FUTURES & PROMISES

in Scala 2.10

HEATHER MILLER
PHILIPP HALLER

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Typesafe

Agenda

FUTURES/PROMISES
EXECUTION CTXS
TRY

COMMON THEME:

COMMON THEME: *Pipelining*

Pipelining?

```
List(46, 34, 50, 21, 28)
```

```
map(x => x * 2)
```

```
List(92, 68, 100, 42, 56)
```

```
filter(x => x < 50)
```

```
List(42)
```

Pipelining?

```
val lst = List(46, 34, 50, 21, 28)

lst.map(x => x*2).filter(x => x<50)
```

filter(x => x < 50)

scala.concurrent.

FUTURE & PROMISE

FIRST, SOME *Motivation*

SEVERAL IMPORTANT LIBRARIES HAVE THEIR OWN FUTURE/PROMISE IMPLEMENTATION

SEVERAL IMPORTANT LIBRARIES HAVE THEIR OWN FUTURE/PROMISE IMPLEMENTATION

java.util.concurrent.	FUTURE
scala.actors.	FUTURE
com.twitter.util.	FUTURE
akka.dispatch.	FUTURE
scalaz.concurrent.	PROMISE
net.liftweb.actor.	LAFUTURE

1 SEVERAL IMPORTANT LIBRARIES HAVE THEIR OWN FUTURE/PROMISE IMPLEMENTATION

java.util.concurrent. **FUTURE**

~~scala.actors.~~ **FUTURE**

com.twitter.util. **FUTURE**

~~akka.dispatch.~~ **FUTURE**

scalaz.concurrent. **PROMISE**

net.liftweb.actor. **LAFUTURE**

THIS MAKES IT CLEAR THAT...

THIS MAKES IT CLEAR THAT...

FUTURES ARE AN IMPORTANT, POWERFUL ABSTRACTION

THIS MAKES IT CLEAR THAT...

FUTURES ARE AN IMPORTANT, POWERFUL ABSTRACTION

THERE'S FRAGMENTATION IN THE SCALA ECOSYSTEM

no hope of interop!

Furthermore...

Furthermore...

2 **JAVA FUTURES NEITHER EFFICIENT NOR COMPOSABLE**

Furthermore...

② JAVA FUTURES NEITHER EFFICIENT NOR COMPOSABLE

Furthermore...

2 JAVA FUTURES NEITHER EFFICIENT NOR COMPOSABLE

```
val lst = List(46, 34, 50, 21, 28)

lst.map(x => x*2).filter(x => x<50)
```

Furthermore...

2 JAVA FUTURES NEITHER EFFICIENT NOR COMPOSABLE

COMPOSABILITY MEANS:

- "DRY"ER CODE.
- MORE POWERFUL CODE, BUILD/COMPOSE RICH FUNCTIONALITY FROM SMALLER PARTS

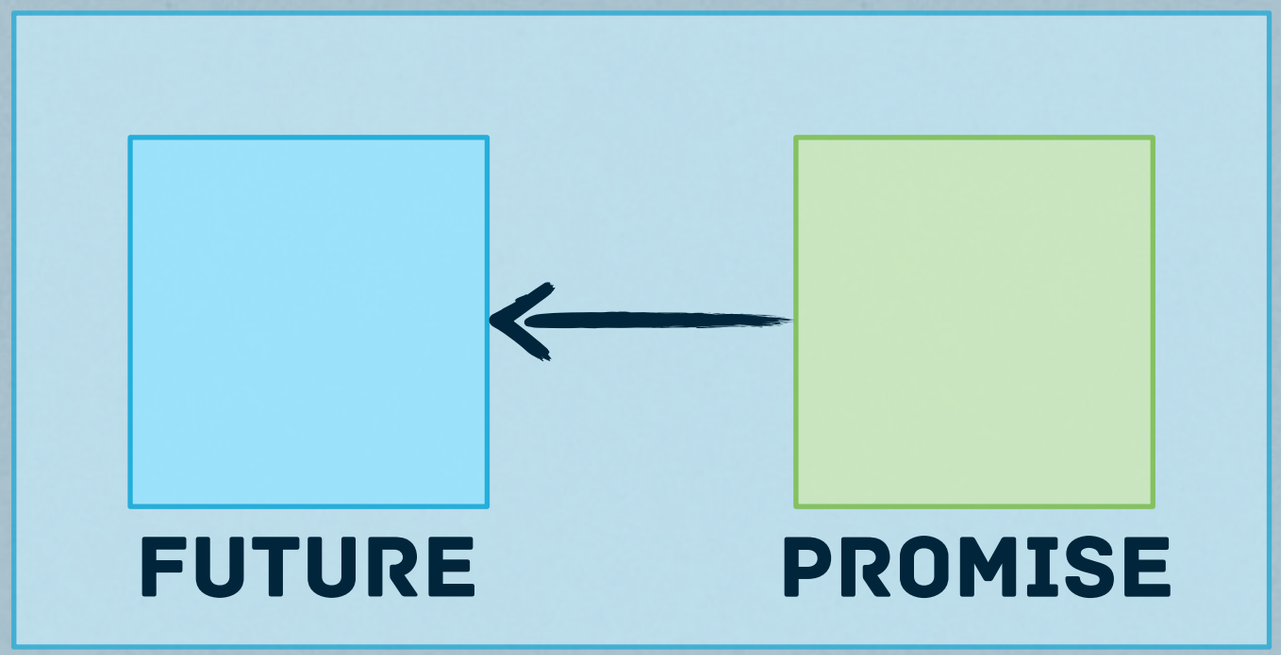
Furthermore...

2. JAVA FUTURES NEITHER EFFICIENT NOR COMPOSABLE

3. WE COULD MAKE FUTURES MORE POWERFUL, BY TAKING ADVANTAGE OF SCALA'S FEATURES

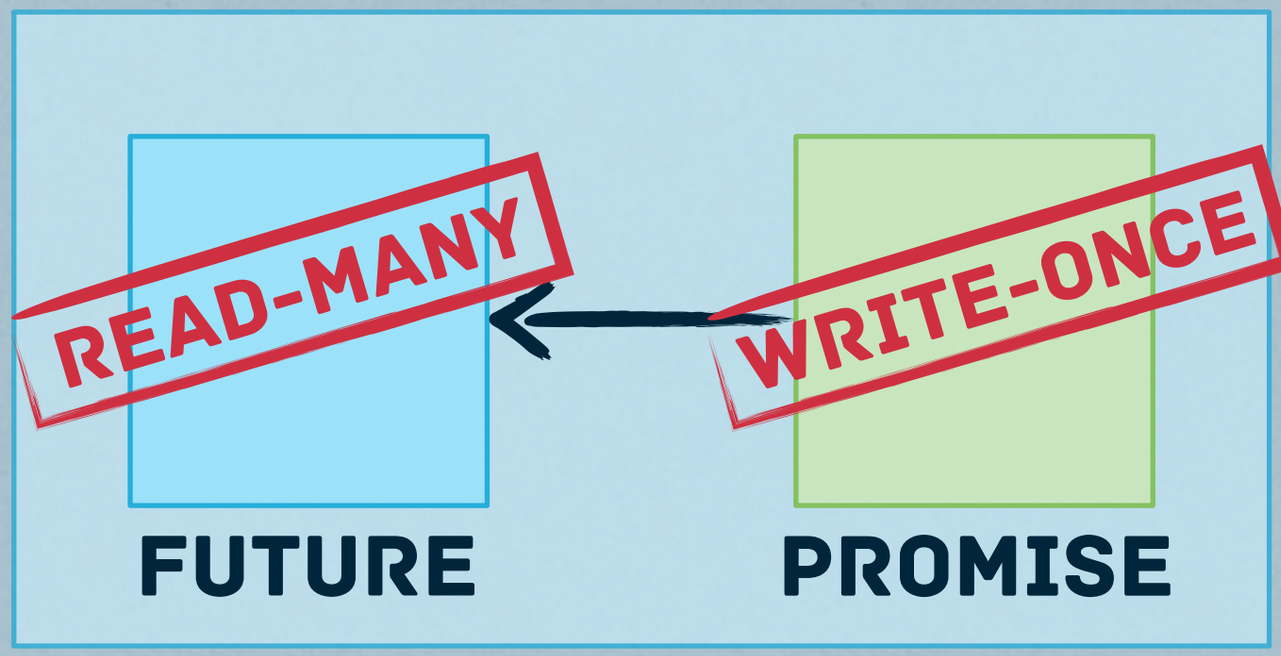
Futures & Promises

CAN BE THOUGHT OF AS A SINGLE CONCURRENCY ABSTRACTION



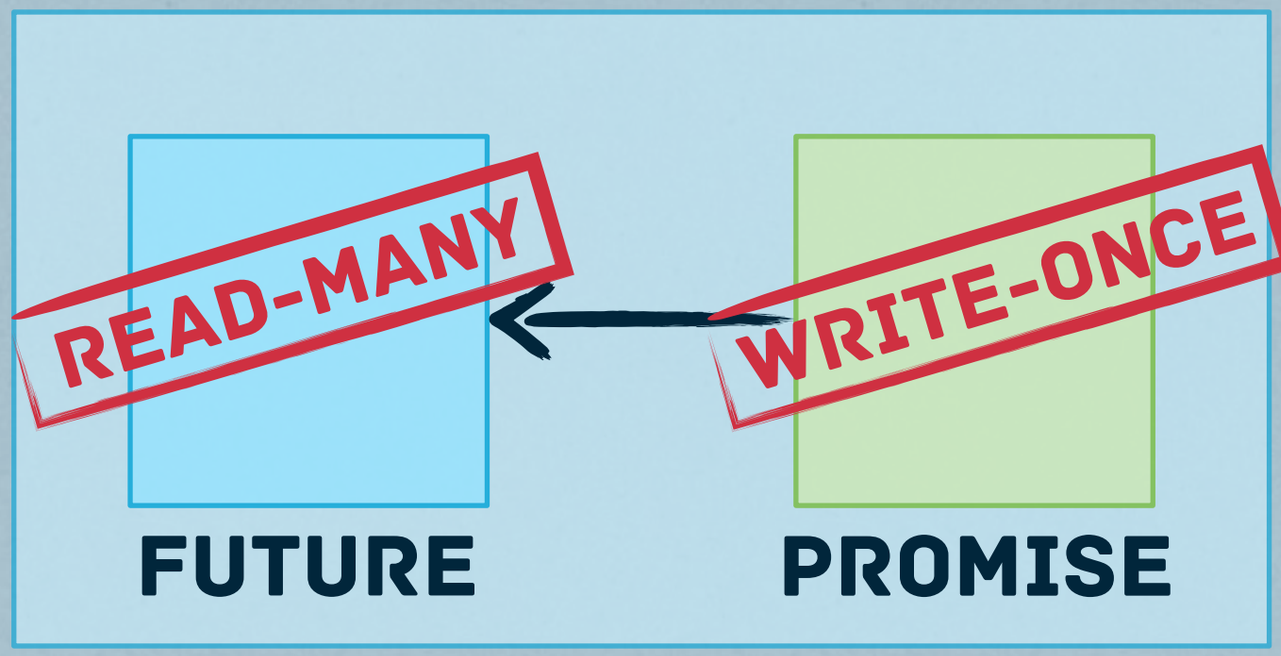
Futures & Promises

CAN BE THOUGHT OF AS A SINGLE CONCURRENCY ABSTRACTION



Futures & Promises

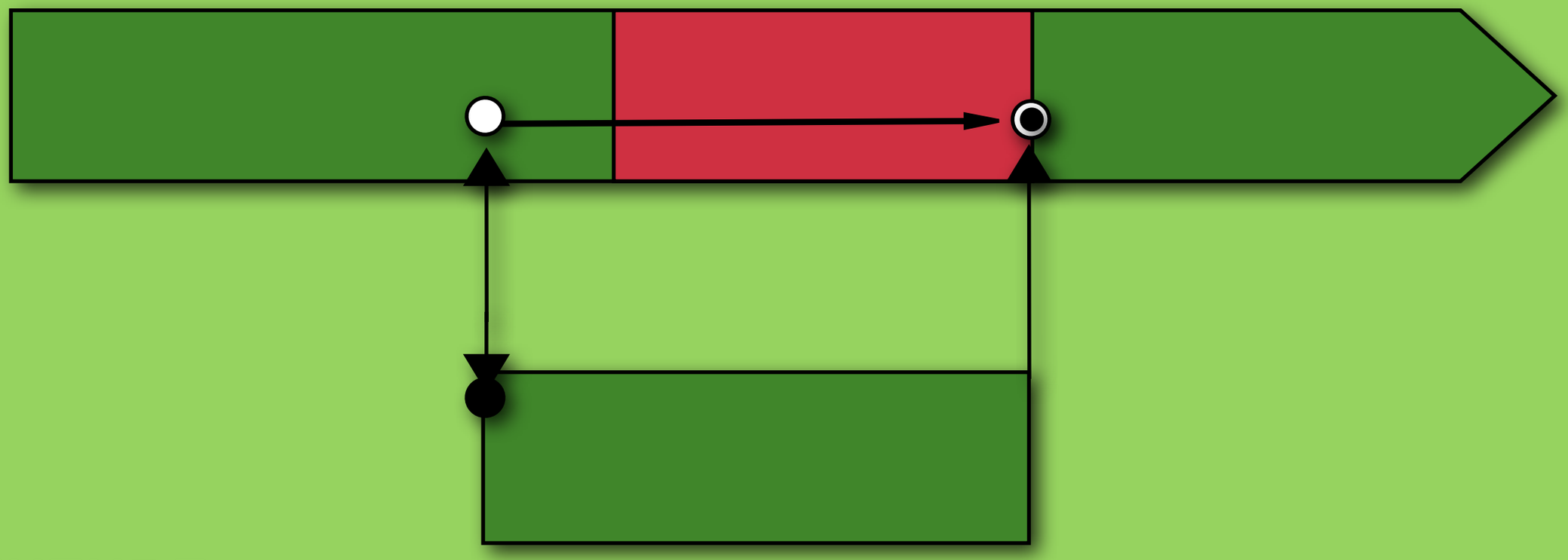
CAN BE THOUGHT OF AS A SINGLE CONCURRENCY ABSTRACTION



IMPORTANT OPS

- ✔ *Start async computation*
- ✔ *Wait for result*
- ✔ *Assign result value*
- ✔ *Obtain associated future object*

java.util.concurrent.FUTURE

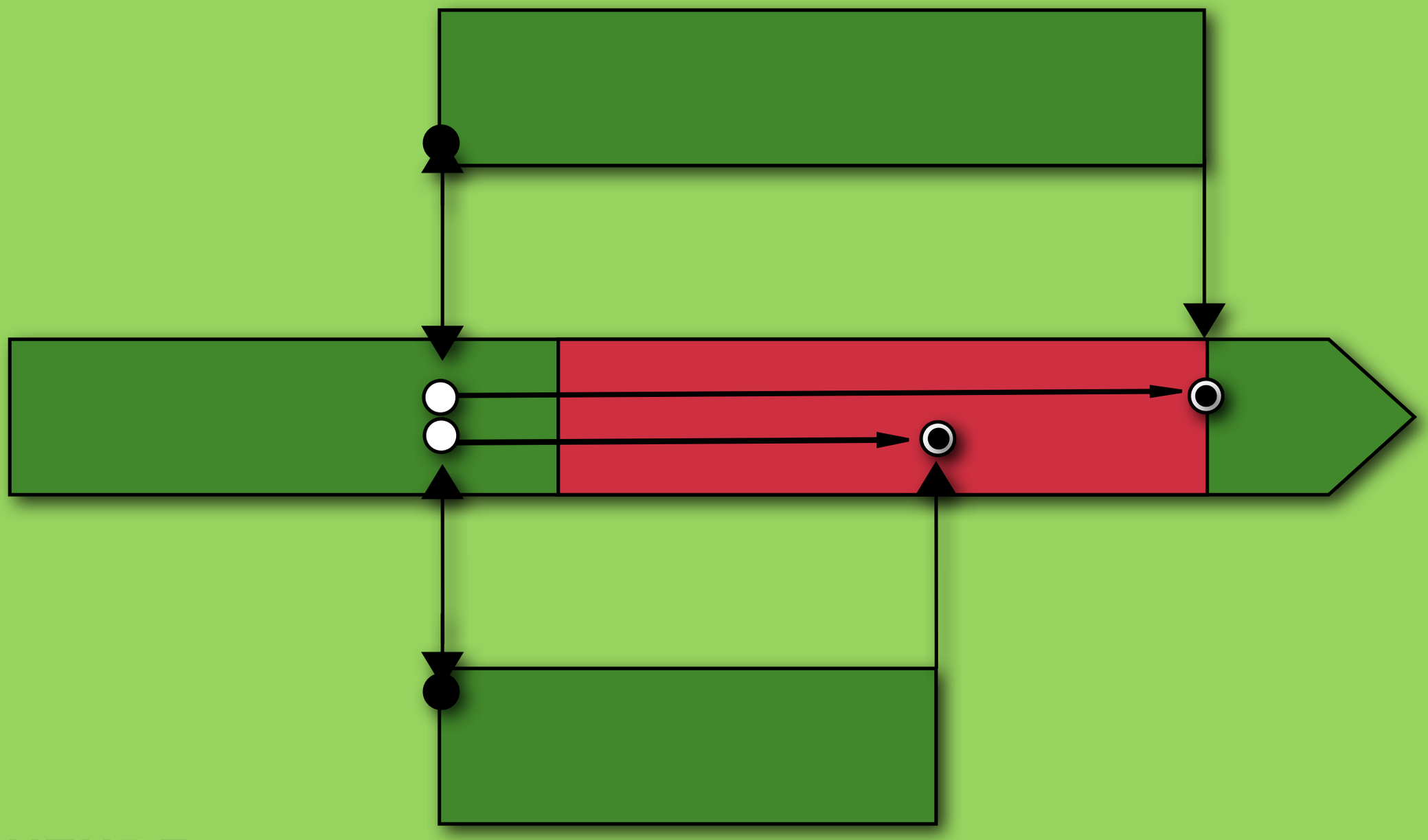


- FUTURE
- PROMISE
- FUTURE WITH VALUE

Green meaningful work

Red thread waiting on the result of another thread

java.util.concurrent.FUTURE

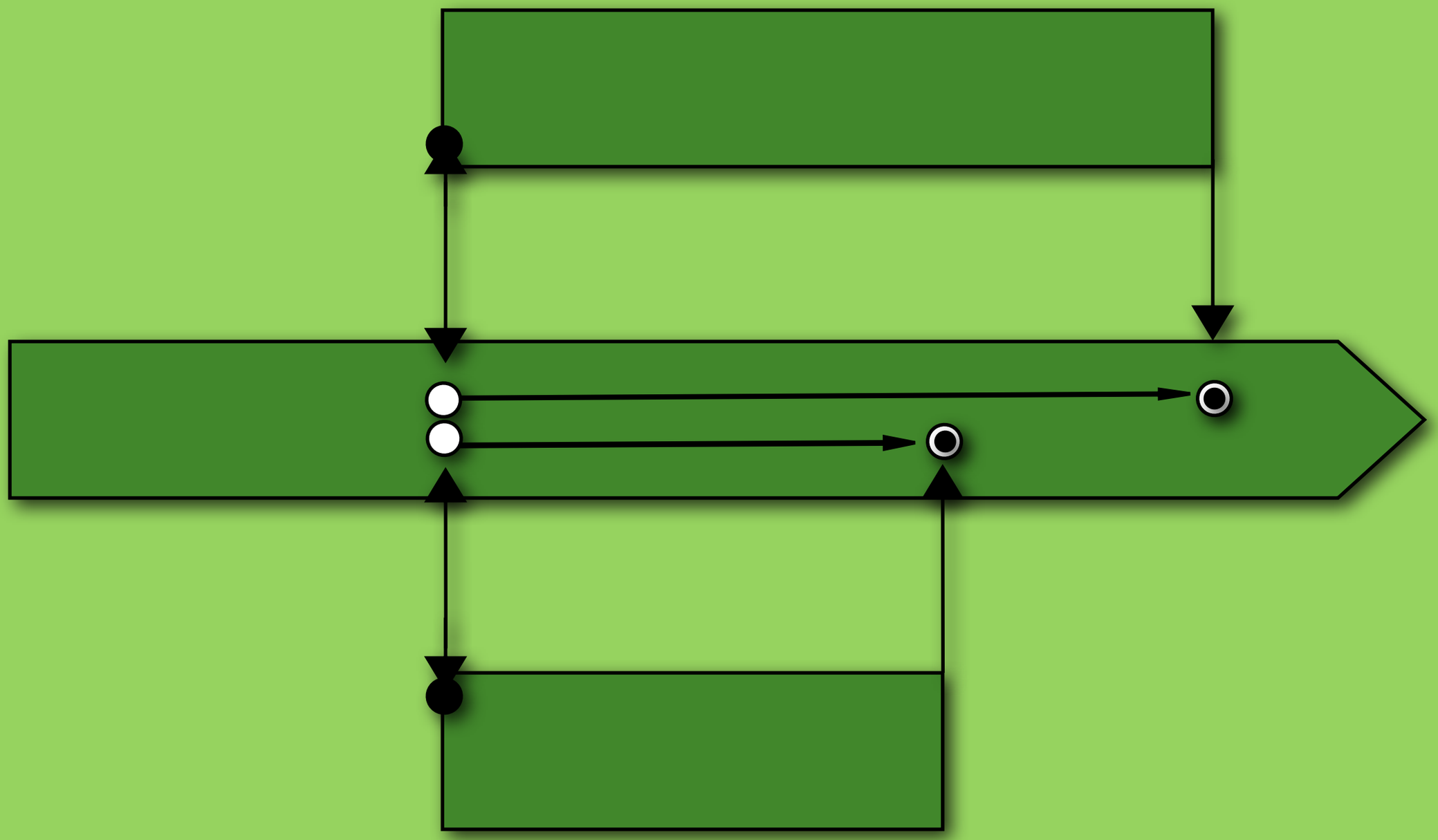


- FUTURE
- PROMISE
- FUTURE WITH VALUE

Green meaningful work

Red thread waiting on the result of another thread

what we'd like to do instead



- FUTURE
- PROMISE
- FUTURE WITH VALUE

Green meaningful work

Red thread waiting on the result of another thread

Async&NonBlocking

Async&NonBlocking

GOAL: *Do not block current thread while waiting for result of future*

Async&NonBlocking

GOAL: *Do not block current thread while waiting for result of future*

Callbacks

→ **REGISTER CALLBACK** *which is invoked (asynchronously) when future is completed*

→ **ASYNC COMPUTATIONS NEVER BLOCK** *(except for managed blocking)*

Async&NonBlocking

GOAL: *Do not block current thread while waiting for result of future*

Callbacks

→ **REGISTER CALLBACK** *which is invoked (asynchronously) when future is completed*

→ **ASYNC COMPUTATIONS NEVER BLOCK** *(except for managed blocking)*

USER DOESN'T HAVE TO EXPLICITLY MANAGE CALLBACKS. HIGHER-ORDER FUNCTIONS INSTEAD!

Success & Failure

**A PROMISE `p` OF TYPE `Promise[T]`
CAN BE COMPLETED IN TWO WAYS...**

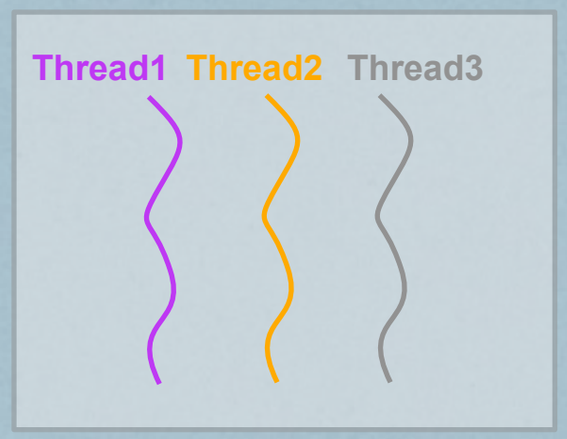
Success

```
val result: T = ...
p.success(result)
```

Failure

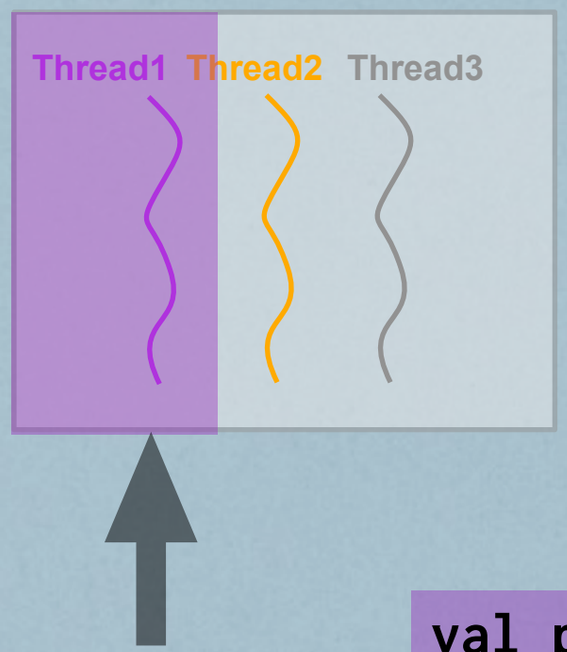
```
val exc = new Exception("something went wrong")
p.failure(exc)
```

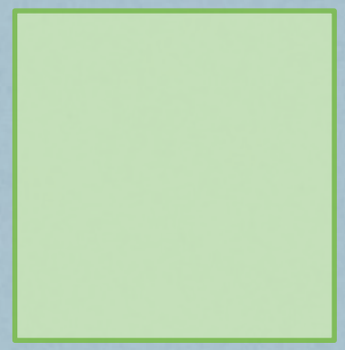
Futures&Promises
EXAMPLE



Futures&Promises

EXAMPLE





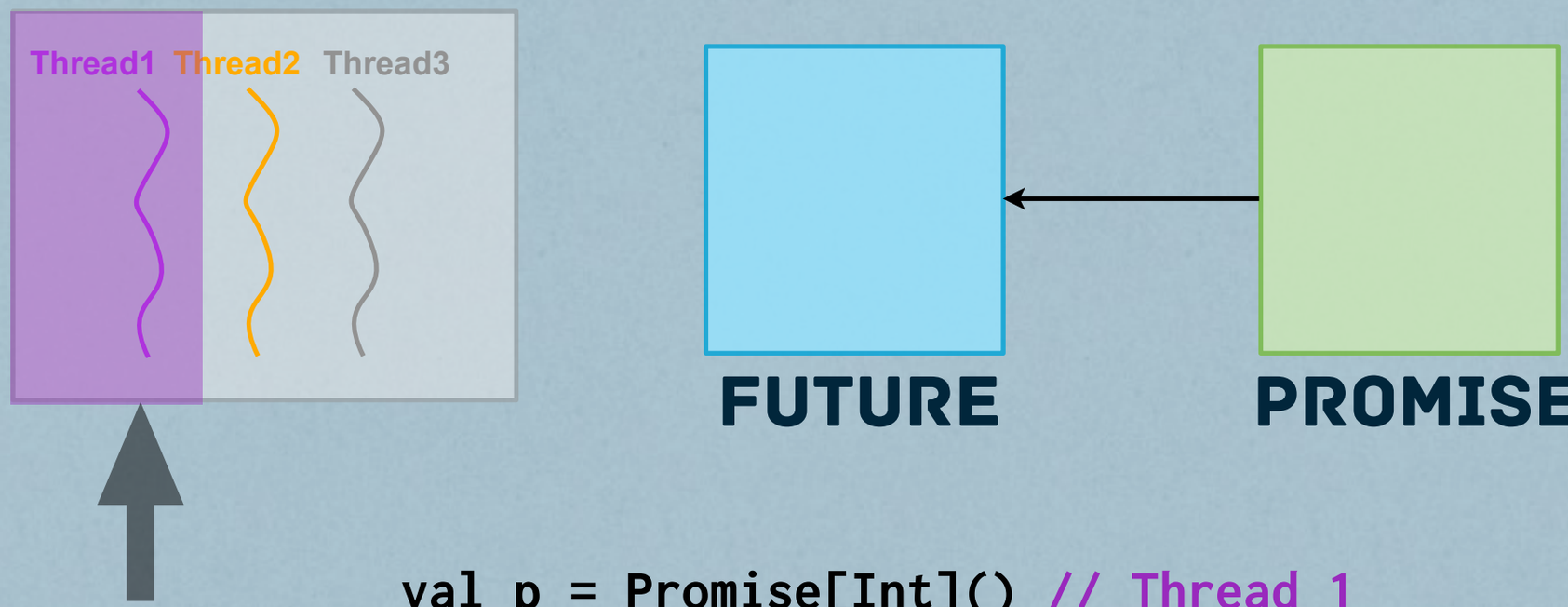
PROMISE

```
val p = Promise[Int]() // Thread 1
```

(CREATE PROMISE)

Futures&Promises

EXAMPLE



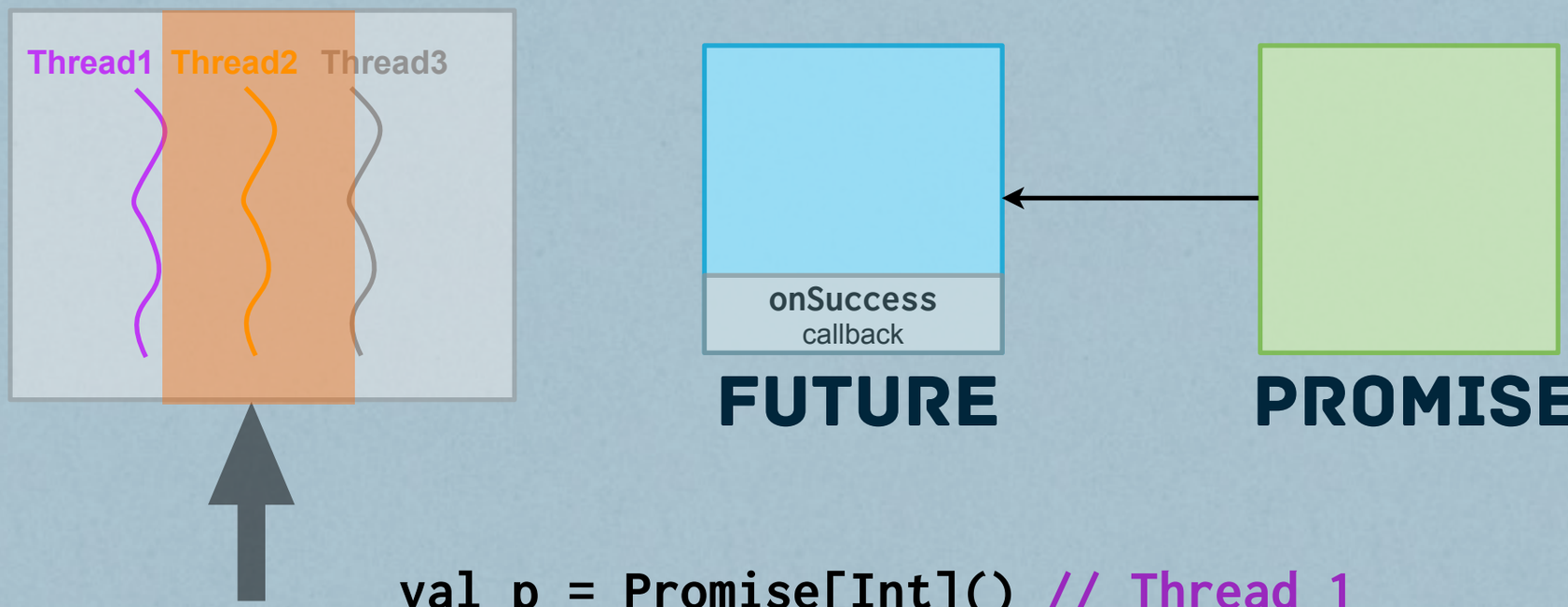

```
val p = Promise[Int]() // Thread 1
val f = p.future       // Thread 1
```

(CREATE PROMISE)

(GET REFERENCE TO FUTURE)

Futures&Promises

EXAMPLE




```
val p = Promise[Int]() // Thread 1
val f = p.future       // Thread 1
f onSuccess {          // Thread 2
  case x: Int => println(“Successful!”)
}
```

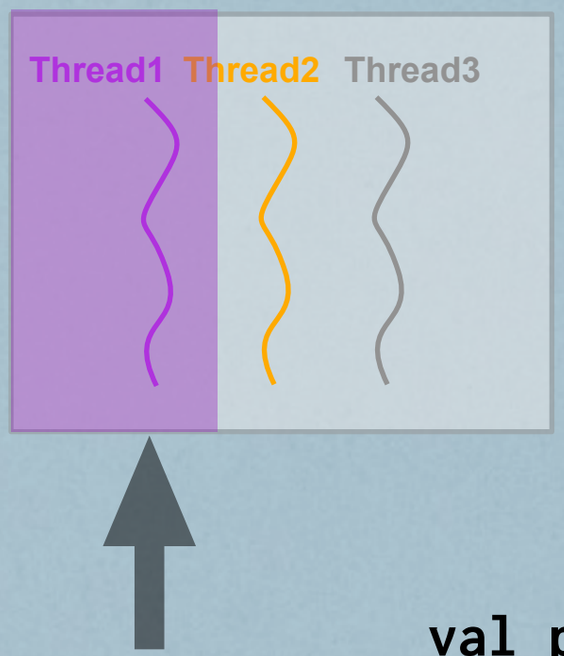
(CREATE PROMISE)

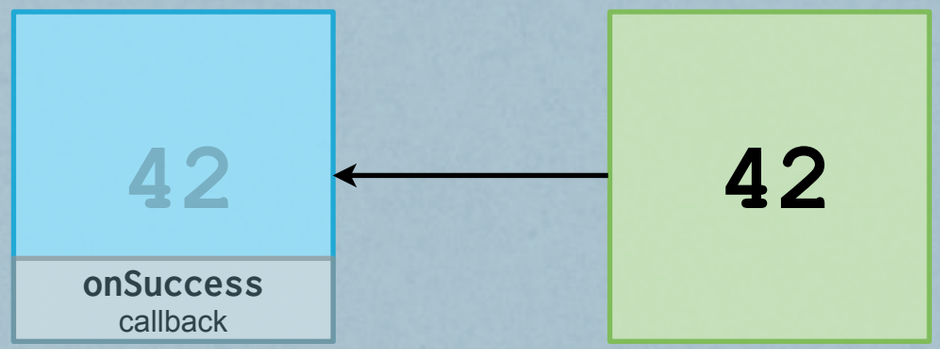
(GET REFERENCE TO FUTURE)

(REGISTER CALLBACK)

Futures&Promises

EXAMPLE





FUTURE PROMISE

```
val p = Promise[Int]() // Thread 1
val f = p.future       // Thread 1
f onSuccess {           // Thread 2
  case x: Int => println(“Successful!”)
}
p.success(42)          // Thread 1
```

(CREATE PROMISE)

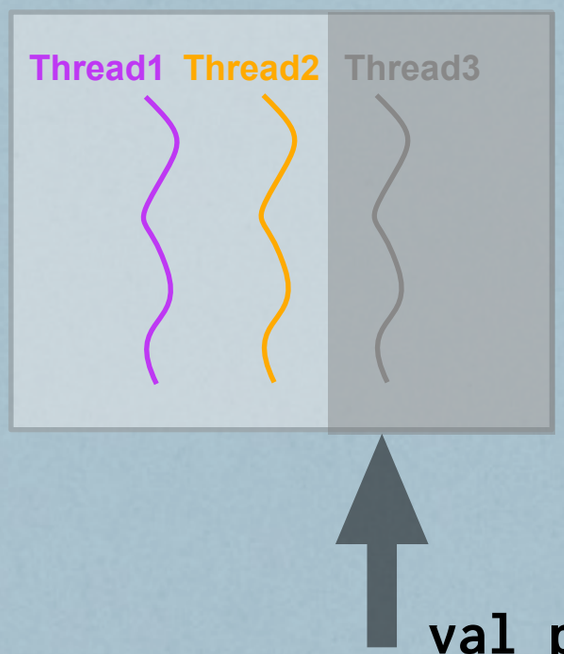
(GET REFERENCE TO FUTURE)

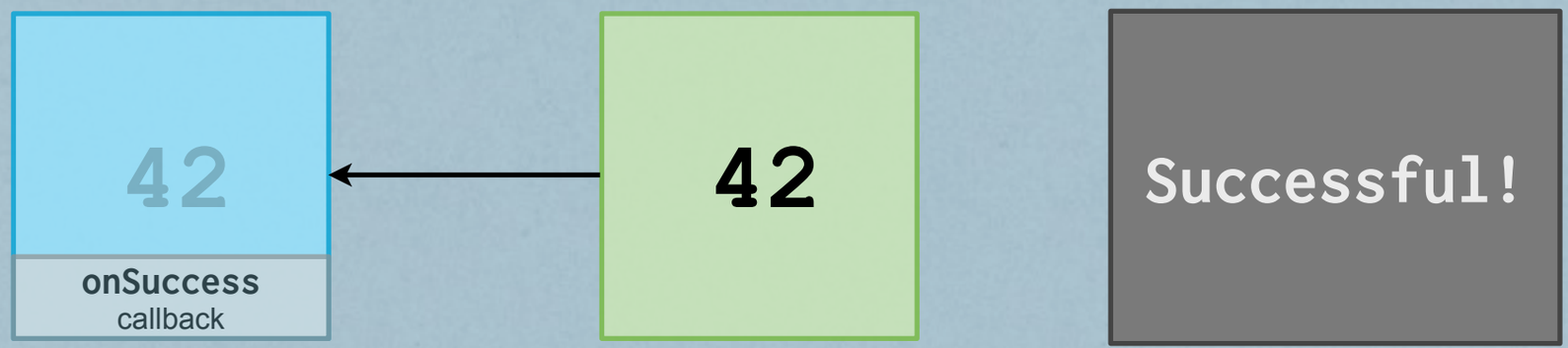
(REGISTER CALLBACK)

(WRITE TO PROMISE)

Futures&Promises

EXAMPLE





FUTURE PROMISE CONSOLE

```
val p = Promise[Int]() // Thread 1          (CREATE PROMISE)
val f = p.future       // Thread 1          (GET REFERENCE TO FUTURE)
f onSuccess {           // Thread 2         (REGISTER CALLBACK)
  case x: Int => println(“Successful!”)     (EXECUTE CALLBACK)
}                      // Thread
p.success(42)          // Thread 1          (WRITE TO PROMISE)
```

NOTE: onSuccess **CALLBACK EXECUTED EVEN IF** f **HAS ALREADY BEEN COMPLETED AT TIME OF REGISTRATION**

Combinators

- **COMPOSABILITY THRU HIGHER-ORDER FUNCS**
- **STANDARD MONADIC COMBINATORS**

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

```
val purchase: Future[Int] = rateQuote map {
  quote => connection.buy(amount, quote)
}
```

```
def filter(pred: T => Boolean): Future[T]
```

```
val postBySmith: Future[Post] =
   post.filter(_.author == “Smith”)
```

Combinators

- **COMPOSABILITY THRU HIGHER-ORDER FUNCS**
- **STANDARD MONADIC COMBINATORS**

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

```
val purchase: Future[Int] = rateQuote map {
  quote => connection.buy(amount, quote)
}
```

IF MAP FAILS: purchase is completed with unhandled exception

```
def filter(pred: T => Boolean): Future[T]
```

```
val postBySmith: Future[Post] =
   post.filter(_.author == “Smith”)
```

IF FILTER FAILS: postBySmith completed with NoSuchElementException

Combinators

ADDITIONAL FUTURE-SPECIFIC HIGHER-ORDER FUNCTIONS HAVE BEEN INTRODUCED

```
def fallbackTo[U >: T](that: Future[U]): Future[U]
```

```
def firstCompletedOf[T](futures: Traversable[Future[T]]): Future[T]
```

```
def andThen(pf: PartialFunction[...]): Future[T]
```

Combinators

ADDITIONAL FUTURE-SPECIFIC HIGHER-ORDER FUNCTIONS HAVE BEEN INTRODUCED

```
def fallbackTo[U >: T](that: Future[U]): Future[U]
```

"falls back" to **that** future in case of failure

```
def firstCompletedOf[T](futures: Traversable[Future[T]]): Future[T]
```

returns a future completed with result of first completed future

```
def andThen(pf: PartialFunction[...]): Future[T]
```

allows one to define a sequential execution over a chain of futures

Q: Which exceptions ARE CONSIDERED A Failure?

Q: Which exceptions ARE CONSIDERED A Failure?

A:

ONLY “NONFATAL” ONES

NonFatal

CAN DISTINGUISH FATAL EXCEPTIONS FROM NONFATAL ONES USING PATTERN MATCHING

```
try {
  // dangerous stuff
} catch {
  case NonFatal(e) => log.error(e, “Something not so bad”)
}
```

NonFatal

CAN DISTINGUISH FATAL EXCEPTIONS FROM NONFATAL ONES USING PATTERN MATCHING

```
try {
  // dangerous stuff
} catch {
  case NonFatal(e) => log.error(e, “Something not so bad”)
}
```

Examples of *fatal exceptions:*

`VirtualMachineError, LinkageError, ThreadDeath, ...`

Does `NonFatal` match an exception you want to throw?

Then just rethrow it

scala.concurrent.

EXECUTION CONTEXT

Threadpools... ARE NEEDED BY:

- **FUTURES** *for executing callbacks and function arguments*
- **ACTORS** *for executing message handlers, scheduled tasks, etc.*
- **PARALLEL COLLECTIONS** *for executing data-parallel operations*

Scala 2.10 introduces

EXECUTION CONTEXTS

Scala 2.10 introduces

EXECUTION CONTEXTS

Goal

PROVIDE GLOBAL THREADPOOL AS PLATFORM SERVICE TO BE SHARED BY ALL PARALLEL FRAMEWORKS

Scala 2.10 introduces

EXECUTION CONTEXTS

Goal

PROVIDE GLOBAL THREADPOOL AS PLATFORM SERVICE TO BE SHARED BY ALL PARALLEL FRAMEWORKS

- `scala.concurrent` *package provides global* `ExecutionContext`
- Default `ExecutionContext` backed by the most recent fork join pool (collaboration with Doug Lea, SUNY Oswego)

Implicit Execution Ctxs

Asynchronous computations are executed on an `ExecutionContext` which is provided implicitly.

```
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S]

def onSuccess[U](pf: PartialFunction[T, U])
                (implicit executor: ExecutionContext): Unit
```

Implicit parameters enable (fine-grained) selection of the `ExecutionContext`:

```
implicit val context: ExecutionContext = customExecutionContext
val fut2 = fut1.filter(pred)
               .map(fun)
```

Implicit Execution Ctxs

IMPLICIT ExecutionContexts **ALLOW SHARING ECS BETWEEN FRAMEWORKS**

```
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S]

def onSuccess[U](pf: PartialFunction[T, U])
                (implicit executor: ExecutionContext): Unit
```

ENABLES FLEXIBLE SELECTION OF EXECUTION POLICY

```
implicit val context: ExecutionContext = customExecutionContext
val fut2 = fut1.filter(pred)
               .map(fun)
```

Future

THE IMPLEMENTATION

Many operations implemented in terms of promises

SIMPLIFIED EXAMPLE

```
def map[S](f: T => S): Future[S] = {
  val p = Promise[S]()

  onComplete {
    case result =>
      try {
        result match {
          case Success(r) => p success f(r)
          case Failure(t) => p failure t
        }
      } catch {
        case t: Throwable => p failure t
      }
  }
  p.future
}
```

Future
THE *REAL* IMPLEMENTATION

The real implementation (a) adds an implicit `ExecutionContext`, (b) avoids extra object creations, and (c) catches only non-fatal exceptions:

```
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S] = {
  val p = Promise[S]()

  onComplete {
    case result =>
      try {
        result match {
          case Success(r) => p success f(r)
          case f: Failure[_] => p complete f.asInstanceOf[Failure[S]]
        }
      } catch {
        case NonFatal(t) => p failure t
      }
  }

  p.future
}
```

Promise

THE IMPLEMENTATION

`Promise` *is the work horse of the futures implementation.*

A `Promise[T]` *can be in one of two states:*

PENDING

No result has been written to the promise.
State represented using a list of callbacks (initially empty).

COMPLETED

The promise has been assigned a successful result or exception.
State represented using an instance of `Try[T]`

Invoking `Promise.complete` triggers a transition from state `Pending` to `Completed`

A PROMISE CAN BE COMPLETED AT MOST ONCE:

```
def complete(result: Try[T]): this.type =
  if (tryComplete(result)) this
  else throw new IllegalStateException("Promise already completed.")
```

scala.util.TRY

AND NOW ONTO SOMETHING COMPLETELY DIFFERENT.

AND NOW ONTO SOMETHING

NOT CONCURRENT,
NOT ASYNCHRONOUS

Try
IS A SIMPLE DATA CONTAINER

Great for monadic-style exception handling.

✔ *Composable*

✔ *Combinators for exceptions*

“DIVORCING EXCEPTION HANDLING FROM THE STACK.”

Try IS A SIMPLE DATA CONTAINER

```scala
sealed abstract class Try[+T]

final case class Success[+T](value: T) extends Try[T]

final case class Failure[+T](exception: Throwable)
  extends Try[T]
```

Try IS A SIMPLE DATA CONTAINER

```scala
sealed abstract class Try[+T]

final case class Success[+T](value: T) extends Try[T]

final case class Failure[+T](exception: Throwable)
  extends Try[T]
```

Try IS A SIMPLE DATA CONTAINER

```scala
sealed abstract class Try[+T]

final case class Success[+T](value: T) extends Try[T]

final case class Failure[+T](exception: Throwable)
  extends Try[T]
```

METHODS ON *Try*

BASIC OPS

get

```
def get: T
```

SUCCESS

Returns `value` *stored within* `Success`

FAILURE

Throws exception stored within `Failure`

METHODS ON *Try*

BASIC OPS

getOrElse

```
def getOrElse[U >: T](default: => U): U
```

SUCCESS

Returns `value` *stored within* `Success`

FAILURE

Returns the given `default` *argument if this is a* `Failure`

METHODS ON *Try*

BASIC OPS

orElse

```
def orElse[U >: T](default: => Try[U]): Try[U]
```

SUCCESS

Returns `this` *Try if this is a* `Success`

FAILURE

Returns the given `default` *argument if this is a* `Failure`

METHODS ON *Try*

MONADIC OPS

map

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

SUCCESS

Applies the function `f` *to the* `value` *from* `Success`

FAILURE

Returns `this` *if this is a* `Failure`

METHODS ON *Try*

MONADIC OPS

flatMap

```
def flatMap[U](f: T => Try[U]): Try[U]
```

SUCCESS

Applies the function `f` *to the* `value` *from* `Success`

FAILURE

Returns `this` *if this is a* `Failure`

METHODS ON *Try*

MONADIC OPS

filter

```
def filter(p: T => Boolean): Try[T]
```

SUCCESS

Converts `this` *to a* `Failure` *if predicate* `p` *not satisfied.*

FAILURE

Returns `this` *if this is a* `Failure`

METHODS ON *Try*

EXCEPTION-SPECIFIC OPS

recover

```
def recover[U >: T](f: PartialFunction[Throwable, U]): Try[U]
```

SUCCESS

Returns `this` *if this is a* `Success`

FAILURE

Applies function `f` *if this is a* `Failure`. (`map` *on exptn*)

METHODS ON *Try*

EXCEPTION-SPECIFIC OPS

recoverWith

```
def recoverWith[U >: T](f: PartialFunction[Throwable, Try[U]]): Try[U]
```

SUCCESS

Returns `this` *if this is a* Success

FAILURE

Applies function `f` *if this is a* `Failure`. (`flatMap` *on exptn*)

METHODS ON *Try*

EXCEPTION-SPECIFIC OPS

transform

```
def transform[U](s: T => Try[U], f: Throwable => Try[U]): Try[U]
```

SUCCESS

Creates Try by applying function `s` *if this is a* `Success`

FAILURE

Creates Try by applying function `f` *if this is a* `Failure`

USING THESE TO BUILD *Pipelines*

USING THESE TO BUILD *Pipelines*

REMEMBER: NOT CONCURRENT, NOT ASYNCHRONOUS

SIMPLE PIPELINING ON *Try* EXAMPLE 1

```
case class Account(acctNum: Int, balance: Double, interestRate: Double)

val withdrawal = 1500
val adjustment = 0.4
val in = Try(getAcct())

val withdrawalResult = in map {
    (x: Account) => Account(x.acctNum, x.balance - withdrawal, x.interestRate)
  } filter {
    (x: Account) => x.balance > 12000 // acct in good standing
  } map {
    (x: Account) =>
    val toUpdate = Account(x.acctNum, x.balance, x.interestRate + adjustment)
    updateAcct(toUpdate)
  }
```

SIMPLE PIPELINING ON *Try* EXAMPLE 1

```
case class Account(acctNum: Int, balance: Double, interestRate: Double)

val withdrawal = 1500
val adjustment = 0.4
val in = Try(getAcct())

val withdrawalResult = in map {
    (x: Account) => Account(x.acctNum, x.balance - withdrawal, x.interestRate)
  } filter {
    (x: Account) => x.balance > 12000 // acct in good standing
  } map {
    (x: Account) =>
    val toUpdate = Account(x.acctNum, x.balance, x.interestRate + adjustment)
    updateAcct(toUpdate)
  }
```

GETACCT MIGHT FAIL

SIMPLE PIPELINING ON *Try*

EXAMPLE 1

```
case class Account(acctNum: Int, balance: Double, interestRate: Double)

val withdrawal = 1500
val adjustment = 0.4
val in = Try(getAcct())

val withdrawalResult = in map {
    (x: Account) => Account(x.acctNum, x.balance - withdrawal, x.interestRate)
  } filter {
    (x: Account) => x.balance > 12000 // acct in good standing
  } map {
    (x: Account) =>
    val toUpdate = Account(x.acctNum, x.balance, x.interestRate + adjustment)
    updateAcct(toUpdate)
  }
```

PREDICATE MIGHT NOT BE SATISFIED

SIMPLE PIPELINING ON *Try* EXAMPLE 1

```
case class Account(acctNum: Int, balance: Double, interestRate: Double)

val withdrawal = 1500
val adjustment = 0.4
val in = Try(getAcct())

val withdrawalResult = in map {
    (x: Account) => Account(x.acctNum, x.balance - withdrawal, x.interestRate)
  } filter {
    (x: Account) => x.balance > 12000 // acct in good standing
  } map {
    (x: Account) =>
    val toUpdate = Account(x.acctNum, x.balance, x.interestRate + adjustment)
    updateAcct(toUpdate)
  }
```

UPDATEACCT MIGHT FAIL

SIMPLE PIPELINING ON *Try*

EXAMPLE 1

```
case class Account(acctNum: Int, balance: Double, interestRate: Double)

val withdrawal = 1500
val adjustment = 0.4
val in = Try(getAcct())

val withdrawalResult = in map {
    (x: Account) => Account(x.acctNum, x.balance - withdrawal, x.interestRate)
  } filter {
    (x: Account) => x.balance > 12000 // acct in good standing
  } map {
    (x: Account) =>
    val toUpdate = Account(x.acctNum, x.balance, x.interestRate + adjustment)
    updateAcct(toUpdate)
  }
```

ELIMINATES NESTED TRY BLOCKS

but how can we handle these failures?

SIMPLE PIPELINING ON *Try* EXAMPLE 2

...by using recoverWith, recover, or orElse

```
case class Tweet(from: String, retweets: Int)

val importantTweets = Try {
  server.getTweetList()
} orElse {
  cachedTweetList.get
} filter { twts =>
  val avgRetweet = twts.map(_.retweets).reduce(_ + _) / twts.length
  twts.exists(_.retweets > 2 * avgRetweet)
} recover {
  case nose: NoSuchElementException => // handle individually
  case usop: ArithmeticException    => // handle individually
  case other                        => // handle individually
}
```

COMBINING Try& Futures

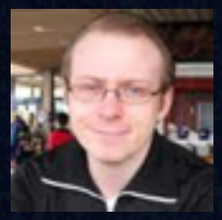
```
case class Friend(name: String, age: String)

val avgAge = Promise[Int]()

val fut = future {
  // query a social network...
  List(Friend("Zoe", "25"), Friend("Jean", "27"), Friend("Paul", "30"))
}

fut onComplete { tr =>
  // compute average age of friends
  val result = tr map {
    friends => friends.map(_.age.toInt).reduce(_ + _) / friends.length
  }
  avgAge complete result
}
```

CREDITS

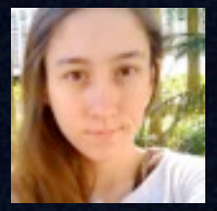


VIKTOR KLANG
TYPESAFE

PHILIPP HALLER
TYPESAFE

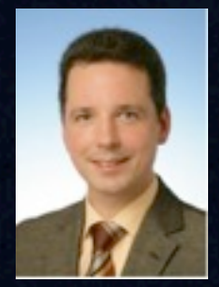
MARIUS ERIKSEN
TWITTER

HEATHER MILLER
EPFL



ALEX PROKOPEC
EPFL

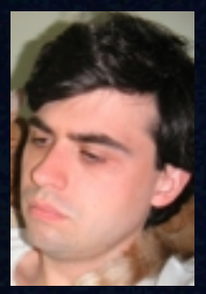
ROLAND KUHN
TYPESAFE



VOJIN JOVANOVIC
EPFL

DOUG LEA
SUNY

HAVOC PENNINGTON
TYPESAFE



QUESTIONS?

http://docs.scala-lang.org/sips/pending/futures-promises.html