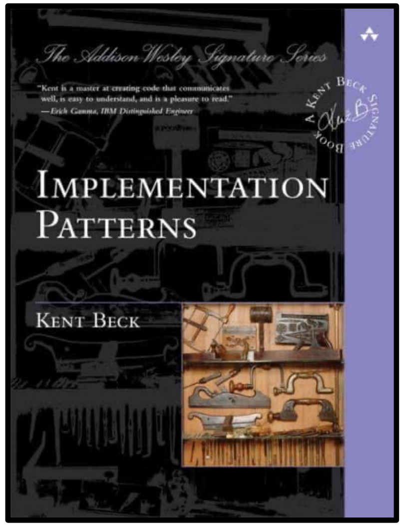
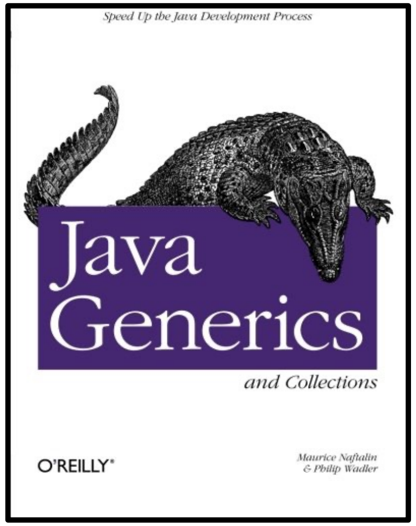
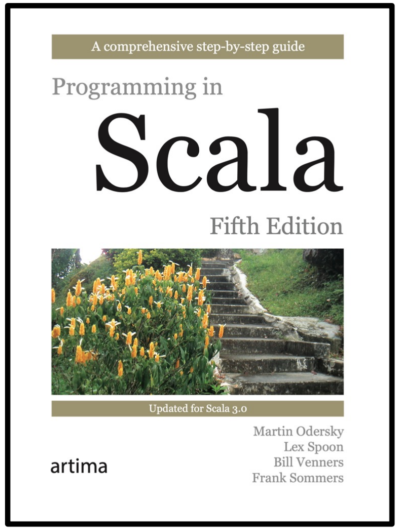
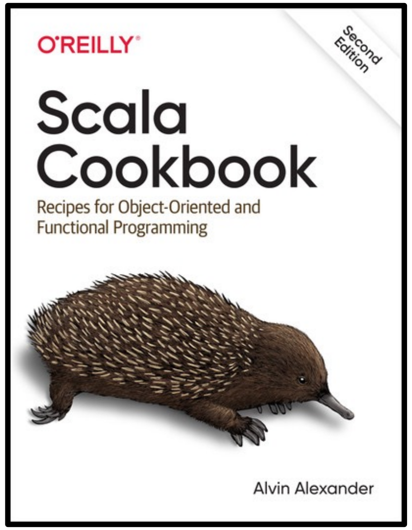
'go-to' general-purpose sequential collections from Java To Scala

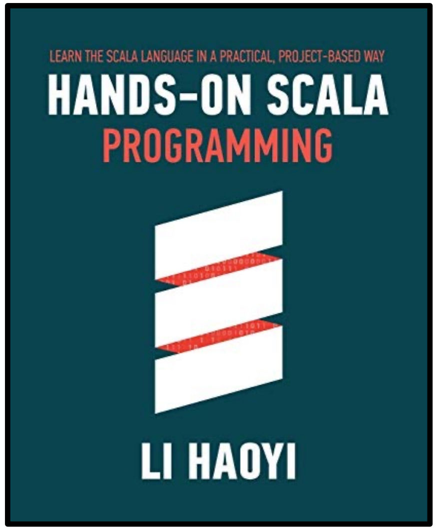
based on excerpts from the following











slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

@philip_schwarz

The simple idea of this slide deck is that it collects in a single place quite a bit of information that can be used to gain a basic understanding of some key differences between the **'goto' sequential collections** of **Java** and **Scala**.

Hopefully the authors of the books referenced in this deck will forgive me for sharing excerpts from their books, and just as hopefully, such sharing will promote the books, which have much more to offer than what I have highlighted for the purposes of this deck.

Chapter 9. Collections

I must say that I didn't expect this chapter to amount to much. When I started writing it, I thought I would end up with an API document—types and operations.

The basic idea is simple: **a collection distinguishes between objects in the collection and those not in the collection**. What more was there to say?

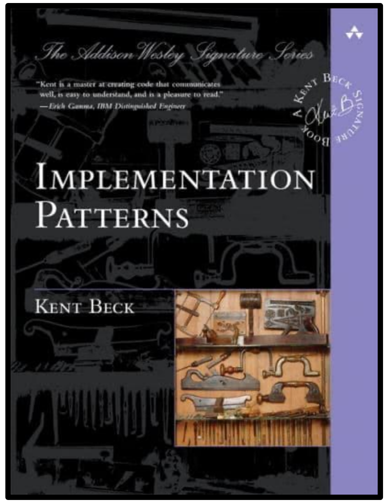
What I discovered is that **collections are a far richer topic than I ever suspected, both in their structure and the possibilities they offer for communicating intent.**

The concept of collections blends several different metaphors. The metaphor you emphasize changes how you use collections.

Each of the collection interfaces communicates a different variation on the theme of a sack of objects.

Each of the implementations also communicates variations, mostly with regard to performance. The result is that mastering collections is a big part of learning to communicate well with code.

…



Kent Beck
@KentBeck

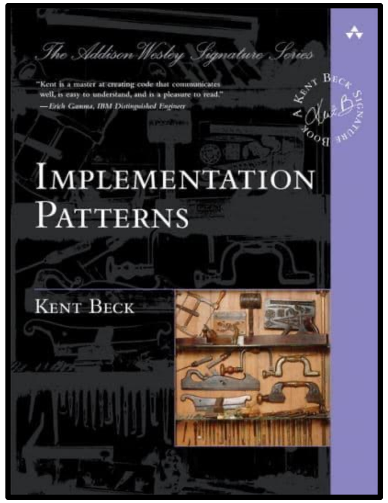
Sidebar: Performance

Most programmers don't have to worry about the performance of small-scale operations most of the time. This is a refreshing change from the old days, when performance tuning was daily business. However, computing resources are not infinite. When experience has shown that performance needs to be better and measurement has shown where the bottlenecks are, **it is important to express performance-related decisions clearly. Many times, better performance results in less of some other quality in the code, like readability or flexibility. It is important to pay as little as possible for the needed performance.**

Coding for performance can violate the principle of local consequences. A small change to one part of a program can degrade performance in another part. If a method works efficiently only if the collection it is passed can test for membership quickly, then an innocent substitution of ArrayList for HashSet elsewhere in the program can make the method intolerably slow. Distant consequences are another argument for coding carefully when coding for performance.

Performance is connected with collections because most collections can grow without limit. The data structure holding the characters I am typing right now needs to be able to hold millions of characters. I would like inserting the millionth character to be just as fast as inserting the first.

My overall strategy for performance coding with collections is to use the simplest possible implementation at first and pick a more specialized collection class when it becomes necessary. When I make performance-related decisions I try to localize them as much as possible even if that requires some changes to the design. Then, when the performance is good enough again, I stop tuning.



Kent Beck
@KentBeck

Issues

Collections are used to express several orthogonal concepts in programs. In principle, you should express yourself as precisely as possible. With collections, this means using the most general possible interface as a declaration and the most specific implementation class. However, this is not an absolute rule.

…

The first concept expressed by collections is their size. Arrays (which are primitive collections) have a fixed size, set when the array is created. Most collections can change size after they are created.

A second concept expressed through collections is whether or not the order of elements is important.

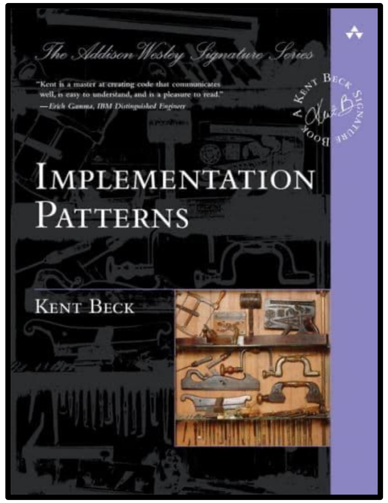
…

Another issue to be expressed by collections is the uniqueness of elements.

…

How are the elements accessed? Sometimes it is enough to iterate over the elements, doing some calculation with them one at a time. At other times it is important to be able to store and retrieve elements with a key.

Finally, performance considerations are communicated through choice of collection. If a linear search is fast enough, a generic Collection is good enough. If the collection grows too large it will be important to be able to test for or access elements by a key, suggesting a Set or Map. Time and space can both be optimized through the judicious selection of collections.



Kent Beck

@KentBeck

Interfaces

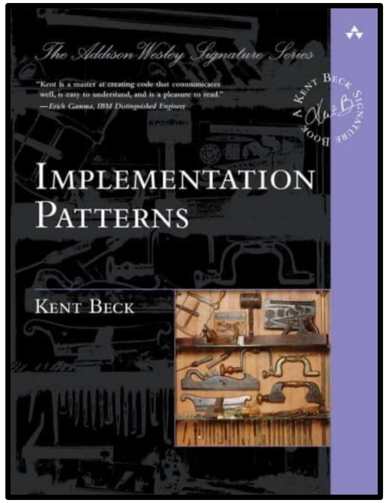
Readers of collection-based code are looking for answers to different questions when they look at the interfaces you have declared for your variables and the implementations you chose for those variables.

The interface declaration tells the reader about the collection: whether the collection is in a particular order, whether there are duplicate elements, and whether there is any way to look up elements by key or only through iteration.

The interfaces described below are:

- Array— Arrays are the simplest and least flexible collection: fixed size, simple accessing syntax, and fast.
- Iterable— The basic collection interface, allowing a collection to be used for iteration but nothing else.
- Collection— Offers adding, removing, and testing for elements.
- List— A collection whose elements are ordered and can be accessed by their location in the collection (i.e., “give me the third element”).

…



Kent Beck
@KentBeck

Array

Arrays are the simplest interface for collections. Unfortunately, they don't have the same protocol as other collections, so it's harder to change from an array to a collection than from one kind of collection to another.

Unlike most collections, the size of an array is fixed when it is created. Arrays are also different as they are built into the language, not provided by a library.

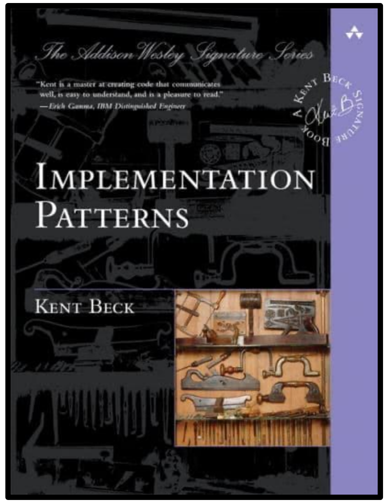
Arrays are more efficient in time and space than other collections for simple operations. The timing tests I did to accompany writing this suggest that array access (i.e. elements[i]) is more than ten times faster than the equivalent ArrayList operation (elements.get(i)). (As these numbers vary substantially in different operating environments, if you care about the performance difference you should time the operations yourself.)

The flexibility of the other collection classes makes them more valuable in most cases, but arrays are a handy trick to be able to pull out when you need more performance in a small part of an application.

Iterable

Declaring a variable Iterable only says that it contains multiple values. Iterable is the basis for the loop construct in Java 5. Any object declared as Iterable can be used in a for loop. This is implemented by quietly calling the method iterator().

One of the issues to be communicated when using collections is whether clients are expected to modify them. Unfortunately, Iterable and its helper, Iterator, provide no way to state declaratively that a collection shouldn't be modified. Once you have an Iterator, you can invoke its remove() method, which deletes an element from the underlying Iterable. While your Iterables are safe from having elements added, they can have elements removed without the object that owns the collection being notified.



Kent Beck
@KentBeck

As described in …, there are a few ways to ensure that a collection is not modified: wrapping it in a unmodifiable collection, creating a custom iterator that throws an exception when a client tries to modify the collection, or returning a safe copy.

Iterable is simple. It doesn't even allow you to measure the size of instances; all you can do is iterate over the elements. Sub-interfaces of Iterable provide more useful behavior.

Collection

Collection inherits from Iterable, but it adds methods to add, remove, search for and count elements.

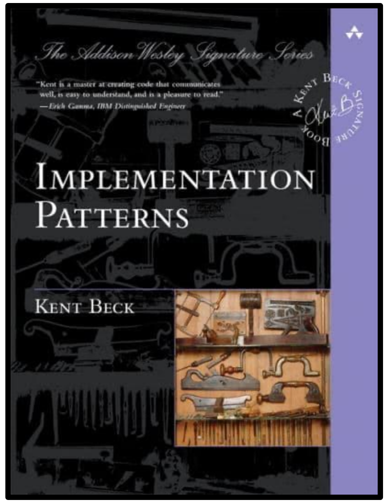
Declaring a variable or method as a Collection leaves many options for an implementation class.

By leaving the declaration as vaguely specified as possible, you retain the freedom to change implementation classes later without having the change ripple through the code.

Collections are a bit like the mathematical notion of sets, except that the operations performing the equivalent of union, intersection, and difference (addAll(), retainAll(), and removeAll()) modify the receiver instead of returning newly allocated collections.

List

To Collection, List adds the idea that elements are in a stable order. An element can be retrieved by providing its index to the collection. A stable sequence is important when the elements of a collection interact with each other. For example, a queue of messages that should be processed in their arrival order should be stored in a list.



Kent Beck
@KentBeck

Implementations

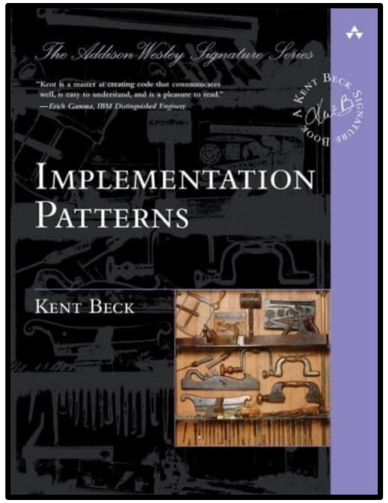
Choosing implementation classes for collections is primarily a matter of performance. As with all performance issues, it is best to pick a simple implementation to begin with and then tune based on experience.

…

In this section, each interface introduces alternative implementations. Because performance considerations dominate the choice of implementation class, each set of alternatives is accompanied by performance measurements for important operations. Appendix, "Performance Measurement," provides the source code for the tool I used to gather this data.

By far the majority of collections are implemented by ArrayList, with HashSet a distant second (~3400 references to ArrayList in Eclipse+JDK versus ~800 references to HashSet). The quick-and-dirty solution is to choose whichever of these classes suits your needs. However, for those times when experience shows that performance matters, the remainder of this section presents the details of the alternative implementations.

A final factor in choosing a collection implementation class is the size of the collections involved. The data presented below shows the performance of collections sized one to one hundred thousand. If your collections only contain one or two elements, your choice of implementation class may be different than if you expect them to scale to millions of elements. In any case, the gains available from switching implementation classes are often limited, and you'll need to look for larger-scale algorithmic changes if you want to further improve performance.



Kent Beck
@KentBeck

Collection

The default class to use when implementing a Collection is ArrayList.

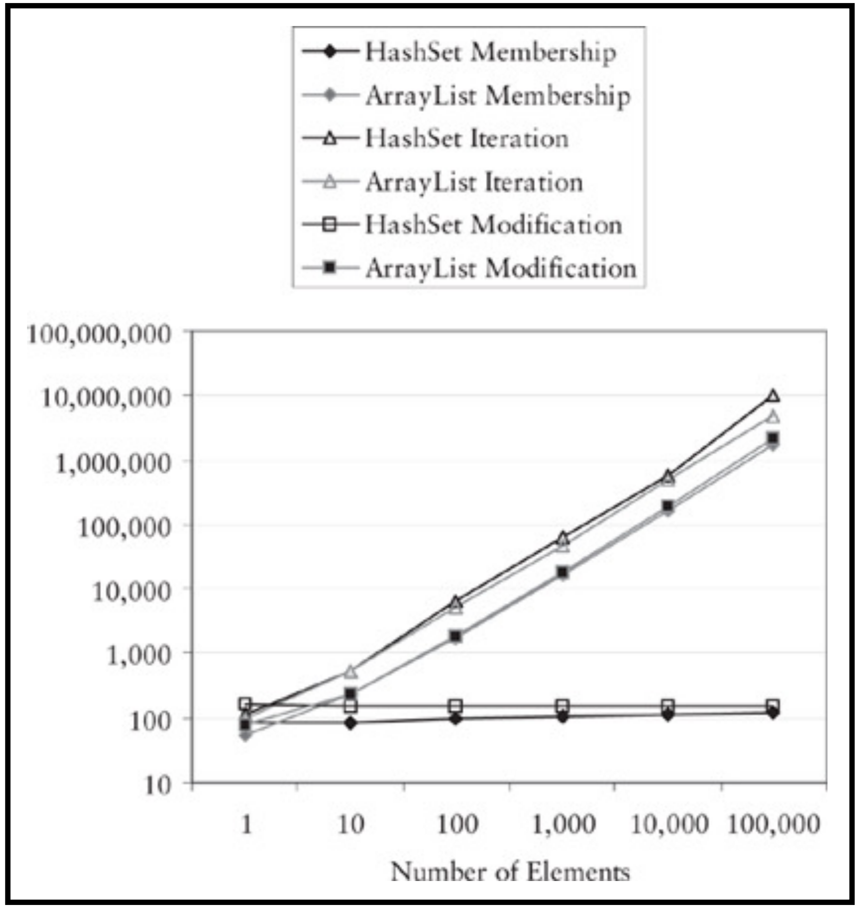
The potential performance problem with ArrayList is that contains(Object) and other operations that rely on it like remove(Object) take time proportional to the size of the collection.

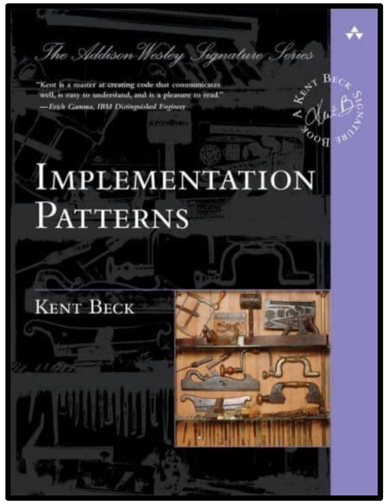
If a performance profile shows one of these methods to be a bottleneck, consider replacing your ArrayList with a HashSet.

Before doing so, make sure that your algorithm is insensitive to discarding duplicate elements.

When you have data that is already guaranteed to contain no duplicates, the switch won't make a difference.

Figure 9.2 compares the performance of ArrayList and HashSet. (See Appendix A for the details of how I collected this information.)



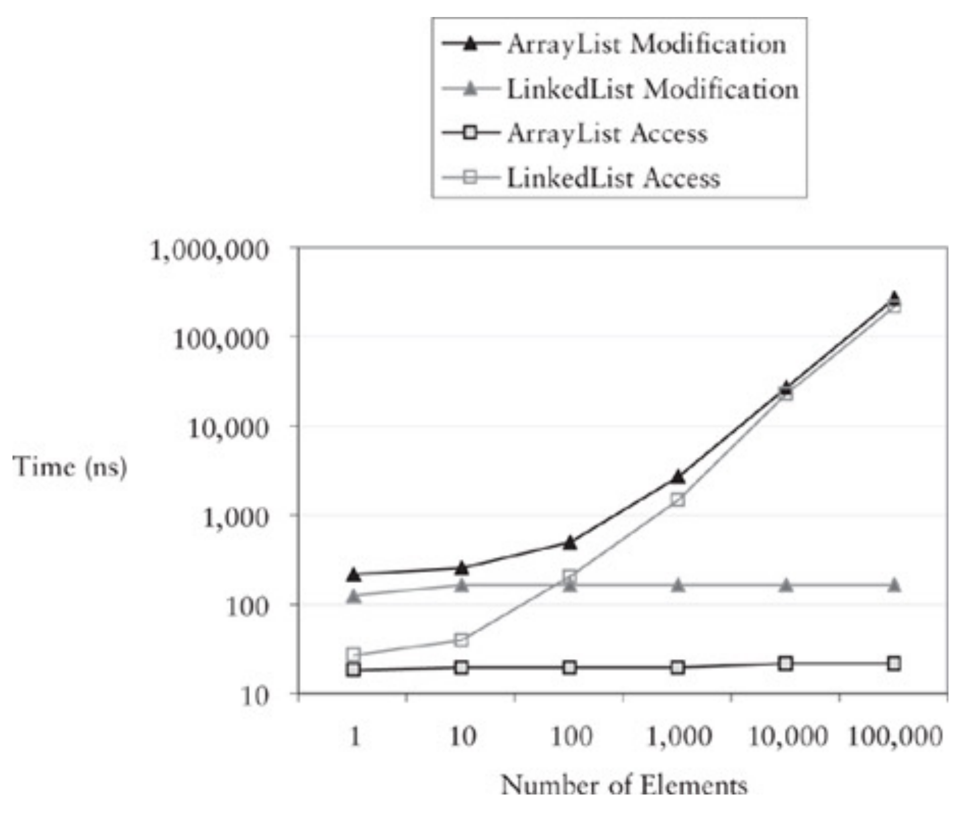


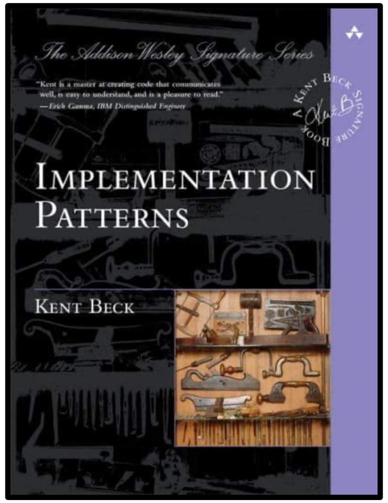
Kent Beck
@KentBeck

List

To the Collection protocol, List adds the idea that the elements are in a stable order. The two implementations of List in common use are ArrayList and LinkedList.

The performance profiles of these two implementations are mirror images. ArrayList is fast at accessing elements and slow at adding and removing elements, while LinkedList is slow at accessing elements and fast at adding and removing elements (see Figure 9.3). If you see a profile dominated by calls to add() or remove(), consider switching an ArrayList to a LinkedList.



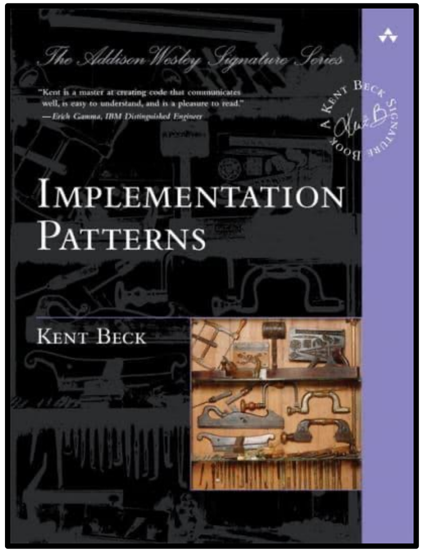


Kent Beck
@KentBeck

Unmodifiable Collections

As mentioned in the discussion of Iterable above, even the most basic collection interfaces allow collections to be modified. If you are passing a collection to untrusted code, you can ensure that it won’t be modified by having Collections wrap it in an implementation that throws a runtime exception if clients try to modify it. There are variants that work with Collection, List, Set, and Map.

```
@Test(expected=UnsupportedOperationException.class)
public void unmodifiableCollectionsThrowExceptions() {
  List<String> l= new ArrayList<String>();
  l.add("a");
  Collection<String> unmodifiable= Collections.unmodifiableCollection(l);
  Iterator<String> all= unmodifiable.iterator();
  all.next();
  all.remove();
}
```



Kent Beck
@KentBeck

2.5 Arrays

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

In Java, array subtyping is *covariant*, meaning that type S[] is considered to be a subtype of T[] whenever S is a subtype of T. Consider the following code fragment, which allocates an array of integers, assigns it to an array of numbers, and then attempts to assign a double into the array:

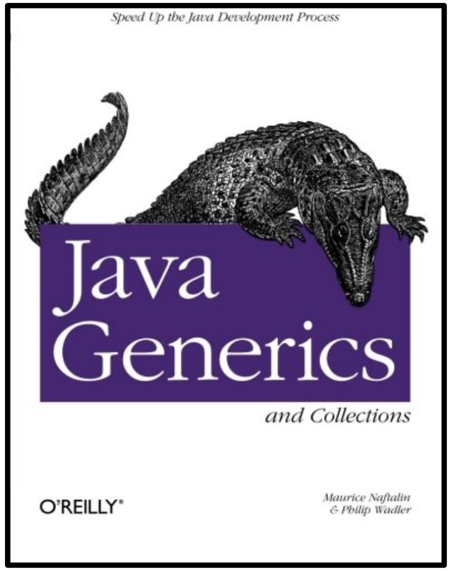
```
Integer[] ints = new Integer[] {1,2,3};
Number[] nums = ints;
nums[2] = 3.14; // array store exception
assert Arrays.toString(ints).equals("[1, 2, 3.14]"); // uh oh!
```

Something is wrong with this program, since it puts a double into an array of integers! Where is the problem? Since Integer[] is considered a subtype of Number[], according to the Substitution Principle the assignment on the second line must be legal. Instead, the problem is caught on the third line, and it is caught at run time. When an array is allocated (as on the first line), it is tagged with its reified type (a run-time representation of its component type, in this case, Integer), and every time an array is assigned into (as on the third line), an array store exception is raised if the reified type is not compatible with the assigned value (in this case, a double cannot be stored into an array of Integer).

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

```
List<Integer> ints = Arrays.asList(1,2,3);
List<Number> nums = ints; // compile-time error
nums.set(2, 3.14);
assert ints.toString().equals("[1, 2, 3.14]"); // uh oh!
```

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



Maurice Naftalin
@mauricenaftalin

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

```
List<Integer> ints = Arrays.asList(1,2,3);
List<? extends Number> nums = ints;
nums.set(2, 3.14); // compile-time error
assert ints.toString().equals("[1, 2, 3.14]"); // uh oh!
```

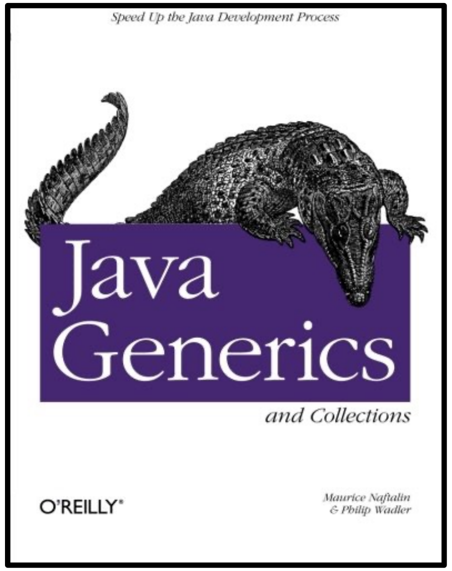
```
Integer[] ints = new Integer[] {1,2,3};
Number[] nums = ints;
nums[2] = 3.14; // array store exception
assert Arrays.toString(ints).equals("[1, 2, 3.14]");
```

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

Wildcards also introduce *contravariant* subtyping for generics, in that type List<S> is considered to be a *subtype* of List<? super T> when S is a *supertype* of T (as opposed to a subtype). Arrays do not support contravariant subtyping. …

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

Apart from the fact that errors are caught earlier, there are many other reasons to prefer collection classes to arrays. Collections are far more flexible than arrays. The only operations supported on arrays are to get or set a component, and the representation is fixed. Collections support many additional operations, including testing for containment, adding and removing elements, comparing or combining two collections, and extracting a sublist of a list. Collections may be either lists (where order is significant and elements may be repeated) or sets (where order is not significant and elements may not be repeated), and a number of representations are available, including arrays, linked lists, trees, and hash tables.



Maurice Naftalin
@mauricenaftalin

Finally, a comparison of the convenience classes Collections and Arrays shows that collections offer many operations not provided by arrays, including operations to rotate or shuffle a list, to find the maximum of a collection, and to make a collection unmodifiable or synchronized.

Nonetheless, there are a few cases where arrays are preferred over collections. Arrays of primitive type are much more efficient since they don't involve boxing; and assignments into such an array need not check for an array store exception, because arrays of primitive type do not have subtypes. And despite the check for array store exceptions, even arrays of reference type may be more efficient than collection classes with the current generation of compilers, so you may want to use arrays in crucial inner loops. As always, you should measure performance to justify such a design, especially since future compilers may optimize collection classes specially. Finally, in some cases arrays may be preferable for reasons of compatibility.

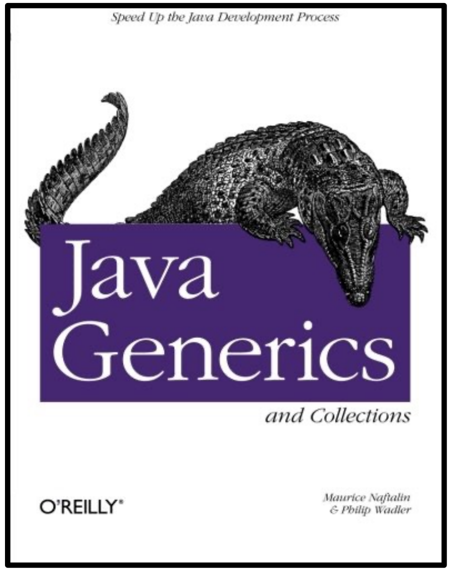
To summarize, it is better to detect errors at compile time rather than run time, but Java arrays are forced to detect certain errors at run time by the decision to make array subtyping covariant. Was this a good decision? Before the advent of generics, it was absolutely necessary. For instance, look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

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

Thanks to covariance, these methods can be used to sort or fill arrays of any reference type. Without covariance and without generics, there would be no way to declare methods that apply for all types. However, now that we have generics, covariant arrays are no longer necessary. Now we can give the methods the following signatures, directly stating that they work for all types:

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

In some sense, covariant arrays are an artifact of the lack of generics in earlier versions of Java. Once you have generics, covariant arrays are probably the wrong design choice, and the only reason for retaining them is backward compatibility…. For many purposes, it may be sensible to consider arrays a deprecated type. …



Maurice Naftalin
@mauricenaftalin

6.5 Array Creation

…

Inability to create generic arrays is one of the most serious restrictions in Java. Because it is so annoying, it is worth reiterating the reason it occurs: generic arrays are problematic because generics are implemented via erasure, but erasure is beneficial because it eases evolution.

The best workaround is to use ArrayList or some other class from the Collections Framework in preference to an array. We discussed the tradeoffs between collection classes and arrays in Arrays, and we noted that in many cases collections are preferable to arrays: because they catch more errors at compile time, because they provide more operations, and because they offer more flexibility in representation. By far, the best solution to the problems offered by arrays is to "just say no": use collections in preference to arrays.

Sometimes this won't work, because you need an array for reasons of compatibility or efficiency. Examples of this occur in the Collections Framework: for compatibility, the method toArray converts a collection to an array; and, for efficiency, the class ArrayList is implemented by storing the list elements in an array.

…

6.7 How to Define ArrayList

We have argued elsewhere that it is usually preferable to use a list than to use an array. There are a few places where this is not appropriate. In rare circumstances, you will need to use an array for reasons of efficiency or compatibility. Also, of course, you need to use arrays to implement ArrayList itself. …

…

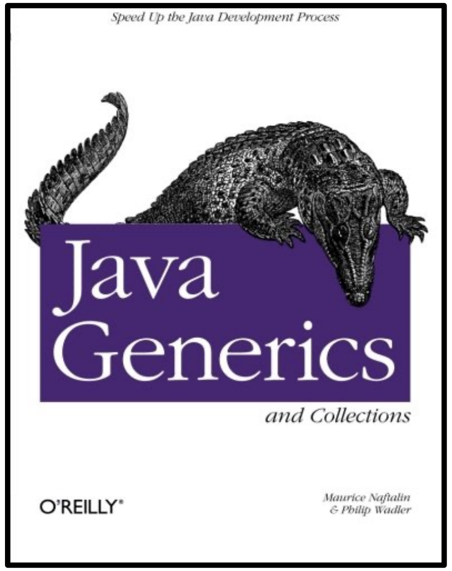
6.9 Arrays as a Deprecated Type?

We have seen that collections are superior to arrays in a number of ways: …

In retrospect, there are several places in Java 5 where avoiding the use of arrays might have improved the design: …

…

Just as the Java 5 design might have been improved if it had put less emphasis on arrays, your own code designs may be improved if you use collections and lists in preference to arrays. Perhaps the time has come to regard arrays as a deprecated type?



Maurice Naftalin
@mauricenaftalin

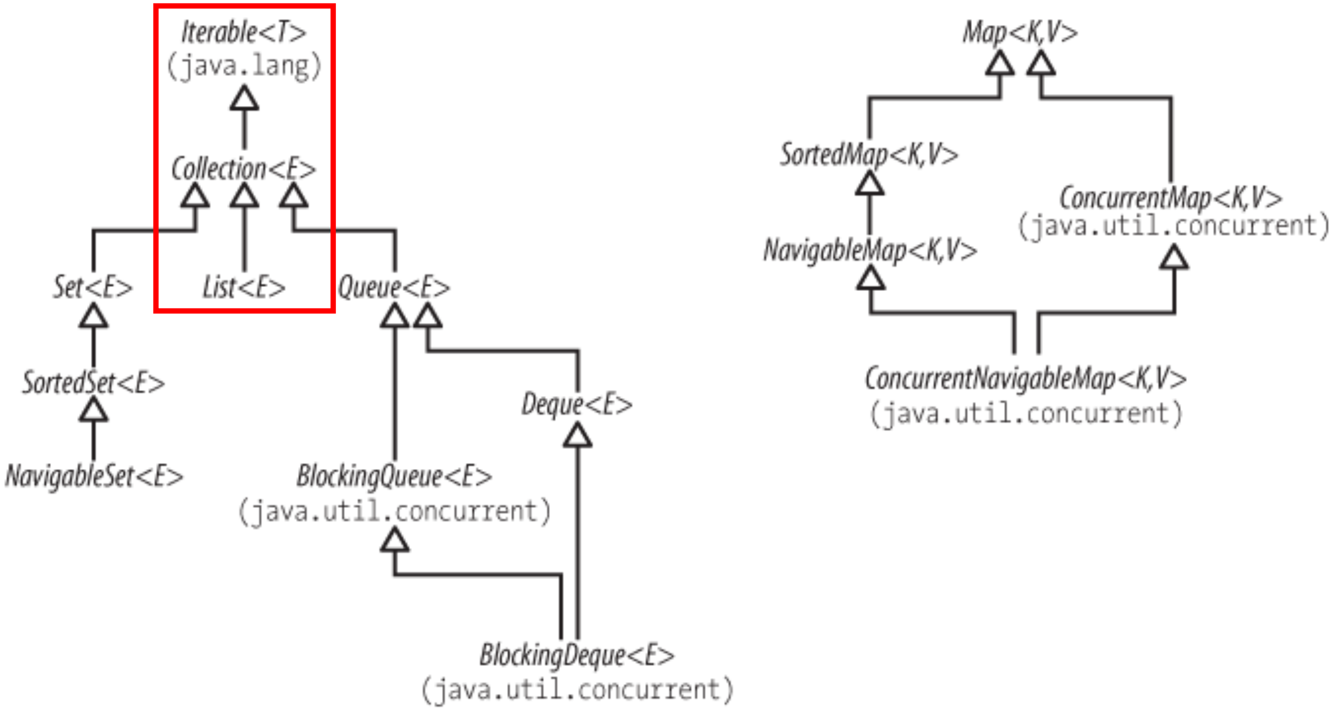
Chapter 10. The Main Interfaces of the Java Collections Framework

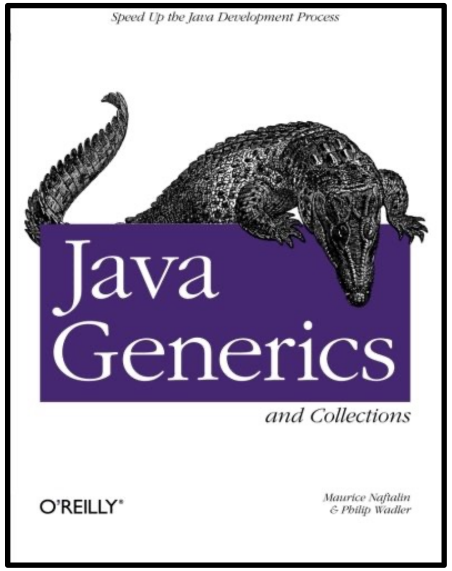
Figure 10-1 shows the main **interfaces** of the **Java Collections Framework**, together with one other—**Iterable**—which is outside the **Framework** but is an essential adjunct to it. Its purpose is as follows:

> **Iterable** defines the contract that a class has to fulfill for its instances to be usable with the *foreach* statement.

And the **Framework interfaces** have the following purposes:

- **Collection** contains the core functionality required of any collection other than a map. It has no direct concrete implementations; the concrete collection classes all implement one of its subinterfaces as well.
- **Set** is a collection, without duplicates, in which order is not significant....
- Queue …
- **List is a collection in which order is significant, accommodating duplicate elements.**
- **Map** …





Maurice Naftalin
@mauricenaftalin

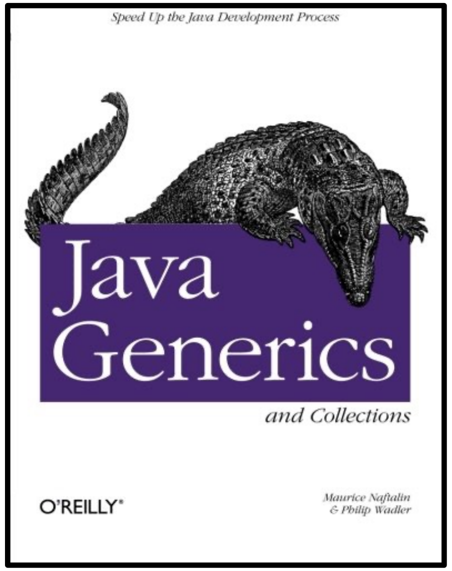
11.2 Implementations

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

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

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

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



Maurice Naftalin
@mauricenaftalin

6.5 Efficiency and the O-Notation

In the last section, we talked about different implementations being "good" for different operations.

A good algorithm is economical in its use of two resources: time and space. Implementations of collections usually use space proportional to the size of the collection, but they can vary greatly in the time required for access and update, so that will be our primary concern.

It's very hard to say precisely how quickly a program will execute, as that depends on many factors, including some that are outside the province of the programmer, such as the quality of the compiled code and the speed of the hardware.

Even if we ignore these and limit ourselves to thinking only about how the execution time for an algorithm depends on its data, detailed analysis can be complex.

A relatively simple example is provided in **Donald Knuth**'s classic book ***Sorting and Searching*** (Addison-Wesley), where the worst-case execution time for a multiple list insertion sort program on **Knuth**'s notional MIX machine is derived as

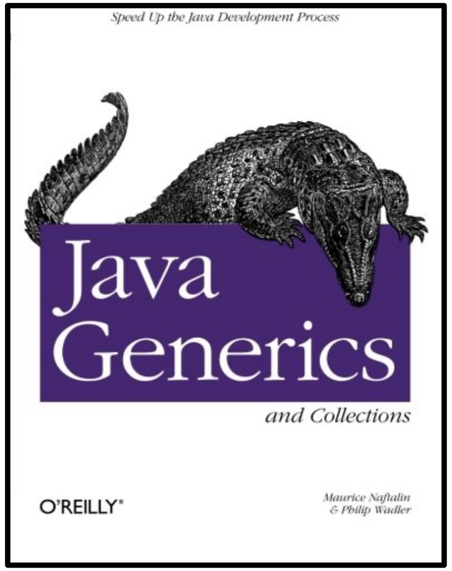
$$3.5N^2 + 24.5N + 4M + 2$$

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

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

The *O*-notation is a way of describing the performance of an algorithm in an abstract way, without the detail required to predict the precise performance of a particular program running on a particular machine.

Our main reason for using it is that it gives us a way of describing how the execution time for an algorithm depends on the size of its data set, provided the data set is large enough.



Maurice Naftalin
@mauricenaftalin

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

For example, in the previous expression the first two terms are comparable for low values of N; in fact, for $N < 8$, the second term is larger.

But as N grows, the first term increasingly dominates the expression and, by the time it reaches 100, the first term is 15 times as large as the second one.

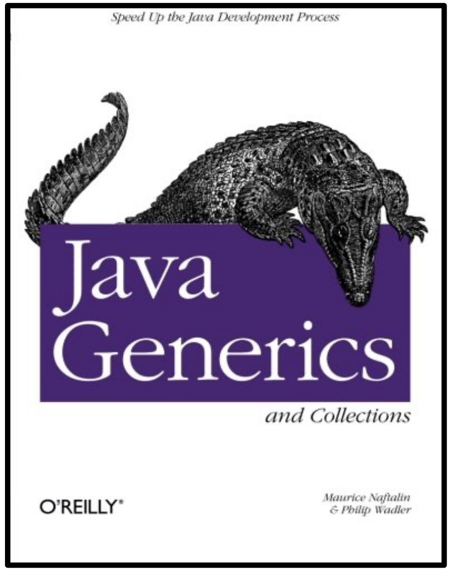
Using a very broad brush, we say that the worst case for this algorithm takes time $O(N^2)$.

We don't care too much about the coefficient because that doesn't make any difference to the single most important question we want to ask about any algorithm: what happens to the running time when the data size increases—say, when it doubles? For the worst-case insertion sort, the answer is that the running time goes up fourfold.

That makes $O(N^2)$ pretty bad—worse than any we will meet in practical use in this book.

Time	Common name	Effect on the running time if N is doubled	Example algorithms
$O(1)$	**Constant**	Unchanged	Insertion into a hash table (Implementing Set)
$O(\log N)$	**Logarithmic**	Increased by a constant amount	Insertion into a tree (TreeSet)
$O(N)$	**Linear**	Doubled	Linear search
$O(N \log N)$		Doubled plus an amount proportional to N	Merge sort (Changing the Order of List Elements)
$O(N^2)$	**Quadratic**	Increased fourfold	

Table 11-1 shows some commonly found running times, together with examples of algorithms to which they apply.



Maurice Naftalin
@mauricenaftalin

For example, many other running times are possible, including some that are much worse than those in the Figure.

Many important problems can be solved only by algorithms that take $O(2^N)$—for these, when N doubles, the running time is squared! For all but the smallest data sets, such algorithms are infeasibly slow.

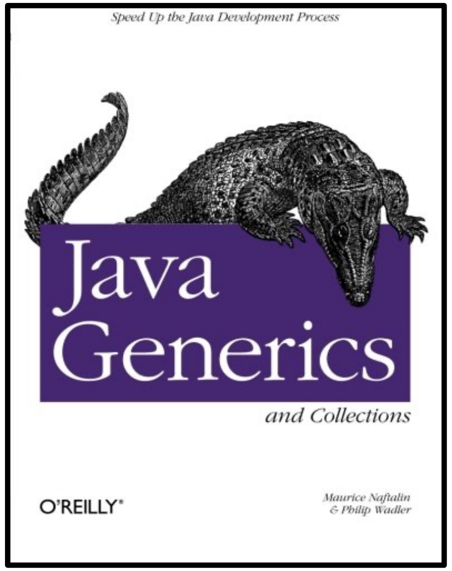
Sometimes we have to think about situations in which the cost of an operation varies with the state of the data structure.

For example, adding an element to the end of an ArrayList can normally be done in constant time, unless the ArrayList has reached its capacity.

In that case, a new and larger array must be allocated, and the contents of the old array transferred into it. The cost of this operation is linear in the number of elements in the array, but it happens relatively rarely.

In situations like this, we calculate the *amortized cost* of the operation—that is, the total cost of performing it n times divided by n, taken to the limit as n becomes arbitrarily large.

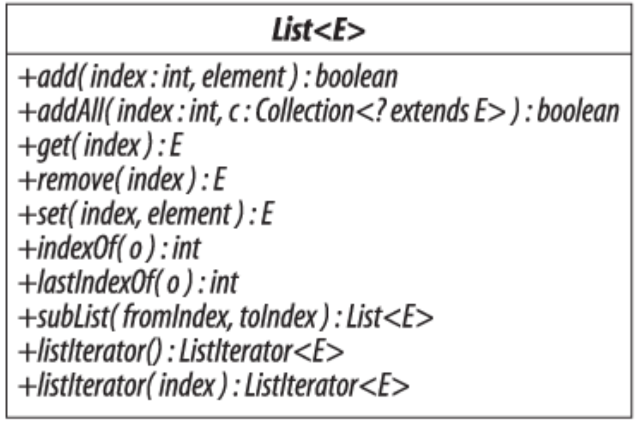
In the case of adding an element to an ArrayList, the total cost for N elements is $O(N)$, so the amortized cost is $O(1)$.



Maurice Naftalin
@mauricenaftalin

Chapter 15. Lists

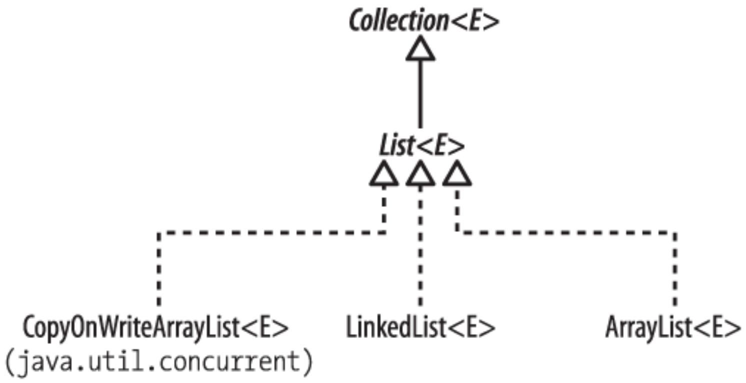
Lists are probably the most widely used Java collections in practice. A *list* is a collection which—unlike a set—can contain duplicates, and which—unlike a queue—gives the user full visibility and control over the ordering of its elements. The corresponding Collections Framework interface is **List** (see Figure 15-1).

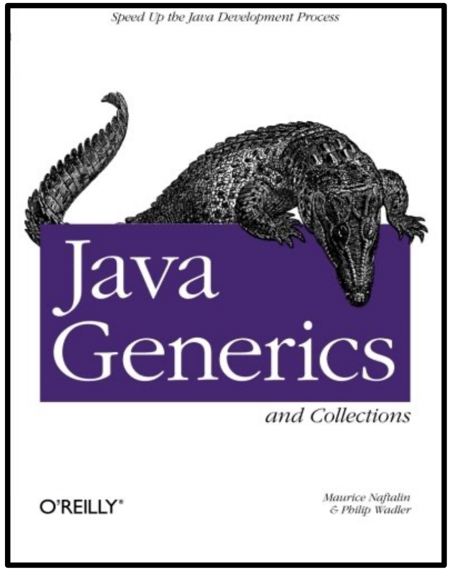


…

15.2 Implementing List

There are three concrete implementations of List in the Collections Framework (see Figure 15-3), differing in how fast they perform the various operations defined by the interface and how they behave in the face of concurrent modification; unlike Set and Queue, however, List has no subinterfaces to specify differences in functional behavior. In this and the following section we look at each implementation in turn and provide a performance comparison.





Maurice Naftalin
@mauricenaftalin

15.2.1 ArrayList

Arrays are provided as part of the Java language and have a very convenient syntax, but their key disadvantage—that, once created, they cannot be resized—makes them increasingly less popular than List implementations, which (if resizable at all) are indefinitely extensible.

The most commonly used implementation of List is, in fact, ArrayList—that is, a List backed by an array.

The standard implementation of ArrayList stores the List elements in contiguous array locations, with the first element always stored at index 0 in the array.

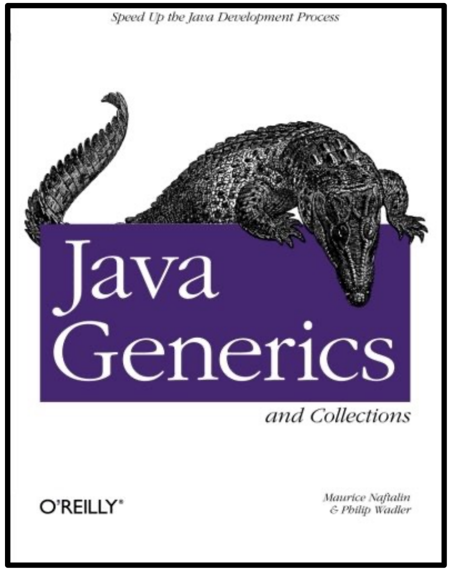
It requires an array at least large enough (with sufficient *capacity*) to contain the elements, together with a way of keeping track of the number of "occupied" locations (the size of the List).

If an ArrayList has grown to the point where its size is equal to its capacity, attempting to add another element will require it to replace the backing array with a larger one capable of holding the old contents and the new element, and with a margin for further expansion (the standard implementation actually uses a new array that is double the length of the old one).

As we explained in Efficiency and the O-Notation, this leads to an amortized cost of $O(1)$.

The performance of ArrayList reflects array performance for "random-access" operations: set and get take constant time. The downside of an array implementation is in inserting or removing elements at arbitrary positions, because that may require adjusting the position of other elements. (We have already met this problem with the remove method of the iterators of array-based queues—for example, ArrayBlockingQueue (see Implementing BlockingQueue).

But the performance of positional add and remove methods are much more important for lists than iterator.remove is for queues.)



Maurice Naftalin
@mauricenaftalin

15.2.2 LinkedList

We discussed **LinkedList** as a **Deque** implementation in Implementing Deque.

You will avoid it as a List implementation if your application makes much use of random access; since the list must iterate internally to reach the required position, positional add and remove have linear time complexity, on average.

Where LinkedList does have a performance advantage over ArrayList is in adding and removing elements anywhere other than at the end of the list; for LinkedList this takes constant time, against the linear time required for noncircular array implementations.

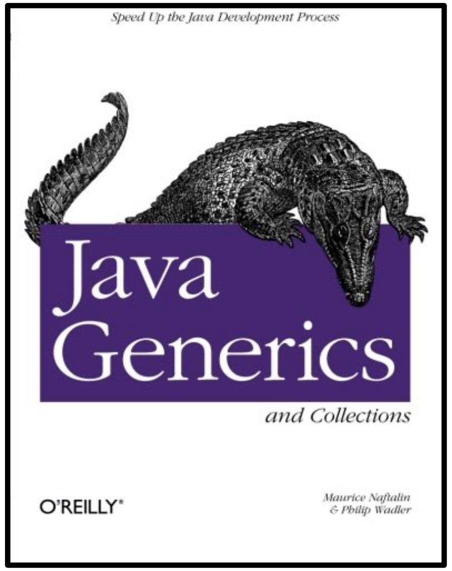
15.2.3 CopyOnWriteArrayList

In Implementing Set **we met CopyOnWriteArraySet, a set implementation designed to provide thread safety together with very fast read access**.

CopyOnWriteArrayList is a List implementation with the same design aims. This **combination of thread safety with fast read access** is useful in some concurrent programs, especially when a collection of observer objects needs to receive frequent event notifications.

The cost is that the array which backs the collection has to be treated as immutable, so a new copy is created whenever any changes are made to the collection. This cost may not be too high to pay if changes in the set of observers occur only rarely.

…



Maurice Naftalin
@mauricenaftalin

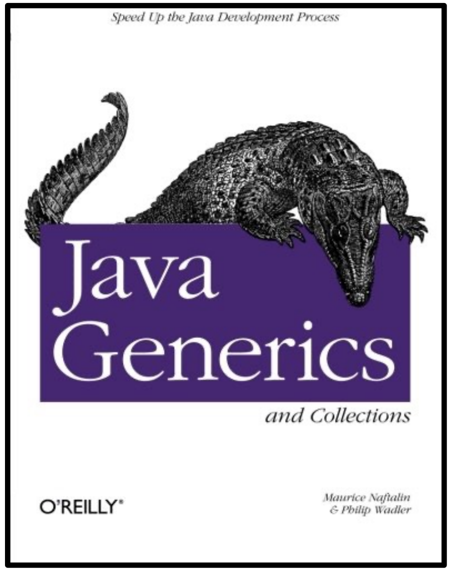
15.3 Comparing List Implementations

Table 15-1 **gives the comparative performance for some sample operations on List classes**. Even though the choice here is much narrower than with queues or even sets, the same process of elimination can be used. As with queues, **the first question to ask is whether your application requires thread safety**. **If so, you should use CopyOnWriteArrayList, if you can**—that is, if writes to the list will be relatively infrequent. If not, you will have to use a synchronized wrapper (see Synchronized Collections) around **ArrayList** or **LinkedList**.

For most list applications the choice is between ArrayList and LinkedList, synchronized or not. Once again, your decision will depend on how the list is used in practice. If set and get predominate, or element insertion and removal is mainly at the end of the list, then ArrayList will be the best choice. If, instead, your application needs to frequently insert and remove elements near the start of the list as part of a process that uses iteration, LinkedList may be better. If you are in doubt, test the performance with each implementation. A Java 6 alternative for single-threaded code that may be worth considering in the last case—if the insertions and removals are actually *at* the start of the list—is to write to the Deque interface, taking advantage of its very efficient ArrayDeque implementation. For relatively infrequent random access, use an iterator, or copy the ArrayDeque elements into an array using toArray.

	get	add	contains	next	remove(0)	iterator .remove
ArrayList	*O(1)*	*O(1)*	*O(n)*	*O(1)*	*O(n)*	*O(n)*
LinkedList	*O(n)*	*O(1)*	*O(n)*	*O(1)*	*O(1)*	*O(1)*
CopyOnWrite-ArrayList	*O(1)*	*O(n)*	*O(n)*	*O(1)*	*O(n)*	*O(n)*

It is possible that, in a future release, ArrayDeque will be retrofitted to implement the List interface; if that happens, it will become the implementation of choice for both Queue and List in single-threaded environments.



Maurice Naftalin
@mauricenaftalin

Step 8. Use lists

One of the big ideas of the functional style of programming is that methods should not have side effects.

A method's only act should be to compute and return a value.

Some benefits gained when you take this approach are that methods become less entangled, and therefore more reliable and reusable.

Another benefit (in a statically typed language) is that everything that goes into and out of a method is checked by a type checker, so logic errors are more likely to manifest themselves as type errors.

Applying this functional philosophy to the world of objects means making objects immutable.

As you've seen, a Scala array is a mutable sequence of objects that all share the same type.

An Array[String] contains only strings, for example.

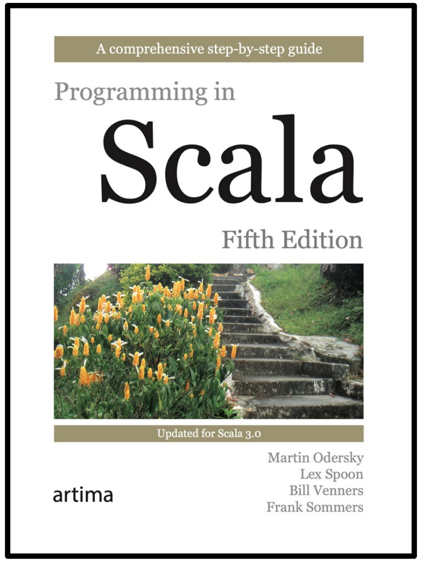
Although you can't change the length of an array after it is instantiated, you can change its element values. Thus, arrays are mutable objects.

For an immutable sequence of objects that share the same type you can use Scala's List class.

As with arrays, a List[String] contains only strings.

Scala's List differs from Java's java.util.List type in that Scala Lists are always immutable (whereas Java Lists can be mutable).

More generally, Scala's List is designed to enable a functional style of programming.



Martin Odersky
@odersky

Step 12. Transform with map and for-yield

When programming in an imperative style, you mutate data structures in place until you achieve the goal of the algorithm. In a functional style, you transform immutable data structures into new ones to achieve the goal.

An important method that facilitates functional transformations on **immutable collections** is **map**.

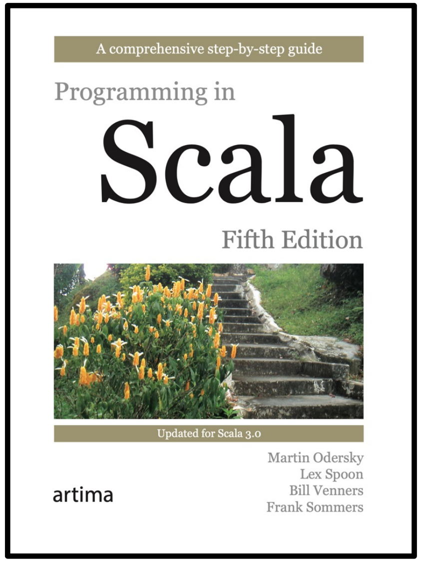
Like foreach, **map** takes a function as a parameter.

But unlike foreach, which uses the passed function to perform a **side effect** for each element, **map** uses the passed function to **transform** each element into a **new value**.

The result of **map** is a new collection containing those **new values**.

…

The **map** method appears on many types, not just **List**. This enables for expressions to be used with many types. One example is **Vector**, which is an immutable sequence that provides "effectively constant time" performance for all its operations. Because **Vector** offers a **map** method with an appropriate signature, you can perform the same kinds of **functional transformations** on **Vectors** as you can on **Lists**, either by calling **map** directly or using for-yield.



Bill Venners
@bvenners

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.

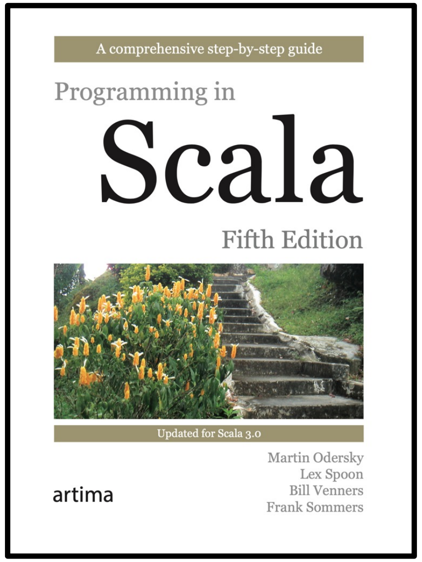
Most collection classes needed by client code exist in three variants, each of which has different characteristics with respect to mutability. The three variants are located in packages scala.collection, scala.collection.immutable, and scala.collection.mutable.

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].



Lex Spoon

Generally, the root collections in package scala.collection support transformation operations affecting the whole collection, such as map and filter. The immutable collections in package scala.collection.immutable typically add operations for adding and removing single values, and the mutable collections in package scala.collection.mutable add some side-effecting modification operations to the root interface.

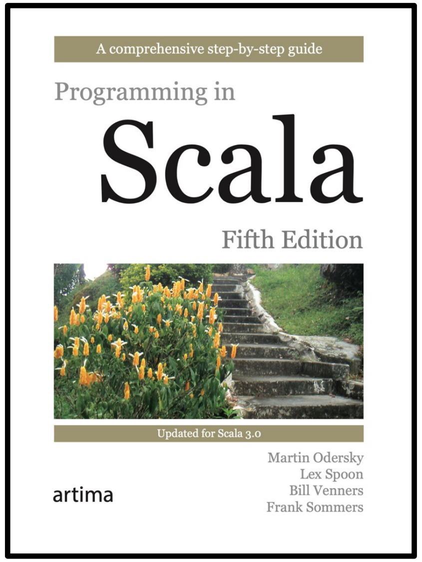
Another difference between root collections and immutable collections is that clients of an immutable collection have a guarantee that nobody can mutate the collection, whereas clients of a root collection only know that they can’t change the collection themselves. Even though the static type of such 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 Set without 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 explicitly collection.mutable.Set, or collection.mutable.Iterable.

The last package in the collection hierarchy is collection.generic. This package contains building blocks for abstracting over concrete collections. Everyday users of the collection framework should need to refer to classes in generic only in exceptional circumstances.



Frank Sommers

24.2 Collections consistency

The most important collection classes are shown in Figure 24.1.

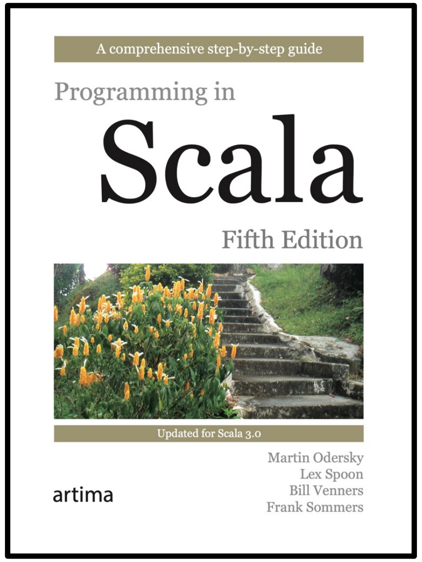
```
Iterable
    Seq
        IndexedSeq
            …
            Vector
            …
        LinearSeq
            List
        …
        Buffer
            ListBuffer
            ArrayBuffer
    Set
        …
    Map
        …
```

24.3 Trait Iterable

At the top of the collection hierarchy is trait Iterable[A], where A is the type of the collection's elements. All methods in this trait are defined in terms of an abstract method, **iterator**, which yields the collection's elements one by one.

```
def iterator: Iterator[A]
```

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



Martin Odersky
@odersky

Iterable also defines many concrete methods

- *Iteration operations* …
- *Addition* …
- *Map operations*
- *Conversions* …
- *Copying operations* …
- *Size operations* …
- *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.** …
- *Subcollection retrieval operations* …
- *Subdivision operations* …
- *Element tests* …
- *Specific folds* …
- *String operations* …
- *View operation* …

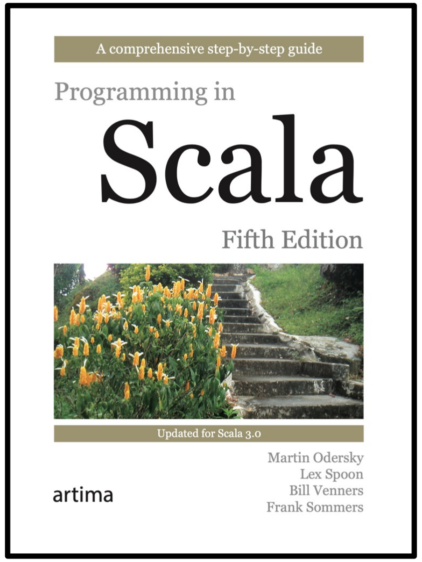
…

24.4 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 Table 24.2, fall into the following categories:

Indexing and length operations **apply**, isDefinedAt, **length**, indices, lengthCompare, and lengthIs. **For a Seq, apply 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.**



Bill Venners
@bvenners

- *Addition operations* +: (alias, prepended), ++: (alias, prependedAll), :+ (alias, appended), :++ (alias, appendedAll), 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* …
- *Reversal operations* …
- *Comparison operations* …
- *Multiset operations* …

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 update method 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.

Each Seq trait has two subtraits, LinearSeq and IndexedSeq, which offer different performance characteristics.

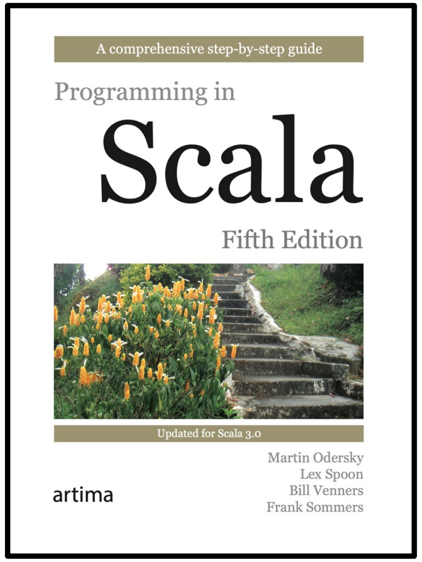
A linear sequence has efficient head and tail operations, whereas an indexed sequence has efficient apply, length, and (if mutable) update operations.

List is a frequently used linear sequence, as is LazyList.

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 24.7.



Lex Spoon

Mutable IndexedSeq adds operations for transforming its elements in place. These operations (mapInPlace, sortInPlace, sortInPlaceBy, sortInPlaceWith) …, contrast with operations such as map and sort, available on Seq, which return a new collection instance.

Buffers

An important sub-category of mutable sequences is buffers.

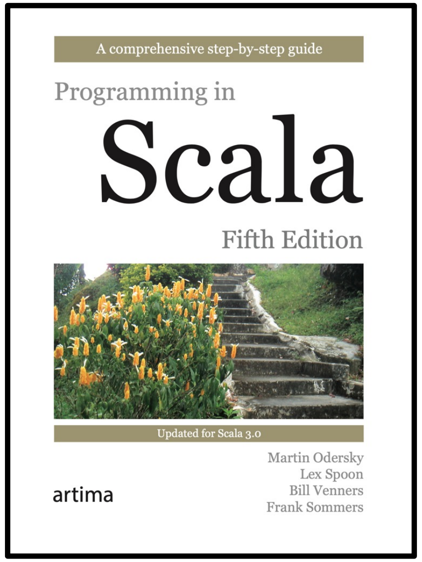
Buffers allow not only updates of existing elements but also element insertions, element removals, and efficient additions of new elements at the end of the buffer.

The principal new methods supported by a buffer are += (alias, append) and ++= (alias, appendAll) for element addition at the end, +=: (alias, prepend) and ++=: (alias, prependAll) for addition at the front, insert and insertAll for element insertions, and remove, -= (alias, subtractOne) and --= (alias, subtractAll) for element removal. …

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.

…



Frank Sommers

24.7 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 14 and 1 for extensive discussions about lists.

LazyLists

…

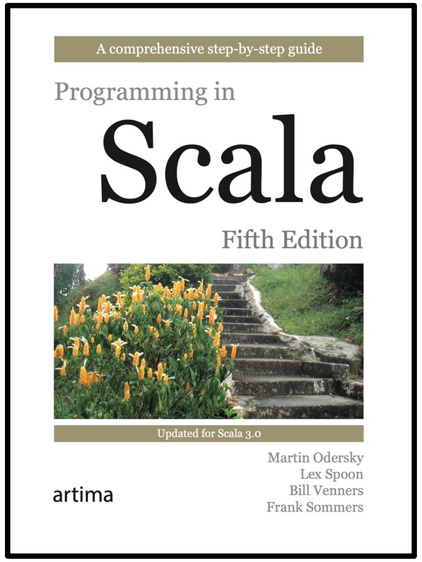
Immutable ArraySeqs

Lists are very efficient if you use algorithms that work exclusively at the front of the list. Accessing, adding, and removing the head of a list takes constant time. Accessing or modifying elements deeper in the list, however, takes time linear in the depth into the list. As a result, a list may not be the best choice for algorithms that don't limit themselves to processing just the front of the sequence.

ArraySeq is an immutable sequence type, backed by a private Array, that addresses the inefficiency of random access on lists.

ArraySeqs allow you to access any element of the collection in constant time. As a result, you need not worry about accessing just the head of an ArraySeq. Because you can access elements at arbitrary locations in contant time, ArraySeqs can be more efficient than lists for some algorithms.

On the other hand, since ArraySeqs are backed by an Array, prepending to an ArraySeq requires linear time, not constant time as with list. Moreover, any addition or update of a single element requires linear time on ArraySeq, because the entire underlying array must be copied.



Martin Odersky
@odersky

Vectors

List and ArraySeq are efficient data structures for some use cases but inefficient for others.

For example, prepending an element is constant time for List, but linear time for ArraySeq.

Conversely, indexed access is constant time for ArraySeq, but linear time for List.

Vector provides good performance for all its operations.

Access and update to any elements of a vector takes 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 ArraySeq, but it's a constant nonetheless.

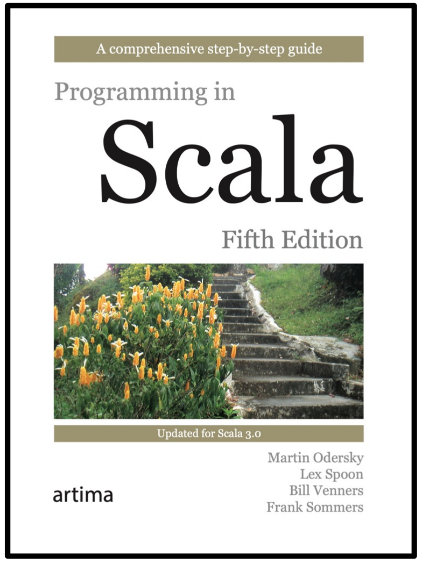
As a result, algorithms using vectors do not have to be careful about accessing or updating just the head of the sequence. They can access and update elements at arbitrary locations, and thus they can be much more convenient to write.

Vectors are built and modified just like any other sequence: …

…

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 2^{15} elements, three hops for vectors with 2^{20}, four hops for vectors with 2^{25} elements and five hops for vectors with up to 2^{30} 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."



Bill Venners
@bvenners

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:

```
val vec = Vector(1, 2, 3)
vec.updated(2, 4) // Vector(1, 2, 4)
vec // Vector(1, 2, 3)
```

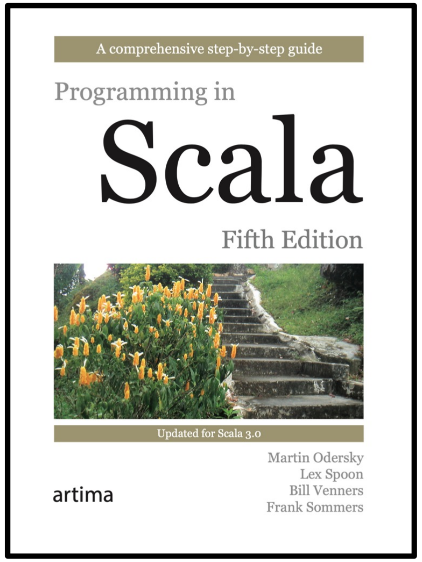
As the last line above shows, a call to updated has no effect on the original vector vec.

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:

```
collection.immutable.IndexedSeq(1, 2, 3) // Vector(1, 2, 3)
```



Frank Sommers

While Scala favours immutability, it always gives you a choice. If you have a use case that requires mutable collections, preferably locally inside of a method, you have all the tools you need. In this chapter we'll only take a quick look at the mutable collection types and focus on the immutable ones.

Immutable collections

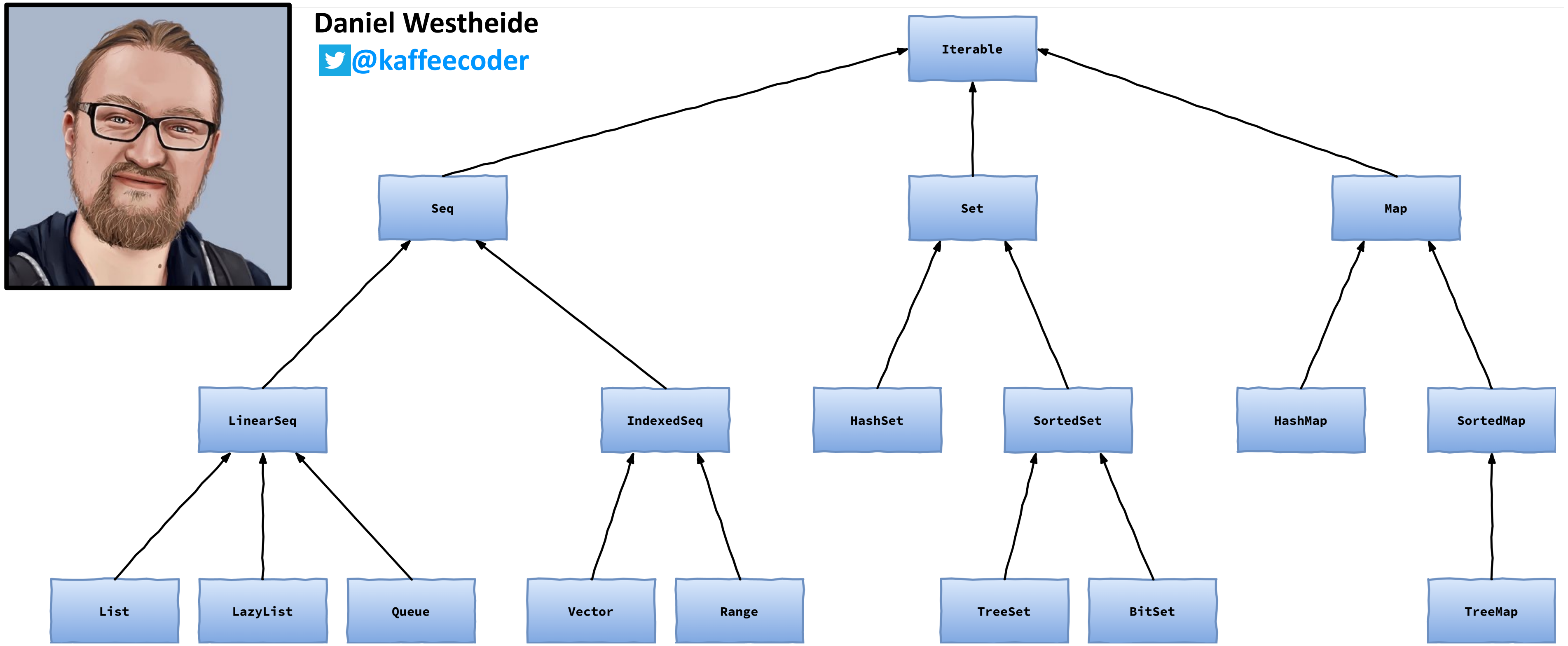
Figure 7.1 provides a somewhat simplified overview of the immutable collection type hierarchy. All types that are merely implementation details have been omitted.

The most general immutable collection type is `Iterable`. A data structure of this type allows you to iterate over its elements — but also to do all kinds of other things that are possible because we can iterate. For example, you can `map`, `flatMap`, and `filter` an `Iterable`.

There are three subtypes of `Iterable`: `Seq`, `Set`, and `Map`. While all of them can be iterated over, each of these has certain unique properties:

- A `Seq` allows access to elements by index. Being ordered like this, it also allows you to prepend or append elements.

Daniel Westheide
@kaffeecoder



Indexed versus linear sequences

Often, all you care about is that the elements of a collection form a sequence with a well-defined ordering. In that case, it's fine to program against the interface provided by the `Seq` trait. But sometimes you know that the way you're going to process these sequences requires efficient random access by index, or that you need to know the number of elements in the sequence. In other cases, you might know that you're going to need efficient access to the head and the tail of the sequence — the first element and the remaining elements of the sequence.

In such cases you may want to program against a more specific interface. The Scala collections library provides two subtypes of `Seq`: `IndexedSeq` and `LinearSeq`, respectively. Each of them makes different promises regarding the time complexity of these operations. While an `IndexedSeq` is a good fit for the first use case, a `LinearSeq` caters to the second one. Both of these are purely marker traits. They do not provide any additional operators, they are merely about signaling different time complexity for certain operations at runtime.

Linear sequences

There are three concrete types of `LinearSeq`: `List`, `LazyList`, and `Queue`. We're going to look at LazyList a bit later in this chapter. `Queue` provides a first-in-first-out data structure that is optimized both for efficient enqueuing and dequeuing.

`List` is a linked list. It's defined as an algebraic data type. A `List[A]` is either an empty list or a value plus a reference to the remaining items.

…

Indexed sequences

There are two concrete types of `IndexedSeq`. `Vector` allows you fast random access to its elements. `Range` is interesting because it's not a generic collection type. `Range` is a subtype of `IndexedSeq[Int]`. If you need to create an inclusive range of `Int` numbers, you can call the to method available on `Int` as an extension method…

…

Daniel Westheide
@kaffeecoder

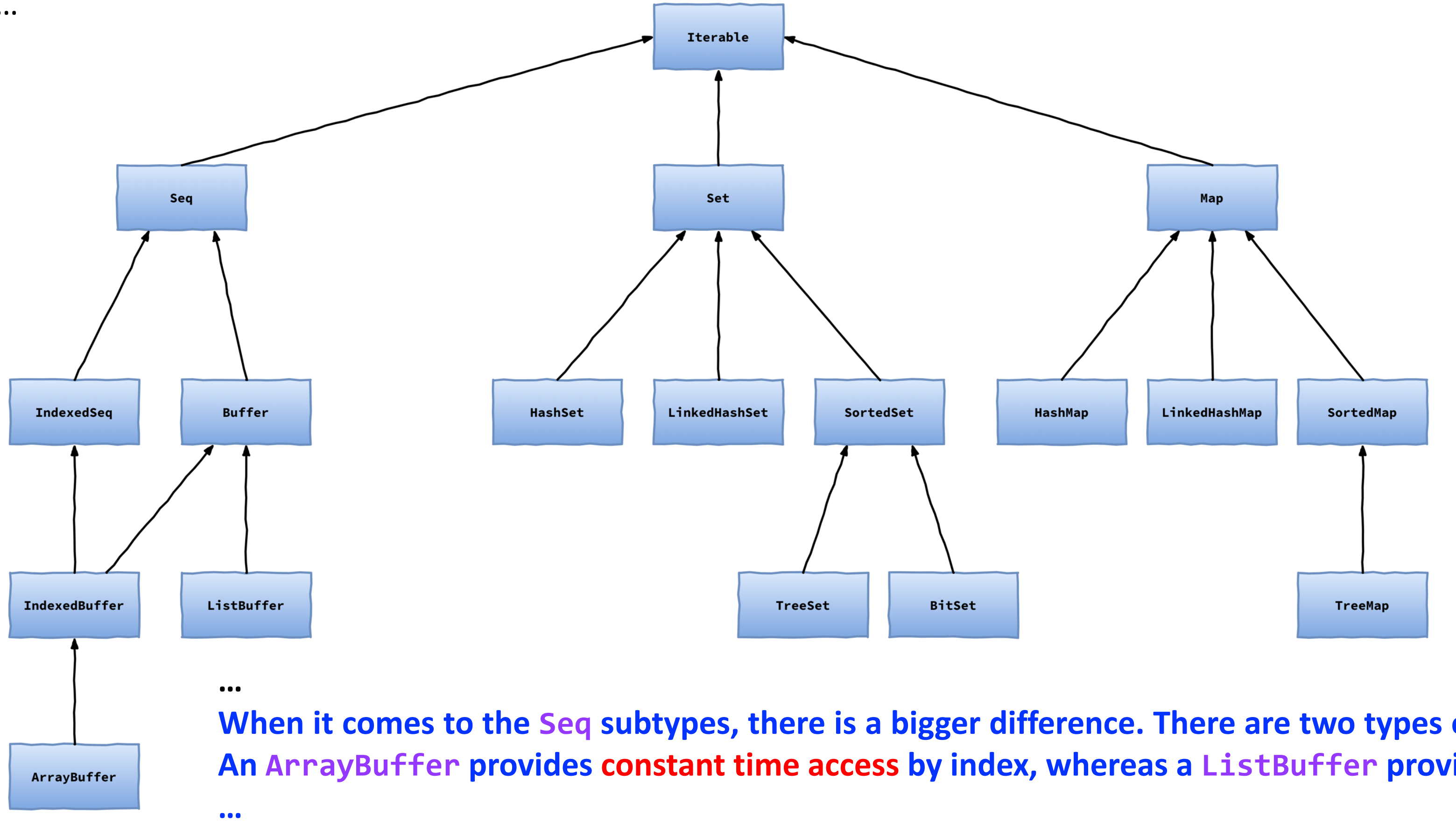
Mutable collections

While the immutable collection types are used most commonly, some people have the occasional need for working with mutable collections. Scala caters to these needs as well, so you don't have to resort to the Java collections library in these situations.

Figure 7.2 provides an overview of the mutable collection type hierarchy. Please note that it's somewhat simplified, leaving out some types that are merely implementation details.

At a high level, the hierarchy of mutable collection types looks very similar to the immutable ones. Again, the most general collection type is `Iterable`. It doesn't actually provide any mutable operations, so its capabilities are identical to those of an immutable `Iterable`. Again, there are three subtypes of `Iterable`: `Seq`, `Set`, and `Map`. Unlike an immutable `Seq`, a mutable one allows you to update an element at a specific index in place, using the update method.

…



Daniel Westheide
@kaffeecoder

…

When it comes to the `Seq` subtypes, there is a bigger difference. There are two types of `Buffer`, which is a growable and shrinkable kind of `Seq`. An `ArrayBuffer` provides constant time access by index, whereas a `ListBuffer` provides constant time prepend and append operations.

…

11.1 Choosing a Collection Class

Problem

You want to choose a Scala collection class to solve a particular problem.

Solution

...

Choosing a sequence

When choosing a sequence— a sequential collection of elements — you have two main decisions:

- Should the sequence be indexed, allowing rapid access to any elements, or should it be implemented as a linked list?
- Do you want a mutable or immutable collection?

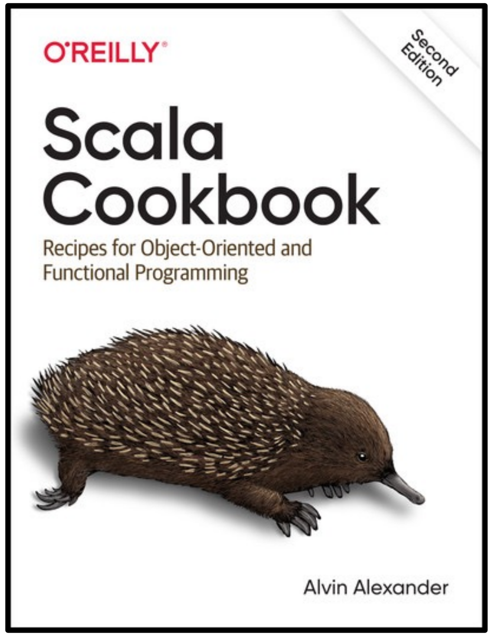
Beginning with Scala 2.10 and continuing with Scala 3, the recommended general-purpose go-to sequential collections for the combinations of mutable/immutable and indexed/linear are shown in Table 11-2.

Table 11-2. Scala's recommended general-purpose sequential collections

	Immutable	Mutable
Indexed	`Vector`	`ArrayBuffer`
Linear (Linked lists)	`List`	`ListBuffer`

As an example of reading that table, if you want an immutable, indexed collection, in general you should use a Vector; if you want a mutable, indexed collection, use an ArrayBuffer (and so on).

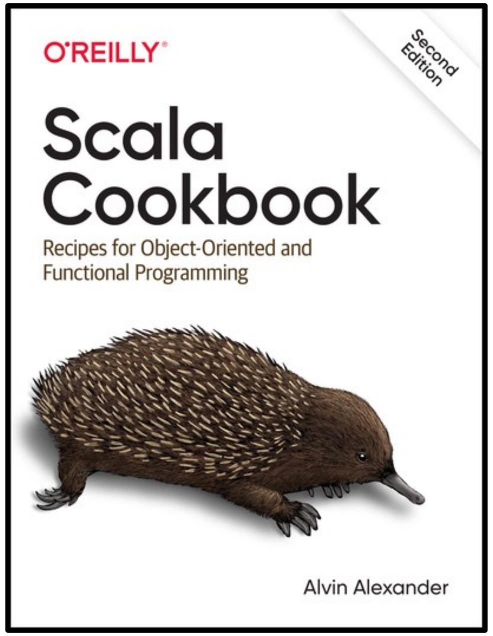
While those are the general-purpose recommendations, there are many more sequence alternatives.



Alvin Alexander
@alvinalexander

While those are the general-purpose **recommendations**, there are many more **sequence** alternatives. The most common ***immutable* sequence** choices are shown in

Class	IndexedSeq	LinearSeq	Description
`LazyList`		…	…
`List`		✅	The go-to **immutable linear sequence**, it is a **singly linked list**. Suited for **prepending** elements, and for **recursive algorithms** that work by operating on the list's head and tail.
`Queue`		…	…
`Range`	…		…
`Vector`	✅		The go-to **immutable indexed sequence**. The Scaladoc states, "It provides **random access** and **updates** in **effectively constant time**, as well as **very fast append** and **prepend**."

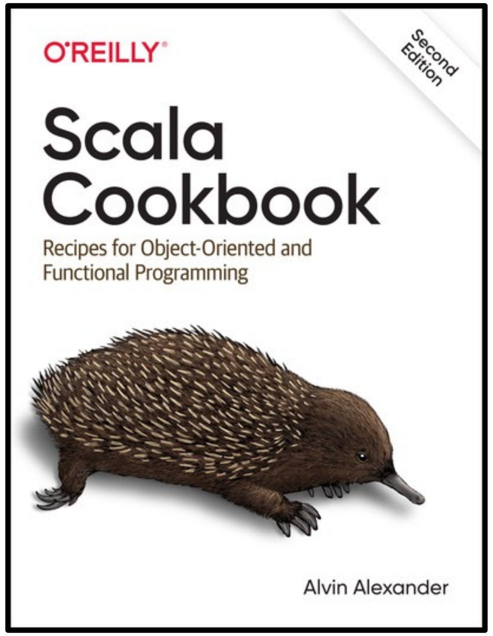


Alvin Alexander
@alvinalexander

The most common ***mutable*** sequence choices are shown in Table 11-4. Queue and Stack are also in this table because there are **immutable** and **mutable** versions of these classes. All quotes in the descriptions come from the Scaladoc for each class.

Class	IndexedSeq	LinearSeq	Buffer	Description
Array	✅			Backed by a **Java array**, **its elements are mutable, but it can't change in size**.
ArrayBuffer	✅		✅	The **go-to** class for a **mutable indexed sequence**. "Uses an **array** internally. **Append**, **update** and **random access** take **constant time** (**amortized time**). **Prepends** and **removes** are **linear** in the buffer size."
ArrayDeque	…			…
ListBuffer		✅	✅	Like an **ArrayBuffer**, but backed by a **list**. The documentation states, "If you plan to convert the **buffer** to a **list**, use **ListBuffer** instead of **ArrayBuffer**." Offers **constant-time prepend** and **append**; most other operations are **linear**.
Queue	…			…
Stack	…			…
StringBuilder	…			…

Note that I list **ArrayBuffer** and **ListBuffer** under two columns. That's because while they are both descendants of **Buffer**—which is a **Seq** that can grow and shrink—**ArrayBuffer** behaves like an **IndexedSeq** and **ListBuffer** behaves like a **LinearSeq**.
In addition to the information shown in these tables, performance can be a consideration. **See** Recipe 11.2 **if performance is important to your selection process**.

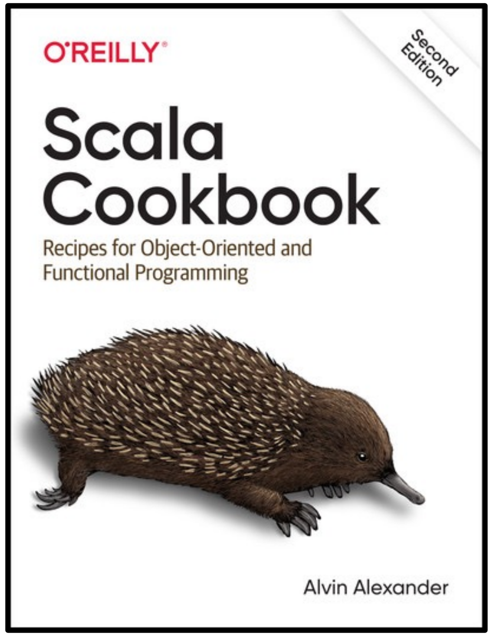


Alvin Alexander
@alvinalexander

When creating an API for a library, you may want to refer to your **sequences** in terms of their superclasses. Table 11-5 shows the **traits** that are often used when referring generically to a collection in an API. Note that all quotes in the descriptions come from the Scaladoc for each class.

Table 11-5. **Traits** commonly used in library APIs

Trait	Description
`IndexedSeq`	A **sequence** that implies that **random access** of elements is **efficient**. "Have **efficient** **apply** and **length**."
`LinearSeq`	A **sequence** that implies that **linear access** to elements is **efficient**. "Have **efficient** **head** and **tail** operations."
`Seq`	The **base trait** for **sequential collections**. Use when it isn't important to indicate that the **sequence** is **indexed** or **linear** in nature.
`Iterable`	The **highest collection level**. Use it when you want to be **very generic** about the type being returned. (It's the rough equivalent of declaring that a Java method returns Collection.)



Alvin Alexander
@alvinalexander

11.2 Understanding the Performance of Collections

Problem

When choosing a collection for an application where performance is important, you want to choose the right collection for the algorithm.

Solution

In many cases, **you can reason about the performance of a collection by understanding its basic structure**.

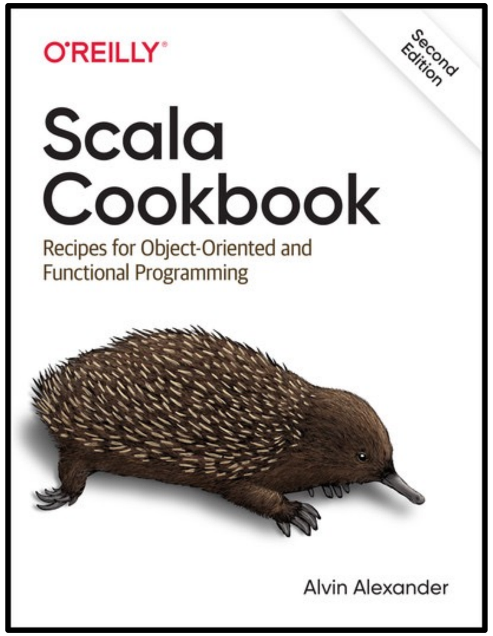
For instance, a **List** is a **singly linked list**, and because it's not **indexed**, if you need to access an element like list(1_000_000), that requires traversing one million elements. Therefore it's going to be much slower than accessing the one-millionth element of a **Vector**, because **Vector** is **indexed**.

In other cases, it can help to look at the tables. For instance, Table 11-10 shows that the ***append*** operation on a **Vector** is **eC**, or ***effectively constant time***. As a result, I can create a large **Vector** in the REPL on my computer in under a second like this:

```
var a = Vector[Int]()
for i <- 1 to 50_000 do a = a :+ i
```

However, as the table shows, the **append operation** on a **List** requires **linear time**, so attempting to create a **List** of the same size takes a much longer time—over 15 seconds.

Note that neither of those approaches is recommended for real-world code. I only use them to demonstrate the performance difference between **Vector** and **List** for **append operations**.



Alvin Alexander

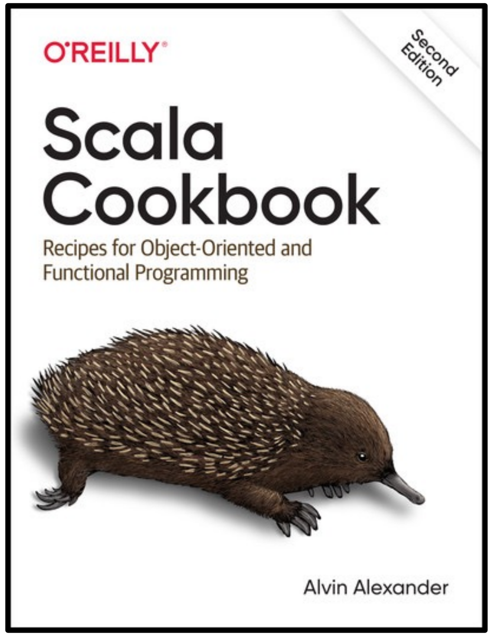
@alvinalexander

Performance characteristics keys

Before looking at the performance tables, Table 11-9 shows the performance characteristic keys that are used in the tables that follow it.

Table 11-9. Performance characteristic keys for the subsequent tables

Key	Description
Con	The operation takes (**fast**) **constant time**.
eC	The operation takes **effectively constant time**, but this might depend on some assumptions, such as maximum length of a vector, or 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**.
Lin	The operation is **linear**, so the **time is proportional to the collection size**.
-	The operation is not supported.



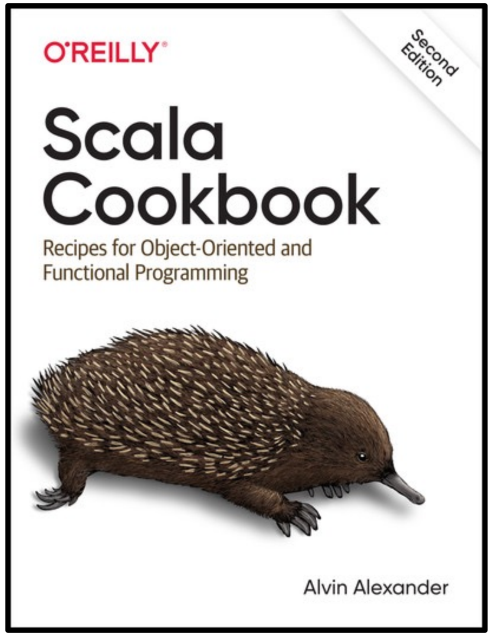
Alvin Alexander
@alvinalexander

Performance characteristics for sequential collections

Table 11-10 shows the performance characteristics for operations on immutable and mutable sequential collections.

Table 11-10. Performance characteristics for sequential collections

	head	tail	apply	update	prepend	append	insert
Immutable							
List	Con	Con	Lin	Lin	Con	Lin	-
LazyList							
ArraySeq							
Vector	eC	eC	eC	eC	eC	eC	-
Queue							
Range							
String							
Mutable							
ArrayBuffer	Con	Lin	Con	Con	Lin	aC	Lin
ListBuffer	Con	Lin	Lin	Lin	Con	Con	Lin
StringBuilder							
Queue							
ArraySeq							
Stack							
Array	Con	Lin	Con	Con	-	-	-
ArrayDeque							



Alvin Alexander
@alvinalexander

Performance characteristics for sequential collections

Table 11-10 shows the performance characteristics for operations on immutable and mutable sequential collections.

Table 11-10. Performance characteristics for sequential collections

Operation	Description
head	Selecting the first element of the sequence.
tail	Producing a new sequence that consists of all elements of the sequence except the first one.
apply	Indexing.
update	Functional update for immutable sequences, side-effecting 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 supported directly only for mutable sequences.

…

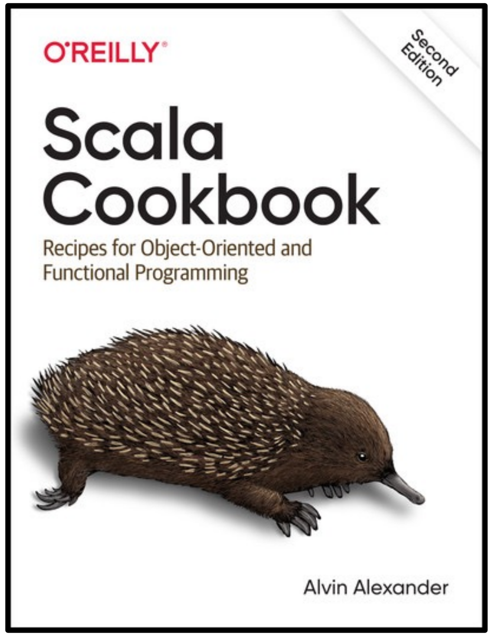
Discussion

As you can tell from the descriptions of the keys in Table 11-9, when choosing a collection you'll generally want to look for the Con, eC, and aC keys to find your best performance.

For instance, because List is a singly linked list, accessing the head and tail elements are fast operations, as is the process of prepending elements, so those operations are shown with the Con key in Table 11-10. But appending elements to a List is a very slow operation—linear in proportion to the size of the List—so the append operation is shown with the Lin key.

See Also

With permission from EPFL, the tables in this recipe have been reproduced from the performance characteristics Scala documentation page.



Alvin Alexander

@alvinalexander

Chapter 12. Collections: Common Sequence Classes

In this chapter on the Scala collections, we'll examine the most common sequence classes.

As mentioned in Recipe 11.1, "Choosing a Collections Class", the general sequence class recommendations are to use:

- Vector as your go-to immutable indexed sequence
- List as your go-to immutable linear sequence
- ArrayBuffer as your go-to mutable indexed sequence
- ListBuffer as your go-to mutable linear sequence

	Immutable	Mutable
Indexed	`Vector`	`ArrayBuffer`
Linear (Linked lists)	`List`	`ListBuffer`

Vector

As discussed in Recipe 11.1, "Choosing a Collections Class", **Vector is the preferred immutable indexed sequence class because of its general performance characteristics. You'll use it all the time when you need an immutable sequence.**
Because Vector is immutable, you apply filtering and transformation methods on one Vector to create another one. …
…

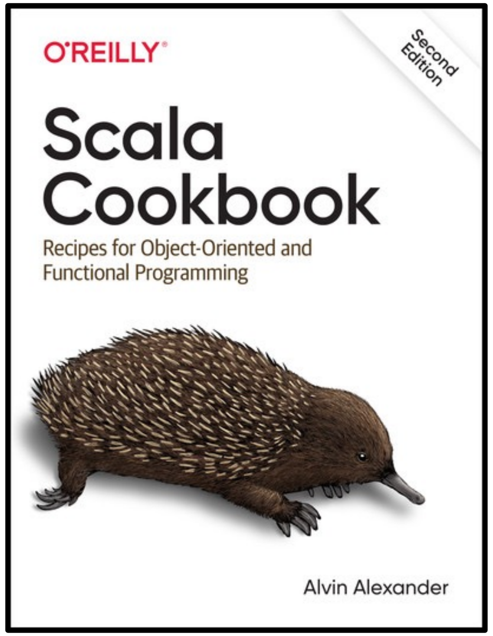
List

If you're coming to Scala from Java, you'll quickly see that despite their names, the Scala List class is nothing like the Java List classes, such as the Java ArrayList. The Scala List class is immutable, so its size as well as the elements it contains can't change. It's implemented as a linked list, where the preferred approach is to *prepend* elements. Because it's a linked list, you typically traverse the list from head to tail, and indeed, it's often thought of in terms of its head and tail methods (along with isEmpty).

Like Vector, because a List is immutable, you apply filtering and transformation methods on one list to create another list. …
…

LIST VERSUS VECTOR

You may wonder when you should use a List instead of a Vector. The performance characteristics detailed in Recipe 11.2, "Understanding the Performance of Collections", **provide the general rules about when to select one or the other.**
…
So List definitely has its uses, especially when you think of it as what it is, a simple singly linked list. …



Alvin Alexander
@alvinalexander

ArrayBuffer

ArrayBuffer is the preferred mutable indexed sequence class. Because it's mutable, you apply transformation methods directly on it to update its contents.

…

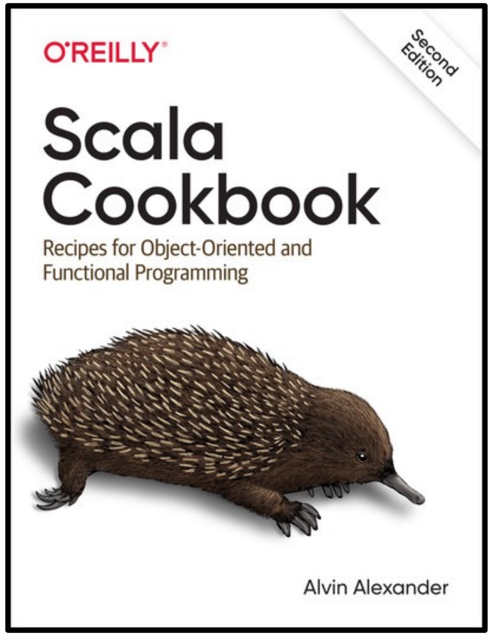
Array

The Scala Array is unique: it's mutable in that its elements can be changed, but immutable in size—it can't grow or shrink. By comparison, other collections like List and Vector are completely immutable, and ArrayBuffer is completely mutable.

Array has the unique distinction of being backed by the Java array, so a Scala Array[Int] is backed by a Java int[].

Although the Array may often be demonstrated in Scala examples, the recommendation is to use the Vector class as your go-to *immutable* indexed sequence class, and ArrayBuffer as your *mutable* indexed sequence of choice. In keeping with this suggestion, in my real-world code, I use Vector and ArrayBuffer for those use cases, and then convert them to an Array when needed.

For some operations the Array can have better performance than other collections, so it's important to know how it works. See Recipe 11.2, "Understanding the Performance of Collections", for those details.



Alvin Alexander
@alvinalexander

12.1 Making Vector Yout Go-To Immutable Sequence

Problem

You want a fast general-purpose immutable sequential collection type for your Scala applications.

Solution

The Vector class is considered the go-to general-purpose *indexed* immutable sequential collection. Use a List if you prefer working with a *linear* immutable sequential collection.

…

Discussion

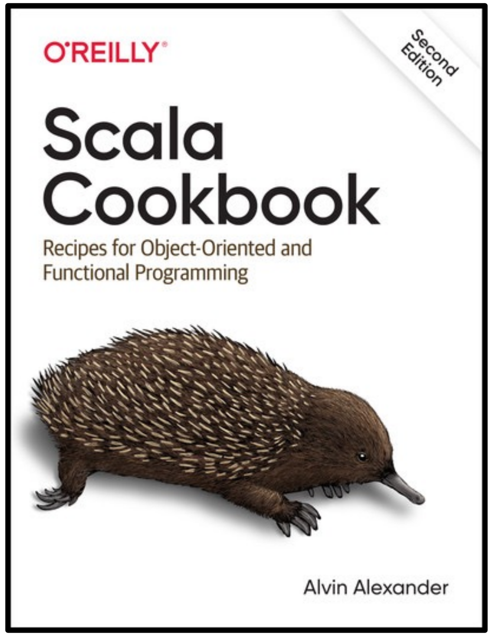
The Scala documentation on concrete immutable collection classes states the following:

> *Vector is a collection type that addresses the inefficiency for random access on lists. Vectors allow accessing any element of the list in "effectively" constant time….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.*

As noted in "Understanding the Collections Hierarchy", when you create an instance of an IndexedSeq, Scala returns a Vector:

```
scala> val x = IndexedSeq(1,2,3)
x: IndexedSeq[Int] = Vector(1, 2, 3)
```

As a result, I've seen some developers use an IndexedSeq in their code rather than a Vector to express their desire to create an indexed immutable sequence and leave the implementation details to the compiler.



Alvin Alexander
@alvinalexander

12.7 Making ArrayBuffer Yout Go-To Mutable Sequence

Problem

You want to create an array whose size can change, i.e., a completely mutable array.

Solution

An Array is mutable in that its elements can change, but its size can't change. To create a mutable indexed sequence whose size can change, use the ArrayBuffer class.

…

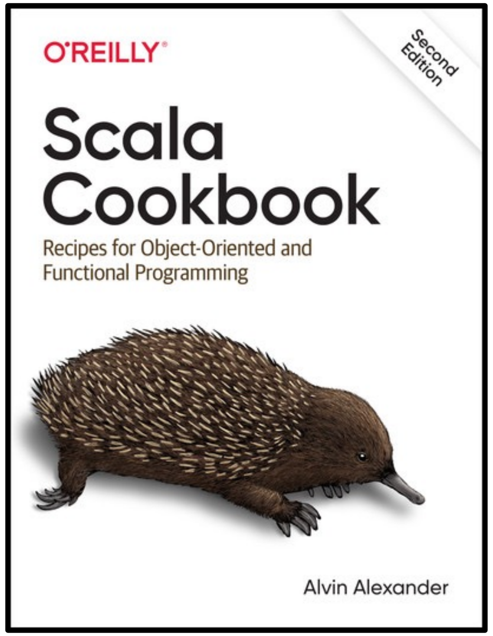
Discussion

…

Notes about ArrayBuffer and ListBuffer

The ArrayBuffer Scaladoc provides these details about **ArrayBuffer** performance: **"Append, update, and random access take constant time (amortized time). Prepends and removes are linear in the buffer size."**

If you need a mutable sequential collection that works more like a List (i.e., a linear sequence rather than an indexed sequence), use ListBuffer instead of ArrayBuffer. The **Scala** documentation on the **ListBuffer** states, "**A Buffer implementation backed by a list. It provides constant time prepend and append. Most other operations are linear.**" See Recipe 12.5 for more **ListBuffer** details.



Alvin Alexander
@alvinalexander

4.2 Immutable Collections.

While Arrays are the low-level primitive, most Scala applications are built upon its mutable and immutable collections: Vectors, Lists, Sets and Maps. Of these, immutable collections are by far the most common.

Immutable collections rule out an entire class of bugs due to unexpected modifications, and are especially useful in multi-threaded scenarios, where you can safely pass immutable collections between threads without worrying about thread-safety issues. Most immutable collections use Structural Sharing (4.4.2) to make creating and updated copies cheap, allowing you to use them in all but the most performance critical code.

4.2.1 Immutable Vectors.

Vectors are fixed-size, immutable linear sequences. They are a good general-purpose sequence data structure, and provide efficient $O(\log n)$ performance for most operations.

…

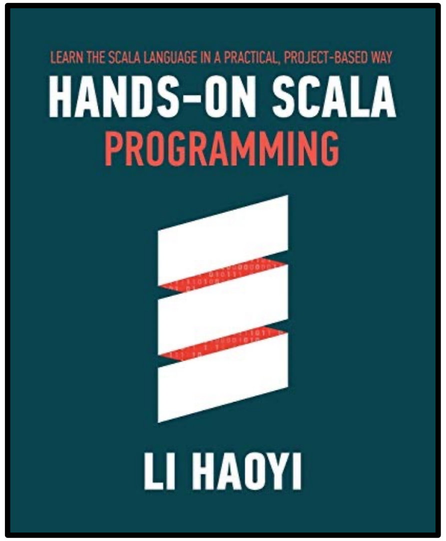
Unlike Arrays, where `a(…) = …` mutates it in place, a Vector's .updated method returns a new Vector with the modification, while leaving the old Vector unchanged. Due to Structural Sharing, this is a reasonably efficient $O(\log n)$ operation. Similarly, using :+ and +: to create a new Vector with additional elements on either side, or using tail to create a new Vector with one element removed, are all $O(\log n)$ as well.

Vectors support the same Operations (4.1) that Arrays and other collections do: builders (4.1.1), factory methods (4.1.2), transforms (4.1.3), etc.

In general, using Vectors is handy when you have a sequence you know you will not change, but need flexibility in how you work with it. Their tree structure makes most operations reasonably efficient, although they will never be quite as fast as Arrays for in-place updates or immutable Lists (4.2.5) for adding and removing elements at the front.

4.2.2 Structural Sharing.

Vectors implement their $O(\log n)$ copy-and-update operations by re-using portions of their tree structure. This avoids copying the whole tree, resulting in a "new" Vector that shares much of the old tree structure with only minor modifications.

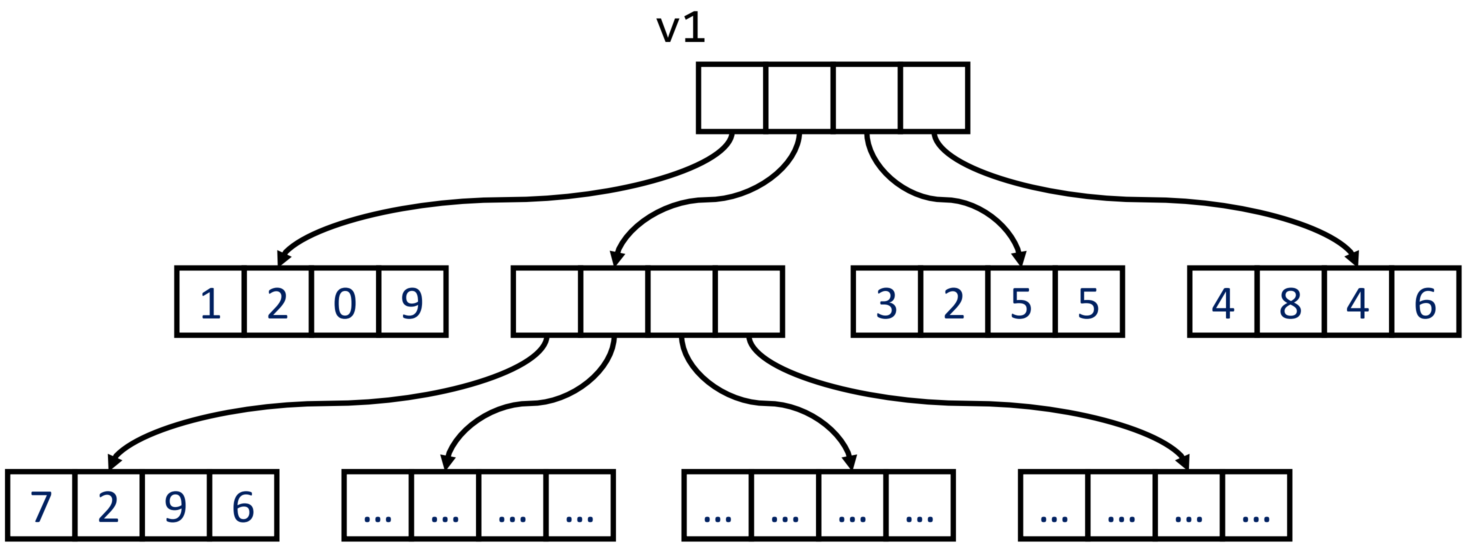


Li Haoyi
@lihaoyi

Consider a large **Vector**, v1:

```
@ val v1 = Vector(1, 2, 0, 9,  7, 2, 9, 6,  …,   3, 2, 5, 5,  4, 8, 4, 6)
```

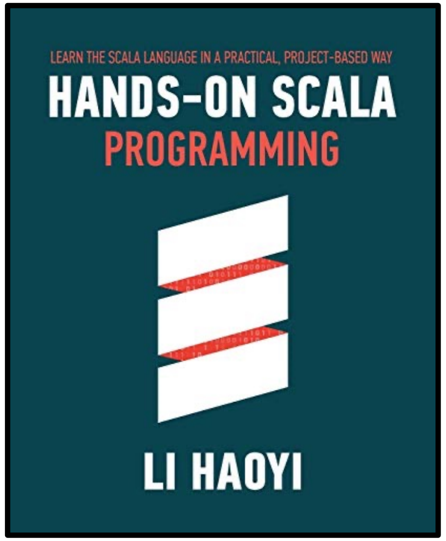
This is represented in-memory as a tree structure, whose breadth and depth depend on the size of the Vector:



This example is somewhat simplified – a Vector in Scala has 32 elements per tree node rather than the 4 shown above – but it will serve us well enough to illustrate how the **Vector** data structure works.

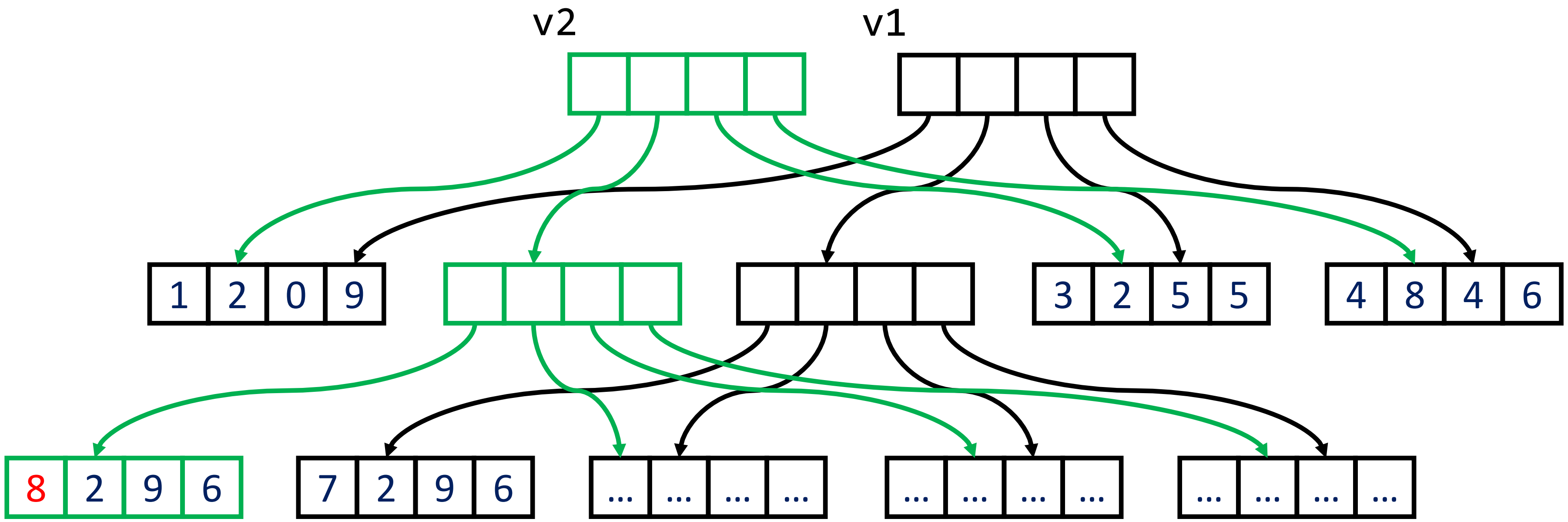
Let us consider **what happens if we want to perform an update**, e.g. replacing the fifth value 7 in the above **Vector** with the value 8:

```
@ val v2 = v1.updated(4, 8)

@ v2
res50: Vector[Int] = Vector(1, 2, 0, 9,  8, 2, 9, 6,  …,   3, 2, 5, 5,  4, 8, 4, 6)
```



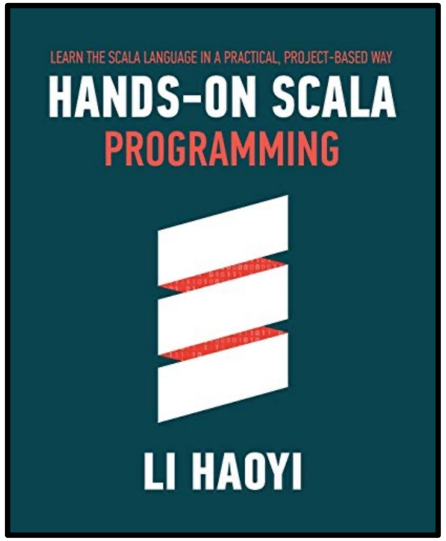
Li Haoyi
@lihaoyi

This is done by making updated copies of the nodes in the tree that are in the direct path down to the value we wish to update, but re-using all other nodes unchanged:



In this example Vector with 9 nodes, only 3 of the nodes end up needing to be copied. In a large Vector, the number of nodes that need to be copied is proportional to the height of the tree, while other nodes can be re-used: this structural sharing is what allows updated copies of the Vector to be created in only *O(log n)* time. This is much less than the *O(n)* time it takes to make a full copy of a mutable Array or other data structure.

Nevertheless, updating a Vector does always involve a certain amount of copying, and will never be as fast as updating mutable data structures in-place. In some cases where performance is important and you are updating a collection very frequently, you might consider using a mutable ArrayDeque (4.3.1), which has faster *O(1)* update/append/prepend operations, or raw Arrays, if you know the size of your collection in advance.



Li Haoyi
@lihaoyi

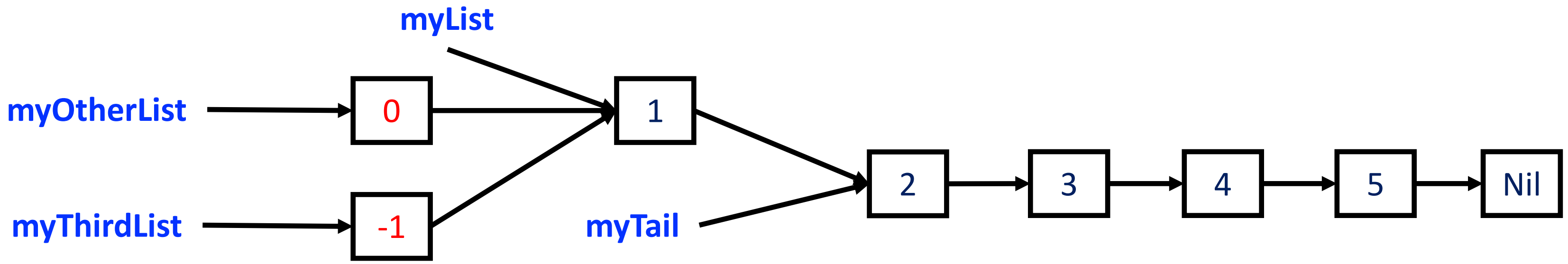
4.2.5 Immutable Lists.

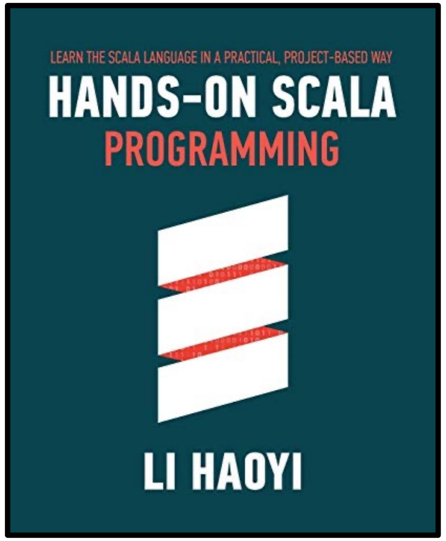
…

Scala's immutable Lists are a singly-linked data structure. Each node in the List has a value and a pointer to the next node, terminating in a Nil node. Lists have a fast *O(1)* .head method to look up the first item in the list, a fast *O(1)* .tail method to create a list without the first element, and a fast *O(1)* :: operator to create a new list with one more element in front.

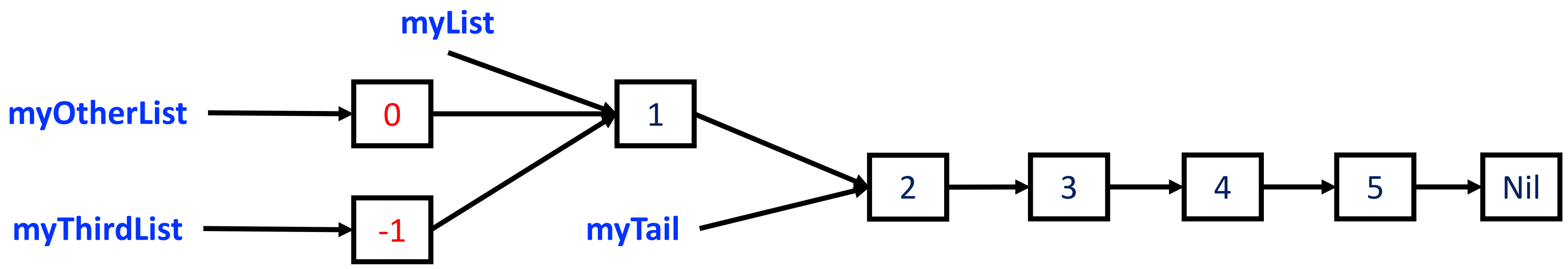
```
scala> val myList = List(1, 2, 3, 4, 5)
val myList: List[Int] = List(1, 2, 3, 4, 5)

scala> myList.head
val res0: Int = 1

scala> val myTail = myList.tail
val myTail: List[Int] = List(2, 3, 4, 5)

scala> val myOtherList = 0 :: myList
val myOtherList: List[Int] = List(0, 1, 2, 3, 4, 5)

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

.tail and :: are efficient because they can share much of the existing List: .tail returns a reference to the next node in the singly-linked structure, while :: adds a new node in front. The fact that multiple lists can share nodes means that in the above example, myList, myTail, myOtherList and myThirdList are actually mostly the same data structure:





Li Haoyi
@lihaoyi



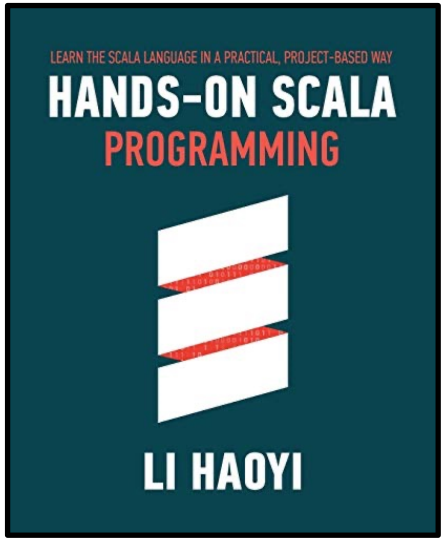
This can result in significant memory savings if you have a large number of collections that have identical elements on one side, e.g. paths on a file system which all share the same prefix.

Rather than creating an updated copy of an Array in *O(n)* time, or an updated copy of a Vector in *O(log n)* time, prepending an item to a List is a fast *O(1)* operation.

The downside of Lists is that indexed lookup via myList(i) is a slow *O(n)* operation, since you need to traverse the list starting from the left to find the element you want.

Appending/removing elements on the right hand side of the list is also a slow *O(n)*, since it needs to make a copy of the entire list.

For use cases where you want fast indexed lookup or fast appends/removes on the right, you should consider using Vectors (4.2.1) or mutable ArrayDeques (4.3.1) instead.



Li Haoyi
@lihaoyi

@philip_schwarz

In the rest of this deck we are going to consider all the possible pairs of collections drawn from the following list

- `List`
- `Vector`
- `ListBuffer`
- `ArrayBuffer`
- `Array`

and then for each such pair, we are going to see a single slide providing a quick visual reminder of the differences between the time complexities of the following operations, as implemented by the pair's two collections:

- `head` `head xs`
- `tail` `tail xs`
- `apply` `xs(i)`
- `update` `xs(i) = x`
- `prepend` `x +: xs`
- `append` `xs :+ x`
- `insert` `xs.insert(i,x)`

In the upcoming tables we indicate the various time complexities as follows

Scala Documentation	Scala Cookbook	**This Slide Deck**	Description	**Notes**
C	Con	$O(1)$	The operation takes (fast) constant time.	
eC	eC	$O(\log n)$	The operation takes effectively constant time, but this might depend on some assumptions, such as maximum length of a vector, or distribution of hash keys.	We use this for **Vector**'s effectively constant operations.
aC	aC	amortized $O(1)$	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.	We use this for **ArrayBuffer**'s append operation, whose time complexity is amortized constant time.
Log	Log	N/A	The operation takes time proportional to the logarithm of the collection size.	We don't need this.
L	Lin	$O(n)$	The operation is linear, so the time is proportional to the collection size.	
-	-	-	The operation is not supported.	

The Scala documentation is from https://docs.scala-lang.org/overviews/collections-2.13/performance-characteristics.html

			head	tail	apply	update	prepend	append	insert
List	Linear	Immutable	O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
Vector	Indexed	Immutable	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-

The **apply**, **update** and **append** operations of a **Vector** have much better time complexity profiles than those of a **List**.

head	tail	apply	update	prepend	append	insert
O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-

The **head**, **tail** and **prepend** operations of a **Vector** have time complexity profiles that are not much worse than that those of a **List**.

head	tail	apply	update	prepend	append	insert
O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-

The operations of a **Vector** have complexity profiles that are a good compromise over those of a **List**.

head	tail	apply	update	prepend	append	insert
O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-

Neither **List** nor **Vector** support insertions.

head	tail	apply	update	prepend	append	insert
O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-

			head	tail	apply	update	prepend	append	insert
List	Linear	Immutable	O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
ListBuffer	Linear	Mutable	O(1)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)

The **head**, **apply**, **update** and **prepend** operations of **List** have the same time complexity profile as those of **ListBuffer**.

head	tail	apply	update	prepend	append	insert
O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
O(1)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)

In **List**, **tail** is **constant** and **append** is **linear**. In **ListBuffer** it is the other way round.

head	tail	apply	update	prepend	append	insert
O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
O(1)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)

The size of **ListBuffer** can change, so it allows insertions.

head	tail	apply	update	prepend	append	insert
O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
O(1)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)

			head	tail	apply	update	prepend	append	insert
Vector	Indexed	Immutable	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-
ArrayBuffer	Indexed	Mutable	O(1)	O(n)	O(1)	O(1)	O(n)	amort O(1)	O(n)

The **head**, **random-access**, **update** and **append** operations of an **ArrayBuffer** have better time complexity profiles than those of a **Vector**.

head	tail	apply	update	prepend	append	insert
O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-
O(1)	O(n)	O(1)	O(1)	O(n)	amort O(1)	O(n)

The **tail** and **prepend** operations of an **ArrayBuffer** have much worse time complexity profiles than those of a **Vector**.

head	tail	apply	update	prepend	append	insert
O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-
O(1)	O(n)	O(1)	O(1)	O(n)	amort O(1)	O(n)

The size of an **ArrayBuffer** can change, so it allows insertions.

head	tail	apply	update	prepend	append	insert
O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-
O(1)	O(n)	O(1)	O(1)	O(n)	amort O(1)	O(n)

			head	tail	apply	update	prepend	append	insert
ListBuffer	Linear	Mutable	O(1)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)
ArrayBuffer	Indexed	Mutable	O(1)	O(n)	O(1)	O(1)	O(n)	amort O(1)	O(n)

The **head** and **tail** operations of **ArrayBuffer** have the same time complexity profiles as those of **ListBuffer**.

head	tail	apply	update	prepend	append	insert
O(1)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)
O(1)	O(n)	O(1)	O(1)	O(n)	amort O(1)	O(n)

The **random-access** and **update** operations of **ArrayBuffer** have much better time complexity profiles than those of **ListBuffer**.

head	tail	apply	update	prepend	append	insert
O(1)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)
O(1)	O(n)	O(1)	O(1)	O(n)	amort O(1)	O(n)

The prepend operation of **ListBuffer** has a much better time complexity profile than that of **ArrayBuffer**.

head	tail	apply	update	prepend	append	insert
O(1)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)
O(1)	O(n)	O(1)	O(1)	O(n)	amort O(1)	O(n)

ListBuffer and **ArrayBuffer** both allow insertions since their size is allowed to change.

head	tail	apply	update	prepend	append	insert
O(1)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)
O(1)	O(n)	O(1)	O(1)	O(n)	amort O(1)	O(n)

			head	tail	apply	update	prepend	append	insert
List	Linear	Immutable	O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
Array	Indexed	Mutable	O(1)	O(n)	O(1)	O(1)	-	-	-

The **head** operation of **Array** has the same complexity profile as that of **List**.

head	tail	apply	update	prepend	append	insert
O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
O(1)	O(n)	O(1)	O(1)	-	-	-

In **List**, **tail** is **constant** whereas **apply** and **update** are **linear**. In **Array** it is the other way round.

head	tail	apply	update	prepend	append	insert
O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
O(1)	O(n)	O(1)	O(1)	-	-	-

While the size of an **Array** is fixed, so it cannot support **insertion**, the size of a **List** can grow, and yet it also does not support **insertion**.

head	tail	apply	update	prepend	append	insert
O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
O(1)	O(n)	O(1)	O(1)	-	-	-

While a **List** supports **prepending** and **appending**, the size of an **Array** is fixed, and so it does support them.

head	tail	apply	update	prepend	append	insert
O(1)	O(1)	O(n)	O(n)	O(1)	O(n)	-
O(1)	O(n)	O(1)	O(1)	-	-	-

			head	tail	apply	update	prepend	append	insert
Vector	Indexed	Immutable	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-
Array	Indexed	Mutable	O(1)	O(n)	O(1)	O(1)	-	-	-

While **Vector** supports **prepend** and **append**, **Array** does not, since its size cannot change.

head	tail	apply	update	prepend	append	insert
O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-
O(1)	O(n)	O(1)	O(1)	-	-	-

The **random-access** and **update** operations of **Array** have better time complexity profiles than those of **Vector**.

head	tail	apply	update	prepend	append	insert
O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-
O(1)	O(n)	O(1)	O(1)	-	-	-

The **tail** operation of **Array** has a much worse time complexity profiles than that of **Vector**.

head	tail	apply	update	prepend	append	insert
O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-
O(1)	O(n)	O(1)	O(1)	-	-	-

Neither **Vector** nor **Array** allow insertions.

head	tail	apply	update	prepend	append	insert
O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	-
O(1)	O(n)	O(1)	O(1)	-	-	-

			head	tail	apply	update	prepend	append	insert
ArrayBuffer	Indexed	Mutable	O(1)	O(n)	O(1)	O(1)	O(n)	amort O(1)	O(n)
Array	Indexed	Mutable	O(1)	O(n)	O(1)	O(1)	-	-	-

The **head**, **tail**, **random-access** and **update** operations of **Array** have the same time complexity profiles as those of **ArrayBuffer**.

head	tail	apply	update	prepend	append	insert
O(1)	O(n)	O(1)	O(1)	O(n)	amort O(1)	O(n)
O(1)	O(n)	O(1)	O(1)	-	-	-

While **ArrayBuffer** supports **prepending**, **appending** and **inserting**, **Array** has a fixed size, and so it does not support those operations.

head	tail	apply	update	prepend	append	insert
O(1)	O(n)	O(1)	O(1)	O(n)	amort O(1)	O(n)
O(1)	O(n)	O(1)	O(1)	-	-	-

			head	tail	apply	update	prepend	append	insert
ListBuffer	Linear	Mutable	O(1)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)
Array	Indexed	Mutable	O(1)	O(n)	O(1)	O(1)	-	-	-

The **head** and **tail** operations of **Array** have the same time complexity profiles as those of **ListBuffer**.

head	tail	apply	update	prepend	append	insert
O(1)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)
O(1)	O(n)	O(1)	O(1)	-	-	-

The **random-access** and **update** operations of **Array** have much better time complexity profiles than those of **ListBuffer**.

head	tail	apply	update	prepend	append	insert
O(1)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)
O(1)	O(n)	O(1)	O(1)	-	-	-

While **ListBuffer** supports **prepending**, **appending** and **inserting**, **Array** has a fixed size, and so it does not support those operations.

head	tail	apply	update	prepend	append	insert
O(1)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)
O(1)	O(n)	O(1)	O(1)	-	-	-

That’s all. I hope you found it useful!

@philip_schwarz