Chapter 24

Collections in Depth

Scala includes an elegant and powerful collection library. Even though the collections API is subtle at first glance, the changes it can provoke in your programming style can be profound. Quite often it's as if you work on a higher level with the basic building blocks of a program being whole collections instead of their elements. This new style of programming requires some adaptation. Fortunately, the adaptation is helped by several nice properties of Scala collections. They are easy to use, concise, safe, fast, and universal.

- **Easy to use:** A small vocabulary of twenty to fifty methods is enough to solve most collection problems in a couple of operations. No need to wrap your head around complicated looping structures or recursions. Persistent collections and side-effect-free operations mean that you need not worry about accidentally corrupting existing collections with new data. Interference between iterators and collection updates is eliminated.
- **Concise:** You can achieve with a single word what used to take one or several loops. You can express functional operations with lightweight syntax and combine operations effortlessly, so that the result feels like a custom algebra.
- **Safe:** This one has to be experienced to sink in. The statically typed and functional nature of Scala's collections means that the overwhelming majority of errors you might make are caught at compile-time. The reason is that (1) the collection operations themselves are heavily used and therefore well tested. (2) the usages of the collection operation make inputs and output explicit as function parameters and results. (3) These explicit inputs and outputs are subject to static type checking. The bottom line is that the large majority of misuses will manifest themselves as type errors. It's not at all uncommon to have programs of several hundred lines run at first try.
- **Fast:** Collection operations are tuned and optimized in the libraries. As a result, using collections is typically quite efficient. You might be able to do a little bit better with carefully hand-tuned data structures and operations, but you might also do a lot worse by making some suboptimal implementation decisions along the way. What's more, collections are have been adapted to parallel execution on multi-cores. Parallel collections support the same operations as sequential ones, so no new operations need to be learned and no code needs to be rewritten. You can turn a sequential collection into a parallel one simply by invoking the par method.
- **Universal:** Collections provide the same operations on any type where it makes sense to do so. So you can achieve a lot with a fairly small vocabulary of operations. For instance, a string is conceptually a sequence of characters. Consequently, in Scala collections, strings support all sequence operations. The same holds for arrays.

This chapter describes in depth the APIs of the Scala collection classes from a user perspective. You've already seen a quick tour of the collections library, in Chapter 17. This chapter takes you on a more detailed tour, showing all the collection classes and all the methods they define, so it includes everything you need to know to use Scala collections. Looking ahead, Chapter 25 will concentrate on the architecture and extensibility aspects of the library, for people implementing new collection types.

24.1 MUTABLE AND IMMUTABLE COLLECTIONS

As is now familiar to you, Scala collections systematically distinguish between mutable and immutable collections. A mutable collection can be updated or extended in place. This means you can change, add, or remove elements of a collection as a side effect. Immutable collections, by contrast, never change. You still have operations that simulate additions, removals, or updates, but those operations will in each case return a new collection and leave the old collection unchanged.

All collection classes are found in the package scala.collection or one of its subpackages:mutable, immutable, and generic. Most collection classes needed by client code exist in three variants, each of which has different characteristics with respect to mutability. The three variants are located in packages scala.collection, scala.collection.immutable, andscala.collection.mutable.

A collection in package scala.collection.immutable is guaranteed to be immutable for everyone. Such a collection will never change after it is created. Therefore, you can rely on the fact that accessing the same collection value repeatedly at different points in time will always yield a collection with the same elements.

A collection in package scala.collection.mutable is known to have some operations that change the collection in place. These operations let you write code to mutate the collection yourself. However, you must be careful to understand and defend against any updates performed by other parts of the code base.

A collection in package scala.collection can be either mutable or immutable. For instance,scala.collection.IndexedSeq[T] is a supertrait of both scala.collection.immutable.IndexedSeq[T] and its mutable sibling scala.collection.mutable.IndexedSeq[T]. Generally, the root collections in package scala.collection define the same interface as the immutable collections. And typically, the mutable collections in package scala.collection.mutable add some side-effecting modification operations to this immutable interface.

The difference between root collections and immutable collections is that clients of an immutable collection have a guarantee that nobody can mutate the collection, whereas clients of a root collection only know that they can't change the collection themselves. Even though the static type of such a collection provides no operations for modifying the collection, it might still be possible that the run-time type is a mutable collection that can be changed by other clients.

By default, Scala always picks immutable collections. For instance, if you just write Setwithout any prefix or without having imported anything, you get an immutable set, and if you write Iterable you get an immutable iterable, because these are the default bindings imported from the scala package. To get the mutable default versions, you need to write explicitlycollection.mutable.Set, or collection.mutable.Iterable.

The last package in the collection hierarchy is collection.generic. This package contains building blocks for implementing collections. Typically, collection classes defer the implementations of some of their operations to classes in generic. Everyday users of the collection framework on the other hand should need to refer to classes in generic only in exceptional circumstances.

```
Traversable
    Iterable
        Seq
            IndexedSeq
                Vector
                ResizableArray
                GenericArray
            LinearSeq
                MutableList
                List
                Stream
            Buffer
                ListBuffer
                ArrayBuffer
        Set
            SortedSet
                TreeSet
            HashSet (mutable)
            LinkedHashSet
            HashSet (immutable)
            BitSet
            EmptySet, Set1, Set2, Set3, Set4
        Map
            SortedMap
                TreeMap
            HashMap (mutable)
            LinkedHashMap (mutable)
            HashMap (immutable)
            EmptyMap, Map1, Map2, Map3, Map4
```

Collection hierarchy.

24.2 COLLECTIONS CONSISTENCY

The most important collection classes are shown in Figure 24.1. There is quite a bit of commonality shared by all these classes. For instance, every kind of collection can be created by the same uniform syntax, writing the collection class name followed by its elements:

```
Traversable(1, 2, 3)
Iterable("x", "y", "z")
Map("x" -> 24, "y" -> 25, "z" -> 26)
Set(Color.Red, Color.Green, Color.Blue)
SortedSet("hello", "world")
Buffer(x, y, z)
IndexedSeq(1.0, 2.0)
LinearSeq(a, b, c)
```

The same principle also applies for specific collection implementations:

```
List(1, 2, 3)
HashMap("x" -> 24, "y" -> 25, "z" -> 26)
```

The toString methods for all collections produce output written as above, with a type name followed by the elements of the collection in parentheses. All collections support the API provided by Traversable, but their methods all return their own class rather than the root classTraversable. For instance, the map method on List has a return type of List, whereas the mapmethod on Set has a return type of Set. Thus the static return type of these methods is fairly precise:

```
scala> List(1, 2, 3) map (_ + 1)
res0: List[Int] = List(2, 3, 4)

scala> Set(1, 2, 3) map (_ * 2)
res1: scala.collection.immutable.Set[Int] = Set(2, 4, 6)
```

Equality is also organized uniformly for all collection classes; more on this in Section 24.13.

Most of the classes in Figure 24.1 exist in three variants: root, mutable, and immutable. The only exception is the Buffer trait, which only exists as a mutable collection.

In the remainder of this chapter, we will review these classes one by one.

24.3 TRAIT TRAVERSABLE

At the top of the collection hierarchy is trait Traversable. Its only abstract operation is foreach:

```
def foreach[U](f: Elem => U)
```

Collection classes implementing Traversable just need to define this method; all other methods can be inherited from Traversable.

The foreach method is meant to traverse all elements of the collection, and apply the given operation, f, to each element. The type of the operation is Elem => U, where Elem is the type of the collection's elements and U is an arbitrary result type. The invocation of f is done for its side effect only; in fact any function result of f is discarded by foreach.

Traversable also defines many concrete methods, which are all listed in Table 24.1 here. These methods fall into the following categories:

- **Addition** ++, which appends two traversables together, or appends all elements of an iterator to a traversable.
- **Map *operations*** map, flatMap, and collect, which produce a new collection by applying some function to collection elements.
- **Conversions** toIndexedSeq, toIterable, toStream, toArray, toList, toSeq, toSet, and toMap, which turn a Traversable collection into a more specific collection. All these conversions return the receiver object if it already matches the demanded collection type. For instance, applying toList to a list will yield the list itself.
- **Copying operations** copyToBuffer and copyToArray. As their names imply, these copy collection elements to a buffer or array, respectively.

- **Size operations** isEmpty, nonEmpty, size, and hasDefiniteSize. Collections that are traversable can be finite or infinite. An example of an infinite traversable collection is the stream of natural numbers Stream.from(0). The method hasDefiniteSize indicates whether a collection is possibly infinite. If hasDefiniteSize returns true, the collection is certainly finite. If it returns false, the collection might be infinite, in which case size will emit an error or not return.
- **Element retrieval operations** head, last, headOption, lastOption, and find. These select the first or last element of a collection, or else the first element matching a condition. Note, however, that not all collections have a well-defined meaning of what "first" and "last" means. For instance, a hash set might store elements according to their hash keys, which might change from run to run. In that case, the "first" element of a hash set could also be different for different runs of a program. A collection is ordered if it always yields its elements in the same order. Most collections are ordered, but some (such as hash sets) are not—dropping the ordering provides a little bit of extra efficiency. Ordering is often essential to give reproducible tests and help in debugging. That's why Scala collections provide ordered alternatives for all collection types. For instance, the ordered alternative for HashSet is LinkedHashSet.
- **Subcollection retrieval *operations*** takeWhile, tail, init, slice, take, drop, filter,dropWhile, filterNot, and withFilter. These all return some subcollection identified by an index range or a predicate.
- **Subdivision *operations*** splitAt, span, partition, and groupBy, which split the elements of this collection into several subcollections.
- **Element tests** exists, forall, and count, which test collection elements with a given predicate.
- **Folds** foldLeft, foldRight, /:, :\, reduceLeft, reduceRight, which apply a binary operation to successive elements.
- **Specific folds** sum, product, min, and max, which work on collections of specific types (numeric or comparable).
- **String operations** mkString, addString, and stringPrefix, which provide alternative ways of converting a collection to a string.
- **View operations** consisting of two overloaded variants of the view method. A view is a collection that's evaluated lazily. You'll learn more about views in Section 24.14.

Operations in trait Traversable

What it is	What it does
Abstract method:	
xs foreach f	Executes function f for every element of xs.
Addition:	
xs ++ ys	A collection consisting of the elements of both xs and ys. ysis a TraversableOnce collection, *i.e.*, either a Traversable or anIterator.
Maps:	
xs map f	The collection obtained from applying the function f to every element in xs.
xs flatMap f	The collection obtained from applying the collection-valued function f to every element in xs and concatenating the results.

xs collect f	The collection obtained from applying the partial functionf to every element in xs for which it is defined and collecting the results.
Conversions:	
xs.toArray	Converts the collection to an array.
xs.toList	Converts the collection to a list.
xs.toIterable	Converts the collection to an iterable.
xs.toSeq	Converts the collection to a sequence.
xs.toIndexedSeq	Converts the collection to an indexed sequence.
xs.toStream	Converts the collection to a stream (a lazily computed sequence).
xs.toSet	Converts the collection to a set.
xs.toMap	Converts a collection of key/value pairs to a map.
Copying:	
xs copyToBuffer buf	Copies all elements of the collection to buffer buf.
xs copyToArray(arr, s, len)	Copies at most len elements of arr, starting at index s. The last two arguments are optional.
Size info:	
xs.isEmpty	Tests whether the collection is empty.
xs.nonEmpty	Tests whether the collection contains elements.
xs.size	The number of elements in the collection.
xs.hasDefiniteSize	True if xs is known to have finite size.
Element retrieval:	
xs.head	The first element of the collection (or, some element, if no order is defined).
xs.headOption	The first element of xs in an option value, or None if xs is empty.
xs.last	The last element of the collection (or, some element, if no order is defined).
xs.lastOption	The last element of xs in an option value, or None if xs is empty.
xs find p	An option containing the first element in xs that satisfies p, or None if no element qualifies.
Subcollections:	
xs.tail	The rest of the collection except xs.head.
xs.init	The rest of the collection except xs.last.
xs slice (from, to)	A collection consisting of elements in some index range ofxs (from from, up to and excluding to).
xs take n	A collection consisting of the first n elements of xs (or, some arbitrary n elements, if no order is defined).
xs drop n	The rest of the collection except xs take n.
xs takeWhile p	The longest prefix of elements in the collection that all satisfy p.
xs dropWhile p	The collection without the longest prefix of elements that all satisfy p.
xs filter p	The collection consisting of those elements of xs that satisfy the predicate p.
xs withFilter p	A non-strict filter of this collection. All operations on the resulting filter will only apply to those elements of xs for which the condition p is true.
xs filterNot p	The collection consisting of those elements of xs that do not satisfy the predicate p.

Subdivisions:	
xs splitAt n	Splits xs at a position, giving the pair of collections(xs take n, xs drop n).
xs span p	Splits xs according to a predicate, giving the pair of collections (xs takeWhile p, xs.dropWhile p).
xs partition p	Splits xs into a pair of collections; one with elements that satisfy the predicate p, the other with elements that do not, giving the pair of collections(xs filter p, xs.filterNot p).
xs groupBy f	Partitions xs into a map of collections according to a discriminator function f.
Element conditions:	
xs forall p	A boolean indicating whether the predicate p holds for all elements of xs.
xs exists p	A boolean indicating whether the predicate p holds for some element in xs.
xs count p	The number of elements in xs that satisfy the predicate p.
Folds:	
(z /: xs)(op)	Applies binary operation op between successive elements of xs, going left to right, starting with z.
(xs :\ z)(op)	Applies binary operation op between successive elements of xs, going right to left, starting with z.
xs.foldLeft(z)(op)	Same as (z /: xs)(op).
xs.foldRight(z)(op)	Same as (xs :\ z)(op).
xs reduceLeft op	Applies binary operation op between successive elements of non-empty collection xs, going left to right.
xs reduceRight op	Applies binary operation op between successive elements of non-empty collection xs, going right to left.
Specific folds:	
xs.sum	The sum of the numeric element values of collection xs.
xs.product	The product of the numeric element values of collectionxs.
xs.min	The minimum of the ordered element values of collectionxs.
xs.max	The maximum of the ordered element values of collectionxs.
Strings:	
xs addString (b, start, sep, end)	Adds a string to StringBuilder b that shows all elements ofxs between separators sep enclosed in strings start and end.start, sep, and end are all optional.
xs mkString (start, sep, end)	Converts the collection to a string that shows all elements of xs between separators sep enclosed in strings start andend. start, sep, and end are all optional.
xs.stringPrefix	The collection name at the beginning of the string returned from xs.toString.
Views:	
xs.view	Produces a view over xs.
xs view (from, to)	Produces a view that represents the elements in some index range of xs.

24.4 TRAIT ITERABLE

The next trait from the top in Figure 24.1 is Iterable. All methods in this trait are defined in terms of an abstract method, iterator, which yields the collection's elements one by one. The
abstract foreach method inherited from trait Traversable is implemented in Iterable in terms ofiterator. Here is the actual implementation:

```
def foreach[U](f: Elem => U): Unit = {
  val it = iterator
  while (it.hasNext) f(it.next())
}
```

Quite a few subclasses of Iterable override this standard implementation of foreach in Iterable, because they can provide a more efficient implementation. Remember that foreach is the basis of the implementation of all operations in Traversable, so its performance matters.

Two more methods exist in Iterable that return iterators: grouped and sliding. These iterators, however, do not return single elements but whole subsequences of elements of the original collection. The maximal size of these subsequences is given as an argument to these methods. The grouped method chunks its elements into increments, whereas sliding yields a sliding window over the elements. The difference between the two should become clear by looking at the following interpreter interaction:

```
scala> val xs = List(1, 2, 3, 4, 5)
xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val git = xs grouped 3
git: Iterator[List[Int]] = non-empty iterator

scala> git.next()
res2: List[Int] = List(1, 2, 3)

scala> git.next()
res3: List[Int] = List(4, 5)

scala> val sit = xs sliding 3
sit: Iterator[List[Int]] = non-empty iterator

scala> sit.next()
res4: List[Int] = List(1, 2, 3)

scala> sit.next()
res5: List[Int] = List(2, 3, 4)

scala> sit.next()
res6: List[Int] = List(3, 4, 5)
```

Trait Iterable also adds some other methods to Traversable that can be implemented efficiently only if an iterator is available. They are summarized in Table 24.2:

Operations in trait Iterable

What it is	What it does
Abstract method:	

xs.iterator	An iterator that yields every element in xs, in the same order asforeach traverses elements
Other iterators:	
xs grouped size	An iterator that yields fixed-sized "chunks" of this collection
xs sliding size	An iterator that yields a sliding fixed-sized window of elements in this collection
Subcollections:	
xs takeRight n	A collection consisting of the last n elements of xs (or, some arbitrary n elements, if no order is defined)
xs dropRight n	The rest of the collection except xs takeRight n
Zippers:	
xs zip ys	An iterable of pairs of corresponding elements from xs and ys
xs zipAll (ys, x, y)	An iterable of pairs of corresponding elements from xs and ys, where the shorter sequence is extended to match the longer one by appending elements x or y
xs.zipWithIndex	An iterable of pairs of elements from xs with their indicies
Comparison:	
xs sameElements ys	Tests whether xs and ys contain the same elements in the same order

Why have both Traversable and Iterable?

You might wonder why the extra trait Traversable is above Iterable. Can we not do everything with an iterator? So what's the point of having a more abstract trait that defines its methods in terms of foreach instead of iterator? One reason for having Traversable is that sometimes it is easier or more efficient to provide an implementation of foreach than to provide an implementation of iterator. Here's a simple example. Let's say you want a class hierarchy for binary trees that have integer elements at the leaves. You might design this hierarchy like this:

```
sealed abstract class Tree
case class Branch(left: Tree, right: Tree) extends Tree
case class Node(elem: Int) extends Tree
```

Now assume you want to make trees traversable. To do this, have Tree inherit fromTraversable[Int] and define a foreach method like this:

```
sealed abstract class Tree extends Traversable[Int] {
  def foreach[U](f: Int => U) = this match {
   case Node(elem) => f(elem)
   case Branch(l, r) => l foreach f; r foreach f
  }
}
```

That's not too hard, and it is also very efficient—traversing a balanced tree takes time proportional to the number of elements in the tree. To see this, consider that for a balanced tree with N leaves you will have N - 1 interior nodes of class Branch. So the total number of steps to traverse the tree is N + N - 1.

Now, compare this with making trees iterable. To do this, have Tree inherit from Iterable[Int]and define an iterator method like this:

```
sealed abstract class Tree extends Iterable[Int] {
    def iterator: Iterator[Int] = this match {
      case Node(elem) => Iterator.single(elem)
      case Branch(l, r) => l.iterator ++ r.iterator
    }
  }
```

At first glance, this looks no harder than the foreach solution. However, there's an efficiency problem that has to do with the implementation of the iterator concatenation method, ++. Every time an element is produced by a concatenated iterator such as l.iterator ++ r.iterator, the computation needs to follow one indirection to get at the right iterator (either l.iterator, or r.iterator). Overall, that makes *log*(*N*) indirections to get at a leaf of a balanced tree with Nleaves. So the cost of visiting all elements of a tree went up from about 2N for the foreachtraversal method to N *log*(N) for the traversal with iterator. If the tree has a million elements that means about two million steps for foreach and about twenty million steps for iterator. So the foreach solution has a clear advantage.

Subcategories of Iterable

In the inheritance hierarchy below Iterable you find three traits: Seq, Set, and Map. A common aspect of these three traits is that they all implement the PartialFunction trait[1] with its applyand isDefinedAt methods. However, the way each trait implements PartialFunction differs.

For sequences, apply is positional indexing, where elements are always numbered from 0. That is, Seq(1, 2, 3)(1) == 2. For sets, apply is a membership test. For instance, Set('a', 'b', 'c')('b') == true whereas Set()('a') == false. Finally for maps, apply is a selection. For instance,Map('a' -> 1, 'b' -> 10, 'c' -> 100)('b') == 10.

In the following three sections, we will explain each of the three kinds of collections in more detail.

24.5 THE SEQUENCE TRAITS SEQ, INDEXEDSEQ, AND LINEARSEQ

The Seq trait represents sequences. A sequence is a kind of iterable that has a length and whose elements have fixed index positions, starting from 0.

The operations on sequences, summarized in Figure 24.3, fall into the following categories:

- **Indexing and length operations** apply, isDefinedAt, length, indices, and lengthCompare. For a Seq, the apply operation means indexing; hence a sequence of type Seq[T] is a partial function that takes an Int argument (an index) and yields a sequence element of type T. In other words Seq[T] extends PartialFunction[Int, T]. The elements of a sequence are indexed from zero up to the length of the sequence minus one. The length method on sequences is an alias of the size method of general collections. The lengthCompare method allows you to compare the lengths of two sequences even if one of the sequences has infinite length.
- **Index search *operations*** indexOf, lastIndexOf, indexOfSlice, lastIn- dexOfSlice, indexWhere,lastIndexWhere, segmentLength, and prefixLength, which return the index of an element equal to a given value or matching some predicate.

- **Addition *operations*** +:, :+, and padTo, which return new sequences obtained by adding elements at the front or the end of a sequence.
- **Update operations** updated and patch, which return a new sequence obtained by replacing some elements of the original sequence.
- **Sorting operations** sorted, sortWith, and sortBy, which sort sequence elements according to various criteria.
- **Reversal operations** reverse, reverseIterator, and reverseMap, which yield or process sequence elements in reverse order, from last to first.
- **Comparison operations** startsWith, endsWith, contains, corresponds, and containsSlice, which relate two sequences or search an element in a sequence.
- **Multiset operations** intersect, diff, union, and distinct, which perform set-like operations on the elements of two sequences or remove duplicates.

If a sequence is mutable, it offers in addition a side-effecting update method, which lets sequence elements be updated. Recall from Chapter 3 that syntax like seq(idx) = elem is just a shorthand for seq.update(idx, elem). Note the difference between update and updated. The updatemethod changes a sequence element in place, and is only available for mutable sequences. The updated method is available for all sequences and always returns a new sequence instead of modifying the original.

Operations in trait Seq

What it is	**What it does**
Indexing and length:	
xs(i)	(or, written out, xs apply i) The element of xs at index i.
xs isDefinedAt i	Tests whether i is contained in xs.indices.
xs.length	The length of the sequence (same as size).
xs.lengthCompare ys	Returns -1 if xs is shorter than ys, +1 if it is longer, and 0 is they have the same length. Works even if one of the sequences is infinite.
xs.indices	The index range of xs, extending from 0 to xs.length - 1.
Index search:	
xs indexOf x	The index of the first element in xs equal to x (several variants exist).
xs lastIndexOf x	The index of the last element in xs equal to x (several variants exist).
xs indexOfSlice ys	The first index of xs such that successive elements starting from that index form the sequence ys.
xs lastIndexOfSlice ys	The last index of xs such that successive elements starting from that index form the sequence ys.
xs indexWhere p	The index of the first element in xs that satisfies p (several variants exist).
xs segmentLength (p, i)	The length of the longest uninterrupted segment of elements in xs, starting with xs(i), that all satisfy the predicate p.
xs prefixLength p	The length of the longest prefix of elements in xs that all satisfy the predicate p.
Additions:	
x +: xs	A new sequence consisting of x prepended to xs.

xs :+ x	A new sequence that consists of x append to xs.
xs padTo (len, x)	The sequence resulting from appending the value x to xs until length len is reached.
Updates:	
xs patch (i, ys, r)	The sequence resulting from replacing r elements of xsstarting with i by the patch ys.
xs updated (i, x)	A copy of xs with the element at index i replaced by x.
xs(i) = x	(or, written out, xs.update(i, x), only available for mutable.Seqs) Changes the element of xs at index i to y.
Sorting:	
xs.sorted	A new sequence obtained by sorting the elements of xs using the standard ordering of the element type of xs.
xs sortWith lessThan	A new sequence obtained by sorting the elements of xs, usinglessThan as comparison operation.
xs sortBy f	A new sequence obtained by sorting the elements of xs. Comparison between two elements proceeds by mapping the function f over both and comparing the results.
Reversals:	
xs.reverse	A sequence with the elements of xs in reverse order.
xs.reverseIterator	An iterator yielding all the elements of xs in reverse order.
xs reverseMap f	A sequence obtained by mapping f over the elements of xs in reverse order.
Comparisons:	
xs startsWith ys	Tests whether xs starts with sequence ys (several variants exist).
xs endsWith ys	Tests whether xs ends with sequence ys (several variants exist).
xs contains x	Tests whether xs has an element equal to x.
xs containsSlice ys	Tests whether xs has a contiguous subsequence equal to ys.
(xs corresponds ys)(p)	Tests whether corresponding elements of xs and ys satisfy the binary predicate p.
Multiset operations:	
xs intersect ys	The multi-set intersection of sequences xs and ys that preserves the order of elements in xs.
xs diff ys	The multi-set difference of sequences xs and ys that preserves the order of elements in xs.
xs union ys	Multiset union; same as xs ++ ys.
xs.distinct	A subsequence of xs that contains no duplicated element.

Each Seq trait has two subtraits, LinearSeq and IndexedSeq. These do not add any new operations, but each offers different performance characteristics. A linear sequence has
efficient head and tail operations, whereas an indexed sequence has efficient apply, length, and (if mutable) update operations. List is a frequently used linear sequence, as is Stream. Two frequently used indexed sequences are Array and ArrayBuffer. The Vector class provides an interesting compromise between indexed and linear access. It has both effectively constant time indexing overhead and constant

time linear access overhead. Because of this, vectors are a good foundation for mixed access patterns where both indexed and linear accesses are used. More on vectors in Section 24.8.

Buffers

An important sub-category of mutable sequences is buffers. Buffers allow not only updates of existing elements but also element insertions, element removals, and efficient additions of new elements at the end of the buffer. The principal new methods supported by a buffer are+= and ++=, for element addition at the end, +=: and ++=: for addition at the front, insert andinsertAll for element insertions, as well as remove and -= for element removal. These operations are summarized in Table 24.4.

Two Buffer implementations that are commonly used are ListBuffer and ArrayBuffer. As the name implies, a ListBuffer is backed by a List and supports efficient conversion of its elements to a List, whereas an ArrayBuffer is backed by an array, and can be quickly converted into one. You saw a glimpse of the implementation of ListBuffer in Section 22.2.

Operations in trait Buffer

What it is	**What it does**
Additions:	
buf += x	Appends element x to buffer buf, and returns buf itself as result
buf += (x, y, z)	Appends given elements to buffer
buf ++= xs	Appends all elements in xs to buffer
x +=: buf	Prepends element x to buffer
xs ++=: buf	Prepends all elements in xs to buffer
buf insert (i, x)	Inserts element x at index i in buffer
buf insertAll (i, xs)	Inserts all elements in xs at index i in buffer
Removals:	
buf -= x	Removes element x from buffer
buf remove i	Removes element at index i from buffer
buf remove (i, n)	Removes n elements starting at index i from buffer
buf trimStart n	Removes first n elements from buffer
buf trimEnd n	Removes last n elements from buffer
buf.clear()	Removes all elements from buffer
Cloning:	
buf.clone	A new buffer with the same elements as buf

24.6 SETS

Sets are Iterables that contain no duplicate elements. The operations on sets are summarized in Table 24.5 for general sets and Table 24.6 for mutable sets. They fall into the following categories:

- **Tests** contains, apply, and subsetOf. The contains method indicates whether a set contains a given element. The apply method for a set is the same as contains, so set(elem) is the same as set contains elem. That means sets can also be used as test functions that return true for the elements they contain. For example:

```
scala> val fruit = Set("apple", "orange", "peach", "banana")
fruit: scala.collection.immutable.Set[String] =
  Set(apple, orange, peach, banana)

scala> fruit("peach")
res7: Boolean = true

scala> fruit("potato")
res8: Boolean = false
```

- **Additions** + and ++, which add one or more elements to a set, yielding a new set as a result.
- **Removals -** and --, which remove one or more elements from a set, yielding a new set.
- **Set operations** for union, intersection, and set difference. These set operations exist in two forms: alphabetic and symbolic. The alphabetic versions are intersect, union, and diff, whereas the symbolic versions are &, |, and &~. The ++ that Set inherits from Traversable can be seen as yet another alias of union or |, except that ++ takes a Traversable argument whereas union and | take sets.

Operations in trait Set

What it is	What it does
Tests:	
xs contains x	Tests whether x is an element of xs
xs(x)	Same as xs contains x
xs subsetOf ys	Tests whether xs is a subset of ys
Additions:	
xs + x	The set containing all elements of xs as well as x
xs + (x, y, z)	The set containing all elements of xs as well as the given additional elements
xs ++ ys	The set containing all elements of xs as well as all elements of ys
Removals:	
xs - x	The set containing all elements of xs except x
xs - (x, y, z)	The set containing all elements of xs except the given elements
xs -- ys	The set containing all elements of xs except the elements of ys
xs.empty	An empty set of the same class as xs
Binary operations:	
xs & ys	The set intersection of xs and ys
xs intersect ys	Same as xs & ys
xs | ys	The set union of xs and ys
xs union ys	Same as xs | ys
xs &~ ys	The set difference of xs and ys
xs diff ys	Same as xs &~ ys

Mutable sets have methods that add, remove, or update elements, which are summarized inTable 24.6:

Operations in trait mutable.Set

What it is	What it does
Additions:	
xs += x	Adds element x to set xs as a side effect and returns xs itself
xs += (x, y, z)	Adds the given elements to set xs as a side effect and returns xs itself
xs ++= ys	Adds all elements in ys to set xs as a side effect and returns xs itself
xs add x	Adds element x to xs and returns true if x was not previously contained in the set, false if it was previously contained
Removals:	
xs -= x	Removes element x from set xs as a side effect and returns xs itself
xs -= (x, y, z)	Removes the given elements from set xs as a side effect and returns xsitself
xs --= ys	Removes all elements in ys from set xs as a side effect and returns xsitself
xs remove x	Removes element x from xs and returns true if x was previously contained in the set, false if it was not previously contained
xs retain p	Keeps only those elements in xs that satisfy predicate p
xs.clear()	Removes all elements from xs
Update:	
xs(x) = b	(or, written out, xs.update(x, b)) If boolean argument b is true, adds x toxs, otherwise removes x from xs
Cloning:	
xs.clone	A new mutable set with the same elements as xs

Just like an immutable set, a mutable set offers the + and ++ operations for element additions and the - and -- operations for element removals. But these are less often used for mutable sets since they involve copying the set. As a more efficient alternative, mutable sets offer the update methods += and -=. The operation s += elem adds elem to the set s as a side effect, and returns the mutated set as a result. Likewise, s -= elem removes elem from the set, and returns the mutated set as a result. Besides += and -= there are also the bulk operations ++= and --=, which add or remove all elements of a traversable or an iterator.

The choice of the method names += and -= means that very similar code can work with either mutable or immutable sets. Consider first the following interpreter dialogue that uses an immutable set s:

```
scala> var s = Set(1, 2, 3)
s: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> s += 4; s -= 2

scala> s
res10: scala.collection.immutable.Set[Int] = Set(1, 3, 4)
```

In this example, we used += and -= on a var of type immutable.Set. As was explained in Step 10 inChapter 3, a statement such as s += 4 is an abbreviation for s = s + 4. So this invokes the addition method + on the set s and then assigns the result back to the s variable. Consider now an analogous interaction with a mutable set:

```
scala> val s = collection.mutable.Set(1, 2, 3)
s: scala.collection.mutable.Set[Int] = Set(1, 2, 3)

scala> s += 4
res11: s.type = Set(1, 2, 3, 4)

scala> s -= 2
res12: s.type = Set(1, 3, 4)
```

The end effect is very similar to the previous interaction; we start with a Set(1, 2, 3) and end up with a Set(1, 3, 4). However, even though the statements look the same as before, they do something different. The s += 4 statement now invokes the += method on the mutable set values, changing the set in place. Likewise, the s -= 2 statement now invokes the -= method on the same set.

Comparing the two interactions shows an important principle. You often can replace a mutable collection stored in a val by an immutable collection stored in a var, and vice versa. This works at least as long as there are no alias references to the collection through which you can observe whether it was updated in place or a new collection was created.

Mutable sets also provide add and remove as variants of += and -=. The difference is that add andremove return a boolean result indicating whether the operation had an effect on the set.

The current default implementation of a mutable set uses a hash table to store the set's elements. The default implementation of an immutable set uses a representation that adapts to the number of elements of the set. An empty set is represented by just a singleton object. Sets of sizes up to four are represented by a single object that stores all elements as fields. Beyond that size, immutable sets are implemented as hash tries.[2]

A consequence of these representation choices is that for sets of small sizes, up to about four, immutable sets are more compact and more efficient than mutable sets. So if you expect the size of a set to be small, try to make it immutable.

24.7 MAPS

Maps are Iterables of pairs of keys and values (also named mappings or associations). As explained in Section 21.4, Scala's Predef class offers an implicit conversion that lets you writekey -> value as an alternate syntax for the pair (key, value). Therefore, Map("x" -> 24, "y" -> 25, "z" -> 26) means exactly the same as Map(("x", 24), ("y", 25), ("z", 26)), but reads better.

The fundamental operations on maps, summarized in Table 24.7, are similar to those on sets. Mutable maps additionally support the operations shown in Table 24.8. Map operations fall into the following categories:

- **Lookups** apply, get, getOrElse, contains, and isDefinedAt. These operations turn maps into partial functions from keys to values. The fundamental lookup method for a map is:

  ```
  def get(key): Option[Value]
  ```

The operation "m get key" tests whether the map contains an association for the given key. If so, it returns the associated value in a Some. If no key is defined in the map, get returns None. Maps also define an apply method that returns the value associated with a given key directly, without wrapping it in an Option. If the key is not defined in the map, an exception is raised.

- **Additions and updates** +, ++, and updated, which let you add new bindings to a map or change existing bindings.
- **Removals -** and --, which remove bindings from a map.
- **Subcollection producers** keys, keySet, keysIterator, valuesIterator, and values, which return a map's keys and values separately in various forms.
- **Transformations** filterKeys and mapValues, which produce a new map by filtering and transforming bindings of an existing map.

Operations in trait Map

What it is	What it does
Lookups:	
ms get k	The value associated with key k in map ms as an option, or Noneif not found
ms(k)	(or, written out, ms apply k) The value associated with key k in map ms, or a thrown exception if not found
ms getOrElse (k, d)	The value associated with key k in map ms, or the default value dif not found
ms contains k	Tests whether ms contains a mapping for key k
ms isDefinedAt k	Same as contains
Additions and updates:	
ms + (k -> v)	The map containing all mappings of ms as well as the mappingk -> v from key k to value v
ms + (k -> v, l -> w)	The map containing all mappings of ms as well as the given key/value pairs
ms ++ kvs	The map containing all mappings of ms as well as all key/value pairs of kvs
ms updated (k, v)	Same as ms + (k -> v)
Removals:	
ms - k	The map containing all mappings of ms except for any mapping of key k
ms - (k, l, m)	The map containing all mappings of ms except for any mapping with the given keys
ms -- ks	The map containing all mappings of ms except for any mapping with a key in ks
Subcollections:	
ms.keys	An iterable containing each key in ms
ms.keySet	A set containing each key in ms
ms.keysIterator	An iterator yielding each key in ms
ms.values	An iterable containing each value associated with a key in ms
ms.valuesIterator	An iterator yielding each value associated with a key in ms
Transformation:	

ms filterKeys p	A map view containing only those mappings in ms where the key satisfies predicate p
ms mapValues f	A map view resulting from applying function f to each value associated with a key in ms

Operations in trait mutable.Map

What it is	What it does
Additions and updates:	
ms(k) = v	(or, written out, ms.update(k, v)) Adds mapping from key k to value v to map ms as a side effect, overwriting any previous mapping of k
ms += (k -> v)	Adds mapping from key k to value v to map ms as a side effect and returns ms itself
ms += (k -> v, l -> w)	Adds the given mappings to ms as a side effect and returns msitself
ms ++= kvs	Adds all mappings in kvs to ms as a side effect and returns msitself
ms put (k, v)	Adds mapping from key k to value v to ms and returns any value previously associated with k as an option
ms getOrElseUpdate (k, d)	If key k is defined in map ms, returns its associated value. Otherwise, updates ms with the mapping k -> d and returns d
Removals:	
ms -= k	Removes mapping with key k from ms as a side effect and returns ms itself
ms -= (k, l, m)	Removes mappings with the given keys from ms as a side effect and returns ms itself
ms --= ks	Removes all keys in ks from ms as a side effect and returns msitself
ms remove k	Removes any mapping with key k from ms and returns any value previously associated with k as an option
ms retain p	Keeps only those mappings in ms that have a key satisfying predicate p.
ms.clear()	Removes all mappings from ms
Transformation and cloning:	
ms transform f	Transforms all associated values in map ms with function f
ms.clone	Returns a new mutable map with the same mappings as ms

The addition and removal operations for maps mirror those for sets. As for sets, mutable maps also support the non-destructive addition operations +, -, and updated, but they are used less frequently because they involve a copying of the mutable map. Instead, a mutable map mis usually updated "in place," using the two variants m(key) = value or m += (key -> value). There is also the variant m put (key, value), which returns an Option value that contains the value previously associated with key, or None if the key did not exist in the map before.

The getOrElseUpdate is useful for accessing maps that act as caches. Say you have an expensive computation triggered by invoking a function f:

```
scala> def f(x: String) = {
       println("taking my time."); Thread.sleep(100)
       x.reverse }
```

```
f: (x: String)String
```

Assume further that f has no side-effects, so invoking it again with the same argument will always yield the same result. In that case you could save time by storing previously computed bindings of argument and results of f in a map, and only computing the result of f if a result of an argument was not found there. You could say the map is a cache for the computations of the function f.

```
scala> val cache = collection.mutable.Map[String, String]()
cache: scala.collection.mutable.Map[String,String] = Map()
```

You can now create a more efficient caching version of the f function:

```
scala> def cachedF(s: String) = cache.getOrElseUpdate(s, f(s))
cachedF: (s: String)String

scala> cachedF("abc")
taking my time.
res16: String = cba

scala> cachedF("abc")
res17: String = cba
```

Note that the second argument to getOrElseUpdate is "by-name," so the computation of f("abc")above is only performed if getOrElseUpdate requires the value of its second argument, which is precisely if its first argument is not found in the cache map. You could also have implementedcachedF directly, using just basic map operations, but it would have have taken more code to do so:

```
def cachedF(arg: String) = cache get arg match {
  case Some(result) => result
  case None =>
    val result = f(arg)
    cache(arg) = result
    result
}
```

24.8 CONCRETE IMMUTABLE COLLECTION CLASSES

Scala provides many concrete immutable collection classes for you to choose from. They differ in the traits they implement (maps, sets, sequences), whether they can be infinite, and the speed of various operations. We'll start by reviewing the most common immutable collection types.

Lists

Lists are finite immutable sequences. They provide constant-time access to their first element as well as the rest of the list, and they have a constant-time cons operation for adding a new element to the front of the list. Many other operations take linear time. See Chapters 16 and22 for extensive discussions about lists.

Streams

A stream is like a list except that its elements are computed lazily. Because of this, a stream can be infinitely long. Only those elements requested will be computed. Otherwise, streams have the same performance characteristics as lists.

Whereas lists are constructed with the :: operator, streams are constructed with the similar-looking #::. Here is a simple example of a stream containing the integers 1, 2, and 3:

```
scala> val str = 1 #:: 2 #:: 3 #:: Stream.empty
str: scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

The head of this stream is 1, and the tail of it has 2 and 3. The tail is not printed here, though, because it hasn't been computed yet! Streams are required to compute lazily, and the toStringmethod of a stream is careful not to force any extra evaluation.

Below is a more complex example. It computes a stream that contains a Fibonacci sequence starting with the given two numbers. A Fibonacci sequence is one where each element is the sum of the previous two elements in the series:

```
scala> def fibFrom(a: Int, b: Int): Stream[Int] =
         a #:: fibFrom(b, a + b)
fibFrom: (a: Int, b: Int)Stream[Int]
```

This function is deceptively simple. The first element of the sequence is clearly a, and the rest of the sequence is the Fibonacci sequence starting with b followed by a + b. The tricky part is computing this sequence without causing an infinite recursion. If the function used :: instead of #::, then every call to the function would result in another call, thus causing an infinite recursion. Since it uses #::, though, the right-hand side is not evaluated until it is requested.

Here are the first few elements of the Fibonacci sequence starting with two ones:

```
scala> val fibs = fibFrom(1, 1).take(7)
fibs: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> fibs.toList
res23: List[Int] = List(1, 1, 2, 3, 5, 8, 13)
```

Vectors

Lists are very efficient when the algorithm processing them is careful to only process their heads. Accessing, adding, and removing the head of a list takes only constant time, whereas accessing or modifying elements later in the list takes time linear in the depth into the list.

Vectors are a collection type that give efficient access to elements beyond the head. Access to any elements of a vector take only "effectively constant time," as defined below. It's a larger constant than for access to the head of a list or for reading an element of an array, but it's a constant nonetheless. As a result, algorithms using vectors do not have to be careful about accessing just the head of the sequence. They can access and modify elements at arbitrary locations, and thus they can be much more convenient to write.

Vectors are built and modified just like any other sequence:

```
scala> val vec = scala.collection.immutable.Vector.empty
vec: scala.collection.immutable.Vector[Nothing] = Vector()

scala> val vec2 = vec :+ 1 :+ 2
vec2: scala.collection.immutable.Vector[Int] = Vector(1, 2)

scala> val vec3 = 100 +: vec2
vec3: scala.collection.immutable.Vector[Int]
  = Vector(100, 1, 2)

scala> vec3(0)
res24: Int = 100
```

Vectors are represented as broad, shallow trees. Every tree node contains up to 32 elements of the vector or contains up to 32 other tree nodes. Vectors with up to 32 elements can be represented in a single node. Vectors with up to 32 * 32 = 1024 elements can be represented with a single indirection. Two hops from the root of the tree to the final element node are sufficient for vectors with up to 215 elements, three hops for vectors with 220, four hops for vectors with 225 elements and five hops for vectors with up to 230 elements. So for all vectors of reasonable size, an element selection involves up to five primitive array selections. This is what we meant when we wrote that element access is "effectively constant time."

Vectors are immutable, so you cannot change an element of a vector in place. However, with the updated method you can create a new vector that differs from a given vector only in a single element:

```
scala> val vec = Vector(1, 2, 3)
vec: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)

scala> vec updated (2, 4)
res25: scala.collection.immutable.Vector[Int] = Vector(1, 2, 4)

scala> vec
res26: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)
```

As the last line above shows, a call to updated has no effect on the original vector vec. Like selection, functional vector updates are also "effectively constant time." Updating an element in the middle of a vector can be done by copying the node that contains the element, and every node that points to it, starting from the root of the tree. This means that a functional update creates between one and five nodes that each contain up to 32 elements or subtrees. This is certainly more expensive than an in-place update in a mutable array, but still a lot cheaper than copying the whole vector.

Because vectors strike a good balance between fast random selections and fast random functional updates, they are currently the default implementation of immutable indexed sequences:

```
scala> collection.immutable.IndexedSeq(1, 2, 3)
res27: scala.collection.immutable.IndexedSeq[Int]
  = Vector(1, 2, 3)
```

Immutable stacks

If you need a last-in-first-out sequence, you can use a Stack. You push an element onto a stack with push, pop an element with pop, and peek at the top of the stack without removing it withtop. All of these operations are constant time.

Here are some simple operations performed on a stack:

```
scala> val stack = scala.collection.immutable.Stack.empty
stack: scala.collection.immutable.Stack[Nothing] = Stack()

scala> val hasOne = stack.push(1)
hasOne: scala.collection.immutable.Stack[Int] = Stack(1)

scala> stack
res28: scala.collection.immutable.Stack[Nothing] = Stack()

scala> hasOne.top
res29: Int = 1

scala> hasOne.pop
res30: scala.collection.immutable.Stack[Int] = Stack()
```

Immutable stacks are used rarely in Scala programs because their functionality is subsumed by lists: A push on an immutable stack is the same as a :: on a list, and a pop on a stack is the same a tail on a list.

Immutable queues

A queue is just like a stack except that it is first-in-first-out rather than last-in-first-out. A simplified implementation of immutable queues was discussed in Chapter 19. Here's how you can create an empty immutable queue:

```
scala> val empty = scala.collection.immutable.Queue[Int]()
empty: scala.collection.immutable.Queue[Int] = Queue()
```

You can append an element to an immutable queue with enqueue:

```
scala> val has1 = empty.enqueue(1)
has1: scala.collection.immutable.Queue[Int] = Queue(1)
```

To append multiple elements to a queue, call enqueue with a collection as its argument:

```
scala> val has123 = has1.enqueue(List(2, 3))
has123: scala.collection.immutable.Queue[Int] = Queue(1, 2,
3)
```

To remove an element from the head of the queue, use dequeue:

```
scala> val (element, has23) = has123.dequeue
element: Int = 1
has23: scala.collection.immutable.Queue[Int] = Queue(2, 3)
```

Note that dequeue returns a pair consisting of the element removed and the rest of the queue.

Ranges

A range is an ordered sequence of integers that are equally spaced apart. For example, "1, 2, 3" is a range, as is "5, 8, 11, 14." To create a range in Scala, use the predefined methods to andby. Here are some examples:

```
scala> 1 to 3
res31: scala.collection.immutable.Range.Inclusive
  = Range(1, 2, 3)

scala> 5 to 14 by 3
res32: scala.collection.immutable.Range = Range(5, 8, 11, 14)
```

If you want to create a range that is exclusive of its upper limit, use the convenience methoduntil instead of to:

```
scala> 1 until 3
res33: scala.collection.immutable.Range = Range(1, 2)
```

Ranges are represented in constant space, because they can be defined by just three numbers: their start, their end, and the stepping value. Because of this representation, most operations on ranges are extremely fast.

Hash tries

Hash tries[3] are a standard way to implement immutable sets and maps efficiently. Their representation is similar to vectors in that they are also trees where every node has 32 elements or 32 subtrees, but selection is done based on a hash code. For instance, to find a given key in a map, you use the lowest five bits of the hash code of the key to select the first subtree, the next five bits the next subtree, and so on. Selection stops once all elements stored in a node have hash codes that differ from each other in the bits that are selected so far. Thus, not all the bits of the hash code are necessarily used.

Hash tries strike a nice balance between reasonably fast lookups and reasonably efficient functional insertions (+) and deletions (-). That's why they underlie Scala's default implementations of immutable maps and sets. In fact, Scala has a further optimization for immutable sets and maps that contain less than five elements. Sets and maps with one to four elements are stored as single objects that just contain the elements (or key/value pairs in the case of a map) as fields. The empty immutable set and empty immutable map is in each case a singleton object—there's no need to duplicate storage for those because an empty immutable set or map will always stay empty.

Red-black trees

Red-black trees are a form of balanced binary trees where some nodes are designated "red" and others "black." Like any balanced binary tree, operations on them reliably complete in time logarithmic to the size of the tree.

Scala provides implementations of sets and maps that use a red-black tree internally. You access them under the names TreeSet and TreeMap:

```
scala> val set = collection.immutable.TreeSet.empty[Int]
set: scala.collection.immutable.TreeSet[Int] = TreeSet()

scala> set + 1 + 3 + 3
res34: scala.collection.immutable.TreeSet[Int] = TreeSet(1, 3)
```

Red-black trees are also the standard implementation of SortedSet in Scala, because they provide an efficient iterator that returns all elements of the set in sorted order.

Immutable bit sets

A bit set represents a collection of small integers as the bits of a larger integer. For example, the bit set containing 3, 2, and 0 would be represented as the integer 1101 in binary, which is 13 in decimal.

Internally, bit sets use an array of 64-bit Longs. The first Long in the array is for integers 0 through 63, the second is for 64 through 127, and so on. Thus, bit sets are very compact so long as the largest integer in the set is less than a few hundred or so.

Operations on bit sets are very fast. Testing for inclusion takes constant time. Adding an item to the set takes time proportional to the number of Longs in the bit set's array, which is typically a small number. Here are some simple examples of the use of a bit set:

```
scala> val bits = scala.collection.immutable.BitSet.empty
bits: scala.collection.immutable.BitSet = BitSet()

scala> val moreBits = bits + 3 + 4 + 4
moreBits: scala.collection.immutable.BitSet = BitSet(3, 4)

scala> moreBits(3)
res35: Boolean = true

scala> moreBits(0)
res36: Boolean = false
```

List maps

A list map represents a map as a linked list of key-value pairs. In general, operations on a list map might have to iterate through the entire list. Thus, operations on a list map take time linear in the size of the map. In fact there is little usage for list maps in Scala because standard immutable maps are almost always faster. The only possible difference is if the map is for some reason constructed in such a way that the first elements in the list are selected much more often than the other elements.

```
scala> val map = collection.immutable.ListMap(
         1 -> "one", 2 -> "two")
map: scala.collection.immutable.ListMap[Int,String] = Map(1
-> one, 2 -> two)

scala> map(2)
res37: String = "two"
```

24.9 CONCRETE MUTABLE COLLECTION CLASSES

Now that you've seen the most commonly used immutable collection classes that Scala provides in its standard library, take a look at the mutable collection classes.

Array buffers

You've already seen array buffers in Section 17.1. An array buffer holds an array and a size. Most operations on an array buffer have the same speed as an array, because the operations simply access and modify the underlying array. Additionally, array buffers can have data efficiently added to the end. Appending an item to an array buffer takes amortized constant time. Thus, array buffers are useful for efficiently building up a large collection whenever the new items are always added to the end. Here are some examples:

```
scala> val buf = collection.mutable.ArrayBuffer.empty[Int]
buf: scala.collection.mutable.ArrayBuffer[Int]
  = ArrayBuffer()

scala> buf += 1
res38: buf.type = ArrayBuffer(1)

scala> buf += 10
res39: buf.type = ArrayBuffer(1, 10)

scala> buf.toArray
res40: Array[Int] = Array(1, 10)
```

List buffers

You've also already seen list buffers in Section 17.1. A list buffer is like an array buffer except that it uses a linked list internally instead of an array. If you plan to convert the buffer to a list once it is built up, use a list buffer instead of an array buffer. Here's an example:[4]

```
scala> val buf = collection.mutable.ListBuffer.empty[Int]
buf: scala.collection.mutable.ListBuffer[Int]
  = ListBuffer()

scala> buf += 1
res41: buf.type = ListBuffer(1)

scala> buf += 10
res42: buf.type = ListBuffer(1, 10)

scala> buf.toList
res43: List[Int] = List(1, 10)
```

String builders

Just like an array buffer is useful for building arrays, and a list buffer is useful for building lists, a string builder is useful for building strings. String builders are so commonly used that they are already imported into the default namespace. Create them with a simplenew StringBuilder, like this:

```
scala> val buf = new StringBuilder
```

```
buf: StringBuilder =

scala> buf += 'a'
res44: buf.type = a

scala> buf ++= "bcdef"
res45: buf.type = abcdef

scala> buf.toString
res46: String = abcdef
```

Linked lists

Linked lists are mutable sequences that consist of nodes that are linked with next pointers. In most languages null would be picked as the empty linked list. That does not work for Scala collections, because even empty sequences must support all sequence methods.LinkedList.empty.isEmpty, in particular, should return true and not throw a NullPointerException. Empty linked lists are encoded instead in a special way: Their next field points back to the node itself.

Like their immutable cousins, linked lists are best operated on sequentially. In addition, linked lists make it easy to insert an element or linked list into another linked list.

Double linked lists

DoubleLinkedLists are like the single linked lists described in the previous subsection, except besides next, they have another mutable field, prev, that points to the element preceding the current node. The main benefit of that additional link is that it makes element removal very fast.

Mutable lists

A MutableList consists of a single linked list together with a pointer that refers to the terminal empty node of that list. This makes list append a constant time operation because it avoids having to traverse the list in search for its terminal node. MutableList is currently the standard implementation of mutable.LinearSeq in Scala.

Queues

Scala provides mutable queues in addition to immutable ones. You use a mutable queue similarly to the way you use an immutable one, but instead of enqueue, you use the += and ++=operators to append. Also, on a mutable queue, the dequeue method will just remove the head element from the queue and return it. Here's an example:

```
scala> val queue = new scala.collection.mutable.Queue[String]
queue: scala.collection.mutable.Queue[String] = Queue()

scala> queue += "a"
res47: queue.type = Queue(a)

scala> queue ++= List("b", "c")
res48: queue.type = Queue(a, b, c)

scala> queue
```

```
res49: scala.collection.mutable.Queue[String] = Queue(a, b, c)

scala> queue.dequeue
res50: String = a

scala> queue
res51: scala.collection.mutable.Queue[String] = Queue(b, c)
```

Array sequences

Array sequences are mutable sequences of fixed size that store their elements internally in anArray[AnyRef]. They are implemented in Scala by class ArraySeq.

You would typically use an ArraySeq if you want an array for its performance characteristics, but you also want to create generic instances of the sequence where you do not know the type of the elements and do not have a ClassTag to provide it at run-time. You will find out about these issues with arrays shortly, in Section 24.10.

Stacks

You saw immutable stacks earlier. There is also a mutable version. It works exactly the same as the immutable version except that modifications happen in place. Here's an example:

```
scala> val stack = new scala.collection.mutable.Stack[Int]
stack: scala.collection.mutable.Stack[Int] = Stack()

scala> stack.push(1)
res52: stack.type = Stack(1)

scala> stack
res53: scala.collection.mutable.Stack[Int] = Stack(1)

scala> stack.push(2)
res54: stack.type = Stack(2, 1)

scala> stack
res55: scala.collection.mutable.Stack[Int] = Stack(2, 1)

scala> stack.top
res56: Int = 2

scala> stack
res57: scala.collection.mutable.Stack[Int] = Stack(2, 1)

scala> stack.pop
res58: Int = 2

scala> stack
res59: scala.collection.mutable.Stack[Int] = Stack(1)
```

Array stacks

ArrayStack is an alternative implementation of a mutable stack, which is backed by an Arraythat gets resized as needed. It provides fast indexing and is generally slightly more efficient for most operations than a normal mutable stack.

Hash tables

A hash table stores its elements in an underlying array, placing each item at a position in the array determined by the hash code of that item. Adding an element to a hash table takes only constant time, so long as there isn't already another element in the array that has the same hash code. Hash tables are thus very fast so long as the objects placed in them have a good distribution of hash codes. As a result, the default mutable map and set types in Scala are based on hash tables.

Hash sets and maps are used just like any other set or map. Here are some simple examples:

```
scala> val map = collection.mutable.HashMap.empty[Int,String]
map: scala.collection.mutable.HashMap[Int,String] = Map()

scala> map += (1 -> "make a web site")
res60: map.type = Map(1 -> make a web site)

scala> map += (3 -> "profit!")
res61: map.type = Map(1 -> make a web site, 3 -> profit!)

scala> map(1)
res62: String = make a web site

scala> map contains 2
res63: Boolean = false
```

Iteration over a hash table is not guaranteed to occur in any particular order. Iteration simply proceeds through the underlying array in whichever order it happens to be. To get a guaranteed iteration order, use a *linked* hash map or set instead of a regular one. A linked hash map or set is just like a regular hash map or set except that it also includes a linked list of the elements in the order they were added. Iteration over such a collection is always in the same order that the elements were initially added.

Weak hash maps

A weak hash map is a special kind of hash map in which the garbage collector does not follow links from the map to the keys stored in it. This means that a key and its associated value will disappear from the map if there is no other reference to that key. Weak hash maps are useful for tasks such as caching, where you want to re-use an expensive function's result if the function is called again on the same key. If keys and function results are stored in a regular hash map, the map could grow without bounds, and no key would ever become garbage. Using a weak hash map avoids this problem. As soon as a key object becomes unreachable, it's entry is removed from the weak hash map. Weak hash maps in Scala are implemented as a wrapper of an underlying Java implementation, java.util.WeakHashMap.

Concurrent Maps

A concurrent map can be accessed by several threads at once. In addition to the usual Mapoperations, it provides the following atomic operations:

Operations in trait ConcurrentMap

What it is	What it does
m putIfAbsent(k, v)	Adds key/value binding k -> m unless k is already defined in m
m remove (k, v)	Removes entry for k if it is currently mapped to v
m replace (k, old, new)	Replaces value associated with key k to new, if it was previously bound to old
m replace (k, v)	Replaces value associated with key k to v, if it was previously bound to some value

ConcurrentMap is a trait in the Scala collections library. Currently, its only implementation is Java's java.util.concurrent.ConcurrentMap, which can be converted automatically into a Scala map using the standard Java/Scala collection conversions, which will be described in Section 24.17.

Mutable bit sets

A mutable bit set is just like an immutable one, except that it can be modified in place. Mutable bit sets are slightly more efficient at updating than immutable ones, because they don't have to copy around Longs that haven't changed. Here is an example:

```
scala> val bits = scala.collection.mutable.BitSet.empty
bits: scala.collection.mutable.BitSet = BitSet()

scala> bits += 1
res64: bits.type = BitSet(1)

scala> bits += 3
res65: bits.type = BitSet(1, 3)

scala> bits
res66: scala.collection.mutable.BitSet = BitSet(1, 3)
```

24.10 ARRAYS

Arrays are a special kind of collection in Scala. One the one hand, Scala arrays correspond one-to-one to Java arrays. That is, a Scala array Array[Int] is represented as a Java int[], anArray[Double] is represented as a Java double[] and an Array[String] is represented as a JavaString[]. But at the same time, Scala arrays offer much more their Java analogues. First, Scala arrays can be generic. That is, you can have an Array[T], where T is a type parameter or abstract type. Second, Scala arrays are compatible with Scala sequences—you can pass anArray[T] where a Seq[T] is required. Finally, Scala arrays also support all sequence operations. Here's an example of this in action:

```
scala> val a1 = Array(1, 2, 3)
a1: Array[Int] = Array(1, 2, 3)

scala> val a2 = a1 map (_ * 3)
a2: Array[Int] = Array(3, 6, 9)
```

```
scala> val a3 = a2 filter (_ % 2 != 0)
a3: Array[Int] = Array(3, 9)

scala> a3.reverse
res1: Array[Int] = Array(9, 3)
```

Given that Scala arrays are represented just like Java arrays, how can these additional features be supported in Scala?

The answer lies in systematic use of implicit conversions. An array cannot pretend to be a sequence, because the data type representation of a native array is not a subtype of Seq. Instead, whenever an array would be used as a Seq, implicitly wrap it in a subclass of Seq. The name of that subclass is scala.collection.mutable.WrappedArray. Here you see it in action:

```
scala> val seq: Seq[Int] = a1
seq: Seq[Int] = WrappedArray(1, 2, 3)

scala> val a4: Array[Int] = seq.toArray
a4: Array[Int] = Array(1, 2, 3)

scala> a1 eq a4
res2: Boolean = true
```

This interaction demonstrates that arrays are compatible with sequences, because there's an implicit conversion from Array to WrappedArray. To go the other way, from a WrappedArray to anArray, you can use the toArray method defined in Traversable. The last interpreter line above shows that wrapping then unwrapping with toArray gives you back the same array you started with.

There is yet another implicit conversion that gets applied to arrays. This conversion simply "adds" all sequence methods to arrays but does not turn the array itself into a sequence. "Adding" means that the array is wrapped in another object of type ArrayOps, which supports all sequence methods. Typically, this ArrayOps object is short-lived; it will usually be inaccessible after the call to the sequence method and its storage can be recycled. Modern VMs often avoid creating this object entirely.

The difference between the two implicit conversions on arrays is demonstrated here:

```
scala> val seq: Seq[Int] = a1
seq: Seq[Int] = WrappedArray(1, 2, 3)

scala> seq.reverse
res2: Seq[Int] = WrappedArray(3, 2, 1)

scala> val ops: collection.mutable.ArrayOps[Int] = a1
ops: scala.collection.mutable.ArrayOps[Int] = [I(1, 2, 3)

scala> ops.reverse
res3: Array[Int] = Array(3, 2, 1)
```

You see that calling reverse on seq, which is a WrappedArray, will give again a WrappedArray. That's logical, because wrapped arrays are Seqs, and calling reverse on any Seq will give again a Seq. On the other hand, calling reverse on the ops value of class ArrayOps will result in an Array, not aSeq.

The ArrayOps example above was quite artificial, intended only to show the difference toWrappedArray. Normally, you'd never define a value of class ArrayOps. You'd just call a Seq method on an array:

```
scala> a1.reverse
res4: Array[Int] = Array(3, 2, 1)
```

The ArrayOps object gets inserted automatically by the implicit conversion. So the line above is equivalent to the following line, where intArrayOps was the conversion that was implicitly inserted previously:

```
scala> intArrayOps(a1).reverse
res5: Array[Int] = Array(3, 2, 1)
```

This raises the question how the compiler picked intArrayOps over the other implicit conversion to WrappedArray in the line above. After all, both conversions map an array to a type that supports a reverse method, which is what the input specified. The answer to that question is that the two implicit conversions are prioritized. The ArrayOps conversion has a higher priority than the WrappedArray conversion. The first is defined in the Predef object whereas the second is defined in a class scala.LowPriorityImplicits, which is a superclass of Predef. Implicits in subclasses and subobjects take precedence over implicits in base classes. So if both conversions are applicable, the one in Predef is chosen. A very similar scheme, which was described in Section 21.7, works for strings.

So now you know how arrays can be compatible with sequences and how they can support all sequence operations. What about genericity? In Java you cannot write a T[] where T is a type parameter. How then is Scala's Array[T] represented? In fact a generic array like Array[T] could be at run time any of Java's eight primitive array types byte[], short[], char[], int[], long[],float[], double[], boolean[], or it could be an array of objects. The only common run-time type encompassing all of these types is AnyRef (or, equivalently java.lang.Object), so that's the type to which the Scala compiler maps Array[T]. At run-time, when an element of an array of typeArray[T] is accessed or updated there is a sequence of type tests that determine the actual array type, followed by the correct array operation on the Java array. These type tests slow down array operations somewhat. You can expect accesses to generic arrays to be three to four times slower than accesses to primitive or object arrays. This means that if you need maximal performance, you should prefer concrete over generic arrays.

Representing the generic array type is not enough, however, there must also be a way tocreate generic arrays. This is an even harder problem, which requires a little bit of help from you. To illustrate the problem, consider the following attempt to write a generic method that creates an array:

```
// This is wrong!
def evenElems[T](xs: Vector[T]): Array[T] = {
  val arr = new Array[T]((xs.length + 1) / 2)
  for (i <- 0 until xs.length by 2)
    arr(i / 2) = xs(i)
  arr
}
```

The evenElems method returns a new array that consists of all elements of the argument vectorxs that are at even positions in the vector. The first line of the body of evenElems creates the result array, which has the same element type as the argument. So depending on the actual type parameter for T, this could be an Array[Int], or an Array[Boolean], or an array of some of the other primitive types in Java, or an array of some reference type. But these types all have different runtime representations, so how is the Scala runtime going to pick the correct one? In fact, it can't do that based on the information it is given, because the actual type that corresponds to the type parameter T is erased at runtime. That's why you will get the following error message if you attempt to compile the code above:

```
error: cannot find class tag for element type T
  val arr = new Array[T]((arr.length + 1) / 2)
            ^
```

What's required here is that you help the compiler by providing a runtime hint of what the actual type parameter of evenElems is. This runtime hint takes the form of a class tag of typescala.reflect.ClassTag. A class tag describes the *erased type* of a given type, which is all the information needed to construct an array of that type.

In many cases the compiler can generate a class tag on its own. Such is the case for a concrete type like Int or String. It's also the case for certain generic types, like List[T], where enough information is known to predict the erased type; in this example the erased type would beList.

For fully generic cases, the usual idiom is to pass the class tag using a context bound, as discussed in Section 21.6. Here is how the above definition could be fixed by using a context bound:

```
// This works
import scala.reflect.ClassTag
def evenElems[T: ClassTag](xs: Vector[T]): Array[T] = {
  val arr = new Array[T]((xs.length + 1) / 2)
  for (i <- 0 until xs.length by 2)
    arr(i / 2) = xs(i)
  arr
}
```

In this new definition, when the Array[T] is created, the compiler looks for a class tag for the type parameter T, that is, it will look for an implicit value of type ClassTag[T]. If such a value is found, the class tag is used to construct the right kind of array. Otherwise, you'll see an error message like the one shown previously.

Here is an interpreter interaction that uses the evenElems method:

```
scala> evenElems(Vector(1, 2, 3, 4, 5))
res6: Array[Int] = Array(1, 3, 5)

scala> evenElems(Vector("this", "is", "a", "test", "run"))
res7: Array[java.lang.String] = Array(this, a, run)
```

In both cases, the Scala compiler automatically constructed a class tag for the element type (first Int, then String) and passed it to the implicit parameter of the evenElems method. The compiler can do that for all concrete types, but not if the argument is itself another type parameter without its class tag. For instance, the following fails:

```
scala> def wrap[U](xs: Vector[U]) = evenElems(xs)
<console>:9: error: No ClassTag available for U
     def wrap[U](xs: Vector[U]) = evenElems(xs)
                                           ^
```

What happened here is that the evenElems demands a class tag for the type parameter U, but none was found. The solution in this case is, of course, to demand another implicit class tag for U. So the following works:

```
scala> def wrap[U: ClassTag](xs: Vector[U]) = evenElems(xs)
wrap: [U](xs: Vector[U])(implicit evidence$1:
     scala.reflect.ClassTag[U])Array[U]
```

This example also shows that the context bound in the definition of U is just a shorthand for an implicit parameter named here evidence$1 of type ClassTag[U].

24.11 STRINGS

Like arrays, strings are not directly sequences, but they can be converted to them, and they also support all sequence operations. Here are some examples of operations you can invoke on strings:

```
scala> val str = "hello"
str: java.lang.String = hello

scala> str.reverse
res6: String = olleh

scala> str.map(_.toUpper)
res7: String = HELLO

scala> str drop 3
res8: String = lo

scala> str slice (1, 4)
res9: String = ell

scala> val s: Seq[Char] = str
s: Seq[Char] = WrappedString(h, e, l, l, o)
```

These operations are supported by two implicit conversions, which were explained in Section 21.7. The first, low-priority conversion maps a String to a WrappedString, which is a subclass ofimmutable.IndexedSeq. This conversion was applied in the last line of the previous example in which a string was converted into a Seq. The other, high-priority conversion maps a string to aStringOps object, which adds all methods on immutable sequences to strings. This conversion was implicitly inserted in the method calls of reverse, map, drop, and slice in the previous example.

24.12 PERFORMANCE CHARACTERISTICS

As the previous explanations have shown, different collection types have different performance characteristics. That's often the primary reason for picking one collection type over another. You can see the performance characteristics of some common operations on collections summarized in two tables, Table 24.12 and Table 24.12.

	head	tail	apply	update	prepend	append	insert
immutable							
List	C	C	L	L	C	L	-
Stream	C	C	L	L	C	L	-
Vector	eC	eC	eC	eC	eC	eC	-
Stack	C	C	L	L	C	L	-
Queue	aC	aC	L	L	L	C	-
Range	C	C	C	-	-	-	-
String	C	L	C	L	L	L	-
mutable							
ArrayBuffer	C	L	C	C	L	aC	L
ListBuffer	C	L	L	L	C	C	L
StringBuilder	C	L	C	C	L	aC	L
MutableList	C	L	L	L	C	C	L
Queue	C	L	L	L	C	C	L
ArraySeq	C	L	C	C	-	-	-
Stack	C	L	L	L	C	L	L
ArrayStack	C	L	C	C	aC	L	L
Array	C	L	C	C	-	-	-

Performance characteristics of sequence types

	lookup	add	remove	min
immutable				
HashSet/HashMap	eC	eC	eC	L
TreeSet/TreeMap	Log	Log	Log	Log
BitSet	C	L	L	eC[5]
ListMap	L	L	L	L
mutable				
HashSet/HashMap	eC	eC	eC	L
WeakHashMap	eC	eC	eC	L
BitSet	C	aC	C	eC^*a*

Performance characteristics of set and map types

The entries in these two tables are explained as follows:

C	The operation takes (fast) constant time.

eC	The operation takes effectively constant time, but this might depend on some assumptions such as the maximum length of a vector or the distribution of hash keys.
aC	The operation takes amortized constant time. Some invocations of the operation might take longer, but if many operations are performed on average only constant time per operation is taken.
Log	The operation takes time proportional to the logarithm of the collection size.
L	The operation is linear, that is it takes time proportional to the collection size.
-	The operation is not supported.

Table 24.12 treats sequence types—both immutable and mutable—with the following operations:

head	Selecting the first element of the sequence.
tail	Producing a new sequence that consists of all elements except the first one.
apply	Indexing.
update	Functional update (with updated) for immutable sequences, side-effecting update (with update) for mutable sequences.
prepend	Adding an element to the front of the sequence. For immutable sequences, this produces a new sequence. For mutable sequences it modifies the existing sequence.
append	Adding an element at the end of the sequence. For immutable sequences, this produces a new sequence. For mutable sequences it modifies the existing sequence.
insert	Inserting an element at an arbitrary position in the sequence. This is only supported directly for mutable sequences.

Table 24.12 treats mutable and immutable sets and maps with the following operations:

lookup	Testing whether an element is contained in set, or selecting a value associated with a key.
add	Adding a new element to a set or a new key/value pair to a map.
remove	Removing an element from a set or a key from a map.
min	The smallest element of the set, or the smallest key of a map.

24.13 EQUALITY

The collection libraries have a uniform approach to equality and hashing. The idea is, first, to divide collections into sets, maps, and sequences. Collections in different categories are always unequal. For instance, Set(1, 2, 3) is unequal to List(1, 2, 3) even though they contain the same elements. On the other hand, within the same category, collections are equal if and only if they have the same elements (for sequences: the same elements in the same order). For example, List(1, 2, 3) == Vector(1, 2, 3), and HashSet(1, 2) == TreeSet(2, 1).

It does not matter for the equality check whether a collection is mutable or immutable. For a mutable collection, equality simply depends on the current elements at the time the equality test is performed. This means that a mutable collection might be equal to different collections at different times, depending what elements are added or removed. This is a potential trap when using a mutable collection as a key in a hash map. For example:

```
scala> import collection.mutable.{HashMap, ArrayBuffer}
import collection.mutable.{HashMap, ArrayBuffer}
```

```
scala> val buf = ArrayBuffer(1, 2, 3)
buf: scala.collection.mutable.ArrayBuffer[Int] =
ArrayBuffer(1, 2, 3)

scala> val map = HashMap(buf -> 3)
map: scala.collection.mutable.HashMap[scala.collection.
mutable.ArrayBuffer[Int],Int] = Map((ArrayBuffer(1, 2, 3),3))

scala> map(buf)
res13: Int = 3

scala> buf(0) += 1

scala> map(buf)
java.util.NoSuchElementException: key not found:
  ArrayBuffer(2, 2, 3)
```

In this example, the selection in the last line will most likely fail because the hash code of the array xs has changed in the second-to-last line. Therefore, the hash-code-based lookup will look at a different place than the one in which xs was stored.

24.14 VIEWS

Collections have quite a few methods that construct new collections. Some examples are map,filter, and ++. We call such methods transformers because they take at least one collection as their receiver object and produce another collection in their result.

Transformers can be implemented in two principal ways: strict and non-strict (or lazy). A strict transformer constructs a new collection with all of its elements. A non-strict, or lazy, transformer constructs only a proxy for the result collection, and its elements are constructed on demand.

As an example of a non-strict transformer, consider the following implementation of a lazy map operation:

```
def lazyMap[T, U](coll: Iterable[T], f: T => U) =
  new Iterable[U] {
    def iterator = coll.iterator map f
  }
```

Note that lazyMap constructs a new Iterable without stepping through all elements of the given collection coll. The given function f is instead applied to the elements of the new collection'siterator as they are demanded.

Scala collections are by default strict in all their transformers, except for Stream, which implements all its transformer methods lazily. However, there is a systematic way to turn every collection into a lazy one and vice versa, which is based on collection views. A view is a special kind of collection that represents some base collection, but implements all of its transformers lazily.

To go from a collection to its view, you can use the view method on the collection. If xs is some collection, then xs.view is the same collection, but with all transformers implemented lazily. To get back from a view to a strict collection, you can use the force method.

As an example, say you have a vector of Ints over which you want to map two functions in succession:

```
scala> val v = Vector(1 to 10: _*)
v: scala.collection.immutable.Vector[Int] =
  Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> v map (_ + 1) map (_ * 2)
res5: scala.collection.immutable.Vector[Int] =
  Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

In the last statement, the expression v map (_ + 1) constructs a new vector that is then transformed into a third vector by the second call to map (_ * 2). In many situations, constructing the intermediate result from the first call to map is a bit wasteful. In the pseudo example, it would be faster to do a single map with the composition of the two functions (_ + 1)and (_ * 2). If you have the two functions available in the same place you can do this by hand. But quite often, successive transformations of a data structure are done in different program modules. Fusing those transformations would then undermine modularity. A more general way to avoid the intermediate results is by turning the vector first into a view, applying all transformations to the view, and finally forcing the view to a vector:

```
scala> (v.view map (_ + 1) map (_ * 2)).force
res12: Seq[Int] = Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

We'll do this sequence of operations again, one by one:

```
scala> val vv = v.view
vv: scala.collection.SeqView[Int,Vector[Int]] =
  SeqView(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

The application v.view gives you a SeqView, *i.e.*, a lazily evaluated Seq. The type SeqView has two type parameters. The first, Int, shows the type of the view's elements. The second, Vector[Int], shows you the type constructor you get back when forcing the view.

Applying the first map to the view gives you:

```
scala> vv map (_ + 1)
res13: scala.collection.SeqView[Int,Seq[_]] = SeqViewM(...)
```

The result of the map is a value that prints SeqViewM(...). This is in essence a wrapper that records the fact that a map with function (_ + 1) needs to be applied on the vector v. It does not apply that map until the view is forced, however. The "M" after SeqView is an indication that the view encapsulates a map operation. Other letters indicate other delayed operations. For instance "S" indicates a delayed slice operation, and "R" indicates a reverse. We'll now apply the second map to the last result.

```
scala> res13 map (_ * 2)
res14: scala.collection.SeqView[Int,Seq[_]] = SeqViewMM(...)
```

You now get a SeqView that contains two map operations, so it prints with a double "M":SeqViewMM(...). Finally, forcing the last result gives:

```
scala> res14.force
res15: Seq[Int] = Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

Both stored functions get applied as part of the execution of the force operation and a new vector is constructed. That way, no intermediate data structure is needed.

One detail to note is that the static type of the final result is a Seq, not a Vector. Tracing the types back we see that as soon as the first delayed map was applied, the result had static typeSeqViewM[Int, Seq[_]]. That is, the "knowledge" that the view was applied to the specific sequence type Vector got lost. The implementation of a view for any particular class requires quite a bit of code, so the Scala collection libraries provide views mostly only for general collection types, not for specific implementations.[6]

There are two reasons why you might want to consider using views. The first is performance. You have seen that by switching a collection to a view the construction of intermediate results can be avoided. These savings can be quite important. As another example, consider the problem of finding the first palindrome in a list of words. A palindrome is a word that reads backwards the same as forwards. Here are the necessary definitions:

```
def isPalindrome(x: String) = x == x.reverse
def findPalindrome(s: Seq[String]) = s find isPalindrome
```

Now, assume you have a very long sequence words and you want to find a palindrome in the first million words of that sequence. Can you re-use the definition of findPalindrome? Of course, you could write:

```
findPalindrome(words take 1000000)
```

This nicely separates the two aspects of taking the first million words of a sequence and finding a palindrome in it. But the downside is that it always constructs an intermediary sequence consisting of one million words, even if the first word of that sequence is already a palindrome. So potentially, 999,999 words are copied into the intermediary result without being inspected at all afterwards. Many programmers would give up here and write their own specialized version of finding palindromes in some given prefix of an argument sequence. But with views, you don't have to. Simply write:

```
findPalindrome(words.view take 1000000)
```

This has the same nice separation of concerns, but instead of a sequence of a million elements it will only construct a single lightweight view object. This way, you do not need to choose between performance and modularity.

The second use case applies to views over mutable sequences. Many transformer functions on such views provide a window into the original sequence that can then be used to update selectively some elements of that sequence. To see this in an example, suppose you have an array arr:

```
scala> val arr = (0 to 9).toArray
arr: Array[Int] = Array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

You can create a subwindow into that array by creating a slice of a view of the array:

```
scala> val subarr = arr.view.slice(3, 6)
```

```
subarr: scala.collection.mutable.IndexedSeqView[
  Int,Array[Int]] = IndexedSeqViewS(...)
```

This gives a view, subarr, which refers to the elements at positions 3 through 5 of the array arr. The view does not copy these elements, it just provides a reference to them. Now, assume you have a method that modifies some elements of a sequence. For instance, the following negatemethod would negate all elements of the sequence of integers it's given:

```
scala> def negate(xs: collection.mutable.Seq[Int]) =
         for (i <- 0 until xs.length) xs(i) = -xs(i)
negate: (xs: scala.collection.mutable.Seq[Int])Unit
```

Assume now you want to negate elements at positions three through five of the array arr. Can you use negate for this? Using a view, this is simple:

```
scala> negate(subarr)

scala> arr
res4: Array[Int] = Array(0, 1, 2, -3, -4, -5, 6, 7, 8, 9)
```

What happened here is that negate changed all elements of subarr, which were a slice of the elements of arr. Again, you see that views help in keeping things modular. The code above nicely separated the question of what index range to apply a method to from the question what method to apply.

After having seen all these nifty uses of views you might wonder why have strict collections at all? One reason is that performance comparisons do not always favor lazy over strict collections. For smaller collection sizes the added overhead of forming and applying closures in views is often greater than the gain from avoiding the intermediary data structures. A possibly more important reason is that evaluation in views can be very confusing if the delayed operations have side effects.

Here's an example that bit a few users of versions of Scala before 2.8. In these versions theRange type was lazy, so it behaved in effect like a view. People were trying to create a number of actors[7] like this:

```
val actors = for (i <- 1 to 10) yield actor { ... }
```

They were surprised that none of the actors were executing afterwards, even though the actormethod should create and start an actor from the code that's enclosed in the braces following it. To explain why nothing happened, remember that the for expression above is equivalent to an application of the map method:

```
val actors = (1 to 10) map (i => actor { ... })
```

Since previously the range produced by (1 to 10) behaved like a view, the result of the map was again a view. That is, no element was computed, and, consequently, no actor was created! Actors would have been created by forcing the range of the whole expression, but it's far from obvious that this is what was required to make the actors do their work.

To avoid surprises like this, the Scala collections gained more regular rules in version 2.8. All collections except streams and views are strict. The only way to go from a strict to a lazy collection is via the view method. The only way to go back is via force. So the actors definition above would behave as expected in Scala 2.8 in that it would create and start ten actors. To get back the surprising previous behavior, you'd have to add an explicit view method call:

```
val actors = for (i <- (1 to 10).view) yield actor { ... }
```

In summary, views are a powerful tool to reconcile concerns of efficiency with concerns of modularity. But in order not to be entangled in aspects of delayed evaluation, you should restrict views to two scenarios. Either you apply views in purely functional code where collection transformations do not have side effects. Or you apply them over mutable collections where all modifications are done explicitly. What's best avoided is a mixture of views and operations that create new collections while also having side effects.

24.15 ITERATORS

An iterator is not a collection, but rather a way to access the elements of a collection one by one. The two basic operations on an iterator it are next and hasNext. A call to it.next() will return the next element of the iterator and advance the state of the iterator. Calling next again on the same iterator will then yield the element one beyond the one returned previously. If there are no more elements to return, a call to next will throw a NoSuchElementException. You can find out whether there are more elements to return using Iterator's hasNext method.

The most straightforward way to "step through" all the elements returned by an iterator is to use a while loop:

```
while (it.hasNext)
  println(it.next())
```

Iterators in Scala also provide analogues of most of the methods that you find in theTraversable, Iterable, and Seq traits. For instance, they provide a foreach method that executes a given procedure on each element returned by an iterator. Using foreach, the loop above could be abbreviated to:

```
it foreach println
```

As always, for expressions can be used as an alternate syntax for expressions involvingforeach, map, filter, and flatMap, so yet another way to print all elements returned by an iterator would be:

```
for (elem <- it) println(elem)
```

There's an important difference between the foreach method on iterators and the same method on traversable collections: When called on an iterator, foreach will leave the iterator at its end when it is done. So calling next again on the same iterator will fail with aNoSuchElementException. By contrast, when called on a collection, foreach leaves the number of elements in the collection unchanged (unless the passed function adds or removes elements, but this is discouraged, because it can easily lead to surprising results).

The other operations that Iterator has in common with Traversable have the same property of leaving the iterator at its end when done. For instance, iterators provide a map method, which returns a new iterator:

```
scala> val it = Iterator("a", "number", "of", "words")
it: Iterator[java.lang.String] = non-empty iterator

scala> it.map(_.length)
res1: Iterator[Int] = non-empty iterator

scala> res1 foreach println
1
6
2
5

scala> it.next()
java.util.NoSuchElementException: next on empty iterator
```

As you can see, after the call to map, the it iterator has advanced to its end.

Another example is the dropWhile method, which can be used to find the first element of an iterator that has a certain property. For instance, to find the first word in the iterator shown previously that has at least two characters, you could write:

```
scala> val it = Iterator("a", "number", "of", "words")
it: Iterator[java.lang.String] = non-empty iterator

scala> it dropWhile (_.length < 2)
res4: Iterator[java.lang.String] = non-empty iterator

scala> it.next()
res5: java.lang.String = number
```

Note again that it has changed by the call to dropWhile: it now points to the second word "number" in the list. In fact, it and the result res4 returned by dropWhile will return exactly the same sequence of elements.

There is only one standard operation, duplicate, which allows you to re-use the same iterator:

```
val (it1, it2) = it.duplicate
```

The call to duplicate gives you two iterators, which each return exactly the same elements as the iterator it. The two iterators work independently; advancing one does not affect the other. By contrast the original iterator, it, is advanced to its end by duplicate and is thus rendered unusable.

In summary, iterators behave like collections if you never access an iterator again after invoking a method on it. The Scala collection libraries make this explicit with an abstraction called TraversableOnce, which is a common supertrait of Traversable and Iterator. As the name implies, TraversableOnce objects can be traversed using foreach, but the state of that object after the traversal is not specified. If the TraversableOnce object is in fact an Iterator, it will be at its end after the traversal, but if it is a Traversable, it will still exist as before. A common use case of TraversableOnce is as an argument type for methods that can take either an iterator or traversable as argument. An example is the appending method ++ in trait Traversable. It takes aTraversableOnce parameter, so you can append elements coming from either an iterator or a traversable collection.

All operations on iterators are summarized in Table 24.12.

Operations in trait Iterator

What it is	**What it does**
Abstract methods:	
it.next()	Returns the next element in the iterator and advances pastit.
it.hasNext	Returns true if it can return another element.
Variations:	
it.buffered	A buffered iterator returning all elements of it.
it grouped size	An iterator that yields the elements returned by it in fixed-sized sequence "chunks."
xs sliding size	An iterator that yields the elements returned by it in sequences representing a sliding fixed-sized window.
Copying:	
it copyToBuffer buf	Copies all elements returned by it to buffer buf.
it copyToArray(arr, s, l)	Copies at most l elements returned by it to array arrstarting at index s. The last two arguments are optional.
Duplication:	
it.duplicate	A pair of iterators that each independently return all elements of it.
Additions:	
it ++ jt	An iterator returning all elements returned by iterator it, followed by all elements returned by iterator jt.
it padTo (len, x)	The iterator that returns all elements of it followed by copies of x until length len elements are returned overall.
Maps:	
it map f	The iterator obtained from applying the function f to every element returned from it.
it flatMap f	The iterator obtained from applying the iterator-valued function f to every element in it and appending the results.
it collect f	The iterator obtained from applying the partial function f to every element in it for which it is defined and collecting the results.
Conversions:	
it.toArray	Collects the elements returned by it in an array.
it.toList	Collects the elements returned by it in a list.
it.toIterable	Collects the elements returned by it in an iterable.

it.toSeq	Collects the elements returned by it in a sequence.
it.toIndexedSeq	Collects the elements returned by it in an indexed sequence.
it.toStream	Collects the elements returned by it in a stream.
it.toSet	Collects the elements returned by it in a set.
it.toMap	Collects the key/value pairs returned by it in a map.
Size info:	
it.isEmpty	Tests whether the iterator is empty (opposite of hasNext).
it.nonEmpty	Tests whether the collection contains elements (alias ofhasNext).
it.size	The number of elements returned by it. Note: it will be at its end after this operation!
it.length	Same as it.size.
it.hasDefiniteSize	Returns true if it is known to return finitely many elements (by default the same as isEmpty).
Element retrieval index search:	
it find p	An option containing the first element returned by it that satisfies p, or None if no element qualifies. Note: The iterator advances to just after the element, or, if none is found, to the end.
it indexOf x	The index of the first element returned by it that equals x. Note: The iterator advances past the position of this element.
it indexWhere p	The index of the first element returned by it that satisfies p. Note: The iterator advances past the position of this element.
Subiterators:	
it take n	An iterator returning of the first n elements of it. Note: itwill advance to the position after the n'th element, or to its end, if it contains less than n elements.
it drop n	The iterator that starts with the (n + 1)'th element of it. Note: it will advance to the same position.
it slice (m, n)	The iterator that returns a slice of the elements returned from it, starting with the m'th element and ending before then'th element.
it takeWhile p	An iterator returning elements from it as long as conditionp is true.
it dropWhile p	An iterator skipping elements from it as long as condition pis true, and returning the remainder.
it filter p	An iterator returning all elements from it that satisfy the condition p.
it withFilter p	Same as it filter p. Needed so that iterators can be used infor expressions.
it filterNot p	An iterator returning all elements from it that do not satisfy the condition p.
Subdivisions:	
it partition p	Splits it into a pair of two iterators; one returning all elements from it that satisfy the predicate p, the other returning all elements from it that do not.
Element conditions:	
it forall p	A boolean indicating whether the predicate p holds for all elements returned by it.
it exists p	A boolean indicating whether the predicate p holds for some element in it.
it count p	The number of elements in it that satisfy the predicate p.

Folds:	
(z /: it)(op)	Applies binary operation op between successive elements returned by it, going left to right, starting with z.
(it :\ z)(op)	Applies binary operation op between successive elements returned by it, going right to left, starting with z.
it.foldLeft(z)(op)	Same as (z /: it)(op).
it.foldRight(z)(op)	Same as (it :\ z)(op).
it reduceLeft op	Applies binary operation op between successive elements returned by non-empty iterator it, going left to right.
it reduceRight op	Applies binary operation op between successive elements returned by non-empty iterator it, going right to left.
Specific folds:	
it.sum	The sum of the numeric element values returned by iteratorit.
it.product	The product of the numeric element values returned by iterator it.
it.min	The minimum of the ordered element values returned by iterator it.
it.max	The maximum of the ordered element values returned by iterator it.
Zippers:	
it zip jt	An iterator of pairs of corresponding elements returned from iterators it and jt.
it zipAll (jt, x, y)	An iterator of pairs of corresponding elements returned from iterators it and jt, where the shorter iterator is extended to match the longer one by appending elements xor y.
it.zipWithIndex	An iterator of pairs of elements returned from it with their indicies.
Update:	
it patch (i, jt, r)	The iterator resulting from it by replacing r elements starting with i by the patch iterator jt.
Comparison:	
it sameElements jt	A test whether iterators it and jt return the same elements in the same order. Note: At least one of it and jt will be at its end after this operation.
Strings:	
it addString (b, start, sep, end)	Adds a string to StringBuilder b that shows all elements returned by it between separators sep enclosed in stringsstart and end. start,sep, and end are all optional.
it mkString (start, sep, end)	Converts the iterator to a string that shows all elements returned by it between separators sep enclosed in stringsstart and end. start,sep, and end are all optional.

Buffered iterators

Sometimes you want an iterator that can "look ahead" so that you can inspect the next element to be returned without advancing past that element. Consider, for instance, the task to skip leading empty strings from an iterator that returns a sequence of strings. You might be tempted to write something like the following method:

```
// This won't work
def skipEmptyWordsNOT(it: Iterator[String]) = {
  while (it.next().isEmpty) {}
}
```

But looking at this code more closely, it's clear that this is wrong: the code will indeed skip leading empty strings, but it will also advance it past the first non-empty string!

The solution to this problem is to use a buffered iterator, an instance of trait BufferedIterator.BufferedIterator is a subtrait of Iterator, which provides one extra method, head. Calling head on a buffered iterator will return its first element, but will not advance the iterator. Using a buffered iterator, skipping empty words can be written like this:

```
def skipEmptyWords(it: BufferedIterator[String]) =
  while (it.head.isEmpty) { it.next() }
```

Every iterator can be converted to a buffered iterator by calling its buffered method. Here's an example:

```
scala> val it = Iterator(1, 2, 3, 4)
it: Iterator[Int] = non-empty iterator

scala> val bit = it.buffered
bit: java.lang.Object with scala.collection.
  BufferedIterator[Int] = non-empty iterator

scala> bit.head
res10: Int = 1

scala> bit.next()
res11: Int = 1

scala> bit.next()
res11: Int = 2
```

Note that calling head on the buffered iterator, bit, did not advance it. Therefore, the subsequent call, bit.next(), returned again the same value as bit.head.

24.16 CREATING COLLECTIONS FROM SCRATCH

You have already seen syntax like List(1, 2, 3), which creates a list of three integers, andMap('A' -> 1, 'C' -> 2), which creates a map with two bindings. This is actually a universal feature of Scala collections. You can take any collection name and follow it by a list of elements in parentheses. The result will be a new collection with the given elements. Here are some more examples:

```
Traversable()             // An empty traversable object
List()                    // The empty list
List(1.0, 2.0)            // A list with elements 1.0, 2.0
Vector(1.0, 2.0)          // A vector with elements 1.0, 2.0
Iterator(1, 2, 3)         // An iterator returning three integers.
Set(dog, cat, bird)       // A set of three animals
HashSet(dog, cat, bird)   // A hash set of the same animals
Map('a' -> 7, 'b' -> 0)   // A map from characters to integers
```

"Under the covers" each of the above lines is a call to the apply method of some object. For instance, the third line above expands to:

```
List.apply(1.0, 2.0)
```

So this is a call to the apply method of the companion object of the List class. That method takes an arbitrary number of arguments and constructs a list from them. Every collection class in the Scala library has a companion object with such an apply method. It does not matter whether the collection class represents a concrete implementation, like List, Stream, orVector, or whether it is an trait such as Seq, Set, or Traversable. In the latter case, calling applywill produce some default implementation of the trait. Here are some examples:

```
scala> List(1, 2, 3)
res17: List[Int] = List(1, 2, 3)

scala> Traversable(1, 2, 3)
res18: Traversable[Int] = List(1, 2, 3)

scala> mutable.Traversable(1, 2, 3)
res19: scala.collection.mutable.Traversable[Int] =
  ArrayBuffer(1, 2, 3)
```

Besides apply, every collection companion object also defines a member empty, which returns an empty collection. So instead of List() you could write List.empty, instead of Map(), Map.empty, and so on.

Descendants of Seq traits also provide other factory operations in their companion objects. These are summarized in Table 24.13. In short, there's:

- concat, which concatenates an arbitrary number of traversables together,
- fill and tabulate, which generate single or multi-dimensional sequences of given dimensions initialized by some expression or tabulating function,
- range, which generates integer sequences with some constant step length, and
- iterate, which generates the sequence resulting from repeated application of a function to a start element.

Factory methods for sequences

What it is	**What it does**
S.empty	The empty sequence
S(x, y, z)	A sequence consisting of elements x, y, and z
S.concat(xs, ys, zs)	The sequence obtained by concatenating the elements of xs,ys, and zs
S.fill(n)(e)	A sequence of length n where each element is computed by expression e
S.fill(m, n)(e)	A sequence of sequences of dimension m x n where each element is computed by expression e (exists also in higher dimensions)
S.tabulate(n)(f)	A sequence of length n where the element at each index *i* is computed by f(*i*)
S.tabulate(m, n)(f)	A sequence of sequences of dimension m x n where the element at each index (*i*, *j*) is computed by f(*i*, *j*) (exists also in higher dimensions)

S.range(start, end)	The sequence of integers start ... end - 1
S.range(start, end, step)	The sequence of integers starting with start and progressing by step increments up to, and excluding, the endvalue
S.iterate(x, n)(f)	The sequence of length n with elements x, f(x), f(f(x)), ...

24.17 CONVERSIONS BETWEEN JAVA AND SCALA COLLECTIONS

Like Scala, Java has a rich collections library. There are many similarities between the two. For instance, both libraries know iterators, iterables, sets, maps, and sequences. But there are also important differences. In particular, the Scala libraries put much more emphasis on immutable collections, and provide many more operations that transform a collection into a new one.

Sometimes you might need to convert from one collection framework to the other. For instance, you might want to access to an existing Java collection, as if it were a Scala collection. Or you might want to pass one of Scala's collections to a Java method that expects the Java counterpart. It is quite easy to do this, because Scala offers implicit conversions between all the major collection types in the JavaConversions object. In particular, you will find bidirectional conversions between the following types:

```
Iterator         \null    java.util.Iterator
Iterator         \null    java.util.Enumeration
Iterable         \null    java.lang.Iterable
Iterable         \null    java.util.Collection
mutable.Buffer   \null    java.util.List
mutable.Set      \null    java.util.Set
mutable.Map      \null    java.util.Map
```

To enable these conversions, simply import like this:

```
scala> import collection.JavaConversions._
import collection.JavaConversions._
```

You have now automatic conversions between Scala collections and their corresponding Java collections.

```
scala> import collection.mutable._
import collection.mutable._

scala> val jul: java.util.List[Int] = ArrayBuffer(1, 2, 3)
jul: java.util.List[Int] = [1, 2, 3]

scala> val buf: Seq[Int] = jul
buf: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3)

scala> val m: java.util.Map[String, Int] =
          HashMap("abc" -> 1, "hello" -> 2)
m: java.util.Map[String,Int] = {hello=2, abc=1}
```

Internally, these conversion work by setting up a "wrapper" object that forwards all operations to the underlying collection object. So collections are never copied when converting between Java and Scala. An interesting property is that if you do a round-trip conversion from, say, a Java type to its corresponding Scala type, and back to the same Java type, you end up with the identical collection object you started with.

Some other common Scala collections exist that can also be converted to Java types, but for which no corresponding conversion exists in the other direction. These are:

```
Seq          \null   java.util.List
mutable.Seq  \null   java.util.List
Set          \null   java.util.Set
Map          \null   java.util.Map
```

Because Java does not distinguish between mutable and immutable collections in their type, a conversion from, say, collection.immutable.List will yield a java.util.List, on which all attempted mutation operations will throw an UnsupportedOperationException. Here's an example:

```
scala> val jul: java.util.List[Int] = List(1, 2, 3)
jul: java.util.List[Int] = [1, 2, 3]

scala> jul.add(7)
java.lang.UnsupportedOperationException
        at java.util.AbstractList.add(AbstractList.java:131)
```

24.18 CONCLUSION

You've now seen how to use Scala's collection in great detail. Scala's collections take the approach of giving you powerful building blocks rather than a number of ad hoc utility methods. Putting together two or three such building blocks allows you to express an enormous number of useful computations. This style of library is especially effective due to Scala having a light syntax for function literals, and due to it providing many collection types that are persistent and immutable.

This chapter has shown collections from the point of view of a programmer using the collection library. The next chapter will show you how the collections are built and how you can add your own collection types.

Footnotes for Chapter 24:

[1] Partial functions were described in Section 15.7.

[2] Hash tries are described in Section 24.8.

[3] "Trie" comes from the word "re*trie*val" and is pronounced *tree* or *try*.

[4] The "buf.type" that appears in the interpreter responses in this and several other examples in this section is a *singleton type*. As will be explained in Section 29.6, buf.type means the variable holds exactly the object referred to by buf.

[5] Assuming bits are densely packed.

[6] An exception to this is arrays: applying delayed operations on arrays will again give results with static type Array.

[7] The Scala actors library has been deprecated, but this historical example is still relevant.