**COLLECTIONS**

**CONCRETE MUTABLE COLLECTION CLASSES**

You’ve now seen the most commonly used immutable collection classes that Scala provides in its standard library. Take a look now at the mutable collection classes.

Array Buffers

An [ArrayBuffer](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/ArrayBuffer.html) buffer holds an array and a size. Most operations on an array buffer have the same speed as for an array, because the operations simply access and modify the underlying array. Additionally, array buffers can have data efficiently added to the end. Appending an item to an array buffer takes amortized constant time. Thus, array buffers are useful for efficiently building up a large collection whenever the new items are always added to the end.

scala> **val** buf = scala.collection.mutable.**ArrayBuffer**.empty[**Int**]

buf: scala.collection.mutable.**ArrayBuffer**[**Int**] = **ArrayBuffer**()

scala> buf += 1

res32: buf.**type** = **ArrayBuffer**(1)

scala> buf += 10

res33: buf.**type** = **ArrayBuffer**(1, 10)

scala> buf.toArray

res34: **Array**[**Int**] = **Array**(1, 10)

List Buffers

A [ListBuffer](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/ListBuffer.html) is like an array buffer except that it uses a linked list internally instead of an array. If you plan to convert the buffer to a list once it is built up, use a list buffer instead of an array buffer.

scala> **val** buf = scala.collection.mutable.**ListBuffer**.empty[**Int**]

buf: scala.collection.mutable.**ListBuffer**[**Int**] = **ListBuffer**()

scala> buf += 1

res35: buf.**type** = **ListBuffer**(1)

scala> buf += 10

res36: buf.**type** = **ListBuffer**(1, 10)

scala> buf.toList

res37: **List**[**Int**] = **List**(1, 10)

StringBuilders

Just like an array buffer is useful for building arrays, and a list buffer is useful for building lists, a [StringBuilder](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/StringBuilder.html) is useful for building strings. String builders are so commonly used that they are already imported into the default namespace. Create them with a simple new StringBuilder, like this:

scala> **val** buf = **new** **StringBuilder**

buf: **StringBuilder** =

scala> buf += 'a'

res38: buf.**type** = a

scala> buf ++= "bcdef"

res39: buf.**type** = abcdef

scala> buf.toString

res41: **String** = abcdef

Linked Lists

Linked lists are mutable sequences that consist of nodes which are linked with next pointers. They are supported by class [LinkedList](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/LinkedList.html). In most languages null would be picked as the empty linked list. That does not work for Scala collections, because even empty sequences must support all sequence methods. In particular LinkedList.empty.isEmpty should return true and not throw a NullPointerException. Empty linked lists are encoded instead in a special way: Their next field points back to the node itself. Like their immutable cousins, linked lists are best traversed sequentially. In addition linked lists make it easy to insert an element or linked list into another linked list.

Double Linked Lists

Double linked lists are like single linked lists, except that they have besides nextanother mutable field prev that points to the element preceding the current node. The main benefit of that additional link is that it makes element removal very fast. Double linked lists are supported by class [DoubleLinkedList](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/DoubleLinkedList.html).

Mutable Lists

A [MutableList](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/MutableList.html) consists of a single linked list together with a pointer that refers to the terminal empty node of that list. This makes list append a constant time operation because it avoids having to traverse the list in search for its terminal node. [MutableList](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/MutableList.html) is currently the standard implementation of [mutable.LinearSeq](http://www.scala-lang.org/api/2.12.2/scala/collection/LinearSeq.html)in Scala.

Queues

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

scala> **val** queue = **new** scala.collection.mutable.**Queue**[**String**]

queue: scala.collection.mutable.**Queue**[**String**] = **Queue**()

scala> queue += "a"

res10: queue.**type** = **Queue**(a)

scala> queue ++= **List**("b", "c")

res11: queue.**type** = **Queue**(a, b, c)

scala> queue

res12: scala.collection.mutable.**Queue**[**String**] = **Queue**(a, b, c)

scala> queue.dequeue

res13: **String** = a

scala> queue

res14: scala.collection.mutable.**Queue**[**String**] = **Queue**(b, c)

Array Sequences

Array sequences are mutable sequences of fixed size which store their elements internally in an Array[Object]. They are implemented in Scala by class [ArraySeq](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/ArraySeq.html).

You would typically use an ArraySeq if you want an array for its performance characteristics, but you also want to create generic instances of the sequence where you do not know the type of the elements and you do not have a ClassTag to provide it at run-time. These issues are explained in the section on [arrays](https://docs.scala-lang.org/overviews/collections/arrays.html).

Stacks

You saw immutable stacks earlier. There is also a mutable version, supported by class [mutable.Stack](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/Stack.html). It works exactly the same as the immutable version except that modifications happen in place.

scala> **val** stack = **new** scala.collection.mutable.**Stack**[**Int**]

stack: scala.collection.mutable.**Stack**[**Int**] = **Stack**()

scala> stack.push(1)

res0: stack.**type** = **Stack**(1)

scala> stack

res1: scala.collection.mutable.**Stack**[**Int**] = **Stack**(1)

scala> stack.push(2)

res0: stack.**type** = **Stack**(1, 2)

scala> stack

res3: scala.collection.mutable.**Stack**[**Int**] = **Stack**(1, 2)

scala> stack.top

res8: **Int** = 2

scala> stack

res9: scala.collection.mutable.**Stack**[**Int**] = **Stack**(1, 2)

scala> stack.pop

res10: **Int** = 2

scala> stack

res11: scala.collection.mutable.**Stack**[**Int**] = **Stack**(1)

Array Stacks

[ArrayStack](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/ArrayStack.html) is an alternative implementation of a mutable stack which is backed by an Array that gets re-sized as needed. It provides fast indexing and is generally slightly more efficient for most operations than a normal mutable stack.

Hash Tables

A hash table stores its elements in an underlying array, placing each item at a position in the array determined by the hash code of that item. Adding an element to a hash table takes only constant time, so long as there isn’t already another element in the array that has the same hash code. Hash tables are thus very fast so long as the objects placed in them have a good distribution of hash codes. As a result, the default mutable map and set types in Scala are based on hash tables. You can access them also directly under the names [mutable.HashSet](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/HashSet.html) and [mutable.HashMap](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/HashMap.html).

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

scala> **val** map = scala.collection.mutable.**HashMap**.empty[**Int**,**String**]

map: scala.collection.mutable.**HashMap**[**Int**,**String**] = **Map**()

scala> map += (1 -> "make a web site")

res42: map.**type** = **Map**(1 -> make a web site)

scala> map += (3 -> "profit!")

res43: map.**type** = **Map**(1 -> make a web site, 3 -> profit!)

scala> map(1)

res44: **String** = make a web site

scala> map contains 2

res46: **Boolean** = false

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

Weak Hash Maps

A weak hash map is a special kind of hash map where the garbage collector does not follow links from the map to the keys stored in it. This means that a key and its associated value will disappear from the map if there is no other reference to that key. Weak hash maps are useful for tasks such as caching, where you want to re-use an expensive function’s result if the function is called again on the same key. If keys and function results are stored in a regular hash map, the map could grow without bounds, and no key would ever become garbage. Using a weak hash map avoids this problem. As soon as a key object becomes unreachable, it’s entry is removed from the weak hashmap. Weak hash maps in Scala are implemented by class [WeakHashMap](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/WeakHashMap.html) as a wrapper of an underlying Java implementation java.util.WeakHashMap.

Concurrent Maps

A concurrent map can be accessed by several threads at once. In addition to the usual [Map](http://www.scala-lang.org/api/2.12.2/scala/collection/Map.html) operations, it provides the following atomic operations:

Operations in class ConcurrentMap

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

ConcurrentMap is a trait in the Scala collections library. Currently, its only implementation is Java’s java.util.concurrent.ConcurrentMap, which can be converted automatically into a Scala map using the [standard Java/Scala collection conversions](https://docs.scala-lang.org/overviews/collections/conversions-between-java-and-scala-collections.html).

Mutable Bitsets

A mutable bit of type [mutable.BitSet](http://www.scala-lang.org/api/2.12.2/scala/collection/mutable/BitSet.html) set is just like an immutable one, except that it is modified in place. Mutable bit sets are slightly more efficient at updating than immutable ones, because they don’t have to copy around Longs that haven’t changed.

scala> **val** bits = scala.collection.mutable.**BitSet**.empty

bits: scala.collection.mutable.**BitSet** = **BitSet**()

scala> bits += 1

res49: bits.**type** = **BitSet**(1)

scala> bits += 3

res50: bits.**type** = **BitSet**(1, 3)

scala> bits

res51: scala.collection.mutable.**BitSet** = **BitSet**(1, 3)