ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Asynchronous Programming with `Future`

Programming Reactive Systems

Julien Richard-Foy

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

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

Let's massage this type signature...

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
```

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

```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

// 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

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

Towards a Brighter Future

```scala
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

```scala
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

```scala
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]`

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

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

`flatMap` Operation on `Future`

```scala
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.

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

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

zip Operation on Future

```scala
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!

```scala
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)

```scala
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?

```scala
trait Future[+A] {
  def onComplete(k: Try[A] => Unit)(implicit ec: 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
}
```

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*