Agenda

FUTURES/PROMISES
EXECUTION CTXS
FUTURES IN PLAY
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
THIS MAKES IT CLEAR THAT...
THIS MAKES IT CLEAR THAT...

FUTURES ARE AN IMPORTANT, POWERFUL ABSTRACTION
FUTURES ARE AN IMPORTANT, POWERFUL ABSTRACTION

THERE'S FRAGMENTATION IN THE SCALA ECOSYSTEM

no hope of interop!
Furthermore...
Furthermore...

JAVA FUTURES NEITHER EFFICIENT NOR COMPOSABLE
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 combined concurrency abstraction.
Futures & Promises can be thought of as a combined concurrency abstraction.
Futures & Promises can be thought of as a combined concurrency abstraction.

**IMPORTANT OPS**
- ✔ Start async computation
- ✔ Wait for result
- ✔ Assign result value
- ✔ Obtain associated future object
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)`
Future Promise Future with value

Green meaningful work
Red thread waiting on the result of another thread

java.util.concurrent.FUTURE
java.util.concurrent.Future

Promise

Future with value

Green

meaningful work

Red

thread waiting on the result of another thread

FUTURE

PROMISE

FUTURE WITH VALUE
what we'd like to do instead

Future
Promise
Future with value

Green
Red
thread waiting on the result of another thread

meaningful work

FUTURE
PROMISE
FUTURE WITH VALUE
Async & Non-Blocking
Async & Non-Blocking

**GOAL:** Do not block current thread while waiting for result of future
Async & Non-Blocking

**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 & Non-Blocking

**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!**
Futures & Promises

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

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

f onSuccess {
  // Thread 2
  case x: Int => println("Successful!")
}

(Thread1) (create promise)
(Thread2) (get reference to future)
(Thread3) (register callback)
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)
val p = Promise[Int]() // Thread 1
val f = p.future // Thread 1
f onSuccess {
  case x: Int => println(“Successful!”) // Thread 2
}
p.success(42) // Thread 1

(CREATE PROMISE)
(GET REFERENCE TO FUTURE)
(REGISTER CALLBACK)
(EXECUTE CALLBACK)
(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**

```scala
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

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

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

**If map fails**: purchase is completed with unhandled exception

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

```scala
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.

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

val fut: Future[T] = Future.firstCompletedOf[T](futures)

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

**ADDITIONAL FUTURE-SPECIFIC HIGHER-ORDER FUNCTIONS HAVE BEEN INTRODUCED**

```scala
def fallbackTo[U <: T](that: Future[U]): Future[U]  
  "falls back" to `that` future in case of failure

val fut: Future[T] = Future.firstCompletedOf[T](futures)  
  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
```
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 global threadpool as platform service to be shared by all parallel frameworks.

**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.

```scala
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`:

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

**IMPLICIT** ExecutionContexts ALLOW SHARING ECS BETWEEN FRAMEWORKS

```scala
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

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

Many operations implemented in terms of promises

SIMPLIFIED EXAMPLE

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

```scala
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 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:

```python
def complete(result: Try[T]): this.type =
  if (tryComplete(result)) this
  else throw new IllegalStateException("Promise already completed.")
```
def tryComplete(value: Try[T]): Boolean = {
    val resolved = resolveTry(value)
    (try {
        @tailrec
        def tryComplete(v: Try[T]): List[CallbackRunnable[T]] = {
            getState match {
                case raw: List[_] =>
                    val cur = raw.asInstanceOf[List[CallbackRunnable[T]]]
                    if (updateState(cur, v)) cur else tryComplete(v)
                case _ => null
            }
        }
        tryComplete(resolved)
    } finally {
        synchronized { notifyAll() } // Notify any blockers
    }) match {
        case null => false
        case rs if rs.isEmpty => true
        case rs =>
            rs.foreach(_.executeWithValue(resolved)); true
    }
}
abstract class AbstractPromise {
    private volatile Object _ref;
    final static long _refoffset;

    static {
        try {
            _refoffset = Unsafe.instance.objectFieldOffset(
                AbstractPromise.class.getDeclaredField("_ref"));
        } catch (Throwable t) {
            throw new ExceptionInInitializerError(t);
        }
    }

    protected boolean updateState(Object oldState, Object newState) {
        return Unsafe.instance.compareAndSwapObject(this, _refoffset,
            oldState, newState);
    }

    protected final Object getState() {
        return _ref;
    }
}
INTEGRATING Futures & Actors

Futures are results of asynchronous message sends when a response is expected.

val response: Future[Any] = socialGraph ? getFriends(user)

Implementing synchronous send (untyped):

def syncSend(to: ActorRef, msg: Any, timeout: Duration): Any = {
    val fut = to ? msg
    Await.result(fut, timeout)
}

RECOVERING TYPES

val friendsFut: Future[Seq[Friend]] = response.mapTo[Seq[Friend]]
INTEGRATING Futures & Actors

Futures are results of asynchronous message sends WHEN A RESPONSE IS EXPECTED

val response: Future[Any] = socialGraph ? getFriends(user)

friendsFut IS EITHER COMPLETED WITH A SUCCESSFUL RESULT OR WITH A WRAPPED EXCEPTION IF RESPONSE TIMES OUT OR IS NOT OF TYPE Seq[Friend]

RECOVERING TYPES

val friendsFut: Future[Seq[Friend]] = response.mapTo[Seq[Friend]]
THE PLAY

Example
Ye Olde Webapp

databases

ORM

Hibernate

Oracle Database
The Future of WebApps is Now

Services

XYZ
Synchronous IO

Thread 1

Thread 2

Blocking

Blocking

Important work

Waiting for response
Synchronous IO

MEANS: \( N \text{ requests} = N \text{ threads} \)
Synchronous IO

**Means:**

\[ N \text{ requests} = N \text{ threads} \]
Asynchronous IO
Asynchronous I/O
Asynchronous IO

MEANS: We now scale!
GET / HTTP/1.1
User-Agent: ...

HTTP/1.1: 200 Ok
Location: ....

Client
Controller
Routing
Action
models
views
Controller

Request → Action → Result
package controllers

//imports...

object Application extends Controller {

  def index = Action { request =>
    Ok("It is November 19th - there are 42 days left of the year!")
  }

}
package controllers

//imports...

object Application extends Controller {

  def index = Action { request =>
    val dayOfYear = ???
    Ok(s"It is $dayOfYear - there are 42 days left of the year!"")
  }

}
package controllers

//imports...

object Application extends Controller {

  def index = Action { request =>
    f.map { response =>
      val dayOfYear = response.body
      Ok(s"It is $dayOfYear - there are 42 days left of the year!")
    }
  }
}

Play Future in
package controllers

//imports...

object Application extends Controller {

  def index = Action { request =>
    import play.api.libs.concurrent.Execution.Implicits._
    import play.api.libs.concurrent.Execution.Implicits._
    import play.api.libs.concurrent.Execution.Implicits._
    import play.api.libs.concurrent.Execution.Implicits._
    Async {
      f.map { response =>
        val dayOfYear = response.body
        Ok(s"It is $dayOfYear - there are 42 days left of the year!")
      }
    }
  }
}
def index = Action { request =>
    import play.api.libs.concurrent.Execution.Implicits._
    Async {
        val futureDOYResponse: Future[Response] =
            WS.url("http://api.day-of-year/today").get
        val futureDaysLeftResponse: Future[Response] =
            WS.url("http://api.days-left/today").get
    }
}
def index = Action { request =>
  import play.api.libs.concurrent.Execution.Implicits._
  Async {
    val futureDOYResponse: Future[Response] =
      WS.url("http://api.day-of-year/today").get
    val futureDaysLeftResponse: Future[Response] =
      WS.url("http://api.days-left/today").get
    futureDOYResponse.map { doyResponse =>
      val dayOfYear = doyResponse.body
      futureDaysLeftResponse.map { daysLeftResponse =>
        val daysLeft = daysLeftResponse.body
        Ok(s "It is $dayOfYear - there are $daysLeft days left of the year!")
      }
    }
  }
}
def index = Action { request =>
    import play.api.libs.concurrent.Execution.Implicits._
    Async {
        val futureDOYResponse: Future[Response] =
            WS.url("http://api.day-of-year/today").get
        val futureDaysLeftResponse: Future[Response] =
            WS.url("http://api.days-left/today").get

        futureDOYResponse.map{ doyResponse =>
            val dayOfYear = doyResponse.body
            futureDaysLeftResponse.map { daysLeftResponse =>
                val daysLeft = daysLeftResponse.body
                Ok(s "It is $dayOfYear - there are $daysLeft days left of the year!")
            }
        }
    }
}
def index = Action { request =>
    import play.api.libs.concurrent.Execution.Implicits._
    Async {
        val futureDOYResponse: Future[Response] =
            WS.url("http://api.day-of-year/today").get
        val futureDaysLeftResponse: Future[Response] =
            WS.url("http://api.days-left/today").get
        futureDOYResponse.flatMap{ doyResponse =>
            val dayOfYear = doyResponse.body
            futureDaysLeftResponse.map { daysLeftResponse =>
                val daysLeft = daysLeftResponse.body
                Ok(s "It is $dayOfYear - there are $daysLeft days left of the year!")
            }
        }
    }
}
def index = Action { request =>
    import play.api.libs.concurrent.Execution.Implicits._
    Async {
        val futureDOYResponse: Future[Response] =
            WS.url("http://api.day-of-year/today").get
        val futureDaysLeftResponse: Future[Response] =
            WS.url("http://api.days-left/today").get
        for {
            doyResponse <- futureDOYResponse
            dayOfYear = doyResponse.body
            daysLeftResponse <- futureDaysLeftResponse
            daysLeft = daysLeftResponse.body
        } yield {
            Ok(s"It is $dayOfYear - there are $daysLeft days left of the year!")
        }
    }
}
Async {
    val futureDOYResponse: Future[Response] = // ...
    val futureDaysLeftResponse: Future[Response] = // ...

    val futureResult = for {
        doyResponse <- futureDOYResponse
        dayOfYear = doyResponse.body
        daysLeftResponse <- futureDaysLeftResponse
        daysLeft = daysLeftResponse.body
    } yield {
        Ok(s"It is $dayOfYear - there are $daysLeft days left of the year!")
    }

    futureResult.recover {
        case t: Throwable =>
            BadRequest(s"It is 21st December 2012 - end of the world?")
    }
}
QUESTIONS?

http://docs.scala-lang.org/sips/pending/futures-promises.html
http://www.playframework.org/documentation/2.0.4/ScalaAsync