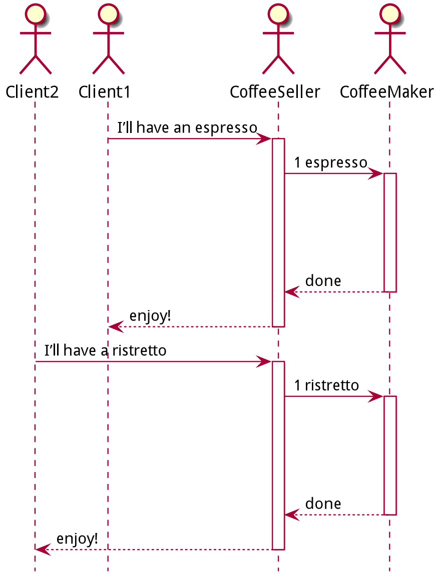
Asynchronous Programming with `Future`

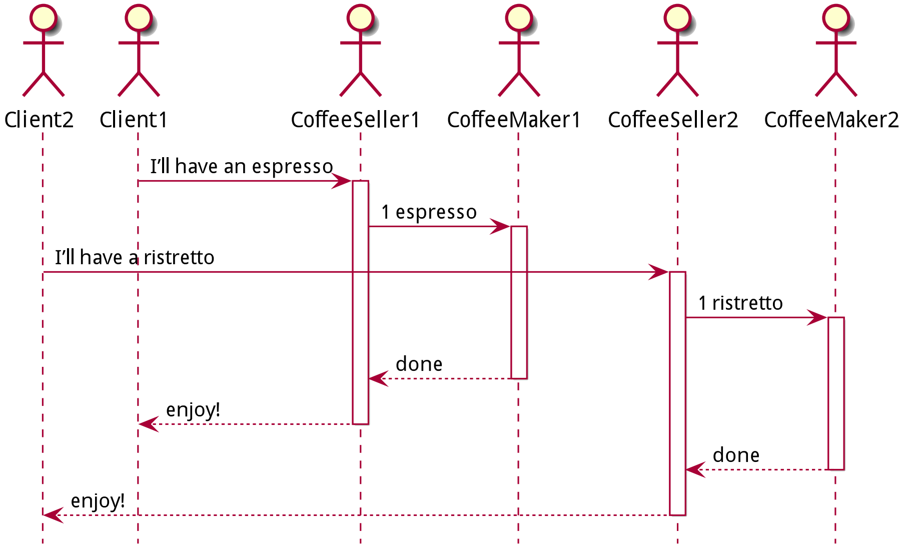
Principles of Functional Programming

Julien Richard-Foy, Martin Odersky

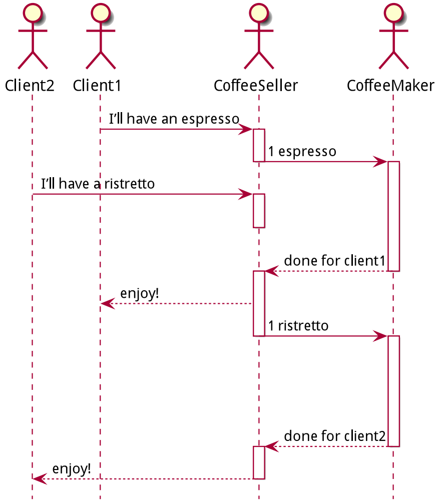
StarBlocks


Client2
Client1
CoffeeSeller
CoffeeMaker
I'll have an espresso
1 espresso
done
enjoy!
I'll have a ristretto
1 ristretto
done
enjoy!

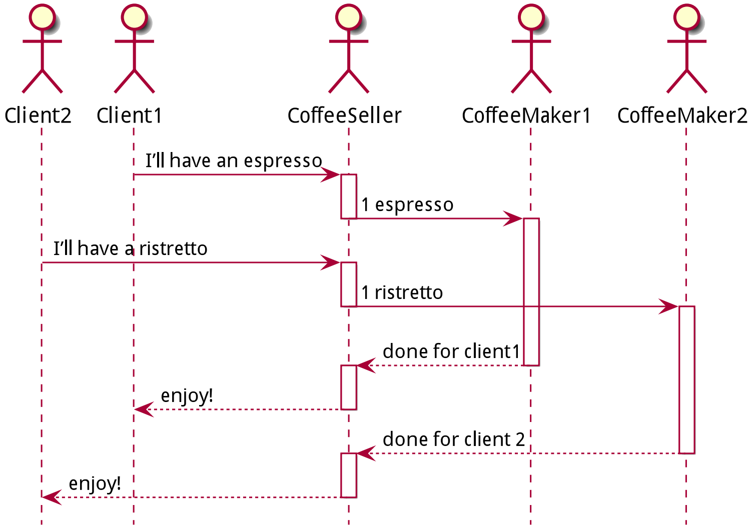
StarBlocks Scaled



ScalaBucks


Client2
Client1
CoffeeSeller
CoffeeMaker
I'll have an espresso
1 espresso
I'll have a ristretto
done for client1
enjoy!
1 ristretto
done for client2
enjoy!

ScalaBucks Scaled



Asynchronous Execution

- Execution of a computation on *another* computing unit, without *waiting* for its termination ;
- Better resource efficiency.

Concurrency Control of Asynchronous Programs

What if a program `A` *depends on* the result of an asynchronously executed program `B`?

```
def coffeeBreak(): Unit =
  val coffee = makeCoffee()
  drink(coffee)
  chatWithColleagues()
```

Callback

```scala
def makeCoffee(coffeeDone: Coffee => Unit): Unit =
  // work hard ...
  // ... and eventually
  val coffee = ...
  coffeeDone(coffee)

def coffeeBreak(): Unit =
  makeCoffee { coffee =>
   drink(coffee)
  }
  chatWithColleagues()
```

From Synchronous to Asynchronous Type Signatures

A synchronous type signature can be turned into an asynchronous type signature by:

- returning `Unit`
- and taking as parameter a **continuation** defining what to do after the return value has been computed

```
def program(a: A): B
```

```
def program(a: A, k: B => Unit): Unit
```

Combining Asynchronous Programs (1)

```scala
def makeCoffee(coffeeDone: Coffee => Unit): Unit = ...

def makeTwoCoffees(coffeesDone: (Coffee, Coffee) => Unit): Unit = ???
```

Combining Asynchronous Programs (2)

```
def makeCoffee(coffeeDone: Coffee => Unit): Unit = ...

def makeTwoCoffees(coffeesDone: (Coffee, Coffee) => Unit): Unit =
  var firstCoffee: Option[Coffee] = None
  val k = { coffee: Coffee =>
    firstCoffee match
      case None          => firstCoffee = Some(coffee)
      case Some(coffee2) => coffeesDone(coffee, coffee2)
  }
  makeCoffee(k)
  makeCoffee(k)
```

Callbacks All the Way Down (1)

What if another program *depends on* the coffee break to be done?

```
def coffeeBreak(): Unit = ...
```

- We need to make `coffeeBreak` take a callback too!

Callbacks all the Way Down (2)

```
def coffeeBreak(breakDone: Unit => Unit): Unit = ...

def workRoutine(workDone: Work => Unit): Unit =
  work { work1 =>
    coffeeBreak { _ =>
      work { work2 =>
        workDone(work1 + work2)
      }
    }
  }
```

Callbacks all the Way Down (2)

```
def coffeeBreak(breakDone: Unit => Unit): Unit = ...

def workRoutine(workDone: Work => Unit): Unit =
  work { work1 =>
    coffeeBreak { _ =>
      work { work2 =>
        workDone(work1 + work2)
      }
    }
  }
```

- Order of execution follows the indentation level!

Handling Failures

- In synchronous programs, failures are handled with exceptions ;
- What happens if an asynchronous call fails?
    - We need a way to propagate the failure to the call site

Handling Failures

- In synchronous programs, failures are handled with exceptions ;
- What happens if an asynchronous call fails?
  - We need a way to propagate the failure to the call site

```
def makeCoffee(coffeeDone: Try[Coffee] => Unit): Unit = ...
```

Summary

What we have seen so far:

- How to *sequence* asynchronous computations using **callbacks**
- Callbacks introduce complex type signatures
- The continuation passing style is tedious to use

From Synchronous to Asynchronous Type Signatures (using `Future`)

Remember the transformation we applied to a synchronous type signature to make it asynchronous:

```
def program(a: A): B
```

```
def program(a: A, k: B => Unit): Unit
```

From Synchronous to Asynchronous Type Signatures (using `Future`)

Remember the transformation we applied to a synchronous type signature to make it asynchronous:

```
def program(a: A): B
```

```
def program(a: A, k: B => Unit): Unit
```

What if we could model an asynchronous result of type `T` as a return type `Future[T]`?

```
def program(a: A): Future[B]
```

From Continuation Passing Style to `Future`

```
def program(a: A, k: B => Unit): Unit
```

Let's massage this type signature...

From Continuation Passing Style to Future

```scala
def program(a: A, k: B => Unit): Unit
```

Let's massage this type signature...

```scala
// by currying the continuation parameter
def program(a: A): (B => Unit) => Unit
```

From Continuation Passing Style to Future

```
def program(a: A, k: B => Unit): Unit
```

Let's massage this type signature...

```
// by currying the continuation parameter
def program(a: A): (B => Unit) => Unit

// by introducing a type alias
type Future[+T] = (T => Unit) => Unit
def program(a: A): Future[B]
```

From Continuation Passing Style to Future

```
def program(a: A, k: B => Unit): Unit
```

Let's massage this type signature...

```
// by currying the continuation parameter
def program(a: A): (B => Unit) => Unit

// by introducing a type alias
type Future[+T] = (T => Unit) => Unit
def program(a: A): Future[B]

// bonus: adding failure handling
type Future[+T] = (Try[T] => Unit) => Unit
```

Towards a Brighter `Future`

```scala
type Future[+T] = (Try[T] => Unit) => Unit
```

Towards a Brighter Future

```
type Future[+T] = (Try[T] => Unit) => Unit

// by reifying the alias into a proper trait
trait Future[+T] extends ((Try[T] => Unit) => Unit):
  def apply(k: Try[T] => Unit): Unit
```

Towards a Brighter Future

```
type Future[+T] = (Try[T] => Unit) => Unit

// by reifying the alias into a proper trait
trait Future[+T] extends ((Try[T] => Unit) => Unit):
  def apply(k: Try[T] => Unit): Unit

// by renaming ‘apply‘ to ‘onComplete‘
trait Future[+T]:
  def onComplete(k: Try[T] => Unit): Unit
```

coffeeBreak Revisited With Future

```
def makeCoffee(): Future[Coffee] = ...

def coffeeBreak(): Unit =
  val eventuallyCoffee = makeCoffee()
  eventuallyCoffee.onComplete { tryCoffee =>
    tryCoffee.foreach(drink)
  }
  chatWithColleagues()
```

Handling Failures

```scala
def makeCoffee(): Future[Coffee] = ...

def coffeeBreak(): Unit =
  makeCoffee().onComplete {
    case Success(coffee) => drink(coffee)
    case Failure(reason) => ...
  }
  chatWithColleagues()
```

Handling Failures

```
def makeCoffee(): Future[Coffee] = ...

def coffeeBreak(): Unit =
  makeCoffee().onComplete {
    case Success(coffee) => drink(coffee)
    case Failure(reason) => ...
  }
  chatWithColleagues()
```

- However, most of the time you want to transform a successful result and delay failure handling to a later point in the program

Transformation Operations

- `onComplete` suffers from the same composability issues as callbacks
- `Future` provides convenient high-level transformation operations

(Simplified) API of `Future`:

```
trait Future[+A]:
  def onComplete(k: Try[A] => Unit): Unit
  // transform successful results
  def map[B](f: A => B): Future[B]
  def flatMap[B](f: A => Future[B]): Future[B]
  def zip[B](fb: Future[B]): Future[(A, B)]
  // transform failures
  def recover(f: Exception => A): Future[A]
  def recoverWith(f: Exception => Future[A]): Future[A]
```

map Operation on Future

```
trait Future[+A]:
  ...
  def map[B](f: A => B): Future[B]
```

- Transforms a successful `Future[A]` into a `Future[B]` by applying a function `f: A => B` after the `Future[A]` has completed
- Automatically propagates the failure of the former `Future[A]` (if any), to the resulting `Future[B]`

```
def grindBeans(): Future[GroundCoffee]
def brew(groundCoffee: GroundCoffee): Coffee

def makeCoffee(): Future[Coffee] =
  grindBeans().map(groundCoffee => brew(groundCoffee))
```

flatMap Operation on Future

```
trait Future[+A]:
  ...
  def flatMap[B](f: A => Future[B]): Future[B]
```

- Transforms a successful `Future[A]` into a `Future[B]` by applying a function `f: A => Future[B]` after the `Future[A]` has completed
- Returns a failed `Future[B]` if the former `Future[A]` failed or if the `Future[B]` resulting from the application of the function `f` failed.

```
def grindBeans(): Future[GroundCoffee]
def brew(groundCoffee: GroundCoffee): Future[Coffee]

def makeCoffee(): Future[Coffee] =
  grindBeans().flatMap(groundCoffee => brew(groundCoffee))
```

`zip` Operation on `Future`

```
trait Future[+A]:
  ...
  def zip[B](other: Future[B]): Future[(A, B)]
```

- Joins two successful `Future[A]` and `Future[B]` values into a single successful `Future[(A, B)]` value
- Returns a failure if any of the two `Future` values failed
- Does *not* create any dependency between the two `Future` values!

```
def makeTwoCoffees(): Future[(Coffee, Coffee)] =
  makeCoffee().zip(makeCoffee())
```

zip vs flatMap

```scala
def makeTwoCoffees(): Future[(Coffee, Coffee)] =
  makeCoffee().zip(makeCoffee())
```

```scala
def makeTwoCoffees(): Future[(Coffee, Coffee)] =
  makeCoffee().flatMap { coffee1 =>
    makeCoffee().map(coffee2 => (coffee1, coffee2))
  }
```

zip vs flatMap (2)

```scala
def makeTwoCoffees(): Future[(Coffee, Coffee)] =
  makeCoffee().zip(makeCoffee())
```

```scala
def makeTwoCoffees(): Future[(Coffee, Coffee)] = {
  val eventuallyCoffee1 = makeCoffee()
  val eventuallyCoffee2 = makeCoffee()
  eventuallyCoffee1.flatMap { coffee1 =>
    eventuallyCoffee2.map(coffee2 => (coffee1, coffee2))
  }
}
```

Sequencing Futures (1)

```
def work(): Future[Work] = ...
def coffeeBreak(): Future[Unit] = ...

def workRoutine(): Future[Work] =
  work().flatMap { work1 =>
    coffeeBreak().flatMap { _ =>
      work().map { work2 =>
        work1 + work2
      }
    }
  }
```

Sequencing Futures (2)

```
def work(): Future[Work] = ...
def coffeeBreak(): Future[Unit] = ...

def workRoutine(): Future[Work] =
  for
    work1 <- work()
    _     <- coffeeBreak()
    work2 <- work()
  yield work1 + work2
```

- Back to a familiar layout to sequence computations!

coffeeBreak, Again

```
def coffeeBreak(): Future[Unit] =
  val eventuallyCoffeeDrunk = makeCoffee().flatMap(drink)
  val eventuallyChatted     = chatWithColleagues()
  eventuallyCoffeeDrunk.zip(eventuallyChatted)
    .map(_ => ())
```

recover and recoverWith Operations on Future

Turn a failed Future into a successful one

```
trait Future[+A]:
  ...
  def recover[B >: A](pf: PartialFunction[Throwable, B]): Future[B]
  def recoverWith[B >: A](pf: PartialFunction[Throwable, Future[B]]): Future[B]

grindBeans()
  .recoverWith { case BeansBucketEmpty =>
    refillBeans().flatMap(_ => grindBeans())
  }
  .flatMap(coffeePowder => brew(coffeePowder))
```

Execution Context

- So far, we haven't said anything about where continuations are executed, *physically*
- How do we control that?
  - Single thread? Fixed size thread pool?

Execution Context

- So far, we haven't said anything about where continuations are executed, *physically*
- How do we control that?
  - Single thread? Fixed size thread pool?

```
trait Future[+A]:
  def onComplete(k: Try[A] => Unit)(using ExecutionContext): Unit

import scala.concurrent.ExecutionContext.Implicits.global
```

Lift a Callback-Based API to Future (1)

```scala
def makeCoffee(
  coffeeDone: Coffee => Unit,
  onFailure: Exception => Unit
): Unit

def makeCoffee2(): Future[Coffee] = ...
```

Lift a Callback-Based API to Future (2)

```scala
def makeCoffee(
  coffeeDone: Coffee => Unit,
  onFailure: Exception => Unit
): Unit

def makeCoffee2(): Future[Coffee] =
  val p = Promise[Coffee]()
  makeCoffee(
    coffee => p.trySuccess(coffee),
    reason => p.tryFailure(reason)
  )
  p.future
```

Making it Run in Parallel

```scala
def makeCoffee(
  coffeeDone: Coffee => Unit,
  onFailure: Exception => Unit
): Unit

def makeCoffee2(): Future[Coffee] =
  val p = Promise[Coffee]()
  execute { // run in parallel
    makeCoffee(
      coffee => p.trySuccess(coffee),
      reason => p.tryFailure(reason)
    )
  }
  p.future
```

Summary

In this video, we have seen:

- The `Future[T]` type is an equivalent alternative to continuation passing
- Offers convenient *transformation* and *failure recovering* operations
- `map` and `flatMap` operations introduce *sequentiality*