Scala

Parallel & Concurrent Programming

Scala

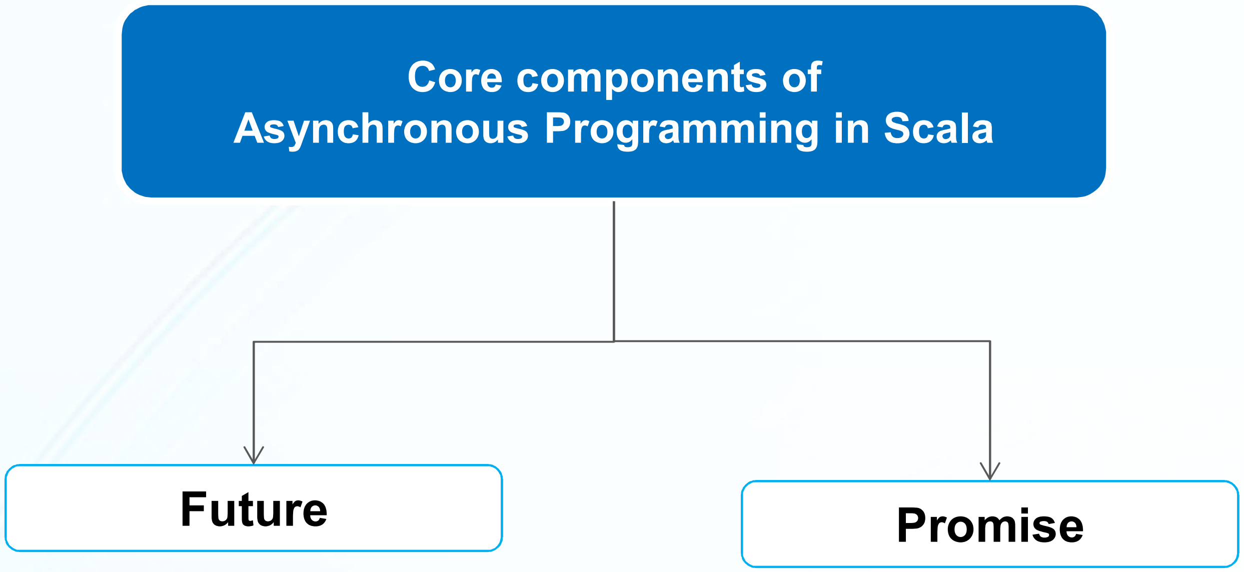
PARALLEL & CONCURRENT PROGRAMMING

Asynchronous Programming

An asynchronous computation is any task or thread that:

- executes outside of your program's main flow or from the point of view of the caller, it doesn't execute on the current call-stack.
- receives a callback that will get called once the result is finished processing.
- provides no guarantee about when the result is signaled, no guarantee that a result will be signaled at all.

Asynchronous Programming



Futures

Future

Future is an object holding a value which may become available at some point.

- If the computation has not yet completed, we say that the Future is not completed.
- If the computation has completed with a value or with an exception, we say that the Future is completed.
    - Future successfully completes with a value
    - Future fails with an exception thrown by the computation

Future

Future may only be assigned once.

Once a Future object is given a value or an exception, it becomes in effect immutable – it can never be overwritten.

Creating a Future

- The simplest way to create a future object is to invoke the **Future.apply** method which starts an asynchronous computation and returns a future holding the result of that computation.

- The result becomes available once the future completes.

```
val f = Future {
    .........
}(executionContext)
```

Execution Context

`ExecutionContext` is responsible for executing computations.

An `ExecutionContext` is similar to an Executor: it is free to execute computations in a new thread, in a pooled thread or in the current thread.

The `scala.concurrent` package comes out of the box with an `ExecutionContext` implementation, a global static thread pool.

Creating a Future

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext

val executionContext = ExecutionContext.global

val f = Future {
  for(i <- (1 to 1000)) { }
  println("Hello Future")
}(executionContext)
```

Creating a Future

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext

implicit val executionContext = ExecutionContext.global

val f = Future {
  for(i <- (1 to 1000)) { }
  println("Hello Future")
}
```

Creating a Future

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.implicits.global

val f : Fututе[String] = Future {
  for(i <- (1 to 1000)) { }
  "Hello Future”
}
```

Note that **`Future[T]`** is a type which denotes future objects, whereas **`Future.apply`** is a method which creates and schedules an asynchronous computation, and then returns a future object which will be completed with the result of that computation.

Creating a Future

The line **`import ExecutionContext.Implicits.global`** above imports the default global execution context. Execution contexts execute tasks submitted to them, and you can think of execution contexts as thread pools. They are essential for the **`Future.apply`** method because they handle how and when the asynchronous computation is executed.

Callbacks

Callbacks

A **callback** is called asynchronously once the future is completed.

If the future has already been completed when registering the callback, then the callback may either be executed asynchronously, or sequentially on the same thread.

Callbacks

The most general form of registering a callback is by using the **`onComplete`** method, which takes a callback function of type **`Try[T] => U`**.

The callback is applied to the value of type **`Success[T]`** if the future completes successfully, or to a value of type **`Failure[T]`** otherwise.

The **`Try[T]`** is similar to **`Option[T]`**. **`Try[T]`** is a **`Success[T]`** when it holds a value and otherwise **`Failure[T]`**, which holds an exception.

Callbacks

```
val f = Future {
  sleep(Random.nextInt(500))
  42
}

f.onComplete {
  case Success(value) => println(s”Value = $value")
  case Failure(e) => e.printStackTrace
}
```

Callbacks

The **onComplete** method is general in the sense that it allows the client to handle the result of both failed and successful future computations.

In the case where only successful results need to be handled, the **foreach** callback can be used.

```
val f = Future {
  sleep(Random.nextInt(500))
  42
}

f.foreach(println)
```

When a callback gets called?

- Since a callback requires the value in the future to be available, it can only be called **after the future is completed**.
- However, there is no guarantee it will be called by the thread that completed the future or the thread which created the callback. Instead, the callback is executed by some thread, at some time after the future object is completed. We say that the callback is executed ***eventually***.
- The order in which the callbacks are executed is not predefined, even between different runs of the same application. In fact, the callbacks may not be called sequentially one after the other, but may concurrently execute at the same time.

Callback Semantics

- Registering an `onComplete` callback on the future ensures that the corresponding closure is invoked after the future is completed, eventually.
- Registering a `foreach` callback has the same semantics as `onComplete`, with the difference that the closure is only called if the future is completed successfully.
- Registering a callback on the future which is already completed will result in the callback being executed eventually (as implied by 1).

Callback Semantics

- In the event that multiple callbacks are registered on the future, the order in which they are executed is not defined. In fact, the callbacks may be executed concurrently with one another.

- In the event that some of the callbacks throw an exception, the other callbacks are executed regardless.

- Once executed, the callbacks are removed from the future object, thus being eligible for GC.

Promise

Promise allows you to complete a Future by putting a value into it.

- Where Future provides an interface exclusively for querying, **Promise is a companion type that allows you to complete a Future by putting a value into it.**
- This can be done exactly once.
- Once a Promise has been completed, it’s not possible to change it any more.
- A Promise instance is always linked to exactly one instance of Future.

Promises

Promise

```
import concurrent.Future
import concurrent.ExecutionContext.Implicits.global

val f: Future[String] = Future { "Hello world!" }

// REPL output:
// f: scala.concurrent.Future[String] =
scala.concurrent.impl.Promise$DefaultPromise@793e6657
```

The object you get back is a `DefaultPromise`, which implements both `Future` and `Promise`. What this little example shows is that there is obviously no way to complete a Future other than through a `Promise` – the `apply` method on `Future` is just a nice helper function that shields you from this.

Promise

- A promise can be thought of as a writable, single-assignment container, which completes a future.
- A promise can be used to successfully complete a future with a value (by “completing” the promise) using the **success** method.
- Conversely, a promise can also be used to complete a future with an exception, by failing the promise, using the **failure** method.
- A promise can be completed with **complete** method for both **success** and **failure**.
- A promise p completes the future returned by p.future. This future is specific to the promise p.

Declaring a Promise

```
import scala.concurrent.Promise

val p = Promise[Int]()
val p2: Promise[Int] = Promise()
```

Once you have created a `Promise`, you can get the `Future` belonging to it by calling the ***`future`*** method on the `Promise` instance:

```
val f: Future[Int] = p.future
```

Completing a Promise

In Scala, you can complete a `Promise` either with a **success** or a **failure**.

To complete a `Promise` with a success, you call its ***success*** method, passing it the value that the `Future` associated with it is supposed to have:

```
p.success(20)
```

Once you have done this, that Promise instance is no longer writable, and future attempts to do so will lead to an exception.

Also, completing your Promise like this leads to the successful completion of the associated Future. Any success or completion handlers on that future will now be called.

Completing a Promise

You can complete your Promise, by calling its failure method too and passing it an exception.

```
p.failure(Try(“Sorry”).failed.get)
```

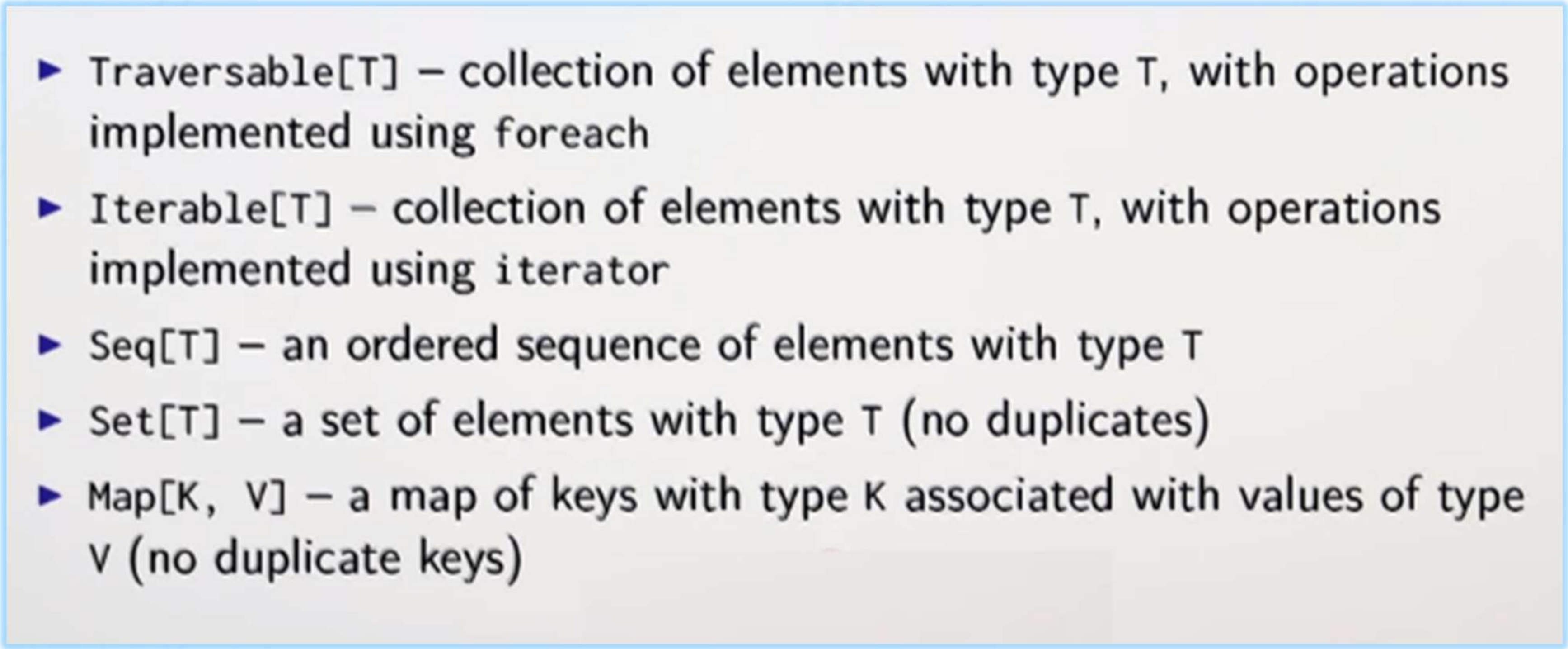
Once you have completed a Promise with the failure method, it is no longer writable, just as is the case with the success method. The associated Future will now be completed with a Failure, too, so the callback function above would run into the failure case.

Parallel Collections

Parallel Collections

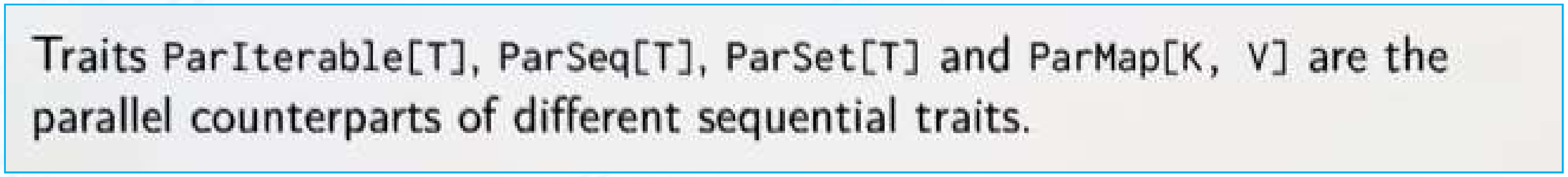
Parallel collections were included in the Scala standard library in an effort to facilitate parallel programming by sparing users from low-level parallelization details, meanwhile providing them with a familiar and simple high-level abstraction.

Collection Hierarchy



- `Traversable[T]` – collection of elements with type `T`, with operations implemented using `foreach`
- `Iterable[T]` – collection of elements with type `T`, with operations implemented using `iterator`
- `Seq[T]` – an ordered sequence of elements with type `T`
- `Set[T]` – a set of elements with type `T` (no duplicates)
- `Map[K, V]` – a map of keys with type `K` associated with values of type `V` (no duplicate keys)

Parallel Collections

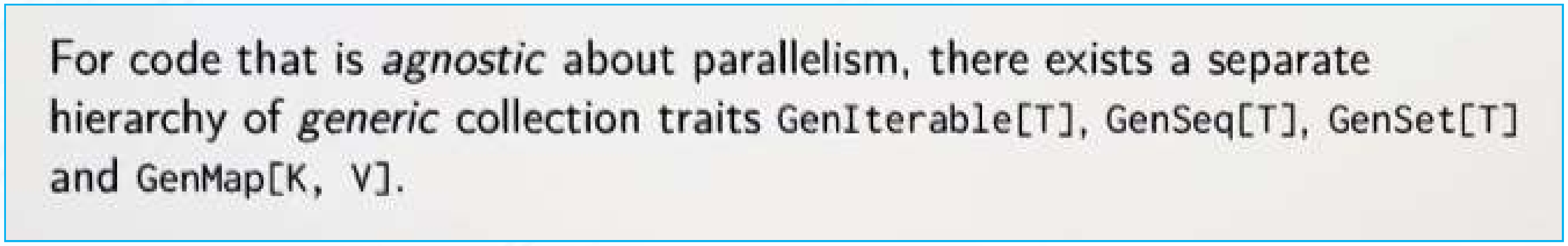


Traits ParIterable[T], ParSeq[T], ParSet[T] and ParMap[K, V] are the parallel counterparts of different sequential traits.

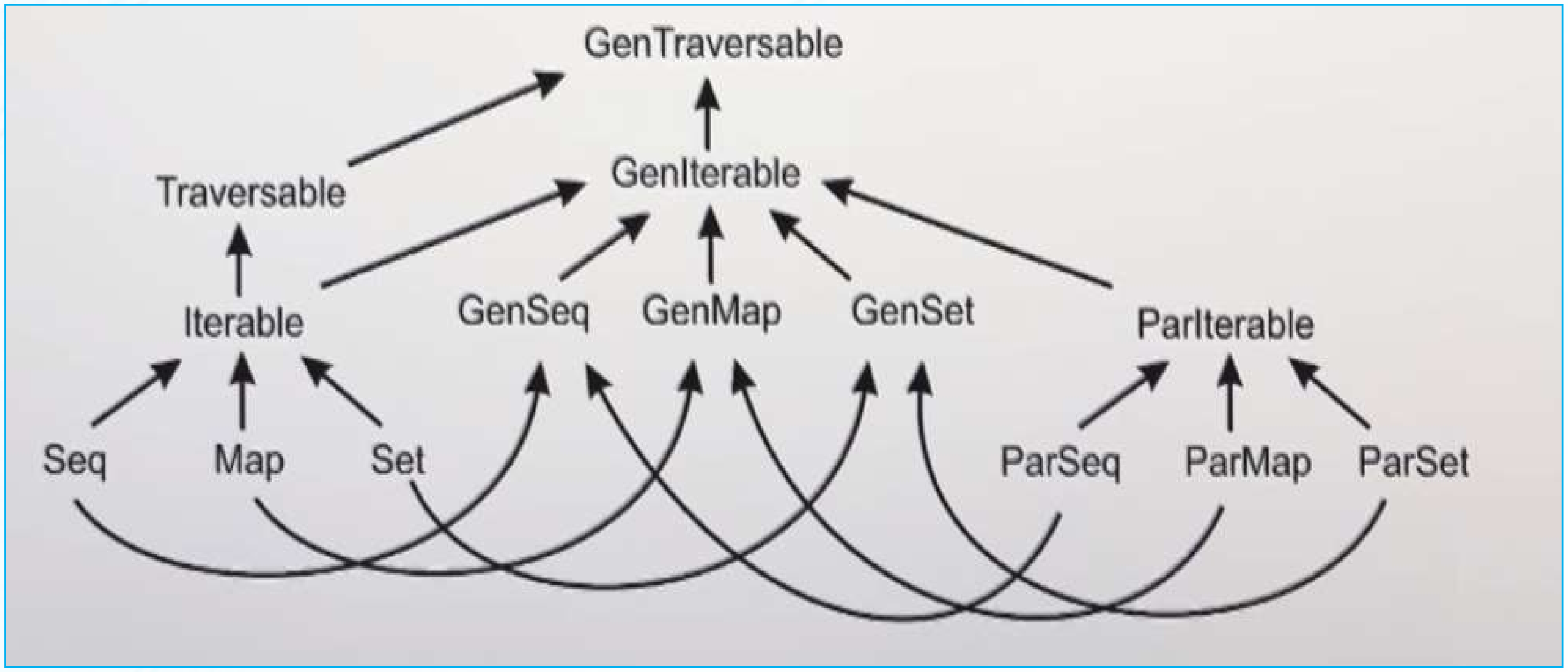
- ParArray
- ParVector
- mutable.ParHashMap
- mutable.ParHashSet
- immutable.ParHashMap
- immutable.ParHashSet
- ParRange

Concrete implementations of the above traits

Generic Collections


For code that is *agnostic* about parallelism, there exists a separate hierarchy of *generic* collection traits `GenIterable[T]`, `GenSeq[T]`, `GenSet[T]` and `GenMap[K, V]`.

Collection Hierarchy



Parallelism-agnostic Code

Generic collection traits allow us to write code that is unaware of parallelism

In other words, we can use either regular collections or parallised collections as a Generic collection.

Creating Parallel Collections

A sequential collection can be converted into a parallel collection by calling *.par* method.

```
val arr = (0 until 100000).toArray
val parArr = arr.par
val parVec = Vector(1,2,3,4,5,6,7,8,9).par
```

You can also create parallel collection directly:

```
import scala.collection.parallel.immutable.ParVector
val pv = new ParVector[Int]
```

Creating Parallel Collections

- Collections that are inherently sequential (in the sense that the elements must be accessed one after the other), like lists, queues, and streams, are converted to their parallel counterparts by copying the elements into a similar parallel collection.

- An example is **List** – it's converted into a standard immutable parallel sequence, which is a **ParVector**.

- The copying required for these collection types introduces an overhead not incurred by any other collection types, like Array, Vector, HashMap, etc.

Collection Semantics

Conceptually, parallel collections framework parallelizes an operation on a parallel collection by recursively “splitting” a given collection, applying an operation on each partition of the collection in parallel, and re-“combining” all of the results that were completed in parallel.

These concurrent, and “out-of-order” semantics of parallel collections lead to the following two implications:

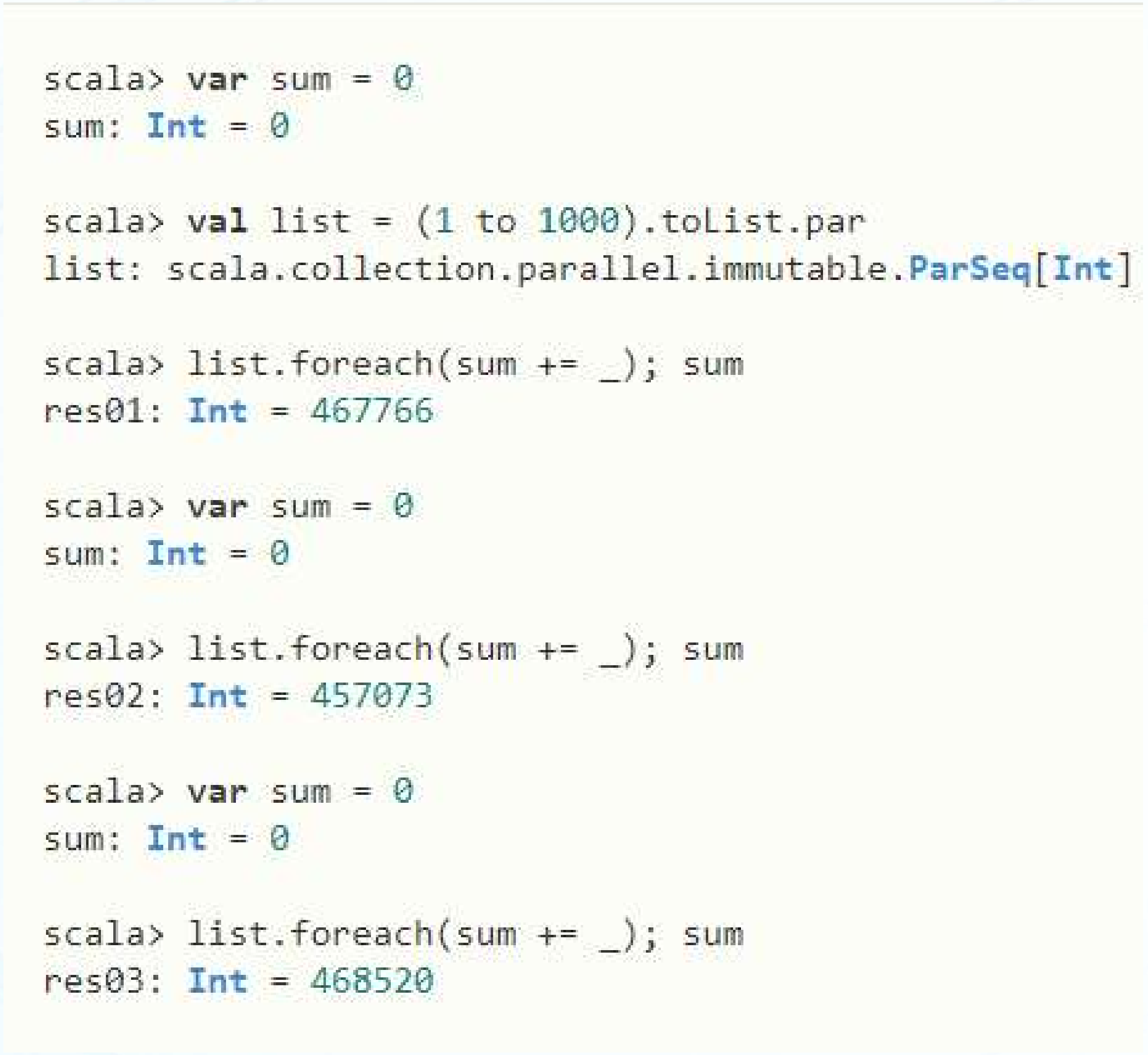
- **Side-effecting operations can lead to non-determinism**
- **Non-associative operations lead to non-determinism**

Side-Effecting Operations

- Given the *concurrent* execution semantics of the parallel collections framework, operations performed on a collection which cause side-effects should generally be avoided, in order to maintain determinism.

- A simple example is by using an accessor method, like `foreach` to increment a `var` declared outside of the closure which is passed to `foreach`.

Side-Effecting Operations



```
scala> var sum = 0
sum: Int = 0

scala> val list = (1 to 1000).toList.par
list: scala.collection.parallel.immutable.ParSeq[Int]

scala> list.foreach(sum += _); sum
res01: Int = 467766

scala> var sum = 0
sum: Int = 0

scala> list.foreach(sum += _); sum
res02: Int = 457073

scala> var sum = 0
sum: Int = 0

scala> list.foreach(sum += _); sum
res03: Int = 468520
```

Non-Associative Operations

- Given this “out-of-order” semantics, also must be careful to perform only associative operations in order to avoid non-determinism.

- That is, given a parallel collection, `pcoll`, one should be sure that when invoking a higher-order function on `pcoll`, such as `pcoll.reduce(func)`, the order in which `func` is applied to the elements of `pcoll` can be arbitrary.

```
// Non-determinism due to non-associative operation
val pArray = (1 to 10000).toArray.par
println( pArray.fold(0)(_ - _) )
```

THANK YOU