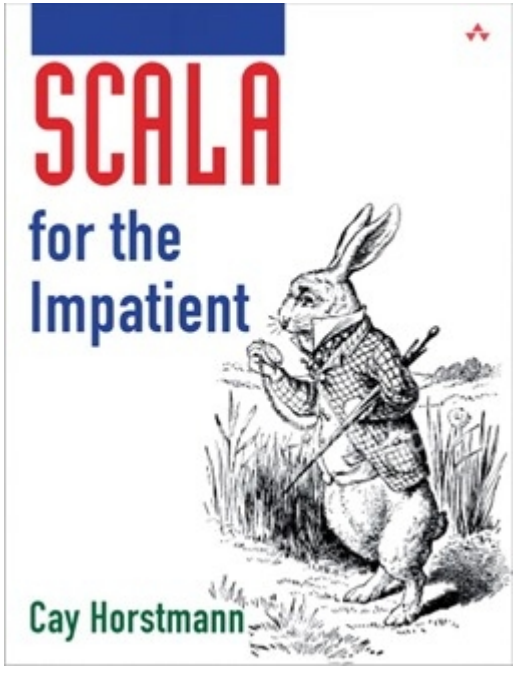
# Chapter 17. Futures



# Challenges

- Writing concurrent applications that work correctly and with high performance is very challenging.
- The traditional approach, in which concurrent tasks have side effects that mutate shared data, is tedious and error-prone.

# Scala’s solution

A computation yields a value, sometime in the future. As long as the computations don’t have side effects, you can let them run concurrently and combine the results when they become available.

- Future
- Promise

# Overview

- A block of code wrapped in a Future { ... } executes concurrently.
- A future succeeds with a result or fails with an exception.
- You can wait for a future to complete, but you don't usually want to.
- You can use callbacks to get notified when a future completes, but that gets tedious when chaining callbacks.
- Use methods such as map/flatMap, or the equivalent for expressions, to compose futures.
- A promise has a future whose value can be set (once), which gives added flexibility for implementing tasks that produce results.
- Pick an execution context that is suitable for the concurrent workload of your computation.

# 17.1 Running Tasks in the Future

```
import java.time._
import scala.concurrent._
import ExecutionContext.Implicits.global
Future {
	Thread.sleep(10000)
	println(s"This is the future at ${LocalTime.now}")
}
println(s"This is the present at ${LocalTime.now}")
```

When running this code, a line similar to the following is printed:
This is the present at 13:01:19.400

About ten seconds later, a second line appears:
This is the future at 13:01:29.140

# 17.1 Running Tasks in the Future

- Future code is run on some thread.
- Thread creation is not free.
- A data structure that assigns tasks to threads is usually called a thread pool.
- Scala uses the ExecutionContext trait .

Each Future must be constructed with a reference to an ExecutionContext.
The simplest way is to import:

```
import ExecutionContext.Implicits.global
```

Then the tasks execute on a global thread pool.

# 17.1 Running Tasks in the Future

When you construct multiple futures, they can execute concurrently.

```
Future { for (i <- 1 to 100) { print("A"); Thread.sleep(10) } }
Future { for (i <- 1 to 100) { print("B"); Thread.sleep(10) } }
```

Output could be:

ABABABABABABABABABABABABABA...AABABBBABABABA
BABABABABBBBBBBBBBBBBBBB

# 17.1 Running Tasks in the Future

```
val f2 = Future {
        if (LocalTime.now.getHour > 12)
                throw new Exception("too late")
        42
}
```

If it is after noon, the task terminates with an exception.

```
res14: scala.concurrent.Future[Int] = Future(Failure(java.lang.Exception: too late))
```

# 17.1 Running Tasks in the Future

- Future is an object that will give you a result (or failure) at some point in the future.

# 17.2 Waiting for Results

```
import scala.concurrent.duration._
val f = Future { Thread.sleep(10000); 42 }
val result = Await.result(f, 10.seconds)
```

- The call to Await.result blocks for ten seconds and then yields the result of the future.
- If the task is not ready by the allotted time → throws a TimeoutException

# 17.2 Waiting for Results

```
val f = Future { ... }
Await.ready(f, 10.seconds)
val Some(t) = f.value
```

- The value method returns an Option[Try[T]], which is None when the future is not completed and Some(t) when it is. Here, t is an object of the Try class, which holds either the result or the exception that caused the task to fail.

# 17.3 The Try Class

A Try[T] instance is either a Success(v), where v is a value of type T or a Failure(ex), where ex is a Throwable.

One way of processing it is with a match statement.

```
t match {
        case Success(v) => println(s"The answer is $v")
        case Failure(ex) => println(ex.getMessage)
}
```

# The Try Class

You can use the isSuccess or isFailure methods to find out whether the Try object represents success or failure.

```
if (t.isSuccess) println(s"The answer is ${t.get}")
```

```
if (t.isFailure) println(t.failed.get.getMessage)
```

You can also turn a Try object into an Option with the toOption method if you want to pass it on to a method that expects an option. This turns Success into Some and Failure into None.

# 17.4 Callbacks

As already mentioned, one does not usually use a blocking wait to get the result of a future. For better performance, the future should report its result to a callback function.

This is easy to arrange with the onComplete method.

```
f.onComplete(t => ...)
```

# 17.4 Callbacks

```
val f = Future { Thread.sleep(10000)
        if (random() < 0.5) throw new Exception
        42
}

f.onComplete {
        case Success(v) => println(s"The answer is $v")
        case Failure(ex) => println(ex.getMessage)
}
```

By using a callback, we avoid blocking.

# 17.5 Composing Future Tasks

Suppose we need to get some information from two web services and then combine the two. Each task is long-running and should be executed in a Future. It is possible to link them together with callbacks:

```
val future1 = Future { getData1() }
val future2 = Future { getData2() }
future1 onComplete {
      case Success(n1) =>
            future2 onComplete {
                  case Success(n2) => {
                        val n = n1 + n2
                        println(s"Result: $n")
                  }
                  case Failure(ex) => ...
            }
      case Failure(ex) => ...
      }
```

- The tasks run concurrently.
- We don’t know which of f1 and f2 completes first, and it doesn’t matter.
- We can’t process the result until both tasks complete.

Messy!

# 17.4 Callbacks

```
val future1 = Future { getData1() }
val future2 = Future { getData2() }
val combined = future1.map(n1 => future2.map(n2 => n1 + n2))
```

Here future1/future2 is a Future[Int]—a collection of (hopefully, eventually) one value.

When future1 and future2 have delivered their results, the sum is computed.

```
val combined = f1.flatMap(n1 => f2.map(n2 => n1 + n2))
```

# 17.4 Callbacks

```
val combined = for (n1 <- future1; n2 <- future2) yield n1 + n2
```

# 17.4 Callbacks

A Future starts execution immediately when it is created. To delay the creation, use Functions. To delay the creation, use functions.

```
def future1 = Future { getData() }
def future2 = Future { getMoreData() } // def, not val
val combined = for (n1 <- future1; n2 <- future2) yield n1 + n2
```

Now future2 is only evaluated when future1 has completed.

# 17.6 Other Future Transformations

The table (next slide) shows several ways of applying functions to the contents of a future that differ in subtle details.

| Method | Result | Description |
| --- | --- | --- |
| `collect(pf: PartialFunction[T, S])` | `Future[S]` | Like `map`, but with a partial function. The result fails with a `NoSuchElementException` if `pf(v)` is not defined. |
| `foreach(f: T => U)` | `Unit` | Calls `f(v)` like `map`, but only for its side effect. |
| `andThen(pf: PartialFunction[Try[T], U])` | `Future[T]` | Calls `pf(v)` for its side effect and returns a future with `v`. |
| `filter(p: T => Boolean)` | `Future[T]` | Calls `p(v)` and returns a future with `v` or a `NoSuchElementException`. |
| `recover(pf: PartialFunction[Throwable, U])`<br>`recoverWith(pf: PartialFunction[Throwable, Future[U]])` | `Future[U]` (where `U` is a supertype of `T`) | A future with value `v` or `pf(ex)`, flattened in the asynchronous case. |
| `fallbackTo(f2: Future[U])` | `Future[U]` (where `U` is a supertype of `T`) | A future with value `v`, or if this future failed, with the value of `f2`, or if that also failed, with exception `ex`. |
| `failed` | `Future[Throwable]` | A future with value `ex`. |
| `transform(s: T => S, f: Throwable => Throwable)`<br>`transform(f: Try[T] => Try[S])`<br>`transformWith(f: Try[T] => Future[Try[S]])` | `Future[S]` | Transforms both the success and failure. |

# 17.6 Other Future Transformations

The foreach method works exactly like it does for collections, applying a method for its side effect. The method is applied to the single value in the future. It is convenient for harvesting the answer when it materializes.

```
val combined = for (n1 <- future1; n2 <- future2) yield n1 + n2
combined.foreach(n => println(s"Result: $n"))
```

# 17.6 Other Future Transformations

The recover method accepts a partial function that can turn an exception into a successful result.

```
val f = Future { persist(data) } recover { case e: SQLException => 0 }
```

If a SQLException occurs, the future succeeds with result 0.

# 17.6 Other Future Transformations

The fallbackTo method provides a different recovery mechanism. When you call f.fallbackTo(f2), then f2 is executed if f fails, and its value becomes the value of the future. However, f2 cannot inspect the reason for the failure.

# 17.6 Other Future Transformations

Scala Documentation:
https://docs.scala-lang.org/overviews/core/futures.html

# 17.7 Methods in the Future Object

The Future companion object contains useful methods for working on collections of futures.

Suppose that, as you are computing a result, you organize the work so that you can concurrently work on different parts. For example, each part might be a range of the inputs. Make a future for each part:

```
val futures = parts.map(p => Future { compute result in p })
```

Now you have a collection of futures.

# 17.7 Methods in the Future Object

By using the Future.sequence method, you can get a collection of all results for further processing:

```
val futures = parts.map(p => Future { compute result in p })
val result = Future.sequence(futures);
```

When the results for all elements of futures are available, the result future will complete with a set of the results.

```
val result = Future.traverse(parts)(p => Future { compute result in p })
```

# 17.7 Methods in the Future Object

```
Future[T] result = Future.firstCompletedOf(futures)
```

You get a future that, when it completes, has the result or failure of the first completed element of futures.

# 17.7 Methods in the Future Object

```
val result = Future.find(futures)(predicate)
                        // Yields a Future[Option[T]]
```

You get a future that, when it completes successfully, yields Some(r), where r is the result of one of the given futures that fulfills the predicate. Failed futures are ignored. If all futures complete but none yields a result that matches the predicate, then find returns None.

# 17.7 Methods in the Future Object

A potential problem with firstCompletedOf and find is that the other computations keep on going even when the result has been determined. Scala futures do not have a mechanism for cancellation. If you want to stop unnecessary work, you have to provide your own mechanism.

# 17.7 Methods in the Future Object

- Future.successful(r) is an already completed future with result r.
- Future.failed(e) is an already completed future with exception e.
- Future.fromTry(t) is an already completed future with the result or exception given in the Try object t.
- Future.unit is an already completed future with Unit result.
- Future.never is a future that never completes.

# 17.8 Promises

A Future object is read-only. The value of the future is set implicitly when the task has finished or failed. A Promise is similar, but the value can be set explicitly.

```
def computeAnswer(arg: String) = {
        val p = Promise[Int]()
        Future {
                val n = workHard(arg)
                p.success(n)
                workOnSomethingElse()
        }
        p.future
}
```

Calling future on a promise yields the associated Future object. Note that the method returns the Future right away, immediately after starting the task that will eventually yield the result. That task is run in another Future, defined by the expression Future { ... }, that is unrelated to the promise's future.

Don’t forget to sign up for the projects.

# Exercises