Java Tutorials

Updated for Java SE 8

ORACLE

Java Tutorials

The Java Tutorials are a one stop shop for learning the Java language. The tutorials are practical guides for programmers who want to use the Java programming language to create applets and applications. They also serve as a reference for the experienced Java programmer.

Groups of related lessons are organized into "trails"; for example, the "Getting Started" trail, or the "Learning the Java Language" trail. This ebook contains a single trail. You can download other trails via the link on the Java Tutorials home page.

To read an ebook file that you have downloaded, transfer the document to your device.

We would love to hear what you think of this ebook, including any problems that you find. Please send us your feedback.

Legal Notices

Copyright 1995, 2014, Oracle Corporation and/or its affiliates (Oracle). All rights reserved.

This tutorial is a guide to developing applications for the Java Platform, Standard Edition and contains documentation (Tutorial) and sample code. The sample code made available with this Tutorial is licensed separately to you by Oracle under the Berkeley license. If you download any such sample code, you agree to the terms of the Berkeley license.

This Tutorial is provided to you by Oracle under the following license terms containing restrictions on use and disclosure and is protected by intellectual property laws. Oracle grants to you a limited, non-exclusive license to use this Tutorial for information purposes only, as an aid to learning about the Java SE platform. Except as expressly permitted in these license terms, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means this Tutorial. Reverse engineering, disassembly, or decompilation of this Tutorial is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If the Tutorial is licensed on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This Tutorial is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this Tutorial in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use.

THE TUTORIAL IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND. ORACLE FURTHER DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT.

IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, PUNITIVE OR CONSEQUENTIAL DAMAGES, OR DAMAGES FOR LOSS OF PROFITS, REVENUE, DATA OR DATA USE, INCURRED BY YOU OR ANY THIRD PARTY, WHETHER IN AN ACTION IN CONTRACT OR TORT, EVEN IF ORACLE HAS BEEN

ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. ORACLES ENTIRE LIABILITY FOR DAMAGES HEREUNDER SHALL IN NO EVENT EXCEED ONE THOUSAND DOLLARS (U.S. $1,000).

No Technical Support

Oracles technical support organization will not provide technical support, phone support, or updates to you.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

The sample code and Tutorial may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Supported Platforms

The `mobi` file format works best on the following devices:

- Kindle Fire
- Kindle DX

The `ePub` file format works best on the following devices:

- iPad
- Nook
- Other eReaders that support the `ePub` format.

For best results when viewing preformatted code blocks, adjust the landscape/portrait orientation and font size of your device to enable the maximum possible viewing area.

Trail: Collections

This section describes the Java Collections Framework. Here, you will learn what collections are and how they can make your job easier and programs better. You'll learn about the core elements — interfaces, implementations, aggregate operations, and algorithms — that comprise the Java Collections Framework.

Introduction tells you what collections are, and how they'll make your job easier and your programs better. You'll learn about the core elements that comprise the Collections Framework: *interfaces*, *implementations* and *algorithms*.

Interfaces describes the *core collection interfaces*, which are the heart and soul of the Java Collections Framework. You'll learn general guidelines for effective use of these interfaces, including when to use which interface. You'll also learn idioms for each interface that will help you get the most out of the interfaces.

Aggregate Operations iterate over collections on your behalf, which enable you to write more concise and efficient code that process elements stored in collections.

Implementations describes the JDK's *general-purpose collection implementations* and tells you when to use which implementation. You'll also learn about the *wrapper implementations*, which add functionality to general-purpose implementations.

Algorithms describes the *polymorphic algorithms* provided by the JDK to operate on collections. With any luck you'll never have to write your own sort routine again!

Custom Implementations tells you why you might want to write your own collection implementation (instead of using one of the general-purpose implementations provided by the JDK), and how you'd go about it. It's easy with the JDK's *abstract collection implementations*!

Interoperability tells you how the Collections Framework interoperates with older APIs that predate the addition of Collections to Java. Also, it tells you how to design new APIs so that they'll interoperate seamlessly with other new APIs.

- Legal Notices
- Supported Platforms

Trail: Collections: Table of Contents

Lesson: Introduction to Collections

A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers). If you have used the Java programming language — or just about any other programming language — you are already familiar with collections.

What Is a Collections Framework?

A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy. Historically, collections frameworks have been quite complex, which gave them a reputation for having a steep learning curve. We believe that the Java Collections Framework breaks with this tradition, as you will learn for yourself in this chapter.

Benefits of the Java Collections Framework

The Java Collections Framework provides the following benefits:

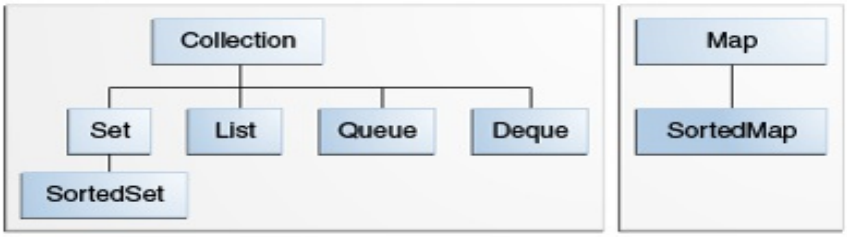
- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.

- **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
- **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Note: See online version of topics in this ebook to download complete source code.

Lesson: Interfaces

The *core collection interfaces* encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the following figure, the core collection interfaces form a hierarchy.



The core collection interfaces.

A `Set` is a special kind of `Collection`, a `SortedSet` is a special kind of `Set`, and so forth. Note also that the hierarchy consists of two distinct trees — a `Map` is not a true `Collection`.

Note that all the core collection interfaces are generic. For example, this is the declaration of the `Collection` interface.

```
public interface Collection<E>...
```

The `<E>` syntax tells you that the interface is generic. When you declare a `Collection` instance you can *and should* specify the type of object contained in the collection. Specifying the type allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime. For information on generic types, see the Generics (Updated) lesson.

When you understand how to use these interfaces, you will know most of what there is to know about the Java Collections Framework. This chapter discusses general guidelines for effective use of the interfaces, including when to use which interface. You'll also learn programming idioms for each interface to help you get the most out of it.

To keep the number of core collection interfaces manageable, the Java platform doesn't provide separate interfaces for each variant of each collection type. (Such variants might include immutable, fixed-size, and append-only.) Instead, the modification operations in each interface are designated *optional* — a given implementation may elect not to support all operations. If an unsupported operation is invoked, a collection throws an `UnsupportedOperationException`. Implementations are responsible for documenting which of the optional operations they support. All of the Java platform's general-purpose implementations support all of the optional operations.

The following list describes the core collection interfaces:

- `Collection` — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The `Collection` interface is the least common denominator that all

collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as `Set` and `List`. Also see The Collection Interface section.

- `Set` — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine. See also The Set Interface section.
- `List` — an ordered collection (sometimes called a *sequence*). `List`s can contain duplicate elements. The user of a `List` generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you've used `Vector`, you're familiar with the general flavor of `List`. Also see The List Interface section.
- `Queue` — a collection used to hold multiple elements prior to processing. Besides basic `Collection` operations, a `Queue` provides additional insertion, extraction, and inspection operations. Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to `remove` or `poll`. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties. Also see The Queue Interface section.
- `Deque` — a collection used to hold multiple elements prior to processing. Besides basic `Collection` operations, a `Deque` provides additional insertion, extraction, and inspection operations. Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends. Also see The Deque Interface section.
- `Map` — an object that maps keys to values. A `Map` cannot contain duplicate keys; each key can map to at most one value. If you've used `Hashtable`, you're already familiar with the basics of `Map`. Also see The Map Interface section.

The last two core collection interfaces are merely sorted versions of `Set` and `Map`:

- `SortedSet` — a `Set` that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls. Also see The SortedSet Interface section.
- `SortedMap` — a `Map` that maintains its mappings in ascending key order. This is the `Map` analog of `SortedSet`. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories. Also see The SortedMap Interface section.

To understand how the sorted interfaces maintain the order of their elements, see the Object Ordering section.

Note: See online version of topics in this ebook to download complete source code.

The Collection Interface

A `Collection` represents a group of objects known as its elements. The `Collection` interface is used to pass around collections of objects where maximum generality is desired. For example, by convention all general-purpose collection implementations have a constructor that takes a `Collection` argument. This constructor, known as a *conversion constructor*, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's subinterface or implementation type. In other words, it allows you to *convert* the collection's type.

Suppose, for example, that you have a `Collection<String> c`, which may be a `List`, a `Set`, or another kind of `Collection`. This idiom creates a new `ArrayList` (an implementation of the `List` interface), initially containing all the elements in `c`.

```
List<String> list = new ArrayList<String>(c);
```

Or if you are using JDK 7 or later you can use the diamond operator:

```
List<String> list = new ArrayList<>(c);
```

The `Collection` interface contains methods that perform basic operations, such as `int size()`, `boolean isEmpty()`, `boolean contains(Object element)`, `boolean add(E element)`, `boolean remove(Object element)`, and `Iterator<E> iterator()`.

It also contains methods that operate on entire collections, such as `boolean containsAll(Collection<?> c)`, `boolean addAll(Collection<? extends E> c)`, `boolean removeAll(Collection<?> c)`, `boolean retainAll(Collection<?> c)`, and `void clear()`.

Additional methods for array operations (such as `Object[] toArray()` and `<T> T[] toArray(T[] a)` exist as well.

In JDK 8 and later, the `Collection` interface also exposes methods `Stream<E> stream()` and `Stream<E> parallelStream()`, for obtaining sequential or parallel streams from the underlying collection. (See the lesson entitled Aggregate Operations for more information about using streams.)

The `Collection` interface does about what you'd expect given that a `Collection` represents a group of objects. It has methods that tell you how many elements are in the collection (`size`, `isEmpty`), methods that check whether a given object is in the collection (`contains`), methods that add and remove an element from the collection (`add`, `remove`), and methods that provide an iterator over the collection (`iterator`).

The `add` method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't. It guarantees that the `Collection` will contain the specified element after the call completes, and returns `true` if the `Collection` changes as a result of the call. Similarly, the `remove` method is designed to remove a single instance of the specified element from the `Collection`, assuming that it contains the element to start with, and to return `true` if the `Collection` was modified as a result.

Traversing Collections

There are three ways to traverse collections: (1) using aggregate operations (2) with the `for-each` construct and (3) by using `Iterator`s.

Aggregate Operations

In JDK 8 and later, the preferred method of iterating over a collection is to obtain a stream and perform aggregate operations on it. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

```
myShapesCollection.stream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e -> System.out.println(e.getName()));
```

Likewise, you could easily request a parallel stream, which might make sense if the collection is large enough and your computer has enough cores:

```
myShapesCollection.parallelStream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e -> System.out.println(e.getName()));
```

There are many different ways to collect data with this API. For example, you might want to convert the elements of a `Collection` to `String` objects, then join them, separated by commas:

```
String joined = elements.stream()
.map(Object::toString)
.collect(Collectors.joining(", "));
```

Or perhaps sum the salaries of all employees:

```
int total = employees.stream()
.collect(Collectors.summingInt(Employee::getSalary)));
```

These are but a few examples of what you can do with streams and aggregate operations. For more information and examples, see the lesson entitled Aggregate Operations.

The Collections framework has always provided a number of so-called "bulk operations" as part of its API. These include methods that operate on entire collections, such as `containsAll`, `addAll`, `removeAll`, etc. Do not confuse those methods with the aggregate operations that were introduced in JDK 8. The key difference between the new aggregate operations and the existing bulk operations (`containsAll`, `addAll`, etc.) is that the old versions are all *mutative*, meaning that they all modify the underlying collection. In contrast, the new aggregate operations *do not* modify the underlying collection. When using the new aggregate operations and lambda expressions, you must take care to avoid mutation so as not to introduce problems in the future, should your code be run later from a parallel stream.

for-each Construct

The `for-each` construct allows you to concisely traverse a collection or array using a `for` loop — see The for Statement. The following code uses the `for-each` construct to print out each element of a collection on a separate line.

```
for (Object o : collection)
    System.out.println(o);
```

Iterators

An `Iterator` is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an `Iterator` for a collection by calling its `iterator` method. The following is the `Iterator` interface.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

The `hasNext` method returns `true` if the iteration has more elements, and the `next` method returns the next element in the iteration. The `remove` method removes the last element that was returned by `next` from the underlying `Collection`. The `remove` method may be called only once per call to `next` and throws an exception if this rule is violated.

Note that `Iterator.remove` is the *only* safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

Use `Iterator` instead of the `for-each` construct when you need to:

- Remove the current element. The `for-each` construct hides the iterator, so you cannot call `remove`. Therefore, the `for-each` construct is not usable for filtering.
- Iterate over multiple collections in parallel.

The following method shows you how to use an `Iterator` to filter an arbitrary `Collection` — that is, traverse the collection removing specific elements.

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

This simple piece of code is polymorphic, which means that it works for *any* `Collection` regardless of implementation. This example demonstrates how easy it is to write a polymorphic algorithm using the Java Collections Framework.

Collection Interface Bulk Operations

Bulk operations perform an operation on an entire `Collection`. You could implement these shorthand operations using the basic operations, though in most cases such implementations would be less efficient. The following are the bulk operations:

- `containsAll` — returns `true` if the target `Collection` contains all of the elements in the specified `Collection`.
- `addAll` — adds all of the elements in the specified `Collection` to the target `Collection`.
- `removeAll` — removes from the target `Collection` all of its elements that are also contained in the specified `Collection`.
- `retainAll` — removes from the target `Collection` all its elements that are *not* also contained in the specified `Collection`. That is, it retains only those elements in the target `Collection` that are also contained in the specified `Collection`.
- `clear` — removes all elements from the `Collection`.

The `addAll`, `removeAll`, and `retainAll` methods all return `true` if the target `Collection` was modified in the process of executing the operation.

As a simple example of the power of bulk operations, consider the following idiom to remove *all* instances of a specified element, `e`, from a `Collection`, `c`.

```
c.removeAll(Collections.singleton(e));
```

More specifically, suppose you want to remove all of the `null` elements from a `Collection`.

```
c.removeAll(Collections.singleton(null));
```

This idiom uses `Collections.singleton`, which is a static factory method that returns an immutable `Set` containing only the specified element.

Collection Interface Array Operations

The `toArray` methods are provided as a bridge between collections and older APIs that expect arrays on input. The array operations allow the contents of a `Collection` to be translated into an array. The simple form with no arguments creates a new array of `Object`. The more complex form allows the caller to provide an array or to choose the runtime type of the output array.

For example, suppose that `c` is a `Collection`. The following snippet dumps the contents of `c` into a newly allocated array of `Object` whose length is identical to the number of elements in `c`.

```
Object[] a = c.toArray();
```

Suppose that `c` is known to contain only strings (perhaps because `c` is of type `Collection<String>`).

The following snippet dumps the contents of `c` into a newly allocated array of `String` whose length is identical to the number of elements in `c`.

```
String[] a = c.toArray(new String[0]);
```

The Set Interface

A `Set` is a `Collection` that cannot contain duplicate elements. It models the mathematical set abstraction. The `Set` interface contains *only* methods inherited from `Collection` and adds the restriction that duplicate elements are prohibited. `Set` also adds a stronger contract on the behavior of the `equals` and `hashCode` operations, allowing `Set` instances to be compared meaningfully even if their implementation types differ. Two `Set` instances are equal if they contain the same elements.

The Java platform contains three general-purpose `Set` implementations: `HashSet`, `TreeSet`, and `LinkedHashSet`. `HashSet`, which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration. `TreeSet`, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than `HashSet`. `LinkedHashSet`, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). `LinkedHashSet` spares its clients from the unspecified, generally chaotic ordering provided by `HashSet` at a cost that is only slightly higher.

Here's a simple but useful `Set` idiom. Suppose you have a `Collection`, `c`, and you want to create another `Collection` containing the same elements but with all duplicates eliminated. The following one-liner does the trick.

```
Collection<Type> noDups = new HashSet<Type>(c);
```

It works by creating a `Set` (which, by definition, cannot contain duplicates), initially containing all the elements in `c`. It uses the standard conversion constructor described in the The Collection Interface section.

Or, if using JDK 8 or later, you could easily collect into a `Set` using aggregate operations:

```
c.stream()
.collect(Collectors.toSet()); // no duplicates
```

Here's a slightly longer example that accumulates a `Collection` of names into a `TreeSet`:

```
Set<String> set = people.stream()
.map(Person::getName)
.collect(Collectors.toCollection(TreeSet::new));
```

And the following is a minor variant of the first idiom that preserves the order of the original collection while removing duplicate elements:

```
Collection<Type> noDups = new LinkedHashSet<Type>(c);
```

The following is a generic method that encapsulates the preceding idiom, returning a `Set` of the same

generic type as the one passed.

```
public static <E> Set<E> removeDups(Collection<E> c) {
    return new LinkedHashSet<E>(c);
}
```

Set Interface Basic Operations

The `size` operation returns the number of elements in the `Set` (its *cardinality*). The `isEmpty` method does exactly what you think it would. The `add` method adds the specified element to the `Set` if it is not already present and returns a boolean indicating whether the element was added. Similarly, the `remove` method removes the specified element from the `Set` if it is present and returns a boolean indicating whether the element was present. The `iterator` method returns an `Iterator` over the `Set`.

The following `program` prints out all distinct words in its argument list. Two versions of this program are provided. The first uses JDK 8 aggregate operations. The second uses the for-each construct.

Using JDK 8 Aggregate Operations:

```
import java.util.*;
import java.util.stream.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> distinctWords = Arrays.asList(args).stream()
                .collect(Collectors.toSet());
        System.out.println(distinctWords.size()+
                           " distinct words: " +
                           distinctWords);
    }
}
```

Using the `for-each` Construct:

```
import java.util.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
               s.add(a);
               System.out.println(s.size() + " distinct words: " + s);
    }
}
```

Now run either version of the program.

```
java FindDups i came i saw i left
```

The following output is produced:

```
4 distinct words: [left, came, saw, i]
```

Note that the code always refers to the `Collection` by its interface type (`Set`) rather than by its implementation type. This is a *strongly* recommended programming practice because it gives you the flexibility to change implementations merely by changing the constructor. If either of the variables used to store a collection or the parameters used to pass it around are declared to be of the `Collection`'s implementation type rather than its interface type, *all* such variables and parameters must be changed in order to change its implementation type.

Furthermore, there's no guarantee that the resulting program will work. If the program uses any nonstandard operations present in the original implementation type but not in the new one, the program will fail. Referring to collections only by their interface prevents you from using any nonstandard operations.

The implementation type of the `Set` in the preceding example is `HashSet`, which makes no guarantees as to the order of the elements in the `Set`. If you want the program to print the word list in alphabetical order, merely change the `Set`'s implementation type from `HashSet` to `TreeSet`. Making this trivial one-line change causes the command line in the previous example to generate the following output.

```
java FindDups i came i saw i left

4 distinct words: [came, i, left, saw]
```

Set Interface Bulk Operations

Bulk operations are particularly well suited to `Set`s; when applied, they perform standard set-algebraic operations. Suppose `s1` and `s2` are sets. Here's what bulk operations do:

- `s1.containsAll(s2)` — returns `true` if `s2` is a **subset** of `s1`. (`s2` is a subset of `s1` if set `s1` contains all of the elements in `s2`.)
- `s1.addAll(s2)` — transforms `s1` into the **union** of `s1` and `s2`. (The union of two sets is the set containing all of the elements contained in either set.)
- `s1.retainAll(s2)` — transforms `s1` into the intersection of `s1` and `s2`. (The intersection of two sets is the set containing only the elements common to both sets.)
- `s1.removeAll(s2)` — transforms `s1` into the (asymmetric) set difference of `s1` and `s2`. (For example, the set difference of `s1` minus `s2` is the set containing all of the elements found in `s1` but not in `s2`.)

To calculate the union, intersection, or set difference of two sets *nondestructively* (without modifying either set), the caller must copy one set before calling the appropriate bulk operation. The following are the resulting idioms.

```
Set<Type> union = new HashSet<Type>(s1);
union.addAll(s2);
```

```
Set<Type> intersection = new HashSet<Type>(s1);
intersection.retainAll(s2);

Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2);
```

The implementation type of the result `Set` in the preceding idioms is `HashSet`, which is, as already mentioned, the best all-around `Set` implementation in the Java platform. However, any general-purpose `Set` implementation could be substituted.

Let's revisit the `FindDups` program. Suppose you want to know which words in the argument list occur only once and which occur more than once, but you do not want any duplicates printed out repeatedly. This effect can be achieved by generating two sets — one containing every word in the argument list and the other containing only the duplicates. The words that occur only once are the set difference of these two sets, which we know how to compute. Here's how `the resulting program` looks.

```
import java.util.*;

public class FindDups2 {
    public static void main(String[] args) {
        Set<String> uniques = new HashSet<String>();
        Set<String> dups    = new HashSet<String>();

        for (String a : args)
            if (!uniques.add(a))
                dups.add(a);

        // Destructive set-difference
        uniques.removeAll(dups);

        System.out.println("Unique words:    " + uniques);
        System.out.println("Duplicate words: " + dups);
    }
}
```

When run with the same argument list used earlier (`i came i saw i left`), the program yields the following output.

```
Unique words:    [left, saw, came]
Duplicate words: [i]
```

A less common set-algebraic operation is the *symmetric set difference* — the set of elements contained in either of two specified sets but not in both. The following code calculates the symmetric set difference of two sets nondestructively.

```
Set<Type> symmetricDiff = new HashSet<Type>(s1);
symmetricDiff.addAll(s2);
Set<Type> tmp = new HashSet<Type>(s1);
tmp.retainAll(s2);
symmetricDiff.removeAll(tmp);
```

Set Interface Array Operations

The array operations don't do anything special for `Set`s beyond what they do for any other `Collection`. These operations are described in The Collection Interface section.

The List Interface

A `List` is an ordered `Collection` (sometimes called a *sequence*). Lists may contain duplicate elements. In addition to the operations inherited from `Collection`, the `List` interface includes operations for the following:

- `Positional access` — manipulates elements based on their numerical position in the list. This includes methods such as `get`, `set`, `add`, `addAll`, and `remove`.
- `Search` — searches for a specified object in the list and returns its numerical position. Search methods include `indexOf` and `lastIndexOf`.
- `Iteration` — extends `Iterator` semantics to take advantage of the list's sequential nature. The `listIterator` methods provide this behavior.
- `Range-view` — The `sublist` method performs arbitrary *range operations* on the list.

The Java platform contains two general-purpose `List` implementations. `ArrayList`, which is usually the better-performing implementation, and `LinkedList` which offers better performance under certain circumstances.

Collection Operations

The operations inherited from `Collection` all do about what you'd expect them to do, assuming you're already familiar with them. If you're not familiar with them from `Collection`, now would be a good time to read The Collection Interface section. The `remove` operation always removes *the first* occurrence of the specified element from the list. The `add` and `addAll` operations always append the new element(s) to the *end* of the list. Thus, the following idiom concatenates one list to another.

```
list1.addAll(list2);
```

Here's a nondestructive form of this idiom, which produces a third `List` consisting of the second list appended to the first.

```
List<Type> list3 = new ArrayList<Type>(list1);
list3.addAll(list2);
```

Note that the idiom, in its nondestructive form, takes advantage of `ArrayList`'s standard conversion constructor.

And here's an example (JDK 8 and later) that aggregates some names into a `List`:

```
List<String> list = people.stream()
.map(Person::getName)
.collect(Collectors.toList());
```

Like the `Set` interface, `List` strengthens the requirements on the `equals` and `hashCode` methods so that two `List` objects can be compared for logical equality without regard to their implementation

classes. Two `List` objects are equal if they contain the same elements in the same order.

Positional Access and Search Operations

The basic `positional access` operations are `get`, `set`, `add` and `remove`. (The `set` and `remove` operations return the old value that is being overwritten or removed.) Other operations (`indexOf` and `lastIndexOf`) return the first or last index of the specified element in the list.

The `addAll` operation inserts all the elements of the specified `Collection` starting at the specified position. The elements are inserted in the order they are returned by the specified `Collection`'s iterator. This call is the positional access analog of `Collection`'s `addAll` operation.

Here's a little method to swap two indexed values in a `List`.

```
public static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

Of course, there's one big difference. This is a polymorphic algorithm: It swaps two elements in any `List`, regardless of its implementation type. Here's another polymorphic algorithm that uses the preceding `swap` method.

```
public static void shuffle(List<?> list, Random rnd) {
    for (int i = list.size(); i > 1; i--)
        swap(list, i - 1, rnd.nextInt(i));
}
```

This algorithm, which is included in the Java platform's `Collections` class, randomly permutes the specified list using the specified source of randomness. It's a bit subtle: It runs up the list from the bottom, repeatedly swapping a randomly selected element into the current position. Unlike most naive attempts at shuffling, it's *fair* (all permutations occur with equal likelihood, assuming an unbiased source of randomness) and *fast* (requiring exactly `list.size()-1` swaps). The following program uses this algorithm to print the words in its argument list in random order.

```
import java.util.*;

public class Shuffle {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        for (String a : args)
            list.add(a);
        Collections.shuffle(list, new Random());
        System.out.println(list);
    }
}
```

In fact, this program can be made even shorter and faster. The `Arrays` class has a static factory method called `asList`, which allows an array to be viewed as a `List`. This method does not copy the

array. Changes in the `List` write through to the array and vice versa. The resulting List is not a general-purpose `List` implementation, because it doesn't implement the (optional) `add` and `remove` operations: Arrays are not resizable. Taking advantage of `Arrays.asList` and calling the library version of `shuffle`, which uses a default source of randomness, you get the following `tiny program` whose behavior is identical to the previous program.

```
import java.util.*;

public class Shuffle {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }
}
```

Iterators

As you'd expect, the `Iterator` returned by `List`'s `iterator` operation returns the elements of the list in proper sequence. `List` also provides a richer iterator, called a `ListIterator`, which allows you to traverse the list in either direction, modify the list during iteration, and obtain the current position of the iterator.

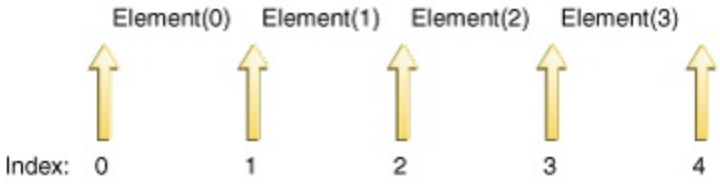
The three methods that `ListIterator` inherits from `Iterator` (`hasNext`, `next`, and `remove`) do exactly the same thing in both interfaces. The `hasPrevious` and the `previous` operations are exact analogues of `hasNext` and `next`. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The `previous` operation moves the cursor backward, whereas `next` moves it forward.

Here's the standard idiom for iterating backward through a list.

```
for (ListIterator<Type> it = list.listIterator(list.size()); it.hasPrevious(); ) {
    Type t = it.previous();
    ...
}
```

Note the argument to `listIterator` in the preceding idiom. The `List` interface has two forms of the `listIterator` method. The form with no arguments returns a `ListIterator` positioned at the beginning of the list; the form with an `int` argument returns a `ListIterator` positioned at the specified index. The index refers to the element that would be returned by an initial call to `next`. An initial call to `previous` would return the element whose index was `index-1`. In a list of length `n`, there are `n+1` valid values for `index`, from `0` to `n`, inclusive.

Intuitively speaking, the cursor is always between two elements — the one that would be returned by a call to `previous` and the one that would be returned by a call to `next`. The `n+1` valid `index` values correspond to the `n+1` gaps between elements, from the gap before the first element to the gap after the last one. The following figure shows the five possible cursor positions in a list containing four elements.



The five possible cursor positions.

Calls to `next` and `previous` can be intermixed, but you have to be a bit careful. The first call to `previous` returns the same element as the last call to `next`. Similarly, the first call to `next` after a sequence of calls to `previous` returns the same element as the last call to `previous`.

It should come as no surprise that the `nextIndex` method returns the index of the element that would be returned by a subsequent call to `next`, and `previousIndex` returns the index of the element that would be returned by a subsequent call to `previous`. These calls are typically used either to report the position where something was found or to record the position of the `ListIterator` so that another `ListIterator` with identical position can be created.

It should also come as no surprise that the number returned by `nextIndex` is always one greater than the number returned by `previousIndex`. This implies the behavior of the two boundary cases: (1) a call to `previousIndex` when the cursor is before the initial element returns `-1` and (2) a call to `nextIndex` when the cursor is after the final element returns `list.size()`. To make all this concrete, the following is a possible implementation of `List.indexOf`.

```
public int indexOf(E e) {
    for (ListIterator<E> it = listIterator(); it.hasNext(); )
        if (e == null ? it.next() == null : e.equals(it.next()))
            return it.previousIndex();
    // Element not found
    return -1;
}
```

Note that the `indexOf` method returns `it.previousIndex()` even though it is traversing the list in the forward direction. The reason is that `it.nextIndex()` would return the index of the element we are about to examine, and we want to return the index of the element we just examined.

The `Iterator` interface provides the `remove` operation to remove the last element returned by `next` from the `Collection`. For `ListIterator`, this operation removes the last element returned by `next` or `previous`. The `ListIterator` interface provides two additional operations to modify the list — `set` and `add`. The `set` method overwrites the last element returned by `next` or `previous` with the specified element. The following polymorphic algorithm uses `set` to replace all occurrences of one specified value with another.

```
public static <E> void replace(List<E> list, E val, E newVal) {
    for (ListIterator<E> it = list.listIterator(); it.hasNext(); )
        if (val == null ? it.next() == null : val.equals(it.next()))
            it.set(newVal);
}
```

The only bit of trickiness in this example is the equality test between `val` and `it.next`. You need to special-case a `val` value of `null` to prevent a `NullPointerException`.

The `add` method inserts a new element into the list immediately before the current cursor position. This method is illustrated in the following polymorphic algorithm to replace all occurrences of a specified value with the sequence of values contained in the specified list.

```
public static <E>
    void replace(List<E> list, E val, List<? extends E> newVals) {
    for (ListIterator<E> it = list.listIterator(); it.hasNext(); ){
        if (val == null ? it.next() == null : val.equals(it.next())) {
            it.remove();
            for (E e : newVals)
                it.add(e);
        }
    }
}
```

Range-View Operation

The `range-view` operation, `subList(int fromIndex, int toIndex)`, returns a `List` view of the portion of this list whose indices range from `fromIndex`, inclusive, to `toIndex`, exclusive. This *half-open range* mirrors the typical `for` loop.

```
for (int i = fromIndex; i < toIndex; i++) {
    ...
}
```

As the term *view* implies, the returned `List` is backed up by the `List` on which `subList` was called, so changes in the former are reflected in the latter.

This method eliminates the need for explicit range operations (of the sort that commonly exist for arrays). Any operation that expects a `List` can be used as a range operation by passing a `subList` view instead of a whole `List`. For example, the following idiom removes a range of elements from a `List`.

```
list.subList(fromIndex, toIndex).clear();
```

Similar idioms can be constructed to search for an element in a range.

```
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

Note that the preceding idioms return the index of the found element in the `subList`, not the index in the backing `List`.

Any polymorphic algorithm that operates on a `List`, such as the `replace` and `shuffle` examples, works with the `List` returned by `subList`.

Here's a polymorphic algorithm whose implementation uses `subList` to deal a hand from a deck. That is, it returns a new `List` (the "hand") containing the specified number of elements taken from the end of the specified `List` (the "deck"). The elements returned in the hand are removed from the deck.

```
public static <E> List<E> dealHand(List<E> deck, int n) {
    int deckSize = deck.size();
    List<E> handView = deck.subList(deckSize - n, deckSize);
    List<E> hand = new ArrayList<E>(handView);
    handView.clear();
    return hand;
}
```

Note that this algorithm removes the hand from the *end* of the deck. For many common `List` implementations, such as `ArrayList`, the performance of removing elements from the end of the list is substantially better than that of removing elements from the beginning.

The following is `a program` that uses the `dealHand` method in combination with `Collections.shuffle` to generate hands from a normal 52-card deck. The program takes two command-line arguments: (1) the number of hands to deal and (2) the number of cards in each hand.

```
import java.util.*;

public class Deal {
    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Usage: Deal hands cards");
            return;
        }
        int numHands = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);

        // Make a normal 52-card deck.
        String[] suit = new String[] {
            "spades", "hearts",
            "diamonds", "clubs"
        };
        String[] rank = new String[] {
            "ace", "2", "3", "4",
            "5", "6", "7", "8", "9", "10",
            "jack", "queen", "king"
        };

        List<String> deck = new ArrayList<String>();
        for (int i = 0; i < suit.length; i++)
            for (int j = 0; j < rank.length; j++)
                deck.add(rank[j] + " of " + suit[i]);

        // Shuffle the deck.
        Collections.shuffle(deck);

        if (numHands * cardsPerHand > deck.size()) {
            System.out.println("Not enough cards.");
            return;
        }

        for (int i = 0; i < numHands; i++)
            System.out.println(dealHand(deck, cardsPerHand));
    }

    public static <E> List<E> dealHand(List<E> deck, int n) {
```

```
        int deckSize = deck.size();
        List<E> handView = deck.subList(deckSize - n, deckSize);
        List<E> hand = new ArrayList<E>(handView);
        handView.clear();
        return hand;
    }
}
```

Running the program produces output like the following.

```
% java Deal 4 5

[8 of hearts, jack of spades, 3 of spades, 4 of spades,
    king of diamonds]
[4 of diamonds, ace of clubs, 6 of clubs, jack of hearts,
    queen of hearts]
[7 of spades, 5 of spades, 2 of diamonds, queen of diamonds,
    9 of clubs]
[8 of spades, 6 of diamonds, ace of spades, 3 of hearts,
    ace of hearts]
```

Although the `subList` operation is extremely powerful, some care must be exercised when using it. The semantics of the `List` returned by `subList` become undefined if elements are added to or removed from the backing `List` in any way other than via the returned `List`. Thus, it's highly recommended that you use the `List` returned by `subList` only as a transient object — to perform one or a sequence of range operations on the backing `List`. The longer you use the `subList` instance, the greater the probability that you'll compromise it by modifying the backing `List` directly or through another `subList` object. Note that it is legal to modify a sublist of a sublist and to continue using the original sublist (though not concurrently).

List Algorithms

Most polymorphic algorithms in the `Collections` class apply specifically to `List`. Having all these algorithms at your disposal makes it very easy to manipulate lists. Here's a summary of these algorithms, which are described in more detail in the Algorithms section.

- `sort` — sorts a `List` using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
- `shuffle` — randomly permutes the elements in a `List`.
- `reverse` — reverses the order of the elements in a `List`.
- `rotate` — rotates all the elements in a `List` by a specified distance.
- `swap` — swaps the elements at specified positions in a `List`.
- `replaceAll` — replaces all occurrences of one specified value with another.
- `fill` — overwrites every element in a `List` with the specified value.
- `copy` — copies the source `List` into the destination `List`.
- `binarySearch` — searches for an element in an ordered `List` using the binary search algorithm.
- `indexOfSubList` — returns the index of the first sublist of one `List` that is equal to another.
- `lastIndexOfSubList` — returns the index of the last sublist of one `List` that is equal to another.

The Queue Interface

A `Queue` is a collection for holding elements prior to processing. Besides basic `Collection` operations, queues provide additional insertion, removal, and inspection operations. The `Queue` interface follows.

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

Each `Queue` method exists in two forms: (1) one throws an exception if the operation fails, and (2) the other returns a special value if the operation fails (either `null` or `false`, depending on the operation). The regular structure of the interface is illustrated in the following table.

Queue Interface Structure

Type of Operation	Throws exception	Returns special value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to their values — see the Object Ordering section for details). Whatever ordering is used, the head of the queue is the element that would be removed by a call to `remove` or `poll`. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties.

It is possible for a `Queue` implementation to restrict the number of elements that it holds; such queues are known as *bounded*. Some `Queue` implementations in `java.util.concurrent` are bounded, but the implementations in `java.util` are not.

The `add` method, which `Queue` inherits from `Collection`, inserts an element unless it would violate the queue's capacity restrictions, in which case it throws `IllegalStateException`. The `offer` method, which is intended solely for use on bounded queues, differs from `add` only in that it indicates failure to insert an element by returning `false`.

The `remove` and `poll` methods both remove and return the head of the queue. Exactly which element gets removed is a function of the queue's ordering policy. The `remove` and `poll` methods differ in

their behavior only when the queue is empty. Under these circumstances, `remove` throws `NoSuchElementException`, while `poll` returns `null`.

The `element` and `peek` methods return, but do not remove, the head of the queue. They differ from one another in precisely the same fashion as `remove` and `poll`: If the queue is empty, `element` throws `NoSuchElementException`, while `peek` returns `null`.

`Queue` implementations generally do not allow insertion of `null` elements. The `LinkedList` implementation, which was retrofitted to implement `Queue`, is an exception. For historical reasons, it permits `null` elements, but you should refrain from taking advantage of this, because `null` is used as a special return value by the `poll` and `peek` methods.

Queue implementations generally do not define element-based versions of the `equals` and `hashCode` methods but instead inherit the identity-based versions from `Object`.

The `Queue` interface does not define the blocking queue methods, which are common in concurrent programming. These methods, which wait for elements to appear or for space to become available, are defined in the interface `java.util.concurrent.BlockingQueue`, which extends `Queue`.

In the following example program, a queue is used to implement a countdown timer. The queue is preloaded with all the integer values from a number specified on the command line to zero, in descending order. Then, the values are removed from the queue and printed at one-second intervals. The program is artificial in that it would be more natural to do the same thing without using a queue, but it illustrates the use of a queue to store elements prior to subsequent processing.

```
import java.util.*;

public class Countdown {
    public static void main(String[] args) throws InterruptedException {
        int time = Integer.parseInt(args[0]);
        Queue<Integer> queue = new LinkedList<Integer>();

        for (int i = time; i >= 0; i--)
            queue.add(i);

        while (!queue.isEmpty()) {
            System.out.println(queue.remove());
            Thread.sleep(1000);
        }
    }
}
```

In the following example, a priority queue is used to sort a collection of elements. Again this program is artificial in that there is no reason to use it in favor of the `sort` method provided in `Collections`, but it illustrates the behavior of priority queues.

```
static <E> List<E> heapSort(Collection<E> c) {
    Queue<E> queue = new PriorityQueue<E>(c);
    List<E> result = new ArrayList<E>();

    while (!queue.isEmpty())
        result.add(queue.remove());
```

```
    return result;
}
```

The Deque Interface

Usually pronounced as `deck`, a deque is a double-ended-queue. A double-ended-queue is a linear collection of elements that supports the insertion and removal of elements at both end points. The `Deque` interface is a richer abstract data type than both `Stack` and `Queue` because it implements both stacks and queues at the same time. The `Deque` interface, defines methods to access the elements at both ends of the `Deque` instance. Methods are provided to insert, remove, and examine the elements. Predefined classes like `ArrayDeque` and `LinkedList` implement the `Deque` interface.

Note that the `Deque` interface can be used both as last-in-first-out stacks and first-in-first-out queues. The methods given in the `Deque` interface are divided into three parts:

Insert

The `addfirst` and `offerFirst` methods insert elements at the beginning of the `Deque` instance. The methods `addLast` and `offerLast` insert elements at the end of the `Deque` instance. When the capacity of the `Deque` instance is restricted, the preferred methods are `offerFirst` and `offerLast` because `addFirst` might fail to throw an exception if it is full.

Remove

The `removeFirst` and `pollFirst` methods remove elements from the beginning of the `Deque` instance. The `removeLast` and `pollLast` methods remove elements from the end. The methods `pollFirst` and `pollLast` return `null` if the `Deque` is empty whereas the methods `removeFirst` and `removeLast` throw an exception if the `Deque` instance is empty.

Retrieve

The methods `getFirst` and `peekFirst` retrieve the first element of the `Deque` instance. These methods dont remove the value from the `Deque` instance. Similarly, the methods `getLast` and `peekLast` retrieve the last element. The methods `getFirst` and `getLast` throw an exception if the `deque` instance is empty whereas the methods `peekFirst` and `peekLast` return `NULL`.

The 12 methods for insertion, removal and retieval of Deque elements are summarized in the following table:

Deque Methods

Type of Operation	First Element (Beginning of the `Deque` instance)	Last Element (End of the `Deque` instance)
Insert	`addFirst(e)` `offerFirst(e)`	`addLast(e)` `offerLast(e)`
Remove	`removeFirst()` `pollFirst()`	`removeLast()` `pollLast()`

Examine	`getFirst()` `peekFirst()`	`getLast()` `peekLast()`

In addition to these basic methods to insert,remove and examine a `Deque` instance, the `Deque` interface also has some more predefined methods. One of these is `removeFirstOccurence`, this method removes the first occurence of the specified element if it exists in the `Deque` instance. If the element does not exist then the `Deque` instance remains unchanged. Another similar method is `removeLastOccurence`; this method removes the last occurence of the specified element in the `Deque` instance. The return type of these methods is `boolean`, and they return `true` if the element exists in the `Deque` instance.

The Map Interface

A `Map` is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical *function* abstraction. The `Map` interface includes methods for basic operations (such as `put`, `get`, `remove`, `containsKey`, `containsValue`, `size`, and `empty`), bulk operations (such as `putAll` and `clear`), and collection views (such as `keySet`, `entrySet`, and `values`).

The Java platform contains three general-purpose `Map` implementations: `HashMap`, `TreeMap`, and `LinkedHashMap`. Their behavior and performance are precisely analogous to `HashSet`, `TreeSet`, and `LinkedHashSet`, as described in The Set Interface section.

The remainder of this page discusses the `Map` interface in detail. But first, here are some more examples of collecting to `Map`s using JDK 8 aggregate operations. Modeling real-world objects is a common task in object-oriented programming, so it is reasonable to think that some programs might, for example, group employees by department:

```
// Group employees by department
Map<Department, List<Employee>> byDept = employees.stream()
.collect(Collectors.groupingBy(Employee::getDepartment));
```

Or compute the sum of all salaries by department:

```
// Compute sum of salaries by department
Map<Department, Integer> totalByDept = employees.stream()
.collect(Collectors.groupingBy(Employee::getDepartment,
Collectors.summingInt(Employee::getSalary)));
```

Or perhaps group students by passing or failing grades:

```
// Partition students into passing and failing
Map<Boolean, List<Student>> passingFailing = students.stream()
.collect(Collectors.partitioningBy(s -> s.getGrade()>= PASS_THRESHOLD));
```

You could also group people by city:

```
// Classify Person objects by city
Map<String, List<Person>> peopleByCity
        = personStream.collect(Collectors.groupingBy(Person::getCity));
```

Or even cascade two collectors to classify people by state and city:

```
// Cascade Collectors
Map<String, Map<String, List<Person>>> peopleByStateAndCity
  = personStream.collect(Collectors.groupingBy(Person::getState,
  Collectors.groupingBy(Person::getCity)))
```

Again, these are but a few examples of how to use the new JDK 8 APIs. For in-depth coverage of lambda expressions and aggregate operations see the lesson entitled Aggregate Operations.

Map Interface Basic Operations

The basic operations of `Map` (`put`, `get`, `containsKey`, `containsValue`, `size`, and `isEmpty`) behave exactly like their counterparts in `Hashtable`. The `following program` generates a frequency table of the words found in its argument list. The frequency table maps each word to the number of times it occurs in the argument list.

```
import java.util.*;

public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();

        // Initialize frequency table from command line
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }

        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}
```

The only tricky thing about this program is the second argument of the `put` statement. That argument is a conditional expression that has the effect of setting the frequency to one if the word has never been seen before or one more than its current value if the word has already been seen. Try running this program with the command:

```
java Freq if it is to be it is up to me to delegate
```

The program yields the following output.

```
8 distinct words:
{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}
```

Suppose you'd prefer to see the frequency table in alphabetical order. All you have to do is change the implementation type of the `Map` from `HashMap` to `TreeMap`. Making this four-character change causes the program to generate the following output from the same command line.

```
8 distinct words:
{be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}
```

Similarly, you could make the program print the frequency table in the order the words first appear on the command line simply by changing the implementation type of the map to `LinkedHashMap`. Doing so results in the following output.

```
8 distinct words:
{if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}
```

This flexibility provides a potent illustration of the power of an interface-based framework.

Like the `Set`and `List`interfaces, `Map` strengthens the requirements on the `equals` and `hashCode` methods so that two `Map` objects can be compared for logical equality without regard to their implementation types. Two `Map` instances are equal if they represent the same key-value mappings.

By convention, all general-purpose `Map` implementations provide constructors that take a `Map` object and initialize the new `Map` to contain all the key-value mappings in the specified `Map`. This standard `Map` conversion constructor is entirely analogous to the standard `Collection` constructor: It allows the caller to create a `Map` of a desired implementation type that initially contains all of the mappings in another `Map`, regardless of the other `Map`'s implementation type. For example, suppose you have a `Map`, named `m`. The following one-liner creates a new `HashMap` initially containing all of the same key-value mappings as `m`.

```
Map<K, V> copy = new HashMap<K, V>(m);
```

Map Interface Bulk Operations

The `clear` operation does exactly what you would think it could do: It removes all the mappings from the `Map`. The `putAll` operation is the `Map` analogue of the `Collection` interface's `addAll` operation. In addition to its obvious use of dumping one `Map` into another, it has a second, more subtle use. Suppose a `Map` is used to represent a collection of attribute-value pairs; the `putAll` operation, in combination with the `Map` conversion constructor, provides a neat way to implement attribute map creation with default values. The following is a static factory method that demonstrates this technique.

```
static <K, V> Map<K, V> newAttributeMap(Map<K, V>defaults, Map<K, V> overrides) {
    Map<K, V> result = new HashMap<K, V>(defaults);
    result.putAll(overrides);
    return result;
}
```

Collection Views

The `Collection` view methods allow a `Map` to be viewed as a `Collection` in these three ways:

- `keySet` — the `Set` of keys contained in the `Map`.
- `values` — The `Collection` of values contained in the `Map`. This `Collection` is not a `Set`, because multiple keys can map to the same value.
- `entrySet` — the `Set` of key-value pairs contained in the `Map`. The `Map` interface provides a small nested interface called `Map.Entry`, the type of the elements in this `Set`.

The `Collection` views provide the *only* means to iterate over a `Map`. This example illustrates the standard idiom for iterating over the keys in a `Map` with a `for-each` construct:

```
for (KeyType key : m.keySet())
    System.out.println(key);
```

and with an `iterator`:

```
// Filter a map based on some
// property of its keys.
for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )
    if (it.next().isBogus())
        it.remove();
```

The idiom for iterating over values is analogous. Following is the idiom for iterating over key-value pairs.

```
for (Map.Entry<KeyType, ValType> e : m.entrySet())
    System.out.println(e.getKey() + ": " + e.getValue());
```

At first, many people worry that these idioms may be slow because the `Map` has to create a new `Collection` instance each time a `Collection` view operation is called. Rest easy: There's no reason that a `Map` cannot always return the same object each time it is asked for a given `Collection` view. This is precisely what all the `Map` implementations in `java.util` do.

With all three `Collection` views, calling an `Iterator`'s `remove` operation removes the associated entry from the backing `Map`, assuming that the backing `Map` supports element removal to begin with. This is illustrated by the preceding filtering idiom.

With the `entrySet` view, it is also possible to change the value associated with a key by calling a `Map.Entry`'s `setValue` method during iteration (again, assuming the `Map` supports value modification to begin with). Note that these are the *only* safe ways to modify a `Map` during iteration; the behavior is unspecified if the underlying `Map` is modified in any other way while the iteration is in progress.

The `Collection` views support element removal in all its many forms — `remove`, `removeAll`, `retainAll`, and `clear` operations, as well as the `Iterator.remove` operation. (Yet again, this assumes that the backing `Map` supports element removal.)

The `Collection` views *do not* support element addition under any circumstances. It would make no sense for the `keySet` and `values` views, and it's unnecessary for the `entrySet` view, because the backing `Map`'s `put` and `putAll` methods provide the same functionality.

Fancy Uses of Collection Views: Map Algebra

When applied to the `Collection` views, bulk operations (`containsAll`, `removeAll`, and `retainAll`) are surprisingly potent tools. For starters, suppose you want to know whether one `Map` is a submap of another — that is, whether the first `Map` contains all the key-value mappings in the second. The following idiom does the trick.

```
if (m1.entrySet().containsAll(m2.entrySet())) {
    ...
}
```

Along similar lines, suppose you want to know whether two `Map` objects contain mappings for all of the same keys.

```
if (m1.keySet().equals(m2.keySet())) {
    ...
}
```

Suppose you have a `Map` that represents a collection of attribute-value pairs, and two `Set`s representing required attributes and permissible attributes. (The permissible attributes include the required attributes.) The following snippet determines whether the attribute map conforms to these constraints and prints a detailed error message if it doesn't.

```
static <K, V> boolean validate(Map<K, V> attrMap, Set<K> requiredAttrs, Set<K>permittedAttrs) {
    boolean valid = true;
    Set<K> attrs = attrMap.keySet();

    if (! attrs.containsAll(requiredAttrs)) {
        Set<K> missing = new HashSet<K>(requiredAttrs);
        missing.removeAll(attrs);
        System.out.println("Missing attributes: " + missing);
        valid = false;
    }
    if (! permittedAttrs.containsAll(attrs)) {
        Set<K> illegal = new HashSet<K>(attrs);
        illegal.removeAll(permittedAttrs);
        System.out.println("Illegal attributes: " + illegal);
        valid = false;
    }
    return valid;
}
```

Suppose you want to know all the keys common to two `Map` objects.

```
Set<KeyType>commonKeys = new HashSet<KeyType>(m1.keySet());
commonKeys.retainAll(m2.keySet());
```

A similar idiom gets you the common values.

All the idioms presented thus far have been nondestructive; that is, they don't modify the backing `Map`. Here are a few that do. Suppose you want to remove all of the key-value pairs that one `Map` has in common with another.

```
m1.entrySet().removeAll(m2.entrySet());
```

Suppose you want to remove from one `Map` all of the keys that have mappings in another.

```
m1.keySet().removeAll(m2.keySet());
```

What happens when you start mixing keys and values in the same bulk operation? Suppose you have a `Map`, `managers`, that maps each employee in a company to the employee's manager. We'll be deliberately vague about the types of the key and the value objects. It doesn't matter, as long as they're the same. Now suppose you want to know who all the "individual contributors" (or nonmanagers) are. The following snippet tells you exactly what you want to know.

```
Set<Employee> individualContributors = new HashSet<Employee>(managers.keySet());
individualContributors.removeAll(managers.values());
```

Suppose you want to fire all the employees who report directly to some manager, Simon.

```
Employee simon = ... ;
managers.values().removeAll(Collections.singleton(simon));
```

Note that this idiom makes use of `Collections.singleton`, a static factory method that returns an immutable `Set` with the single, specified element.

Once you've done this, you may have a bunch of employees whose managers no longer work for the company (if any of Simon's direct-reports were themselves managers). The following code will tell you which employees have managers who no longer works for the company.

```
Map<Employee, Employee> m = new HashMap<Employee, Employee>(managers);
m.values().removeAll(managers.keySet());
Set<Employee> slackers = m.keySet();
```

This example is a bit tricky. First, it makes a temporary copy of the `Map`, and it removes from the temporary copy all entries whose (manager) value is a key in the original `Map`. Remember that the original `Map` has an entry for each employee. Thus, the remaining entries in the temporary `Map` comprise all the entries from the original `Map` whose (manager) values are no longer employees. The keys in the temporary copy, then, represent precisely the employees that we're looking for.

There are many more idioms like the ones contained in this section, but it would be impractical and tedious to list them all. Once you get the hang of it, it's not that difficult to come up with the right one when you need it.

Multimaps

A *multimap* is like a `Map` but it can map each key to multiple values. The Java Collections Framework doesn't include an interface for multimaps because they aren't used all that commonly. It's a fairly simple matter to use a `Map` whose values are `List` instances as a multimap. This technique is demonstrated in the next code example, which reads a word list containing one word per line (all lowercase) and prints out all the anagram groups that meet a size criterion. An *anagram group* is a bunch of words, all of which contain exactly the same letters but in a different order. The program

takes two arguments on the command line: (1) the name of the dictionary file and (2) the minimum size of anagram group to print out. Anagram groups containing fewer words than the specified minimum are not printed.

There is a standard trick for finding anagram groups: For each word in the dictionary, alphabetize the letters in the word (that is, reorder the word's letters into alphabetical order) and put an entry into a multimap, mapping the alphabetized word to the original word. For example, the word *bad* causes an entry mapping *abd* into *bad* to be put into the multimap. A moment's reflection will show that all the words to which any given key maps form an anagram group. It's a simple matter to iterate over the keys in the multimap, printing out each anagram group that meets the size constraint.

`The following program` is a straightforward implementation of this technique.

```
import java.util.*;
import java.io.*;

public class Anagrams {
    public static void main(String[] args) {
        int minGroupSize = Integer.parseInt(args[1]);

        // Read words from file and put into a simulated multimap
        Map<String, List<String>> m = new HashMap<String, List<String>>();

        try {
            Scanner s = new Scanner(new File(args[0]));
            while (s.hasNext()) {
                String word = s.next();
                String alpha = alphabetize(word);
                List<String> l = m.get(alpha);
                if (l == null)
                    m.put(alpha, l=new ArrayList<String>());
                l.add(word);
            }
        } catch (IOException e) {
            System.err.println(e);
            System.exit(1);
        }

        // Print all permutation groups above size threshold
        for (List<String> l : m.values())
            if (l.size() >= minGroupSize)
                System.out.println(l.size() + ": " + l);
    }

    private static String alphabetize(String s) {
        char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    }
}
```

Running this program on a 173,000-word dictionary file with a minimum anagram group size of eight produces the following output.

```
9: [estrin, inerts, insert, inters, niters, nitres, sinter,
     triens, trines]
8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]
8: [aspers, parses, passer, prases, repass, spares, sparse,
```

```
     spears]
10: [least, setal, slate, stale, steal, stela, taels, tales,
      teals, tesla]
8: [enters, nester, renest, rentes, resent, tenser, ternes,
     treens]
8: [arles, earls, lares, laser, lears, rales, reals, seral]
8: [earings, erasing, gainers, reagins, regains, reginas,
     searing, seringa]
8: [peris, piers, pries, prise, ripes, speir, spier, spire]
12: [apers, apres, asper, pares, parse, pears, prase, presa,
      rapes, reaps, spare, spear]
11: [alerts, alters, artels, estral, laster, ratels, salter,
      slater, staler, stelar, talers]
9: [capers, crapes, escarp, pacers, parsec, recaps, scrape,
     secpar, spacer]
9: [palest, palets, pastel, petals, plates, pleats, septal,
     staple, tepals]
9: [anestri, antsier, nastier, ratines, retains, retinas,
     retsina, stainer, stearin]
8: [ates, east, eats, etas, sate, seat, seta, teas]
8: [carets, cartes, caster, caters, crates, reacts, recast,
     traces]
```

Many of these words seem a bit bogus, but that's not the program's fault; they're in the dictionary file. Here's the `dictionary file` we used. It was derived from the Public Domain ENABLE benchmark reference word list.

Object Ordering

A `List l` may be sorted as follows.

```
Collections.sort(l);
```

If the `List` consists of `String` elements, it will be sorted into alphabetical order. If it consists of `Date` elements, it will be sorted into chronological order. How does this happen? `String` and `Date` both implement the `Comparable` interface. `Comparable` implementations provide a *natural ordering* for a class, which allows objects of that class to be sorted automatically. The following table summarizes some of the more important Java platform classes that implement `Comparable`.

Classes Implementing Comparable

Class	Natural Ordering
`Byte`	Signed numerical
`Character`	Unsigned numerical
`Long`	Signed numerical
`Integer`	Signed numerical
`Short`	Signed numerical
`Double`	Signed numerical
`Float`	Signed numerical
`BigInteger`	Signed numerical
`BigDecimal`	Signed numerical
`Boolean`	`Boolean.FALSE < Boolean.TRUE`
`File`	System-dependent lexicographic on path name
`String`	Lexicographic
`Date`	Chronological
`CollationKey`	Locale-specific lexicographic

If you try to sort a list, the elements of which do not implement `Comparable`, `Collections.sort(list)` will throw a `ClassCastException`. Similarly, `Collections.sort(list, comparator)` will throw a `ClassCastException` if you try to sort a list whose elements cannot be compared to one another using the `comparator`. Elements that can be compared to one another are called *mutually comparable*. Although elements of different types may be mutually comparable, none of the classes listed here permit interclass comparison.

This is all you really need to know about the `Comparable` interface if you just want to sort lists of

comparable elements or to create sorted collections of them. The next section will be of interest to you if you want to implement your own `Comparable` type.

Writing Your Own Comparable Types

The `Comparable` interface consists of the following method.

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

The `compareTo` method compares the receiving object with the specified object and returns a negative integer, 0, or a positive integer depending on whether the receiving object is less than, equal to, or greater than the specified object. If the specified object cannot be compared to the receiving object, the method throws a `ClassCastException`.

The `following class representing a person's name` implements `Comparable`.

```
import java.util.*;

public class Name implements Comparable<Name> {
    private final String firstName, lastName;

    public Name(String firstName, String lastName) {
        if (firstName == null || lastName == null)
            throw new NullPointerException();
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String firstName() { return firstName; }
    public String lastName()  { return lastName;  }

    public boolean equals(Object o) {
        if (!(o instanceof Name))
            return false;
        Name n = (Name) o;
        return n.firstName.equals(firstName) && n.lastName.equals(lastName);
    }

    public int hashCode() {
        return 31*firstName.hashCode() + lastName.hashCode();
    }

    public String toString() {
        return firstName + " " + lastName;
    }

    public int compareTo(Name n) {
        int lastCmp = lastName.compareTo(n.lastName);
        return (lastCmp != 0 ? lastCmp : firstName.compareTo(n.firstName));
    }
}
```

To keep the preceding example short, the class is somewhat limited: It doesn't support middle names, it demands both a first and a last name, and it is not internationalized in any way. Nonetheless, it illustrates the following important points:

- `Name` objects are *immutable*. All other things being equal, immutable types are the way to go, especially for objects that will be used as elements in `Set`s or as keys in `Map`s. These collections will break if you modify their elements or keys while they're in the collection.
- The constructor checks its arguments for `null`. This ensures that all `Name` objects are well formed so that none of the other methods will ever throw a `NullPointerException`.
- The `hashCode` method is redefined. This is essential for any class that redefines the `equals` method. (Equal objects must have equal hash codes.)
- The `equals` method returns `false` if the specified object is `null` or of an inappropriate type. The `compareTo` method throws a runtime exception under these circumstances. Both of these behaviors are required by the general contracts of the respective methods.
- The `toString` method has been redefined so it prints the `Name` in human-readable form. This is always a good idea, especially for objects that are going to get put into collections. The various collection types' `toString` methods depend on the `toString` methods of their elements, keys, and values.

Since this section is about element ordering, let's talk a bit more about `Name`'s `compareTo` method. It implements the standard name-ordering algorithm, where last names take precedence over first names. This is exactly what you want in a natural ordering. It would be very confusing indeed if the natural ordering were unnatural!

Take a look at how `compareTo` is implemented, because it's quite typical. First, you compare the most significant part of the object (in this case, the last name). Often, you can just use the natural ordering of the part's type. In this case, the part is a `String` and the natural (lexicographic) ordering is exactly what's called for. If the comparison results in anything other than zero, which represents equality, you're done: You just return the result. If the most significant parts are equal, you go on to compare the next most-significant parts. In this case, there are only two parts — first name and last name. If there were more parts, you'd proceed in the obvious fashion, comparing parts until you found two that weren't equal or you were comparing the least-significant parts, at which point you'd return the result of the comparison.

Just to show that it all works, here's `a program that builds a list of names and sorts them`.

```
import java.util.*;

public class NameSort {
    public static void main(String[] args) {
        Name nameArray[] = {
            new Name("John", "Smith"),
            new Name("Karl", "Ng"),
            new Name("Jeff", "Smith"),
            new Name("Tom", "Rich")
        };

        List<Name> names = Arrays.asList(nameArray);
        Collections.sort(names);
        System.out.println(names);
    }
}
```

If you run this program, here's what it prints.

```
[Karl Ng, Tom Rich, Jeff Smith, John Smith]
```

There are four restrictions on the behavior of the `compareTo` method, which we won't go into now because they're fairly technical and boring and are better left in the API documentation. It's really important that all classes that implement `Comparable` obey these restrictions, so read the documentation for `Comparable` if you're writing a class that implements it. Attempting to sort a list of objects that violate the restrictions has undefined behavior. Technically speaking, these restrictions ensure that the natural ordering is a *total order* on the objects of a class that implements it; this is necessary to ensure that sorting is well defined.

Comparators

What if you want to sort some objects in an order other than their natural ordering? Or what if you want to sort some objects that don't implement `Comparable`? To do either of these things, you'll need to provide a `Comparator` — an object that encapsulates an ordering. Like the `Comparable` interface, the `Comparator` interface consists of a single method.

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

The `compare` method compares its two arguments, returning a negative integer, 0, or a positive integer depending on whether the first argument is less than, equal to, or greater than the second. If either of the arguments has an inappropriate type for the `Comparator`, the `compare` method throws a `ClassCastException`.

Much of what was said about `Comparable` applies to `Comparator` as well. Writing a `compare` method is nearly identical to writing a `compareTo` method, except that the former gets both objects passed in as arguments. The `compare` method has to obey the same four technical restrictions as `Comparable`'s `compareTo` method for the same reason — a `Comparator` must induce a total order on the objects it compares.

Suppose you have a class called `Employee`, as follows.

```
public class Employee implements Comparable<Employee> {
    public Name name()     { ... }
    public int number()    { ... }
    public Date hireDate() { ... }
       ...
}
```

Let's assume that the natural ordering of `Employee` instances is `Name` ordering (as defined in the previous example) on employee name. Unfortunately, the boss has asked for a list of employees in order of seniority. This means we have to do some work, but not much. The following program will

produce the required list.

```
import java.util.*;
public class EmpSort {
    static final Comparator<Employee> SENIORITY_ORDER =
                                        new Comparator<Employee>() {
            public int compare(Employee e1, Employee e2) {
                return e2.hireDate().compareTo(e1.hireDate());
            }
    };

    // Employee database
    static final Collection<Employee> employees = ... ;

    public static void main(String[] args) {
        List<Employee> e = new ArrayList<Employee>(employees);
        Collections.sort(e, SENIORITY_ORDER);
        System.out.println(e);
    }
}
```

The `Comparator` in the program is reasonably straightforward. It relies on the natural ordering of `Date` applied to the values returned by the `hireDate` accessor method. Note that the `Comparator` passes the hire date of its second argument to its first rather than vice versa. The reason is that the employee who was hired most recently is the least senior; sorting in the order of hire date would put the list in reverse seniority order. Another technique people sometimes use to achieve this effect is to maintain the argument order but to negate the result of the comparison.

```
// Don't do this!!
return -r1.hireDate().compareTo(r2.hireDate());
```

You should always use the former technique in favor of the latter because the latter is not guaranteed to work. The reason for this is that the `compareTo` method can return any negative `int` if its argument is less than the object on which it is invoked. There is one negative `int` that remains negative when negated, strange as it may seem.

```
-Integer.MIN_VALUE == Integer.MIN_VALUE
```

The `Comparator` in the preceding program works fine for sorting a `List`, but it does have one deficiency: It cannot be used to order a sorted collection, such as `TreeSet`, because it generates an ordering that is *not compatible with* equals. This means that this `Comparator` equates objects that the `equals` method does not. In particular, any two employees who were hired on the same date will compare as equal. When you're sorting a `List`, this doesn't matter; but when you're using the `Comparator` to order a sorted collection, it's fatal. If you use this `Comparator` to insert multiple employees hired on the same date into a `TreeSet`, only the first one will be added to the set; the second will be seen as a duplicate element and will be ignored.

To fix this problem, simply tweak the `Comparator` so that it produces an ordering that *is compatible with* `equals`. In other words, tweak it so that the only elements seen as equal when using `compare` are those that are also seen as equal when compared using `equals`. The way to do this is to perform a

two-part comparison (as for `Name`), where the first part is the one we're interested in — in this case, the hire date — and the second part is an attribute that uniquely identifies the object. Here the employee number is the obvious attribute. This is the `Comparator` that results.

```
static final Comparator<Employee> SENIORITY_ORDER =
                                        new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        int dateCmp = e2.hireDate().compareTo(e1.hireDate());
        if (dateCmp != 0)
            return dateCmp;

        return (e1.number() < e2.number() ? -1 :
                (e1.number() == e2.number() ? 0 : 1));
    }
};
```

One last note: You might be tempted to replace the final `return` statement in the `Comparator` with the simpler:

```
return e1.number() - e2.number();
```

Don't do it unless you're *absolutely sure* no one will ever have a negative employee number! This trick does not work in general because the signed integer type is not big enough to represent the difference of two arbitrary signed integers. If `i` is a large positive integer and `j` is a large negative integer, `i - j` will overflow and will return a negative integer. The resulting `comparator` violates one of the four technical restrictions we keep talking about (transitivity) and produces horrible, subtle bugs. This is not a purely theoretical concern; people get burned by it.

The SortedSet Interface

A `SortedSet` is a `Set` that maintains its elements in ascending order, sorted according to the elements' natural ordering or according to a `Comparator` provided at `SortedSet` creation time. In addition to the normal `Set` operations, the `SortedSet` interface provides operations for the following:

- `Range view` — allows arbitrary range operations on the sorted set
- `Endpoints` — returns the first or last element in the sorted set
- `Comparator access` — returns the `Comparator`, if any, used to sort the set

The code for the `SortedSet` interface follows.

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

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

Set Operations

The operations that `SortedSet` inherits from `Set` behave identically on sorted sets and normal sets with two exceptions:

- The `Iterator` returned by the `iterator` operation traverses the sorted set in order.
- The array returned by `toArray` contains the sorted set's elements in order.

Although the interface doesn't guarantee it, the `toString` method of the Java platform's `SortedSet` implementations returns a string containing all the elements of the sorted set, in order.

Standard Constructors

By convention, all general-purpose `Collection` implementations provide a standard conversion constructor that takes a `Collection`; `SortedSet` implementations are no exception. In `TreeSet`, this constructor creates an instance that sorts its elements according to their natural ordering. This was probably a mistake. It would have been better to check dynamically to see whether the specified collection was a `SortedSet` instance and, if so, to sort the new `TreeSet` according to the same criterion (comparator or natural ordering). Because `TreeSet` took the approach that it did, it also provides a constructor that takes a `SortedSet` and returns a new `TreeSet` containing the same elements sorted according to the same criterion. Note that it is the compile-time type of the argument, not its runtime type, that determines which of these two constructors is invoked (and whether the

sorting criterion is preserved).

`SortedSet` implementations also provide, by convention, a constructor that takes a `Comparator` and returns an empty set sorted according to the specified `Comparator`. If `null` is passed to this constructor, it returns a set that sorts its elements according to their natural ordering.

Range-view Operations

The `range-view` operations are somewhat analogous to those provided by the `List` interface, but there is one big difference. Range views of a sorted set remain valid even if the backing sorted set is modified directly. This is feasible because the endpoints of a range view of a sorted set are absolute points in the element space rather than specific elements in the backing collection, as is the case for lists. A `range-view` of a sorted set is really just a window onto whatever portion of the set lies in the designated part of the element space. Changes to the `range-view` write back to the backing sorted set and vice versa. Thus, it's okay to use `range-view`s on sorted sets for long periods of time, unlike `range-view`s on lists.

Sorted sets provide three `range-view` operations. The first, `subSet`, takes two endpoints, like `subList`. Rather than indices, the endpoints are objects and must be comparable to the elements in the sorted set, using the `Set`'s `Comparator` or the natural ordering of its elements, whichever the `Set` uses to order itself. Like `subList`, the range is half open, including its low endpoint but excluding the high one.

Thus, the following line of code tells you how many words between `"doorbell"` and `"pickle"`, including `"doorbell"` but excluding `"pickle"`, are contained in a `SortedSet` of strings called `dictionary`:

```
int count = dictionary.subSet("doorbell", "pickle").size();
```

In like manner, the following one-liner removes all the elements beginning with the letter `f`.

```
dictionary.subSet("f", "g").clear();
```

A similar trick can be used to print a table telling you how many words begin with each letter.

```
for (char ch = 'a'; ch <= 'z'; ) {
    String from = String.valueOf(ch++);
    String to = String.valueOf(ch);
    System.out.println(from + ": " + dictionary.subSet(from, to).size());
}
```

Suppose you want to view a *closed interval*, which contains both of its endpoints, instead of an open interval. If the element type allows for the calculation of the successor of a given value in the element space, merely request the `subSet` from `lowEndpoint` to `successor(highEndpoint)`. Although it isn't entirely obvious, the successor of a string `s` in `String`'s natural ordering is `s + "\0"` — that is,

`s` with a `null` character appended.

Thus, the following one-liner tells you how many words between `"doorbell"` and `"pickle"`, including doorbell *and* pickle, are contained in the dictionary.

```
count = dictionary.subSet("doorbell", "pickle\0").size();
```

A similar technique can be used to view an *open interval*, which contains neither endpoint. The open-interval view from `lowEndpoint` to `highEndpoint` is the half-open interval from `successor(lowEndpoint)` to `highEndpoint`. Use the following to calculate the number of words between `"doorbell"` and `"pickle"`, excluding both.

```
count = dictionary.subSet("doorbell\0", "pickle").size();
```

The `SortedSet` interface contains two more `range-view` operations — `headSet` and `tailSet`, both of which take a single `Object` argument. The former returns a view of the initial portion of the backing `SortedSet`, up to but not including the specified object. The latter returns a view of the final portion of the backing `SortedSet`, beginning with the specified object and continuing to the end of the backing `SortedSet`. Thus, the following code allows you to view the dictionary as two disjoint `volumes` (`a-m` and `n-z`).

```
SortedSet<String> volume1 = dictionary.headSet("n");
SortedSet<String> volume2 = dictionary.tailSet("n");
```

Endpoint Operations

The `SortedSet` interface contains operations to return the first and last elements in the sorted set, not surprisingly called `first` and `last`. In addition to their obvious uses, `last` allows a workaround for a deficiency in the `SortedSet` interface. One thing you'd like to do with a `SortedSet` is to go into the interior of the `Set` and iterate forward or backward. It's easy enough to go forward from the interior: Just get a `tailSet` and iterate over it. Unfortunately, there's no easy way to go backward.

The following idiom obtains the first element that is less than a specified object `o` in the element space.

```
Object predecessor = ss.headSet(o).last();
```

This is a fine way to go one element backward from a point in the interior of a sorted set. It could be applied repeatedly to iterate backward, but this is very inefficient, requiring a lookup for each element returned.

Comparator Accessor

The `SortedSet` interface contains an accessor method called `comparator` that returns the

`Comparator` used to sort the set, or `null` if the set is sorted according to the *natural ordering* of its elements. This method is provided so that sorted sets can be copied into new sorted sets with the same ordering. It is used by the `SortedSet` constructor described previously.

The SortedMap Interface

A `SortedMap` is a `Map` that maintains its entries in ascending order, sorted according to the keys' natural ordering, or according to a `Comparator` provided at the time of the `SortedMap` creation. Natural ordering and `Comparator`s are discussed in the Object Ordering section. The `SortedMap` interface provides operations for normal `Map` operations and for the following:

- `Range view` — performs arbitrary range operations on the sorted map
- `Endpoints` — returns the first or the last key in the sorted map
- `Comparator access` — returns the `Comparator`, if any, used to sort the map

The following interface is the `Map` analog of `SortedSet`.

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

Map Operations

The operations `SortedMap` inherits from `Map` behave identically on sorted maps and normal maps with two exceptions:

- The `Iterator` returned by the `iterator` operation on any of the sorted map's `Collection` views traverse the collections in order.
- The arrays returned by the `Collection` views' `toArray` operations contain the keys, values, or entries in order.

Although it isn't guaranteed by the interface, the `toString` method of the `Collection` views in all the Java platform's `SortedMap` implementations returns a string containing all the elements of the view, in order.

Standard Constructors

By convention, all general-purpose `Map` implementations provide a standard conversion constructor that takes a `Map`; `SortedMap` implementations are no exception. In `TreeMap`, this constructor creates an instance that orders its entries according to their keys' natural ordering. This was probably a mistake. It would have been better to check dynamically to see whether the specified `Map` instance was a `SortedMap` and, if so, to sort the new map according to the same criterion (comparator or natural ordering). Because `TreeMap` took the approach it did, it also provides a constructor that takes a `SortedMap` and returns a new `TreeMap` containing the same mappings as the given `SortedMap`, sorted according to the same criterion. Note that it is the compile-time type of the argument, not its runtime type, that determines whether the `SortedMap` constructor is invoked in preference to the ordinary `map` constructor.

`SortedMap` implementations also provide, by convention, a constructor that takes a `Comparator` and returns an empty map sorted according to the specified `Comparator`. If `null` is passed to this constructor, it returns a `Map` that sorts its mappings according to their keys' natural ordering.

Comparison to SortedSet

Because this interface is a precise `Map` analog of `SortedSet`, all the idioms and code examples in The SortedSet Interface section apply to `SortedMap` with only trivial modifications.

Summary of Interfaces

The core collection interfaces are the foundation of the Java Collections Framework.

The Java Collections Framework hierarchy consists of two distinct interface trees:

- The first tree starts with the `Collection` interface, which provides for the basic functionality used by all collections, such as `add` and `remove` methods. Its subinterfaces — `Set`, `List`, and `Queue` — provide for more specialized collections.
- The `Set` interface does not allow duplicate elements. This can be useful for storing collections such as a deck of cards or student records. The `Set` interface has a subinterface, `SortedSet`, that provides for ordering of elements in the set.
- The `List` interface provides for an ordered collection, for situations in which you need precise control over where each element is inserted. You can retrieve elements from a `List` by their exact position.
- The `Queue` interface enables additional insertion, extraction, and inspection operations. Elements in a `Queue` are typically ordered in on a FIFO basis.
- The `Deque` interface enables insertion, deletion, and inspection operations at both the ends. Elements in a `Deque` can be used in both LIFO and FIFO.
- The second tree starts with the `Map` interface, which maps keys and values similar to a `Hashtable`.
- `Map`'s subinterface, `SortedMap`, maintains its key-value pairs in ascending order or in an order specified by a `Comparator`.

These interfaces allow collections to be manipulated independently of the details of their representation.

Questions and Exercises: Interfaces

Questions

1. At the beginning of this lesson, you learned that the core collection interfaces are organized into two distinct inheritance trees. One interface in particular is not considered to be a true `Collection`, and therefore sits at the top of its own tree. What is the name of this interface?
2. Each interface in the collections framework is declared with the `<E>` syntax, which tells you that it is generic. When you declare a `Collection` instance, what is the advantage of specifying the type of objects that it will contain?
3. What interface represents a collection that does not allow duplicate elements?
4. What interface forms the root of the collections hierarchy?
5. What interface represents an ordered collection that may contain duplicate elements?
6. What interface represents a collection that holds elements prior to processing?
7. What interface repesents a type that maps keys to values?
8. What interface represents a double-ended queue?
9. Name three different ways to iterate over the elements of a `List`.
10. True or False: Aggregate operations are mutative operations that modify the underlying collection.

Exercises

1. Write a program that prints its arguments in random order. Do not make a copy of the argument array. Demonstrate how to print out the elements using both streams and the traditional enhanced for statement.
2. Take the `FindDups`example and modify it to use a `SortedSet` instead of a `Set`. Specify a `Comparator` so that case is ignored when sorting and identifying set elements.
3. Write a method that takes a `List<String>` and applies `String.trim` to each element.
4. Consider the four core interfaces, `Set`, `List`, `Queue`, and `Map`. For each of the following four assignments, specify which of the four core interfaces is best-suited, and explain how to use it to implement the assignment.
a. Whimsical Toys Inc (WTI) needs to record the names of all its employees. Every month, an employee will be chosen at random from these records to receive a free toy.
b. WTI has decided that each new product will be named after an employee but only first names will be used, and each name will be used only once. Prepare a list of unique first names.
c. WTI decides that it only wants to use the most popular names for its toys. Count up the number of employees who have each first name.
d. WTI acquires season tickets for the local lacrosse team, to be shared by employees. Create a waiting list for this popular sport.

Check your answers.

Questions

1. Question: At the beginning of this lesson, you learned that the core collection interfaces are organized into two distinct inheritance trees. One interface in particular is not considered to be a true `Collection`, and therefore sits at the top of its own tree. What is the name of this interface?
Answer: `Map`

2. Question: Each interface in the collections framework is declared with the `<E>` syntax, which tells you that it is generic. When you declare a `Collection` instance, what is the advantage of specifying the type of objects that it will contain?
Answer: Specifying the type allows the compiler to verify (at compile time) that the type of object you put into the collection is correct, thus reducing errors at runtime.

3. Question: What interface represents a collection that does not allow duplicate elements?
Answer: `Set`

4. Question: What interface forms the root of the collections hierarchy?
Answer: `Collection`

5. Question: What interface represents an ordered collection that may contain duplicate elements?
Answer: `List`

6. Question: What interface represents a collection that holds elements prior to processing?
Answer: `Queue`

7. Question: What interface repesents a type that maps keys to values?
Answer: `Map`

8. Question: What interface represents a double-ended queue?
Answer: `Deque`

9. Question: Name three different ways to iterate over the elements of a `List`.
Answer: You can iterate over a `List` using streams, the enhanced `for` statement, or iterators.

10. Question: True or False: Aggregate operations are mutative operations that modify the underlying collection.
Answer: False. Aggregate operations do not mutate the underlying collection. In fact, you must be careful to never mutate a collection while invoking its aggregate operations. Doing so could lead to concurrency problems should the stream be changed to a parallel stream at some point in the future.

Exercises

1. Exercise: Write a program that prints its arguments in random order. Do not make a copy of the argument array. Demonstrate how to print out the elements using both streams and the traditional enhanced for statement.

Answer:

```
import java.util.*;

public class Ran {

    public static void main(String[] args) {

        // Get and shuffle the list of arguments
        List<String> argList = Arrays.asList(args);
        Collections.shuffle(argList);

        // Print out the elements using JDK 8 Streams
        argList.stream()
        .forEach(e->System.out.format("%s ",e));

        // Print out the elements using for-each
        for (String arg: argList) {
            System.out.format("%s ", arg);
        }

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

2. Exercise: Take the `FindDups`example and modify it to use a `SortedSet` instead of a `Set`. Specify a `Comparator` so that case is ignored when sorting and identifying set elements.
Answer:

```
import java.util.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
               s.add(a);
               System.out.println(s.size() + " distinct words: " + s);
    }
}
```

3. Exercise: Write a method that takes a `List<String>` and applies `String.trim` to each element.
Answer:
The enhanced `for` statement does not allow you to modify the `List`. Using an instance of the `Iterator` class allows you to delete elements, but not replace an existing element or add a new one. That leaves `ListIterator`:

```
import java.util.*;

public class ListTrim {
    static void listTrim(List<String> strings) {
        for (ListIterator<String> lit = strings.listIterator(); lit.hasNext(); ) {
            lit.set(lit.next().trim());
        }
    }

    public static void main(String[] args) {
        List<String> l = Arrays.asList(" red ", " white ", " blue ");
        listTrim(l);
        for (String s : l) {
            System.out.format("\"%s\"%n", s);
        }
    }
}
```

4. Exercise: Consider the four core interfaces, `Set`, `List`, `Queue`, and `Map`. For each of the following four assignments, specify which of the four core interfaces is best-suited, and explain how to use it to implement the assignment.
Answers:

- Whimsical Toys Inc (WTI) needs to record the names of all its employees. Every month, an employee will be chosen at random from these records to receive a free toy.
  Use a `List`. Choose a random employee by picking a number between `0` and `size()-1`.
- WTI has decided that each new product will be named after an employee — but only first names will be used, and each name will be used only once. Prepare a list of unique first names.
  Use a `Set`. Collections that implement this interface don't allow the same element to be entered more than once.
- WTI decides that it only wants to use the most popular names for its toys. Count up the number of employees who have each first name.
  Use a `Map`, where the keys are first names, and each value is a count of the number of employees with that first name.
- WTI acquires season tickets for the local lacrosse team, to be shared by employees. Create a waiting list for this popular sport.
  Use a `Queue`. Invoke `add()` to add employees to the waiting list, and `remove()` to remove them.

Lesson: Aggregate Operations

Note: To better understand the concepts in this section, review the sections Lambda Expressions and Method References.

For what do you use collections? You don't simply store objects in a collection and leave them there. In most cases, you use collections to retrieve items stored in them.

Consider again the scenario described in the section Lambda Expressions. Suppose that you are creating a social networking application. You want to create a feature that enables an administrator to perform any kind of action, such as sending a message, on members of the social networking application that satisfy certain criteria.

As before, suppose that members of this social networking application are represented by the following `Person` class:

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    // ...

    public int getAge() {
        // ...
    }

    public String getName() {
        // ...
    }
}
```

The following example prints the name of all members contained in the collection `roster` with a for-each loop:

```
for (Person p : roster) {
    System.out.println(p.getName());
}
```

The following example prints all members contained in the collection `roster` but with the aggregate operation `forEach`:

```
roster
    .stream()
    .forEach(e -> System.out.println(e.getName());
```

Although, in this example, the version that uses aggregate operations is longer than the one that uses a for-each loop, you will see that versions that use bulk-data operations will be more concise for more complex tasks.

The following topics are covered:

- Pipelines and Streams
- Differences Between Aggregate Operations and Iterators

Find the code excerpts described in this section in the example `BulkDataOperationsExamples`.

Pipelines and Streams

A *pipeline* is a sequence of aggregate operations. The following example prints the male members contained in the collection `roster` with a pipeline that consists of the aggregate operations `filter` and `forEach`:

```
roster
    .stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .forEach(e -> System.out.println(e.getName());
```

Compare this example to the following that prints the male members contained in the collection `roster` with a for-each loop:

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());
    }
}
```

A pipeline contains the following components:

- A source: This could be a collection, an array, a generator function, or an I/O channel. In this example, the source is the collection `roster`.
- Zero or more *intermediate operations*. An intermediate operation, such as `filter`, produces a new stream.
  A *stream* is a sequence of elements. Unlike a collection, it is not a data structure that stores elements. Instead, a stream carries values from a source through a pipeline. This example creates a stream from the collection `roster` by invoking the method `stream`.
  The `filter` operation returns a new stream that contains elements that match its predicate (this operation's parameter). In this example, the predicate is the lambda expression `e -> e.getGender() == Person.Sex.MALE`. It returns the boolean value `true` if the `gender` field of object `e` has the value `Person.Sex.MALE`. Consequently, the `filter` operation in this example returns a stream that contains all male members in the collection `roster`.
- A *terminal operation*. A terminal operation, such as `forEach`, produces a non-stream result, such as a primitive value (like a double value), a collection, or in the case of `forEach`, no value at all. In this example, the parameter of the `forEach` operation is the lambda expression `e -> System.out.println(e.getName())`, which invokes the method `getName` on the object `e`. (The Java runtime and compiler infer that the type of the object `e` is `Person`.)

The following example calculates the average age of all male members contained in the collection `roster` with a pipeline that consists of the aggregate operations `filter`, `mapToInt`, and `average`:

```
double average = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

The `mapToInt` operation returns a new stream of type `IntStream` (which is a stream that contains only integer values). The operation applies the function specified in its parameter to each element in a particular stream. In this example, the function is `Person::getAge`, which is a method reference that returns the age of the member. (Alternatively, you could use the lambda expression `e -> e.getAge()`.) Consequently, the `mapToInt` operation in this example returns a stream that contains the ages of all male members in the collection `roster`.

The `average` operation calculates the average value of the elements contained in a stream of type `IntStream`. It returns an object of type `OptionalDouble`. If the stream contains no elements, then the `average` operation returns an empty instance of `OptionalDouble`, and invoking the method `getAsDouble` throws a `NoSuchElementException`. The JDK contains many terminal operations such as `average` that return one value by combining the contents of a stream. These operations are called *reduction operations*; see the section Reduction for more information.

Differences Between Aggregate Operations and Iterators

Aggregate operations, like `forEach`, appear to be like iterators. However, they have several fundamental differences:

- **They use internal iteration**: Aggregate operations do not contain a method like `next` to instruct them to process the next element of the collection. With *internal delegation*, your application determines *what* collection it iterates, but the JDK determines *how* to iterate the collection. With *external iteration*, your application determines both what collection it iterates and how it iterates it. However, external iteration can only iterate over the elements of a collection sequentially. Internal iteration does not have this limitation. It can more easily take advantage of parallel computing, which involves dividing a problem into subproblems, solving those problems simultaneously, and then combining the results of the solutions to the subproblems. See the section Parallelism for more information.
- **They process elements from a stream**: Aggregate operations process elements from a stream, not directly from a collection. Consequently, they are also called *stream operations*.
- **They support behavior as parameters**: You can specify lambda expressions as parameters for most aggregate operations. This enables you to customize the behavior of a particular aggregate operation.

Note: See online version of topics in this ebook to download complete source code.

Reduction

The section Aggregate Operations describes the following pipeline of operations, which calculates the average age of all male members in the collection `roster`:

```
double average = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

The JDK contains many terminal operations (such as `average`, `sum`, `min`, `max`, and `count`) that return one value by combining the contents of a stream. These operations are called *reduction operations*. The JDK also contains reduction operations that return a collection instead of a single value. Many reduction operations perform a specific task, such as finding the average of values or grouping elements into categories. However, the JDK provides you with the general-purpose reduction operations `reduce` and `collect`, which this section describes in detail.

This section covers the following topics:

- The Stream.reduce Method
- The Stream.collect Method

You can find the code excerpts described in this section in the example `ReductionExamples`.

The Stream.reduce Method

The `Stream.reduce` method is a general-purpose reduction operation. Consider the following pipeline, which calculates the sum of the male members' ages in the collection `roster`. It uses the `Stream.sum` reduction operation:

```
Integer totalAge = roster
    .stream()
    .mapToInt(Person::getAge)
    .sum();
```

Compare this with the following pipeline, which uses the `Stream.reduce` operation to calculate the same value:

```
Integer totalAgeReduce = roster
   .stream()
   .map(Person::getAge)
   .reduce(
       0,
       (a, b) -> a + b);
```

The `reduce` operation in this example takes two arguments:

- `identity`: The identity element is both the initial value of the reduction and the default result if there are no elements in the stream. In this example, the identity element is `0`; this is the initial value of the sum of ages and the default value if no members exist in the collection `roster`.

- `accumulator`: The accumulator function takes two parameters: a partial result of the reduction (in this example, the sum of all processed integers so far) and the next element of the stream (in this example, an integer). It returns a new partial result. In this example, the accumulator function is a lambda expression that adds two `Integer` values and returns an `Integer` value:

```
(a, b) -> a + b
```

The `reduce` operation always returns a new value. However, the accumulator function also returns a new value every time it processes an element of a stream. Suppose that you want to reduce the elements of a stream to a more complex object, such as a collection. This might hinder the performance of your application. If your `reduce` operation involves adding elements to a collection, then every time your accumulator function processes an element, it creates a new collection that includes the element, which is inefficient. It would be more efficient for you to update an existing collection instead. You can do this with the `Stream.collect` method, which the next section describes.

The Stream.collect Method

Unlike the `reduce` method, which always creates a new value when it processes an element, the `collect` method modifies, or mutates, an existing value.

Consider how to find the average of values in a stream. You require two pieces of data: the total number of values and the sum of those values. However, like the `reduce` method and all other reduction methods, the `collect` method returns only one value. You can create a new data type that contains member variables that keep track of the total number of values and the sum of those values, such as the following class, `Averager`:

```
class Averager implements IntConsumer
{
    private int total = 0;
    private int count = 0;

    public double average() {
        return count > 0 ? ((double) total)/count : 0;
    }

    public void accept(int i) { total += i; count++; }
    public void combine(Averager other) {
        total += other.total;
        count += other.count;
    }
}
```

The following pipeline uses the `Averager` class and the `collect` method to calculate the average age of all male members:

```
Averager averageCollect = roster.stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(Person::getAge)
    .collect(Averager::new, Averager::accept, Averager::combine);

System.out.println("Average age of male members: " +
    averageCollect.average());
```

The `collect` operation in this example takes three arguments:

- `supplier`: The supplier is a factory function; it constructs new instances. For the `collect` operation, it creates instances of the result container. In this example, it is a new instance of the `Averager` class.
- `accumulator`: The accumulator function incorporates a stream element into a result container. In this example, it modifies the `Averager` result container by incrementing the `count` variable by one and adding to the `total` member variable the value of the stream element, which is an integer representing the age of a male member.
- `combiner`: The combiner function takes two result containers and merges their contents. In this example, it modifies an `Averager` result container by incrementing the `count` variable by the `count` member variable of the other `Averager` instance and adding to the `total` member variable the value of the other `Averager` instance's `total` member variable.

Note the following:

- The supplier is a lambda expression (or a method reference) as opposed to a value like the identity element in the `reduce` operation.
- The accumulator and combiner functions do not return a value.
- You can use the `collect` operations with parallel streams; see the section Parallelism for more information. (If you run the `collect` method with a parallel stream, then the JDK creates a new thread whenever the combiner function creates a new object, such as an `Averager` object in this example. Consequently, you do not have to worry about synchronization.)

Although the JDK provides you with the `average` operation to calculate the average value of elements in a stream, you can use the `collect` operation and a custom class if you need to calculate several values from the elements of a stream.

The `collect` operation is best suited for collections. The following example puts the names of the male members in a collection with the `collect` operation:

```
List<String> namesOfMaleMembersCollect = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(p -> p.getName())
    .collect(Collectors.toList());
```

This version of the `collect` operation takes one parameter of type `Collector`. This class encapsulates the functions used as arguments in the `collect` operation that requires three arguments (supplier, accumulator, and combiner functions).

The `Collectors` class contains many useful reduction operations, such as accumulating elements into collections and summarizing elements according to various criteria. These reduction operations return instances of the class `Collector`, so you can use them as a parameter for the `collect` operation.

This example uses the `Collectors.toList` operation, which accumulates the stream elements into a new instance of `List`. As with most operations in the `Collectors` class, the `toList` operator returns an instance of `Collector`, not a collection.

The following example groups members of the collection `roster` by gender:

```
Map<Person.Sex, List<Person>> byGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(Person::getGender));
```

The `groupingBy` operation returns a map whose keys are the values that result from applying the lambda expression specified as its parameter (which is called a *classification function*). In this example, the returned map contains two keys, `Person.Sex.MALE` and `Person.Sex.FEMALE`. The keys' corresponding values are instances of `List` that contain the stream elements that, when processed by the classification function, correspond to the key value. For example, the value that corresponds to key `Person.Sex.MALE` is an instance of `List` that contains all male members.

The following example retrieves the names of each member in the collection `roster` and groups them by gender:

```
Map<Person.Sex, List<String>> namesByGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(
                Person::getGender,
                Collectors.mapping(
                    Person::getName,
                    Collectors.toList())));
```

The `groupingBy` operation in this example takes two parameters, a classification function and an instance of `Collector`. The `Collector` parameter is called a *downstream collector*. This is a collector that the Java runtime applies to the results of another collector. Consequently, this `groupingBy` operation enables you to apply a `collect` method to the `List` values created by the `groupingBy` operator. This example applies the collector `mapping`, which applies the mapping function `Person::getName` to each element of the stream. Consequently, the resulting stream consists of only the names of members. A pipeline that contains one or more downstream collectors, like this example, is called a *multilevel reduction*.

The following example retrieves the total age of members of each gender:

```
Map<Person.Sex, Integer> totalAgeByGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(
                Person::getGender,
                Collectors.reducing(
                    0,
                    Person::getAge,
                    Integer::sum)));
```

The `reducing` operation takes three parameters:

- `identity`: Like the `Stream.reduce` operation, the identity element is both the initial value of the reduction and the default result if there are no elements in the stream. In this example, the identity element is `0`; this is the initial value of the sum of ages and the default value if no members exist.

- `mapper`: The `reducing` operation applies this mapper function to all stream elements. In this example, the mapper retrieves the age of each member.
- `operation`: The operation function is used to reduce the mapped values. In this example, the operation function adds `Integer` values.

The following example retrieves the average age of members of each gender:

```
Map<Person.Sex, Double> averageAgeByGender = roster
    .stream()
    .collect(
        Collectors.groupingBy(
            Person::getGender,
            Collectors.averagingInt(Person::getAge)));
```

Parallelism

Parallel computing involves dividing a problem into subproblems, solving those problems simultaneously (in parallel, with each subproblem running in a separate thread), and then combining the results of the solutions to the subproblems. Java SE provides the fork/join framework, which enables you to more easily implement parallel computing in your applications. However, with this framework, you must specify how the problems are subdivided (partitioned). With aggregate operations, the Java runtime performs this partitioning and combining of solutions for you.

One difficulty in implementing parallelism in applications that use collections is that collections are not thread-safe, which means that multiple threads cannot manipulate a collection without introducing thread interference or memory consistency errors. The Collections Framework provides synchronization wrappers, which add automatic synchronization to an arbitrary collection, making it thread-safe. However, synchronization introduces thread contention. You want to avoid thread contention because it prevents threads from running in parallel. Aggregate operations and parallel streams enable you to implement parallelism with non-thread-safe collections provided that you do not modify the collection while you are operating on it.

Note that parallelism is not automatically faster than performing operations serially, although it can be if you have enough data and processor cores. While aggregate operations enable you to more easily implement parallelism, it is still your responsibility to determine if your application is suitable for parallelism.

This section covers the following topics:

You can find the code excerpts described in this section in the example `ParallelismExamples`.

Executing Streams in Parallel

You can execute streams in serial or in parallel. When a stream executes in parallel, the Java runtime partitions the stream into multiple substreams. Aggregate operations iterate over and process these substreams in parallel and then combine the results.

When you create a stream, it is always a serial stream unless otherwise specified. To create a parallel stream, invoke the operation `Collection.parallelStream`. Alternatively, invoke the operation `BaseStream.parallel`. For example, the following statement calculates the average age of all male members in parallel:

```
double average = roster
    .parallelStream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

Concurrent Reduction

Consider again the following example (which is described in the section Reduction) that groups members by gender. This example invokes the `collect` operation, which reduces the collection `roster` into a `Map`:

```
Map<Person.Sex, List<Person>> byGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(Person::getGender));
```

The following is the parallel equivalent:

```
ConcurrentMap<Person.Sex, List<Person>> byGender =
    roster
        .parallelStream()
        .collect(
            Collectors.groupingByConcurrent(Person::getGender));
```

This is called a *concurrent reduction*. The Java runtime performs a concurrent reduction if all of the the following are true for a particular pipeline that contains the `collect` operation:

- The stream is parallel.
- The parameter of the `collect` operation, the collector, has the characteristic `Collector.Characteristics.CONCURRENT`. To determine the characteristics of a collector, invoke the `Collector.characteristics` method.
- Either the stream is unordered, or the collector has the characteristic `Collector.Characteristics.UNORDERED`. To ensure that the stream is unordered, invoke the `BaseStream.unordered` operation.

Note: This example returns an instance of `ConcurrentMap` instead of `Map` and invokes the `groupingByConcurrent` operation instead of `groupingBy`. (See the section Concurrent Collections for more information about `ConcurrentMap`.) Unlike the operation `groupingByConcurrent`, the operation `groupingBy` performs poorly with parallel streams. (This is because it operates by merging two maps by key, which is computationally expensive.) Similarly, the operation `Collectors.toConcurrentMap` performs better with parallel streams than the operation `Collectors.toMap`.

Ordering

The order in which a pipeline processes the elements of a stream depends on whether the stream is executed in serial or in parallel, the source of the stream, and intermediate operations. For example, consider the following example that prints the elements of an instance of `ArrayList` with the `forEach` operation several times:

```
Integer[] intArray = {1, 2, 3, 4, 5, 6, 7, 8 };
List<Integer> listOfIntegers =
    new ArrayList<>(Arrays.asList(intArray));

System.out.println("listOfIntegers:");
listOfIntegers
    .stream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("listOfIntegers sorted in reverse order:");
Comparator<Integer> normal = Integer::compare;
Comparator<Integer> reversed = normal.reversed();
Collections.sort(listOfIntegers, reversed);
listOfIntegers
    .stream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("Parallel stream");
listOfIntegers
    .parallelStream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("Another parallel stream:");
listOfIntegers
    .parallelStream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("With forEachOrdered:");
listOfIntegers
    .parallelStream()
    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");
```

This example consists of five pipelines. It prints output similar to the following:

```
listOfIntegers:
1 2 3 4 5 6 7 8
listOfIntegers sorted in reverse order:
8 7 6 5 4 3 2 1
Parallel stream:
3 4 1 6 2 5 7 8
Another parallel stream:
6 3 1 5 7 8 4 2
With forEachOrdered:
8 7 6 5 4 3 2 1
```

This example does the following:

- The first pipeline prints the elements of the list `listOfIntegers` in the order that they were added to the list.
- The second pipeline prints the elements of `listOfIntegers` after it was sorted by the method `Collections.sort`.
- The third and fourth pipelines print the elements of the list in an apparently random order. Remember that stream operations use internal iteration when processing elements of a stream. Consequently, when you execute a stream in parallel, the Java compiler and runtime determine the order in which to process the stream's elements to maximize the benefits of parallel computing unless otherwise specified by the stream operation.

- The fifth pipeline uses the method `forEachOrdered`, which processes the elements of the stream in the order specified by its source, regardless of whether you executed the stream in serial or parallel. Note that you may lose the benefits of parallelism if you use operations like `forEachOrdered` with parallel streams.

Side Effects

A method or an expression has a side effect if, in addition to returning or producing a value, it also modifies the state of the computer. Examples include mutable reductions (operations that use the `collect` operation; see the section Reduction for more information) as well as invoking the `System.out.println` method for debugging. The JDK handles certain side effects in pipelines well. In particular, the `collect` method is designed to perform the most common stream operations that have side effects in a parallel-safe manner. Operations like `forEach` and `peek` are designed for side effects; a lambda expression that returns void, such as one that invokes `System.out.println`, can do nothing but have side effects. Even so, you should use the `forEach` and `peek` operations with care; if you use one of these operations with a parallel stream, then the Java runtime may invoke the lambda expression that you specified as its parameter concurrently from multiple threads. In addition, never pass as parameters lambda expressions that have side effects in operations such as `filter` and `map`. The following sections discuss interference and stateful lambda expressions, both of which can be sources of side effects and can return inconsistent or unpredictable results, especially in parallel streams. However, the concept of laziness is discussed first, because it has a direct effect on interference.

Laziness

All intermediate operations are *lazy*. An expression, method, or algorithm is lazy if its value is evaluated only when it is required. (An algorithm is *eager* if it is evaluated or processed immediately.) Intermediate operations are lazy because they do not start processing the contents of the stream until the terminal operation commences. Processing streams lazily enables the Java compiler and runtime to optimize how they process streams. For example, in a pipeline such as the `filter-mapToInt-average` example described in the section Aggregate Operations, the `average` operation could obtain the first several integers from the stream created by the `mapToInt` operation, which obtains elements from the `filter` operation. The `average` operation would repeat this process until it had obtained all required elements from the stream, and then it would calculate the average.

Interference

Lambda expressions in stream operations should not *interfere*. Interference occurs when the source of a stream is modified while a pipeline processes the stream. For example, the following code attempts to concatenate the strings contained in the `List listOfStrings`. However, it throws a `ConcurrentModifiedException`:

```
try {
    List<String> listOfStrings =
        new ArrayList<>(Arrays.asList("one", "two"));

    // This will fail as the peek operation will attempt to add the
    // string "three" to the source after the terminal operation has
    // commenced.
```

```
    String concatenatedString = listOfStrings
        .stream()

        // Don't do this! Interference occurs here.
        .peek(s -> listOfStrings.add("three"))

        .reduce((a, b) -> a + " " + b)
        .get();

    System.out.println("Concatenated string: " + concatenatedString);

} catch (Exception e) {
    System.out.println("Exception caught: " + e.toString());
}
```

This example concatenates the strings contained in `listOfStrings` into an `Optional<String>` value with the `reduce` operation, which is a terminal operation. However, the pipeline here invokes the intermediate operation `peek`, which attempts to add a new element to `listOfStrings`. Remember, all intermediate operations are lazy. This means that the pipeline in this example begins execution when the operation `get` is invoked, and ends execution when the `get` operation completes. The argument of the `peek` operation attempts to modify the stream source during the execution of the pipeline, which causes the Java runtime to throw a `ConcurrentModifiedException`.

Stateful Lambda Expressions

Avoid using *stateful lambda expressions* as parameters in stream operations. A stateful lambda expression is one whose result depends on any state that might change during the execution of a pipeline. The following example adds elements from the `List listOfIntegers` to a new `List` instance with the `map` intermediate operation. It does this twice, first with a serial stream and then with a parallel stream:

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

System.out.println("Serial stream:");
listOfIntegers
    .stream()

    // Don't do this! It uses a stateful lambda expression.
    .map(e -> { serialStorage.add(e); return e; })

    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");

serialStorage
    .stream()
    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("Parallel stream:");
List<Integer> parallelStorage = Collections.synchronizedList(
    new ArrayList<>());
listOfIntegers
    .parallelStream()

    // Don't do this! It uses a stateful lambda expression.
    .map(e -> { parallelStorage.add(e); return e; })

    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");
```

```
parallelStorage
    .stream()
    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");
```

The lambda expression `e -> { parallelStorage.add(e); return e; }` is a stateful lambda expression. Its result can vary every time the code is run. This example prints the following:

```
Serial stream:
8 7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
Parallel stream:
8 7 6 5 4 3 2 1
1 3 6 2 4 5 8 7
```

The operation `forEachOrdered` processes elements in the order specified by the stream, regardless of whether the stream is executed in serial or parallel. However, when a stream is executed in parallel, the `map` operation processes elements of the stream specified by the Java runtime and compiler. Consequently, the order in which the lambda expression `e -> { parallelStorage.add(e); return e; }` adds elements to the `List parallelStorage` can vary every time the code is run. For deterministic and predictable results, ensure that lambda expression parameters in stream operations are not stateful.

Note: This example invokes the method `synchronizedList` so that the `List parallelStorage` is thread-safe. Remember that collections are not thread-safe. This means that multiple threads should not access a particular collection at the same time. Suppose that you do not invoke the method `synchronizedList` when creating `parallelStorage`:

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

The example behaves erratically because multiple threads access and modify `parallelStorage` without a mechanism like synchronization to schedule when a particular thread may access the `List` instance. Consequently, the example could print output similar to the following:

```
Parallel stream:
8 7 6 5 4 3 2 1
null 3 5 4 7 8 1 2
```

Questions and Exercises: Aggregate Operations

Questions

1. A sequence of aggregate operations is known as a ___ .
2. Each pipeline contains zero or more ___ operations.
3. Each pipeline ends with a ___ operation.
4. What kind of operation produces another stream as its output?
5. Describe one way in which the `forEach` aggregate operation differs from the enhanced `for` statement or iterators.
6. True or False: A stream is similar to a collection in that it is a data structure that stores elements.
7. Identify the intermediate and terminal operations in this code:

```
double average = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

8. The code `p -> p.getGender() == Person.Sex.MALE` is an example of what?
9. The code `Person::getAge` is an example of what?
10. Terminal operations that combine the contents of a stream and return one value are known as what?
11. Name one important difference between the `Stream.reduce` method and the `Stream.collect` method.
12. If you wanted to process a stream of names, extract the male names, and store them in a new `List`, would `Stream.reduce` or `Stream.collect` be the most appropriate operation to use?
13. True or False: Aggregate operations make it possible to implement parallelism with non-thread-safe collections.
14. Streams are always serial unless otherwise specified. How do you request that a stream be processed in parallel?

Exercises

1. Write the following enhanced `for` statement as a pipeline with lambda expressions. Hint: Use the `filter` intermediate operation and the `forEach` terminal operation.

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());
    }
}
```

2. Convert the following code into a new implementation that uses lambda expressions and aggregate operations instead of nested `for` loops. Hint: Make a pipeline that invokes the `filter`, `sorted`, and `collect` operations, in that order.

```
List<Album> favs = new ArrayList<>();
for (Album a : albums) {
    boolean hasFavorite = false;
    for (Track t : a.tracks) {
        if (t.rating >= 4) {
```

```
            hasFavorite = true;
            break;
        }
    }
    if (hasFavorite)
        favs.add(a);
}
Collections.sort(favs, new Comparator<Album>() {
                           public int compare(Album a1, Album a2) {
                               return a1.name.compareTo(a2.name);
                           }});
```

Check your answers.

Questions

1. Q: A sequence of aggregate operations is known as a ___ .
A: Pipeline

2. Q: Each pipeline contains zero or more ___ operations.
A: Intermediate

3. Q: Each pipeline ends with a ___ operation.
A: Terminal

4. Q: What kind of operation produces another stream as its output?
A: Intermediate

5. Q: Describe one way in which the `forEach` aggregate operation differs from the enhanced `for` statement or iterators.
A: The `forEach` aggregate operation lets the system decide "how" the iteration takes place. Using aggregate operations lets you focus on "what" instead of "how."

6. Q: True or False: A stream is similar to a collection in that it is a data structure that stores elements.
A: False. Unlike a collection, a stream is not a data structure. It instead carries values from a source through a pipeline.

7. Q: Identify the intermediate and terminal operations in this code:

```
double average = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

A: Intermediate: `filter`, `mapToInt`
Terminal: `average`
The terminal operation `average` returns an `OptionalDouble`. The `getAsDouble` method is then invoked on that returned object. It is always a good idea to consult the API Specification for information about whether an operation is intermediate or terminal.

8. Q: The code `p -> p.getGender() == Person.Sex.MALE` is an example of what?
A: A lambda expression.

9. Q: The code `Person::getAge` is an example of what?
A: A method reference.

10. Q: Terminal operations that combine the contents of a stream and return one value are known as what?

A: Reduction operations.

11. Q: Name one important difference between the `Stream.reduce` method and the `Stream.collect` method.
A: `Stream.reduce` always creates a new value when it processes an element. `Stream.collect` modifies (or mutates) the existing value.

12. Q: If you wanted to process a stream of names, extract the male names, and store them in a new `List`, would `Stream.reduce` or `Stream.collect` be the most appropriate operation to use?
A: The collect operation is most appropriate for collecting into a `List`.

Example:

```
List<String> namesOfMaleMembersCollect = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(p -> p.getName())
    .collect(Collectors.toList());
```

13. Q: True or False: Aggregate operations make it possible to implement parallelism with non-thread-safe collections.
A: True, provided that you do not modify (mutate) the underlying collection while you are operating on it.

14. Q: Streams are always serial unless otherwise specified. How do you request that a stream be processed in parallel?
A: Obtain the parallel stream by invoking `parallelStream()` instead of `stream()`.

Exercises

1. Exercise: Write the following enhanced `for` statement as a pipeline with lambda expressions. Hint: Use the `filter` intermediate operation and the `forEach` terminal operation.

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());
    }
}
```

Answer:

```
roster
    .stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .forEach(e -> System.out.println(e.getName());
```

2. Convert the following code into a new implementation that uses lambda expressions and aggregate operations instead of nested `for` loops. Hint: Make a pipeline that invokes the `filter`, `sorted`, and `collect` operations, in that order.

```
List<Album> favs = new ArrayList<>();
for (Album a : albums) {
    boolean hasFavorite = false;
    for (Track t : a.tracks) {
        if (t.rating >= 4) {
            hasFavorite = true;
            break;
        }
    }
    if (hasFavorite)
        favs.add(a);
}
Collections.sort(favs, new Comparator<Album>() {
                           public int compare(Album a1, Album a2) {
                               return a1.name.compareTo(a2.name);
                           }});
```

Answer:

```
List<Album> sortedFavs =
  albums.stream()
        .filter(a -> a.tracks.anyMatch(t -> (t.rating >= 4)))
        .sorted(Comparator.comparing(a -> a.name))
        .collect(Collectors.toList());
```

Here we have used the stream operations to simplify each of the three major steps -- identification of whether any track in an album has a rating of at least 4 (`anyMatch`), the sorting, and the collection of albums matching our criteria into a `List`. The `Comparator.comparing()` method takes a function that extracts a `Comparable` sort key, and returns a `Comparator` that compares on that key.

Lesson: Implementations

Implementations are the data objects used to store collections, which implement the interfaces described in the Interfaces section. This lesson describes the following kinds of implementations:

- **General-purpose implementations** are the most commonly used implementations, designed for everyday use. They are summarized in the table titled General-purpose-implementations.
- **Special-purpose implementations** are designed for use in special situations and display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent implementations** are designed to support high concurrency, typically at the expense of single-threaded performance. These implementations are part of the `java.util.concurrent` package.
- **Wrapper implementations** are used in combination with other types of implementations, often the general-purpose ones, to provide added or restricted functionality.
- **Convenience implementations** are mini-implementations, typically made available via static factory methods, that provide convenient, efficient alternatives to general-purpose implementations for special collections (for example, singleton sets).
- **Abstract implementations** are skeletal implementations that facilitate the construction of custom implementations — described later in the Custom Collection Implementations section. An advanced topic, it's not particularly difficult, but relatively few people will need to do it.

The general-purpose implementations are summarized in the following table.

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementation
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Queue`					
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

As you can see from the table, the Java Collections Framework provides several general-purpose implementations of the `Set`, `List` , and `Map` interfaces. In each case, one implementation — `HashSet`, `ArrayList`, and `HashMap` — is clearly the one to use for most applications, all other things being equal. Note that the `SortedSet` and the `SortedMap` interfaces do not have rows in the table. Each of those interfaces has one implementation (`TreeSet` and `TreeMap`) and is listed in the `Set` and the `Map` rows. There are two general-purpose `Queue` implementations — `LinkedList`, which is also a `List` implementation, and `PriorityQueue`, which is omitted from the table. These two implementations provide very different semantics: `LinkedList` provides FIFO semantics, while `PriorityQueue` orders its elements according to their values.

Each of the general-purpose implementations provides all optional operations contained in its interface. All permit `null` elements, keys, and values. None are synchronized (thread-safe). All have *fail-fast iterators*, which detect illegal concurrent modification during iteration and fail quickly and cleanly rather than risking arbitrary, nondeterministic behavior at an undetermined time in the future. All are `Serializable` and all support a public `clone` method.

The fact that these implementations are unsynchronized represents a break with the past: The legacy collections `Vector` and `Hashtable` are synchronized. The present approach was taken because collections are frequently used when the synchronization is of no benefit. Such uses include single-threaded use, read-only use, and use as part of a larger data object that does its own synchronization. In general, it is good API design practice not to make users pay for a feature they don't use. Furthermore, unnecessary synchronization can result in deadlock under certain circumstances.

If you need thread-safe collections, the synchronization wrappers, described in the Wrapper Implementations section, allow *any* collection to be transformed into a synchronized collection. Thus, synchronization is optional for general-purpose implementations, whereas it is mandatory for legacy implementations. Moreover, the `java.util.concurrent` package provides concurrent implementations of the `BlockingQueue` interface, which extends `Queue`, and of the `ConcurrentMap` interface, which extends `Map`. These implementations offer much higher concurrency than mere synchronized implementations.

As a rule, you should be thinking about the interfaces, *not* the implementations. That is why there are no programming examples in this section. For the most part, the choice of implementation affects only performance. The preferred style, as mentioned in the Interfaces section, is to choose an implementation when a `Collection` is created and to immediately assign the new collection to a variable of the corresponding interface type (or to pass the collection to a method expecting an argument of the interface type). In this way, the program does not become dependent on any added methods in a given implementation, leaving the programmer free to change implementations anytime that it is warranted by performance concerns or behavioral details.

The sections that follow briefly discuss the implementations. The performance of the implementations is described using words such as *constant-time*, *log*, *linear*, *n log(n)*, and *quadratic* to refer to the asymptotic upper-bound on the time complexity of performing the operation. All this is quite a mouthful, and it doesn't matter much if you don't know what it means. If you're interested in knowing more, refer to any good algorithms textbook. One thing to keep in mind is that this sort of performance metric has its limitations. Sometimes, the nominally slower implementation may be faster. When in doubt, measure the performance!

Note: See online version of topics in this ebook to download complete source code.

Set Implementations

The `Set` implementations are grouped into general-purpose and special-purpose implementations.

General-Purpose Set Implementations

There are three general-purpose `Set` implementations — `HashSet`, `TreeSet`, and `LinkedHashSet`. Which of these three to use is generally straightforward. `HashSet` is much faster than `TreeSet` (constant-time versus log-time for most operations) but offers no ordering guarantees. If you need to use the operations in the `SortedSet` interface, or if value-ordered iteration is required, use `TreeSet`; otherwise, use `HashSet`. It's a fair bet that you'll end up using `HashSet` most of the time.

`LinkedHashSet` is in some sense intermediate between `HashSet` and `TreeSet`. Implemented as a hash table with a linked list running through it, it provides *insertion-ordered* iteration (least recently inserted to most recently) and runs nearly as fast as `HashSet`. The `LinkedHashSet` implementation spares its clients from the unspecified, generally chaotic ordering provided by `HashSet` without incurring the increased cost associated with `TreeSet`.

One thing worth keeping in mind about `HashSet` is that iteration is linear in the sum of the number of entries and the number of buckets (the *capacity*). Thus, choosing an initial capacity that's too high can waste both space and time. On the other hand, choosing an initial capacity that's too low wastes time by copying the data structure each time it's forced to increase its capacity. If you don't specify an initial capacity, the default is 16. In the past, there was some advantage to choosing a prime number as the initial capacity. This is no longer true. Internally, the capacity is always rounded up to a power of two. The initial capacity is specified by using the `int` constructor. The following line of code allocates a `HashSet` whose initial capacity is 64.

```
Set<String> s = new HashSet<String>(64);
```

The `HashSet` class has one other tuning parameter called the *load factor*. If you care a lot about the space consumption of your `HashSet`, read the `HashSet` documentation for more information. Otherwise, just accept the default; it's almost always the right thing to do.

If you accept the default load factor but want to specify an initial capacity, pick a number that's about twice the size to which you expect the set to grow. If your guess is way off, you may waste a bit of space, time, or both, but it's unlikely to be a big problem.

`LinkedHashSet` has the same tuning parameters as `HashSet`, but iteration time is not affected by capacity. `TreeSet` has no tuning parameters.

Special-Purpose Set Implementations

There are two special-purpose `Set` implementations — `EnumSet` and `CopyOnWriteArraySet`.

`EnumSet` is a high-performance `Set` implementation for enum types. All of the members of an enum

set must be of the same enum type. Internally, it is represented by a bit-vector, typically a single `long`. Enum sets support iteration over ranges of enum types. For example, given the enum declaration for the days of the week, you can iterate over the weekdays. The `EnumSet` class provides a static factory that makes it easy.

```
for (Day d : EnumSet.range(Day.MONDAY, Day.FRIDAY))
    System.out.println(d);
```

Enum sets also provide a rich, typesafe replacement for traditional bit flags.

```
EnumSet.of(Style.BOLD, Style.ITALIC)
```

`CopyOnWriteArraySet` is a `Set` implementation backed up by a copy-on-write array. All mutative operations, such as `add`, `set`, and `remove`, are implemented by making a new copy of the array; no locking is ever required. Even iteration may safely proceed concurrently with element insertion and deletion. Unlike most `Set` implementations, the `add`, `remove`, and `contains` methods require time proportional to the size of the set. This implementation is *only* appropriate for sets that are rarely modified but frequently iterated. It is well suited to maintaining event-handler lists that must prevent duplicates.

List Implementations

`List` implementations are grouped into general-purpose and special-purpose implementations.

General-Purpose List Implementations

There are two general-purpose `List` implementations — `ArrayList` and `LinkedList`. Most of the time, you'll probably use `ArrayList`, which offers constant-time positional access and is just plain fast. It does not have to allocate a node object for each element in the `List`, and it can take advantage of `System.arraycopy` when it has to move multiple elements at the same time. Think of `ArrayList` as `Vector` without the synchronization overhead.

If you frequently add elements to the beginning of the `List` or iterate over the `List` to delete elements from its interior, you should consider using `LinkedList`. These operations require constant-time in a `LinkedList` and linear-time in an `ArrayList`. But you pay a big price in performance. Positional access requires linear-time in a `LinkedList` and constant-time in an `ArrayList`. Furthermore, the constant factor for `LinkedList` is much worse. If you think you want to use a `LinkedList`, measure the performance of your application with both `LinkedList` and `ArrayList` before making your choice; `ArrayList` is usually faster.

`ArrayList` has one tuning parameter — the *initial capacity*, which refers to the number of elements the `ArrayList` can hold before it has to grow. `LinkedList` has no tuning parameters and seven optional operations, one of which is `clone`. The other six are `addFirst`, `getFirst`, `removeFirst`, `addLast`, `getLast`, and `removeLast`. `LinkedList` also implements the `Queue` interface.

Special-Purpose List Implementations

`CopyOnWriteArrayList` is a `List` implementation backed up by a copy-on-write array. This implementation is similar in nature to `CopyOnWriteArraySet`. No synchronization is necessary, even during iteration, and iterators are guaranteed never to throw `ConcurrentModificationException`. This implementation is well suited to maintaining event-handler lists, in which change is infrequent, and traversal is frequent and potentially time-consuming.

If you need synchronization, a `Vector` will be slightly faster than an `ArrayList` synchronized with `Collections.synchronizedList`. But `Vector` has loads of legacy operations, so be careful to always manipulate the `Vector` with the `List` interface or else you won't be able to replace the implementation at a later time.

If your `List` is fixed in size — that is, you'll never use `remove`, `add`, or any of the bulk operations other than `containsAll` — you have a third option that's definitely worth considering. See `Arrays.asList` in the Convenience Implementations section for more information.

Map Implementations

`Map` implementations are grouped into general-purpose, special-purpose, and concurrent implementations.

General-Purpose Map Implementations

The three general-purpose `Map` implementations are `HashMap`, `TreeMap` and `LinkedHashMap`. If you need `SortedMap` operations or key-ordered `Collection`-view iteration, use `TreeMap`; if you want maximum speed and don't care about iteration order, use `HashMap`; if you want near-`HashMap` performance and insertion-order iteration, use `LinkedHashMap`. In this respect, the situation for `Map` is analogous to `Set`. Likewise, everything else in the Set Implementations section also applies to `Map` implementations.

`LinkedHashMap` provides two capabilities that are not available with `LinkedHashSet`. When you create a `LinkedHashMap`, you can order it based on key access rather than insertion. In other words, merely looking up the value associated with a key brings that key to the end of the map. Also, `LinkedHashMap` provides the `removeEldestEntry` method, which may be overridden to impose a policy for removing stale mappings automatically when new mappings are added to the map. This makes it very easy to implement a custom cache.

For example, this override will allow the map to grow up to as many as 100 entries and then it will delete the eldest entry each time a new entry is added, maintaining a steady state of 100 entries.

```
private static final int MAX_ENTRIES = 100;

protected boolean removeEldestEntry(Map.Entry eldest) {
    return size() > MAX_ENTRIES;
}
```

Special-Purpose Map Implementations

There are three special-purpose Map implementations — `EnumMap`, `WeakHashMap` and `IdentityHashMap`. `EnumMap`, which is internally implemented as an `array`, is a high-performance `Map` implementation for use with enum keys. This implementation combines the richness and safety of the `Map` interface with a speed approaching that of an array. If you want to map an enum to a value, you should always use an `EnumMap` in preference to an array.

`WeakHashMap` is an implementation of the `Map` interface that stores only weak references to its keys. Storing only weak references allows a key-value pair to be garbage-collected when its key is no longer referenced outside of the `WeakHashMap`. This class provides the easiest way to harness the power of weak references. It is useful for implementing "registry-like" data structures, where the utility of an entry vanishes when its key is no longer reachable by any thread.

`IdentityHashMap` is an identity-based `Map` implementation based on a hash table. This class is useful for topology-preserving object graph transformations, such as serialization or deep-copying. To perform such transformations, you need to maintain an identity-based "node table" that keeps track

of which objects have already been seen. Identity-based maps are also used to maintain object-to-meta-information mappings in dynamic debuggers and similar systems. Finally, identity-based maps are useful in thwarting "spoof attacks" that are a result of intentionally perverse `equals` methods because `IdentityHashMap` never invokes the `equals` method on its keys. An added benefit of this implementation is that it is fast.

Concurrent Map Implementations

The `java.util.concurrent` package contains the `ConcurrentMap` interface, which extends `Map` with atomic `putIfAbsent`, `remove`, and `replace` methods, and the `ConcurrentHashMap` implementation of that interface.

`ConcurrentHashMap` is a highly concurrent, high-performance implementation backed up by a hash table. This implementation never blocks when performing retrievals and allows the client to select the concurrency level for updates. It is intended as a drop-in replacement for `Hashtable`: in addition to implementing `ConcurrentMap`, it supports all the legacy methods peculiar to `Hashtable`. Again, if you don't need the legacy operations, be careful to manipulate it with the `ConcurrentMap` interface.

Queue Implementations

The `Queue` implementations are grouped into general-purpose and concurrent implementations.

General-Purpose Queue Implementations

As mentioned in the previous section, `LinkedList` implements the `Queue` interface, providing first in, first out (FIFO) queue operations for `add`, `poll`, and so on.

The `PriorityQueue` class is a priority queue based on the *heap* data structure. This queue orders elements according to the order specified at construction time, which can be the elements' natural ordering or the ordering imposed by an explicit `Comparator`.

The queue retrieval operations — `poll`, `remove`, `peek`, and `element` — access the element at the head of the queue. The *head of the queue* is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements; ties are broken arbitrarily.

`PriorityQueue` and its iterator implement all of the optional methods of the `Collection` and `Iterator` interfaces. The iterator provided in method `iterator` is not guaranteed to traverse the elements of the `PriorityQueue` in any particular order. For ordered traversal, consider using `Arrays.sort(pq.toArray())`.

Concurrent Queue Implementations

The `java.util.concurrent` package contains a set of synchronized `Queue` interfaces and classes. `BlockingQueue` extends `Queue` with operations that wait for the queue to become nonempty when retrieving an element and for space to become available in the queue when storing an element. This interface is implemented by the following classes:

- `LinkedBlockingQueue` — an optionally bounded FIFO blocking queue backed by linked nodes
- `ArrayBlockingQueue` — a bounded FIFO blocking queue backed by an array
- `PriorityBlockingQueue` — an unbounded blocking priority queue backed by a heap
- `DelayQueue` — a time-based scheduling queue backed by a heap
- `SynchronousQueue` — a simple rendezvous mechanism that uses the `BlockingQueue` interface

In JDK 7, `TransferQueue` is a specialized `BlockingQueue` in which code that adds an element to the queue has the option of waiting (blocking) for code in another thread to retrieve the element. `TransferQueue` has a single implementation:

- `LinkedTransferQueue` — an unbounded `TransferQueue` based on linked nodes

Deque Implementations

The `Deque` interface, pronounced as *"deck"*, represents a double-ended queue. The `Deque` interface can be implemented as various types of `Collections`. The `Deque` interface implementations are grouped into general-purpose and concurrent implementations.

General-Purpose Deque Implementations

The general-purpose implementations include `LinkedList` and `ArrayDeque` classes. The `Deque` interface supports insertion, removal and retrieval of elements at both ends. The `ArrayDeque` class is the resizable array implementation of the `Deque` interface, whereas the `LinkedList` class is the list implementation.

The basic insertion, removal and retieval operations in the `Deque` interface `addFirst`, `addLast`, `removeFirst`, `removeLast`, `getFirst` and `getLast`. The method `addFirst` adds an element at the head whereas `addLast` adds an element at the tail of the `Deque` instance.

The `LinkedList` implementation is more flexible than the `ArrayDeque` implementation. `LinkedList` implements all optional list operations. `null` elements are allowed in the `LinkedList` implementation but not in the `ArrayDeque` implementation.

In terms of efficiency, `ArrayDeque` is more efficient than the `LinkedList` for add and remove operation at both ends. The best operation in a `LinkedList` implementation is removing the current element during the iteration. `LinkedList` implementations are not ideal structures to iterate.

The `LinkedList` implementation consumes more memory than the `ArrayDeque` implementation. For the `ArrayDeque` instance traversal use any of the following:

foreach

The `foreach` is fast and can be used for all kinds of lists.

```
ArrayDeque<String> aDeque = new ArrayDeque<String>();

. . .
for (String str : aDeque) {
    System.out.println(str);
}
```

Iterator

The `Iterator` can be used for the forward traversal on all kinds of lists for all kinds of data.

```
ArrayDeque<String> aDeque = new ArrayDeque<String>();
. . .
for (Iterator<String> iter = aDeque.iterator(); iter.hasNext();  ) {
    System.out.println(iter.next());
}
```

The `ArrayDeque` class is used in this tutorial to implement the `Deque` interface. The complete code of the example used in this tutorial is available in `ArrayDequeSample`. Both the `LinkedList` and `ArrayDeque` classes do not support concurrent access by multiple threads.

Concurrent Deque Implementations

The `LinkedBlockingDeque` class is the concurrent implementation of the `Deque` interface. If the deque is empty then methods such as `takeFirst` and `takeLast` wait until the element becomes available, and then retrieves and removes the same element.

Wrapper Implementations

Wrapper implementations delegate all their real work to a specified collection but add extra functionality on top of what this collection offers. For design pattern fans, this is an example of the *decorator* pattern. Although it may seem a bit exotic, it's really pretty straightforward.

These implementations are anonymous; rather than providing a public class, the library provides a static factory method. All these implementations are found in the `Collections` class, which consists solely of static methods.

Synchronization Wrappers

The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces — `Collection`, `Set`, `List`, `Map`, `SortedSet`, and `SortedMap` — has one static factory method.

```
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```

Each of these methods returns a synchronized (thread-safe) `Collection` backed up by the specified collection. To guarantee serial access, *all* access to the backing collection must be accomplished through the returned collection. The easy way to guarantee this is not to keep a reference to the backing collection. Create the synchronized collection with the following trick.

```
List<Type> list = Collections.synchronizedList(new ArrayList<Type>());
```

A collection created in this fashion is every bit as thread-safe as a normally synchronized collection, such as a `Vector`.

In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation. The following is the idiom to iterate over a wrapper-synchronized collection.

```
Collection<Type> c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
    for (Type e : c)
        foo(e);
}
```

If an explicit iterator is used, the `iterator` method must be called from within the `synchronized` block. Failure to follow this advice may result in nondeterministic behavior. The idiom for iterating over a `Collection` view of a synchronized `Map` is similar. It is imperative that the user synchronize

on the synchronized `Map` when iterating over any of its `Collection` views rather than synchronizing on the `Collection` view itself, as shown in the following example.

```
Map<KeyType, ValType> m = Collections.synchronizedMap(new HashMap<KeyType, ValType>());
    ...
Set<KeyType> s = m.keySet();
    ...
// Synchronizing on m, not s!
synchronized(m) {
    while (KeyType k : s)
        foo(k);
}
```

One minor downside of using wrapper implementations is that you do not have the ability to execute any *noninterface* operations of a wrapped implementation. So, for instance, in the preceding `List` example, you cannot call `ArrayList`'s `ensureCapacity` operation on the wrapped `ArrayList`.

Unmodifiable Wrappers

Unlike synchronization wrappers, which add functionality to the wrapped collection, the unmodifiable wrappers take functionality away. In particular, they take away the ability to modify the collection by intercepting all the operations that would modify the collection and throwing an `UnsupportedOperationException`. Unmodifiable wrappers have two main uses, as follows:

- To make a collection immutable once it has been built. In this case, it's good practice not to maintain a reference to the backing collection. This absolutely guarantees immutability.
- To allow certain clients read-only access to your data structures. You keep a reference to the backing collection but hand out a reference to the wrapper. In this way, clients can look but not modify, while you maintain full access.

Like synchronization wrappers, each of the six core `Collection` interfaces has one static factory method.

```
public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c);
public static <T> Set<T> unmodifiableSet(Set<? extends T> s);
public static <T> List<T> unmodifiableList(List<? extends T> list);
public static <K,V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m);
public static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<? extends T> s);
public static <K,V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> m);
```

Checked Interface Wrappers

The `Collections.checked` *interface* wrappers are provided for use with generic collections. These implementations return a *dynamically* type-safe view of the specified collection, which throws a `ClassCastException` if a client attempts to add an element of the wrong type. The generics mechanism in the language provides compile-time (static) type-checking, but it is possible to defeat this mechanism. Dynamically type-safe views eliminate this possibility entirely.

Convenience Implementations

This section describes several mini-implementations that can be more convenient and more efficient than general-purpose implementations when you don't need their full power. All the implementations in this section are made available via static factory methods rather than `public` classes.

List View of an Array

The `Arrays.asList` method returns a `List` view of its array argument. Changes to the `List` write through to the array and vice versa. The size of the collection is that of the array and cannot be changed. If the `add` or the `remove` method is called on the `List`, an `UnsupportedOperationException` will result.

The normal use of this implementation is as a bridge between array-based and collection-based APIs. It allows you to pass an array to a method expecting a `Collection` or a `List`. However, this implementation also has another use. If you need a fixed-size `List`, it's more efficient than any general-purpose `List` implementation. This is the idiom.

```
List<String> list = Arrays.asList(new String[size]);
```

Note that a reference to the backing array is not retained.

Immutable Multiple-Copy List

Occasionally you'll need an immutable `List` consisting of multiple copies of the same element. The `Collections.nCopies` method returns such a list. This implementation has two main uses. The first is to initialize a newly created `List`; for example, suppose you want an `ArrayList` initially consisting of 1,000 `null` elements. The following incantation does the trick.

```
List<Type> list = new ArrayList<Type>(Collections.nCopies(1000, (Type)null);
```

Of course, the initial value of each element need not be `null`. The second main use is to grow an existing `List`. For example, suppose you want to add 69 copies of the string `"fruit bat"` to the end of a `List<String>`. It's not clear why you'd want to do such a thing, but let's just suppose you did. The following is how you'd do it.

```
lovablePets.addAll(Collections.nCopies(69, "fruit bat"));
```

By using the form of `addAll` that takes both an index and a `Collection`, you can add the new elements to the middle of a `List` instead of to the end of it.

Immutable Singleton Set

Sometimes you'll need an immutable *singleton* `Set`, which consists of a single, specified element.

The `Collections.singleton` method returns such a `Set`. One use of this implementation is to remove all occurrences of a specified element from a `Collection`.

```
c.removeAll(Collections.singleton(e));
```

A related idiom removes all elements that map to a specified value from a `Map`. For example, suppose you have a `Map` — `job` — that maps people to their line of work and suppose you want to eliminate all the lawyers. The following one-liner will do the deed.

```
job.values().removeAll(Collections.singleton(LAWYER));
```

One more use of this implementation is to provide a single input value to a method that is written to accept a collection of values.

Empty Set, List, and Map Constants

The `Collections` class provides methods to return the empty `Set`, `List`, and `Map` — `emptySet`, `emptyList`, and `emptyMap`. The main use of these constants is as input to methods that take a `Collection` of values when you don't want to provide any values at all, as in this example.

```
tourist.declarePurchases(Collections.emptySet());
```

Summary of Implementations

Implementations are the data objects used to store collections, which implement the interfaces described in the Interfaces lesson.

The Java Collections Framework provides several general-purpose implementations of the core interfaces:

- For the `Set` interface, `HashSet` is the most commonly used implementation.
- For the `List` interface, `ArrayList` is the most commonly used implementation.
- For the `Map` interface, `HashMap` is the most commonly used implementation.
- For the `Queue` interface, `LinkedList` is the most commonly used implementation.
- For the `Deque` interface, `ArrayDeque` is the most commonly used implementation.

Each of the general-purpose implementations provides all optional operations contained in its interface.

The Java Collections Framework also provides several special-purpose implementations for situations that require nonstandard performance, usage restrictions, or other unusual behavior.

The `java.util.concurrent` package contains several collections implementations, which are thread-safe but not governed by a single exclusion lock.

The `Collections` class (as opposed to the `Collection` interface), provides static methods that operate on or return collections, which are known as Wrapper implementations.

Finally, there are several Convenience implementations, which can be more efficient than general-purpose implementations when you don't need their full power. The Convenience implementations are made available through static factory methods.

Questions and Exercises: Implementations

Questions

1. You plan to write a program that uses several basic collection interfaces: `Set`, `List`, `Queue`, and `Map`. You're not sure which implementations will work best, so you decide to use general-purpose implementations until you get a better idea how your program will work in the real world. Which implementations are these?
2. If you need a `Set` implementation that provides value-ordered iteration, which class should you use?
3. Which class do you use to access wrapper implementations?

Exercises

1. Write a program that reads a text file, specified by the first command line argument, into a `List`. The program should then print random lines from the file, the number of lines printed to be specified by the second command line argument. Write the program so that a correctly-sized collection is allocated all at once, instead of being gradually expanded as the file is read in. Hint: To determine the number of lines in the file, use `java.io.File.length` to obtain the size of the file, then divide by an assumed size of an average line.

Check your answers.

Questions

1. Question: You plan to write a program that uses several basic collection interfaces: `Set`, `List`, `Queue`, and `Map`. You're not sure which implementations will work best, so you decide to use general-purpose implementations until you get a better idea how your program will work in the real world. Which implementations are these?
Answer:
`Set`: `HashSet`
`List`: `ArrayList`
`Queue`: `LinkedList`
`Map`: `HashMap`
2. Question: If you need a `Set` implementation that provides value-ordered iteration, which class should you use?
Answer:
`TreeSet` guarantees that the sorted set is in ascending element order, sorted according to the natural order of the elements or by the `Comparator` provided.
3. Question: Which class do you use to access wrapper implementations?
Answer:
You use the `Collections` class, which provides static methods that operate on or return collections.

Exercises

1. Exercise: Write a program that reads a text file, specified by the first command line argument, into a `List`. The program should then print random lines from the file, the number of lines printed to be specified by the second command line argument. Write the program so that a correctly-sized collection is allocated all at once, instead of being gradually expanded as the file is read in. Hint: To determine the number of lines in the file, use `java.io.File.length` to obtain the size of the file, then divide by an assumed size of an average line.
Answer:
Since we are accessing the `List` randomly, we will use `ArrayList`. We estimate the number of lines by taking the file size and dividing by 50. We then double that figure, since it is more efficient to overestimate than to underestmate.

```
import java.util.*;
import java.io.*;

public class FileList {
    public static void main(String[] args) {
        final int assumedLineLength = 50;
        File file = new File(args[0]);
        List<String> fileList =
            new ArrayList<String>((int)(file.length() / assumedLineLength) * 2);
        BufferedReader reader = null;
        int lineCount = 0;
        try {
            reader = new BufferedReader(new FileReader(file));
            for (String line = reader.readLine(); line != null;
                    line = reader.readLine()) {
                fileList.add(line);
                lineCount++;
            }
        } catch (IOException e) {
```

```
            System.err.format("Could not read %s: %s%n", file, e);
            System.exit(1);
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException e) {}
            }
        }
        int repeats = Integer.parseInt(args[1]);
        Random random = new Random();
        for (int i = 0; i < repeats; i++) {
            System.out.format("%d: %s%n", i,
                    fileList.get(random.nextInt(lineCount - 1)));
        }
    }
}
```

This program actually spends most of its time reading in the file, so pre-allocating the `ArrayList` has little affect on its performance. Specifying an initial capacity in advance is more likely to be useful when your program repeatly creates large `ArrayList` objects without intervening I/O.

Lesson: Algorithms

The *polymorphic algorithms* described here are pieces of reusable functionality provided by the Java platform. All of them come from the `Collections` class, and all take the form of static methods whose first argument is the collection on which the operation is to be performed. The great majority of the algorithms provided by the Java platform operate on `List` instances, but a few of them operate on arbitrary `Collection` instances. This section briefly describes the following algorithms:

Sorting

The `sort` algorithm reorders a `List` so that its elements are in ascending order according to an ordering relationship. Two forms of the operation are provided. The simple form takes a `List` and sorts it according to its elements' *natural ordering*. If you're unfamiliar with the concept of natural ordering, read the Object Ordering section.

The `sort` operation uses a slightly optimized *merge sort* algorithm that is fast and stable:

- **Fast**: It is guaranteed to run in `n log(n)` time and runs substantially faster on nearly sorted lists. Empirical tests showed it to be as fast as a highly optimized quicksort. A quicksort is generally considered to be faster than a merge sort but isn't stable and doesn't guarantee `n log(n)` performance.
- **Stable**: It doesn't reorder equal elements. This is important if you sort the same list repeatedly on different attributes. If a user of a mail program sorts the inbox by mailing date and then sorts it by sender, the user naturally expects that the now-contiguous list of messages from a given sender will (still) be sorted by mailing date. This is guaranteed only if the second sort was stable.

The following `trivial program` prints out its arguments in lexicographic (alphabetical) order.

```
import java.util.*;

public class Sort {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

Let's run the program.

```
% java Sort i walk the line
```

The following output is produced.

```
[i, line, the, walk]
```

The program was included only to show you that algorithms really are as easy to use as they appear to be.

The second form of `sort` takes a `Comparator` in addition to a `List` and sorts the elements with the `Comparator`. Suppose you want to print out the anagram groups from our earlier example in reverse order of size — largest anagram group first. The example that follows shows you how to achieve this with the help of the second form of the `sort` method.

Recall that the anagram groups are stored as values in a `Map`, in the form of `List` instances. The revised printing code iterates through the `Map`'s values view, putting every `List` that passes the minimum-size test into a `List` of `List`s. Then the code sorts this `List`, using a `Comparator` that expects `List` instances, and implements reverse size-ordering. Finally, the code iterates through the sorted `List`, printing its elements (the anagram groups). The following code replaces the printing code at the end of the `main` method in the `Anagrams` example.

```
// Make a List of all anagram groups above size threshold.
List<List<String>> winners = new ArrayList<List<String>>();
for (List<String> l : m.values())
    if (l.size() >= minGroupSize)
        winners.add(l);

// Sort anagram groups according to size
Collections.sort(winners, new Comparator<List<String>>() {
    public int compare(List<String> o1, List<String> o2) {
        return o2.size() - o1.size();
    }});

// Print anagram groups.
for (List<String> l : winners)
    System.out.println(l.size() + ": " + l);
```

Running `the program` on the `same dictionary` as in The Map Interface section, with the same minimum anagram group size (eight), produces the following output.

```
12: [apers, apres, asper, pares, parse, pears, prase,
       presa, rapes, reaps, spare, spear]
11: [alerts, alters, artels, estral, laster, ratels,
       salter, slater, staler, stelar, talers]
10: [least, setal, slate, stale, steal, stela, taels,
       tales, teals, tesla]
9: [estrin, inerts, insert, inters, niters, nitres,
       sinter, triens, trines]
9: [capers, crapes, escarp, pacers, parsec, recaps,
       scrape, secpar, spacer]
9: [palest, palets, pastel, petals, plates, pleats,
```

```
      septal, staple, tepals]
9: [anestri, antsier, nastier, ratines, retains, retinas,
      retsina, stainer, stearin]
8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]
8: [aspers, parses, passer, prases, repass, spares,
      sparse, spears]
8: [enters, nester, renest, rentes, resent, tenser,
      ternes,��treens]
8: [arles, earls, lares, laser, lears, rales, reals, seral]
8: [earings, erasing, gainers, reagins, regains, reginas,
      searing, seringa]
8: [peris, piers, pries, prise, ripes, speir, spier, spire]
8: [ates, east, eats, etas, sate, seat, seta, teas]
8: [carets, cartes, caster, caters, crates, reacts,
      recast,��traces]
```

Shuffling

The `shuffle` algorithm does the opposite of what `sort` does, destroying any trace of order that may have been present in a `List`. That is, this algorithm reorders the `List` based on input from a source of randomness such that all possible permutations occur with equal likelihood, assuming a fair source of randomness. This algorithm is useful in implementing games of chance. For example, it could be used to shuffle a `List` of `Card` objects representing a deck. Also, it's useful for generating test cases.

This operation has two forms: one takes a `List` and uses a default source of randomness, and the other requires the caller to provide a Random object to use as a source of randomness. The code for this algorithm is used as an example in the `List` section.

Routine Data Manipulation

The `Collections` class provides five algorithms for doing routine data manipulation on `List` objects, all of which are pretty straightforward:

- `reverse` — reverses the order of the elements in a `List`.
- `fill` — overwrites every element in a `List` with the specified value. This operation is useful for reinitializing a `List`.
- `copy` — takes two arguments, a destination `List` and a source `List`, and copies the elements of the source into the destination, overwriting its contents. The destination `List` must be at least as long as the source. If it is longer, the remaining elements in the destination `List` are unaffected.
- `swap` — swaps the elements at the specified positions in a `List`.
- `addAll` — adds all the specified elements to a `Collection`. The elements to be added may be specified individually or as an array.

Searching

The `binarySearch` algorithm searches for a specified element in a sorted `List`. This algorithm has two forms. The first takes a `List` and an element to search for (the "search key"). This form assumes that the `List` is sorted in ascending order according to the natural ordering of its elements. The second form takes a `Comparator` in addition to the `List` and the search key, and assumes that the `List` is sorted into ascending order according to the specified `Comparator`. The `sort` algorithm can

be used to sort the `List` prior to calling `binarySearch`.

The return value is the same for both forms. If the `List` contains the search key, its index is returned. If not, the return value is `(-(insertion point) - 1)`, where the insertion point is the point at which the value would be inserted into the `List`, or the index of the first element greater than the value or `list.size()` if all elements in the `List` are less than the specified value. This admittedly ugly formula guarantees that the return value will be `>= 0` if and only if the search key is found. It's basically a hack to combine a boolean `(found)` and an integer `(index)` into a single `int` return value.

The following idiom, usable with both forms of the `binarySearch` operation, looks for the specified search key and inserts it at the appropriate position if it's not already present.

```
int pos = Collections.binarySearch(list, key);
if (pos < 0)
   l.add(-pos-1, key);
```

Composition

The frequency and disjoint algorithms test some aspect of the composition of one or more `Collections`:

- `frequency` — counts the number of times the specified element occurs in the specified collection
- `disjoint` — determines whether two `Collections` are disjoint; that is, whether they contain no elements in common

Finding Extreme Values

The `min` and the `max` algorithms return, respectively, the minimum and maximum element contained in a specified `Collection`. Both of these operations come in two forms. The simple form takes only a `Collection` and returns the minimum (or maximum) element according to the elements' natural ordering. The second form takes a `Comparator` in addition to the `Collection` and returns the minimum (or maximum) element according to the specified `Comparator`.

Note: See online version of topics in this ebook to download complete source code.

Lesson: Custom Collection Implementations

Many programmers will never need to implement their own `Collection`s classes. You can go pretty far using the implementations described in the preceding sections of this chapter. However, someday you might want to write your own implementation. It is fairly easy to do this with the aid of the abstract implementations provided by the Java platform. Before we discuss *how* to write an implementation, let's discuss why you might want to write one.

Reasons to Write an Implementation

The following list illustrates the sort of custom `Collection`s you might want to implement. It is not intended to be exhaustive:

- **Persistent**: All of the built-in `Collection` implementations reside in main memory and vanish when the program exits. If you want a collection that will still be present the next time the program starts, you can implement it by building a veneer over an external database. Such a collection might be concurrently accessible by multiple programs.
- **Application-specific**: This is a very broad category. One example is an unmodifiable `Map` containing real-time telemetry data. The keys could represent locations, and the values could be read from sensors at these locations in response to the `get` operation.
- **High-performance, special-purpose**: Many data structures take advantage of restricted usage to offer better performance than is possible with general-purpose implementations. For instance, consider a `List` containing long runs of identical element values. Such lists, which occur frequently in text processing, can be *run-length encoded* — runs can be represented as a single object containing the repeated element and the number of consecutive repetitions. This example is interesting because it trades off two aspects of performance: It requires less space but more time than an `ArrayList`.
- **High-performance, general-purpose**: The Java Collections Framework's designers tried to provide the best general-purpose implementations for each interface, but many, many data structures could have been used, and new ones are invented every day. Maybe you can come up with something faster!
- **Enhanced functionality**: Suppose you need an efficient bag implementation (also known as a *multiset*): a `Collection` that offers constant-time containment checks while allowing duplicate elements. It's reasonably straightforward to implement such a collection atop a `HashMap`.
- **Convenience**: You may want additional implementations that offer conveniences beyond those offered by the Java platform. For instance, you may frequently need `List` instances representing a contiguous range of `Integer`s.
- **Adapter**: Suppose you are using a legacy API that has its own ad hoc collections' API. You can write an adapter implementation that permits these collections to operate in the Java Collections Framework. An *adapter implementation* is a thin veneer that wraps objects of one type and makes them behave like objects of another type by translating operations on the latter type into operations on the former.

How to Write a Custom Implementation

Writing a custom implementation is surprisingly easy. The Java Collections Framework provides abstract implementations designed expressly to facilitate custom implementations. We'll start with the following example of an implementation of `Arrays.asList`.

```
public static <T> List<T> asList(T[] a) {
    return new MyArrayList<T>(a);
}

private static class MyArrayList<T> extends AbstractList<T> {

    private final T[] a;

    MyArrayList(T[] array) {
        a = array;
    }

    public T get(int index) {
        return a[index];
    }

    public T set(int index, T element) {
        T oldValue = a[index];
        a[index] = element;
        return oldValue;
    }

    public int size() {
        return a.length;
    }
}
```

Believe it or not, this is very close to the implementation that is contained in `java.util.Arrays`. It's that simple! You provide a constructor and the `get`, `set`, and `size` methods, and `AbstractList` does all the rest. You get the `ListIterator`, bulk operations, search operations, hash code computation, comparison, and string representation for free.

Suppose you want to make the implementation a bit faster. The API documentation for abstract implementations describes precisely how each method is implemented, so you'll know which methods to override to get the performance you want. The preceding implementation's performance is fine, but it can be improved a bit. In particular, the `toArray` method iterates over the `List`, copying one element at a time. Given the internal representation, it's a lot faster and more sensible just to clone the array.

```
public Object[] toArray() {
    return (Object[]) a.clone();
}
```

With the addition of this override and a few more like it, this implementation is exactly the one found in `java.util.Arrays`. In the interest of full disclosure, it's a bit tougher to use the other abstract implementations because you will have to write your own iterator, but it's still not that difficult.

The following list summarizes the abstract implementations:

- `AbstractCollection` — a `Collection` that is neither a `Set` nor a `List`. At a minimum, you

must provide the `iterator` and the `size` methods.

- `AbstractSet` — a `Set`; use is identical to `AbstractCollection`.
- `AbstractList` — a `List` backed up by a random-access data store, such as an array. At a minimum, you must provide the `positional access` methods (`get` and, optionally, `set`, `remove`, and `add`) and the `size` method. The abstract class takes care of `listIterator` (and `iterator`).
- `AbstractSequentialList` — a `List` backed up by a sequential-access data store, such as a linked list. At a minimum, you must provide the `listIterator` and `size` methods. The abstract class takes care of the positional access methods. (This is the opposite of `AbstractList`.)
- `AbstractQueue` — at a minimum, you must provide the `offer`, `peek`, `poll`, and `size` methods and an `iterator` supporting `remove`.
- `AbstractMap` — a `Map`. At a minimum you must provide the `entrySet` view. This is typically implemented with the `AbstractSet` class. If the `Map` is modifiable, you must also provide the `put` method.

The process of writing a custom implementation follows:

1. Choose the appropriate abstract implementation class from the preceding list.
2. Provide implementations for all the abstract methods of the class. If your custom collection is to be modifiable, you will have to override one or more of the concrete methods as well. The API documentation for the abstract implementation class will tell you which methods to override.
3. Test and, if necessary, debug the implementation. You now have a working custom collection implementation.
4. If you are concerned about performance, read the API documentation of the abstract implementation class for all the methods whose implementations you're inheriting. If any seem too slow, override them. If you override any methods, be sure to measure the performance of the method before and after the override. How much effort you put into tweaking performance should be a function of how much use the implementation will get and how critical to performance its use is. (Often this step is best omitted.)

Note: See online version of topics in this ebook to download complete source code.

Lesson: Interoperability

In this section, you'll learn about the following two aspects of interoperability:

- Compatibility: This subsection describes how collections can be made to work with older APIs that predate the addition of `Collections` to the Java platform.
- API Design: This subsection describes how to design new APIs so that they will interoperate seamlessly with one another.

Note: See online version of topics in this ebook to download complete source code.

Compatibility

The Java Collections Framework was designed to ensure complete interoperability between the core collection interfaces and the types that were used to represent collections in the early versions of the Java platform: `Vector`, `Hashtable`, array, and `Enumeration`. In this section, you'll learn how to transform old collections to the Java Collections Framework collections and vice versa.

Upward Compatibility

Suppose that you're using an API that returns legacy collections in tandem with another API that requires objects implementing the collection interfaces. To make the two APIs interoperate smoothly, you'll have to transform the legacy collections into modern collections. Luckily, the Java Collections Framework makes this easy.

Suppose the old API returns an array of objects and the new API requires a `Collection`. The Collections Framework has a convenience implementation that allows an array of objects to be viewed as a `List`. You use `Arrays.asList` to pass an array to any method requiring a `Collection` or a `List`.

```
Foo[] result = oldMethod(arg);
newMethod(Arrays.asList(result));
```

If the old API returns a `Vector` or a `Hashtable`, you have no work to do at all because `Vector` was retrofitted to implement the `List` interface, and `Hashtable` was retrofitted to implement `Map`. Therefore, a `Vector` may be passed directly to any method calling for a `Collection` or a `List`.

```
Vector result = oldMethod(arg);
newMethod(result);
```

Similarly, a `Hashtable` may be passed directly to any method calling for a `Map`.

```
Hashtable result = oldMethod(arg);
newMethod(result);
```

Less frequently, an API may return an `Enumeration` that represents a collection of objects. The `Collections.list` method translates an `Enumeration` into a `Collection`.

```
Enumeration e = oldMethod(arg);
newMethod(Collections.list(e));
```

Backward Compatibility

Suppose you're using an API that returns modern collections in tandem with another API that requires you to pass in legacy collections. To make the two APIs interoperate smoothly, you have to transform modern collections into old collections. Again, the Java Collections Framework makes this easy.

Suppose the new API returns a `Collection`, and the old API requires an array of `Object`. As you're probably aware, the `Collection` interface contains a `toArray` method designed expressly for this situation.

```
Collection c = newMethod();
oldMethod(c.toArray());
```

What if the old API requires an array of `String` (or another type) instead of an array of `Object`? You just use the other form of `toArray` — the one that takes an array on input.

```
Collection c = newMethod();
oldMethod((String[]) c.toArray(new String[0]));
```

If the old API requires a `Vector`, the standard collection constructor comes in handy.

```
Collection c = newMethod();
oldMethod(new Vector(c));
```

The case where the old API requires a `Hashtable` is handled analogously.

```
Map m = newMethod();
oldMethod(new Hashtable(m));
```

Finally, what do you do if the old API requires an `Enumeration`? This case isn't common, but it does happen from time to time, and the `Collections.enumeration` method was provided to handle it. This is a static factory method that takes a `Collection` and returns an `Enumeration` over the elements of the `Collection`.

```
Collection c = newMethod();
oldMethod(Collections.enumeration(c));
```

API Design

In this short but important section, you'll learn a few simple guidelines that will allow your API to interoperate seamlessly with all other APIs that follow these guidelines. In essence, these rules define what it takes to be a good "citizen" in the world of collections.

Parameters

If your API contains a method that requires a collection on input, it is of paramount importance that you declare the relevant parameter type to be one of the collection interface types. **Never** use an implementation type because this defeats the purpose of an interface-based Collections Framework, which is to allow collections to be manipulated without regard to implementation details.

Further, you should always use the least-specific type that makes sense. For example, don't require a `List` or a `Set` if a `Collection` would do. It's not that you should never require a `List` or a `Set` on input; it is correct to do so if a method depends on a property of one of these interfaces. For example, many of the algorithms provided by the Java platform require a `List` on input because they depend on the fact that lists are ordered. As a general rule, however, the best types to use on input are the most general: `Collection` and `Map`.

Caution: Never define your own ad hoc `collection` class and require objects of this class on input. By doing this, you'd lose all the benefits provided by the Java Collections Framework.

Return Values

You can afford to be much more flexible with return values than with input parameters. It's fine to return an object of any type that implements or extends one of the collection interfaces. This can be one of the interfaces or a special-purpose type that extends or implements one of these interfaces.

For example, one could imagine an image-processing package, called `ImageList`, that returned objects of a new class that implements `List`. In addition to the `List` operations, `ImageList` could support any application-specific operations that seemed desirable. For example, it might provide an `indexImage` operation that returned an image containing thumbnail images of each graphic in the `ImageList`. It's critical to note that even if the API furnishes `ImageList` instances on output, it should accept arbitrary `Collection` (or perhaps `List`) instances on input.

In one sense, return values should have the opposite behavior of input parameters: It's best to return the most specific applicable collection interface rather than the most general. For example, if you're sure that you'll always return a `SortedMap`, you should give the relevant method the return type of `SortedMap` rather than `Map`. `SortedMap` instances are more time-consuming to build than ordinary `Map` instances and are also more powerful. Given that your module has already invested the time to build a `SortedMap`, it makes good sense to give the user access to its increased power. Furthermore, the user will be able to pass the returned object to methods that demand a `SortedMap`, as well as those that accept any `Map`.

Legacy APIs

There are currently plenty of APIs out there that define their own ad hoc collection types. While this is unfortunate, it's a fact of life, given that there was no Collections Framework in the first two major releases of the Java platform. Suppose you own one of these APIs; here's what you can do about it.

If possible, retrofit your legacy collection type to implement one of the standard collection interfaces. Then all the collections you return will interoperate smoothly with other collection-based APIs. If this is impossible (for example, because one or more of the preexisting type signatures conflict with the standard collection interfaces), define an *adapter class* that wraps one of your legacy collections objects, allowing it to function as a standard collection. (The `Adapter` class is an example of a *custom implementation*.)

Retrofit your API with new calls that follow the input guidelines to accept objects of a standard collection interface, if possible. Such calls can coexist with the calls that take the legacy collection type. If this is impossible, provide a constructor or static factory for your legacy type that takes an object of one of the standard interfaces and returns a legacy collection containing the same elements (or mappings). Either of these approaches will allow users to pass arbitrary collections into your API.

Collections: End of Trail

You have reached the end of the "Collections" trail.

If you have comments or suggestions about this trail, use our feedback page to tell us about it.

 Essential Classes: This trail covers other classes and interfaces that are essential to most Java programs.

 Internationalization: This trail shows you how to ready your program for users all over the world. The information about formatting dates, numbers, and strings can be used by any program.