Asynchronous Programming with Futures

CS-214 Software Construction

Asynchronous Computations

A common pattern for computing is

- We launch a task
- At some time later in the program we need the result of the task.
- If there are no side effects, the task can be executed concurrently with the other program.

Why is this beneficial?

- The task might take time to compute, so we can use parallel computing resources to save time.
- The task might wait for some external event and we do not want to block the program as a whole.

Example: Preparing Breakfast

```scala
import scala.util.
  .{Try, Success, Failure, Random}
import collection.mutable.ListBuffer

val rand = Random()

class Beans
class Coffee(beans: Beans)
class Croissant

class Burnt extends Exception
```

Example: Preparing Breakfast

```scala
import scala.util.
  .{Try, Success, Failure, Random}
import collection.mutable.ListBuffer

val rand = Random()

class Beans
class Coffee(beans: Beans)
class Croissant

class Burnt extends Exception

def grindBeans() =
  Thread.sleep(500)
  Beans()
```

Example: Preparing Breakfast

```scala
import scala.util.
  .{Try, Success, Failure, Random}
import collection.mutable.ListBuffer

val rand = Random()

class Beans
class Coffee(beans: Beans)
class Croissant

class Burnt extends Exception

def grindBeans() =
  Thread.sleep(500)
  Beans()
```

```scala
def brewCoffee(beans: Beans) =
  Thread.sleep(500)
  Coffee(beans)
```

Example: Preparing Breakfast

```scala
import scala.util.
  .{Try, Success, Failure, Random}
import collection.mutable.ListBuffer

val rand = Random()

class Beans
class Coffee(beans: Beans)
class Croissant

class Burnt extends Exception

def grindBeans() =
  Thread.sleep(500)
  Beans()
```

```scala
def brewCoffee(beans: Beans) =
  Thread.sleep(500)
  Coffee(beans)

def bakeCroissant(): Croissant =
  Thread.sleep(5000)
  if rand.nextInt(5) == 0 then throw Burnt()
  Croissant()
```

Example: Preparing Breakfast

```scala
import scala.util.
  .{Try, Success, Failure, Random}
import collection.mutable.ListBuffer

val rand = Random()

class Beans
class Coffee(beans: Beans)
class Croissant

class Burnt extends Exception

def grindBeans() =
  Thread.sleep(500)
  Beans()
```

```scala
def brewCoffee(beans: Beans) =
  Thread.sleep(500)
  Coffee(beans)

def bakeCroissant(): Croissant =
  Thread.sleep(5000)
  if rand.nextInt(5) == 0 then throw Burnt()
  Croissant()

def makeCoffee(): Coffee =
  val beans = grindBeans()
  brewCoffee(beans)
```

Example: Preparing Breakfast

```scala
import scala.util.
  .{Try, Success, Failure, Random}
import collection.mutable.ListBuffer

val rand = Random()

class Beans
class Coffee(beans: Beans)
class Croissant

class Burnt extends Exception

def grindBeans() =
  Thread.sleep(500)
  Beans()
```

```scala
def brewCoffee(beans: Beans) =
  Thread.sleep(500)
  Coffee(beans)

def bakeCroissant(): Croissant =
  Thread.sleep(5000)
  if rand.nextInt(5) == 0 then throw Burnt()
  Croissant()

def makeCoffee(): Coffee =
  val beans = grindBeans()
  brewCoffee(beans)

def makeBreakfast(): (Coffee, Croissant) =
  (makeCoffee(), bakeCroissant())
```

Examples Ctd

```
@main def demo1 =
  try
    makeBreakfast()
  catch case ex: Burnt =>
    println("sorry, no croissant today")

@main def demo2 =
  def compute(x: Double): Double =
    Thread.sleep(1000)
    x * x

  val xs = (1 to 1000).map(compute(_))
  println(xs.sum)
```

- Problem: Everything is sequential.
- How can we better make use of parallelism?
- fork/join from parallel programming is close but too restrictive

The Rest of This Lecture

I'll present *futures* as a solution to asynchronous computation, in three parts:

Part 1: Simple futures for lenient evaluation.

- This works, but is not very scalable.

The Rest of This Lecture

I'll present *futures* as a solution to asynchronous computation, in three parts:

Part 1: Simple futures for lenient evaluation.

- This works, but is not very scalable.

Part 2: Completable futures that implement inversion of control.

- They are very scalable, but a harder to use.

The Rest of This Lecture

I'll present *futures* as a solution to asynchronous computation, in three parts:

Part 1: Simple futures for lenient evaluation.

- This works, but is not very scalable.

Part 2: Completable futures that implement inversion of control.

- They are very scalable, but a harder to use.

Part 3: Direct style futures with suspensions.

- They combine simplicity with scalability.
- They need new runtime features (continuations or coroutines) to work.

Part 1: Simple Futures

Part 1: Simple Futures

Idea: Split the points where a task is defined and where its result is asked.

Definition:

```
val f = SimpleFuture{ <some task> }
```

Asking:

```
f.await
```

Breakfast with simple Futures

```
def grindBeans() =
  SimpleFuture:
    Thread.sleep(500)
    Beans()

def brewCoffee(beans: Beans) =
  SimpleFuture:
    Thread.sleep(500)
    Coffee(beans)

def bakeCroissant() =
  SimpleFuture:
    Thread.sleep(5000)
    if rand.nextInt(5) == 0 then
      throw Burnt()
    Croissant()
```

```
def makeCoffee(): SimpleFuture[Coffee] =
  SimpleFuture:
    val beans = grindBeans()
    brewCoffee(beans)

def makeBreakfast()
  : SimpleFuture[(Coffee, Croissant)]
  = SimpleFuture:
      ( makeCoffee().await,
        bakeCroissant().await )
```

Breakfast with simple Futures

```scala
def grindBeans() =
  SimpleFuture:
    Thread.sleep(500)
    Beans()

def brewCoffee(beans: Beans) =
  SimpleFuture:
    Thread.sleep(500)
    Coffee(beans)

def bakeCroissant() =
  SimpleFuture:
    Thread.sleep(5000)
    if rand.nextInt(5) == 0 then
      throw Burnt()
    Croissant()
```

```scala
def makeCoffee(): SimpleFuture[Coffee] =
  SimpleFuture:
    val beans = grindBeans()
    brewCoffee(beans)

def makeBreakfast()
  : SimpleFuture[(Coffee, Croissant)]
  = SimpleFuture:
      ( makeCoffee().await,
        bakeCroissant().await )
      // Problem: this is still sequential!
```

Breakfast with Futures - In Parallel

```
def grindBeans() =
  SimpleFuture:
    Thread.sleep(500)
    Beans()

def brewCoffee(beans: Beans) =
  SimpleFuture:
    Thread.sleep(500)
    Coffee(beans)

def bakeCroissant() =
  SimpleFuture:
    Thread.sleep(5000)
    if rand.nextInt(5) == 0 then
      throw Burnt()
    Croissant()
```

```
def makeCoffee(): SimpleFuture[Coffee] =
  SimpleFuture:
    val beans = grindBeans()
    brewCoffee(beans)

def makeBreakfast()
  : SimpleFuture[(Coffee, Croissant)] =
  val coffeeFuture = makeCoffee()
  val croissantFuture = bakeCroissant()
  SimpleFuture:
    ( coffeeFuture.await,
      croissantFuture.await) )
    // Now it's parallel
```

Main Programs

```
@main def demo3 =
  try
    makeBreakfast().await
  catch case ex: Burnt =>
    println("sorry, no croissant today")
```

```
@main def demo4 =
  def compute(x: Double): SimpleFuture[Double]
    SimpleFuture:
      Thread.sleep(1000)
      x * x

  val fs = (1 to 1000).map(compute(_))
  val xs = fs.map(_.await)
  println(xs.sum)
```

Lenient Evalution

We have seen two evaluation strategies so far:

- Strict evaluation: An expression gets evaluated when it is defined
- Lazy evaluation: An expression gets evaluated when its value is needed.

Lenient evaluation is a third option, between the two:

- An expression can be evaluated *as soon as* it is defined, and must be evaluated once its value is needed.

Lenient evaluation is great for exploiting parallelism.

It's related to *dataflow programming*: Evaluate when the data is ready.

A Simple Implementation of Direct-Style Futures

```scala
class SimpleFuture[T](body: => T):
  private var status: Option[Try[T]] = None

  private val thread = new Thread:
    override def run(): Unit =
      status = Some(Try(body))
    start()

  def awaitTry: Try[T] =                     // Futures always return
    if status.isEmpty then thread.join()  // either Success or Failure
    status.get

  def await: T = awaitTry.get // Short form: throw exception on Failure
end SimpleFuture
```

A Problem

What happens if the increase the number of threads in the previous program by a factor of 10?

```
@main def demo4 =
  def compute(x: Double): SimpleFuture[Double] =
    SimpleFuture:
      Thread.sleep(10000)
      x * x
  val fs = (1 to 1000).map(compute(_))
  val xs = fs.map(_.await)
  println(xs.sum)
```

A Problem

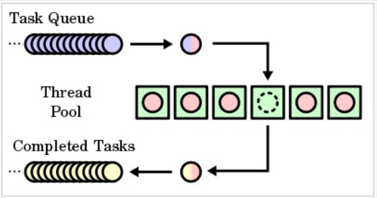
What happens if the increase the number of threads in the previous program by a factor of 10?

```scala
@main def demo4 =
  def compute(x: Double): SimpleFuture[Double] =
    SimpleFuture:
      Thread.sleep(10000)
      x * x
  val fs = (1 to 1000).map(compute(_))
  val xs = fs.map(_.await)
  println(xs.sum)
```

```
[0.868s][warning][os,thread] Failed to start thread - pthread_create failed (EAGAIN) for
Exception in thread "main" java.lang.OutOfMemoryError: unable to create native thread:
possibly out of memory or process/resource limits reached
```

A Problem

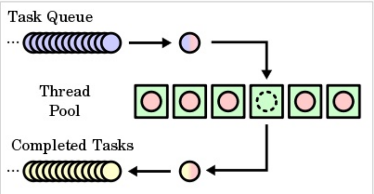
- Threads are a scarce resource; can't have too many of them before running out. (Typically, low 1000s.)
- Going to tasks helps, because tasks are multiplexed over worker threads.
- But it does not help if the tasks block for external events.
- Example: Web server: one task per connection. You might easily block every waiting thread in a task that waits on the connection.



A sample thread pool (green boxes) with waiting tasks (blue) and completed tasks (yellow)

A Problem

- Threads are a scarce resource; can't have too many of them before running out. (Typically, low 1000s.)
- Going to tasks helps, because tasks are multiplexed over worker threads.
- But it does not help if the tasks block for external events.
- Example: Web server: one task per connection. You might easily block every waiting thread in a task that waits on the connection.
- Besides, there are also single-threaded runtimes (for instance, JS). What do we do for these?



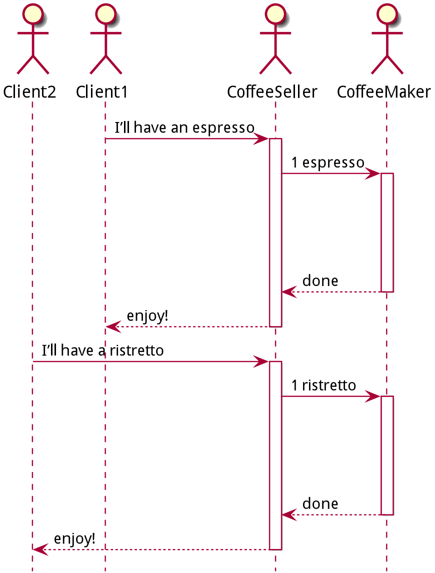
A sample thread pool (green boxes) with waiting tasks (blue) and completed tasks (yellow)

Part 2: Asynchronous Execution

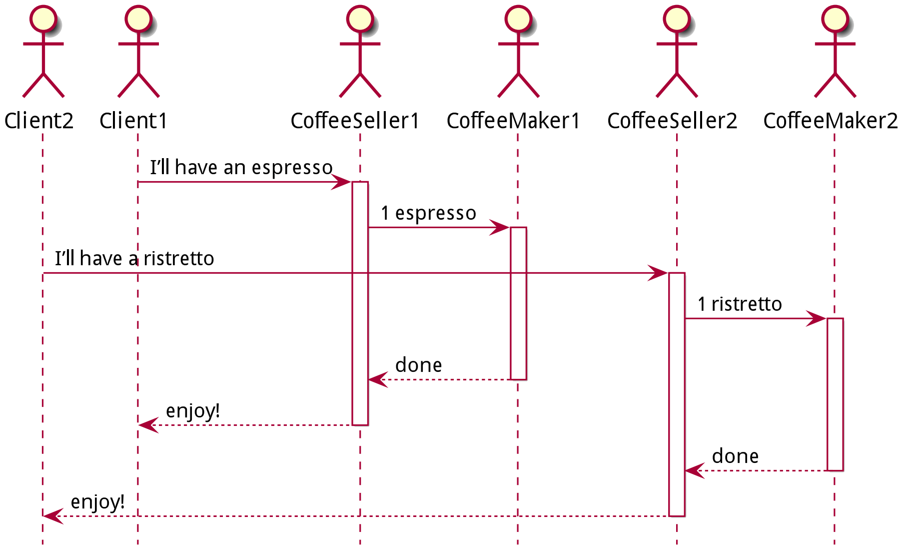
One area where this problem comes up all the time is in *asynchronous execution*. That is,

- Execution of a computation on *another* computing unit, without *waiting* for its termination.
- This gives better resource efficiency.

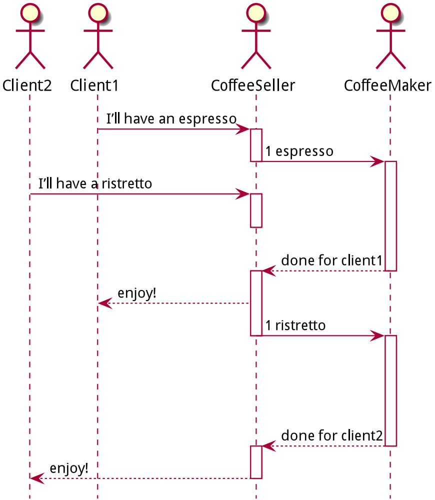
StarBlocks


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

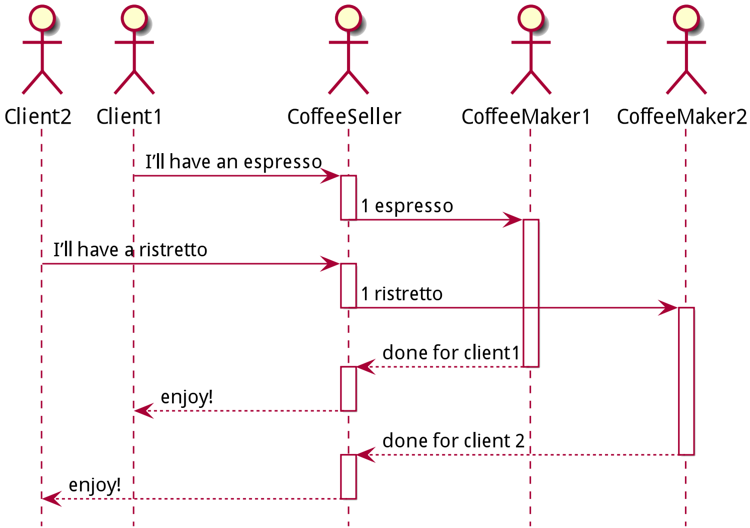
StarBlocks Scaled



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



Don’t Call Us - We’ll Call You!

A Solution: *Inversion of Control*.

Instead of running 100K threads, we prepare 100K work items that are run from some scheduler thread(s).

Every work item is a closure.

It has to run to completion without blocking.

So if it needs another thing to happen it has to prepare that as a work item and register to be called when it is completed.

This pattern is called a *callback*.

Setting Things Up

Here's a minimal scheduler implementation.

```
object scheduler:
  private val tasks: ListBuffer[() => Unit] = ListBuffer()

  def schedule(task: => Unit) =
    tasks.insert(rand.nextInt(tasks.size + 1), () => task)

  def run(): Unit =
    while tasks.nonEmpty do
      val task = tasks.remove(0)
      task()
end scheduler
```

To simulate that waiting times for events are unpredictable, `schedule` inserts a given task in a random position in the `tasks` queue.

Callback-Based Breakfasts

```
def grindBeans
    (callback: Beans => Unit): Unit =
  schedule:
    println("grinding beans...")
    callback(Beans())

def brew
    (beans: Beans)
    (callback: Coffee => Unit): Unit =
  schedule:
    println("brewing coffee...")
    callback(Coffee(beans))
```

```
def makeCoffee
    (callback: Coffee => Unit): Unit =
  grindBeans: beans =>
    brew(beans): coffee =>
      callback(coffee)

def bakeCroissant
    (callback: Croissant => Unit): Unit =
  schedule:
    println("baking croissant...")
    if rand.nextInt(3) == 0 then
      throw Burnt()
    callback(Croissant())
```

Waiting for Two Callbacks

To make breakfast, we have to wait for *both* the coffee and the croissant to be finished.

We could do it in sequence:

```
def makeBreakfast(serve: (Coffee, Croissant) => Unit) =
  makeCoffee: coffee =>
    bakeCroissant: croissant =>
      serve(coffee, croissant)
```

But that would not exploit possible sources of parallelism in the task execution.

Waiting for Two Callbacks in Parallel

- To wait for two callbacks in parallel we need to launch two new tasks and wait for each one separately.
- Need state to keep track of which callback arrived first.

```
def makeBreakfast(serve: (Coffee, Croissant) => Unit) =
  var myCoffee: Option[Coffee] = None
  var myCroissant: Option[Croissant] = None
  makeCoffee: coffee =>
    myCroissant match
      case Some(croissant) => serve(coffee, croissant)
      case None => myCoffee = Some(coffee)
  bakeCroissant: croissant =>
    myCoffee match
      case Some(coffee) => serve(coffee, croissant)
      case None => myCroissant = Some(croissant)
```

Callback-based main

```
@main def demo5 =
  try
    makeBreakfast: (_, _) =>
      println("breakfast served!")
  catch case ex: Burnt =>
    println("sorry, no croissant today")
  scheduler.run()
```

Trying it out:

```
grinding beans...
brewing coffee...
baking croissant...
breakfast served!
```

Callback-based main

```
@main def demo5 =
  try
    makeBreakfast: (_, _) =>
      println("breakfast served!")
  catch case ex: Burnt =>
    println("sorry, no croissant today")
  scheduler.run()
```

Trying once more:

```
baking croissant...
Exception in thread "main" Burnt
  at callback$.bakeCroissant$$anon
```

- Where did the exception come from?
- I thought we handled it?

Callback-based main

```
@main def demo5 =
  try
    makeBreakfast: (_, _) =>
      println("breakfast served!")
  catch case ex: Burnt =>
    println("sorry, no croissant today")
  scheduler.run()
```

Trying once more:

```
baking croissant...
Exception in thread "main" Burnt
  at callback$.bakeCroissant$$anon
```

- Where did the exception come from?
- I thought we handled it?

- Need to add error handling.
- Instead of *throwing* exceptions, need to propagate them in a `Try`.

So instead of `Croissant => Unit`
as a handler type,
it should really be `Try[Croissant] => Unit`

Critique of Callbacks

Callbacks are a direct way to solve inversion of control.

But they have shortcomings.

Critique of Callbacks

Callbacks are a direct way to solve inversion of control.

But they have shortcomings.

Long sequential dependencies lead to code that drifts off towards the right

```
queryWhere: callback1 =>
  queryWhen: callback2 =>
    findHotels: callback3 =>
      getRates: callBack4 =>
        getUserChoice: callBack5 =>
          getCreditCardInfo: callBack6 =>
            getUserConformation: callBack7 =>
              getHotelConfirmation: callBack8 =>
```

Critique of Callbacks

Parallel queries are hard to implement

```
def makeBreakfast(serve: (Coffee, Croissant) => Ur
  var myCoffee: Option[Coffee] = None
  var myCroissant: Option[Croissant] = None
  makeCoffee: coffee =>
    ...
  bakeCroissant: croissant =>
...
```

Critique of Callbacks

Parallel queries are hard to implement

```
def makeBreakfast(serve: (Coffee, Croissant) => Ur
  var myCoffee: Option[Coffee] = None
  var myCroissant: Option[Croissant] = None
  makeCoffee: coffee =>
    ...
  bakeCroissant: croissant =>
...
```

And we have not even yet started to do error handling properly!

From Synchronous to Asynchronous Type Signatures

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

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

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

From Synchronous to Asynchronous Type Signatures

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

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

```
def program(a: A, callback 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]
```

Like `SimpleFutures` before, `Futures` can also implement lenient evaluation but they do it via callbacks.

From Continuation Passing Style to Future

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

Let's massage this type signature...

From Continuation Passing Style to Future

```
def program(a: A, callback 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

```scala
def program(a: A, callback 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]
```

From Continuation Passing Style to Future

```scala
def program(a: A, callback 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
```

Fleshing Out Future

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

Fleshing Out 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(callback Try[T] => Unit): Unit
```

Fleshing Out `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(callback Try[T] => Unit): Unit

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

`Future` is essentially the `scala.concurrent.Future` trait from the standard library.

Breakfast with Future

If it was just for encapsulated oncomplete, our programs would not change much:

```
def grindBeans: Future[Beans] =
  Future:
    println("grinding beans...")
    Beans()

def brew(beans: Beans) =
  Future:
    println("brewing coffee...")
    Coffee(beans)
```

```
def makeCoffee: Future[Coffee] =
  new Future:
    override def onComplete(
        callback: Try[Coffee] => Unit) =
      grindBeans.onComplete: beans =>
        brew(beans.get).onComplete(callback)

def bakeCroissant: Future[Croissant] =
  Future:
    println("baking croissant...")
    if rand.nextInt(3) == 0 then
      throw Burnt()
    Croissant()
```

Future Core API

But `Future` provides convenient high-level transformation operations.

```
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]`

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 quoteInDollars: Future[Double]
def USDtoCHF(usd: Double): Double

def quoteInCHF: Future[Double] =
  quoteInDollars.map(USDtoCHF)
```

`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.

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 quoteInDollars: Future[Double]
def askUSDtoCHFrate(usd: Double): Future[Double]

def quoteInCHF: Future[Double] =
  quoteInDollars.flatMap(askUSDtoCHFrate)
```

Simplifying Breakfast

(`grindBeans`, `brew`, `bakeCroissant` as before)

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

def makeBreakfast
    : Future[(Coffee, Croissant)] =
  makeCoffee.flatMap: coffee =>
    makeCroissant.map: croissant =>
      (coffee, croissant)
```

Simplifying Breakfast

Or, using `for`-expressions:

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

def makeBreakfast
    : Future[(Coffee, Croissant)] =
  makeCoffee.flatMap: coffee =>
    makeCroissant.map: croissant =>
      (coffee, croissant)
```

```
def makeCoffee: Future[Coffee] =
  for beans <- grindBeans;
      coffee <- brew(beans)
  yield coffee

def makeBreakfast
    : Future[(Coffee, Croissant)] =
  for
    coffee <- makeCoffee
    croissant <- bakeCroissant
  yield
    (coffee, croissant)
```

Simplifying Breakfast

Or, using `for`-expressions:

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

def makeBreakfast
    : Future[(Coffee, Croissant)] =
  makeCoffee.flatMap: coffee =>
    makeCroissant.map: croissant =>
      (coffee, croissant)
```

But is it parallel?

```
def makeCoffee: Future[Coffee] =
  for
    beans <- grindBeans
    coffee <- brew(beans)
  yield coffee

def makeBreakfast
    : Future[(Coffee, Croissant)] =
  for
    coffee <- makeCoffee
    croissant <- bakeCroissant
  yield
    (coffee, croissant)
```

Simplifying Breakfast

Getting back parallelism:

```
def makeBreakfast: Future[(Coffee, Croissant)] =
  val coffeeFuture = makeCoffee
  val croissantFuture = bakeCroissant
  for
    coffee <- coffeeFuture
    croissant <- croissantFuture
  yield (coffee, croissant)
```

Same principle as for direct-style futures:

- To gain parallelism, separate the point of definition of a future from the point where its result is used.

Simplifying Breakfast

Getting back parallelism:

```
def makeBreakfast: Future[(Coffee, Croissant)] =
  val coffeeFuture = makeCoffee
  val croissantFuture = bakeCroissant
  for
    coffee <- coffeeFuture
    croissant <- croissantFuture
  yield (coffee, croissant)
```

Same principle as for direct-style futures:

- To gain parallelism, separate the point of definition of a future from the point where its result is used.

Can we do even better? Think about error conditions...

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!

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 makeBreakfast: Future[(Coffee, Croissant)] =
  makeCoffee.zip(bakeCroissant)
```

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())
```

Implementing Future

Here is a simple implementation of completable futures.

```
object completable:
  trait Future[+T]:
    def onComplete(callback: Try[T] => Unit): Unit

  object Future:
    def apply[T](body: => T): Future[T] =
      val promise = Promise[T]
      schedule:
        promise.complete(Try(body))
      promise.future
  end Future
```

Promise is the part that completes a future

Promises

A `Promise` is used to complete a completable future.

It consists of two elements:

```
class Promise[T]:

  // The ‘complete‘ method is called to set the result
  def complete(result: Try[T]): Unit

  // A future that propagates the result to waiting futures
  val future: Future[T]
```

It has an internal state of this type:

```
enum State[T]:
  case Pending(callbacks: List[T => Unit])
  case Complete(result: T)
```

Implementing Promises

```
class Promise[T]:
  private var state: State[Try[T]] = Pending(Nil)

  def complete(result: Try[T]): Unit = state match
    case Pending(cs) =>
      state = Complete(result)
      for callback <- cs do callback(result)
    case _ =>

  val future = new Future[T]:
    def onComplete(c: Try[T] => Unit): Unit =
      state match
        case Complete(r) => c(r)
        case Pending(cs) => state = Pending(c :: cs)
```

Implementing `map` on Futures

```
def map[U](f: T => U) = new Future[U]:
```

Implementing `map` on Futures

```
def map[U](f: T => U) = new Future[U]:
  def onComplete(callback: Try[U] => Unit): Unit =
    ???
```

Implementing map on Futures

```
def map[U](f: T => U) = new Future[U]:
  def onComplete(callback: Try[U] => Unit): Unit =
    Future.this.onComplete:
      case Success(x) => ???
      case Failure(e) => ???
```

Implementing map on Futures

```scala
def map[U](f: T => U) = new Future[U]:
  def onComplete(callback: Try[U] => Unit): Unit =
    Future.this.onComplete:
      case Success(x) => callback(Try(f(x)))
      case Failure(e) => ???
```

Implementing map on Futures

```
def map[U](f: T => U) = new Future[U]:
  def onComplete(callback: Try[U] => Unit): Unit =
    Future.this.onComplete:
      case Success(x) => callback(Try(f(x)))
      case Failure(e) => callback(Failure(e))
```

Exercises:

1. Implement `flatMap` on `Future`
2. Implement `zip` on `Future`

Finishing Breakfast

To run `breakFast`, we can use this code: ...

```
def makeBreakfast: Future[(Coffee, Croissant)] =
  makeCoffee.zip(bakeCroissant)

@main def demo7 =
  makeBreakfast.onComplete:
    case Success(_) => println(”breakfast served!”)
    case Failure(_) => println(”sorry, no croissant today”)
  scheduler.run()
```

Using Standard Futures

Your program is largely unchanged when it uses `scala.concurrent.Future` instead of `completable.Future`.

Main difference: there's no explicit scheduler. Instead, you use a standard scheduler that's defined in a an `ExecutionContext`.

```
package scala.concurrent

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

And your program should

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

Awaiting Futures

Since there is no scheduler, your main program does not need to invoke

```
scheduler.run()
```

at the end.

But, beware! This does not work correctly:

```
@main def demo7 =
  makeBreakfast.onComplete:
    case Success(_) => println("breakfast served!")
    case Failure(_) => println("sorry, no croissant today")
```

The problem is that demo7 will likely exit before we have a chance to run any of the tasks defined by its futures.

Awaiting Futures

To fix this, we have to use a *blocking* wait for the main future (i.e. the result of `makeBreakfast`).

Ideally, we could write this:

```
@main def demo8 =
  makeBreakfast.awaitTry:
    case Success(_) => println("breakfast served!")
    case Failure(_) => println("sorry, no croissant today")
```

This should block the main thread until the `makeBreakfast` future has completed.

Unfortunately, `awaitTry` does not exist in the standard library, but we can make it available as an extension method.

Implementing `awaitTry`

```scala
import scala.concurrent.{Future, Await}
import scala.concurrent.duration.Duration

extension [T](f: Future[T])
  def awaitTry: Try[T] =
    Try(Await.result(f, Duration.Inf))
```

Implementing awaitTry

```scala
import scala.concurrent.{Future, Await}
import scala.concurrent.duration.Duration

extension [T](f: Future[T])
  def awaitTry: Try[T] =
    Try(Await.result(f, Duration.Inf))
```

Q Why is it so complicated?

A Completable futures were originally build for servers (one of the original designers was a team at Twitter)

Servers don't exit and nothing should ever block. Everything is async.

So the designers made it intentionally hard to do a blocking wait on a future.

Completable Futures in Industry

Futures or something like it power many large server installations.

Examples: Twitter, Disney+, Netflix, Coursera, Duolingo, ...

There are three main variants:

- standard `scala.concurrent.Future`
- ZIO
- Cats Effect

The second two do asynchronous execution as a part of general effect handling.

Critique of Completable Futures

- They are better than callbacks
- But they are still hard to compose.
- Notation is more heavyweight than direct simple futures, even when using `for`.
- Choice of async *vs* sync is viral. Hard to mix the two styles.

Part 3: Back to Direct Style

Trend: Runtimes get increasingly support for fibers or continuations.

Examples

- Goroutines,
- Project Loom in Java,
- Kotlin coroutines,
- OCaml or Haskell delimited continuations,
- Research languages such as Effekt, Koka
- Scala Native continuations

- This will deeply influence libraries and frameworks
- It makes it possible and attractive to go back to direct style.

Part 3: Back to Direct Style

Trend: Runtimes get increasingly support for fibers or continuations.

Examples

- Goroutines,
- Project Loom in Java,
- Kotlin coroutines,
- OCaml or Haskell delimited continuations,
- Research languages such as Effekt, Koka
- Scala Native continuations

- This will deeply influence libraries and frameworks
- It makes it possible and attractive to go back to direct style.

How will this influence Scala in the future?

1. There will be native foundations for direct-style reactive programming
   - Delimited continuations on Scala Native
   - Fibers on latest Java
   - Source or bytecode rewriting for older Java, JS
2. This will enable new techniques for designing and composing software
3. There will be a move away from monads as the primary way of code composition.

Direct-Style Operations

Here are some direct style operations that were often implemented with monads before.

- Aborting computations
- Error handling
- Suspending and resuming computations
- Asynchronous computing

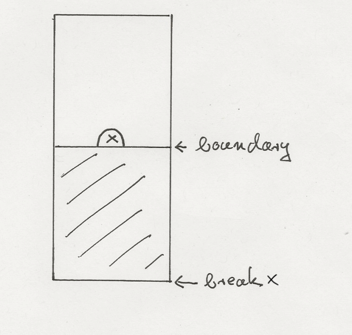
Warmup: Boundary/break

Problem: Want to break out of a loop, returning a value.

```
def firstIndex[T](xs: List[T], elem: T): Int =
  boundary:
    for (x, i) <- xs.zipWithIndex do
      if x == elem then break(i)
    -1
```

`boundary` establishes a boundary
`break` returns with a value from it.

Stack View



API

```scala
package scala.util

object boundary:
  final class Label[-T]

  def break[T](value: T)(using label: Label[T]): Nothing =
    throw Break(label, value)

  inline def apply[T](inline body: Label[T] ?=> T): T = ...
end boundary
```

To `break`, you need a `label` that represents the boundary.

In a sense, `label` is a *capability* that *enables* to `break`.

(This is a common pattern)

Implementation

The implementation of `break` produces efficient code.

- If `break` appears in the same stackframe as its `boundary`, use a jump.
- Otherwise use a fast exception that does not capture a stack trace.

A stack trace is not needed since we know the exception will be handled (*)

(*) To be 100% sure, this needs capture checking, a research topic we are working on.

Implementation

The implementation of `break` produces efficient code.

- If `break` appears in the same stackframe as its `boundary`, use a jump.
- Otherwise use a fast exception that does not capture a stack trace.

A stack trace is not needed since we know the exception will be handled (*)

(*) To be 100% sure, this needs capture checking, a research topic we are working on.

Stage 2: Error handling

boundary/break can be used as the basis for flexible error handling. For instance:

```
def firstColumn[T](xss: List[List[T]]): Option[List[T]] =
  optional:
    xss.map(_.headOption.?)
```

Optionally, returns the first column of the matrix xss.

Returns None if there is an empty row.

Error handling implementation

`optional` and `?` on options can be implemented quite easily on top of `boundary/break`:

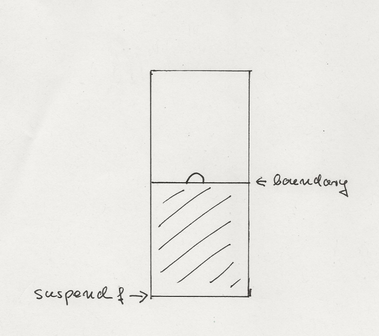
```
object optional:
  inline def apply[T](inline body: Label[None.type] ?=> T)
    : Option[T] = boundary(Some(body))

  extension [T](r: Option[T])
    inline def ? (using label: Label[None.type]): T = r match
      case Some(x) => x
      case None => break(None)
```

Analogous implementations are possible for other result types such as `Either` or a Rust-like `Result`.

My ideal way of error handling would be based on `Result` + `?`.

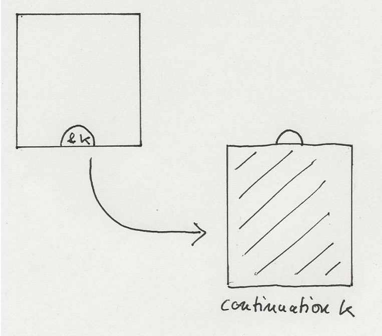
Stage 3: Suspensions

Question: What if we could store the stack segment between a `break` and its `boundary` and re-use it at some later time?



Suspensions

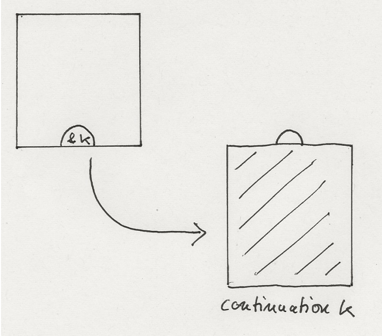
Question: What if we could store the stack segment between a `break` and its `boundary` and re-use it at some later time?



This is the idea of *delimited continuations*.

Suspensions

Question: What if we could store the stack segment between a `break` and its `boundary` and re-use it at some later time?



This is the idea of *delimited continuations*.

Suspension API

```scala
class Suspension[-T, +R]:
  def resume(arg: T): R = ???

def suspend[T, R](body: Suspension[T, R] => R)(using Label[R]): T
```

Suspensions are quite powerful.

They can express at the same time *algebraic effects* and *monads*.

Generators

Python-style generators are a simple example of algebraic effects.

```
def example = generate:
  produce(”We’ll give you all the numbers divisible by 3 or 2”)
  for i <- 1 to 1000 do
    if i % 3 == 0 then
      produce(s”$i is divisible by 3”)
    else if i % 2 == 0 then
      produce(s”$i is even”)
```

Here, `Generator` is essentially a simplified `Iterator`

```
trait Generator[T]:
  def nextOption: Option[T]
```

Algebraic Effects

Task: Build a `generate` implementation of `Generator`, so that one can compute the `leafs` of a `Tree` like this:

```
enum Tree[T]:
  case Leaf(x: T)
  case Inner(xs: List[Tree[T]])

def leafs[T](t: Tree[T]): Generator[T] =
  generate:                                         // effect scope
    def recur(t: Tree[T]): Unit = t match
      case Tree.Leaf(x)   => produce(x)             // effect
      case Tree.Inner(xs) => xs.foreach(recur)
    recur(t)
```

Generator Implementation

```
trait Produce[-T]:
  def produce(x: T): Unit

def generate[T](body: Produce[T] ?=> Unit) = new Generator[T]:
  def nextOption: Option[T] = step()

  var step: () => Option[T] =
```

The Step Function

```
trait Produce[-T]:                        // effect type
  def produce(x: T): Unit

def generate[T](body: Produce[T] ?=> Unit) = new Generator[T]:
  def nextOption: Option[T] = step()

  var step: () => Option[T] = () =>
    boundary:
      given Produce[T] with               // handler
        def produce(x: T): Unit =
          suspend[Unit, Option[T]]: k =>
            step = () => k.resume(())
            Some(x)
      body
      None
```

Summary: Algebraic Effects

Effects are *methods* of effect traits

Handlers are *implementations* of effect traits

- They are passed as *implicit parameters*.
- They can *abort* part of a computation via break
- They can also *suspend* part of a computation as a continuation and resume it later.

Implementing Suspensions

There are several possibilities:

- Directly in the runtime, as shown in the designs
- On top of fibers (requires some compromises)
- By bytecode rewriting (e.g. Quasar, `javactrl`)
- By source rewriting

Direct-Style Futures

With `suspend`(*), we can implement lightweight and universal `await` construct that can be called anywhere.

This can express simple, direct-style futures.

```
val sum = Future:
  val f1 = Future(c1.read)
  val f2 = Future(c2.read)
  f1.await + f2.await
```

Structured concurrency: Local futures `f1` and `f2` complete before `sum` completes. This might mean that one of them is cancelled if the other returns with a failure.

(*) Loom-like fibers would work as well.

An Implementation

`lampepfl/gears` is an early stage prototype of a modern, low-level concurrency library in direct style.

Main elements

- **Futures:** the primary *active* elements. They can be awaited cheaply. No map/flatMap constructions necessary.
- **Channels:** the primary *passive* elements.
- **Async Sources** Futures and Channels both implement a new fundamental abstraction: an *asynchronous source*.
- **Async Contexts** An async context is a *capability* that allows a computation to suspend while waiting for the result of an async source.

Link: `github.com/lampepfl/gears`

Conclusion

We have explored futures, from two different angles:

1. As a simple way to support lenient evaluation
2. As a scalable way to simplify asynchronous computations

The first two implementations of futures each did one of these points well and the other less well.

Using new new control abstractions we can combine simplicity and scalability.