Getting Started ▾ Learn ▾ Scala 3 Migration ▾

SCALA 3 — BOOK

COLLECTIONS TYPES

This page demonstrates the common Scala 3 collections and their accompanying methods. Scala comes with a wealth of collections types, but you can go a long way by starting with just a few of them, and later using the others as needed. Similarly, each collection type has dozens of methods to make your life easier, but you can achieve a lot by starting with just a handful of them.

Therefore, this section introduces and demonstrates the most common types and methods that you'll need to get started. When you need more flexibility, see these pages at the end of this section for more details.

Three main categories of collections

Looking at Scala collections from a high level, there are three main categories to choose from:

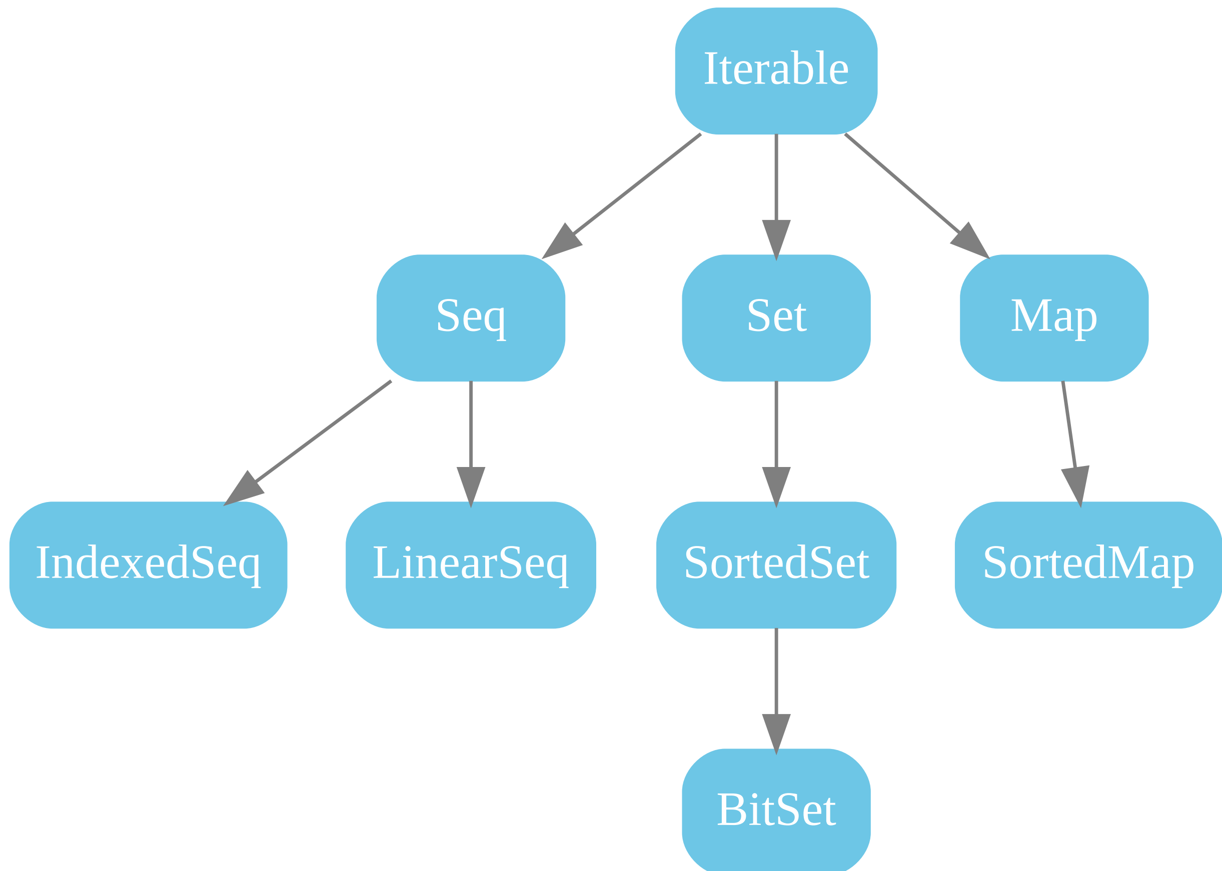
- **Sequences** are a sequential collection of elements and may be *indexed* (like an array) or *linear* (like a linked list)
- **Maps** contain a collection of key/value pairs, like a Java `Map`, Python dictionary, or Ruby `Hash`
- **Sets** are an unordered collection of unique elements

All of those are basic types, and have subtypes for specific purposes, such as concurrency, caching, and streaming. In addition to those three main categories, there are other useful collection types, including ranges, stacks, and queues.

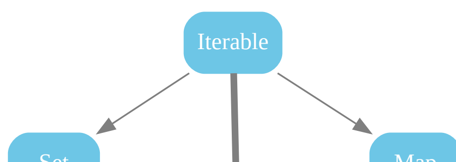
Collections hierarchy

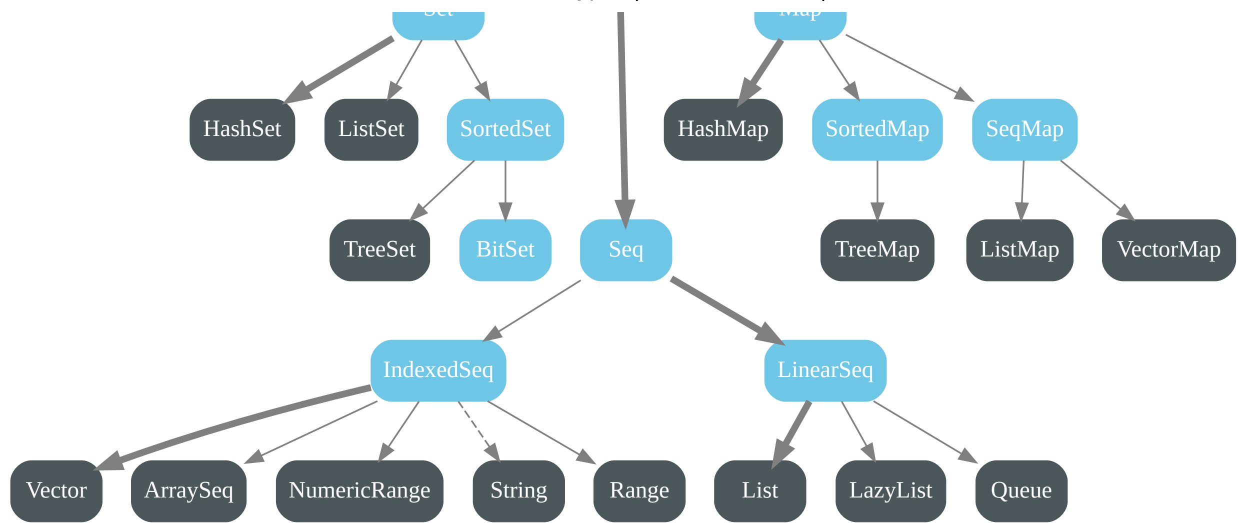
As a brief overview, the next three figures show the hierarchy of classes and traits in the Scala collections.

This first figure shows the collections types in package *scala.collection*. These are all high-level abstract classes or traits, which generally have *immutable* and *mutable* implementations.

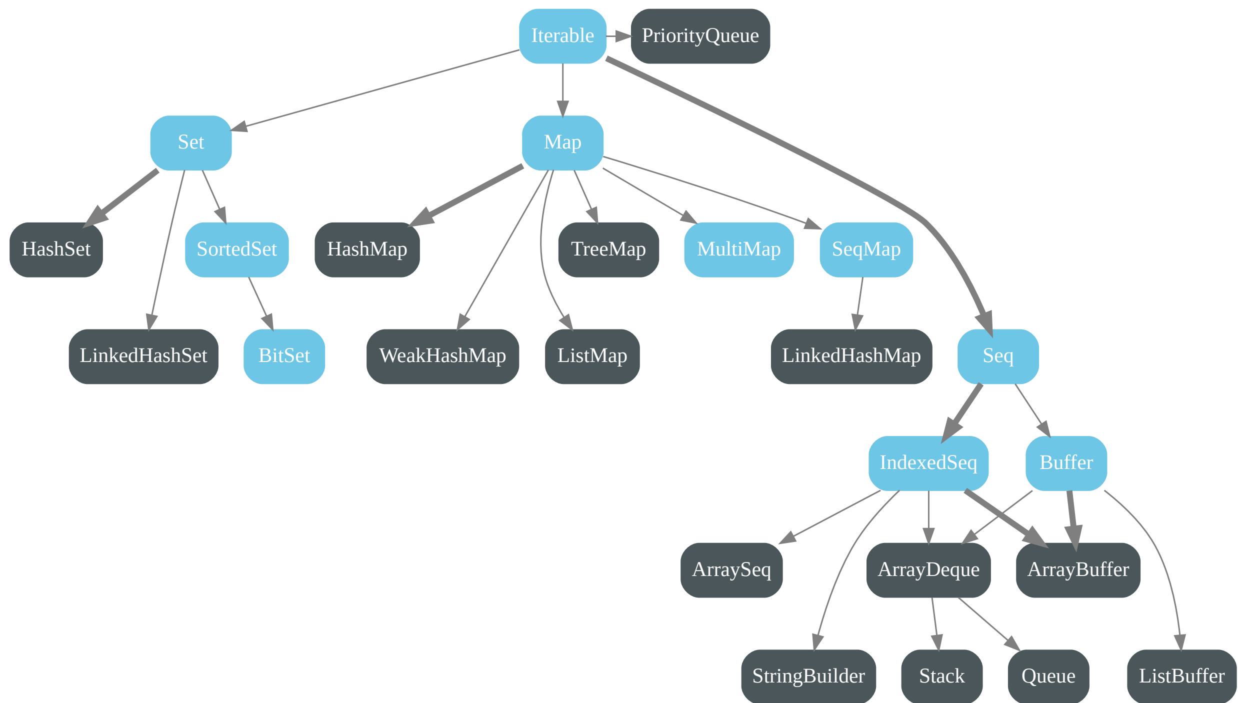


This figure shows all collections in package *scala.collection.immutable*:





And this figure shows all collections in package *scala.collection.mutable*:



Having seen that detailed view of all the collections types, the following sections introduce some common types you'll use on a regular basis.

Common collections

The main collections you'll use on a regular basis are:

Collection Type	Immutable	Mutable	Description
`List`	✓		A linear (linked list), immutable sequence

`Vector`	✓		An indexed, immutable sequence
`LazyList`	✓		A lazy immutable linked list, its elements are computed only when they're needed; Good for large or infinite sequences.
`ArrayBuffer`		✓	The go-to type for a mutable, indexed sequence
`ListBuffer`		✓	Used when you want a mutable `List`; typically converted to a `List`
`Map`	✓	✓	An iterable collection that consists of pairs of keys and values.
`Set`	✓	✓	An iterable collection with no duplicate elements

As shown, `Map` and `Set` come in both immutable and mutable versions.

The basics of each type are demonstrated in the following sections.

> *In Scala, a buffer—such as `ArrayBuffer` and `ListBuffer`—is a sequence that can grow and shrink.*

A note about immutable collections

In the sections that follow, whenever the word *immutable* is used, it's safe to assume that the type is intended for use in a *functional programming* (FP) style. With these types you don't modify the collection; you apply functional methods to the collection to create a new result.

Choosing a sequence

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

- Should the sequence be indexed (like an array), allowing rapid access to any element, or should it be implemented as a linear linked list?
- Do you want a mutable or immutable collection?

The recommended, general-purpose, "go to" sequential collections for the combinations of mutable/immutable and indexed/linear are shown here:

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

For example, if you need an immutable, indexed collection, in general you should use a `Vector`. Conversely, if you need a mutable, indexed collection, use an `ArrayBuffer`.

> *`List` and `Vector` are often used when writing code in a functional style. `ArrayBuffer` is commonly used when writing code in an imperative style. `ListBuffer` is used when you're mixing styles, such as building a list.*

The next several sections briefly demonstrate the `List`, `Vector`, and `ArrayBuffer` types.

`List`

The List type is a linear, immutable sequence. This just means that it's a linked-list that you can't modify. Any time you want to add or remove `List` elements, you create a new `List` from an existing `List`.

Creating Lists

This is how you create an initial `List`:

Scala 2 and 3

```
val ints = List(1, 2, 3)
val names = List("Joel", "Chris", "Ed")

// another way to construct a List
val namesAgain = "Joel" :: "Chris" :: "Ed" :: Nil
```

You can also declare the `List`'s type, if you prefer, though it generally isn't necessary:

Scala 2 and 3

```
val ints: List[Int] = List(1, 2, 3)
val names: List[String] = List("Joel", "Chris", "Ed")
```

One exception is when you have mixed types in a collection; in that case you may want to explicitly specify its type:

Scala 2 Scala 3

```
val things: List[String | Int | Double] = List(1, "two", 3.0) // with un
val thingsAny: List[Any] = List(1, "two", 3.0)                 // with an
```

Adding elements to a List

Because `List` is immutable, you can't add new elements to it. Instead, you create a new list by prepending or appending elements to an existing `List`. For instance, given this `List`:

Scala 2 and 3

```
val a = List(1, 2, 3)
```

When working with a `List`, *prepend* one element with `::`, and prepend another `List` with `:::`, as shown here:

Scala 2 and 3

```
val b = 0 :: a              // List(0, 1, 2, 3)
val c = List(-1, 0) ::: a   // List(-1, 0, 1, 2, 3)
```

You can also *append* elements to a `List`, but because `List` is a singly-linked list, you should generally only prepend elements to it; appending elements to it is a relatively slow operation, especially when you work with large sequences.

> *Tip: If you want to prepend and append elements to an immutable sequence, use `Vector` instead.*

Because `List` is a linked-list, you shouldn't try to access the elements of large lists by their index value. For instance, if you have a `List` with one million elements in it, accessing an element like `myList(999_999)` will take a relatively long time, because

that request has to traverse all those elements. If you have a large collection and want to access elements by their index, use a `Vector` or `ArrayBuffer` instead.

How to remember the method names

These days IDEs help us out tremendously, but one way to remember those method names is to think that the `:` character represents the side that the sequence is on, so when you use `+:` you know that the list needs to be on the right, like this:

Scala 2 and 3

```
0 +: a
```

Similarly, when you use `:+` you know the list needs to be on the left:

Scala 2 and 3

```
a :+ 4
```

There are more technical ways to think about this, but this can be a helpful way to remember the method names.

Also, a good thing about these symbolic method names is that they're consistent. The same method names are used with other immutable sequences, such as `Seq` and `Vector`. You can also use non-symbolic method names to append and prepend elements, if you prefer.

How to loop over lists

Given a `List` of names:

Scala 2 and 3

```
val names = List("Joel", "Chris", "Ed")
```

you can print each string like this:

Scala 2 Scala 3

```
for name <- names do println(name)
```

This is what it looks like in the REPL:

Scala 2 Scala 3

```
scala> for name <- names do println(name)
Joel
Chris
Ed
```

A great thing about using `for` loops with collections is that Scala is consistent, and the same approach works with all sequences, including `Array`, `ArrayBuffer`, `List`, `Seq`, `Vector`, `Map`, `Set`, etc.

A little bit of history

For those interested in a little bit of history, the Scala `List` is similar to the `List` from the Lisp programming language, which was originally specified in 1958. Indeed, in addition to creating a `List` like this:

Scala 2 and 3

```
val ints = List(1, 2, 3)
```

you can also create the exact same list this way:

Scala 2 and 3

```
val list = 1 :: 2 :: 3 :: Nil
```

The REPL shows how this works:

Scala 2 and 3

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

This works because a `List` is a singly-linked list that ends with the `Nil` element, and `::` is a `List` method that works like Lisp's "cons" operator.

Aside: The LazyList

The Scala collections also include a LazyList, which is a *lazy* immutable linked list. It's called "lazy"—or non-strict—because it computes its elements only when they are needed.

You can see how lazy a `LazyList` is in the REPL:

Scala 2 and 3

```
val x = LazyList.range(1, Int.MaxValue)
x.take(1)      // LazyList(<not computed>)
x.take(5)      // LazyList(<not computed>)
x.map(_ + 1)   // LazyList(<not computed>)
```

In all of those examples, nothing happens. Indeed, nothing will happen until you force it to happen, such as by calling its `foreach` method:

Scala 2 and 3

```
scala> x.take(1).foreach(println)
1
```

For more information on the uses, benefits, and drawbacks of strict and non-strict (lazy) collections, see the "strict" and "non-strict" discussions on the The Architecture of Scala 2.13's Collections page.

Vector

Vector is an indexed, immutable sequence. The "indexed" part of the description means that it provides random access and update in effectively constant time, so you can access `Vector` elements rapidly by their index value, such as accessing `listOfPeople(123_456_789)`.

In general, except for the difference that (a) `Vector` is indexed and `List` is not, and (b) `List` has the `::` method, the two types work the same, so we'll quickly run through the following examples.

Here are a few ways you can create a `Vector`:

Scala 2 and 3

```
val nums = Vector(1, 2, 3, 4, 5)

val strings = Vector("one", "two")

case class Person(name: String)
val people = Vector(
  Person("Bert"),
  Person("Ernie"),
  Person("Grover")
)
```

Because `Vector` is immutable, you can't add new elements to it. Instead, you create a new sequence by appending or prepending elements to an existing `Vector`. These examples show how to *append* elements to a `Vector`:

Scala 2 and 3

```
val a = Vector(1,2,3)         // Vector(1, 2, 3)
val b = a :+ 4                // Vector(1, 2, 3, 4)
val c = a ++ Vector(4, 5)     // Vector(1, 2, 3, 4, 5)
```

This is how you *prepend* elements:

Scala 2 and 3

```
val a = Vector(1,2,3)         // Vector(1, 2, 3)
val b = 0 +: a                // Vector(0, 1, 2, 3)
val c = Vector(-1, 0) ++: a   // Vector(-1, 0, 1, 2, 3)
```

In addition to fast random access and updates, `Vector` provides fast append and prepend times, so you can use these features as desired.

See the Collections Performance Characteristics for performance details about `Vector` and other collections.

Finally, you use a `Vector` in a `for` loop just like a `List`, `ArrayBuffer`, or any other sequence:

Scala 2 Scala 3

```
scala> val names = Vector("Joel", "Chris", "Ed")
val names: Vector[String] = Vector(Joel, Chris, Ed)

scala> for name <- names do println(name)
Joel
Chris
Ed
```

ArrayBuffer

Use `ArrayBuffer` when you need a general-purpose, mutable indexed sequence in your Scala applications. It's mutable, so you can change its elements, and also resize it. Because it's indexed, random access of elements is fast.

Creating an ArrayBuffer

To use an `ArrayBuffer`, first import it:

Scala 2 and 3

```
import scala.collection.mutable.ArrayBuffer
```

If you need to start with an empty `ArrayBuffer`, just specify its type:

Scala 2 and 3

```
var strings = ArrayBuffer[String]()
var ints = ArrayBuffer[Int]()
var people = ArrayBuffer[Person]()
```

If you know the approximate size your `ArrayBuffer` eventually needs to be, you can create it with an initial size:

Scala 2 and 3

Scala 2 and 3

```
// ready to hold 100,000 ints
val buf = new ArrayBuffer[Int](100_000)
```

To create a new `ArrayBuffer` with initial elements, just specify its initial elements, just like a `List` or `Vector`:

Scala 2 and 3

```
val nums = ArrayBuffer(1, 2, 3)
val people = ArrayBuffer(
  Person("Bert"),
  Person("Ernie"),
  Person("Grover")
)
```

Adding elements to an ArrayBuffer

Append new elements to an `ArrayBuffer` with the `+=` and `++=` methods. Or if you prefer methods with textual names you can also use `append`, `appendAll`, `insert`, `insertAll`, `prepend`, and `prependAll`.

Here are some examples of `+=` and `++=`:

Scala 2 and 3

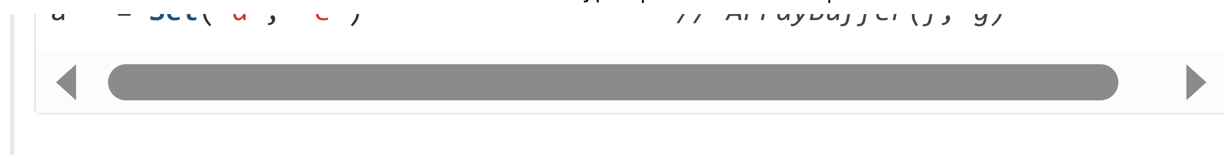
```
val nums = ArrayBuffer(1, 2, 3)   // ArrayBuffer(1, 2, 3)
nums += 4                         // ArrayBuffer(1, 2, 3, 4)
nums ++= List(5, 6)               // ArrayBuffer(1, 2, 3, 4, 5, 6)
```

Removing elements from an ArrayBuffer

`ArrayBuffer` is mutable, so it has methods like `-=`, `--=`, `clear`, `remove`, and more. These examples demonstrate the `-=` and `--=` methods:

Scala 2 and 3

```
val a = ArrayBuffer.range('a', 'h')   // ArrayBuffer(a, b, c, d, e, f, g
a -= 'a'                              // ArrayBuffer(b, c, d, e, f, g)
a --= Seq('b', 'c')                   // ArrayBuffer(d, e, f, g)
a --= Set('d', 'e')                   // ArrayBuffer(f, g)
```



Updating ArrayBuffer elements

Update elements in an `ArrayBuffer` by either reassigning the desired element, or use the `update` method:

Scala 2 and 3

```
val a = ArrayBuffer.range(1,5)        // ArrayBuffer(1, 2, 3, 4)
a(2) = 50                             // ArrayBuffer(1, 2, 50, 4)
a.update(0, 10)                       // ArrayBuffer(10, 2, 50, 4)
```

Maps

A `Map` is an iterable collection that consists of pairs of keys and values. Scala has both mutable and immutable `Map` types, and this section demonstrates how to use the *immutable* `Map`.

Creating an immutable Map

Create an immutable `Map` like this:

Scala 2 and 3

```
val states = Map(
  "AK" -> "Alaska",
  "AL" -> "Alabama",
  "AZ" -> "Arizona"
)
```

Once you have a `Map` you can traverse its elements in a `for` loop like this:

Scala 2 Scala 3

```
for (k, v) <- states do println(s"key: $k, value: $v")
```

The REPL shows how this works:

Scala 2 Scala 3

```
scala> for (k, v) <- states do println(s"key: $k, value: $v")
key: AK, value: Alaska
key: AL, value: Alabama
key: AZ, value: Arizona
```

Accessing Map elements

Access map elements by specifying the desired key value in parentheses:

Scala 2 and 3

```
val ak = states("AK")   // ak: String = Alaska
val al = states("AL")   // al: String = Alabama
```

In practice, you'll also use methods like `keys`, `keySet`, `keysIterator`, `for` loops, and higher-order functions like `map` to work with `Map` keys and values.

Adding elements to a Map

Add elements to an immutable map using `+` and `++`, remembering to assign the result to a new variable:

Scala 2 and 3

```
val a = Map(1 -> "one")    // a: Map(1 -> one)
val b = a + (2 -> "two")   // b: Map(1 -> one, 2 -> two)
val c = b ++ Seq(
  3 -> "three",
  4 -> "four"
)
// c: Map(1 -> one, 2 -> two, 3 -> three, 4 -> four)
```

Removing elements from a Map

Remove elements from an immutable map using `-` or `--` and the key values to remove, remembering to assign the result to a new variable:

Scala 2 and 3

Scala 2 and 3

```
val a = Map(
  1 -> "one",
  2 -> "two",
  3 -> "three",
  4 -> "four"
)

val b = a - 4       // b: Map(1 -> one, 2 -> two, 3 -> three)
val c = a - 4 - 3   // c: Map(1 -> one, 2 -> two)
```

Updating Map elements

To update elements in an immutable map, use the `updated` method (or the `+` operator) while assigning the result to a new variable:

Scala 2 and 3

```
val a = Map(
  1 -> "one",
  2 -> "two",
  3 -> "three"
)

val b = a.updated(3, "THREE!")   // b: Map(1 -> one, 2 -> two, 3 -> THRI
val c = a + (2 -> "TWO...")      // c: Map(1 -> one, 2 -> TWO..., 3 -> t
```

Traversing a Map

As shown earlier, this is a common way to manually traverse elements in a map using a `for` loop:

Scala 2 Scala 3

```
val states = Map(
  "AK" -> "Alaska",
  "AL" -> "Alabama",
  "AZ" -> "Arizona"
)

for (k, v) <- states do println(s"key: $k, value: $v")
```

That being said, there are *many* ways to work with the keys and values in a map. Common `Map` methods include `foreach`, `map`, `keys`, and `values`.

Scala has many more specialized `Map` types, including `CollisionProofHashMap`, `HashMap`, `LinkedHashMap`, `ListMap`, `SortedMap`, `TreeMap`, `WeakHashMap`, and more.

Working with Sets

The Scala Set is an iterable collection with no duplicate elements.

Scala has both mutable and immutable `Set` types. This section demonstrates the *immutable* `Set`.

Creating a Set

Create new empty sets like this:

Scala 2 and 3

```
val nums = Set[Int]()
val letters = Set[Char]()
```

Create sets with initial data like this:

Scala 2 and 3

```
val nums = Set(1, 2, 3, 3, 3)           // Set(1, 2, 3)
val letters = Set('a', 'b', 'c', 'c')   // Set('a', 'b', 'c')
```

Adding elements to a Set

Add elements to an immutable `Set` using `+` and `++`, remembering to assign the result to a new variable:

Scala 2 and 3

```
val a = Set(1, 2)                // Set(1, 2)
val b = a + 3                    // Set(1, 2, 3)
val c = b ++ Seq(4, 1, 5, 5)     // HashSet(5, 1, 2, 3, 4)
```

Notice that when you attempt to add duplicate elements, they're quietly dropped.

Also notice that the order of iteration of the elements is arbitrary.

Deleting elements from a Set

Remove elements from an immutable set using `-` and `--`, again assigning the result to a new variable:

Scala 2 and 3

```
val a = Set(1, 2, 3, 4, 5)   // HashSet(5, 1, 2, 3, 4)
val b = a - 5                // HashSet(1, 2, 3, 4)
val c = b -- Seq(3, 4)       // HashSet(1, 2)
```

Range

The Scala `Range` is often used to populate data structures and to iterate over `for` loops. These REPL examples demonstrate how to create ranges:

Scala 2 and 3

```
1 to 5          // Range(1, 2, 3, 4, 5)
1 until 5       // Range(1, 2, 3, 4)
1 to 10 by 2    // Range(1, 3, 5, 7, 9)
'a' to 'c'      // NumericRange(a, b, c)
```

You can use ranges to populate collections:

Scala 2 and 3

```
val x = (1 to 5).toList      // List(1, 2, 3, 4, 5)
val x = (1 to 5).toBuffer    // ArrayBuffer(1, 2, 3, 4, 5)
```

They're also used in `for` loops:

Scala 2 Scala 3

```
scala> for i <- 1 to 3 do println(i)
```

```
1
2
3
```

There are also `range` methods on :

Scala 2 and 3

```
Vector.range(1, 5)       // Vector(1, 2, 3, 4)
List.range(1, 10, 2)     // List(1, 3, 5, 7, 9)
Set.range(1, 10)         // HashSet(5, 1, 6, 9, 2, 7, 3, 8, 4)
```

When you're running tests, ranges are also useful for generating test collections:

Scala 2 and 3

```
val evens = (0 to 10 by 2).toList     // List(0, 2, 4, 6, 8, 10)
val odds = (1 to 10 by 2).toList      // List(1, 3, 5, 7, 9)
val doubles = (1 to 5).map(_ * 2.0)   // Vector(2.0, 4.0, 6.0, 8.0, 10.

// create a Map
val map = (1 to 3).map(e => (e,s"$e")).toMap
    // map: Map[Int, String] = Map(1 -> "1", 2 -> "2", 3 -> "3")
```

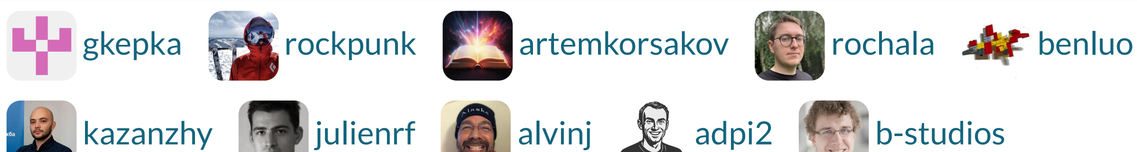
More details

When you need more information about specialized collections, see the following resources:

- Concrete Immutable Collection Classes
- Concrete Mutable Collection Classes
- How are the collections structured? Which one should I choose?

← **previous** **next** →

Contributors to this page:



Contents

Introduction

Scala Features

Why Scala 3?

A Taste of Scala
- Hello, World!
- The REPL
- Variables and Data Types
- Control Structures
- Domain Modeling
- Methods
- First-Class Functions
- Singleton Objects
- Collections
- Contextual Abstractions
- Toplevel Definitions
- Summary

A First Look at Types

String Interpolation

Control Structures

Domain Modeling
- Tools
- OOP Modeling
- FP Modeling

Methods
- Method Features
- Main Methods in Scala 3
- Summary

Functions
- Anonymous Functions
- Function Variables
- Partial Functions
- Eta-Expansion
- Higher-Order Functions
- Write Your Own map Method

Creating a Method That Returns a Function
Summary

Packaging and Imports

Scala Collections

Collections Types

Three main categories of collections

- Collections hierarchy

Common collections

- A note about immutable collections

Choosing a sequence

List

- Creating Lists
- Adding elements to a List
- How to remember the method names
- How to loop over lists
- A little bit of history
- Aside: The LazyList

Vector

ArrayBuffer

- Creating an ArrayBuffer
- Adding elements to an ArrayBuffer
- Removing elements from an ArrayBuffer
- Updating ArrayBuffer elements

Maps

- Creating an immutable Map
- Accessing Map elements
- Adding elements to a Map
- Removing elements from a Map
- Updating Map elements
- Traversing a Map

Working with Sets

- Creating a Set
- Adding elements to a Set
- Deleting elements from a Set

Range

More details

Collections Methods
Summary

Functional Programming

What is Functional Programming?
Immutable Values
Pure Functions
Functions Are Values

Functional Error Handling
Summary

Types and the Type System

Inferred Types
Generics
Intersection Types
Union Types
Algebraic Data Types
Variance
Opaque Types
Structural Types
Dependent Function Types
Other Types

Contextual Abstractions
Extension Methods
Context Parameters
Context Bounds
Given Imports
Type Classes
Multiversal Equality
Implicit Conversions
Summary

Concurrency

Scala Tools
Building and Testing Scala Projects with sbt
Worksheets

Interacting with Java

Scala for Java Developers

Scala for JavaScript Developers

Scala for Python Developers

Where To Go Next

Problem with this page?
Please help us fix it!

DOCUMENTATION

Getting Started
API
Overviews/Guides
Language Specification

DOWNLOAD

Current Version
All versions

COMMUNITY

Community
Scala Ambassadors
Forums
Chat
Libraries and Tools
The Scala Center

CONTRIBUTE

How to help
Report an Issue

SCALA

Governance
Blog
Code of Conduct
License
Security Policy

SOCIAL

GitHub
Mastodon
Bluesky
X
Discord
LinkedIn