Scala 2.8 Collections

Martin Odersky, EPFL

July 20, 2010

1 Introduction

Scala 2.8 contains a major redesign of the collection libraries. This paper describes their API and architecture.

Acknowldegements Several people have shaped the collection design in important ways. The new libraries would not exist without their contributions. Matthias Zenger wrote Scala's original collection libraries for sets, maps, buffers, and other types. Many of his design decisions have survived the redesign. Some have been generalized, such as his partition into mutable and immutable collection packages, which now applies uniformly for all kinds of collections including sequences. Sean McDirmid added projections to the original collection libraries, a concept which has been taken up in the redesign under the name of views. Adriaan Moors developed higher-kinded types in Scala, which gave the primary motivation for the collection redesign, even though in the end their role is more narrow than originally anticipated. Adriaan was also the first to explore builders as a fundamental abstraction for Scala's collections. David McIver gave me the idea of builders as implicit parameters and proposed `Traversable` as a generalization of `Iterable`. Miles Sabin contributed the bidirectional wrappers that convert between Java collections and Scala collections. Phil Bagwell, Gilles Dubochet, Burak Emir, Stepan Koltsov, Stéphane Micheloud, Tony Morris, Jorge Ortiz, Paul Phillips, David Pollak, Tiark Rompf, Lex Spoon, and many others have contributed to specific collection classes or made important suggestions for improvements.

2 Overview

All collection classes are now kept in a package scala.collection. This package has three subpackages: `mutable`, `immutable`, and `generic`. Most collections exist in three forms, depending on their mutability.

A collection in package `scala.collection.immutable` is guaranteed to be immutable for everyone. That means one can rely on the fact that accessing the same collection value over 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.

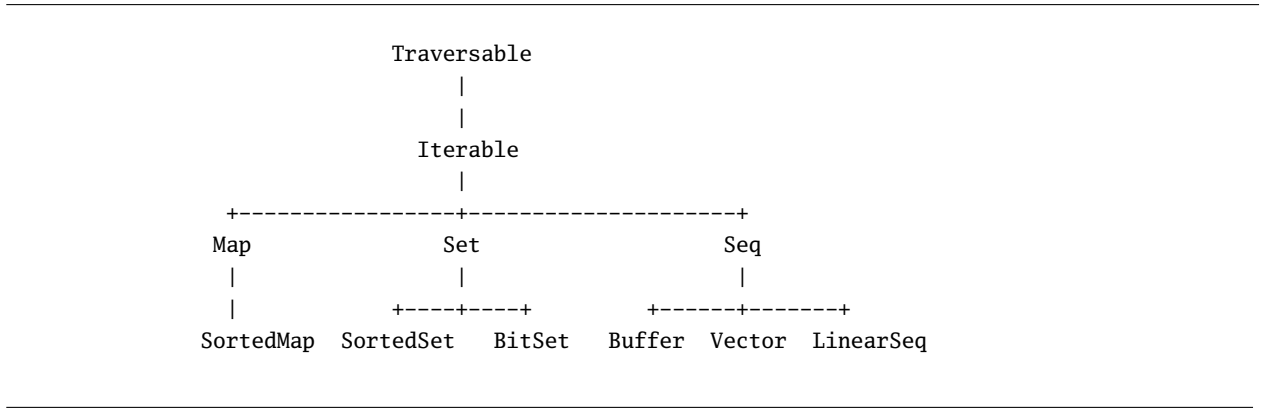

Figure 1: Collection base classes

A collection in package `scala.collection` can be either mutable or immutable. For instance, `collection.Vector[T]` is a superclass of both `collection.immutable.Vector[T]` and `collection.mutable.Vector[T]`. Generally, the root collections in package `scala.collection` define the same interface as the immutable collections, and the mutable collections in package `scala.collection.mutable` typically add some destructive modification operations to this immutable interface. The difference between root collections and immutable collections is that a user of an immutable collection has a guarantee that nobody can mutate the collection, whereas users of root collections have to assume modifications by others, even though they cannot do any modifications themselves.

The `generic` package contains building blocks for implementing various collections. Typically, collection classes defer the implementations of some of their operations to classes in `generic`. Users of the collection framework, on the other hand, should need to refer at classes in `generic` only in exceptional circumstances

3 The Scala Collections API

The most important collection base classes are shown in Figure 1. There is aquite a bit of commonality shared by all these classes. You can form an instance of any collection by writing the class name followed by its element. Examples are:

```
Traversable(1, 2, 3)
Iterable("x", "y", "z")
Map("x" -> 24, "y" -> 25, "z" -> 26)
Set(Color.red, Color.green, Color.blue)
SortedSet("hello", "world")
```

```
Buffer(x, y, z)
Vector(1.0, 2.0)
LinearSeq(a, b, c)
```

The same also holds for specific collection implementations. For instance it's

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

All these collections print out with `toString` the same way they are written above. All collections support the API provided by Traversable, but at their own types. For instance mapping a function with `map` over a list yields again a `List`, mapping it over a set yields again a set and so on. Quality is also organized uniformly for all collection classes; more on this in Section 3.6

As said previously, most of the classes in Figure 1 exist in three variants: base, mutable, and immutable. The only exception is the `Buffer` trait which always describes a mutable collection.

In the following, we will review these classes one by one.

3.1 Trait `Traversable`

At the top of this hierarchy is trait `Traversable`. Its only abstract operation is

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

The operation is meant to traverse all elements of the collection, and apply the given operation `f` to each element. The application is done for its side effect only; in fact any function result of `f` is discarded by `foreach`.

Traversible objects can be finite or infinite. An example of an infinite traversable object is the stream of natural numbers `Stream.from(0)`. The method `hasDefiniteSize` indicates whether a collection is possibly inifinite. If `hasDefiniteSize` returns true, the collection is certainly finite. If it returns false, the collection has not been not fully elaborated yet, so it might be infinite or finite.

Other operations defined on a traversable collection `xs` are defined in Figures 2 and 3

3.2 Trait `Iterable`

The next trait from the top in Figure 1 is `Iterable`. This trait declares an abstract method `iterator` that returns an iterator that yields all the collection's elements one by one. The `foreach` method in `Iterable` is implemented in terms of `iterator`. Subclasses of `Iterable` often override `foreach` with a direct implementation for efficiency.

Class `Iterable` also adds some less-often used methods to `Traversable`, which can be implemented efficiently only if an iterator is available. They are summarized in Figure 4.

After `Iterable` there come three base classes which inherit from it: `Seq`, `Set`, and `Map`. All three have an `apply` method and all three implement the `PartialFunction` trait, but the meaning of `apply` is different in each case.

`xs.isEmpty`	Test whether the collection is empty.
`xs.nonEmpty`	Test whether the collection contains elements.
`xs.size`	The number of elements in the collection.
`xs ++ ys`	A collection consisting of the elements of both `xs` and `ys`. `ys` can be a traversable object or an iterator.
`xs map f`	The collection obtained from applying the function `f` to every element in `xs`.
`xs flatMap f`	The collection obtained from applying the collection-valued function `f` to every element in `xs` and concatenating the results.
`xs filter p`	The collection consisting of those elements of `xs` that satisfy the predicate `p`.
`xs filterNot p`	The collection consisting of those elements of `xs` that do not satisfy the predicate `p`.
`xs partition p`	Split `xs` into a pair of two collections; one with elements that satisfy the predicate `p`, the other with elements that do not.
`xs groupBy f`	Partition `xs` into a map of collections according to a discriminator function `f`.
`xs forall p`	A boolean indicating where the predicate `p` holds for all elements of `xs`.
`xs exists p`	A boolean indicating where the predicate `p` holds for some element in `xs`.
`xs count p`	The number of elements in `xs` that satisfy the predicate `p`.
`xs find p`	An option containing the first element in `xs` that satisfies `p`, or `Non` is no element qualifies.
`(z /: xs)(op)`	Apply binary operation `op` between successive elements of `xs`, going left to right and starting with `z`.
`(xs :\ z)(op)`	Apply binary operation `op` between successive elements of `xs`, going right to left and starting with `z`.
`xs.foldLeft(z)(op)`	Same as `(z /: xs)(op)`.
`xs.foldRight(z)(op)`	Same as `(xs :\ z)(op)`.
`xs reduceLeft op`	Apply binary operation `op` between successive elements of non-empty collection `xs`, going left to right.
`xs reduceRight op`	Apply binary operation `op` between successive elements of non-empty collection `xs`, going right to left.

Figure 2: Operations on `Traversable` collections I.

`xs.head`	The first element of the collection (or, some element, if no order is defined).
`xs.tail`	The rest of the collection except `xs.head`.
`xs.last`	The last element of the collection (or, some element, if no order is defined).
`xs.init`	The rest of the collection except `xs.last`.
`xs take n`	A collection consisting of the first `n` elements of `xs` (or, some arbitrary `n` elements, if no order is defined).
`xs drop n`	The rest of the collection except `xs take n`.
`xs splitAt n`	The pair of collections `(xs take n, xs drop n)`.
`xs slice (from, to)`	A collection consisting of elements in some index range of `xs` (from `from` up to, and excluding `to`).
`xs takeWhile p`	The longest prefix of elements in this collection which all satisfy `p`.
`xs dropWhile p`	This collection without the longest prefix of elements which all satisfy `p`.
`xs span p`	The pair of collections `(xs takeWhile p, xs.dropWhile p)`.
`xs copyToBuffer buf`	Copies all elements of this collection to buffer `buf`.
`xs copyToArray (arr, start, len)`	Copies elements of this collection to array `arr`.
`xs.toArray`	converts this collection to an array.
`xs.toList`	converts this collection to a list.
`xs.toIterable`	converts this collection to an instance of `Iterable`.
`xs.toSeq`	converts this collection to an instance of `Seq`.
`xs.toStream`	converts this collection to a lazily computed stream.
`xs mkString sep`	Produces a string which shows all elements of `xs` between separators `sep` (The method exists also in other overloaded variants).
`xs addString (b, sep)`	Adds a string to `StringBuilder b` which shows all elements of `xs` between separators `sep` (The method exists also in other overloaded variants).
`xs.view`	Produces a view over `xs`.
`xs view (from, to)`	Produces a view that represents the elements in some index range of `xs`.

Figure 3: Operations on `Traversable` collections II.

`xs.iterator`	An iterator that yields every element in `xs`, in the same order as `foreach` traverses elements.
`xs takeRight n`	A collection consisting of the last `n` elements of `xs` (or, some arbitrary `n` elements, if no order is defined).
`xs dropRight n`	The rest of the collection except `xs takeRight n`.
`xs sameElements ys`	A test whether `xs` and `ys` contain the same elements in the same order

Figure 4: Operations on `Iterable` collections.

3.3 Traits `Seq`, `Vector`, `LinearSeq`, and `Buffer`

The operations on sequences are summarized in Figure 5. For a `Seq`, the `apply` operation means indexing; hence sequences are partial functions from `Int` to their element type. The elements of a sequence are indexed from zero up to the `length` of the sequence minus one. Sequences add other methods to `Iterable` as well: The three methods `zip`, `zipAll`, and `zipWithIndex` produce sequences of pairs by combining corresponding elements of two sequences. Another set of methods including `indexOf`, `lastIndexof`, `indexWhere`, `lastIndexWhere` finds the first or last position of a given element, or an element having some given property. Methods `startsWith` and `endsWith` test whether the sequence starts or ends with a given prefix or suffix sequence. New sequences are produced by methods `reverse`, `patch`, and `padTo`. Methods `union`, `intersection` and `diff` operate on sequences seen as ordered multisets. Method `removeDuplicates` removes duplicate elements from a sequence.

If a sequence is mutable, it offers in addition an `update` method, that lets sequence elements be updated using syntax like `seq(idx) = elem`.

The `Seq` classes each have two subclasses, `LinearSeq`, and `Vector`. These do not add any new operations, but each offers different performance characteristics: A linear sequence has efficient `head` and `tail` operations, whereas a vector has efficient `apply`, `length`, and (if mutable) `update` operations. Frequently used linear sequences are `scala.collection.immutable.List` and `scala.collection.immutable.Stream`. Frequently used vectors are `scala.Array` and `scala.collection.mutable.ArrayBuffer` (we are waiting for the integration of some more refined implementations of immutable vectors into the library).

Another sub-category of sequences are `Buffers`. Buffers are always mutable. They 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 `buf += elem` and `buf ++= elems` for element addition at the end, `elem +: buf` and `elems ++: buf` for addition at the front, `insert` and `insertAll` for element insertions, as well as `remove` and `-=` for element removal. These operations are summarized in Figure 6.

Two often used implementations of buffers are `scala.collection.mutable.ListBuffer` and

`xs.length`	The length of the sequence (same as `size`).
`xs.lengthCompare ys`	Returns `-1` if `xs` is shorter than `ys`, `+1` if it is longer and `0` is they have the same length. Works even if one if the sequences is infinite.
`xs(i)`	(or, written out, `xs apply i`). The element of `xs` at index `i`.
`xs.indices`	The index range of `xs`, extending from 0 to `xs.length - 1`.
`xs isDefinedAt i`	A test whether `i` is contained in `xs.indices`.
`xs zip ys`	A sequence of pairs of corresponding elements from `xs` and `ys`.
`xs zipAll (ys, x, y)`	A sequence of pairs of corresponding elements from `xs` and `ys`, where the shorter sequence is extended to match the longer one by appending elements `x` or `y`.
`xs.zipWithIndex`	A sequence of pairs of elements from `xs` with their indicies.
`xs segmentLength (p, i)`	The length of the longest uninterrupted segment of elements in `xs`, starting with `xs(i)`, that all satisfy the predicate `p`.
`xs prefixLength p`	The length of the longest prefix of elements in `xs` that all satisfy the predicate `p`.
`xs indexWhere p`	The index of the first element in `xs` that satisfies `p`. (several variants exist).
`xs indexOf x`	The index of the first element in `xs` equal to `x`. (several variants exist).
`xs.reverse`	A sequence with the elements of `xs` in reversed order.
`xs.reverseIterator`	An iterator yielding all the elements of `xs` in reversed order.
`xs startsWith ys`	A test whether `xs` has sequence `ys` as a prefix. (several variants exist).
`xs contains x`	A test whether `xs` has an element equal to `x`.
`xs intersect ys`	The multi-set intersection of sequences `xs` and `ys` which preserves the order of elements in `xs`.
`xs diff ys`	The multi-set difference of sequences `xs` and `ys` which preserves the order of elements in `xs`.
`xs union ys`	Multiset union; same as `xs ++ ys`.
`xs.removeDuplicates`	A subsequence of `xs` that contains no duplicated element.
`xs patch (i, ys, r)`	The sequence resulting from `xs` by replacing `r` elements starting with `i` by the patch `ys`.
`xs padTo (len, x)`	The sequence resulting from `xs` by appending the value `x` until length `len` is reached.

Figure 5: Operations on `Seq` collections.

`buf += x`	Append element `x` to buffer, and return `buf` itself as result.
`buf += (x, y, z)`	Append given elements to buffer.
`buf ++= xs`	Append all elements in `xs` to buffer.
`x +: buf`	Prepend element `x` at start of buffer.
`xs +: buf`	Prepend all elements in `xs` at start of buffer.
`buf insert (i, x)`	Insert element `x` at index `i` in buffer.
`buf insertAll (i, xs)`	Insert all elements in `xs` at index `i` in buffer.
`buf -= x`	Remove element `x` from buffer.
`buf remove i`	Remove element at index `i` from buffer.
`buf remove (i, n)`	Remove `n` elements starting at index `i` from buffer.
`buf trimStart n`	Remove first `n` elements from buffer.
`buf trimEnd n`	Remove last `n` elements from buffer.
`buf.clear()`	Remove all elements from buffer.
`buf.clone`	A new buffer with the same elements as `buf`.

Figure 6: Operations on buffers.

`scala.collection.mutable.ArrayBuffers`. As the name implies, a `ListBuffer` is backed by a `List`, and supports efficient conversion of its element to a `List`, whereas an `ArrayBuffer` is backed by an array, and can be quickly converted into one.

3.4 The `Set` traits

`Sets` are `Iterables` that contain no duplicate elements. The operations on sets are summarized in Figure 7 for general sets and Figure 8 for mutable sets.

The `contains` method asks whether a set contains a given element. The `apply` method for a set is the same as `contains`, so `set(elem)` is the same as `set contains elem`. That means sets can also be used as test functions that return true for the elements they contain.

How are elements added or removed from a set? It depends whether the set is mutable or immutable. For immutable sets, there's methods + and -. The operation `s + elem` returns a new set that contains all `elem` as well as all elements of set `s`. Analogously, `s - elem` returns a new set that contains all elements of set `s` except `elem`.

These operations also work for mutable sets, but they are less often used there since they involve copying the set `s`. As a more efficient alternative, mutable offer the mutation methods += and -=. The operation `s += elem` adds `elem` to the set `s` as a side effect, and returns the mutated set as a result. Likewise, `s -= elem` removes `elem` from the set, and returns that set as a result.

All operations also allow adding or removing more than one key from a set. For instance:

```
Set(1, 2) == Set() + (1, 2)
```

`xs contains x, xs(x)`	Test whether x is an element of xs.
`xs + x`	The set containing all elements of xs as well as x.
`xs + (x, y, z)`	The set containing all elements of xs as well as with the given additional elements.
`xs ++ ys`	The set containing all elements of xs as well as all elements of ys.
`xs - x`	The set containing all elements of xs except for x.
`xs - (x, y, z)`	The set containing all elements of xs except for the given elements.
`xs - ys`	The set containing all elements of xs except for the elements of ys.
`xs & ys, xs intersect ys`	The set intersection of xs and ys.
`xs | ys, xs union ys`	The set union of xs and ys.
`xs &~ ys, xs diff ys`	The set difference of xs and ys.
`xs subsetof ys`	Test whether xs is a subset of ys.
xs.empty	An empty set of the same class as xs.

Figure 7: Operations on sets.

```
Set(1, 2) - (1, 2) == Set()
```

Finally, there are also the bulk operations ++, –, or ++=, –= for mutable sets, that add or remove all elements in their right-hand argument, which is a traversable collection.

Typical use cases of these operations are shown in the following example:

```
// for immutable sets:
var cities = immutable.Set[String]()
cities = cities + "Paris" + "Dakar"
cities = cities + ("London", "Moscow", "New York")
cities = cities - "Dakar"
cities = cities ++ cities // still the same

// for mutable sets:
val mcities = mutable.Set[String]()
mcities += "Paris" += "Dakar"
mcities = mcities + "Paris" + "Dakar" // also possible, but involves two clonings
mcities += ("London", "Moscow", "New York")
mcities -= "Dakar"
mcities -= cities // produces an empty set
```

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

Figure 8: Operations on mutable sets.

Besides these operations, new sets can also be formed through intersection, union, or set difference from existing sets. These operations exist in two different forms: alphabetic and symbolic. The alphabetic versions are `union`, `intersect`, `diff`, whereas the symbolic versions are `|`, `&`, `&~` . In fact `++` can be seen as yet another alias of `union` or `|`, except that `++` takes a `Traversable` argument whereas `union` and `|` take sets.

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

Two subtraits of sets are `SortedSet` and `BitSet`. A `SortedSet` is a set that produces its elements (using `iterator` or `foreach` in a given ordering (which can be freely chosen). Sorted sets also support ranges of all elements that lie between a pair of given elements in the ordering. The result of such a range is again a sorted set.

Bitsets are sets of non-negative integer elements that are implemented in one or more words of packed bits. They support the same operations as normal sets, but consume less space if the range of possible element values is small (Logically, bitsets should be sorted, but this has not yet been implemented).

`ms get k`	The value associated with key `k` in map `ms` as an option, `None` if not found.
`ms(k)`	(or, written out, `ms apply k`) The value associated with key `k` in map `ms`, exception if not found.
`ms getOrElse (k, d)`	The value associated with key `k` in map `ms`, or the default value `d` if not found.
`ms contains k`	Test whether `ms` contains a mapping for key `k`.
`ms isDefinedAt k`	Same as `contains`.
`ms + (k -> v)`	The map containing all mappings of `ms` as well as the mapping `k -> v` from key `k` to value `v`.
`ms + (k -> v, l -> w)`	The map containing all mappings of `ms` as well as the given key/value pairs.
`ms ++ kvs`	The map containing all mappings of `ms` as well as all key/value pairs of `kvs`.
`ms - k`	The map containing all mappings of `ms` except for any mapping of key `k`.
`ms - (k, l, m)`	The map containing all mappings of `ms` except for any mapping with the given keys.
`ms - ks`	The map containing all mappings of `ms` except for any mapping with a key in `ks`.
`ms updated (k, v)`	Same as `ms + (k -> v)`.
`ms.keysIterator`	An iterator yielding each key in `ms`.
`ms.keys`	A set containing each key in `ms`.
`ms.valuesIterator`	An iterator yielding each value associated with a key in `ms`.
`ms.values`	A set containing each value associated with a key in `ms`.
`ms filterKeys p`	A map view containing only those mappings in `ms` where the key satisfies predicate `p`.
`ms mapValues f`	A map view resulting from applying function `f` to each value associated with a key in `ms`.

Figure 9: Operations on maps.

3.5 The `Map` traits

`Maps` are `Iterables` of pairs of keys and values (also named *mappings* or *associations*). Scala's `Predef` class offers an implicit conversion that lets one write `key -> value` as an alternate syntax for the pair `(key, value)`. For instance `Map("x" -> 24, "y" -> 25, "z" -> 26)` means exactly the same as `Map(("x", 24), ("y", 25), ("z", 26))`, but reads better.

`ms(k) = v`	(or, written out, `ms.update(x, v)`). Adds mapping from key `k` to value `v` to map `ms` as a side effect, overwriting any previous mapping of `k`.
`ms += (k -> v)`	Adds mapping from key `k` to value `v` to map `ms` as a side effect and returns `ms` itself.
`ms += (k -> v, l -> w)`	Adds the given mappings to `ms` as a side effect and returns `ms` itself.
`ms ++= kvs`	Adds all mappings in `kvs` to `ms` as a side effect and returns `ms` itself.
`ms -= k`	Removes mapping with key `k` from `ms` as a side effect and returns `ms` itself.
`ms -= (k, l, m)`	Removes mappings with the given keys from `ms` as a side effect and returns `ms` itself.
`ms -= ks`	Removes all keys in `ks` from `ms` as a side effect and returns `ms` itself.
`ms put (k, v)`	Adds mapping from key `k` to value `v` to `ms` and returns any value previously associated with `k` as an option.
`ms remove k`	Removes any mapping with key `k` from `ms` and returns any value previously associated with `k` as an option.
`ms getOrElseUpdate (k, d)`	If key `k` is defined in map `ms`, return its associated value. Otherwise, update `ms` with the mapping `k -> d` and return `d`.
`ms retain p`	Keep only those mappings in `ms` that have a key satisfying predicate `p`.
`ms transform f`	Transform all associated values in map `ms` with function `f`.
`ms.clear()`	Remove all mappings from `ms`.
`ms.clone`	A new mutable map with the same mappings as `ms`.

Figure 10: Operations on mutable maps.

The fundamental operations on maps are similar to those on sets. They are summarized in Figure 9 for general maps and Figure 10 for mutable maps. Instead of `contains`, the fundamental lookup method is

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

The operation `m get key` tests whether the map contains an association for the given `key`. If yes, it returns the associated value in a `Some`. If no `key` is defined in the map, `get` returns `None`.

Maps also define an `apply` method that returns the value associated with a given key directly, without wrapping it in an `Option`. If the key is not defined in the map, an exception is raised.

The addition and removal operations for maps mirror those for sets. For an immutable map, new mappings can be added with + and keys can be removed with –. The result is in each case a new immutable map. For a mutable map, the corresponding operations are += and –=. These latter operations are side-effecting and return the map itself as a result. Examples:

```
// For immutable maps:
var capitals = immutable.Map[String, String]()
capitals = capitals + ("France" -> "Paris", "Senegal" -> "Dakar")
capitals = capitals - "France"

// For mutable maps:
val mcapitals = mutable.Set[String, String]()
mcapitals += ("France -> "Paris") += ("Senegal" -> "Dakar")
mcapitals -= "France"
mcapitals ++= capitals // (no change)
```

Some variants of these addition and removal operations are also supported. Immutable maps support `updated`, with the following definition:

```
def updated (key: Key, value: Value): Map[Key, Value] = this + ((key, value))
```

That is, `updated` adds a key/value association to a map without forming a pair first. This can give some speedups for compute-intensive collection code, provided a map implements `update` directly.

Mutable maps support `update` instead of `updated` as a mutation operator, which has the following definition:

```
def update(key: Key, value: Value) { this += ((key, value)) }
```

This method can give the same speed advantages for mutable collections as `updated` gives for immutable collections. In addition `update` allows to modify collections with an assignment-like syntax. Here's an example.

```
capitals("Germany") = "Berlin"
capitals.update("Germany", "Berlin")
capitals += ("Germany" -> "Berlin")
```

In the code above, the assignment on the first line is syntactic sugar which is exactly the same to the invocation of the `update` method on the second line. This invocation in turn is functionally equivalent, but potentially slighly faster, than the += invocation on the last line.

Mutable maps also provide `put` and `remove` as variants of += and –=. The difference is that `put` and `remove` return an `Option` result that contains the value previously associated in the map with the given key, or `None` if the key was previously undefined.

Besides `iterator`, which gives an iterator of key/value pairs, maps also let one inspect keys and values separately. `m.keysIterator` produces an iterator of all keys in `m` whereas `m.valuesIterator` produces an iterator of all values. One can also access a map's keys and values as sets, using `m.keys` and `m.values`.

3.6 Equality

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

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

```
val map = HashMap[Array[Int], Int]()
val xs = Array(1, 2, 3)
map += (xs -> 6)
xs(0) = 0
map(xs)
```

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

3.7 Creating sequences

Collection classes also provide a uniform interface for creating new collections from scratch: `C.empty` gives for any collection class `C` the empty collection of that class. `C(`x_1 `,...,` x_n`)` gives a collection with elements x_1 ,..., x_n. Examples are:

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

Sequence classes provide in addition a set of other creation operations. These are summarized in Figure 11. The `concat` operation concatenates an arbitrary number of traversables together. The `fill` and `tabulate` methods generate single or multi-dimensional sequences of given dimensions initialized by some expression or tabulating function. The `range` methods generate integer sequences with some constant step length. The `iterate` method generates the sequence resulting from repeated application of a function to a start element.

`S.empty`	The empty sequence.
`S(x, y, z)`	A sequence consisting of elements `x`, `y`, `z`.
`S.concat(xs, ys, zs)`	The sequence obtained by concatenating the elements of `xs`, `ys`, `zs`.
`S.fill(n){e}`	A sequence of length `n` where each element is computed by expression `e`.
`S.fill(m, n){e}`	A sequence of sequences of dimension `m×n` where each element is computed by expression `e`. (exists also in higher dimensions).
`S.tabulate(n){f}`	A sequence of length `n` where the element at each index i is computed by `f(`i`)`.
`S.tabulate(m, n){f}`	A sequence of sequences of dimension `m×n` where the element at each index (i, j) is computed by `f(`i`, `j`)`. (exists also in higher dimensions).
`S.range(start, end)`	The sequence of integers `start` ... `end-1`.
`S.range(start, end, step)`	The sequence of integers starting with `start` and progressing by `step` increments up to, and excluding, the `end` value.
`S.iterate(x, n)(f)`	The sequence of length `n` with elements `x`, `f(x)`, `f(f(x))`, ...

Figure 11: Creating instances of any sequence collection class `S`

4 The Scala Collections Framework

The previous section has enumerated a large number of collection operations, which exist universally on several different kinds of collections, and an even larger number of collection implementations. Implementing every collection operation anew for every collection type would lead to an excessive amount of code duplication. Such code duplication leads to inconsistencies over time, when an operation is added or modified in one part of the collection libraries but not in others. The principal design objective of the new collections framework was to avoid any duplication, defining every operation in one place only. The design approach was to implement most operations in collection templates which can be flexibly inherited from individual base classes and implementations. These template classes are all defined in package `scala.collection.generic`. This section explains the most important classes and the construction principles they support.

4.1 Builders

Most collection operations are implemented in terms of traversals and builders. Traversals are handled by `Traversable`'s `foreach` method, and building new collections is handled by instances of class `Builder`.

```
package scala.collection.generic
class Builder[-Elem, +To] {
  def +=(elem: Elem): this.type
  def result(): To
  def clear()
  def mapResult(f: To => NewTo): Builder[Elem, NewTo] = ...
}
```

Figure 12: An outline of the `Builder` class.

Figure 12 presents a slightly simplified outline of this class.

One can add an element `x` to a builder `b` with `b += x`. There's also syntax to add more than one element at once, for instance `b += (x, y)`, `b ++= xs` all work as for buffers (in fact, buffers are an enriched version of builders). The `result()` method returns a collection from a builder. The state of the builder is undefined after taking its result, but it can be reset into a new empty state using `clear()`. Builders are generic in both the element type `Elem` and in the type `To` of collections they return.

Often, a builder can refer to some other builder for assembling the elements of a collection, but then would like to transform the result of the other builder, to give a different type, say. This task is simplified by the method `mapResult` in class `Builder`. For instance, assuming a builder `bldr` of `ArrayBuffer` collections, one can turn it into a builder for `Array`s like this:

```
bldr mapResult (_.toArray)
```

4.2 Factoring common operations

The main design objectives of the collection library redesign were to have, at the same time, natural types and maximal sharing. For instance, the `filter` operation should yield, on every collection type, an instance of the same collection type `aList filter p` should give a list, `aMap filter p` should give a map and so on. This is achieved by generic builders and traversals over collections in so called implementation traits. Collection classes such as `Traversable` or `Vector` inherit all their concrete method implementations from an implementation trait. These traits are named with the `Like` suffix; for instance `VectorLike` is the implementation trait for `Vector` and `TraversableLike` is the implementation trait for `Traversable`. Implementation traits have two type parameters instead of one for normal collections. They parameterize not only over the collection's element type, but also over the collection's representation type. For instance, here is the header of trait `TraversableLike`:

```
trait TraversableLike[+Elem, +Repr] { ... }
```

```
package scala.collection
class TraversableLike[+Elem, +Repr] {
  def newBuilder: Builder[Elem, Repr] // deferred
  def foreach[U](f: Elem => U)        // deferred
  ...
  def filter(p: Elem => Boolean): Repr = {
    val b = newBuilder
    foreach { elem => if (p(elem)) b += elem }
    b.result
  }
}
```

Figure 13: Implementation of `filter` in `TraversableLike`

The type parameter `Elem` stands for the element type of the traversable whereas the type parameter `Repr` stands for its representation. There are no constraints on `Repr`. In particular `Repr` might be instantiated to a type that is itself not a subtype of `Traversable`. That way, classes outside the collections hierarchy such as `String` and `Array` can still make use of all operations defined in a collection implementation trait.

Take `filter` as an example, this operation is defined in the same way for all collection classes in the trait `TraversableLike`. An outline of the relevant code is shown in Figure 13. The trait declares two abstract methods `newBuilder` and `foreach`, which are implemented in concrete collection classes. The `filter` operation is implemented in the same way for all collections using these methods.

Even more complicated is the `map` operation on collections: For instance, if `f` is a function from `String` to `Int`, and `xs` is a `List[String]`, then `xs map f` should give a `List[Int]`. Likewise, if `ys` is an `Array[String]`, then `ys map f` should give a `Array[Int]`. The problem is how to achieve that without duplicating the definition of the `map` method in lists and arrays. The `newBuilder`/`foreach` framework shown in Figure 13 is not sufficient for this because it only allows creation of new instances of the same collection *type* whereas `map` needs an instance of the same collection *type constructor*, but possibly with a different element type.

At first, we thought that the key to solving this was higher-kinded types. For instance, `map`, when operating on a collection `C[A]`, would take a function from `A` to `B` and yield a `C[B]`. Higher-kinded types let one abstract over the type constructor `C` so that one can envision just a single implementation of `map` for all collection classes `C`. However, it turned out that higher kinded types were too uniform in that they demand the same parameter shape for all inheriting collection types. This is too restrictive. For instance, a `String` (or rather: its backing class `RichString`) can be seen as a sequence of `Chars`, yet it is not a generic collection type. Nevertheless, mapping a character to character map over a `RichString` should again yield a `RichString`, as in the following interaction with the Scala REPL:

```
scala> "abc" map (x => (x + 1).toChar)
res1: scala.runtime.RichString = bcd
```

But what happens if one applies a function from `Char` to `Int` to a string? In that case, we cannot produce a string as result, it has to be some sequence of `Int` elements instead. Indeed one gets:

```
"abc" map (x => (x + 1))
res2: scala.collection.immutable.Vector[Int] = Vector(98, 99, 100)
```

So it turns out that `map` yields different types depending on what the result type of the passed function argument is! It's impossible to achieve that effect with plain higher-kinded types.

The problem with `String` is not an isolated case. Here are two more interactions with the REPL which both map a function over a map:

```
scala> Map("a" -> 1, "b" -> 2) map { case (x, y) => (y, x) }
res3: scala.collection.immutable.Map[Int,java.lang.String] = Map(1 -> a, 2 -> b)

scala> Map("a" -> 1, "b" -> 2) map { case (x, y) => y }
res4: scala.collection.immutable.Iterable[Int] = List(1, 2)
```

The first function swaps two arguments of a key/value pair. The result of mapping this function is again a map, but now going in the other direction. In fact, the original yields the inverse of the original map, provided it is invertible. The second function, however, maps the key/value pair to an integer, namely its value component. In that case, we cannot form a `Map` from the results, but we still can form an `Iterable`, which is the base trait of `Map`.

One might ask, why not restrict `map` so that the same kind of collection can be returned. For instance, on strings `map` could accept only character-to-character functions and on maps it could only accept pair-to-pair functions. Such restriction would violate the Liskov substitution principle: A `Map` *is* an `Iterable`. So every operation that's legal on `Iterable`s must also be legal on `Map`s.

We solve this problem with overloading. Not the simple form of overloading inherited by Java (that would not be flexible enough), but the more systematic form of overloading that's provided by implicit parameters.[1]

Figure 14 presents the implementation of `map` in the trait `TraversableLike`. It's quite similar to the implementation of `filter` shown in Figure 13. The principal difference is that where `filter` used the `newBuilder` method which is abstract in class `TraversableLike`, `map` uses a builder factory that's passed an as additional implicit parameter.

Figure 15 shows the definition of the trait `BuilderFactory`. Builder factories have three type parameters: `Elem` indicates the element type of the collection to be built, `To` indicates the type of collection to build, and `From` indicates the type for which this builder factory applies. By defining the right implicit definitions of builder factories, one can tailor the right typing behavior as needed. Take class `RichString` as an example, its companion object would contain a builder factory of type

[1] Type classes fulfill a similar role in Haskell.

```
def map[B, That](p: Elem => B)
                (implicit bf: BuilderFactory[B, That, This]): That = {
  val b = bf(this)
  for (x <- this) b += f(x)
  b.result
}
```

Figure 14: Implementation of `map` in `TraversableLike`

```
package scala.collection.generic
trait BuilderFactory[-Elem, +To, -From] {
  def apply(from: From): Builder[Elem, To]  // Creates a new builder
}
```

Figure 15: The `BuilderFactory` trait

`BuilderFactory[Char, RichString, RichString]`. This means that operating on a `RichString` one can construct another `RichString` provided the type of the collection to build is `Char`. If this is not the case, one can always fall back to a different implicit builder factory, this time implemented in `Iterable`'s companion object. The type of this more general builder factory is

```
BuilderFactory[A, Iterable[A], Iterable[_]],
```

where `A` is a generic type parameter. This means that when operating on an arbitrary `Iterable` (expressed by the existential type `Iterable[_]`) one can build again an `Iterable`, no matter what the element type `A` is. Given these two implementations of builder factories, one can then rely on Scala's rules for implicit resolution to pick the one that's appropriate and maximally specific.

So implicit resolution provides the correct static types for tricky collection operations such as `map`. But what about the dynamic types? Specifically, say you have a list value that has `Iterable` as its static type, and you map some function over that value:

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

scala> xs map (x => x * x)
res6: Iterable[Int] = List(1, 4, 9)
```

The static type of `res6` above is `Iterable`, as expected. But its dynamic type is (and should be) still `List`! This behavior is achieved by one more indirection. The `apply` method in `BuilderFactory` is passed the source collection as argument. Most builder factories for generic traversables (in fact all except builder factories for leaf classes) forward the call to a method `genericBuilder` of a collection. `genericBuilder` in turn calls the builder that belongs to the collection in which it is defined. So we use static implicit resolution to resolve constraints on the types of `map` and we use virtual dispatch to pick the best dynamic type that corresponds to these constraints.

```
package scala.collection
package immutable
import generic.GenericTraversableTemplate
trait Vector[+A] extends immutable.Seq[A]
                    with scala.collection.Vector[A]
                    with VectorLike[A, Vector[A]]
                    with GenericTraversableTemplate[A, Vector] {
  override def companion: Companion[Vector]  = Vector
}
object Vector extends SeqFactory[Vector] {
  class Impl[A](buf: ArrayBuffer[A]) extends Vector[A] {
    def length = buf.length
    def apply(idx: Int) = buf.apply(idx)
  }
  implicit def builderFactory[A]: BuilderFactory[A, Vector[A], Vector[_]] =
    new VirtualBuilderFactory[A]
  def newBuilder[A]: Builder[A, Vector[A]] =
    new ArrayBuffer[A] mapResult (buf => new Impl(buf))
}
```

Figure 16: A sample collection implementation.

4.3 Integrating new collections

What needs to be done if one wants to integrate a new collection class, so that it can profit from all predefined operations at the right types? As an example, Figure 16 shows a simple yet complete implementation of immutable vectors (the actual implementation of immutable vectors is considerably more refined algorithmically, though!)

```
trait GenericTraversableTemplate[+A, +CC[X] <: Traversable[X]] {
  def companion: Companion[CC]
  protected[this] def newBuilder: Builder[A, CC[A]] = companion.newBuilder[A]
  def genericBuilder[B]: Builder[B, CC[B]] = companion.newBuilder[B]
}
```

Figure 17: Trait `GenericTraversableTemplate`.

The `Vector` trait inherits from four other traits. It inherits from `immutable.Seq` because all vectors inherit from their corresponding sequence classes. It also inherits from `collection.Vector`, which is a base trait of both immutable and mutable vectors. It inherits most implementations of methods from the `VectorLike` trait. As other traversable templates, `VectorLike` requires an implementation of builders to be given when implementing its deferred `newBuilder` and `genericBuilder` methods. These methods are defined in trait `GenericTraversableTemplate` to forward their calls to the `newBuilder` method of `Vector`'s companion object. Figure 17 presents the definition of this trait.

The trait defines an abstract method `companion` that needs to be implemented in inheriting traits and classes. The method is implemented in class `Vector` to refer to `Vector`'s companion object. `GenericTraversableTemplate` then defines the monomophic `newBuilder` method and the generic `genericBuilder` method to both forward to `companion.newBuilder`.

Note that `GenericTraversableTemplate` is higher-kinded — it takes a type constructor `CC` as type parameter. In the `Vector` trait we instantiate that type parameter with `Vector` itself.

The definition of `companion` is the only definition in trait `Vector`. Like all sequences, vectors declare two abstract methods, `apply` and `length`, which need to be defined in subclasses. All other operations are already defined by inherited traits.

Let's turn now to the definition of `Vector`'s companion object. Its principal task is to define a standard implementation of the abstract `Vector` trait and to provide a builder for this implementation in the `newBuilder` method. The standard implementation is provided by class `Impl`. It simply forwards `apply` and `length` calls to an underlying array buffer. `Vector.newBuilder` creates an array buffer (which is a specialized kind of builder), and transforms results coming out of this buffer into instances of `Impl`. That's the minimal functionality that needs to be provided for instances of `GenericTraversableTemplate`.

As an added convenience, the `Vector` object inherits from class `SeqFactory` which makes available all creation methods described in Figure 11 for vectors.

One other important thing to be arranged is to make operations like `map` return again an `immutable.Vector`. As is explained in Section 4.2, you need an implicit builder factory for that. The `builderFactory` method provided by `Vector` creates a new instance of a `VirtualBuilderFactory`. This factory class produces builders that, by calling back into the `genericBuilder` method of the underlying collection, implement the dynamic type adaptation described in Section 4.2. Here's the definition of

VirtualBuilderFactory:

```
class VirtualBuilderFactory[A] extends BuilderFactory[A, CC[A], CC[_]] {
  def apply(from: Coll) = from.genericBuilder[A]
}
```

The class is defined as an inner class of the SeqFactory class inherited by the Vector object. The CC type parameter refers to the higher-kinded type of the collection that's built.

To summarize: If you want to fully integrate a new collection class into the framework you need to pay attention to the following points:

1. Decide whether the collection should be mutable or immutable.
2. Pick the right base classes for the collection.
3. Inherit from the right template trait to implement most collection operations.
4. If you want map and similar operations return instances of your collection type, provide an implicit builder factory in the companion object.
5. If the collection should have dynamic type adapation for map and operations like it, also inherit from GenericTraversableTemplate, or implement equivalent functionality.

The last point is important only for abstract collection classes that might be further specialized in subclasses. If you expect your collection class to be the final implementation, you can implement its newBuilder method directly, without going through the virtual dispatch provided by the VirtualBuilder class.

A simpler scheme is also possible if you do not need bulk operations like map or filter to return your collection type. In that case you can simply inherit from some general collection class like Seq or Map and implement the additional operations.

5 Migrating to the new collections

The new collections depart in some significant way from the pre-2.8 state. The changes are most pronounced for implementors of new collections, whereas clients of collection classes will probably be less affected.

Two measures have been taken in order to ensure a reasonable amount of backwards compatibility with pre-2.8 collections: the scala package object and deprecated methods.

```
package object scala {
  type Iterable[+A] = scala.collection.Iterable[A]
  val Iterable = scala.collection.Iterable

  /** @deprecated  use Iterable instead */
  @deprecated type Collection[+A] = Iterable[A]
  /** @deprecated  use Iterable instead */
  @deprecated val Collection = Iterable

  type Seq[+A] = scala.collection.Seq[A]
  val Seq = scala.collection.Seq

  type RandomAccessSeq[+A] = scala.collection.Vector[A]
  val RandomAccessSeq = scala.collection.Vector

  type Iterator[+A] = scala.collection.Iterator[A]
  val Iterator = scala.collection.Iterator

  type BufferedIterator[+A] = scala.collection.BufferedIterator[A]

  type List[+A] = scala.collection.immutable.List[A]
  val List = scala.collection.immutable.List

  val Nil = scala.collection.immutable.Nil

  type ::[A] = scala.collection.immutable.::[A]
  val :: = scala.collection.immutable.::

  type Stream[+A] = scala.collection.immutable.Stream[A]
  val Stream = scala.collection.immutable.Stream

  type StringBuilder = scala.collection.mutable.StringBuilder
  val StringBuilder = scala.collection.mutable.StringBuilder
}
```

Figure 18: The `scala` package object.

5.1 The Scala Package Object

Several important pre-2.8 collection classes exist now under different names and in different packages. To smooth the transition, Scala 2.8 defines a package object that defines aliases for common collection classes that used to be in the `scala` package.

Package objects are a new concept in Scala. They permit to add definitions in a special object that are then treated as members of a package. For instance, the package object `scala`, which is found in file

scala/**package**.scala, defines additional members of package scala. Note that one could not have defined these members outside a package object, because **type** aliases and **val** definitions are not permitted as top-level definitions.

Figure 18 shows the definition of the scala package object. In the medium term, at least some the aliases in this object will be deprecated. In particular Collection does not play a role any more – it has been subsumed by Iterable. Also, RandomAccessSeq has been subsumed by collection.Vector, and RandomAccessSeq.Mutable by collection.mutable.Vector.

5.2 Deprecated methods

Where possible, methods and classes that are slated to removed in the new collection framework still continue to exist, but are now marked @deprecated. The Scala compiler will issue a warning when a @deprecated symbol is accessed. The ScalaDoc documentation usually suggests a suitable replacement.

The Scala policy is to keep @deprecated methods for at least one major version. So these methods are guaranteed to exist for Scala 2.8.x versions but not beyond.

5.3 Known migration issues

The following list contains some of the migration issues that go beyond deprecated methods.

- scala.collection.jcl does not exist anymore. Instead, there's an object scala.collection.JavaConversions which contains implicit conversions between Scala collections and java.util collections. The jcl.WeakHashMap and jcl.WeakhashSet classes have been integrated directly into collection.mutable.
- Projections have been replaced by views and some of the functionality is different.
- Buffered iterators have been simplified. They now offer only single element lookahead. The BufferedIterator.Advanced class has been eliminated.
- Counted iterators have been eliminated. They were curiously off by one anyway, so I doubt they have been used much.
- Class Collection is gone. The size method is now concrete in class Traversable (it simply traverses the collection with a counter). This means that all implementations of collections (i.e. sets, maps, sequences) which define a size method must now ensure there's an **override** modifier in the definition.
- The elements method in class Iterable has been renamed to iterator. elements is still available as a deprecated method, so clients of collection classes will continue to work, albeit with deprecation warnings. However, any implementation of a collection class now needs to implement iterator instead of elements. If it does implement elements, it needs to add an **override**, because elements is now a concrete method that forwards to iterator.

- The `keys` and `values` methods in maps now return sets instead of iterators. The iterator-returning methods are now named `keysIterator` and `valuesIterator`.
- Maps and sets in package `scala.collection` now also define the addition and removal operations `+` and `–`, which return a new collection arising from some modification to the existing collection. These are abstract and need to be implemented in classes inheriting from `collecton.Map` or `collection.Set`. Previously, these two classes did not define update operations. To facilitate the definition of these operations there are now classes `collection.DefaultMap` and `collection.DefaultSet` which implement the abstract operations in a default way. If your collection class does not care about specific update operations, it can simply inherit from `collection.DefaultMap` or `collection.DefaultSet` instead of `Map` or `Set`.