the range $0 < x < 1$ is open because it contains neither.

This preliminary explanation is necessary because each of the methods in this group appears in two overloads, one inherited from `SortedSet` and returning a half-open `SortedSet` view, and one defined in `NavigableSet` returning a `NavigableSet` view that can be open, half-open, or closed according to the user's choice.

The `SortedSet` methods return a view of the set elements starting from the

`fromValue`—including it if it is present—and up to but excluding the `toValue`. Notice that although the Javadoc for these methods calls the arguments to these operations "elements"—`fromElement` and `toElement`—this is misleading: in fact they do not themselves have to be members of the set.

```
SortedSet<E> subSet(E fromValue, E toValue)
SortedSet<E> headSet(E toValue)
SortedSet<E> tailSet(E fromValue)
```

The `NavigableSet` methods are similar, but allow you to specify for each bound whether it should be included or excluded.

```
NavigableSet<E> subSet(E fromValue, boolean fromInclusive,
                                    E toValue, boolean toInclusive)
NavigableSet<E> headSet(E toValue, boolean inclusive)
NavigableSet<E> tailSet(E fromValue, boolean inclusive)
```

In our example, these methods could be useful in providing different views of the elements in `priorityTasks`. For instance, we can use `headSet` to obtain a view of the high- and medium-priority tasks. To do this, we need a special task that comes before all others in the task ordering; fortunately, we defined a class `EmptyTask` for just this purpose in "Using the Methods of Collection". Using this, it's easy to extract all tasks that come before any low-priority task:

```
PriorityTask firstLowPriorityTask = new PriorityTask(new EmptyTask(),
Priority.LOW);

NavigableSet<PriorityTask> highAndMediumPriorityTasks =
        priorityTasks.headSet(firstLowPriorityTask, false);

assert(highAndMediumPriorityTasks.toString()).equals("""
    [PriorityTask[task=PhoneTask[name=Paul, number=123 4567],
priority=HIGH], \
    PriorityTask[task=CodingTask[spec=db], priority=MEDIUM], \
    PriorityTask[task=PhoneTask[name=Mike, number=987 6543],
priority=MEDIUM]]""");
```

In fact, because we know that tasks with empty details will never normally occur, we can also use one as the first endpoint in a half-open interval:

```
PriorityTask firstMediumPriorityTask =
  new PriorityTask(new EmptyTask(), Priority.MEDIUM);

NavigableSet<PriorityTask> mediumPriorityTasks =
  priorityTasks.subSet(firstMediumPriorityTask, firstLowPriorityTask,
false);

assert(mediumPriorityTasks.toString()).equals("""
    [PriorityTask[task=CodingTask[spec=db], priority=MEDIUM], \
    PriorityTask[task=PhoneTask[name=Mike, number=987 6543],
priority=MEDIUM]]""");
```

These views are completely “transparent”; all changes in the underlying set are reflected in the view:

```
PriorityTask logicCodeMedium = new PriorityTask(logicCode,
Priority.MEDIUM);
priorityTasks.add(logicCodeMedium);
assert(mediumPriorityTasks.toString()).equals("""
    [PriorityTask[task=CodingTask[spec=db], priority=MEDIUM], \
    PriorityTask[task=CodingTask[spec=logic], priority=MEDIUM], \
    PriorityTask[task=PhoneTask[name=Mike, number=987 6543],
priority=MEDIUM]]""");
```

To understand how this works, think of all the possible values in an ordering as lying on a line, like the number line used in arithmetic. A range is defined as a fixed segment of that line, regardless of which values are actually in the original set. So a subset, defined on a `NavigableSet` and a range, will allow you to work with whichever elements of the `NavigableSet` currently lie within the range.

The converse also applies: changes in the view—including structural changes—are reflected in the underlying set:

```
assert priorityTasks.size() == 5;
mediumPriorityTasks.remove(logicCodeMedium);
assert priorityTasks.size() == 4;
```

Getting Closest Matches

```
E ceiling(E e) // return the least element x in this set such that
               // x≥e, or null if there is no such element
E floor(E e)   // return the greatest element x in this set such that
```

```
                // x≤e, or null if there is no such element
E higher(E e)   // return the least element x in this set such that
                // x>e, or null if there is no such element
E lower(E e)    // return the greatest element in this set such that
                // x<e, or null if there is no such element
```

These methods are useful for short-distance navigation. For example, suppose that we want to find, in a sorted set of strings, the last three strings in the subset that is bounded above by “x-ray”, including that string itself if it is present in the set. `NavigableSet` methods make this easy:

```
        NavigableSet<String> stringSet = new TreeSet<>();
        Collections.addAll(stringSet, "abc", "cde", "x-ray" ,"zed");
        Optional<String> last =
Optional.ofNullable(stringSet.floor("x-ray"));
        assert last.equals(Optional.of("x-ray"));
        Optional<String> secondToLast = last.map(stringSet::lower);
        assert secondToLast.equals(Optional.of("cde"));
        Optional<String> thirdToLast =
secondToLast.map(stringSet::lower);
        assert thirdToLast.equals(Optional.of("abc"));
```

Notice that in line with a general trend in the design of the Collections Framework, `NavigableSet` returns `null` values to signify the absence of elements where, for example, the `first` and `last` methods of `SortedSet` would throw `NoSuchElementException`. For this reason, you should avoid `null` elements in `NavigableSets`, and in fact the newer implementation, `ConcurrentSkipListSet`, does not permit them (though `TreeSet` must continue to do so, for backward compatibility).

Navigating the Set in Reverse Order

```
NavigableSet<E> descendingSet()   // return a reverse-order view of
                                  // the elements in this set
Iterator<E> descendingIterator()  // return a reverse-order iterator
```

Methods of this group make traversing a `NavigableSet` equally easy in the descending (that is, reverse) ordering. As a simple illustration, let’s generalise the example above using the nearest-match methods. Suppose that, instead of finding just the last three strings in the sorted set bounded above by “x-ray”, we want to iterate over all the strings in that set, in descending order:

```
NavigableSet<String> headSet = stringSet.headSet(last, true);
NavigableSet<String> reverseHeadSet = headSet.descendingSet();
assert reverseHeadSet.toString().equals("[x-ray, cde, abc]");
String conc = " ";
for (String s : reverseHeadSet) {
  conc += s + " ";
}
assert conc.equals(" x-ray cde abc ");
```

If the iterative processing involves structural changes to the set, and the implementation being used is `TreeSet` (which has fail-fast iterators), we will have to use an explicit iterator to avoid `ConcurrentModificationException`:

```
for (Iterator<String> itr = headSet.descendingIterator();
itr.hasNext(); ) {
  itr.next(); itr.remove();
}
assert headSet.isempty();
```

Although it has been retrofitted to extend `SequencedSet`, none of the new methods provide any different functionality: they are just renamed versions of existing methods. Table 8-1 shows the correspondence between the new and existing methods.

Table 8-1. `SequencedCollection` methods, with their `NavigableSet` equivalents

SequencedCollection	NavigableSet
`getFirst`	`first` (inherited from `SortedSet`)
`getLast`	`last` (inherited from `SortedSet`)
`removeFirst`	`pollFirst`
`removeLast`	`pollLast`
`addFirst`	unsupported method for internally ordered collections

`addLast`	unsupported method for internally ordered collections
`reversed`	`descendingSet`

TreeSet

This is the first tree implementation that we have seen, so we should take a little time now to consider how trees perform in comparison to the other implementation types used by the Collections Framework.

Trees are the data structure you would choose for an application that needs fast insertion and retrieval of individual elements but which also requires that they be held in sorted order.

For example, suppose you want to match all the words from a set against a given prefix, a common requirement in visual applications where a drop-down should ideally show all the possible elements that match against the prefix that the user has typed. A hash table can't return its elements in sorted order and a list can't retrieve its elements quickly by their content, but a tree can do both.

In computing, a tree is a branching structure that represents hierarchy. Computing trees borrow a lot of their terminology from genealogical trees, though there are some differences; the most important is that, in computing trees, each node has only one parent—except the root, which has none. An important class of tree often used in computing is a *binary* tree—one in which each node can have at most two children. Figure 8-4 shows an example of a binary tree containing the words of this sentence in alphabetical order.

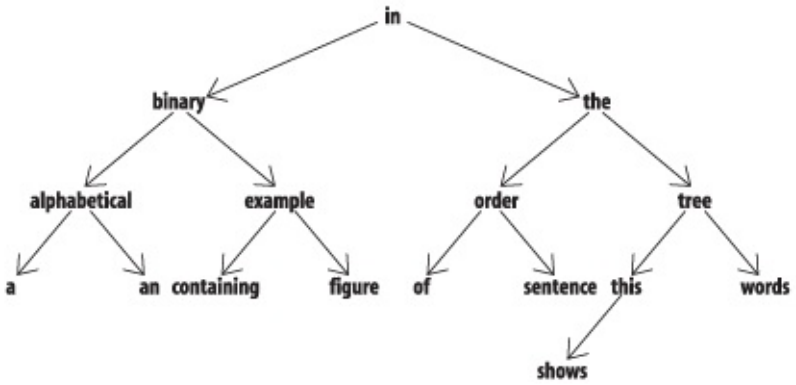

Figure 8-4. An ordered, balanced binary tree

The most important property of this tree can be seen if you look at any non-leaf node—say, the one containing the word *the*: all the nodes below that on the left contain words that precede *the* alphabetically, and all those on the right, words that follow it. To locate a word, you would start at the root and descend level by level, doing an alphabetic comparison at each level, so the cost of retrieving or inserting an element is proportional to the depth of the tree.

How deep, then, is a tree that contains *n* elements? The complete binary tree with two levels has three elements (that's 2^2–1), and the one with three levels has seven elements (2^3–1). In general, a binary tree with *n* complete levels will have 2^n–1 elements, and the depth of a tree with *n* elements will be bounded by log *n* (since 2^log *n*^ = *n*). Just as *n* grows much more slowly than 2^n, log *n* grows much more slowly than *n*. So `contains` on a large tree is much faster than on a list containing the same elements. It's still not as good as on a hash table—whose operations can ideally work in constant time—but a tree has the big advantage over a hash table that its iterator can return its elements in sorted order.

Not all binary trees will have this nice performance, though. Figure 8-4 shows a *balanced* binary tree—one in which each node has an equal number of descendants (or as near as possible) on each side. An unbalanced tree can give much worse performance—in the worst case, as bad as a linked list (see Figure 8-5). `TreeSet` uses a data type called a *red-black tree*, which has the advantage that if it becomes unbalanced through insertion or removal of an element, it can always be rebalanced in *O*(log *n*) time.

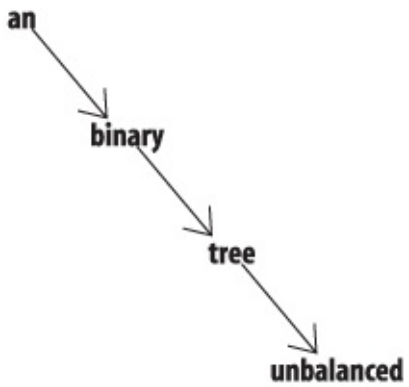

Figure 8-5. An unbalanced binary tree

The constructors for `TreeSet` include, besides the standard ones, one which allows you to supply a `Comparator` (see "Comparator<T>") and one which allows you to create one from another `SortedSet`:

```
TreeSet(Comparator<? super E> c)
              // construct an empty set which will be sorted using the
              // specified comparator
TreeSet(SortedSet<E> s)
              // construct a new set containing the elements of the
              // supplied set, sorted according to the same ordering
```

The second of these is rather too close in its declaration to the standard "conversion constructor" (see "Collection Constructors"):

```
TreeSet(Collection<? extends E> c)
              // construct a new set containing the elements of the
              // supplied set, sorted according to the natural
ordering
```

As Joshua Bloch explains in *Effective Java* (item 52 in the third edition, "Use overloading judiciously"), calling one of two constructor or method overloads which take parameters of related type can give confusing results. This is because, in Java, calls to overloaded constructors and methods are resolved at compile time on the basis of the static type of the argument, so applying a cast to an argument can make a big difference to the result of the call, as the following code shows:

```
// construct and populate a NavigableSet whose iterator returns its
// elements in the reverse of natural order:
NavigableSet<String> base = new TreeSet<>(Comparator.reverseOrder());
Collections.addAll(base, "b", "a", "c");

// call the two different constructors for TreeSet, supplying the
// set just constructed, but with different static types:
NavigableSet<String> sortedSet1 = new TreeSet<>((Set<String>)base);
NavigableSet<String> sortedSet2 = new TreeSet<>(base);
// and the two sets have different iteration orders:
List<String> forward = new ArrayList<>(sortedSet1);
List<String> backward = new ArrayList<>(sortedSet2);
assert !forward.equals(backward);
assert forward.reversed().equals(backward);
```

This problem afflicts the constructors for all the sorted collections in the Framework (`TreeSet`, `TreeMap`, `ConcurrentSkipListSet`, and `ConcurrentSkipListMap`). To avoid it in your own class designs, choose parameter types for different overloads so that an argument of a type appropriate to one overload cannot be cast to the type appropriate to a different one. If that is not possible, the two overloads should be designed to behave identically with the same argument, regardless of its static type. For example, a `PriorityQueue` ("PriorityQueue") constructed from a collection uses the ordering of the original, whether the static type with which the constructor is supplied is one of the `Comparator`-containing types `PriorityQueue` or `SortedSet`, or just a plain `Collection`. To achieve this, the conversion constructor uses a `instanceof` tests to determine the type of the supplied collection, and if that is a `SortedSet` or a `PriorityQueue`, it extracts the `Comparator`, only falling back on natural ordering if there isn't one.

`TreeSet` is unsychronized and not thread-safe; its iterators are fail-fast.

ConcurrentSkipListSet

`ConcurrentSkipListSet` was introduced in Java 6 as the first concurrent set implementation. It is backed by a *skip list*, a modern alternative to the binary trees of the previous section. A skip list for a set is a series of *linked lists*, each of which is a chain of cells consisting of two fields: one to hold a value, and one to hold a reference to the next cell. Elements are inserted into and removed from a linked list in constant time by pointer rearrangement, as shown in Figure 8-6, parts (a) and (b) respectively.

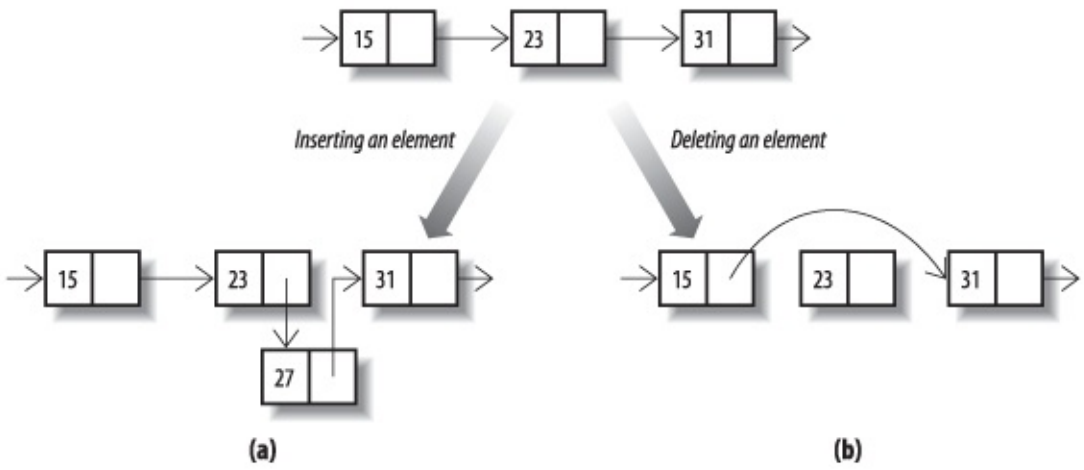

Figure 8-6. Modifying a linked list

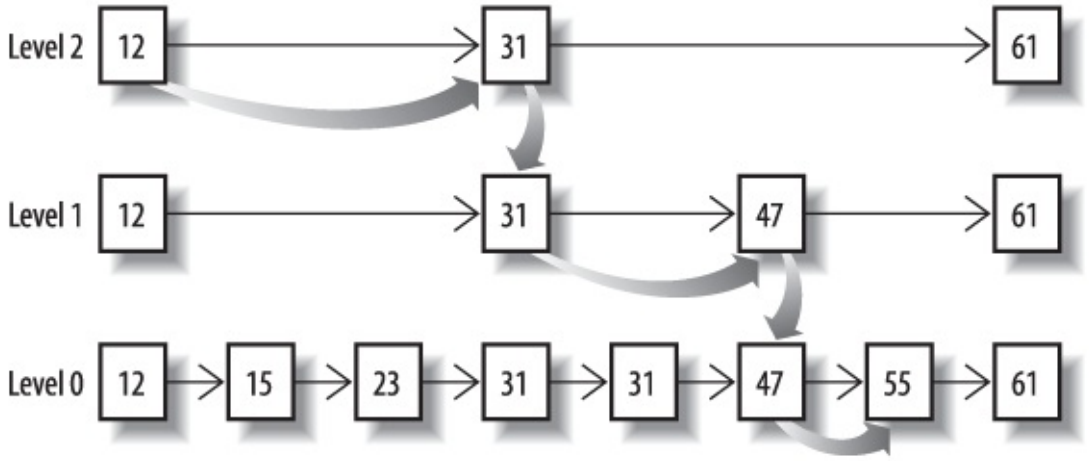

Figure 8-7. Searching a skip list

Figure 8-7 shows a skip list consisting of three linked lists, labelled levels 0, 1 and 2. The first linked list of the collection (level 0 in the figure) contains the elements of the set, sorted according to their natural order or by the comparator of the set. Each list above level 0 contains a subset of the list below, chosen randomly according to a fixed probability. For this example, let's suppose that the probability is 0.5; on average, each list will contain half the elements of the list below it. Navigating between links takes a fixed time, so the quickest way to find an element is to start at the beginning (the left-hand end) of the top list and to go as far as possible on each list before dropping to the one below it.

The curved arrows of Figure 8-7 shows the progress of a search for the element 55. The search starts with the element 12 at the top left of level 2, steps to the element 31 on that level, then finds that the next element is 61, higher than the

search value. So it drops one level, and then repeats the process; element 47 is still smaller than 55, but 61 is again too large, so it once more drops a level and finds the search value in one further step.

Inserting an element into a skip list always involves at least inserting it at level 0. When that has been done, should it also be inserted at level 1? If level 1 contains, on average, half of the elements at level 0, then we should toss a coin (that is, randomly choose with probability 0.5) to decide whether it should be inserted at level 1 as well. If the coin toss does result in it being inserted at level 1, then the process is repeated for level 2, and so on. To remove an element from the skip list, it is removed from each level in which it occurs.

If the coin tossing goes badly, we could end up with every list above level 0 empty—or full, which would be just as bad. These outcomes have very low probability, however, and analysis shows that, in fact, the probability is very high that skip lists will give performance comparable to binary trees: search, insertion and removal all take $O(\log n)$. Their compelling advantage for concurrent use is that they have efficient lock-free insertion and deletion algorithms, whereas there are none known for binary trees.

The iterators of `ConcurrentSkipListSet` are weakly consistent.

Table 8-2. Comparative performance of different Set implementations

	add	contains	next	notes
`HashSet`	$O(1)$	$O(1)$	$O(h/n)$	h is the tal capacity
`LinkedHashSet`	$O(1)$	$O(1)$	$O(1)$	
`CopyOnWriteArraySet`	$O(n)$	$O(n)$	$O(1)$	
`EnumSet`	$O(1)$	$O(1)$	$O(1)$*	
`TreeSet`	$O(\log n)$	$O(\log n)$	$O(\log n)$	

`ConcurrentSkipListSet`	*O*(log *n*)	*O*(log *n*)	*O*(1)

*In the `EnumSet` implementation for `enum` types with more than 64 values, `next` has worst case complexity of *O*(log *m*), where *m* is the number of elements in the enumeration.

Comparing Set Implementations

Table 8-2 shows the comparative performance of the different Set implementations. When you are choosing an implementation, of course, efficiency is only one of the factors you should take into account. Some of these implementations are specialized for specific situations; for example, `EnumSet` should always (and only) be used to represent sets of `enum`. Similarly, `CopyOnWriteArraySet` should only be used where set size will remain relatively small, read operations greatly outnumber writes, thread safety is required, and read-only iterators are acceptable.

That leaves the general-purpose implementations: `HashSet`, `LinkedHashSet`, `TreeSet`, and `ConcurrentSkipListSet`. The first three are not thread-safe, so can only be used in multi-threaded code either in conjunction with client-side locking, or wrapped in `Collection.synchronizedSet` (see "Synchronized Collections"). For single-threaded applications where there is no requirement for the set to be sorted, your choice is between `HashSet` and `LinkedHashSet`. If your application will be frequently iterating over the set, or if you require access ordering, `LinkedHashSet` is the implementation of choice.

Finally, if you require the set to sort its elements, the choice is between `TreeSet` and `ConcurrentSkipListSet`. In a multi-threaded environment, `ConcurrentSkipListSet` is the only sensible choice. Even in single-threaded code `ConcurrentSkipListSet` may not show a significantly worse performance for small set sizes. For larger sets, however, or for applications in which there are frequent element deletions, `TreeSet` will perform better if your application doesn't require thread safety.

Chapter 9. Queues

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 15th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

A *queue* is a collection designed to hold elements for processing, yielding them up in the order in which they are to be processed. The corresponding Collections Framework interface `Queue<E>` has a number of different implementations embodying different rules about what this order should be. Many of the implementations use the rule that tasks are to be processed in the order in which they were submitted (*First In First Out*, or *FIFO*), but other rules are possible—for example, the Collections Framework includes queue classes whose processing order is based on task priority. The `Queue` interface was introduced in Java 5, motivated in part by the need for queues in the concurrency utilities included in that release. A glance at the hierarchy of implementations shown in Figure 9-1 shows that, in fact, nearly all the `Queue` implementations in the Collections Framework are in the package `java.util.concurrent`.

One classic requirement for queues in concurrent systems arises when a number of tasks have to be executed by a number of threads working in parallel. An everyday example of this situation is that of a single queue of airline passengers being handled by a line of check-in operators. Each operator works on processing a single passenger (or a group of passengers) while the remaining passengers wait in the queue. As they arrive, passengers join the *tail* of the queue, wait until they reach its *head*, and are then assigned to the next operator

who becomes free. A good deal of fine detail is involved in implementing a queue such as this; operators have to be prevented from simultaneously attempting to process the same passenger, empty queues have to be handled correctly, and in computer systems there has to be a way of defining queues with a maximum size, or *bound*. (This last requirement may not often be imposed in airline terminals, but it can be very useful in systems in which there is a maximum waiting time for a task to be executed.) The `Queue` implementations in `java.util.concurrent` look after these implementation details for you.

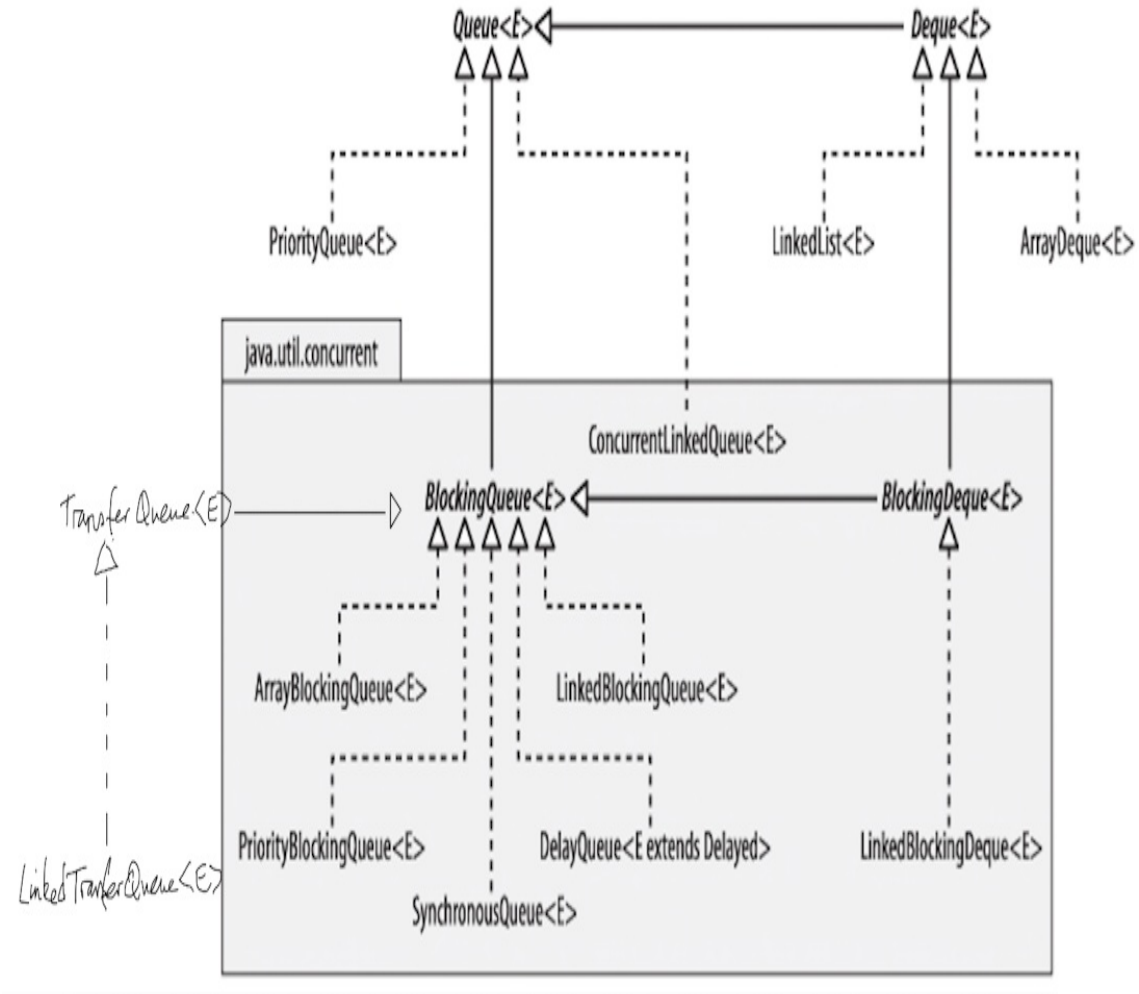

Figure 9-1. Implementations of Queue in the Collections Framework

In addition to the operations inherited from `Collection`, the `Queue` interface includes operations to add an element to the tail of the queue, to inspect the element at its head, and to remove the element at its head. Each of these three

operations comes in two varieties, one which returns a value to indicate failure and one which throws an exception.

Adding an Element to a Queue

The exception-throwing variant of this operation is the add method inherited from `Collection`. Although `add` does return a boolean signifying its success in inserting an element, that value can't be used to report that a bounded queue is full; the contract for `add` specifies that it may return `false` only if the reason for refusing the element is that it was already present—otherwise, it must throw an (unchecked) exception. For a bounded queue, the value-returning variant `offer` is usually a better option:

```
boolean offer (E  e)  // insert the given element if possible
```

The value returned by `offer` indicates whether the element was successfully inserted. Note that `offer` does throw an exception if the element is illegal in some way (for example, the value `null` submitted to a queue that doesn't permit `null`s). Normally, if `offer` returns `false`, it has been called on a bounded queue that has reached capacity.

Retrieving an Element from a Queue

The methods in this group are `peek` and `element` for inspecting the head element, and `poll` and `remove` for removing it from the queue and returning its value.

The methods that throw an exception for an empty queue are:

```
E element()    // retrieve but do not remove the head element
E remove()     // retrieve and remove the head element
```

Notice that this is a different method from the `Collection` method `remove(Object)`. The methods that return `null` for an empty queue are:

```
E peek()        // retrieve but do not remove the head element
E poll()        // retrieve and remove the head element
```

Because these methods return `null` to signify that the queue is empty, you

should avoid using `null` as a queue element. In general, the use of `null` as a queue element is discouraged by the `Queue` interface; in the JDK, the only implementation that allows it is the legacy class `LinkedList`.

Using the Methods of Queue

Let's look at examples of the use of these methods. Queues should provide a good way of implementing a task manager, since their main purpose is to yield up elements, such as tasks, for processing. For the moment we'll use `ArrayDeque` as the fastest and most straightforward implementation of `Queue` (and also of `Deque`, of course). But, as before, we'll confine ourselves to the methods of the interface—but remember that, in choosing a queue implementation, you're also choosing the ordering of task processing. With `ArrayDeque`, you get FIFO ordering—very possibly the best choice for the task manager: since our attempts to get organized using fancy scheduling methods never seem to work very well, let's try something simpler.

`ArrayDeque` is unbounded, so we could use either `add` or `offer` to set up the queue with new tasks.

```
Queue<Task> taskQueue = new ArrayDeque<>();
taskQueue.offer(mikePhone);
taskQueue.offer(paulPhone);
```

Any time we feel ready to do a task, we can take the one that has reached the head of the queue:

```
Task nextTask = taskQueue.poll();
if (nextTask != null) {
  // process nextTask
}
```

The choice between using `poll` and `remove` depends on whether we want to regard queue emptiness as an exceptional condition. Realistically—given the nature of the application—that might be a sensible assumption, so this is an alternative:

```
try {
```

```
    Task nextTask = taskQueue.remove();
    // process nextTask
  } catch (NoSuchElementException e) {
    // but we *never* run out of tasks!
  }
```

This scheme needs some refinement to allow for the nature of different kinds of tasks. Phone tasks fit into relatively short time slots, whereas we don't like to start coding unless there is reasonably substantial time to get into the task. So if time is limited—say, until the next meeting—we might like to check that the next task is of the right kind before we take it off the queue:

```
Task nextTask = taskQueue.peek();
if (nextTask instanceof PhoneTask) {
  taskQueue.remove();
  // process nextTask
}
```

These inspection and removal methods are a major benefit of the `Queue` interface; `Collection` has nothing like them (though `NavigableSet` does). The price we pay for this benefit is that the methods of `Queue` are useful to us only if the head element is actually one that we want. True, the class `PriorityQueue` allows us to provide a comparator that will order the queue elements so that the one you want is at the head, but that may not be a particularly good way of expressing the algorithm for choosing the next task—for example, you might need to know something about *all* the outstanding tasks before you can choose the next one. So in this situation, if our to-do manager is entirely queue-based, you may end up going for coffee until the meeting starts. As an alternative, you could consider using the `List` interface, which provides more flexible means of accessing its elements but has the drawback that its implementations provide much less support for multi-thread use.

This might sound overly pessimistic; after all, `Queue` is a subinterface of `Collection`, so it inherits methods that support traversal, like `iterator`. In fact, although these methods are implemented, their use is not recommended in normal situations. In the design of the queue classes, efficiency in traversal has been traded against fast implementation of the methods of `Queue`; in addition, queue iterators do not guarantee to return their elements in proper sequence and, for some concurrent queues, will actually fail in normal conditions (see

"BlockingQueue Implementations").

In the next section we shall look at the two direct JDK implementations of `Queue`—`PriorityQueue` and `ConcurrentLinkedList`—and, in "BlockingQueue", at `BlockingQueue` and its implementations. The classes in these two sections differ widely in their behavior. Most of them are thread-safe; most provide *blocking* facilities (that is, operations that wait for conditions to be right for them to execute); some support priority ordering; one—`DelayQueue`—holds elements until their delay has expired, and another—`SynchronousQueue`—is purely a synchronization facility. In choosing between `Queue` implementations, you would be influenced more by these functional differences than by their performances.

Queue Implementations

PriorityQueue

`PriorityQueue` is one of the two non-legacy `Queue` implementations (that is, other than `LinkedList`) not designed primarily for concurrent use (the other one is `ArrayDeque`). It is not thread-safe, nor does it provide blocking behavior. It gives up its elements for processing according to an ordering like that used by `NavigableSet`—either the natural order of its elements if they implement `Comparable`, or the ordering imposed by a `Comparator` supplied when the `PriorityQueue` is constructed. So `PriorityQueue` would be an alternative design choice (obviously, given its name) for the priority-based to-do manager that we outlined in "NavigableSet" using `NavigableSet`. Your application will dictate which alternative to choose: if it needs to examine and manipulate the set of waiting tasks, use `NavigableSet`. If its main requirement is efficient access to the next task to be performed, use `PriorityQueue`.

Choosing `PriorityQueue` allows us to reconsider the ordering: since it accommodates duplicates, it does not share the requirement of `NavigableSet` for an ordering consistent with `equals`. To emphasize the point, we will define a new ordering for our to-do manager that depends only on priorities. Contrary

to what you might expect, `PriorityQueue` gives no guarantee of how it presents multiple elements with the same value. So if, in our example, several tasks are tied for the highest priority in the queue, it will choose one of them arbitrarily as the head element.

The constructors for `PriorityQueue` are:

```
PriorityQueue()       // natural ordering, default initial capacity
(11)
PriorityQueue(Collection<? extends E> c)
                      // natural ordering of elements taken from c,
unless
                      // c is a PriorityQueue or SortedSet, in which
case
                      // copy c's ordering
PriorityQueue(int initialCapacity)
                      // natural ordering, specified initial capacity
PriorityQueue(int initialCapacity, Comparator<? super E> comparator)
                      // Comparator ordering, specified initial
capacity
PriorityQueue(PriorityQueue<? extends E> c)
                      // ordering and elements copied from c
PriorityQueue(SortedSet<? extends E> c)
                      // ordering and elements copied from c
```

Notice how the second of these constructors avoids the problem of the overloaded `TreeSet` constructor that we discussed in “TreeSet”. We can use `PriorityQueue` for a simple implementation of our to-do manager with the `PriorityTask` class defined in “NavigableSet”, and a new `Comparator` depending only on the task’s priority:

```
final int INITIAL_CAPACITY = 10;
Comparator<PriorityTask> priorityComp =
Comparator.comparing(PriorityTask::priority);
Queue<PriorityTask> priorityQueue = new PriorityQueue<>
(INITIAL_CAPACITY, priorityComp);
priorityQueue.add(new PriorityTask(mikePhone, Priority.MEDIUM));
priorityQueue.add(new PriorityTask(paulPhone, Priority.HIGH));
PriorityTask nextTask = priorityQueue.poll();
System.out.println(nextTask);
...
PriorityTask nextTask = priorityQueue.poll();
```

Priority queues are usually efficiently implemented by *priority heaps*. A priority

heap is a binary tree somewhat like those we saw implementing `TreeSet` in "TreeSet", but with two differences: first, the only ordering constraint is that each node in the tree should be greater than either of its children, and second, that the tree should be complete at every level except possibly the lowest; if the lowest level is incomplete, the nodes it contains must be grouped together at the left. Figure 9-2(a) shows a small priority heap, with each node shown only by the field containing its priority. To add a new element to a priority heap, it is first attached at the leftmost vacant position, as shown by the circled node in Figure 9-2(b). Then it is repeatedly exchanged with its parent until it reaches a parent that has higher priority. In the figure, this required only a single exchange of the new element with its parent, giving Figure 9-2(c). (Nodes shown circled in Figures Figure 9-2 and Figure 9-3 have just changed position.)

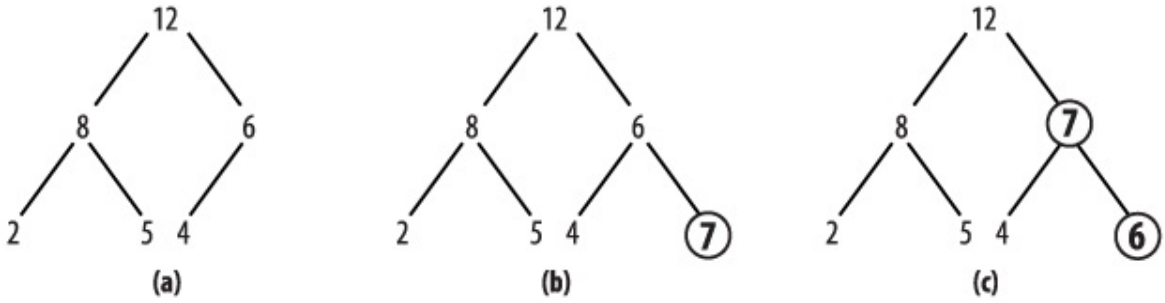

Figure 9-2. Adding an element to a PriorityQueue

Getting the highest-priority element from a priority heap is trivial: it is the root of the tree. But, when that has been removed, the two separate trees that result must be reorganized into a priority heap again. This is done by first placing the rightmost element from the bottom row into the root position. Then—in the reverse of the procedure for adding an element—it is repeatedly exchanged with the larger of its children until it has a higher priority than either. Figure 9-3 shows the process—again requiring only a single exchange—starting from the heap in Figure 9-2(c) after the head has been removed.

Apart from constant overheads, both addition and removal of elements require a number of operations proportional to the height of the tree. So `PriorityQueue` provides $O(\log n)$ time for `offer`, `poll`, `remove()`, and `add`. The methods `remove(Object)` and `contains` may require the entire tree to be traversed, so they require $O(n)$ time. The methods `peek` and `element`, which just retrieve the root of the tree without removing it, take

constant time, as does `size`, which uses an object field that is continually updated.

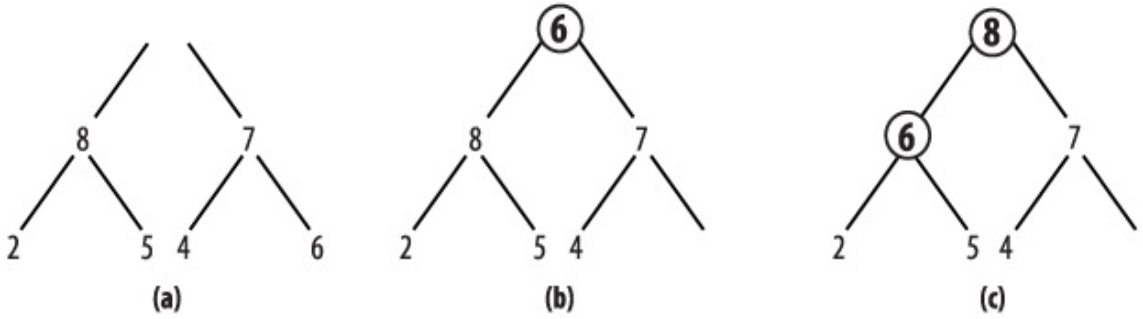

Figure 9-3. Removing the head of a PriorityQueue

`PriorityQueue` is not suitable for concurrent use. Its iterators are fail-fast, and it doesn't offer support for client-side locking. A thread-safe version, `PriorityBlockingQueue` (see "BlockingQueue Implementations"), is provided instead.

ConcurrentLinkedQueue

The other nonblocking `Queue` implementation is `ConcurrentLinkedQueue`, an unbounded, thread-safe, FIFO-ordered queue. It uses a linked structure, similar to those we saw in "ConcurrentSkipListSet" as the basis for skip lists, and in "HashSet" for hash table overflow chaining. We noticed there that one of the main attractions of linked structures is that the insertion and removal operations implemented by pointer rearrangements perform in constant time. This makes them especially useful as queue implementations, where these operations are always required on cells at the ends of the structure—that is, cells that do not need to be located using the slow sequential search of linked structures.

`ConcurrentLinkedQueue` uses a CAS-based (see "Java 5: Concurrent Collections") *wait-free* algorithm—that is, one that guarantees that any thread can always complete its current operation, regardless of the state of other threads accessing the queue. It executes queue insertion and removal operations in constant time, but requires linear time to execute `size`. This is because the algorithm, which relies on co-operation between threads for insertion and removal, does not keep track of the queue size and has to iterate over the queue

to calculate it when it is required.

`ConcurrentLinkedQueue` has the two standard constructors discussed in "Collection Constructors". Its iterators are weakly consistent.

BlockingQueue

Java 5 added a number of classes to the Collections Framework for use in concurrent applications. Most of these are implementations of the `Queue` subinterface `BlockingQueue<E>`, designed primarily to be used in producer-consumer queues.

One common example of the use of producer-consumer queues is in systems that perform print spooling; client processes add print jobs to the spool queue, to be processed by one or more print service processes, each of which repeatedly consumes the task at the head of the queue. The key facilities that `BlockingQueue` provides to such systems are, as its name implies, enqueuing and dequeueing methods that do not return until they have executed successfully. So, in this example, a print server does not need to constantly poll the queue to discover whether any print jobs are waiting; it need only call the `poll` method, supplying a timeout, and the system will suspend it until either a queue element becomes available or the timeout expires. `BlockingQueue` defines seven new methods, in three groups:

Adding an Element

```
boolean offer(E e, long timeout, TimeUnit unit)
                // insert e, waiting up to the timeout
void put(E e)   // add e, waiting as long as necessary
```

The methods inherited from `Queue`—`add` and `offer`—fail immmediately if called on a bounded queue that has reached capacity: `add` by throwing an exception, `offer` by returning `false`. These blocking methods are more patient: the `offer` overload waits for a time specified using `java.util.concurrent.TimeUnit` (an `Enum` which allows timeouts to be defined in units such as milliseconds or seconds) and `put` will block indefinitely.

Removing an Element

```
E poll(long timeout, TimeUnit unit)
                // retrieve and remove the head, waiting up to the
timeout
E take()        // retrieve and remove the head of this queue, waiting
                // as long as necessary
```

Taking these methods together with those inherited from `Queue`, methods for removing an element have the same four possible behaviors in the case of failure as those for adding an element. In this case, failure will typically be caused by the queue currently containing no elements. The inherited `poll` method returns `null` in this situation, blocking `poll` returns `null` but only after waiting until its timeout, `remove` throws an exception immediately, and `take` blocks until it succeeds.

Although the behavior of the various `BlockingQueue` methods varies systematically, their naming can be difficult to understand. The clearest picture is provided by the Javadoc (Figure 9-4).

Summary of BlockingQueue methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

Figure 9-4. `BlockingQueue` methods

Retrieving or Querying the Contents of the Queue

```
int drainTo(Collection<? super E> c)
                // clear the queue into c
int drainTo(Collection<? super E> c, int maxElements)
                // clear at most the specified number of elements into
c
int remainingCapacity()
                // return the number of elements that would be
accepted
                // without blocking, or Integer.MAX_VALUE if unbounded
```

The `drainTo` methods perform atomically and efficiently, so the second

overload is useful in situations in which you know that you have processing capability available immediately for a certain number of elements, and the first is useful—for example—when all producer threads have stopped working. Their return value is the number of elements transferred. `remainingCapacity` reports the spare capacity of the queue, although as with any such value in multi-threaded contexts, the result of a call should not be used as part of a test-then-act sequence: between the test (the call of `remainingCapacity`) and the action (adding an element to the queue) of one thread, another thread might have intervened to add or remove elements.

`BlockingQueue` guarantees that the queue operations of its implementations will be thread-safe and atomic. But this guarantee doesn't extend to the bulk operations inherited from `Collection`—`addAll`, `containsAll`, `retainAll` and `removeAll`—unless the individual implementation provides it. So it is possible, for example, for `addAll` to fail, throwing an exception, after adding only some of the elements in a collection.

Using the Methods of BlockingQueue

A to-do manager that works for just one person at a time is very limited; we really need a cooperative solution—one that will allow us to share both the production and the processing of tasks. Example 9-1 shows `StoppableTaskQueue`, a simple version of a concurrent task manager based on `PriorityBlockingQueue`, that will allow its users—us—to independently add tasks to the task queue as we discover the need for them, and to take them off for processing as we find the time. The class `StoppableTaskQueue` has three methods: `addTasks`, `getTask`, and `shutDown`. A `StoppableTaskQueue` is either working or stopped. The method `addTasks` returns a `boolean` value indicating whether it successfully added its tasks; this value will be true unless the `StoppableTaskQueue` is stopped. The method `getTask` returns the head task from the queue. If no task is available, it does not block but returns `null`. The method `shutDown` stops the `StoppableTaskQueue`, waits until all pending `addTasks` operations are completed, then drains the `StoppableTaskQueue` and returns its contents.

Example 9-1. A concurrent queue-based task manager

```
public class StoppableTaskQueue {
    private final int MAXIMUM_PENDING_OFFERS = Integer.MAX_VALUE;
    private final BlockingQueue<PriorityTask> taskQueue = new
PriorityBlockingQueue<>();
    private final Semaphore semaphore = new
Semaphore(MAXIMUM_PENDING_OFFERS);
    private volatile boolean isStopping;

    // return true if the task was successfully placed on the queue,
false
    // if the queue is being shut down.
    public boolean addTask(PriorityTask task) {
        return addTasks(Collections.singletonList(task));
    }

    public boolean addTasks(Collection<PriorityTask> tasks) {
        if (isStopping) return false;
        if (! semaphore.tryAcquire()) {
            return false;
        } else {
            taskQueue.addAll(tasks);
            semaphore.release();
            return true;
        }
    }

    // return the head task from the queue, or null if no task is
available
    public PriorityTask getFirstTask() {
        return taskQueue.poll();
    }

    // stop the queue, wait for producers to finish, then return the
contents
    public Collection<PriorityTask> shutDown() {
        isStopping = true;
        // blocks until all outstanding addTasks() calls have completed
        semaphore.acquireUninterruptibly(MAXIMUM_PENDING_OFFERS);
        Set<PriorityTask> returnCollection = new HashSet<>();
        taskQueue.drainTo(returnCollection);
        return returnCollection;
    }
}
```

In this example, as in most uses of the `java.util.concurrent` collections, the collection itself takes care of the problems arising from the interaction of different threads in adding or removing items from the queue. Most of the code

of Example 9-1 is instead solving the problem of providing an orderly shutdown mechanism. The reason for this emphasis is that when we go on to use the class `StoppableTaskQueue` as a component in a larger system, we will need to be able to stop daily task queues without losing task information. Achieving graceful shutdown can often be a problem in concurrent systems: for more detail, see Chapter 7 of [Goetz06].

The larger system will model each day's scheduled tasks over the next year, allowing consumers to process tasks from each day's queue. An implicit assumption of the example of this section is that if there are no remaining tasks scheduled for this day, a consumer will not wait for one to become available, but will immediately go on to look for a task in the next day's queue. (In the real world, we would go home at this point, or more likely go out to celebrate.) This assumption simplifies the example, as we don't need to invoke any of the blocking methods of `PriorityBlockingQueue`, though we will use one method, `drainTo`, from the `BlockingQueue` interface.

There are a number of ways of shutting down a producer-consumer queue such as this; in the one we've chosen for this example, the manager exposes a `shutdown` method that can be called by a "supervisor" thread in order to stop producers writing to the queue, to drain it, and to return the outstanding tasks. The difficulty is in ensuring that `addtasks` performs atomically—that is, it adds all of its tasks or none of them—and that once the `shutdown` method has stopped the queue and is draining or has drained it, no other threads can subsequently gain access to it to add further tasks.

Example 9-1 achieves this using a *semaphore*—a thread-safe object that maintains a fixed number of *permits*. Semaphores are usually used to regulate access to a finite set of resources—a pool of database connections, for example. The permits the semaphore has available at any time represent the resources not currently in use. A thread requiring a resource acquires a permit from the semaphore, and releases it when it releases the resource. If all the resources are in use, the semaphore will have no permits available; at that point, a thread attempting to acquire a permit will block until some other thread returns one.

The semaphore in this example is used differently. We don't want to restrict producer threads from writing to the queue—it's a concurrent queue, after all, quite capable of handling multi-thread access without help from us. We just want

the `shutdown` method to be able to tell if there are any writes currently in progress. So we create the semaphore with the largest possible number of permits, which in practice will never all be required. The producer method `addTasks` first checks the volatile boolean flag `isStopping` to ensure that a shutdown has not been initiated, then calls the semaphore method `tryAcquire`, which returns `false` immediately if no permits are available (indicating that the shutdown is in process or has completed). If `tryAcquire` succeeds, `addTasks` adds its tasks to the queue, then releases the permit.

The `shutdown` method first sets the `isStopping` flag, so that no new `addTasks` method executions can begin. Then it has to wait until all the permits previously acquired have been returned. To do that, it calls `acquireUninterruptibly`, specifying that it needs *all* the permits; that call will block until they are all released by the producer threads, and shutdown can be completed by returning the unprocessed tasks.

BlockingQueue Implementations

The Collections Framework provides five implementations of `BlockingQueue`.

LinkedBlockingQueue

This class is a thread-safe, FIFO-ordered queue, based on a linked node structure. It is the implementation of choice whenever you need an unbounded blocking queue. Even for bounded use, it may still be better than `ArrayBlockingQueue` (linked queues typically have higher throughput than array-based queues but less predictable performance in most concurrent applications).

The two standard collection constructors create a thread-safe blocking queue with a capacity of `Integer.MAX_VALUE`. You can specify a lower capacity using a third constructor:

```
LinkedBlockingQueue(int capacity)
```

The ordering imposed by `LinkedBlockingQueue` is FIFO. Queue insertion and removal are executed in constant time; operations such as `contains` which

require traversal of the array require linear time. The iterators are weakly consistent.

ArrayBlockingQueue

This implementation is based on a *circular array*—a linear structure in which the first and last elements are logically adjacent. Figure 9-5(a) shows the idea. The position labeled "head" indicates the head of the queue; each time the head element is removed from the queue, the head index is advanced. Similarly, each new element is added at the tail position, resulting in that index being advanced. When either index needs to be advanced past the last element of the array, it gets the value 0. If the two indices have the same value, the queue is either full or empty, so an implementation must separately keep track of the count of elements in the queue.

A circular array in which the head and tail can be continuously advanced in this way is better as a queue implementation than a noncircular one (e.g., the standard implementation of `ArrayList`, which we cover in "`List` Implementations") in which removing the head element requires changing the position of all the remaining elements so that the new head is at position 0. Notice, though, that only the elements at the ends of the queue can be inserted and removed in constant time. If an element is to be removed from near the middle, which can be done for queues via the method `Iterator.remove`, then all the elements from one end must be moved along to maintain a compact representation. Figure 9-5(b) shows the element at index 6 being removed from the queue. As a result, insertion and removal of elements in the middle of the queue has time complexity $O(n)$.

Constructors for array-backed collection classes generally have a single configuration parameter, the initial length of the array. For fixed-size classes like `ArrayBlockingQueue`, this parameter is necessary in order to define the capacity of the collection. (For variable-size classes like `ArrayList`, a default initial capacity can be used, so constructors are provided that don't require the capacity.) For `ArrayBlockingQueue`, the three constructors are:

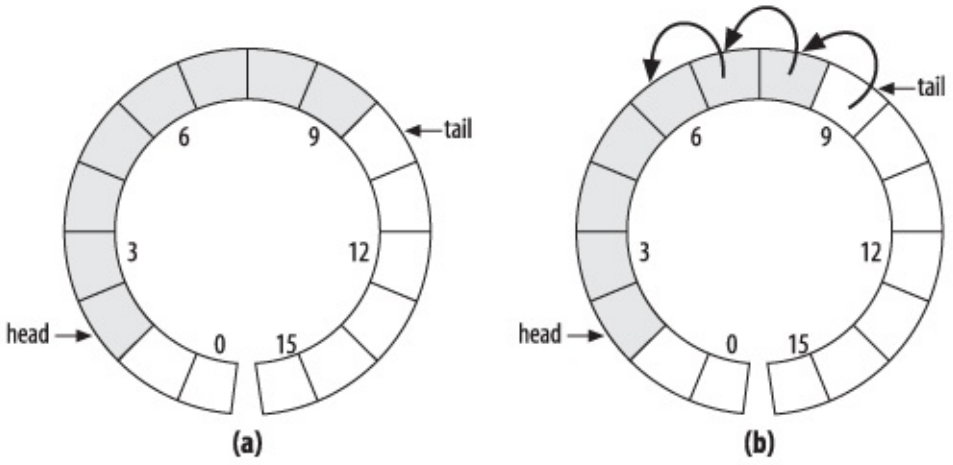

Figure 9-5. A circular array

```
ArrayBlockingQueue(int capacity)
ArrayBlockingQueue(int capacity, boolean fair)
ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends E>
c)
```

The `Collection` parameter to the last of these allows an `ArrayBlockingQueue` to be initialized with the contents of the specified collection, added in the traversal order of the collection's iterator. For this constructor, the specified capacity must be at least as great as the size of the supplied collection, or at least 1 if the supplied collection is empty. Besides configuring the backing array, the last two constructors also require a boolean argument to control how the queue will handle multiple blocked requests. These will occur when multiple threads attempt to remove items from an empty queue or enqueue items on to a full one. When the queue becomes able to service one of these requests, which one should it choose? The alternatives are to provide a guarantee that the queue will choose the one that has been waiting longest—that is, to implement a *fair* scheduling policy—or to allow the implementation to choose one. Fair scheduling sounds like the better alternative, since it avoids the possibility that an unlucky thread might be delayed indefinitely but, in practice, the benefits it provides are rarely important enough to justify incurring the large overhead that it imposes on a queue's operation. If fair scheduling is not specified, `ArrayBlockingQueue` will normally approximate fair operation, but with no guarantees.

The ordering imposed by `ArrayBlockingQueue` is FIFO. Queue insertion

and removal are executed in constant time; operations, such as `contains`, which require traversal of the array, require linear time. The iterators are weakly consistent.

PriorityBlockingQueue

This implementation is a thread-safe, blocking version of `PriorityQueue` (see "Queue Implementations"), with similar ordering and performance characteristics. Its iterators are fail-fast, so in normal use they will throw `ConcurrentModificationException`; only if the queue is quiescent will they succeed. To iterate safely over a `PriorityBlockingQueue`, transfer the elements to an array and iterate over that instead.

DelayQueue

This is a specialized priority queue, in which the ordering is based on the *delay time* for each element—the time remaining before the element will be ready to be taken from the queue. If all elements have a positive delay time—that is, none of their associated delay times has expired—an attempt to `poll` the queue will return `null`. If one or more elements has an expired delay time, the one with the longest-expired delay time will be at the head of the queue. The elements of a `DelayQueue` belong to a class that implements `java.util.concurrent.Delayed`:

```
interface Delayed extends Comparable<Delayed> {
    long getDelay(TimeUnit unit);
}
```

The `getDelay` method of a `Delayed` object returns the remaining delay associated with that object. (This interface precedes the introduction of the `java.time` API but, with that now available, an obvious improvement to this interface would be the addition of a default `getDelay` method returning a `Duration`.) The `compareTo` method (see "Comparable<T>") of `Comparable` must be defined to give results that are consistent with the delays of the objects being compared. This means that it will rarely be compatible with `equals`, so `Delayed` objects are not suitable for use with implementations of `SortedSet` and `SortedMap`.

For example, in our to-do manager we are likely to need reminder tasks, to ensure that we follow up e-mail and phone messages that have gone unanswered. We could define a new class `DelayedTask` as in Example 9-2, and use it to implement a reminder queue.

```
BlockingQueue<DelayedTask> reminderQueue = new DelayQueue<DelayedTask>
();
reminderQueue.offer(new DelayedTask(databaseCode, 1));
reminderQueue.offer(new DelayedTask(interfaceCode, 2));
...
// now get the first reminder task that is ready to be processed
DelayedTask t1 = reminderQueue.poll();
if (t1 == null) {
  // no reminders ready yet
} else {
  // process t1
}
```

Example 9-2. The class DelayedTask

```
public class DelayedTask implements Delayed {

    private final LocalDateTime endTime;
    private final Task task;
    private static Comparator<Delayed> dtComparator =
        Comparator.comparing(dt -> dt.getDelay(TimeUnit.MILLISECONDS));

    public DelayedTask(Task t, int daysDelay) {
        task = t;
        endTime = LocalDateTime.now().plusDays(daysDelay);
    }

    @Override
    public long getDelay(TimeUnit unit) {
        return unit.convert(Duration.between(LocalDateTime.now(),
endTime));
    }

    @Override
    public int compareTo(Delayed d) {
        return dtComparator.compare(this, d);
    }

    public Task getTask() {
        return task;
    }
```

Most queue operations respect delay values and will treat a queue with no unexpired elements as if it were empty. The exceptions are `peek` and `remove`, which, perhaps surprisingly, will allow you to examine the head element of a `DelayQueue` whether or not it is expired. Like them and unlike the other methods of `Queue`, collection operations on a `DelayQueue` do not respect delay values. For example, here are two ways of copying the elements of `reminderQueue` into a set:

```
Set<DelayedTask> delayedTaskSet1 = new HashSet<DelayedTask>();
delayedTaskSet1.addAll(reminderQueue);
Set<DelayedTask> delayedTaskSet2 = new HashSet<DelayedTask>();
reminderQueue.drainTo(delayedTaskSet2);
```

The set `delayedTaskSet1` will contain all the reminders in the queue, whereas the set `delayedTaskSet2` will contain only those ready to be used.

`DelayQueue` shares the performance characteristics of the `PriorityQueue` on which it is based and, like it, has fail-fast iterators. The comments on `PriorityBlockingQueue` iterators apply to these too.

SynchronousQueue

At first sight, you might think there is little point to a queue with no internal capacity, which is a short description of `SynchronousQueue`. But, in fact, it can be very useful; a thread that wants to add an element to a `SynchronousQueue` must wait until another thread is ready to simultaneously take it off, and the same is true—in reverse—for a thread that wants to take an element off the queue. So `SynchronousQueue` has the function that its name suggests, that of a *rendezvous*—a mechanism for synchronizing two threads. (Don't confuse the concept of synchronizing threads in this way—allowing them to cooperate by exchanging data—with Java's keyword `synchronized`, which prevents simultaneous execution of code by different threads.) There are two constructors for `SynchronousQueue`:

```
SynchronousQueue()
SynchronousQueue(boolean fair)
```

A common application for `SynchronousQueue` is in work-sharing systems

where the design ensures that there are enough consumer threads to ensure that producer threads can hand tasks over without having to wait. In this situation, it allows safe transfer of task data between threads without incurring the `BlockingQueue` overhead of enqueuing, then dequeuing, each task being transferred.

As far as the `Collection` methods are concerned, a `SynchronousQueue` behaves like an empty `Collection`; `Queue` and `BlockingQueue` methods behave as you would expect for a queue with zero capacity, which is therefore always empty. The `iterator` method returns an empty iterator, in which `hasNext` always returns `false`.

TransferQueue

The interface `TransferQueue<E>` provides producers with a way of choosing between enqueuing data synchronously and asynchronously—useful, for example, in messaging systems that allow both synchronous and asynchronous messages. As an extension of `BlockingQueue`, it provides a system with the ability to throttle production by blocking producers from adding indefinitely to a bounded queue. (This feature is not actually used in the only platform implementation, `LinkedTransferQueue`, which is always unbounded; so, for this class, `put` always implements asynchronous queueing.) In addition, however, it exposes a new method `transfer`, which a producer can call if it wishes to block until the enqueued element has been taken by a consumer—a synchronous handshake like that provided by `SynchronousQueue`. `transfer` is provided in three versions:

```
void transfer(E e)        // Transfers the element to a consumer,
waiting as long
                          // as necessary
boolean tryTransfer(E e); // Transfers the element to a consumer if
possible
boolean tryTransfer(E e, long timeout, TimeUnit unit)
                          // Transfers the element to a consumer,
waiting up to
                          // the  timeout
```

It also exposes two helper methods that provide a rough metric of the waiting consumer count: like all methods summarizing the state of a concurrent

collection, these methods will at best give a snapshot value, which may have changed by the time the results can be processed:

```
boolean hasWaitingConsumer();  // returns true if there is at least
one waiting consumer
int getWaitingConsumerCount(); // returns an estimate of the number of
waiting consumers
```

The JDK offers one implementation of `TransferQueue`, `LinkedTransferQueue`. This is an unbounded FIFO queue, with some interesting properties: it is lock-free, like `ConcurrentLinkedQueue` but with the blocking methods that that class lacks; it supports the `transfer` methods of its interface via a "dual queue" whose nodes can represent either enqueued data or outstanding dequeue requests; and, unusually among concurrent classes, it provides fairness without degrading performance. In fact, it outperforms `SynchronousQueue` even in the latters "unfair" mode. Relationship to the other classes of `java.util.concurrent` is discussed in [design retrospective].

Enqueueing and dequeueing are both *O*(1). The iterator is weakly consistent.

Deques

A *deque* (pronounced "deck") is a double-ended queue. Unlike a queue, in which elements can be inserted only at the tail and inspected or removed only at the head, a deque can accept elements for insertion and present them for inspection or removal at either end. Also unlike `Queue`, the contract for the Collections Framework interface `Deque<E>` specifies the ordering it will use in presenting its elements: it is a linear structure in which elements added at the tail are yielded up in the same order at the head. Used as a queue, then, a `Deque` is always a FIFO structure; the contract does not allow for, say, priority deques. If elements are removed from the same end (either head or tail) at which they were added, a `Deque` acts as a stack or *LIFO* (*Last In First Out*) structure.

The fast `Deque` implementation `ArrayDeque` uses a circular array (see "BlockingQueue Implementations"), and is the implementation of choice for stacks and queues. Concurrent deques have a special role to play in

parallelization, discussed in “BlockingDeque”.

The `Deque` interface extends `Queue` with methods symmetric with respect to head and tail. For clarity of naming, the `Queue` methods that implicitly refer to one end of the queue acquire a synonym that makes their behavior explicit for `Deque`. For example, the methods `peek` and `offer`, inherited from `Queue`, are equivalent to `peekFirst` and `offerLast`. (First and last refer to the head and tail of the deque; the Javadoc for `Deque` also uses “front” and “end”.)

Collection-like Methods

```
void addFirst(E e)      // insert e at the head if there is enough
space
void addLast(E e)       // insert e at the tail if there is enough
space
void push(E e)          // insert e at the head if there is enough
space
boolean removeFirstOccurrence(Object o);
                        // remove the first occurrence of o
boolean removeLastOccurrence(Object o);
                        // remove the last occurrence of o
Iterator<E> descendingIterator()
                        // get an iterator, returning deque elements
in
                        // reverse order
```

The contracts for the methods `addFirst` and `addLast` are similar to the contract for the add method of `Collection`, but specify in addition where the element to be added should be placed, and that the exception to be thrown if it cannot be added is `IllegalStateException`. As with bounded queues, users of bounded deques should avoid these methods in favor of `offerFirst` and `offerLast`, which can report “normal” failure by means of a returned `boolean` value.

The method name `push` is a synonym of `addFirst`, provided for the use of `Deque` as a stack. The methods `removeFirstOccurrence` and `removeLastOccurrence` are analogues of `Collection.remove`, but specify in addition exactly which occurrence of the element should be removed. The return value signifies whether an element was removed as a result of the call.

Queue-like Methods

```
boolean offerFirst(E e) // insert e at the head if the deque has space
boolean offerLast(E e)  // insert e at the tail if the deque has space
```

The method `offerLast` is a renaming of the equivalent method `offer` on the `Queue` interface.

The methods that return `null` for an empty deque are:

```
E peekFirst()          // retrieve but do not remove the first element
E peekLast()           // retrieve but do not remove the last element
E pollFirst()          // retrieve and remove the first element
E pollLast()           // retrieve and remove the last element
```

The methods `peekFirst` and `pollFirst` are renamings of the equivalent methods `peek` and `poll` on the `Queue` interface.

The methods that throw an exception for an empty deque are:

```
E getFirst()          // retrieve but do not remove the first element
E getLast()           // retrieve but do not remove the last element
E removeFirst()       // retrieve and remove the first element
E removeLast()        // retrieve and remove the last element
E pop()               // retrieve and remove the first element
```

The methods `getFirst` and `removeFirst` are renamings of the equivalent methods `element` and `remove` on the `Queue` interface. The method name `pop` is a synonym `forremoveFirst`, again provided for stack use.

Inherited from `SequencedCollection`

Of the seven methods of `SequencedCollection`, six are in fact promoted from `Deque`. The only new one—which is also the only one providing a view of a `Deque`—is `reversed`, which is a covariant override of the `SequencedCollection` method, returning a `Deque`.

```
Deque<E> reversed()   // return a reverse-ordered view of this Deque
```

Deque Implementations

ArrayDeque

Along with the introduction of the interface `Deque` came a very efficient implementation, `ArrayDeque`, based on a circular array like that of `ArrayBlockingQueue` (see "BlockingQueue Implementations"). It fills a gap among `Queue` classes; previously, if you wanted a FIFO queue to use in a single-threaded environment, you would have had to use the class `LinkedList` (which we cover next, but which should be avoided as a general-purpose `Queue` implementation), or else pay an unnecessary overhead for thread safety with one of the concurrent classes `ArrayBlockingQueue` or `LinkedBlockingQueue`. `ArrayDeque` is instead the general-purpose implementation of choice, for both deques and FIFO queues. It has the performance characteristics of a circular array: adding or removing elements at the head or tail takes constant time. The iterators are fail-fast.

LinkedList

Among `Deque` implementations `LinkedList` is an oddity; for example, it is alone in permitting `null` elements, which are discouraged by the `Queue` interface because of the common use of `null` as a special value. It has been in the Collections Framework from the start, originally as one of the standard implementations of `List` (see "`List` Implementations"), and was retrofitted with the methods of `Queue` when that was introduced, and then later those of `Deque`. It is based on a linked list structure similar to those we saw in "ConcurrentSkipListSet" as the basis for skip lists, but with an extra field in each cell, pointing to the previous entry (see Figure 9-6). These pointers allow the list to be traversed backwards—for example, for reverse iteration, or to remove an element from the end of the list.

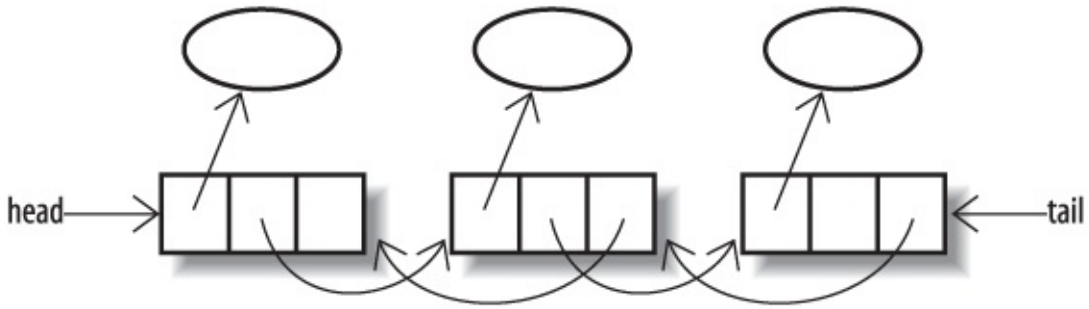

Figure 9-6. A doubly linked list

As an implementation of `Deque`, `LinkedList` is unlikely to be very popular. Its main advantage, that of constant-time insertion and removal, is rivalled for queues and deques by the otherwise superior `ArrayDeque`. The only likely reason for using `LinkedList` as a queue or deque implementation would be that you also need to add or remove elements from the middle of the list. With `LinkedList`, even that comes at a high price; because such elements have to be found by linear search, you can't escape a time complexity of $O(n)$.

The only method exposed by `LinkedList` that is not inherited from its two interfaces, `Deque` and `List` is a covariant override of `reversed`:

```
LinkedList<E> reversed()   // return a reverse-ordered view of this
LinkedList
```

The constructors for `LinkedList` are just the standard ones of "Collection Constructors". Its iterators are fail-fast.

BlockingDeque

In "BlockingQueue" we saw that `BlockingQueue` adds four methods to the `Queue` interface, enqueueing or dequeueing an element either indefinitely or until a fixed timeout has elapsed. `BlockingDeque` provides two new methods for each of those four, to allow for the operation to be applied either to the head or the tail of the deque. So, for example, to the `BlockingQueue` method

```
void put(E e)   // add e to the Queue, waiting as long as necessary
```

`BlockingDeque` adds

```
void putFirst(E e)  // add e to the head of the Deque, waiting as long
as necessary
void putLast(E e)   // add e to the tail of the Deque, waiting as long
as necessary
```

Clearly `put` and `putLast` are synonymous, as are `take` and `takeFirst`, and so on; the name duplication is provided for clarity of use.

Good load balancing algorithms will be increasingly important as multicore and

multiprocessor architectures become standard. Concurrent deques are the basis of one of the best load balancing methods, *work stealing*. To understand work stealing, imagine a load-balancing algorithm that distributes tasks in some way —round-robin, say—to a series of queues, each of which has a dedicated consumer thread that repeatedly takes a task from the head of its queue, processes it, and returns for another. Although this scheme does provide speedup through parallelism, it has a major drawback: we can imagine two adjacent queues, one with a backlog of long tasks and a consumer thread struggling to keep up with them, and next to it an empty queue with an idle consumer waiting for work. It would clearly improve throughput if we allowed the idle thread to take a task from the head of another queue.Work stealing improves still further on this idea; observing that for the idle thread to steal work from the head of another queue risks contention for the head element, it changes the queues for deques and instructs idle threads to take a task from the *tail* of another thread's deque. This turns out to be a highly efficient mechanism, and is becoming widely used.

Implementing BlockingDeque

The interface `BlockingDeque` has a single implementation, in the JDK: `LinkedBlockingDeque`. `LinkedBlockingDeque` is based on a doubly linked list structure like that of `LinkedList`. It can optionally be bounded so, besides the two standard constructors, it provides a third which can be used to specify its capacity:

```
LinkedBlockingDeque(int capacity)
```

It has similar performance characteristics to `LinkedBlockingQueue`—queue insertion and removal take constant time, and operations such as `contains`, which require traversal of the queue, require linear time. The iterators are weakly consistent.

Comparing Queue Implementations

Table 9-1 shows the sequential performance, disregarding locking and CAS overheads, for some sample operations of the `Deque` and `Queue`

implementations we have discussed. These results should be interesting to you in terms of understanding the behavior of your chosen implementation but, as we mentioned at the start of the chapter, they are unlikely to be the deciding factor. Your choice is more likely to be dictated by the functional and concurrency requirements of your application.

In choosing a `Queue`, the first question to ask is whether the implementation you choose needs to support concurrent access; if not, your choice is straightforward. For FIFO ordering, choose `ArrayDeque`; for priority ordering, `PriorityQueue`.

If your application does demand thread safety, you next need to consider ordering. If you need priority or delay ordering, the choice obviously must be `PriorityBlockingQueue` or `DelayQueue`, respectively. If, on the other hand, FIFO ordering is acceptable, the third

Table 9-1. Comparative performance of different Queue and Deque implementations

	offer	peek	poll	size
`PriorityQueue`	*O*(log *n*)	*O*(1)	*O*(log *n*)	*O*(1)
`ConcurrentLinkedQueue`	*O*(1)	*O*(1)	*O*(1)	*O*(*n*)
`ArrayBlockingQueue`	*O*(1)	*O*(1)	*O*(1)	*O*(1)
`LinkedBlockingQueue`	*O*(1)	*O*(1)	*O*(1)	*O*(1)
`PriorityBlockingQueue`	*O*(log *n*)	*O*(1)	*O*(log *n*)	*O*(1)
`DelayQueue`	*O*(log *n*)	*O*(1)	*O*(log *n*)	*O*(1)
`TransferQueue`	*O*(1)	*O*(1)	*O*(1)	*O*(n)

`LinkedList`	*O*(1)	*O*(1)	*O*(1)	*O*(1)
`ArrayDeque`	*O*(1)	*O*(1)	*O*(1)	*O*(1)
`LinkedBlockingDeque`	*O*(1)	*O*(1)	*O*(1)	*O*(1)

question is whether you need blocking methods, as you usually will for producer-consumer problems (either because the consumers must handle an empty queue by waiting, or because you want to constrain demand on them by bounding the queue, and then producers must sometimes wait). If you don't need blocking methods or a bound on the queue size, choose the efficient and wait-free `ConcurrentLinkedQueue`.

If you do need a blocking queue, because your application requires support for producer-consumer cooperation, pause to think whether you really need to buffer data, or whether all you need is to safely hand off data between the threads. If you can do without buffering (usually because you are confident that there will be enough consumers to prevent data from piling up), then `SynchronousQueue` is an efficient alternative to the remaining FIFO blocking implementations, `LinkedBlockingQueue` and `ArrayBlockingQueue`.

Otherwise, we are finally left with the choice between these two. If you cannot fix a realistic upper bound for the queue size, then you must choose `LinkedBlockingQueue`, as `ArrayBlockingQueue` is always bounded. For bounded use, you will choose between the two on the basis of performance. Their performance characteristics in Table 9-1 are the same, but these are only the formulae for sequential access; how they perform in concurrent use is a different question. As we mentioned above, `LinkedBlockingQueue` performs better on the whole than `ArrayBlockingQueue` if more than three or four threads are being serviced. This fits with the fact that the head and tail of a `LinkedBlockingQueue` are locked independently, allowing simultaneous updates of both ends. On the other hand, an `ArrayBlockingQueue` does not have to allocate new objects with each insertion. If queue performance is critical

to the success of your application, you should measure both implementations with the benchmark that means the most to you: your application itself.

Chapter 10. Lists

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 16th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

Lists are probably the most widely used Java collections in practice. A *list* is a collection which—unlike a set—can contain duplicates, and which—unlike a queue—gives the user full visibility and control over the ordering of its elements. The corresponding Collections Framework interface is `List<E>`.

The `List` interface exposes methods for positional access, for searching for a given value, for generating views, and for creating `ListIterator`s—a subtype of `Iterator` with additional features that take advantage of a `List`'s sequential nature. In addition, the methods inherited from `SequencedCollection` provide convenient shorter versions of common positional access calls. Finally, static factory methods are available to create unmodifiable `List`s of different lengths.

Positional Access

These methods access elements based on their numerical position in the list.

```
void add(int index, E e)           // adds element e at given index
boolean addAll(int index, Collection<? extends E> c)
                                   // adds contents of c at given
index
E get(int index)                   // returns element at given index
```

```
E remove(int index)                 // removes element at given index
E set(int index, E e)               // replaces element at given index
by e
```

Search Methods

These methods search for a specified object in the list and return its numerical position, or -1 if the object is not present:

```
int indexOf(Object o)              // returns index of first occurrence
of o
int lastIndexOf(Object o)          // returns index of last occurrence
of o
```

View-Generating Methods

These methods provide different views (see "Views") of a `List`:

```
List<E> subList(int fromIndex, int toIndex)
                                    // returns a view of a portion of
the list
List<E> reversed()                  // provides a reverse-ordered view
of the
                                    // original collection
```

The method `subList` works in a similar way to the `subSet` operations on `SortedSet` (see "NavigableSet"), but uses the position of elements in the list rather than their values: the returned list contains the list elements starting at `fromIndex`, up to but not including `toIndex`. The returned list has no separate existence; it is just a view of part of the list from which it was obtained, and all changes in it are reflected in the original list. There is an important difference from `subSet`, though; changes made to the sublist write through to the backing list, but the reverse is true only for non-structural changes (see Ch. 11 section on structural/non-structural modification). If any structural changes are made to the backing list by inserting or removing elements from it, subsequent attempts to use the sublist will result in a `ConcurrentModificationException`.

The view returned by `reversed` allows any modifications that are permitted by the original list.

List Iteration

These methods return a `ListIterator`, which is an `Iterator` with extended semantics that take advantage of the list's sequential nature:

```
ListIterator<E> listIterator()     // returns a ListIterator for this
list,
                                   // initially positioned at index 0
ListIterator<E> listIterator(int indx)
                                   // returns a ListIterator for this
list,
                                   // initially positioned at index
indx
```

The methods added by `ListIterator` support traversing a list in reverse order, changing list elements or adding new ones, and getting the current position of the iterator. The current position of a `ListIterator` always lies between two elements, so in a list of length *n*, there are *n*+1 valid list iterator positions, from 0 (before the first element) to *n* (after the last one). A call of the first overload returns a `listIterator` set at position 0. The second overload uses the supplied value to set the initial position of the returned `listIterator`.

In addition to the `Iterator` methods `hasNext`, `next`, and `remove`, `ListIterator` exposes the following methods:

```
public interface ListIterator<E> extends Iterator<E> {
  void add(E e);          // inserts the specified element into the
list
  boolean hasPrevious();  // returns true if this list iterator has
further
                          // elements in the reverse direction
  int nextIndex();        // returns the index of the element that
would be
                          // returned by a subsequent call to next
```

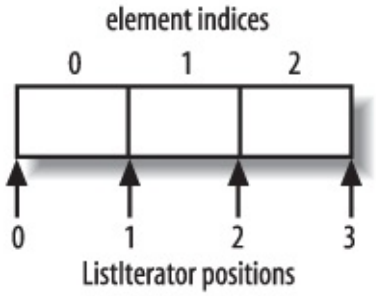

Figure 10-1. ListIterator positions

```
  E previous();           // returns the previous element in the list
  int previousIndex();    // returns the index of the element that
would be
                          // returned by a subsequent call to previous
  void set(E e);          // replaces the last element returned by
next or
                          // previous with the specified element
}
```

Figure 10-1 shows a list of three elements. Consider an iterator at position 2, either moved there from elsewhere or created there by a call to `listIterator(2)`. The effect of most of the operations of this iterator is intuitive; `add` inserts an element at the current iterator position (between the elements at index 1 and 2), `hasPrevious` and `hasNext` return `true`, `previous` and `next` return the elements at indices 1 and 2 respectively, and `previousIndex` and `nextIndex` return those indices themselves. At the extreme positions of the list, 0 and 3 in the figure, `previousIndex` and `nextIndex` would return -1 and 3 (the size of the list) respectively, and `previous` or `next`, respectively, would throw `NoSuchElementException`.

The operations `set` and `remove` work differently. Their effect depends not on the current position of the iterator, but on its "current element", the one last traversed over using `next` or `previous`: `set` replaces the current element, and `remove` removes it. If there is no current element, either because the iterator has just been created, or because the current element has been removed, these methods will throw `IllegalStateException`.

Methods Inherited from `SequencedCollection`

Apart from the method `reversed`, the methods inherited from `SequencedCollection` are convenience versions of positional methods already present in `List`. Table 10-1 below shows the correspondence between calls of `SequencedCollection` and those of the positional access methods of `List`.

Table 10-1. `SequencedCollection` method calls, with their `List` equivalents

`SequencedCollection` call	`List` positional access call
`addFirst(el)`	`add(0, el)`
`addLast(el)`	`add(el)`
`getFirst()`	`get(0)`
`getLast()`	`get(size() - 1)`
`removeFirst()`	`remove(0)`
`removeLast()`	`remove(size() - 1)`

Factory Methods

These methods create unmodifiable `List` objects (see Ch. 11 section on unmodifiable collections). Like all factory methods they are static, so for conciseness this modifier is again omitted from the declarations below.

```
<E> List<E> of() // Returns an unmodifiable list containing zero
elements.
<E> List<E> of(E e1) // Returns an unmodifiable list containing one
element.
<E> List<E> of(E e1, E e2) // Returns an unmodifiable list containing
two elements
<E> List<E> of(E e1, E e2, E e3) // Returns an unmodifiable list
containing three
                                        // elements
```

```
<E> List<E> of(E e1, E e2, E e3, E e4) // Returns an unmodifiable list
containing four
                                       // elements
<E> List<E> of(E e1, E e2, E e3, E e4, E e5) // Returns an
unmodifiable list containing
                                       // five elements
<E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6) // Returns an
unmodifiable list
                                       // containing six elements
<E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7) // Returns an
unmodifiable
                                       // list containing seven
elements
<E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8) //
Returns an
                                       // unmodifiable list
containing eight elements
<E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)
// Returns an
                                       // unmodifiable list
containing nine elements
<E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E
e10) // Returns an
                                       // unmodifiable list
containing ten elements
<E> List<E> of(E... elements) // Returns an unmodifiable list
containing an arbitrary
                                       // number of elements}
```

Using the Methods of `List`

Let's look at examples of using some of these methods in the to-do manager. In the last chapter we considered representing the organization of a single day's tasks in a queue-based class with shutdown capabilities. One useful way of enlarging the scope of the application is to have a number of objects of this type, each one representing the tasks that are scheduled for a day in the future. We can store references to these objects in a `List`, which (to keep things simple and to avoid getting involved with the details of `java.time`) will be indexed on the number of days in the future that it represents. So the queue of tasks scheduled for today will be stored at element 0 of the list, the queue scheduled for tomorrow at element 1, and so on. Example 10-1 shows the scheduler.

Example 10-1. A list-based task scheduler

```
public class TaskScheduler {
```

```
private final List<StoppableTaskQueue> schedule;
private final int FORWARD_PLANNING_DAYS = 365;

    public TaskScheduler() {
        schedule = Stream.generate(StoppableTaskQueue::new)
                .limit(FORWARD_PLANNING_DAYS)

.collect(Collectors.toCollection(CopyOnWriteArrayList::new));
    }

    public Optional<PriorityTask> getTopTask() {
        return schedule.stream()
                .map(StoppableTaskQueue::getFirstTask)
                .filter(Objects::nonNull)
                .findFirst();
    }

    // at midnight, create a new day's queue at the planning horizon,
    // remove and shut down the queue for day 0, and assign its tasks
    // to the new day 0
    public void rollOver() {
        schedule.add(new StoppableTaskQueue());
        StoppableTaskQueue oldDay = schedule.removeFirst();
        schedule.getFirst().addTasks(oldDay.shutDown());
    }

    public void addTask(PriorityTask task, int day) {
        addTasks(Collections.singletonList(task), day);
    }

    public void addTasks(Collection<PriorityTask> tasks, int day) {
        if (day < 0 || day >= FORWARD_PLANNING_DAYS)
            throw new IllegalArgumentException("day out of range");

        // if addTasks() fails, shutdown has been initiated on this
        // queue. So it must be the day 0 queue that has already been
        // removed from the schedule. The correct action, then, is to
        // call addTasks() on the new day 0 queue.
        while (! schedule.get(day).addTasks(tasks));
    }
}
```

Although the example aims primarily to show the use of `List` interface methods rather than to explore any particular implementation, we can’t set it up without choosing one. Since a major factor in the choice will be the concurrency requirements of the application, we need to consider them now. They are quite straightforward: clients consuming or producing tasks only ever read the `List`

representing the schedule so, once it has been constructed, it is only ever written at the end of each day. At those points the current day's queue is removed from the schedule, and a new one is added at the end (at the "planning horizon", which we have set to a year in the example). We don't need to exclude clients from using the current day's queue before that happens, because the `StoppableTaskQueue` design of Example14.1 ensures that they will be able to complete in an orderly way once the queue is stopped. So the only exclusion required is to ensure that clients don't try to read the schedule itself while the rollover procedure is changing its values.

If you recall the discussion of `CopyOnWriteArrayList` in "Java 5: Concurrent Collections", you'll see that it fills these requirements very nicely. It optimizes read access, in line with one of our requirements. And in the event of a write operation, it synchronizes just long enough to create a new copy of its internal backing array, thus filling our other requirement of preventing interference between read and write operations.

With the implementation chosen, we can understand the constructor of Example15.1; writing to the list is expensive, so it is sensible to use a conversion constructor to set it up with a year's worth of task queues in one operation.

The `getTask` method is straightforward; we simply iterate over the task queues, starting with today's queue, looking for a scheduled task. If the method finds no outstanding tasks, it returns `null`—and if finding a task-free day was noteworthy, how should we celebrate a task-free year?

At midnight each day, the system will call the method `rollOver`, which implements the sad ritual of shutting down the old day's task queue and transferring the remaining tasks in it to the new day. The sequence of events here is important; `rollOver` first removes the queue from the list, at which time producers and consumers may still be about to insert or remove elements. It then calls `StoppableTaskQueue.shutDown` which, as we saw in Example 9-1, returns the remaining tasks in the queue and guarantees that no more will be added. Depending on how far they have progressed, calls of `addTask` will either complete or will return `false`, indicating that they failed because the queue was shut down.

This motivates the logic of `addTask`: the only situation in which the `addTask`

method of `StoppableTaskQueue` can return `false` is that in which the queue being called is already stopped. Since the only queue that is stopped is the day 0 queue, a return value of `false` from `addTask` must result from a producer thread getting a reference to this queue just before a midnight rollover. In that case, the current value of element 0 of the list is by now the new day 0, and it is safe to try again. If the second attempt fails, the thread has been suspended for 24 hours!

Notice that the `rollOver` method is quite expensive; it writes to the schedule twice, and since the schedule is represented by a `CopyOnWriteArrayList` (see "`CopyOnWriteArrayList`"), each write causes the entire backing array to be copied. The argument in favour of this implementation choice is that `rollOver` is very rarely invoked compared to the number of calls made on `getTask`, which iterates over the schedule. The alternative to `CopyOnWriteArrayList` would be a `BlockingQueue` implementation, but the improvement that would provide in the rarely-used `rollOver` method would come at the cost of slowing down the frequently-used `getTask` method, since queue iterators are not intended to be used in performance-critical situations.

Using Range-View and Iterator Methods Of the four `List` method groups above, Example 10-1 makes use of the methods of one group, positional access, in several places. To see how range-view and iterator methods could also be useful, consider how the `TaskScheduler` could export its schedule, or a part of it, for a client to modify. You would want the client to be able to view this subschedule and perhaps to insert or remove tasks, but you would definitely want to forbid the insertion or removal of elements of the list itself, since these represent the sequence of days for which tasks are being scheduled. The standard way to achieve this would be by means of an unmodifiable list, as provided by the `Collections` class (see "Unmodifiable Collections"). An alternative in this case would be to return a list iterator, as the snapshot iterators for copy-on-write collections do not support modification of the backing collection. So we could define a method to provide clients with a "planning window":

```
listIterator<StoppableTaskQueue> getSubSchedule(int startDay, int
endDay) {
  return schedule.subList(startDay, endDay).listIterator();
```

```
}
```

This view will be fine for today, but we have to remember to discard it at midnight, when the structural changes of removing and adding entries will invalidate it.

`List` Implementations

There are three concrete implementations of `List` in the Collections Framework (see Figure 10-2), differing in how fast they perform the various operations defined by the interface and how they behave in the face of concurrent modification; unlike `Set` and `Queue`, however, `List` has no subinterfaces to specify differences in functional behavior. In this and the following section we look at each implementation in turn and provide a performance comparison.

`ArrayList`

Arrays are provided as part of the Java language and have a very convenient syntax, but their key disadvantage—that, once created, they cannot be resized—makes them increasingly less popular than `List` implementations, which (if resizable at all) are indefinitely extensible. The most commonly used implementation of `List` is, in fact, `ArrayList`—that is, a `List` backed by an array.

The standard implementation of `ArrayList` stores the `List` elements in contiguous array locations, with the first element always stored at index 0 in the array. It requires an array at least large enough (with sufficient *capacity*) to contain the elements, together with a way of keeping track of the number of "occupied" locations (the size of the `List`). If an `ArrayList` has grown to the point where its size is equal to its capacity, attempting to add another element will require it to replace the backing array with a larger one capable of holding the old contents and the new element, and with a margin for further expansion (the standard implementation actually uses a new array that is double the length of the old one). As we explained in [Link to Come], this leads to an amortized cost of $O(1)$.

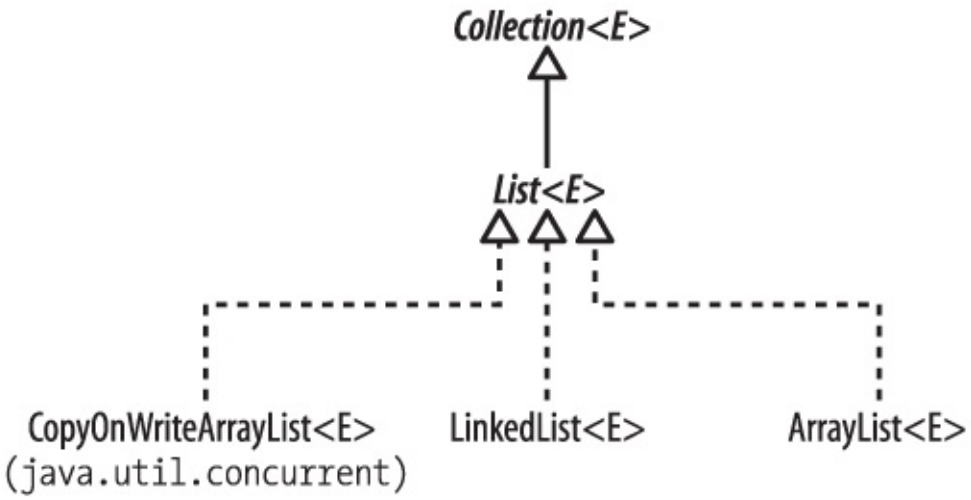

Figure 10-2. Implementations of the List interface

The performance of `ArrayList` reflects array performance for "random-access" operations: `set` and `get` take constant time. The downside of an array implementation is in inserting or removing elements at arbitrary positions, because that may require adjusting the position of other elements. (We have already met this problem with the `remove` method of the iterators of array-based queues—for example, `ArrayBlockingQueue` (see "BlockingQueue Implementations"). But the performance of positional add and remove methods are much more important for lists than `iterator.remove` is for queues.)

For example, Figure 10-3(a) shows a new `ArrayList` after three elements have been added by means of the following statements:

```
List<Character>; charList = new ArrayList<Character>();
Collections.addAll(charList,  'a', 'b', 'c');
```

If we now want to remove the element at index 1 of an array, the implementation must preserve the order of the remaining elements and ensure that the occupied region of the array is still to start at index 0. So the element at index 2 must be moved to index 1, that at index 3 to index 2, and so on. Figure 10-3(b) shows our sample `ArrayList` after this operation has been carried out. Since every element must be moved in turn, the time complexity of this operation is proportional to the size of the list (even though, because this operation can usually be implemented in hardware, the constant factor is low).

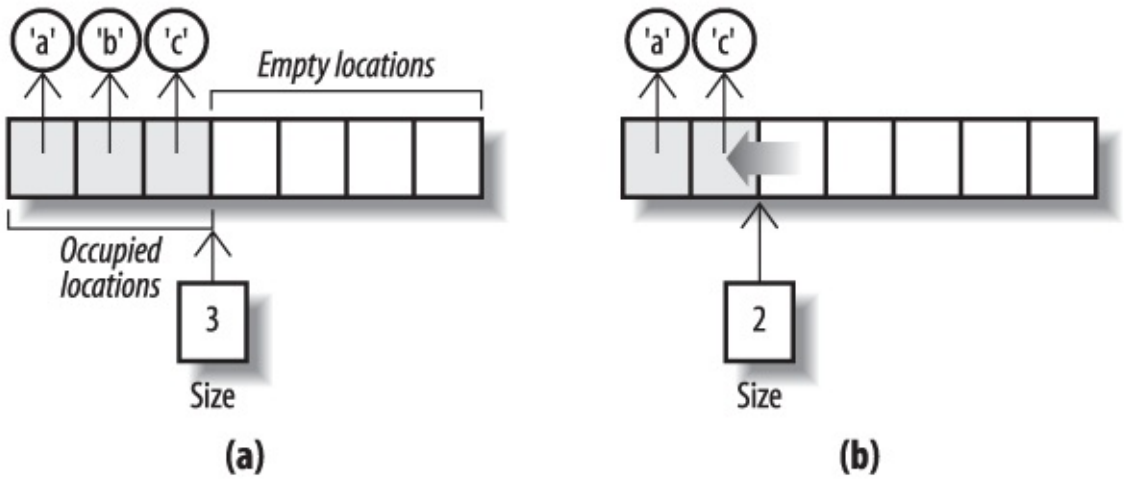

Figure 10-3. Removing an element from an ArrayList

Even so, the alert reader, recalling the discussion of the circular arrays used to implement `ArrayBlockingQueue` and `ArrayDeque` (see "Deque Implementations") may wonder why a circular array was not chosen for the implementation of `ArrayList`, too. It is true that the `add` and `remove` methods of a circular array show much better performance only when they are called with an index argument of 0, but this is such a common case and the overhead of using a circular array is so small, that the question remains.

Indeed, an outline implementation of a circular array list was presented by Heinz Kabutz in *The Java Specialists' Newsletter* (*http://www.javaspecialists.co.za/archive/Issue027.xhtml*). In principle it is still possible that `ArrayList` may be reimplemented in this way, possibly leading to real performance gains in many existing Java applications. A possible alternative is that the circular `ArrayDeque` may be retrofitted to implement the methods of `List`. In the meantime, if your application is using a `List` in which the performance of element insertion and removal from the beginning of a list is more important than that of randomaccess operations, consider writing to the `Deque` interface and taking advantage of its very efficient `ArrayDeque` implementation.

As we mentioned in the discussion of `ArrayBlockingQueue` ("Queue Implementations"), variable-size array-backed collection classes can have one configuration parameter: the initial length of the array. So besides the standard Collections Framework constructors, `ArrayList` has one that allows you to

choose the value of the initial capacity to be large enough to accommodate the elements of the collection without frequent create-copy operations. The initial capacity of an `ArrayList` created by the default constructor is 10, and that of one initialized with the elements of another collection is 110% of the size of that collection.

The iterators of `ArrayList` are fail-fast.

`LinkedList`

We discussed `LinkedList` as a `Deque` implementation in "Deque Implementations". You will avoid it as a `List` implementation if your application makes much use of random access; since the list must iterate internally to reach the required position, positional `add` and `remove` have linear time complexity, on average. Where `LinkedList` does have a performance advantage over `ArrayList` is in adding and removing elements anywhere other than at the end of the list; for `LinkedList` this takes constant time, against the linear time required for noncircular array implementations.

`CopyOnWriteArrayList`

In "Set Implementations" we met `CopyOnWriteArraySet`, a set implementation designed to provide thread safety together with very fast read access. `CopyOnWriteArrayList` is a `List` implementation with the same design aims. This combination of thread safety with fast read access is useful in some concurrent programs, especially when a collection of observer objects needs to receive frequent event notifications. The cost is that the array which backs the collection has to be treated as immutable, so a new copy is created whenever any changes are made to the collection. This cost may not be too high to pay if changes in the set of observers occur only rarely.

The class `CopyOnWriteArraySet` in fact delegates all of its operations to an instance of `CopyOnWriteArrayList`, taking advantage of the atomic operations `addIfAbsent` and `addAllAbsent` provided by the latter to enable the `Set` methods `add` and `addAll` to avoid introducing duplicates to the set. In addition to the two standard constructors (see "Collection Constructors"), `CopyOnWriteArrayList` has an extra one that allows it to

be created using the elements of a supplied array as its initial contents. Its iterators are snapshot iterators, reflecting the state of the list at the time of their creation.

Comparing `List` Implementations

Table 10-2 gives the comparative performance for some sample operations on `List` classes. Even though the choice here is much narrower than with queues or even sets, the same process of elimination can be used. As with queues, the first question to ask is whether your application requires thread safety. If so, you should use `CopyOnWriteArrayList`, if you can—that is, if writes to the list will be relatively infrequent. If not, you will have to use a synchronized wrapper (see "Synchronized Collections") around `ArrayList` or `LinkedList`.

For most list applications the choice is between `ArrayList` and `LinkedList`, synchronized or not. Once again, your decision will depend on how the list is used in practice. If `set` and get predominate, or element insertion and removal is mainly at the end of the list, then `ArrayList` will be the best choice. If, instead, your application needs to frequently insert and remove elements near the start of the list as part of a process that uses iteration, `LinkedList` may be better. If you are in doubt, test the performance with each implementation. A Java 6 alternative for single-threaded code that may be worth considering in the last case—if the insertions and removals are actually *at* the start of the list—is to write to the `Deque` interface, taking advantage of its very efficient `ArrayDeque` implementation. For relatively infrequent random access, use an iterator, or copy the `ArrayDeque` elements into an array using `toArray`.

Table 10-2. Comparative performance of different list implementations

	get	add	contains	next
`ArrayList`	*O*(1)	*O*(1)	*O*(*n*)	*O*(1)
`LinkedList`	*O*(*n*)	*O*(1)	*O*(*n*)	*O*(1)

`CopyOnWriteArrayList`	*O*(1)	*O*(*n*)	*O*(*n*)	*O*(1)

It is possible that, in a future release, `ArrayDeque` will be retrofitted to implement the `List` interface; if that happens, it will become the implementation of choice for both `Queue` and `List` in single-threaded environments.

Chapter 11. Maps

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 17th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at shunter@oreilly.com.

The `Map` interface is the last of the major Collections Framework interfaces, and the only one that does not inherit from `Collection`. A `Map` stores key-to-value associations, or *entries*, in which the keys are unique. The importance of this interface lies in the fact that its implementations provide very fast—ideally, constant-time—operations to look up the value corresponding to a given key.

The operations of `Map` can be divided according to the two ways in which a map can be seen: as a set of entries or as a lookup mechanisms. From the first viewpoint, the first two sets of operations to consider correspond broadly to the operations of `Iterable`—performing an action on each key-value pair in turn, and `Collection`—adding or modifying entries, removing them, or querying the contents of the collection. Later we'll see two other important operation groups: compound operations—seeing maps very much from the second viewpoint—and factory methods.

Where `Map` operations accept lambdas as parameters, the types of the parameters to these lambdas are often bounded, above or below, by the key or value type. Similarly, `Map.Entry` is usually parameterized on bounded types. In this section, however, the listings provide the precise types instead, in the interests of brevity.

`Iterable`–like Operations

```
void forEach(Consumer<K,V> action)
                                  // perform action on each entry in the
map, in the
                                  // iteration order of the entry set,
if that is specified
```

`Collection`–like Operations

These are broadly parallel to the first three operation groups of `Collection`: adding or modifying entries, removing entries, and querying map contents

Adding or Modifying Associations:

```
V put(K key, V value)             // add or replace a key-value
association,
                                  // returning the old value if the
                                  // key was present; otherwise return
null
void putAll(Map<K,V> m)           // copy all the mappings in m into
the receiver
void replaceAll(BiFunction<K,V,V> remapper)
                                  // replace each value with the result
of invoking remapper
```

Removing Associations:

```
void clear()                      // remove all associations from this
map
V remove(Object key)              // remove the association, if any,
with the
                                  // given key; return the value with
which it
                                  // was associated if any, otherwise
null
```

Querying the Contents of a Map:

```
V get(Object k)                   // return the value corresponding to
k, or
                                  // null if k is not present as a key
boolean containsKey(Object k)     // return true if k is present as a
key
boolean containsValue(Object v) // return true if v is present as a
```

```
value
int size()                      // return the number of associations
boolean isEmpty()               // return true if there are no
associations
```

Providing Collection Views of the Keys, Values, or Associations:

```
Set<Map.Entry<K,V>> entrySet() // return a Set view of the
associations
Set<K> keySet()                 // return a Set view of the keys
Collection<V> values()          // return a Collection view of the
values
```

The view collections returned by these methods are backed by the map, so they reflect changes to the map. The connection in the opposite direction is more limited: you can remove elements from the view collections, but attempting to add them will result in an `UnsupportedOperationException`. Removing a key from the keyset removes the single corresponding key-value association; removing a value, on the other hand, removes only one of the associations mapping to it; the value may still be present as part of an association with a different key. An iterator over the view will become undefined if the backing map is concurrently modified.

Map.Entry

The members of the set returned by `entrySet` implement the interface `Map.Entry`, representing a key-value association. This interface exposes factory methods for creating `Comparator`s by key and value, and instance methods for accessing the components of the entry. An optional `setValue` method can be used to change the value in an entry if the backing map is modifiable and, if so, will write changes through. According to the Javadoc, a `Map.Entry` object obtained by iterating over a set returned by `entrySet` retains its connection only for the duration of the iteration, but in fact this is only a guaranteed minimum: implementation behaviors vary, some preserving the connection indefinitely. However, modern idioms for map processing depend less than before on durable `Map.Entry` objects: where in the past you might

have used a `Map.Entry` set iterator to remove some entries, you would now be more likely to call `entrySet.removeIf`. Alternatively, and also for use-cases in which you want to change association values, the instance methods of `Map`, listed in this section, provide a variety of ways to remove mappings or alter their values.

You can create `Map.Entry` objects using the `Map.entry` factory method; this is most commonly useful for creating unmodifiable `Map`s (see ???).

Compound Operations

Because `Maps` are very often used in multi-threaded environments, the interface exposes a variety of compound actions. These fuse a conditional test—whether a key is absent or present, possibly with a specific value—either with a supplied value or with an action, represented by a lambda, to compute the value lazily. They are essential for use with concurrent implementations to avoid unsafe test-then-act operation sequences as seen in "Java 2: Synchronized Collections and Fail-Fast Iterators", but also extremely useful as convenience methods for non-thread-safe maps.

```
V getOrDefault(Object k, V default)
                                    // return the value to which k is
mapped, or default
                                    // if this map contains no mapping
for the key
```

If a map has values only for certain keys, this method allows you to use a default value for all other keys without having to store that value in the map against them all.

```
V putIfAbsent(K key, V value)    // create a key-value mapping if the
key is
                                    // absent or mapped to null; in these
cases,
                                    // return null, otherwise return the
current value
```

This method is useful if you want to write something the first time you see it but not thereafter. For example, to record the timestamp corresponding to the first occurrence of a particular kind of event, you could write:

```
Map<EventKind,Long> firstOccurrenceMap = ... ;
...
firstOccurrenceMap.putIfAbsent(event.getKind(),
System.currentTimeMillis());
```

In this example, the overhead of boxing the timestamp into a `Long` will be incurred for every event, not only the first. You could avoid this performance cost using another compound method, `computeIfAbsent`, which computes the new value lazily:

```
firstOccurrenceMap.computeIfAbsent(event.getKind(), key ->
System.currentTimeMillis());
```

This is the declaration of `computeIfAbsent`:

```
V computeIfAbsent(K k, Function<K,V> mapper)
                                    // apply the mapper function to k and
use the
                                    // result to create a key-value pair,
unless
                                    // either k is currently mapped to a
non-null
                                    // value or the result is null.
Return the value
                                    // now associated with k, or null if
k is not now
                                    // present in the map
```

The Javadoc for `computeIfAbsent` offers an example of its most common use, creating a map from a key to a list (or other accumulation) of multiple values:

```
map.computeIfAbsent(key, k -> new ArrayList<V>()).add(newValue);
```

In a different scenario, you might want to accumulate the value by means of an operator, like addition or concatentation, that combines the old and newly-supplied values, to create a new one to store in the map. The most useful method in that scenario is merge:

```
V merge(K k, V newValue, BiFunction<V,V,V> remapper)
                                    // if k is not present or is mapped
to null, associate it
```

```
                                    // with newValue. Otherwise, apply
remapper to existing
                                    // value and newValue; associate k
with the result if it is
                                    // non-null, otherwise remove k
```

The Javadoc for `merge` gives this example, to either initialise an entry value to a given string `msg`, or to append a new value to an existing one:

```
 map.merge(key, msg, String::concat);
```

The next method, `compute`, is similar to `merge`, but with the difference that instead of an initial value it allows the contents of the key to be used in a lazy computation of the new value:

```
V compute(K k, BiFunction<K,V,V> remapper)
                                      // use the result computed by
remapper from k and its
                                      // existing value (or null if k is
not present) to create
                                      // or modify a key-value pair. If
the computed value is
                                      // null, remove any existing entry.
Return the new value,
                                      // or null if there is none
```

For example, suppose we want to map each word in a text to the total character count that all its occurrences contribute to the text length:

```
map.compute(word, (s,i) -> s.length() + (i == null ? 0 : i));
```

The last member of the `compute*` family is `computeIfPresent`:

```
V computeIfPresent(K k, BiFunction<K,V,V> remapper)
                                      // if k is currently mapped to a
non-null value, use the result
                                      // computed by remapper from k and
the existing value to replace that
                                      // value, and return the result. If
the computed result is null,
                                      // remove the existing entry and
return null
```

In the same way that `computeIfAbsent` is most useful where it may be necessary to add a new key, `computeIfPresent` can be used to remove an existing one. In an earlier example, we saw how to use `putIfAbsent` to produce a map from each kind of event to the timestamp of its first occurrence. Now suppose that we later want to process a different phase of the event stream so that the first event—and only the first—of each kind in this later phase triggers the writing of a log entry with the previously recorded timestamp:

```
firstOccurrenceMap.computeIfPresent(event.getKind(),(kind,timestamp) -
> {
    log.info("first occurrence of event " + kind + " was at " +
timestamp);
    return null;
});
```

Returning `null` from the lambda ensures that the key will be removed from the map, so that the log message can only be triggered once for each kind of event.

Commentary and examples to be supplied for last three methods:

```
V replace(K k, V newValue)         // replace existing value for k,
provided k is currently in the
                                   // map. Return the old value, or
null if k was not present
boolean replace(K k, V oldValue, V newValue)
                                   // replace existing value for k,
provided it is currently
                                   // mapped to oldValue. Return true
if oldValue was replaced
boolean remove(Object key, Object value)
                                   // remove the entry for this key if
it is mapped to
                                   // this value, returning true if it
succeeded
```

Factory Methods

These methods create unmodifiable `Map` objects (see Ch. 11), which we are calling `UnmodifiableMap<K,V>` in “UnmodifiableMap”, where the factory methods are also discussed.

Using the Methods of Map

One problem with basing the to-do manager on priority queues, as we have done in the last two chapters, is that priority queues are unable to preserve the order in which mappings are added to them (unless that can be incorporated in the priority ordering, for example as a timestamp or sequence number). To avoid this, we could use as an alternative model a series of FIFO queues, each one assigned to a single priority. A `Map` would be suitable for holding the association between priorities and task queues; `EnumMap` in particular is a highly efficient `Map` implementation specialized for use with keys which are members of an `enum`.

This model will rely on a `Queue` implementation that maintains FIFO ordering. To focus on the use of the `Map` methods, let's assume a single-threaded client and use a series of `ArrayDeques` as the implementation:

```
var taskMap = new EnumMap<Priority,Queue<Task>>(Priority.class);
for (Priority p : Priority.values()) {
  taskMap.put(p, new ArrayDeque<Task>());
}
// populate the lists, for example:
taskMap.get(Priority.MEDIUM).add(mikePhone);
taskMap.get(Priority.HIGH).add(databaseCode);
```

Now, to get to one of the task queues—say, the one with the highest-priority tasks—we can write:

```
Queue<Task> highPriorityTaskList = taskMap.get(Priority.HIGH);
```

Polling this queue will now give us the high priority to-dos, in the order in which they were entered into the system.

To see the use of some other methods of `Map`, let's extend the example a little to allow for the possibility that some of these tasks might actually earn us some money by being billable. One way of representing this would be by defining a class `Client`:

```
class Client {...}
Client acme = new Client("Acme Corp.",...);
```

and creating a mapping from tasks to clients:

```
Map<Task,Client> billingMap = new HashMap<Task,Client>();
billingMap.put(interfaceCode, acme);
```

Only billable tasks will appear in the domain of `billingMap`. Now, as part of the code for processing a task `t`, we can write:

```
Task t = ...
Client client = billingMap.get(t);
if (client != null) {
  client.bill(t);
}
```

When we have finally finished all the work we were contracted to do by our client Acme Corp., we have a couple of options for removing the map entries that associate tasks with Acme:

```
billingMap.values().removeAll(List.of(acme));
clients.removeIf(client -> client.equals(acme));
```

The `values` view of `billingMap` can contain duplicate clients, if there are multiple tasks associated with the same client. If we want to find the set of clients that need to be billed, we can copy and deduplicate the clients from the `values` view using a `Set` constructor:

```
Set<Client> billedClients = new HashSet<>(billingMap.values());
```

Extending the example further, suppose you have created a set of tasks that can only be accomplished when you're on-site at the client:

```
Set<Task> onsiteTasks = ... ;
```

Now you want to determine which clients have on-site, billable tasks, so that you can schedule visits to them. You could of course build the set of on-site clients by iterating the entry set obtained from `billingMap`, including the client extracted from each entry if the corresponding task was contained in `onsiteTasks`. However, it's more concise, and a more modern idiom, to copy

`billingMap` then use a destructive bulk operation, `retainAll`, on its key set, then deduplicating the clients as above:

```
Map<Task,Client> onsiteMap = new HashMap<>(billingMap);
onsiteMap.keySet().retainAll(onsiteTasks);
Set<Client> clientsToVisit = new HashSet<>(onsiteMap.values());
```

> Streaming the entry set is also a common idiom. For example, you could create a map associating each client with the list of their onsite tasks:
>
> ```
> Map<Client,List<Task>> tasksByClient =
>     billingMap.entrySet().stream()
>         .filter(entry -> onsiteTasks.contains(entry.getKey()))
>         .collect(Collectors.groupingBy(Map.Entry::getKey));
> ```

`Map` Implementations

The implementations, nine in all, that the Collections Framework provides for `Map` are shown in Figure 11-1. We shall discuss `HashMap`, `LinkedHashMap`, `WeakHashMap`, `IdentityHashMap`, `EnumMap`, and `UnmodifiableMap` here; the interfaces `NavigableMap`, `ConcurrentMap`, and `ConcurrentNavigableMap` are discussed, along with their implementations, in the sections following this one.

For constructors, the general rule for `Map` implementations is like that for `Collection` implementations (see "Collection Constructors"). Every implementation excluding `EnumMap` has at least two constructors; taking `HashMap` as an example, they are:

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

The first of these creates an empty map, and the second a map that will contain the key-value mappings contained in the supplied map `m`. The keys and values of map `m` must have types that are the same as (or are subtypes of) the keys and

values, respectively, of the map being created. Using this second constructor has the same effect as creating an empty map with the default constructor, and then adding the contents of map `m` using `putAll`. In addition to these two constructors, `HashMap`, `LinkedHashMap`, and `WeakHashMap` have other constructors for configuration purposes, but these have been replaced in practice by factory methods, for the reasons described in ???).

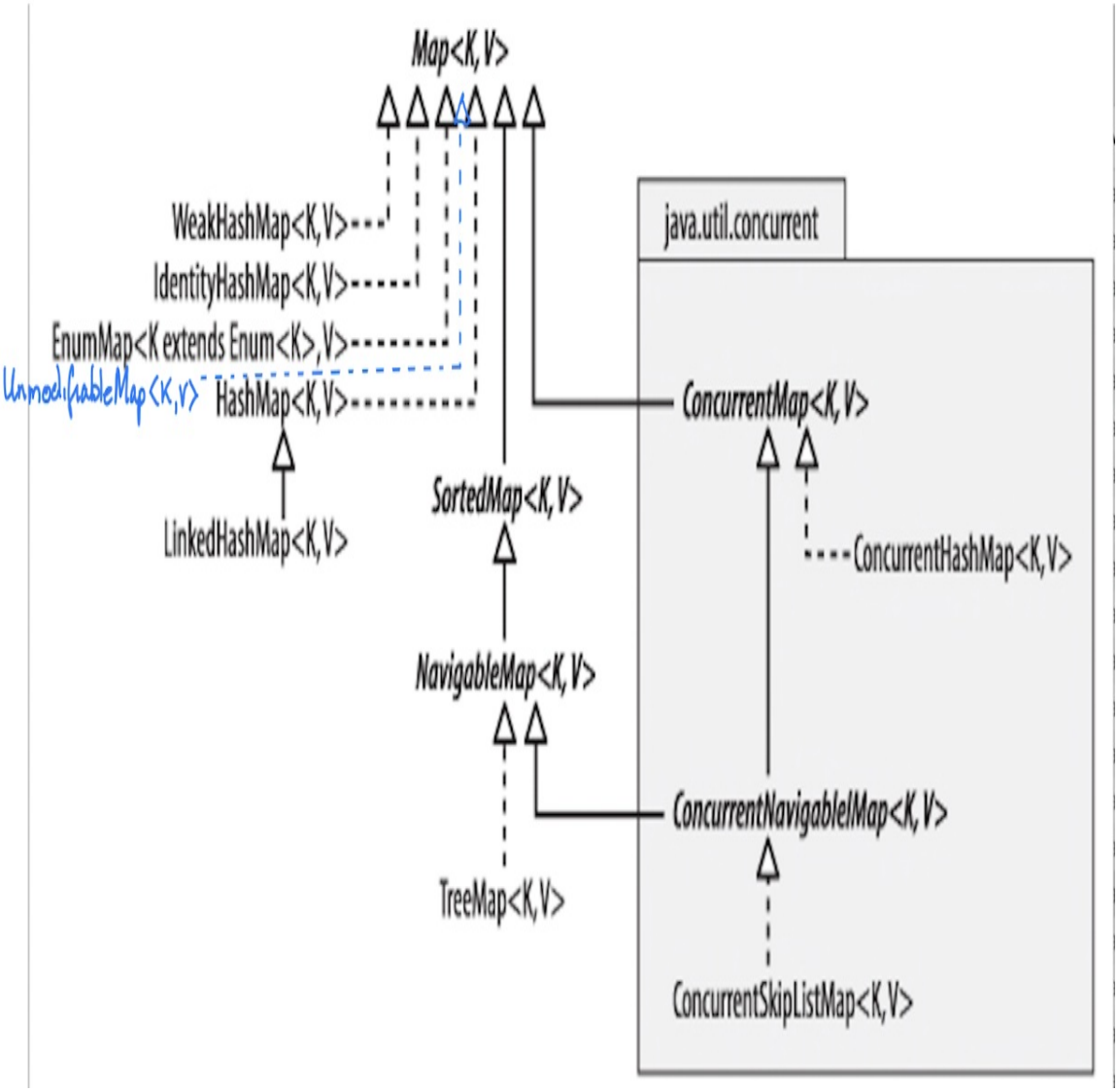

Figure 11-1. The structure of Map implementations in the Collections Framework

HashMap

In discussing `HashSet` in "HashSet", we mentioned that it delegates all its operations to a private instance of `HashMap`. Figure 11-2(a) is similar to Figure 8-1, but without the simplification that removed the value elements from the map (all elements in a `HashSet` are stored as keys with the same constant value). The discussion in "Set Implementations" of hash tables and their performance applies equally to `HashMap`. In particular, `HashMap` provides constant-time performance for `put` and `get`. Although in principle constant-time performance is only attainable with no collisions, it can be closely approached by the use of rehashing to control the load and thereby to minimize the number of collisions.

Iteration over a collection of keys or values requires time proportional to the capacity of the map plus the number of key-value mappings that it contains. The iterators are fail-fast.

The factory method to use when you can predict the maximum number of entries is `newHashMap`:

```
static <K,V>> HashMap<K,V> newHashMap(int numElements)
```

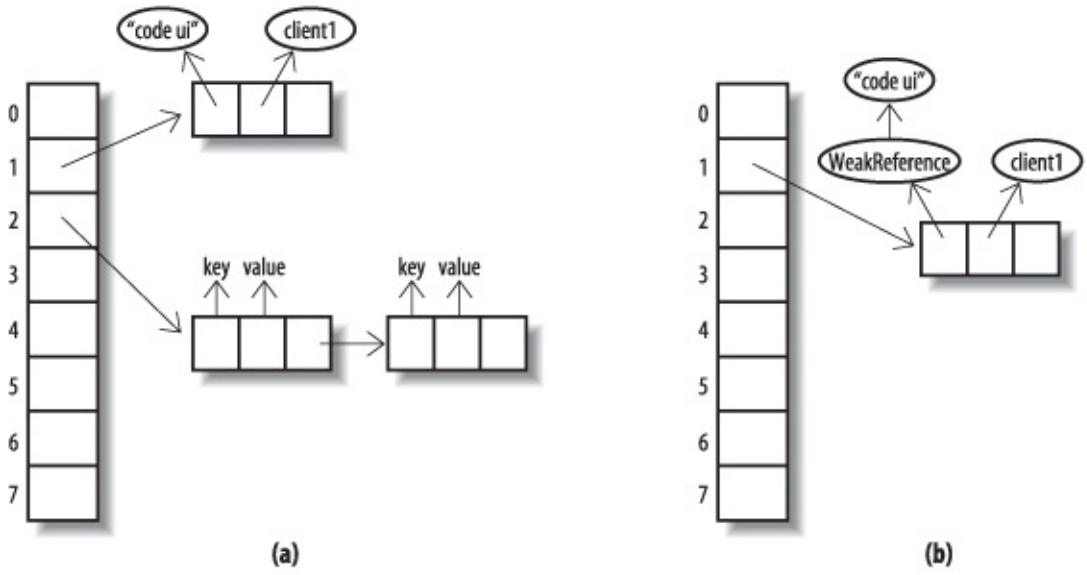

Figure 11-2. HashMap and WeakHashMap

WeakHashMap

Most `Map`s keep ordinary (“strong”) references to all the objects they contain. That means that even when a key has become unreachable by any means other than through the map itself, its mapping cannot be garbage collected. In the example at the beginning of this chapter, that is the situation of task–client mappings: they reference both task and client objects, both of which occupy memory, so preserving them unnecessarily has the potential to degrade garbage collection performance and create memory leaks. The idea behind `WeakHashMap` is to avoid this situation by allowing a mapping and its referenced objects to be reclaimed once the key is no longer reachable.

Internally `WeakHashMap` holds references to its key objects through references of the class `java.lang.ref.WeakReference`. A `WeakReference` introduces an extra level of indirection in reaching an object (see Figure 11-2(b)). For example, to make a weak reference to the GUI coding task you would write:

```
WeakReference<CodingTask> wref = new WeakReference<>(new
CodingTask("code gui"));
```

And at a later time you would recover a strong reference to it using the `get` method of `WeakReference`:

```
CodingTask recoveredRef = wref.get();
```

If there are no strong references to the `CodingTask` object, the existence of the weak reference will not by itself prevent the garbage collector from reclaiming the object. So the recovered reference value `recoveredRef` may, or may not, be `null`. (A `null` value may not always indicate that garbage collection has occurred: for example, a map might contain a key with the value `null`. For this reason, when you create a `WeakReference`, you can supply a queue on to which the garbage collector can add that weak reference when it has reclaimed its referent. `WeakHashMap` makes use of this mechanism.)

To see how `WeakHashMap` works, think of a tracing garbage collector that works by determining which objects are reachable, and reclaiming all others. The starting points for a reachability search include the static variables of currently loaded classes and the local variables currently in scope. Only strong

references are followed to determine reachability, so the keys of a `WeakHashMap` will be available to be reclaimed if they are not reachable by any other route. Note that a key cannot be reclaimed if it is strongly referenced from the corresponding value (including from the values they correspond to). Before most operations on a `WeakHashMap` are executed, the map examines its reference queue and, for each key that has been reclaimed it removes the corresponding entry.

What is a `WeakHashMap` good for? Imagine you have a program that allocates some transient system resource—a buffer, for example—on request from a client. Besides passing a reference to the resource back to the client, your program might also need to store information about it locally—for example, associating the buffer with the client that requested it. That could be implemented by means of a map from resource to client objects. But now, even after the client has disposed of the resource, the map will still hold a reference that will prevent the resource object from being garbage collected—if, that is, the reference is strong. Memory will gradually be used up by resources which are no longer in use. But if the reference is weak, held by a `WeakHashMap`, the garbage collector will be able to reclaim the object once it is no longer strongly referenced, and the memory leak is prevented.

A more general use is in those applications—for example, caches—where you don't mind information disappearing if memory is low. `WeakHashMap` isn't perfect for this purpose; one of its drawbacks is that it weakly references the map's keys rather than its values, which usually occupy much more memory. So even after the garbage collector has reclaimed a key, the real benefit in terms of available memory will not be experienced until the map has removed the stale entry. A second drawback is that weak references are *too* weak; the garbage collector is liable to reclaim a weakly reachable object at any time, and the programmer cannot influence this in any way. (A sister class of `WeakReference`, `java.lang.ref.SoftReference`, is treated differently: the garbage collector postpones reclaiming these until it is under severe memory pressure. Heinz Kabutz has written a `SoftReference`-based map that will work better as a cache; see *http://www.javaspecialists.co.za/archive/Issue098.xhtml*.)

`WeakHashMap` performs similarly to `HashMap`, though more slowly because

of the overheads of the extra level of indirection for keys. The cost of clearing out unwanted key-value associations before each operation is proportional to the number of associations that need to be removed. The iterators over collections of keys and values returned by `WeakHashMap` are fail-fast.

The factory method to use when you can predict the maximum number of entries is `newWeakHashMap`:

```
static <K,V>> WeakHashMap<K,V> newWeakHashMap(int numElements)
```

IdentityHashMap

The distinguishing characteristic of `IdentityHashMap` is discussed in Chapter 8: for the equivalence relation on its keys, it uses the identity relation. In other words, every physically distinct object is a distinct key.

An `IdentityHashMap` differs from an ordinary `HashMap` in that two keys are considered equal only if they are physically the same object: identity, rather than `equals`, is used for key comparison. That sets the contract for `IdentityHashMap` at odds with the contract for `Map`, the interface it implements, which specifies that equality should be used for key comparison. The problem here is exactly analogous to the one we encountered for `Set`: the Javadoc for `Map` assumes that the equivalence relation for maps is always defined by the `equals` method, whereas in fact `IdentityHashMap` uses a different relation, as do all implementations of `NavigableMap`, as we shall see (“NavigableMap”).

The main purpose of `IdentityHashMap` is to support operations in which a graph has to be traversed and information stored about each node. Serialization is one such operation. The algorithm used for traversing the graph must be able to check, for each node it encounters, whether that node has already been seen; otherwise, graph cycles could be followed indefinitely. For cyclic graphs, we must use identity rather than equality to check whether nodes are the same. Calculating equality between two graph node objects requires calculating the equality of their fields, which in turn means computing all their successors—and we are back to the original problem. An `IdentityHashMap`, by contrast, will report a node as being present only if that same node has previously been put

into the map.

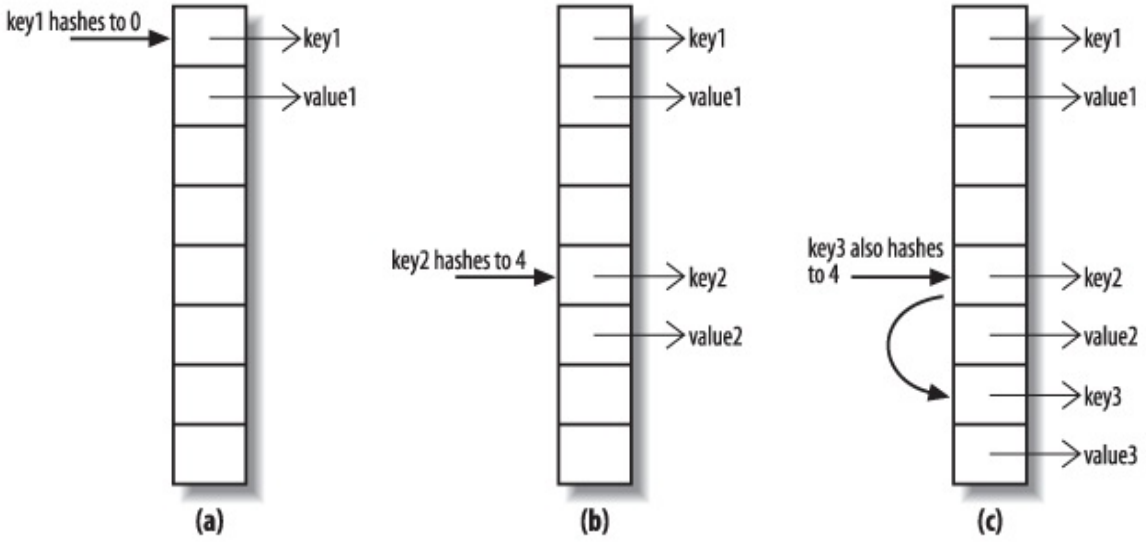

Figure 11-3. Resolving collisions by linear probing

The standard implementation of `IdentityHashMap` handles collisions differently than the chaining method shown in Figure 8-1 and used by all the other variants of `HashSet` and `HashMap`. This implementation uses a technique called *linear probing*, in which the key and value references are stored directly in adjacent locations in the table itself rather than in cells referenced from it. With linear probing, collisions are handled by simply stepping along the table until the first free pair of locations is found. Figure 11-3 shows three stages in filling an `IdentityHashMap` with a capacity of 8. In (a) we are storing a key-value pair whose key hashes to 0, and in (b) a pair whose key hashes to 4. The third key, added in (c), also hashes to 4, so the algorithm steps along the table until it finds an unused location. In this case, the first one it tries, with index 6, is free and can be used. Deletions are trickier than with chaining; if `key2` and `value2` were removed from the table in Figure 8-1, `key3` and `value3` would have to be moved along to take their place.

Out of all the Collections Framework hash implementations, why does `IdentityHashMap` alone use linear probing when all the others use chaining? The motivation for using probing is that it is somewhat faster than following a linked list, but that is only true when a reference to the value can be placed directly in the array, as in Figure 11-3. That isn't practical for all other hash-based collections, because they store the hash code as well as the value. This is

for reasons of efficiency: a `get` operation must check whether it has found the right key, and since equality is an expensive operation, it makes sense to check first whether it even has the right hash code. Of course, this reasoning doesn't apply to `IdentityHashMap`, which checks object identity rather than object equality.

There are three constructors for `IdentityHashMap`:

```
public IdentityHashMap()
public IdentityHashMap(Map<? extends K,? extends V> m)
public IdentityHashMap(int expectedMaxSize)
```

The first two are the standard constructors found in every general-purpose `Map` implementation. The third takes the place of the two constructors that in other `HashMaps` allow the user to control the initial capacity of the table and the load factor at which it will be rehashed. `IdentityHashMap` doesn't allow this, fixing it instead (at .67 in the standard implementation) in order to protect the user from the consequences of setting the load factor inappropriately: whereas the cost of finding a key in a table using chaining is proportional to the load factor *l*, in a table using linear probing it is proportional to 1/(1–*l*). So avoiding high load factors is too important to be left to the programmer! This decision is in line with the policy, mentioned earlier, of no longer providing configuration parameters when new classes are introduced into the Framework.

EnumMap

Implementing a mapping from an enumerated type is straightforward and very efficient, for reasons similar to those described for `EnumSet` (see "EnumSet"); in an array implementation, the ordinal value of each enumerated type constant can serve as the index of the corresponding value. The basic operations of `get` and `put` can be implemented as array accesses, in constant time. An iterator over the key set takes time proportional to the number of constants in the enumerated type and returns the keys in their natural order (the order in which the `enum` constants are declared). Although, as with `EnumSet`, this class is not thread-safe, the iterators over its collection views are not fail-fast but weakly consistent.

`EnumMap` has three public constructors:

```
EnumMap(Class<K> keyType)          // create an empty enum map
EnumMap(EnumMap<K, ? extends V> m) // create an enum map, with the
same
                                   // key type and elements as m
EnumMap(Map<K, ? extends V> m)     // create an enum map using the
elements
                                   // of the supplied Map, which
(unless it
                                   // is an EnumMap) must contain at
least
                                   // one association
```

An `EnumMap` contains a reified key type, which is used at run time for checking the validity of new entries being added to the map. This type is supplied by the three constructors in different ways. In the first, it is supplied as a class token; in the second, it is copied from the specified `EnumMap`. In the third, there are two possibilities: either the specified `Map` is actually an `EnumMap`, in which case it behaves like the second constructor, or the class of the first key of the specified `Map` is used (which is why the supplied `Map` may not be empty).

UnmodifiableMap

The family of implementations that we're calling `UnmodifiableMap<K,V>` is the last of the unmodifiable collections that we will meet in this book. They share many characteristics with their sister collections, `UnmodifiableSet` and `UnmodifiableList`: keys and values cannot be added, removed or updated, null values are disallowed, as with `UnmodifiableSet` duplicate values are rejected at creation time, and they are value-based.

The factory methods closely follow those for `UnmodifiableSet`: the `of` method has eleven overloads for creating unmodifiable maps containing zero to ten mappings:

```
<K,V> Map<K,V> of()                    // Returns an unmodifiable empty
map
<K,V> Map<K,V> of(K k1, V v1)          // Returns an unmodifiable map
containing one mapping
<K,V> Map<K,V> of(K k1, V v1, K k2, V v2)
                                       // Returns an unmodifiable map
```

```
containing two mappings
<K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3)
                                        // Returns an unmodifiable map
containing three mappings
<K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4)
                                        // Returns an unmodifiable map
containing four mappings
<K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K
e5)
                                        // Returns an unmodifiable map
containing five mappings
<K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K
k5, V v5, K k6)
                                        // Returns an unmodifiable map
containing six mappings
<K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K
k5, V v5, K k6, V v6, K k7)
                                        // Returns an unmodifiable map
containing seven mappings
<K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K
k5, V v5, K k6, V v6, K k7,
                                                          V v7, K k8, V
v8)
                                        // Returns an unmodifiable map
containing eight mappings
<K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K
k5, V v5, K k6, V v6, K k7,
                                                          V v7, K k8, V
v8, K k9, V v9)
                                        // Returns an unmodifiable map
containing nine mappings
<K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K
k5, V v5, K k6, V v6, K k7,
                                                          V v7, K k8, V
v8, K k9, V v9, K k10, V v10)
                                        // Returns an unmodifiable map
containing ten mappings
```

And two further methods support the creation of unmodifiable sets from any map, or from an array or varargs-supplied list of mappings:

```
<K,V> Map<K,V> ofEntries(Map.Entry<K,V>... entries)
                                        // Returns an unmodifiable map
with keys and
                                        // values extracted from entries
<K,V> Map<K,V> copyOf(Map<K,V> map)   // Returns an unmodifiable map
containing the entries of
                                        // the given Map.
```

Like the other unmodifiable collections, unmodifiable maps are backed by fixed-length arrays, in efficiency terms trading the advantage of reduced memory, faster iteration, and better spatial locality, for the disadvantage of linear-time lookup. Like `UnmodifiableSet`, and for the same reason, the order of iteration over the entry or key set is randomly determined for each virtual machine instance.

SequencedMap

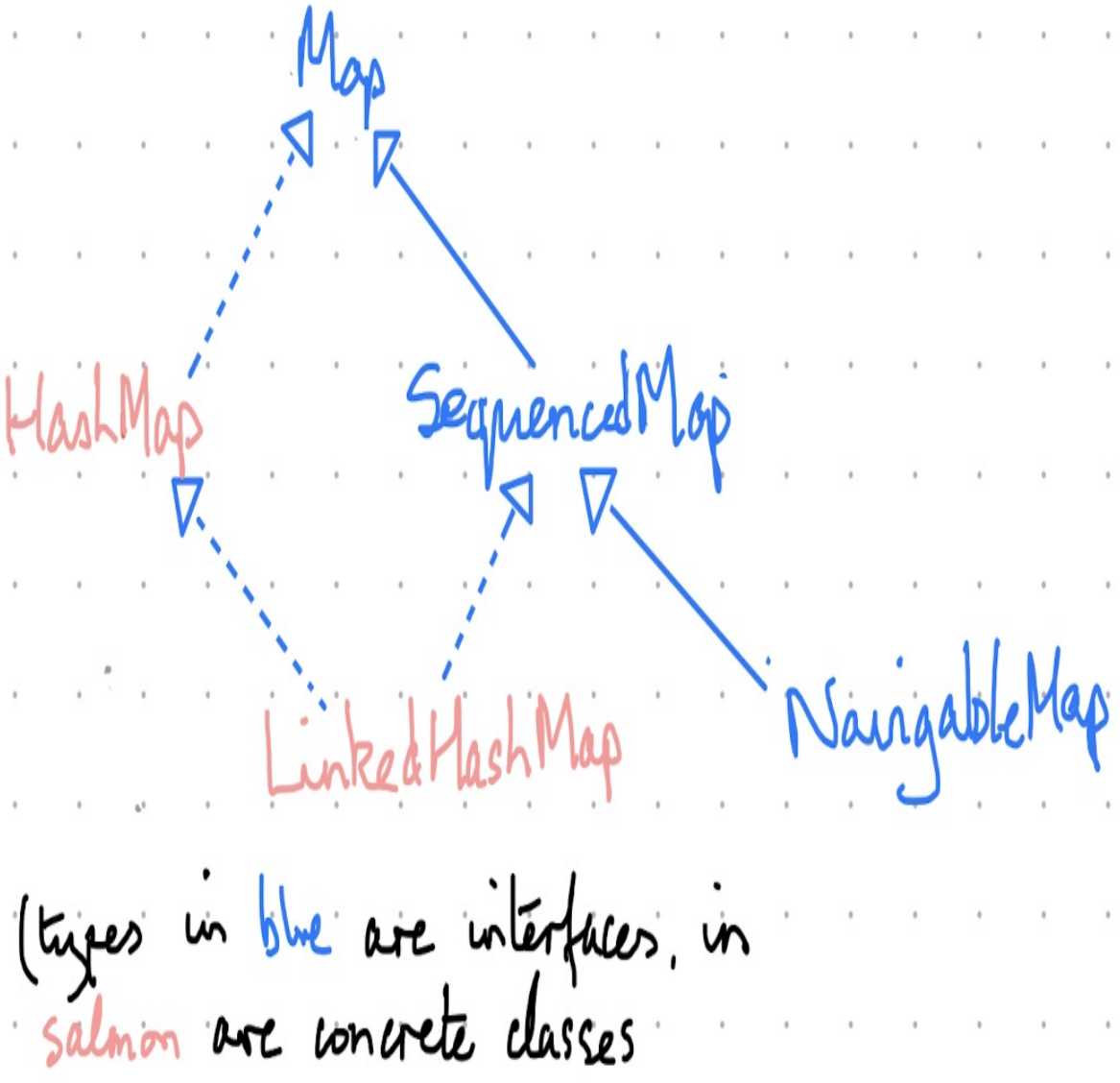

Figure 11-4. SequencedMap and Related Types

A `SequencedMap` (see Figure 11-4) is a `Map` whose keys and entries form

`SequencedSet`s. This interface exposes methods to add or inspect the first or last key or entry of a map, and to remove the first or last entry. Four other methods provide different views of the map: `reversed`, which returns a `SequencedMap`, `SequencedKeySet` and `SequencedEntrySet` which return `SequencedSet`s, and `SequencedValues`, which returns a `SequencedCollection`, preserving the ordering of the entries in the map.

LinkedHashMap

Like `LinkedHashSet` ("LinkedHashSet"), the class `LinkedHashMap` refines the contract of its parent class, `HashMap`, by guaranteeing the order in which iterators return its elements. Like `LinkedHashSet`, it implements the sequenced subinterface (`SequencedMap`) of its main interface. Unlike `LinkedHashSet`, however, `LinkedHashMap` offers a choice of iteration orders; elements can be returned either in the order in which they were inserted in the map—the default—or in the order in which they were accessed (from least-recently to most-recently accessed). An access-ordered `LinkedHashMap` is created by supplying an argument of `true` for the last parameter of the constructor:

```
public LinkedHashMap(int initialCapacity,
                     float loadFactor,
                     boolean accessOrder)
```

Supplying `false` will give an insertion-ordered map. The other constructors, which are just like those of `HashMap`, also produce insertion-ordered maps. As with `LinkedHashSet`, iteration over a `LinkedHashMap` takes time proportional only to the number of elements in the map, not its capacity.

Access-ordered maps are especially useful for constructing 'least recently used' (*LRU*) caches. A cache is an area of memory that stores frequently accessed data for fast access. In designing a cache, the key issue is the choice of algorithm that will be used to decide what data to remove in order to conserve memory. When an item from a cached data set needs to be found, the cache will be searched first. Typically, if the item is not found in the cache, it will be retrieved from the main store and added to the cache. But the cache cannot be allowed to continue growing indefinitely, so a strategy must be chosen for removing the least useful

item from the cache when a new one is added. If the strategy chosen is LRU, the entry removed will be the one least recently used. This simple strategy is suitable for situations in which access of an element increases the probability of further access in the near future of the same element. Its simplicity and speed have made it the most popular caching strategy. `LinkedHashMap` exposes a method specifically designed to make it easy to use as an LRU cache:

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest)
```

This method is called each time entries are added, by `put` or `putAll`. The implementation in `LinkedHashMap` simply returns `false`—an indication to the calling method that no action is needed. But you can subclass `LinkedHashMap` and override `removeEldestEntry` to return `true` under specific circumstances, in which case the calling method will remove the first entry in the map—the one least recently accessed (and if some entries have never been accessed, it is the one amongst these which was least recently added). The Javadoc gives the example of a map that should not grow past a given size:

```
class BoundedSizeMap<K,V> extends LinkedHashMap<K,V> {
    private final int maxEntries;
    public BoundedSizeMap1(int maxEntries) {
        super(16, 0.75f, true);
        this.maxEntries = maxEntries;
    }
    protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
        return size() > maxEntries;
    }
}
```

Notice that in an insertion-ordered `LinkedHashMap`, the eldest entry will be the one that was least recently added to the map, so overriding `removeEldestEntry` as shown above will implement a FIFO strategy. (FIFO caching has often been used in preference to LRU because it is much simpler to implement in maps that do not offer access ordering. However LRU is usually more effective than FIFO, because the reduced cost of cache refreshes outweighs the overhead of maintaining access ordering.)

If you need to modify the very specific action for which `removeEldestEntry` is specialised, you can just treat it as a callback, adding

whatever action you want, and returning `false`. For example, if a method `isReservedName` returned `true` for elements that should never be removed from the cache, you could implement `removeEldestEntry` like this:

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
    if (size() > maxEntries) {
        for (var itr = entrySet().iterator() ; itr.hasNext() ; ) {
            if (!isReservedName(itr.next().getKey())) {
                itr.remove();
                return false;
            }
        }
    }
    return false;
}
```

From Java 21, `LinkedHashMap` has implemented `SequencedMap`, providing much greater flexibility in caching policies. For example, in some applications recent access to an entry reduces rather than increasing the likelihood of it being accessed again soon. In that case, the best strategy is 'most recently used' (*MRU*), which discards the most recently used entry. This is straightforward to implement in a `SequencedMap`, which exposes a method `pollLastEntry` (analogous to `SequencedSet.removeLast`):

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
    if (size() > maxEntries) {
        pollLastEntry();
    }
    return false;
}
```

Extending this example one last time, we can combine the two variations above in a bounded MRU cache with reserved names, using the method `SequencedKeySet`, which provides a `SequencedSet` (see "SequencedSet") view of the keys:

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
   if (size() > maxEntries) {
       for (var itr = sequencedKeySet().reversed().iterator() ;
itr.hasNext() ; ) {
           if (!isReservedName(itr.next())) {
               itr.remove();
```

```
                return false;
            }
        }
    }
    return false;
}
```

These examples should not be taken as realistic except for the simplest uses: if you subclass `LinkedHashMap` in a real application, you will probably want more flexibility than can be provided by overriding `removeEldestEntry`. In that case, it would make more sense to override `put` and `putAll`—or indeed to provide other methods to allow users to adjust the cache, for example by using the `SequencedMap` methods `putFirst` or `putLast` to relocate entries to the beginning or end of the map.

Iteration over a collection of keys or values returned by a `LinkedHashMap` is linear in the number of elements. The iterators over such collections are fail-fast.

The factory method to use when you can predict the maximum number of entries is `newLinkedHashMap`:

```
static <K,V>> LinkedHashMap<K,V> newLinkedHashMap(int numElements)
```

NavigableMap

The interface `NavigableMap` adds to `SequencedMap` a guarantee that its iterator will traverse the map in ascending key order and, like `NavigableSet`, adds further methods to find the entries adjacent to a target key value. As with `NavigableSet`, `NavigableMap` extends, and in effect replaces, an older interface, `SortedMap`, which imposes an ordering on its keys, either that of their natural ordering or of a `Comparator`. The equivalence relation on the keys of a `NavigableMap` is again defined by the ordering relation: two keys that compare as equal will be regarded as duplicates by a `NavigableMap` (see Chapter 8).

The extra methods defined by the `NavigableMap` interface fall into four groups, as described below:

Retrieving the Comparator

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

This method returns the map's key comparator if it has been given one, instead of relying on the natural ordering of the keys. Otherwise, it returns `null`.

Getting Range Views

As with `NavigableSet`, each of the methods in this group appears in two overloads, one inherited from `SortedMap` and returning a half-open `SortedMap` view, and one defined in `NavigableMap` returning a `NavigableSet` view that can be open, half-open, or closed according to the user's choice. These operations work like the corresponding operations in `SortedSet`: the key arguments do not themselves have to be present in the map, and the sets returned include the `fromKey`— if, in fact, it is present in the map—and do not include the `toKey`.

```
SortedMap<K,V> subMap(K fromKey, K toKey)
SortedMap<K,V> headMap(K toKey)
SortedMap<K,V> tailMap(K fromKey)
NavigableMap<K,V> subMap(
            K fromKey, boolean fromInclusive, K toKey, boolean
toInclusive)
NavigableMap<K,V> headMap(K toKey, boolean inclusive)
NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)
```

Like the `NavigableSet` methods, the `NavigableMap` provide more flexibility than the range-view methods of `SortedMap`. Instead of always returning a half-open interval, these methods accept `boolean` parameters that are used to determine whether to include the key or keys defining the interval.

Getting Closest Matches

```
Map.Entry<K,V> ceilingEntry(K Key)
K ceilingKey(K Key)
Map.Entry<K,V> floorEntry(K Key)
K floorKey(K Key)
Map.Entry<K,V> higherEntry(K Key)
K higherKey(K Key)
Map.Entry<K,V> lowerEntry(K Key)
K lowerKey(K Key)
```

These are similar to the corresponding closest-match methods of

`NavigableSet`, but they return `Map.Entry` objects. If you want the key belonging to one of these entries, use the corresponding convenience key-returning method, with the performance benefit of allowing the map to avoid the unnecessary creation of a `Map.Entry` object.

Other Views

The `SequencedMap` method `reversed` is inherited by `NavigableMap`, with a covariant override to allow it to return a `NavigableMap`. This is a synonym for `descendingMap`, which remains for historic reasons, having been present before `NavigableMap` was retrofitted to inherit from `SequencedMap`. Similarly, the method `descendingKeySet` has almost the same effect as calling `reversed` on the key set of a `SequencedMap`, but again returns the more precise type of `NavigableSet`.

```
NavigableMap<K,V> descendingMap()  // return a reverse-order view of
the map
NavigableSet<K> descendingKeySet() // return a reverse-order key set
NavigableSet<K> navigableKeySet() // return a forward-order key set
```

You might wonder why the method `keySet`, inherited from `Map`, could not simply be overridden using a covariant return type to return a `NavigableSet`. Indeed, the platform implementations of `NavigableMap.keySet` do return a `NavigableSet`. But there is a compatibility concern: if `TreeMap.keySet` were to have its return type changed from `Set` to `NavigableSet`, any existing `TreeMap` subclasses which override that method would now fail to compile unless they too changed their return type. (This concern is similar to those discussed in [Link to Come].)

TreeMap

`NavigableMap` is implemented in the Collections Framework by `TreeMap`. We met trees as a data structure for storing elements in order when we discussed `TreeSet` (see “TreeSet”). In fact, the internal representation of a `TreeSet` is just a `TreeMap` in which every key is associated with the same standard value, so the explanation of the mechanism and performance of red-black trees given in “TreeSet” applies equally here.

The constructors for `TreeMap` include, besides the standard ones, one that allows you to supply a `Comparator` and one that allows you to create one from another `NavigableMap` (strictly speaking, from a `SortedMap`), using both the same comparator and the same mappings:

```
public TreeMap(Comparator<? super K> comparator)
public TreeMap(SortedMap<K, ? extends V> m)
```

Notice that the second of these constructors suffer from a similar problem to the corresponding constructor of `TreeSet` (see "TreeSet"), because the standard conversion constructor always uses the natural ordering of the keys, even when its argument is actually a `SortedMap`.

`TreeMap` has similar performance characteristics to `TreeSet`: the basic operations (`get`, `put`, and `remove`) perform in *O*(log *n*) time). The collection view iterators are fail-fast.

ConcurrentMap

Maps are often used in high-performance server applications—for example, as cache implementations—so a high-throughput thread-safe map implementation is an essential part of the Java platform. This requirement cannot be met by synchronized maps such as those provided by `Collections.synchronizedMap`, because with full synchronization each operation needs to obtain an exclusive lock on the entire map, effectively serializing access to it. Locking only a part of the collection at a time—*lock striping*—can achieve very large gains in throughput, as we shall see shortly with `ConcurrentHashMap`. But because there is no single lock for a client to hold to gain exclusive access, client-side locking no longer works, and clients need assistance from the collection itself to carry out atomic actions.

That is the purpose of the interface `ConcurrentMap`. It provides declarations for methods that perform compound operations atomically. There are four of these methods:

```
V putIfAbsent(K key, V value)
        // associate key with value only if key is not currently
```

```
present.
        // return the old value (may be null) if the key was present,
        // otherwise return null
boolean remove(Object key, Object value)
        // remove key only if it is currently mapped to value. Returns
        // true if the value was removed, false otherwise
V replace(K key, V value)
        // replace entry for key only if it is currently present.
Return
        // the old value (may be null) if the key was present,
otherwise
        // return null
boolean replace(K key, V oldValue, V newValue)
        // replace entry for key only if it is currently mapped to
oldValue.
        // return true if the value was replaced, false otherwise
```

ConcurrentHashMap

`ConcurrentHashMap` provides an implementation of `ConcurrentMap` and offers a highly effective solution to the problem of reconciling throughput with thread safety. It is optimized for reading, so retrievals do not block even while the table is being updated (to allow for this, the contract states that the results of retrievals will reflect the latest update operations completed before the start of the retrieval). Updates also can often proceed without blocking, because a `ConcurrentHashMap` consists of not one but a set of tables, called *segments*, each of which can be independently locked. If the number of segments is large enough relative to the number of threads accessing the table, there will often be no more than one update in progress per segment at any time. The constructors for `ConcurrentHashMap` are similar to those of `HashMap`, but with an extra one that provides the programmer with control over the number of segments that the map will use (its *concurrency level*):

```
ConcurrentHashMap()
ConcurrentHashMap(int initialCapacity)
ConcurrentHashMap(int initialCapacity, float loadFactor)
ConcurrentHashMap(
  int initialCapacity, float loadFactor, int concurrencyLevel)
ConcurrentHashMap(Map<? extends K,? extends V> m)
```

The class `ConcurrentHashMap` is a useful implementation of `Map` in any application where it is not necessary to lock the entire table; this is the one

capability of `SynchronizedMap` which it does not support. That can sometimes present problems: for example, the `size` method attempts to count the entries in the map without using locks. If the map is being concurrently updated, however, the `size` method will not succeed in obtaining a consistent result. In this situation, it obtains exclusive access to the map by locking all the segments, obtaining the entry count from each, then unlocking them again. The performance cost involved in this is a justifiable tradeoff against the highly optimized performance for common operations, especially reads. Overall, `ConcurrentHashMap` is indispensable in highly concurrent contexts, where it performs far better than any available alternative.

Disregarding locking overheads such as those just described, the cost of the operations of `ConcurrentHashMap` are similar to those of `HashMap`. The collection views return weakly consistent iterators.

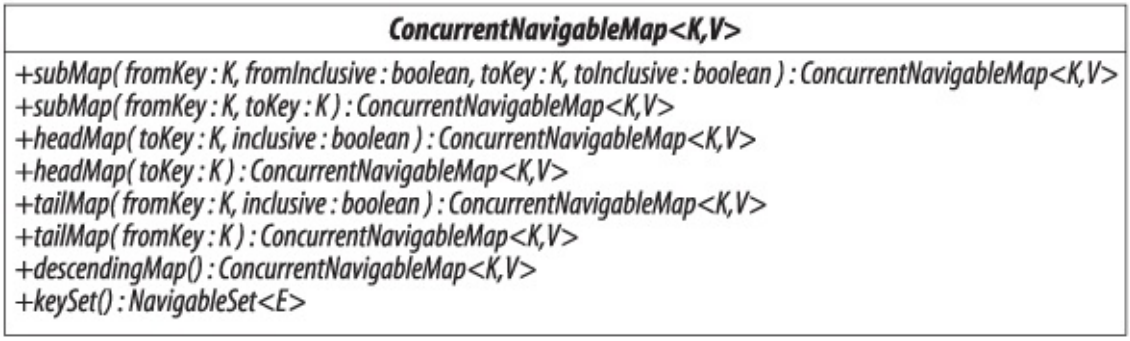

Figure 11-5. ConcurrentNavigableMap

ConcurrentNavigableMap

`ConcurrentNavigableMap` (see Figure 11-5) inherits from both `ConcurrentMap` and `NavigableMap`, and contains just the methods of these two interfaces with a few changes to make the return types more precise. The range-view methods inherited from `SortedMap` and `NavigableMap` now return views of type `ConcurrentNavigableMap`. The compatibility concerns that prevented `NavigableMap` from overriding the methods of `SortedMap` don't apply here to overriding the range-view methods of `NavigableMap` or `SortedMap`; because neither of these has any implementations that have been retrofitted to the new interface, the danger of breaking implementation subclasses does not arise. For the same reason, it is

now possible to override `keySet` to return `NavigableSet`.

ConcurrentSkipListMap

The relationship between `ConcurrentSkipListMap` and `ConcurrentSkipListSet` is like that between `TreeMap` and `TreeSet`; a `ConcurrentSkipListSet` is implemented by a `ConcurrentSkipListMap` in which every key is associated with the same standard value, so the mechanism and performance of the skip list implementation given in "ConcurrentSkipListSet" applies equally here: the basic operations (`get`, `put`, and `remove`) perform in $O(\log n)$ time); iterators over the collection views execute `next` in constant time. These iterators are fail-fast.

Comparing Map Implementations

Table 11-1 shows the relative performance of the different platform implementations of `Map` (the column headed "next" shows the cost of the `next` operation of iterators over the key set). As with the implementations of queue, your choice of map class is likely to be influenced more by the functional requirements of your application and the concurrency properties that you need.

Some specialized situations dictate the implementation: `EnumMap` should always (and only) be used for mapping from enums. Problems such as the graph traversals described in "IdentityHashMap" call for `IdentityHashMap`. For a sorted map, use `TreeMap` where thread safety is not required, and `ConcurrentSkipListMap` otherwise.

Table 11-1. Comparative performance of different Map implementations

++	get	containsKey	next	Notes
`HashMap`	$O(1)$	$O(1)$	$O(h/n)$	h is the tal capacity
`LinkedHashMap`	$O(1)$	$O(1)$	$O(1)$	

`IdentityHashMap`	*O*(1)	*O*(1)	*O*(*h*/*n*)	*h* is the tal capacity
`EnumMap`	*O*(1)	*O*(1)	*O*(1)	
`TreeMap`	*O*(log *n*)	*O*(log *n*)	*O*(log *n*)	
`ConcurrentHashMap`	*O*(1)	*O*(1)	*O*(*h*/*n*)	*h* is the tal capacity
`ConcurrentSkipListMap`	*O*(log *n*)	*O*(log *n*)	*O*(1)	

That leaves the choice of implementation for general-purpose maps. For concurrent applications, `ConcurrentHashMap` is the only choice. Otherwise, favor `LinkedHashMap` over HashMap (and accept its slightlyworse performance) if you need to make use of the insertion or access order of the map —for example, to use it as a cache.

Chapter 12. The Collections Class

The class `java.util.Collections` consists entirely of static methods that operate on or return collections. There are three main categories: generic algorithms, methods that return empty or prepopulated collections, and methods that create wrappers. We discuss these three categories in turn, followed by a number of other methods which do not fit into a neat classification.

All the methods of `Collections` are public and static, so for readability we will omit these modifiers from the individual declarations.

Generic Algorithms

The generic algorithms fall into four major categories: changing element order in a list, changing the contents of a list, finding extreme values in a collection, and finding specific values in a list. They represent reusable functionality, in that they can be applied to `Lists` (or in some cases to `Collections`) of any type. Generifying the types of these methods has led to some fairly complicated declarations, so each section discusses the declarations briefly after presenting them.

Changing the Order of List Elements

There are eight methods for reordering lists in various ways. The simplest of these is `swap`, which exchanges two elements and, in the case of a `List` which implements `RandomAccess`, executes in constant time. The most complex is `sort`, which transfers the elements into an array, applies a merge sort to them in time $O(n \log n)$, and then returns them to the `List`. All of the remaining methods execute in time $O(n)$.

```
void reverse(List<?> list)
          // reverse the order of the elements
void rotate(List<?> list, int distance)
          // rotate the elements of the list; the element at index
      // i is moved to index (distance + i) % list.size()
void shuffle(List<?> list)
          // randomly permute the list elements
void shuffle(List<?> list, Random rnd)
          // randomly permute the list using the randomness source rnd
void shuffle(List<?> list, RandomGenerator rnd)
          // randomly permute the list using the randomness source rnd
<T extends Comparable<? super T>> void sort(List<T> list)
          // sort the supplied list using natural ordering
<T> void sort(List<T> list, Comparator<? super T> c)
          // sort the supplied list using the supplied ordering
void swap(List<?> list, int i, int j)
          // swap the elements at the specified positions
```

For each of these methods, except `sort` and `swap`, there are two algorithms, one using iteration and another using random access. The method `sort` transfers the `List` elements to an array, where they are sorted using—in the current implementation—a mergesort algorithm with $n \log n$ performance. The method `swap` always uses random access. The standard implementations for the other methods in this section use either iteration or random access, depending on whether the list implements the `RandomAccess` interface (see [Link to Come]). If it does, the implementation chooses the random-access algorithm; even for a list that does not implement `RandomAccess`, however, the random-access algorithms are used if the list size is below a given threshold, determined on a per-method basis by performance testing.

Changing the Contents of a List

These methods change some or all of the elements of a list. The method `copy` transfers elements from the source list into an initial sublist of the destination list (which has to be long enough to accommodate them), leaving any remaining elements of the destination list unchanged. The method `fill` replaces every element of a list with a specified object, and `replaceAll` replaces every occurrence of one value in a list with another—where either the old or new value can be `null`—returning `true` if any replacements were made.

```
<T> void copy(List<? super T> dest, List<? extends T> src)
          // copy all of the elements from one list into another
<T> void fill(List<? super T> list, T obj)
          // replace every element of list with obj
<T> boolean replaceAll(List<T> list, T oldVal, T newVal)
          // replace all occurrences of oldVal in list with newVal
```

The signatures of these methods can be explained using the Get and Put Principle (see "The Get and Put Principle"). The signature of `copy` was discussed in "Wildcards with super". It gets elements from the source list and puts them into the destination, so the types of these lists are, respectively, `? extends T` and `? super T`. This fits with the intuition that the type of the source list elements should be a subtype of the destination list. Although there are simpler alternatives for the signature of `copy`, "Wildcards with super" makes the case that using wildcards where possible widens the range of permissible calls.

For `fill`, the Get and Put Principle dictates that you should use `super` if you are putting values into a parameterized collection and, for `replaceAll`, it states that if you are putting values into and getting values out of the same structure, you should not use wildcards at all.

Finding Extreme Values in a Collection

The methods `min` and `max` are supplied for this purpose, each with two overloads—one using natural ordering of the elements, and one accepting a `Comparator` to impose an ordering. They execute in linear time.

```
<T extends Object & Comparable<? super T>>
  T max(Collection<? extends T> coll)  // return the maximum element
```

```
                                          // using natural ordering
<T> T max(Collection<? extends T> coll, Comparator<? super T> comp)
                                          // return the maximum element
                                          // using the supplied
comparator
<T extends Object & Comparable<? super T>>
  T min(Collection<? extends T> coll)  // return the minimum element
                                          // using natural ordering
<T> T min(Collection<? extends T> coll, Comparator<? super T> comp)
                                          // return the minimum element
                                          // using the supplied
comparator
```

Sections “Multiple Bounds” and [Link to Come] explain these methods and the types assigned to them.

Finding Specific Values in a List

Methods in this group locate elements or groups of elements in a `List`, again choosing between alternative algorithms on the basis of the list’s size and whether it implements `RandomAccess`.

```
<T> int binarySearch(List<? extends Comparable<? super T>> list, T
key)
          // search for key using binary search
<T> int binarySearch(List<? extends T> list, T key, Comparator<? super
T> c)
          // search for key using binary search
int indexOfSubList(List<?> source, List<?> target)
          // find the first sublist of source which matches target
int lastIndexOfSubList(List<?> source, List<?> target)
          // find the last sublist of source which matches target
```

The signature of the first `binarySearch` overload says that you can use it to search for a key of type `T` in a list of objects that can have any type that can be compared with objects of type `T`. The second is like the `Comparator` overloads of `min` and `max` except that, in this case, the type parameter of the `Collection` must be a subtype of the type of the key, which in turn must be a subtype of the type parameter of the `Comparator`.

Binary search requires a sorted list for its operation. At the start of a search, the range of indices in which the search value may occur corresponds to the entire

list. The binary search algorithm samples an element in the middle of this range, using the value of the sampled element to determine whether the new range should be the part of the old range above or the part below the index of the element. A third possibility is that the sampled value is equal to the search value, in which case the search is complete. Since each step halves the size of the range, *m* steps are required to find a search value in a list of length 2^m, and the time complexity for a list of length *n* is *O*(log *n*).

The methods `indexOfSubList` and `lastIndexOfSubList` do not require sorted lists for their operation. Their signatures allow the source and target lists to contain elements of any type (remember that the two wildcards may stand for two different types). The design decision behind these signatures is the same as that behind the `Collection` methods `containsAll`, `retainAll`, and `removeAll` (see "Wildcards Versus Type Parameters").

Collection Factories

The `Collections` class provides convenient ways of creating some kinds of collections containing zero or more references to the same object. The simplest possible such collections are empty:

```
<T> List<T> emptyList()                        // return an empty List
<K,V> Map<K,V> emptyMap()                      // return an empty Map
<T> Set<T> emptySet()                          // return an empty Set
<T> Iterator<T> emptyIterator()                // return an empty
Iterator
<T> ListIterator<T> emptyListIterator()        // return an empty
ListIterator
<T> NavigableSet<T> emptyNavigableSet()        // return an empty
NavigableSet
<K,V> NavigableMap<K,V> emptyNavigableMap() // return an empty
NavigableMap
```

In this section, and in the following section on wrapper factories, we have omitted the methods relating to `SortedSet` and `SortedMap`, as these have been made effectively obsolete by the introduction of `NavigableSet` and `NavigableMap`.

Empty collections can be useful in implementing methods to return collections

of values, where they can be used to signify that there were no values to return. Each method returns a reference to an instance of a singleton inner class of `Collections`. Because these instances are immutable, they can safely be shared, so calling one of these factory methods does not result in object creation. The `Collections` fields `EMPTY_SET`, `EMPTY_LIST`, and `EMPTY_MAP` were commonly used for the same purpose in Java before generics, but are less useful now because their raw types generate unchecked warnings whenever they are used. These methods have themselves become less useful as their function has been duplicated by the unmodifiable factory methods `Set.of()`, `List.of()`, and `Map.of()`. The methods `emptyNavigableSet` and `emptyNavigableMap` may become similarly redundant in the future if factory methods for unmodifiable `NavigableSet`s and `NavigableMap`s are introduced.

The `Collections` class also provides you with ways of creating collection objects containing only a single member:

```
<T> Set<T> singleton(T o)
        // return an immutable set containing only the specified
object
<T> List<T> singletonList(T o)
        // return an immutable list containing only the specified
object
<K,V> Map<K,V> singletonMap(K key, V value)
        // return an immutable map, mapping only the key K to the
value V
```

Again, these can be useful in providing a single input value to a method that is written to accept a `Collection` of values and, again, they have been duplicated by the unmodifiable factory methods.

Finally, it is possible to create a list containing a number of copies of a given object.

```
<T> List<T> nCopies(int n, T o)
        // return an immutable list containing n references to the
object o
```

Because the list produced by `nCopies` is immutable, it need contain only a single physical element to provide a list view of the required length. Such lists

are often used as the basis for building further collections—for example, as the argument to a constructor or an `addAll` method.

Wrappers

The `Collections` class provides wrapper objects that modify the behavior of standard collections classes in one of three ways—by synchronizing them, by making them unmodifiable, or by checking the type of elements being added to them. These wrapper objects implement the same interfaces as the wrapped objects, and they delegate their work to them. Their purpose is to restrict the circumstances under which that work will be carried out. These are examples of the use of *protection proxies* (see *Design Patterns* by Gamma, Helm, Johnson, and Vlissides, Addison-Wesley), a variant of the Proxy pattern in which the proxy controls access to the real subject.

Proxies can be created in different ways. Here, they are created by factory methods that wrap the supplied collection object in an inner class of `Collections` that implements the collection's interface. Subsequently, method calls to the proxy are (mostly) delegated to the collection object, but the proxy controls the conditions of the call: in the case of the synchronized wrappers, all method calls are delegated but the proxy uses synchronization to ensure that the collection is accessed by only one thread at a time. In the case of unmodifiable and checked collections, method calls that break the contract for the proxy fail, throwing the appropriate exception.

Synchronized Collections

As we explained in "Collections and Thread Safety", most of the Framework classes are not thread-safe—by design—in order to avoid the overhead of unnecessary synchronization (as incurred by the legacy classes `Vector` and `Hashtable`). But for the occasions when you do need to provide a small number of threads with concurrent access to the same collection, these synchronized wrappers are provided by the `Collections` class:

```
<T> Collection<T> synchronizedCollection(Collection<T> c);
<T> Set<T> synchronizedSet(Set<T> s);
<T> List<T> synchronizedList(List<T> list);
```

```
<K, V> Map<K, V> synchronizedMap(Map<K, V> m);
<T> NavigableSet<T> synchronizedNavigableSet(NavigableSet<T> s);
<K, V> NavigableMap<K, V> synchronizedNavigableMap(NavigableMap<K, V>
m);
```

The classes that provide these synchronized views are conditionally thread-safe (see “Collections and Thread Safety”); although each of their operations is guaranteed to be atomic, you may need to synchronize multiple method calls in order to obtain consistent behavior. In particular, iterators must be created and used entirely within a code block synchronized on the collection; otherwise, the result will at best be failure with `ConcurrentModificationException`. This is very coarse-grained synchronization; if your application makes heavy use of synchronized collections, its effective concurrency will be greatly reduced. In this situation, you should turn to the appropritae concurrent collection (see [Link to Come]).

Unmodifiable Collections

An unmodifiable collection will throw `UnsupportedOperationException` in response to any attempt to change its structure or the elements that compose it. Used with care, this can be useful when you want to allow clients read access to an internal data structure: passing the structure in an unmodifiable wrapper will prevent a client from changing it, but it will not prevent the client from changing the objects it contains, if they are modifiable. Often, you will have to protect your internal data structure by providing clients instead with a *defensive copy* made for that purpose, or by also placing these objects in unmodifiable wrappers.

There are unmodifiable wrapper factory methods corresponding to the major interfaces of the Collections Framework:

```
<T> Collection<T> unmodifiableCollection(Collection<? extends T> c)
<T> Set<T> unmodifiableSet(Set<? extends T> s)
<T> List<T> unmodifiableList(List<? extends T> list)
<K, V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m)
<T> NavigableSet<T> unmodifiableNavigableSet(NavigableSet<? extends T>
s)
<K, V> NavigableMap<K, V> unmodifiableNavigableMap(NavigableMap<K, ?
extends V> m)
```

In many situations where you want an unmodifiable view of a modifiable `Set`, `List`, or `Map`, you have a choice between wrapping the collection using one of these methods, and copying it into a new unmodifiable collection created using the appropriate `of` factory method. The tradeoffs involved in this choice are discussed in [Link to Come].

Checked Collections

Unchecked warnings from the compiler are a signal to us to take special care to avoid runtime type violations. For example, after we have passed a typed collection reference to an ungenerified library method, we can't be sure that it has added only correctly typed elements to the collection. Instead of losing confidence in the collection's type safety, we can pass in a checked wrapper, which will test every element added to the collection for membership of the type supplied when it was created. [Link to Come] shows an example of this technique.

```
<E> Collection<E>
    checkedCollection(Collection<E> c, Class<E> elementType)
<E> List<E>
    checkedList(List<E> c, Class<E> elementType)
<E> Set<E>
    checkedSet(Set<E> c, Class<E> elementType)
<E> NavigableSet<E>
    checkedNavigableSet(NavigableSet<E> c, Class<E> elementType)
<K, V> Map<K, V>
    checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)
<K, V> NavigableMap
    checkedNavigableMap(NavigableMap<K, V> c, Class<K>
keyType,Class<V> valueType)
<E> Queue<E>
    checkedQueue(Queue<E> c, Class<E> elementType)
```

Other Methods

The `Collections` class provides a number of utility methods, some of which we have already seen in use. Here we review them in alphabetical order.

addAll

```
<T> boolean addAll(Collection<? super T> c, T... elements)
        // adds all of the specified elements to the specified
collection.
```

We have used this method a number of times as a convenient and efficient way of initializing a collection with individual elements, or with the contents of an array.

asLifoQueue

```
<T> Queue<T> asLifoQueue(Deque<T> deque)
        // returns a view of a Deque as a Last-in-first-out (Lifo)
Queue.
```

Recall from Chapter 9 that while queues can impose various different orderings on their elements, there is no standard `Queue` implementation that provides LIFO ordering. `Deque` implementations, on the other hand, all support LIFO ordering if elements are removed from the same end of the dequeue as they were added. The method `asLifoQueue` allows you to use this functionality through the conveniently concise `Queue` interface.

disjoint

```
boolean disjoint(Collection<?> c1, Collection<?> c2)
        // returns true if c1 and c2 have no elements in common
```

The implementation of this method in the OpenJDK iterates over one of these collections, testing each element for membership in the other and returning `true` if none are found. This is straightforward for non-set collections, which generally use the `equals` method to test for membership; some sets, however, use other equivalence relations to characterise membership (see Chapter 8). So if two sets using different equivalence relations are being tested for disjointness, the result may depend on which of the two has been chosen by the implementation to test for membership. As with equality of sets, you should avoid using `disjoint` on sets with different equivalence relations.

enumeration

```
<T> Enumeration<T> enumeration(Collection<T> c)
        // returns an enumeration over the specified collection
```

This method is provided for interoperation with APIs whose methods take arguments of type `Enumeration`, a legacy version of `Iterator`. The `Enumeration` it returns yields the same elements, in the same order, as the `Iterator` provided by `c`. This method forms a pair with the method `list`, which converts an `Enumeration` value to an `ArrayList`.

frequency

```
int frequency(Collection<?> c, Object o)
        // returns the number of elements in c that are equal to o
```

If the supplied value `o` is `null`, then `frequency` returns the number of `null` elements in the collection `c`.

list

```
<T> ArrayList<T> list(Enumeration<T> e)
        // returns an ArrayList containing the elements returned by
the specified Enumeration
```

This method is provided for interoperation with APIs whose methods return results of type `Enumeration`, a legacy version of `Iterator`. The `ArrayList` that it returns contains the same elements, in the same order, as provided by `e`. This method forms a pair with the method `enumeration`, which converts a Framework collection to an `Enumeration`.

newSetFromMap

```
<E> Set<E> newSetFromMap(Map<E, Boolean> map)
        // returns a set backed by the specified map.
```

As we saw earlier, many sets (such as `TreeSet` and `NavigableSkipListSet`) are implemented by maps, and share their ordering, concurrency, and performance characteristics. Some maps, however (such as `WeakHashMap` and `IdentityHashMap`) do not have standard `Set` equivalents. The purpose of the method `newSetFromMap` is to provide equivalent `Set` implementations for such maps. The method `newSetFromMap` wraps its argument, which must be empty when supplied and should never be subsequently accessed directly. This code shows the standard idiom for using it

to create a weak `HashSet`, one whose elements are held via weak references:

```
Set<Object> weakHashSet = Collections.newSetFromMap(
    new WeakHashMap<Object, Boolean>());
```

reverseOrder

```
<T> Comparator<T> reverseOrder()
        // returns a comparator that reverses natural ordering
```

This method provides a simple way of sorting or maintaining a collection of `Comparable` objects in reverse natural order. Here is an example of its use:

```
SortedSet<Integer> s = new TreeSet<Integer>
(Collections.reverseOrder());
Collections.addAll(s, 1, 2, 3);
assert s.toString().equals("[3, 2, 1]");
```

This method predates Java 8, which introduced the feature of static interface methods. The `Comparator` method `reverseOrder`, which has exactly the same functionality as `Collections.reverseOrder`, was provided in Java 8 as an alternative that developers would find more easily when looking for ways of manipulating `Comparator`s.

There is also a second overload of this method:

```
<T> Comparator<T> reverseOrder(Comparator<T> cmp)
```

This method is like the preceding one, but instead of reversing the natural order of an object collection, it reverses the order of the `Comparator` supplied as its argument. Its behaviour when supplied with `null` is unusual for a method of the `Collections` class. The contract for `Collections` states that its methods throw a `NullPointerException` if the collections or class objects provided to them are null, but if this method is supplied with `null` it returns the same result as a call of `reverseOrder()`—that is, it returns a `Comparator` that reverses the natural order of a collection of objects.

This method too has an equivalent in the `Comparator` interface: in this case, it is the default (i.e. instance) method `Comparator.reversed()`, in which the

receiver takes the place of the parameter to the static `Collections` method.