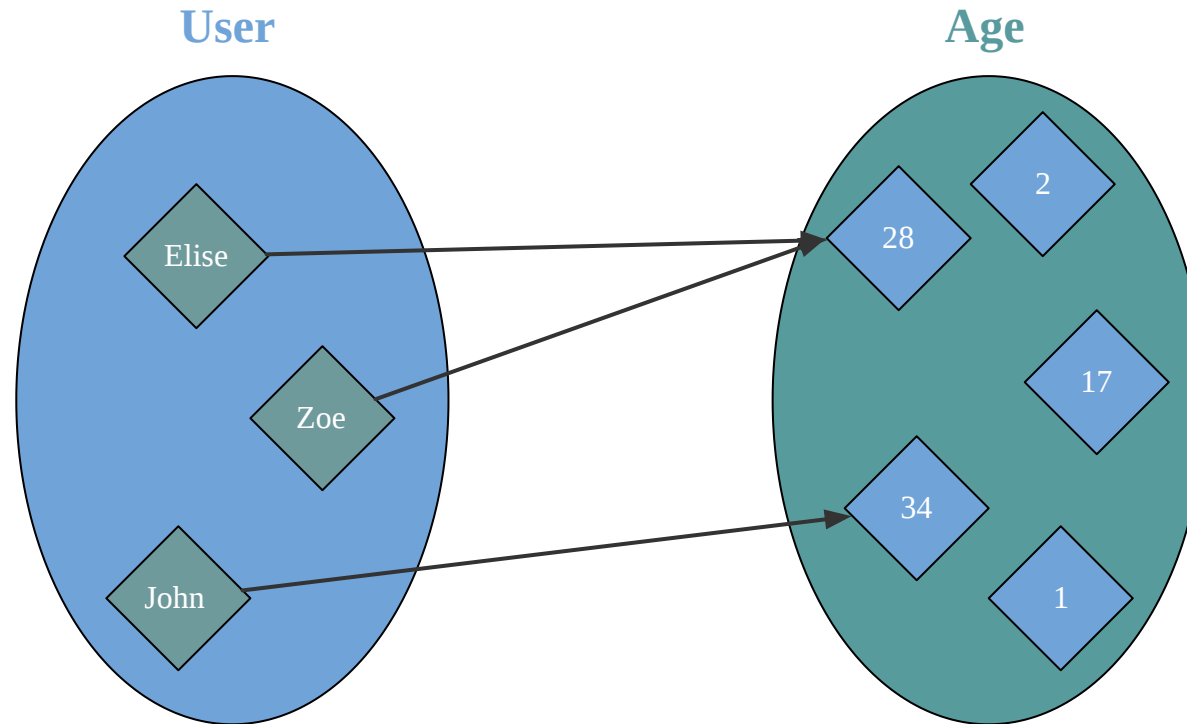


FOUNDATION

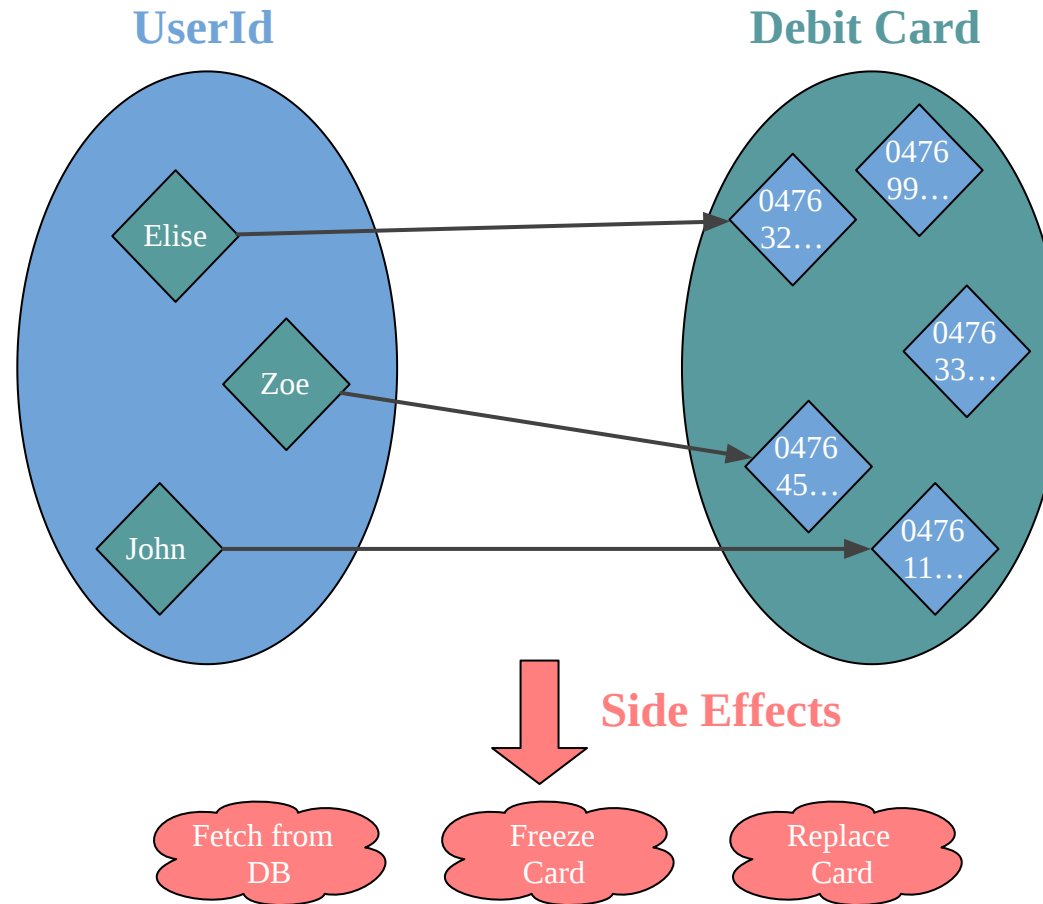


Side Effect

Pure function



Functions with side effects are not pure



Functional programming is useless *

[Simon Peyton Jones](#) co-author of haskell



A pure function cannot do anything
it can only produce a value



Create a value that describes actions



Create a value that describes actions

Interpret this value in Main



1. Encode description of actions

```
trait Description[A]
```

2. Define an interpreter of Description

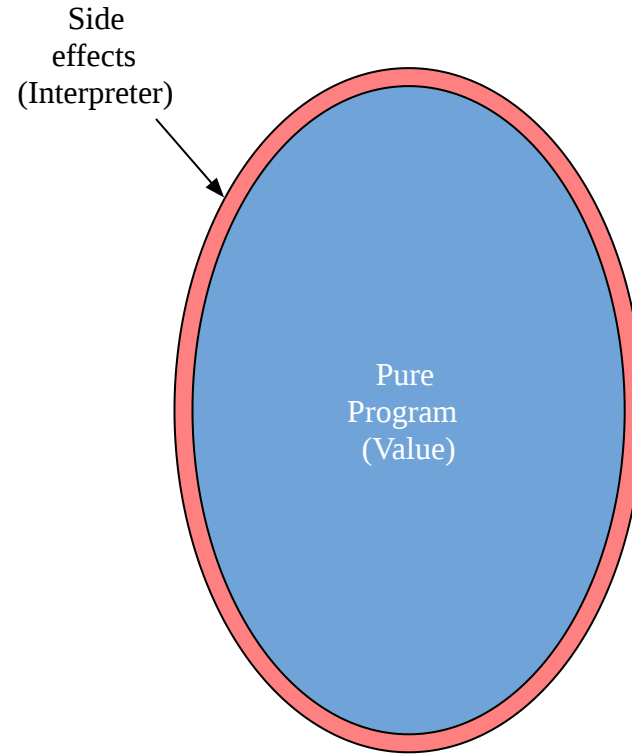
```
def unsafeRun[A](fa: Description[A]): A = ???
```

3. Combine everything in Main

```
object Main extends App {  
  val description: Description[Unit] = ???  
  unsafeRun(description)  
}
```



Run side effects at the edges



Examples of description / evaluation



Cooking

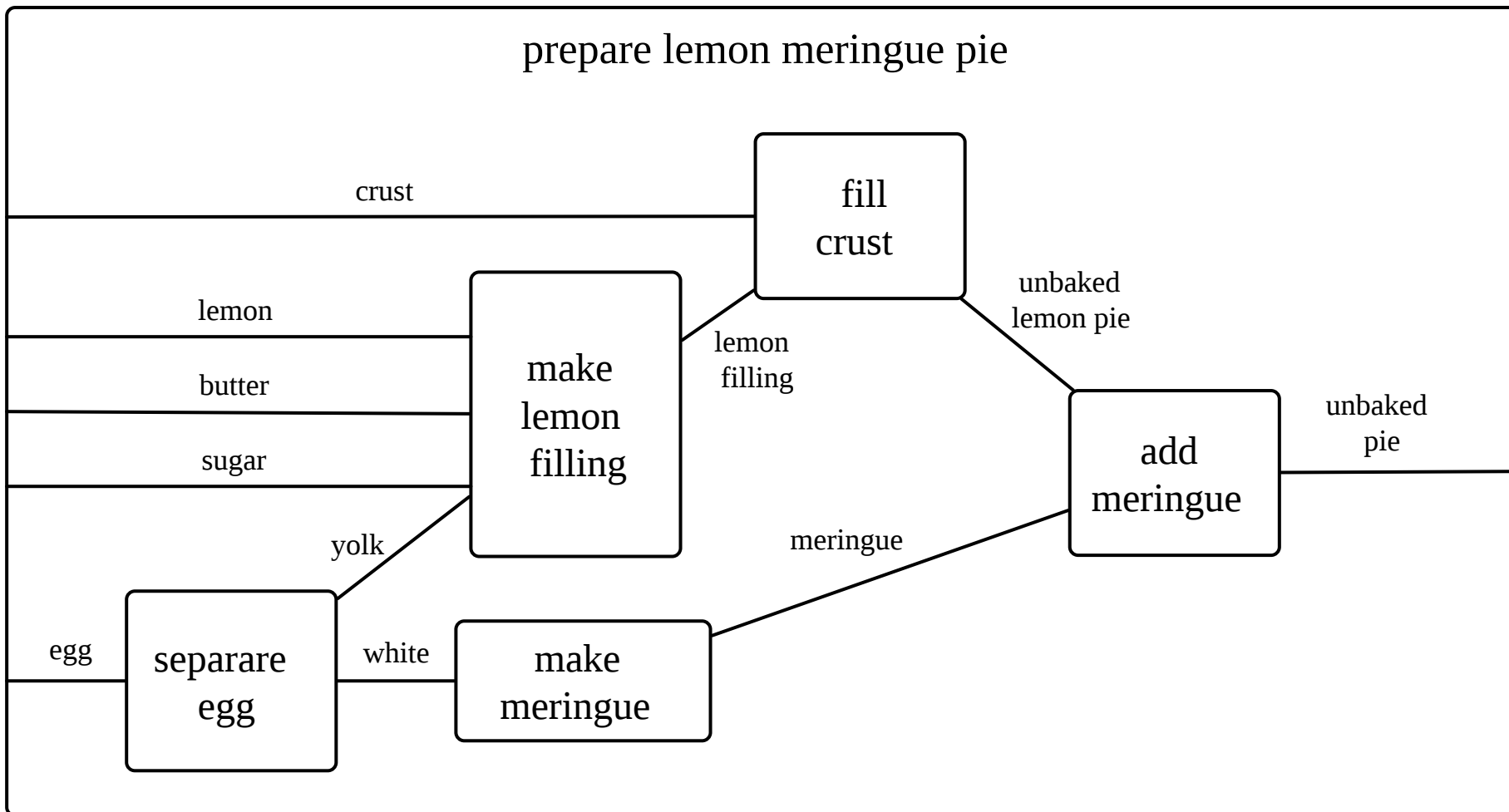
1. Secret pasta recipe (Description)

1. Boil 200 ml of water
2. Add 250 g of dry pasta
3. Wait 11 minutes
4. Drain the pasta

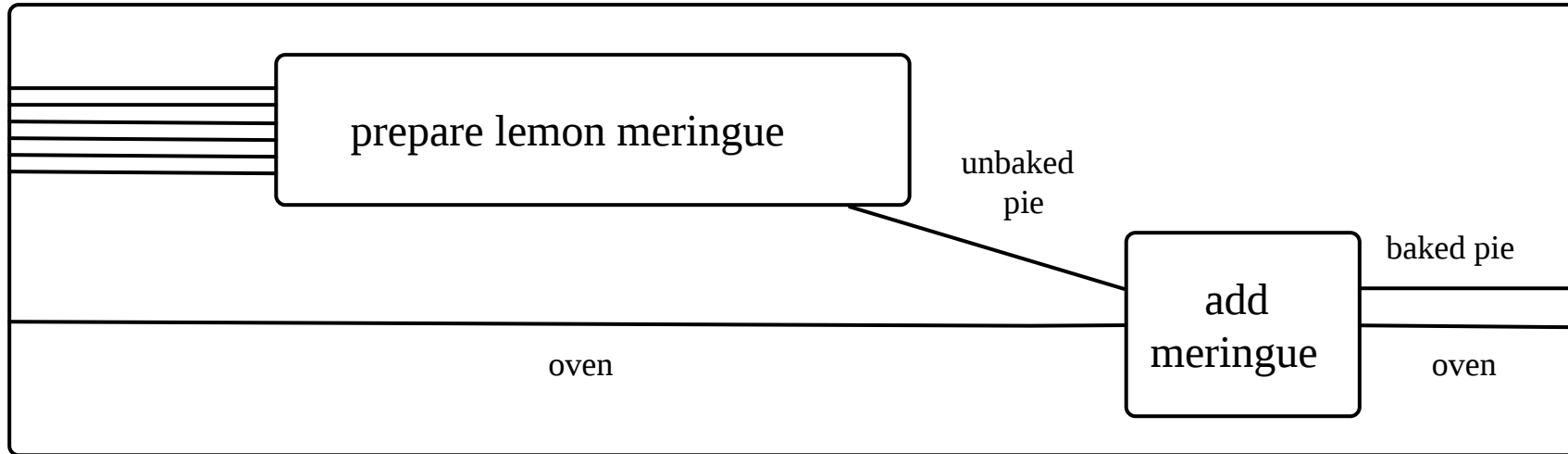
2. Cook (Unsafe evaluation)

Take the recipe and do it at home





String diagrams compose



Mathematical formula

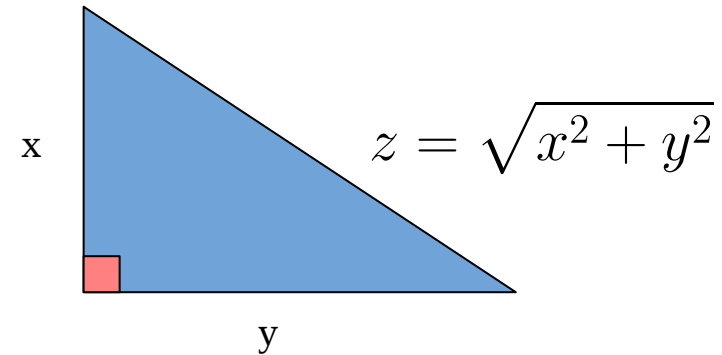
```
scala> val x = 2
x: Int = 2

scala> val y = 3
y: Int = 3

scala> val x2 = Math.pow(x, 2)
x2: Double = 4.0

scala> val y2 = Math.pow(y, 2)
y2: Double = 9.0

scala> val z = Math.sqrt(x2 + y2)
z: Double = 3.605551275463989
```



Mathematical formula

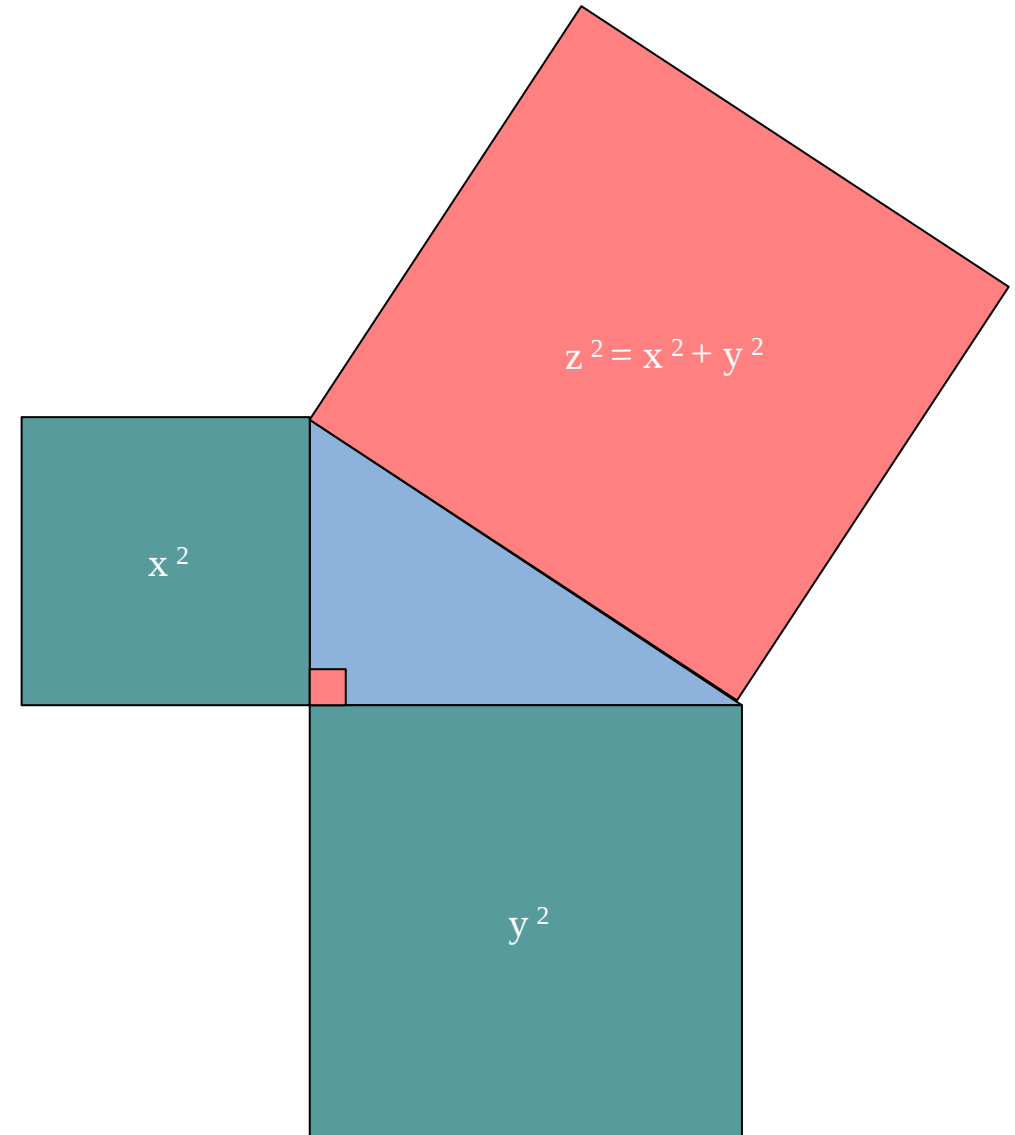
```
scala> val x2 = Math.pow(x, 2)
x2: Double = 4.0

scala> val y2 = Math.pow(y, 2)
y2: Double = 9.0

scala> val z = Math.sqrt(x2 + y2)
z: Double = 3.605551275463989

scala> Math.pow(z, 2)
res0: Double = 12.999999999999998

scala> x2 + y2
res1: Double = 13.0
```



How to encode description?

```
trait Description[A]  
  
def unsafeRun[A](fa: Description[A]): A = ???
```



Thunk

```
type Thunk[A] = () => A // Unit => A

def writeLine(message: String): Thunk[Unit] =
  () => println(message)

val today: Thunk[LocalDate] =
  () => LocalDate.now()

def fetch(url: String): Thunk[Iterator[String]] =
  () => scala.io.Source.fromURL(url)("ISO-8859-1").getLines
```



Thunk

```
type Thunk[A] = () => A // Unit => A

def writeLine(message: String): Thunk[Unit] =
  () => println(message)

val today: Thunk[LocalDate] =
  () => LocalDate.now()

def fetch(url: String): Thunk[Iterator[String]] =
  () => scala.io.Source.fromURL(url)("ISO-8859-1").getLines
```

```
def unsafeRun[A](fa: Thunk[A]): A = fa()
```



Thunk

```
type Thunk[A] = () => A // Unit => A

def writeLine(message: String): Thunk[Unit] =
  () => println(message)

val today: Thunk[LocalDate] =
  () => LocalDate.now()

def fetch(url: String): Thunk[Iterator[String]] =
  () => scala.io.Source.fromURL(url)("ISO-8859-1").getLines
```

```
def unsafeRun[A](fa: Thunk[A]): A = fa()
```

```
scala> val google = fetch("http://google.com")
google: Thunk[Iterator[String]] = $$Lambda$4459/0x0000000101817c40@3a1fde4

scala> unsafeRun(google).take(1).toList
res2: List[String] = List(<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en"><head><me
```



IO

```
trait IO[A] {  
  def unsafeRun(): A // unique abstract method  
  
  def map[B](f: A => B): IO[B] = ???  
  def flatMap[B](f: A => IO[B]): IO[B] = ???  
  def retry: IO[A] = ???  
}
```



IO

```
trait IO[A] {  
  def unsafeRun(): A // unique abstract method  
  
  def map[B](f: A => B): IO[B] = ???  
  def flatMap[B](f: A => IO[B]): IO[B] = ???  
  def retry: IO[A] = ???  
}
```

```
def writeLine(message: String): IO[Unit] =  
  new IO[Unit] {  
    def unsafeRun(): Unit = println(message)  
  }
```

```
scala> val helloWorld = writeLine("Hello World")  
helloWorld: IO[Unit] = $anon$1@7ccf7ae6
```

```
scala> helloWorld.unsafeRun()  
Hello World
```



Implement our own IO



Smart constructors

```
object IO {  
  def succeed[A](constant: A): IO[A] = ???  
  def effect[A](block: => A): IO[A] = ???  
  def fail[A](error: Throwable): IO[A] = ???  
}  
  
trait IO[A] {  
  def unsafeRun(): A  
}
```



IO Exercises

`exercises.sideeffect.IOExercises.scala`



IO Summary

- An IO is a thunk of potentially impure code
- Composing IO is referentially transparent, nothing get executed
- It is easier to test IO if they are defined in a interface



I/O Execution



IO execution

```
case class UserId (value: String)
case class OrderId(value: String)

case class User(userId: UserId, name: String, orderIds: List[OrderId])
```

```
def getUser(userId: UserId): IO[User] =
  IO.effect{
    val response = httpClient.get(s"http://foo.com/user/${userId.value}")
    if(response.status == 200) parseJson[User](response.body)
    else throw new Exception(s"Invalid status ${response.status}")
  }

def deleteOrder(orderId: OrderId): IO[Unit] =
  IO.effect{
    val response = httpClient.delete(s"http://foo.com/order/${orderId.value}")
    if(response.status == 200) () else throw new Exception(s"Invalid status ${response.status}")
  }
```



How is it executed?

```
def deleteAllUserOrders(userId: UserId): IO[Unit] =  
  for {  
    user <- getUser(userId)  
    _ <- traverse(user.orderIds)(deleteOrder)  
  } yield ()  
  
object Main extends App {  
  deleteAllUserOrders(UserId("1234")).unsafeRun()  
}
```

Discuss with your neighbour 3-4 min



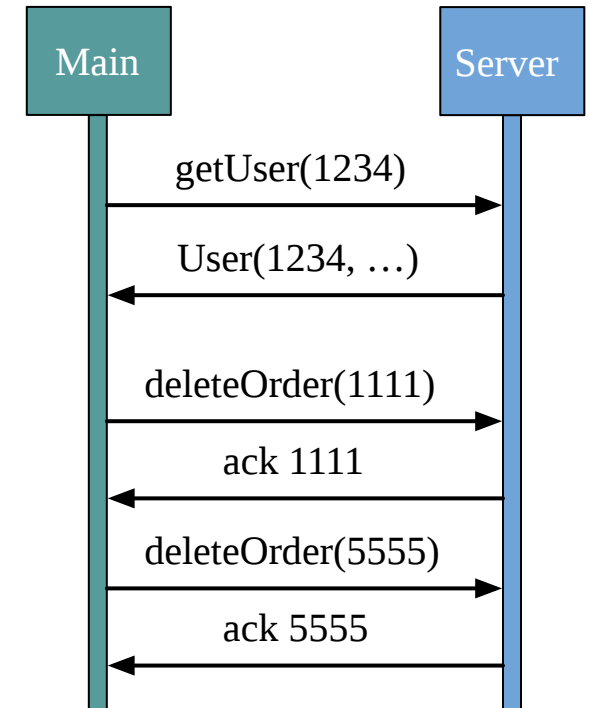
How is it executed?

```
def deleteAllUserOrders(userId: UserId): IO[Unit] =  
  for {  
    user <- getUser(userId)  
    // User("1234", "Rob", List("1111", "5555"))  
    _ <- deleteOrder(user.orderIds(0)) // 1111  
    _ <- deleteOrder(user.orderIds(1)) // 5555  
  } yield ()  
  
object Main extends App {  
  deleteAllUserOrders(UserId("1234")).unsafeRun()  
}
```



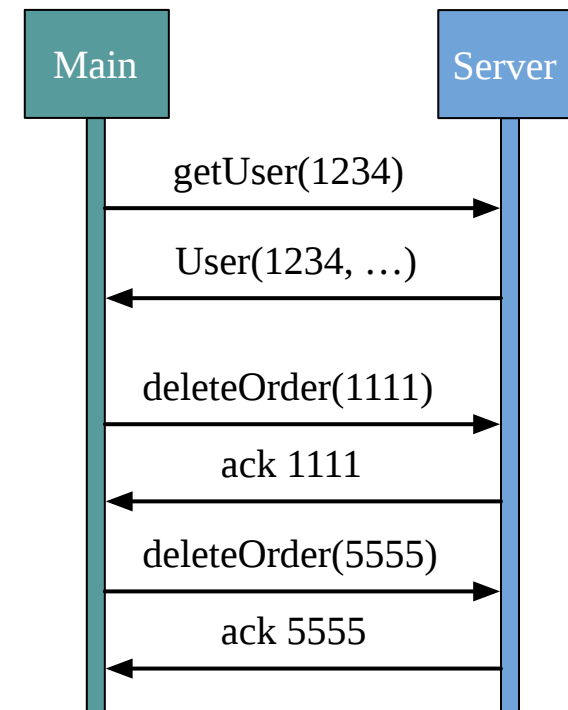
IO execution is sequential

```
def deleteAllUserOrders(userId: UserId): IO[Unit] =  
  for {  
    user <- getUser(userId)  
    // User("1234", "Rob", List("1111", "5555"))  
    _ <- deleteOrder(user.orderIds(0)) // 1111  
    _ <- deleteOrder(user.orderIds(1)) // 5555  
  } yield ()  
  
object Main extends App {  
  deleteAllUserOrders(UserId("1234")).unsafeRun()  
}
```



Which IO could be evaluated concurrently?

```
def deleteAllUserOrders(userId: UserId): IO[Unit] =  
  for {  
    user <- getUser(userId)  
    // User("1234", "Rob", List("1111", "5555"))  
    _ <- deleteOrder(user.orderIds(0)) // 1111  
    _ <- deleteOrder(user.orderIds(1)) // 5555  
  } yield ()  
  
object Main extends App {  
  deleteAllUserOrders(UserId("1234")).unsafeRun()  
}
```

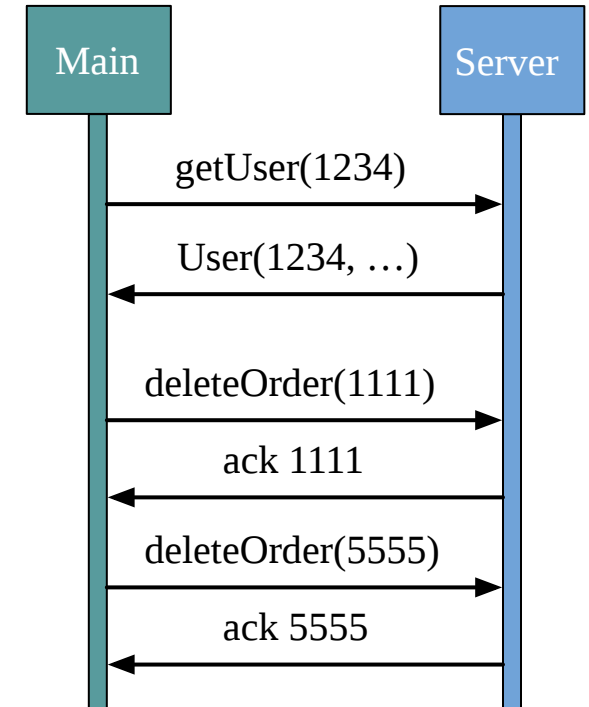


Discuss with your neighbour 3-4 min



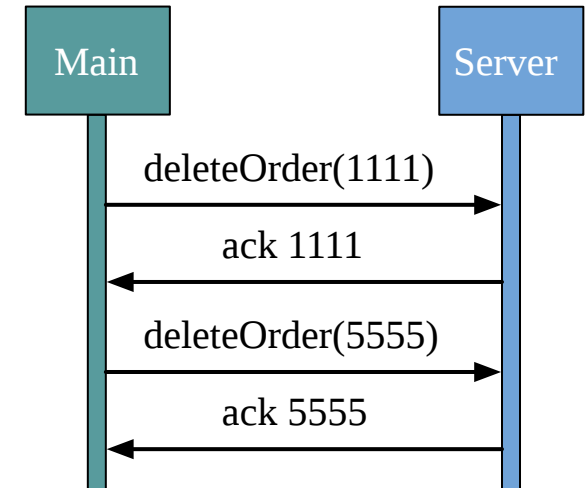
For comprehension cannot be done concurrently

```
def deleteAllUserOrders(userId: UserId): IO[Unit] =  
  for {  
    user <- getUser(userId)  
    // User("1234", "Rob", List("1111", "5555"))  
    _ <- deleteOrder(user.orderIds(0)) // 1111  
    _ <- deleteOrder(user.orderIds(1)) // 5555  
  } yield ()
```



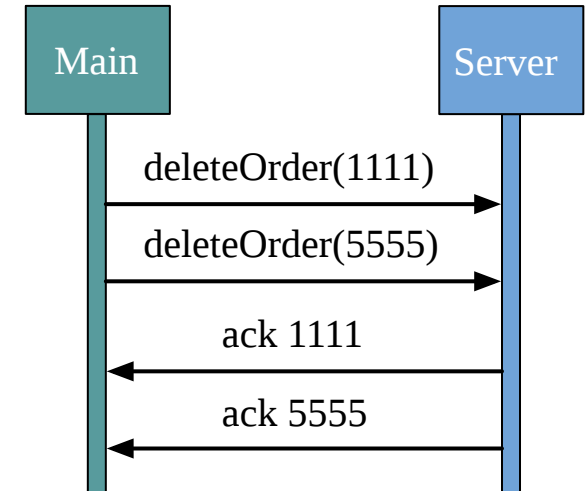
For comprehension cannot be done concurrently

```
def delete2Orders(orderId1: OrderId, orderId2: OrderId): IO[Unit] =  
  for {  
    ackOrder1 <- deleteOrder(orderId1)  
    ackOrder2 <- deleteOrder(orderId2)  
  } yield ()
```



Concurrent execution

```
def concurrentExec(io1: IO[Unit], io2: IO[Unit]): IO[Unit] = ???  
  
def delete2Orders(orderId1: OrderId, orderId2: OrderId): IO[Unit] =  
  concurrentExec(deleteOrder(orderId1), deleteOrder(orderId2))
```



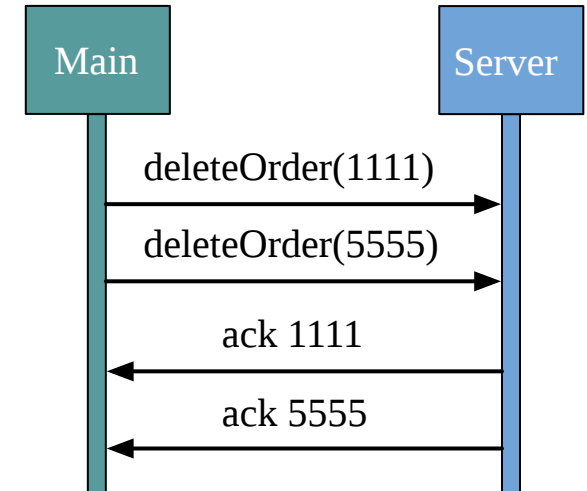
concurrentExec is loosely defined

```
def concurrentExec(io1: IO[Unit], io2: IO[Unit]): IO[Unit] =  
  io1  
  
def concurrentExec(io1: IO[Unit], io2: IO[Unit]): IO[Unit] =  
  io2  
  
def concurrentExec(io1: IO[Unit], io2: IO[Unit]): IO[Unit] =  
  for {  
    _ <- io1  
    _ <- io2  
  } yield ()  
  
def concurrentExec(io1: IO[Unit], io2: IO[Unit]): IO[Unit] =  
  IO.succeed(())
```



Parametricity

```
def concurrentMap2[A, B, C](fa: IO[A], fb: IO[B])  
    (f: (A, B) => C): IO[C] = ???  
  
def delete2Orders(orderId1: OrderId, orderId2: OrderId): IO[Unit] =  
    concurrentMap2(  
        deleteOrder(orderId1),  
        deleteOrder(orderId2)  
    )((_,_) => ())
```



How concurrency is done with Future?



Future

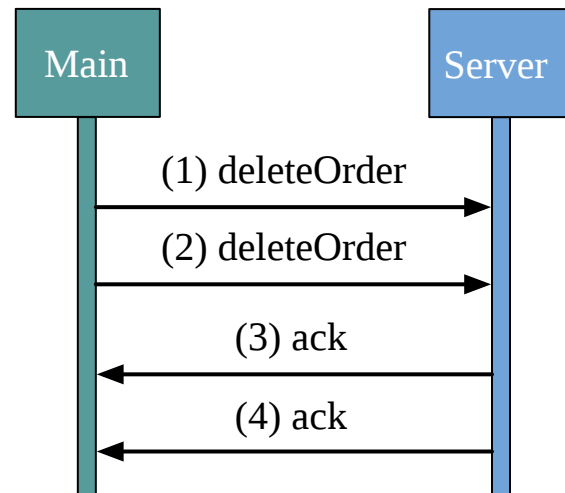
```
import scala.concurrent.{ExecutionContext, Future}

def deleteOrder(orderId: OrderId)
  (implicit ec: ExecutionContext): Future[Unit] =
  Future { ??? }

def delete2Orders(
  orderId1: OrderId,
  orderId2: OrderId
)(implicit ec: ExecutionContext): Future[Unit] = {

  val delete1: Future[Unit] = deleteOrder(orderId1) // (1) side effect
  val delete2: Future[Unit] = deleteOrder(orderId2) // (2) side effect

  for {
    _ /* (3) */ <- delete1
    _ /* (4) */ <- delete2
  } yield ()
}
```



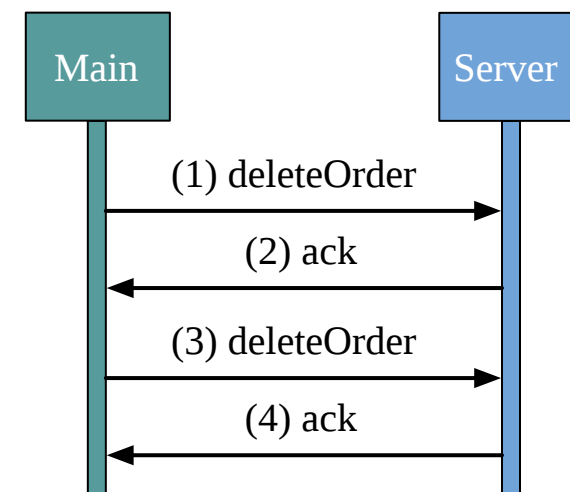
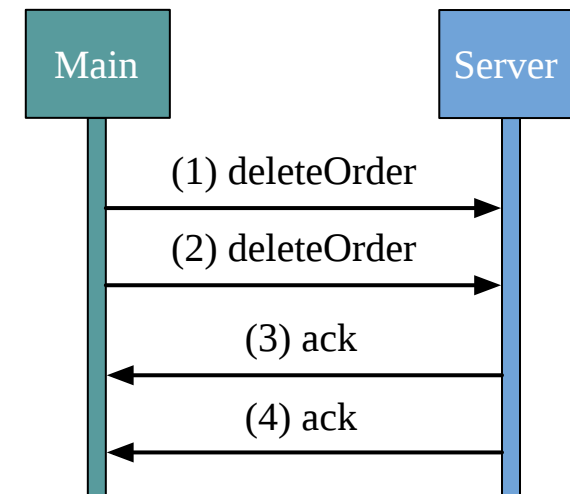
Creating a Future is a side effect

```
def deleteOrdersConcurrent(orderId1: OrderId, orderId2: OrderId)
  (implicit ec: ExecutionContext): Future[Unit] = {

  val delete1 = deleteOrder(orderId1) // (1)
  val delete2 = deleteOrder(orderId2) // (2)

  for {
    _ /* (3) */ <- delete1
    _ /* (4) */ <- delete2
  } yield ()
}
```

```
def deleteOrdersSequential(orderId1: OrderId, orderId2: OrderId)
  (implicit ec: ExecutionContext): Future[Unit] =
  for {
    _ /* (2) */ <- deleteOrder(orderId1) // (1)
    _ /* (4) */ <- deleteOrder(orderId2) // (3)
  } yield ()
```



FUTURE



1. CREATE YOUR FUTURES



2. WIRE THEM TOGETHER



3. OOPS! SEEMS LIKE
WE FORGOT SMTH

IO



1. CREATE YOUR IOS



2. WIRE THEM TOGETHER



3. PROFIT!



Execution Context

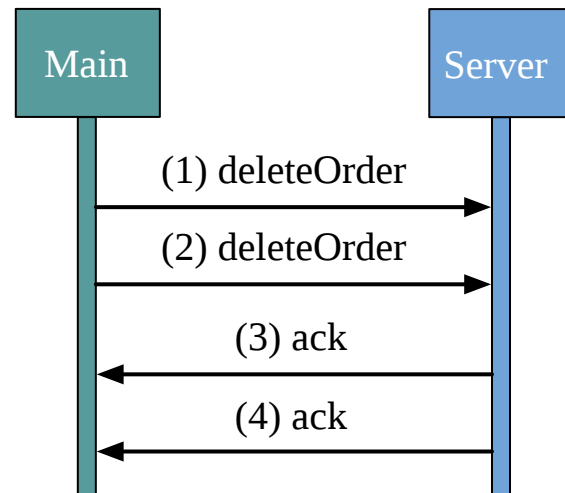
```
import scala.concurrent.{ExecutionContext, Future}

def deleteOrder(orderId: OrderId)(ec: ExecutionContext): Future[Unit] =
  Future { ??? }(ec)

def delete2Orders(
  orderId1: OrderId,
  orderId2: OrderId
)(ec: ExecutionContext): Future[Unit] = {

  val delete1 = deleteOrder(orderId1)(ec) // (1) side effect
  val delete2 = deleteOrder(orderId2)(ec) // (2) side effect

  delete1.flatMap(_ => // (3)
    delete2.map(_ => ()))(ec) // (4)
  )(ec)
}
```



Execution Context

```
import java.util.concurrent.Executors
import scala.concurrent.ExecutionContext

val factory = threadFactory("test")
val pool = Executors.newFixedThreadPool(2, factory)
val ec = ExecutionContext.fromExecutorService(pool)

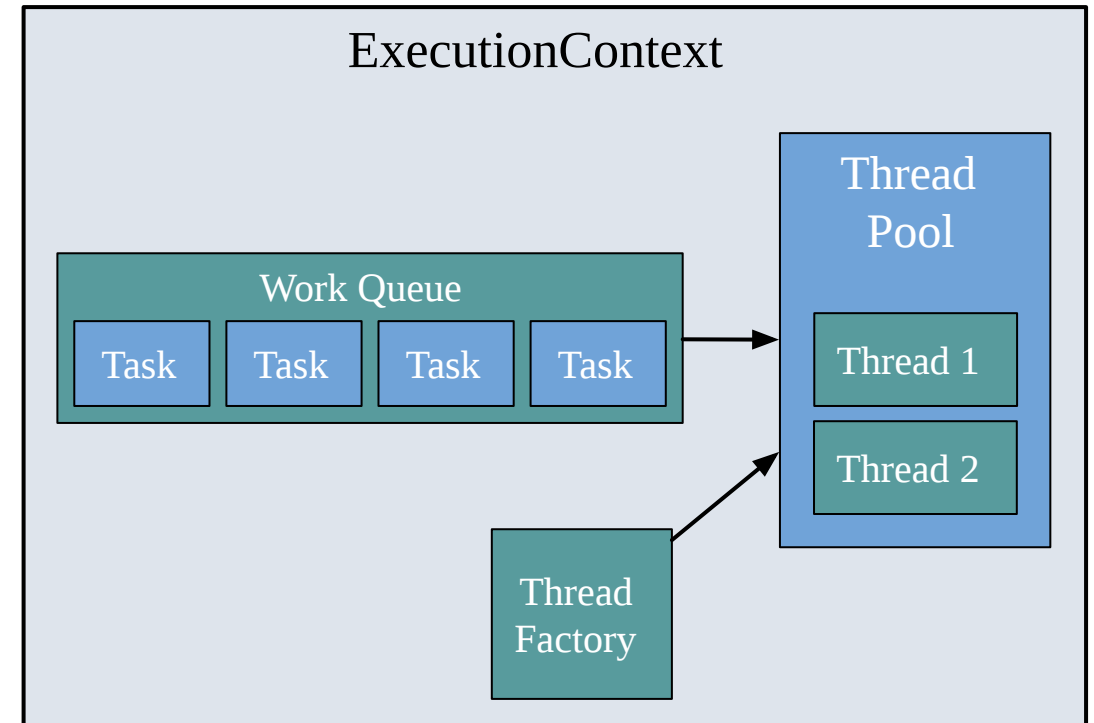
var x: Int = 0

val inc: Runnable = new Runnable {
  def run(): Unit = x += 1
}
```

```
scala> x
res4: Int = 0

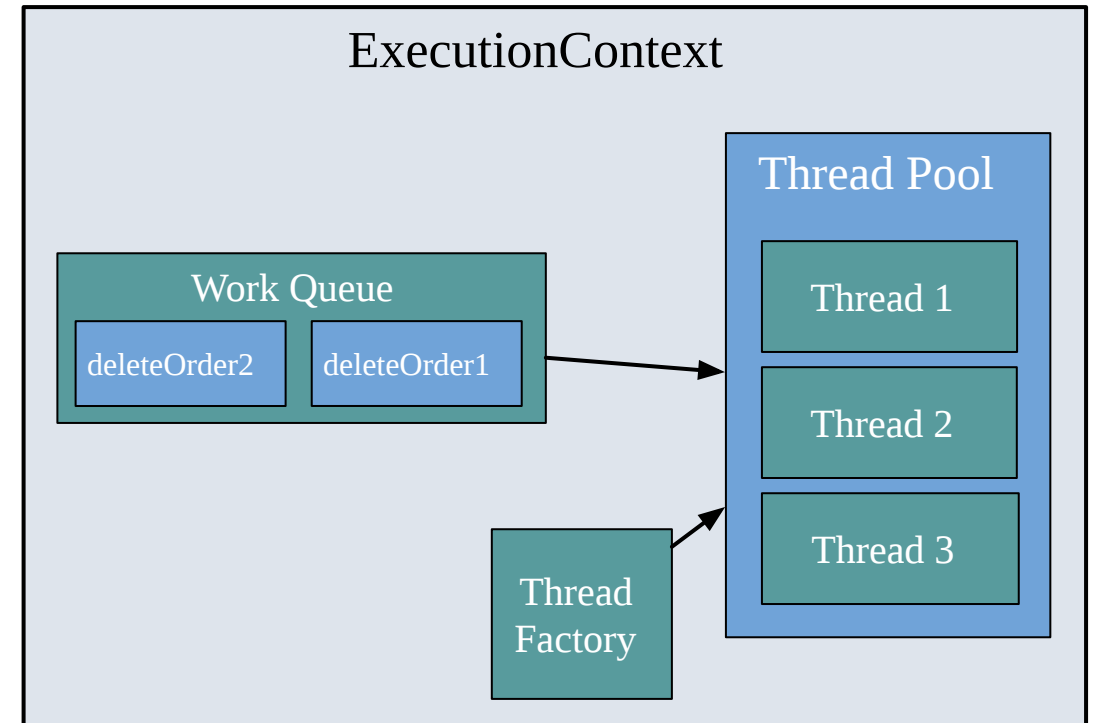
scala> (1 to 10).foreach(_ => ec.execute(inc))

scala> x
res6: Int = 10
```



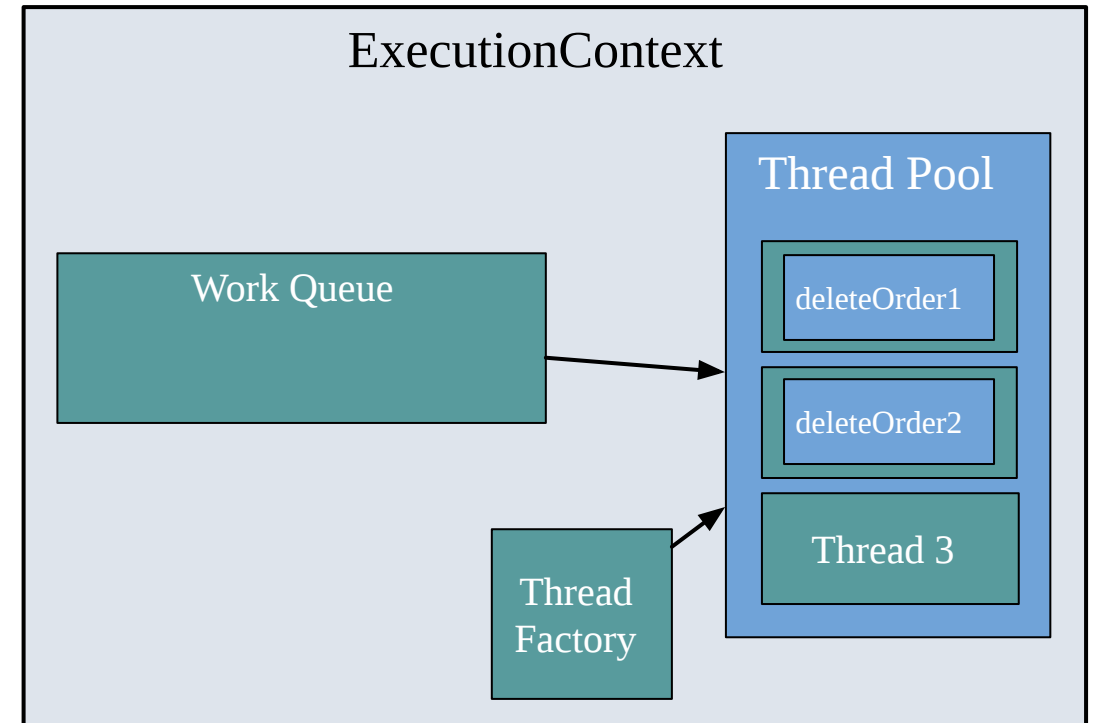
Execution Context

```
def delete2Orders(  
  orderId1: OrderId,  
  orderId2: OrderId  
) (ec: ExecutionContext): Future[Unit] = {  
  
  val delete1 = deleteOrder(orderId1)(ec) // (1)  
  val delete2 = deleteOrder(orderId2)(ec) // (2)  
  
  delete1.flatMap(_ =>           // (3)  
    delete2.map(_ => ())(ec) // (4)  
  )(ec)  
}
```



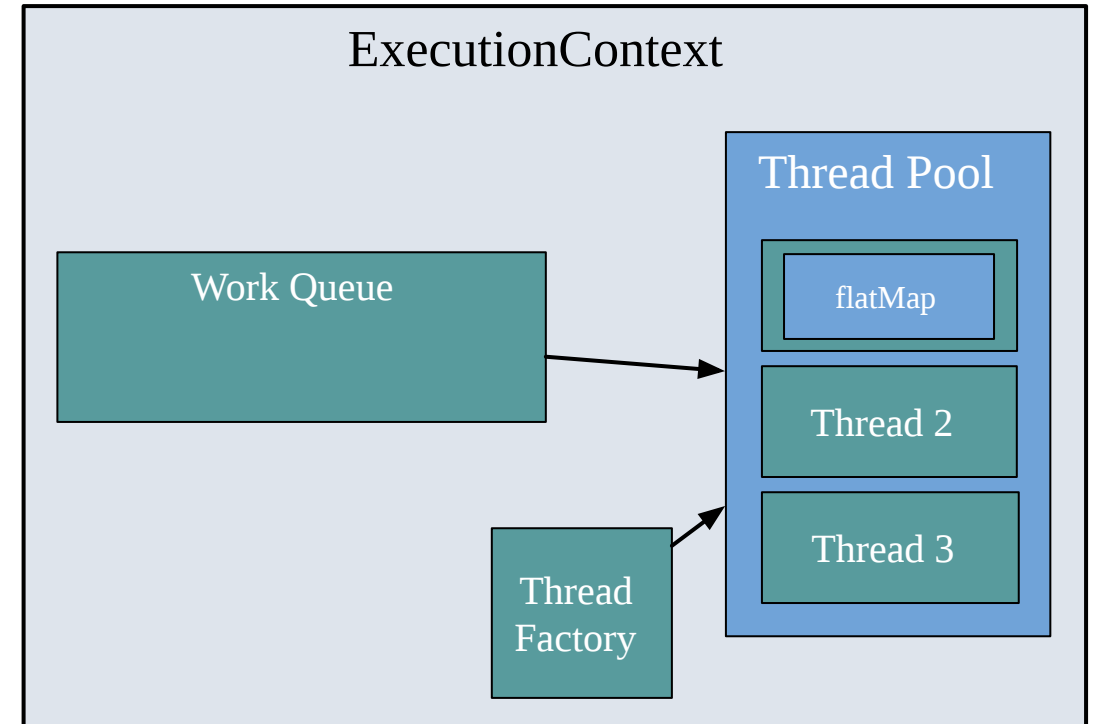
Execution Context

```
def delete2Orders(  
  orderId1: OrderId,  
  orderId2: OrderId  
) (ec: ExecutionContext): Future[Unit] = {  
  
  val delete1 = deleteOrder(orderId1)(ec) // (1)  
  val delete2 = deleteOrder(orderId2)(ec) // (2)  
  
  delete1.flatMap(_ =>           // (3)  
    delete2.map(_ => ())(ec) // (4)  
  )(ec)  
}
```



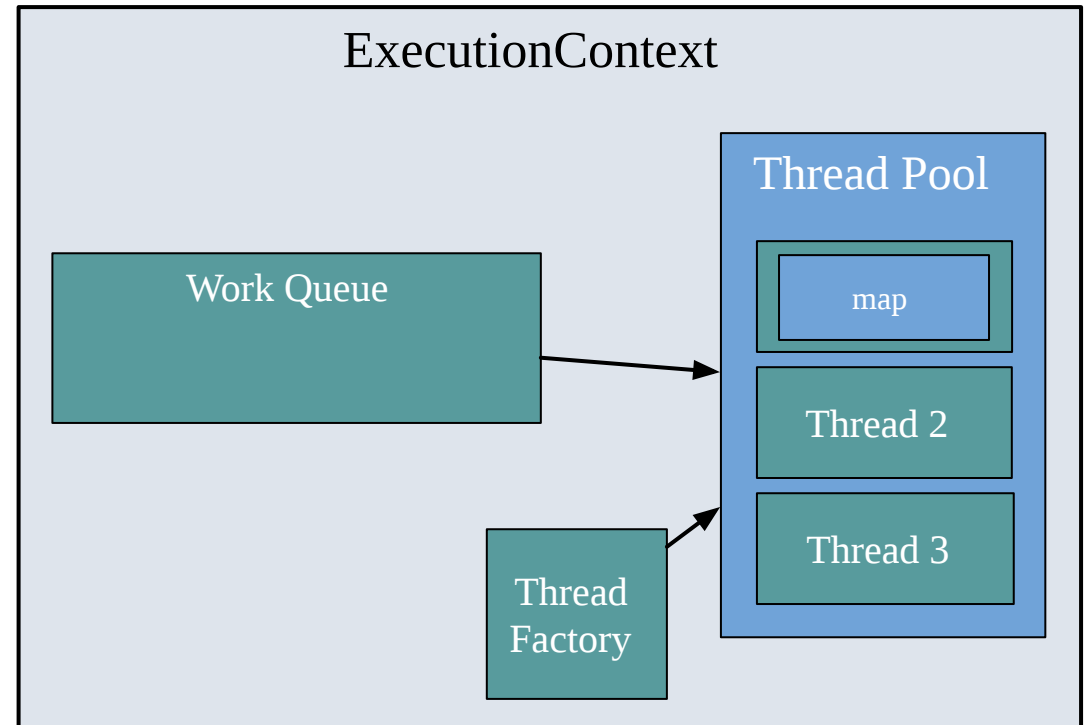
Execution Context

```
def delete2Orders(  
  orderId1: OrderId,  
  orderId2: OrderId  
) (ec: ExecutionContext): Future[Unit] = {  
  
  val delete1 = deleteOrder(orderId1)(ec) // (1)  
  val delete2 = deleteOrder(orderId2)(ec) // (2)  
  
  delete1.flatMap(_ =>           // (3)  
    delete2.map(_ => ()) (ec) // (4)  
  )(ec)  
}
```



Execution Context

```
def delete2Orders(  
  orderId1: OrderId,  
  orderId2: OrderId  
) (ec: ExecutionContext): Future[Unit] = {  
  
  val delete1 = deleteOrder(orderId1)(ec) // (1)  
  val delete2 = deleteOrder(orderId2)(ec) // (2)  
  
  delete1.flatMap(_ => // (3)  
    delete2.map(_ => ()) (ec) // (4)  
  ) (ec)  
}
```



How can we adapt Future behaviour to pure IO?



Concurrent IO

```
trait IO[+A] {  
  def start(ec: ExecutionContext): ???  
}
```

Discuss with your neighbour 3-4 min



Concurrent IO

```
trait IO[+A] {  
  def start(ec: ExecutionContext): IO[??]  
}
```



Concurrent IO

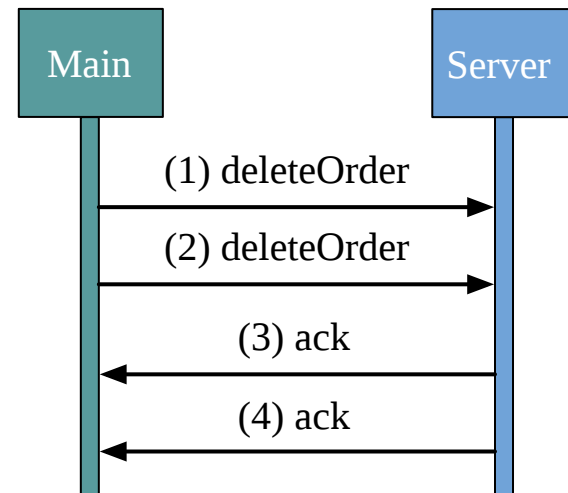
```
trait IO[+A] {  
  def start(ec: ExecutionContext): IO[IO[A]]  
}
```



Concurrent IO: concurrentMap2

```
trait IO[+A] {  
  def start(ec: ExecutionContext): IO[IO[A]]  
}
```

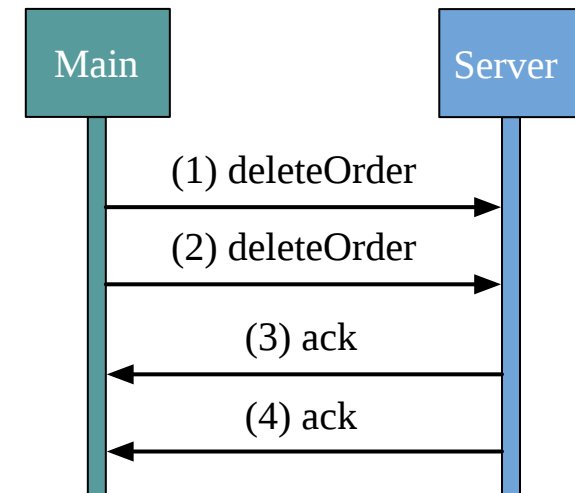
```
def concurrentMap2[A, B, C](  
  fa: IO[A],  
  fb: IO[B]  
) (f: (A, B) => C) (ec: ExecutionContext): IO[C] = ???
```



Concurrent IO: concurrentMap2

```
trait IO[+A] {  
  def start(ec: ExecutionContext): IO[IO[A]]  
}
```

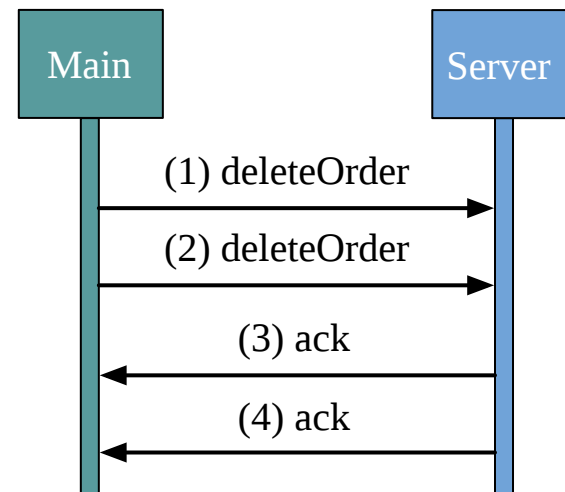
```
def concurrentMap2[A, B, C](  
  fa: IO[A],  
  fb: IO[B]  
) (f: (A, B) => C) (ec: ExecutionContext): IO[C] =  
  for {  
    awaitForA <- fa.start(ec) // (1)  
    awaitForB <- fb.start(ec) // (2)  
    a <- awaitForA // (3)  
    b <- awaitForB // (4)  
  } yield f(a, b)
```



Concurrent IO is referentially transparent

```
trait IO[+A] {  
  def start(ec: ExecutionContext): IO[IO[A]]  
}
```

```
def concurrentMap2[A, B, C](  
  fa: IO[A],  
  fb: IO[B]  
) (f: (A, B) => C) (ec: ExecutionContext): IO[C] = {  
  
  val asyncIOA = fa.start(ec)  
  val asyncIOB = fb.start(ec)  
  
  for {  
    awaitForA <- asyncIOA           // (1)  
    awaitForB <- asyncIOB           // (2)  
    a         <- awaitForA           // (3)  
    b         <- awaitForB           // (4)  
  } yield f(a, b)  
}
```



Concurrent IO with Async

```
type Callback[-A] = Either[Throwable, A] => Unit

sealed trait IO[+A]

object IO {
  case class Thunk[+A](f: () => A) extends IO[A]

  case class Async[+A](f: Callback[A] => Unit, ec: ExecutionContext) extends IO[A]
}
```



Concurrent IO with Async

```
type Callback[-A] = Either[Throwable, A] => Unit

sealed trait IO[+A]

object IO {
  case class Thunk[+A](f: () => A) extends IO[A]

  case class Async[+A](f: Callback[A] => Unit, ec: ExecutionContext) extends IO[A]
}
```

An IO is either a Thunk or a Async computation with a CallBack



Concurrent IO with Async

```
type Callback[-A] = Either[Throwable, A] => Unit

sealed trait IO[+A]

object IO {
  case class Thunk[+A](f: () => A) extends IO[A]

  case class Async[+A](f: Callback[A] => Unit, ec: ExecutionContext) extends IO[A]
}
```

An IO is either a Thunk or a Async computation with a CallBack

More details in [How do Fibers work](#) from Fabio Labella



IO Async Exercises

`exercises.sideeffect.IOAsyncExercises.scala`



Libraries do much more

- Stack safety and JVM optimisation
- Cancellation, e.g. race two IO and cancel the loser
- Safe resource shutdown, e.g. close file, shutdown server
- Efficient Timer, retry utilities
- Help to chose right thread pool for different type of work: blocking, compute, dispatcher



Resources and further study

- [Seven Sketches in Compositionality: An Invitation to Applied Category Theory](#).
- [Constraints Liberate, Liberties Constrain](#)
- [How do Fibers work](#)



Module 3: Error Handling

