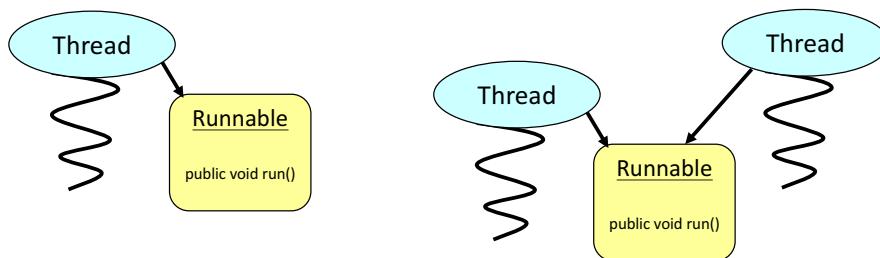


Functional Concurrent and Asynchronous Programming in Scala



JVM Threading: Threads and Runnable Objects

- Thread encapsulates a schedulable entity
- Runnable interface used to represent work for thread to do
- Scala provides access to these "primitive" features



JVM Threading: Threads and Runnable Objects

- Easy to use basic JVM threading from Scala

```
class TickTock ( val word: String, val delay: Int ) extends Runnable {
    override def run = {
        while ( !Thread.interrupted ) {
            try {
                print(s"$word ")
                Thread.sleep(delay)
            } catch {
                case ie: InterruptedException => {
                    println("Interrupted: shutting down");
                    Thread.currentThread.interrupt
                }
            }
        }
    }
}
```

© J&G Services Ltd, 2017

JVM Threading: Threads and Runnable Objects

- Two instances of the task executed in separate threads

```
object TickTock extends App {
    val t1 = new Thread(new TickTock("tick", 500))
    val t2 = new Thread(new TickTock("tock", 750))

    t1.start
    t2.start

    Thread.sleep(5000)

    t1.interrupt
    t2.interrupt
}
```

tock tick tick tock tick tock tick tick tock ... Interrupted: shutting down
Interrupted: shutting down

© J&G Services Ltd, 2017

JVM Threading: Threads and Runnable Objects

- Executor framework simplifies execution

```
object TickTockExecutor extends App {
    import java.util.concurrent._

    val ticker = new TickTock("tick", 500)
    val tocker = new TickTock("tock", 750)

    val engine = Executors.newFixedThreadPool(2)
    engine.execute(ticker)
    engine.execute(tocker)

    Thread.sleep(5000)
    engine.shutdownNow
}
```

tock tick tick tock tick tock tick tick tock ...Interrupted: shutting down
Interrupted: shutting down

© J&G Services Ltd, 2017

Problems

- Limited capabilities
 - run method returns Unit
 - often want task to return a value
- Scalability is limited
 - threads relatively heavyweight objects
 - limited number can be supported in VM
- Shared state difficult to protect
 - locks/synchronized blocks
 - introduces complexity to code
 - difficult to debug or reason about
 - blocking threads wastes resources

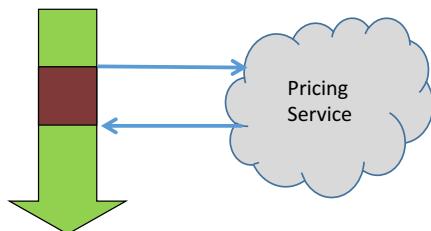


© J&G Services Ltd, 2017

Asynchronous Programming

- A different approach
 - "do something" while calling thread continues
 - hand back a result when finished
- Higher level of abstraction

```
val price = PricingService.getPrice(...)
```

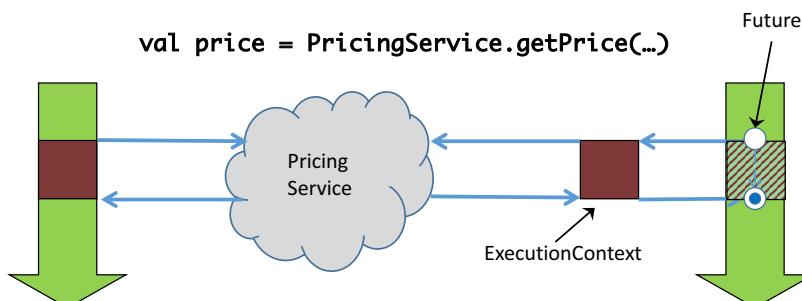


© J&G Services Ltd, 2017

Asynchronous Programming

- A different approach
 - "do something" while calling thread continues
 - hand back a result when finished
- Higher level of abstraction

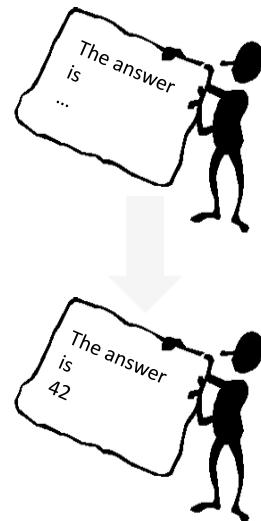
```
val price = PricingService.getPrice(...)
```



© J&G Services Ltd, 2017

Futures

- Placeholder representing a value that will be available
 - at some time in the future
- Common idea in asynchronous programming
- Implementations available in different languages
 - Java
 - Scala
 - ...



© J&G Services Ltd, 2017

Futures

- Service with asynchronous interface

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

object PricingService {
  val prices: Map[String, Double] = Map (
    "GOOG" -> 733.20,
    "AAPL" -> 112.0,
    "MSFT" -> 35.33
  )
  def priceForSym( sym: String ): Future[Double] = Future {
    Thread.sleep(2000)
    prices(sym)
  }
}
```

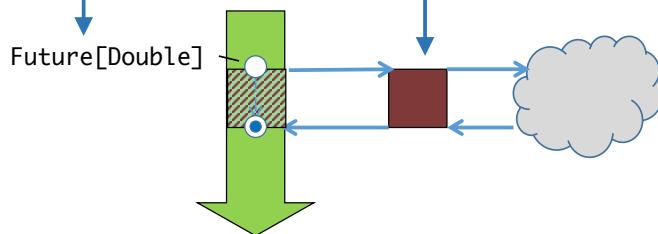
© J&G Services Ltd, 2017

Futures

- Using the service

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

val fPrice = PricingService.priceForSym("AAPL")
```



© J&G Services Ltd, 2017

Futures – Completion

- Two possibilities
 - operation succeeds and result returned
 - operation fails and Throwable object returned
- Use callbacks for each case

```
val pf = PricingService.priceForSym( ... )

...
pf onSuccess {
  case p: Double => println(f"The price is $p%.4f")
}
The price is 500.1000

pf onFailure {
  case ex: Throwable => println(s"There was a problem: $ex")
}
There was a problem: java.util.NoSuchElementException: key not found: IBM
```

© J&G Services Ltd, 2017

Futures – Completion

- Unified callback handling both success and failure
 - function passed object of type Try[T]

```
...  
import scala.util.{Success,Failure}  
...  
pf onComplete {  
    case Success(p: Double) => println(f"Price: $p%.4f")  
    case Failure(ex) => println(s"Failed: $ex")  
}  
...
```

© J&G Services Ltd, 2017

Futures – Completion

- Future type provides foreach method
 - Asynchronously invokes supplied function to process future value once it becomes available

```
scala> PricingService.priceForSym("AAPL")  
|           .foreach ( p => println(s"Price for AAPL: $p") )  
scala> Price for AAPL: 112.0
```

© J&G Services Ltd, 2017

Working with Futures

- Possible to block for a Future[T]
 - not advised unless unavoidable
- Await.result(*future, duration*)
 - return value of *future* on success
 - throw exception if completion fails
 - throw timeout exception if *duration* expires
- Await.ready(*future, duration*)
 - return *future* object
 - throw timeout exception if *duration* expires

```
import scala.concurrent.duration._

...
val pf = Future { PricingService.priceForSym("AAPL") }
val price = Await.result(pf, 2 seconds)
println(f"Waited for result: $price%.4f")
```

© J&G Services Ltd, 2017

Working with Futures

- Future[T] is a monadic type
- Higher order functions available

- map, filter, flatMap, ...
- Build pipelines of processing on Futures
- Allow sophisticated concurrency behaviour to be specified

```
def strategy ( price: Double ): String = {
  val profit = price - 400.0
  if ( profit > 100.0 ) "sell" else "hold"
}
val action = PricingService.priceForSym( "GOOG" )
  .map ( strategy(_) )

action foreach {
  println(a => println(s"Action for GOOG: $a"))
}
```

Action for GOOG: sell

© J&G Services Ltd, 2017

Working with Futures

- Future[T] is a monadic type

- Higher order functions available

- map, filter, flatMap, ...
- Build pipelines of processing on Futures

```
def asStrategy ( price: Double ): Future[String] = {
  Future {
    val profit = price - 100.0
    if ( profit > 100.0 ) "sell" else "hold"
  }
}
val action = PricingService.priceForSym("AAPL")
  .flatMap ( asStrategy(_) )

pf foreach {
  println(a => println(s"Action for GOOG: $a") )
}
```

Action for GOOG: sell

© J&G Services Ltd, 2017

Working with Futures

- Use with for comprehension

- Based on translation to flatMap/map

```
scala> val action = for (
  |   price <- PricingService.priceForSym("GOOG");
  |   act <- asStrategy(price)
  | ) yield act

action foreach {
  println(a => println(s"Action for GOOG: $a") )
}
```

Action for GOOG: sell

© J&G Services Ltd, 2017

Working with Futures

- Future[T] may not complete successfully
- Use onFailure callback to deal with failure case
 - or onComplete

```
...
val action = PricingService.priceForSym("XXX").flatMap( asStrategy(_) )

action onComplete {
  case Success(action: String) =>
    println(s"Success: $action")
  case Failure(ex) =>
    println(s"Failed: $ex")
}
```

Failed: java.util.NoSuchElementException: key not found: XXX

© J&G Services Ltd, 2017

A Word About for Comprehension

- Remember that for comprehension translates to calls to flatMap
- Based on sequencing of computation
- Even if computations are asynchronous, they will be sequenced

```
val answer = for (
  a <- Future { Thread.sleep(2000); scala.util.Random.nextInt(100) };
  b <- Future { Thread.sleep(1000); scala.util.Random.nextInt(100) };
  c <- Future { Thread.sleep(3000); scala.util.Random.nextInt(100) }
) yield ( a + b + c )

answer foreach ( println(_) )
141
```

© J&G Services Ltd, 2017

A Word About for Comprehension

- If the steps do not depend on each other then they should be executed concurrently
- Start Futures before the for comprehension

```
val first = Future { Thread.sleep(5000); scala.util.Random.nextInt(100) }
val second = Future { Thread.sleep(10000); scala.util.Random.nextInt(100) }
val third = Future { Thread.sleep(4000); scala.util.Random.nextInt(100) }

val sum = for (
    a <- first;
    b <- second;
    c <- third
) yield (a + b + c)

sum foreach (println(_))
143
```

© J&G Services Ltd, 2017

Filter and Future[T]

- filter works as for other container types
 - If predicate is true, Future completes successfully
 - If predicate is false, Future fails with NoSuchElementException

```
scala> val f = Future { r.nextInt(100) } filter(_ % 2 == 0)
scala> f onComplete {
|   case Success(i)  => println(s"$i")
|   case Failure(e)  => println(s"$e")
| }
```

76

```
scala> val f = Future { r.nextInt(100) } filter(_ % 2 == 0)
scala> f onComplete {
|   case Success(i)  => println(s"$i")
|   case Failure(e)  => println(s"$e")
| }
```

java.util.NoSuchElementException: Future.filter predicate is not satisfied

© J&G Services Ltd, 2017

Zip and Future[T]

- zip combines two containers (collections) into a single container of tuples
 - Combine two Futures into a single Future of tuple
 - Completes when both Futures have completed

```
scala> val a = Future { Thread.sleep(10000); scala.util.Random.nextInt(100) }
a: scala.concurrent.Future[Int] = List()

scala> val combined = a zip Future { Thread.sleep(4000); 100 }
combined: scala.concurrent.Future[(Int, Nothing)] = List()

scala> combined onComplete {
  |   case Success(t) => println(t)
  |   case Failure(e) => println(e)
  | }
(65,100)
```

© J&G Services Ltd, 2017

Zip and Future[T]

- zip combines two containers (collections) into a single container of tuples
 - If one of the Futures does not complete successfully, the combined Future fails with the exception

```
scala> val a = Future { Thread.sleep(10000); scala.util.Random.nextInt(100) }
a: scala.concurrent.Future[Int] = List()

scala> val combined = a zip Future { Thread.sleep(4000);
  |   throw new RuntimeException("oops") }
combined: scala.concurrent.Future[(Int, Nothing)] = List()

scala> combined onComplete {
  |   case Success(t) => println(t)
  |   case Failure(e) => println(e)
  | }
java.lang.RuntimeException: oops
```

© J&G Services Ltd, 2017

Sequence and Future[T]

- sequence is a common monad transformer

List [M[_]] => M [List[_]]

- Transform List of Monads into Monad of List

```
val a = Future { Thread.sleep(5000); scala.util.Random.nextInt(100) }
val b = Future { Thread.sleep(10000); scala.util.Random.nextInt(100) }
val c = Future { Thread.sleep(4000); scala.util.Random.nextInt(100) }

scala> val lf = List ( a, b, c )
lf: List[scala.concurrent.Future[Int]] = List(List(), List(), List())

scala> val fl = Future.sequence(lf)
fl: scala.concurrent.Future[List[Int]] = List()

scala> fl foreach ( println(_) )
List(28, 67, 24)
```

© J&G Services Ltd, 2017

Traverse and Future[T]

- Traverse is another common monad transformer
- Transform a List of A into a Monad of List of B
 - Applying function A => M[B] to each element

```
scala> val timeList = List( 5, 9, 4 )
timeList: List[Int] = List(5, 9, 4)

scala> val fl = Future.traverse(timeList)
        ( t => Future { Thread.sleep( t * 1000 );
                      scala.util.Random.nextInt(100) } )
fl: scala.concurrent.Future[List[Int]] = List()

scala> fl foreach ( println(_) )
List(23, 64, 75)
```

© J&G Services Ltd, 2017

Working with a Sequence of Future[T]

- `Future.firstCompletedOf(List[Future[_]])`
 - Return a "Future" that completes when the first Future in the list completes

```
scala> val l = List( Future{Thread.sleep(7000); 1},
    Future{Thread.sleep(6000); 2}, ← Should complete first
    Future{Thread.sleep(10000); 3} )
l: List[scala.concurrent.Future[Int]] = List(List(), List(), List())

scala> Future.firstCompletedOf(l) onComplete {
|   case Success(i) => println(i)
|   case Failure(e) => println(e)
| }
2
```

© J&G Services Ltd, 2017

Future[T] Timeout Without Blocking

- `Future.firstCompletedOf` can be used to implement a basic timeout capability for Future
 - Without using the blocking `Await...` calls

```
scala> val lf = List ( Future{ Thread.sleep(5000); throw timeOutEx },
|           Future{ Thread.sleep(7000); "Success"} )
lf: List[scala.concurrent.Future[String]] = List(List(), List())

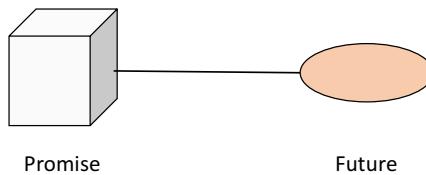
scala> val ff = Future.firstCompletedOf(lf)
ff: scala.concurrent.Future[String] = List()

scala> ff onComplete {
|   case Success(i) => println(i)
|   case Failure(e) => println(e)
| }
java.lang.RuntimeException: Out of time
```

© J&G Services Ltd, 2017

Promises

- Future is a read-only value
 - Value written from another context
- Promise used to represent write side
 - Connected to a Future
 - Strictly "write once"



© J&G Services Ltd, 2017

Promises

```
...
val vow = promise[Int]

val p1 = vow.future
val p2 = vow.future    // Returns same Future as for p1

p1 onComplete {
  case Success(v) => println(s"p1 got $v")
  case Failure(ex) => println(s"p1 failed with $ex")
}

p2 onComplete {
  case Success(v) => println(s"p2 got $v")
  case Failure(ex) => println(s"p2 failed with $ex")
}

Thread.sleep(1000)
vow success 42
// vow failure new ArithmeticException
```

p1 got 42
p2 got 42

© J&G Services Ltd, 2017

Alternative Approach to Timeout

- Previous example relied on async operation that included Thread.sleep()
 - Not recommended approach
 - Blocks thread, prevents execution context scheduling other work
- Alternative approach can use explicit Promise completion
 - Define a single Promise and obtain a future from this Promise
 - Utilise "external" scheduler to fire a timeout event, which completes the with a Failure (a timeout exception)
 - Arrange for successful completion when the main task completes successfully
 - This will be completely non-blocking

© J&G Services Ltd, 2017

Alternative Approach to Timeout

- Code for timeout via Promise completion

```
val timeoutScheduler = Executors.newScheduledThreadPool(1)
val p = Promise[String]()
val rf = p.future
val timeoutAction = new Runnable {
    def run = p tryFailure timeoutException
}

p tryCompleteWith Future { longRunningTask }
timeoutScheduler.schedule( timeoutAction, 8000, TimeUnit.MILLISECONDS )

rf onComplete {
    case Success(s) => println(s)
    case Failure(e) => println(s"Error: $e")
}

val longRunningTask = { Thread.sleep(5000); "Finished" }
```

Finished

© J&G Services Ltd, 2017