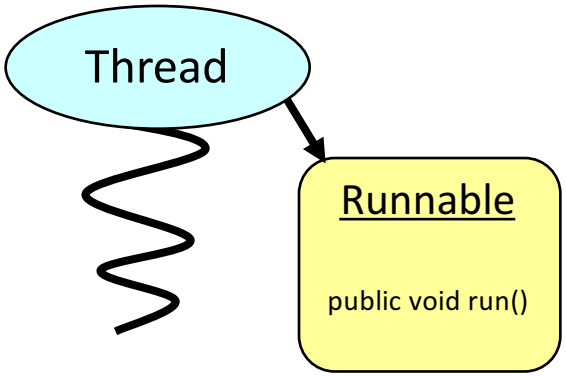
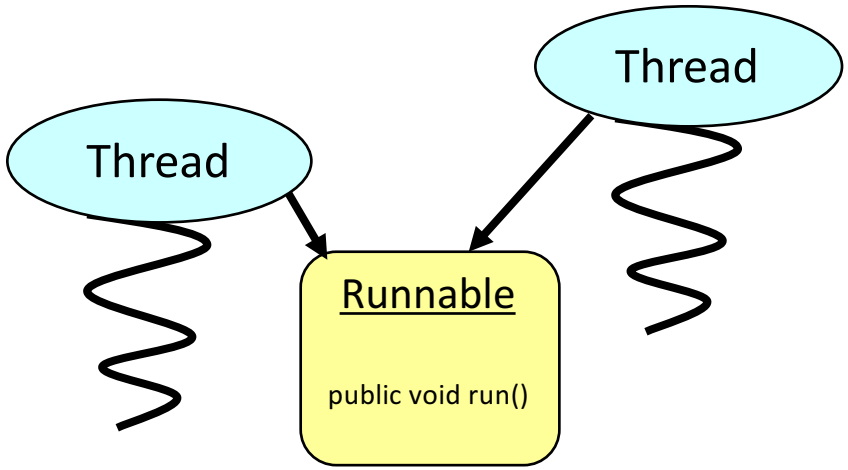
JVM Concurrency and Scala

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 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
        }
      }
    }
  }
}
```

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
```

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
```

Problems

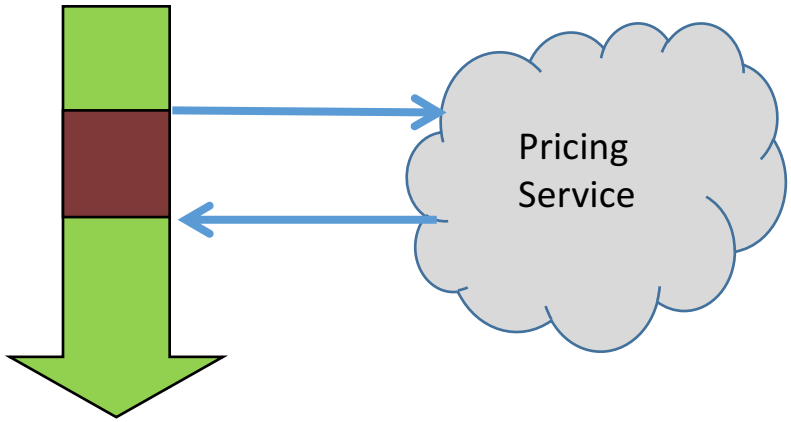
- Limited capabilities
  - `run` method returns `Unit`
  - often want task to return a value
- Scalability is limited
  - threads relatively heavyweight objects
  - limited number can be supported in VM
- Shared state difficult to protect
  - locks/synchronized blocks
  - introduces complexity to code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

Asynchronous Programming

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

- Higher level of abstraction

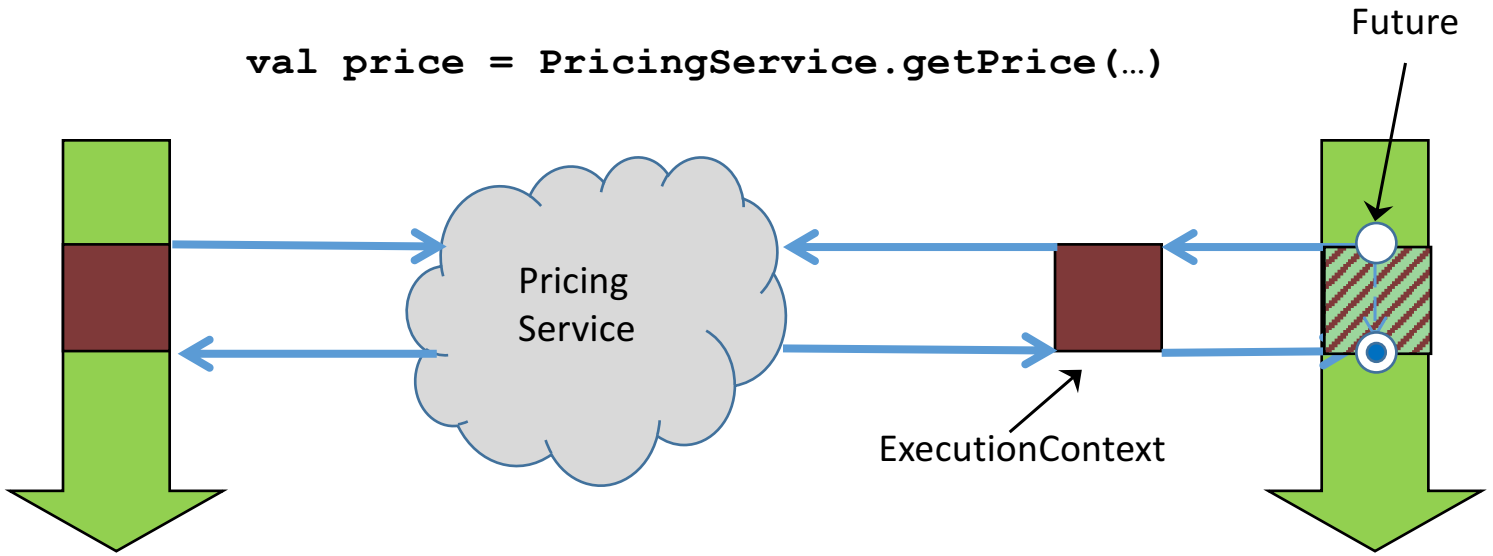
```
val price = PricingService.getPrice(…)
```



Asynchronous Programming

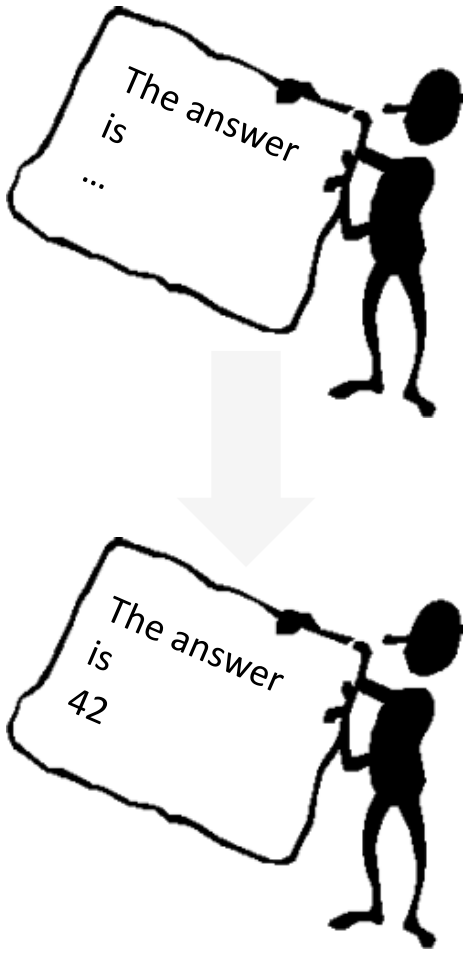
- A different approach
  - "do something" while calling thread continues
  - hand back a result when finished

- Higher level of abstraction



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
  - …



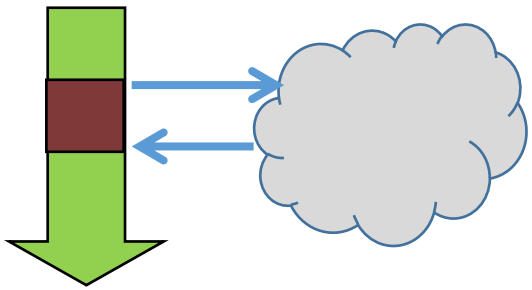
Futures

- Example

```
object PricingService {
  val assets = Map("AAPL" -> 500.10,
                   "GOOG" -> 700.0,
                   "MS" -> 34.50    )
  def getPrice (asset: String ): Double = {
    Thread.sleep(1000)                          // Simulated delay
    assets(asset)
  }
}
```

- Normal call

```
 val p = PricingService.getPrice("AAPL")
 println(f"Price for AAPL: $p%.4f")
```

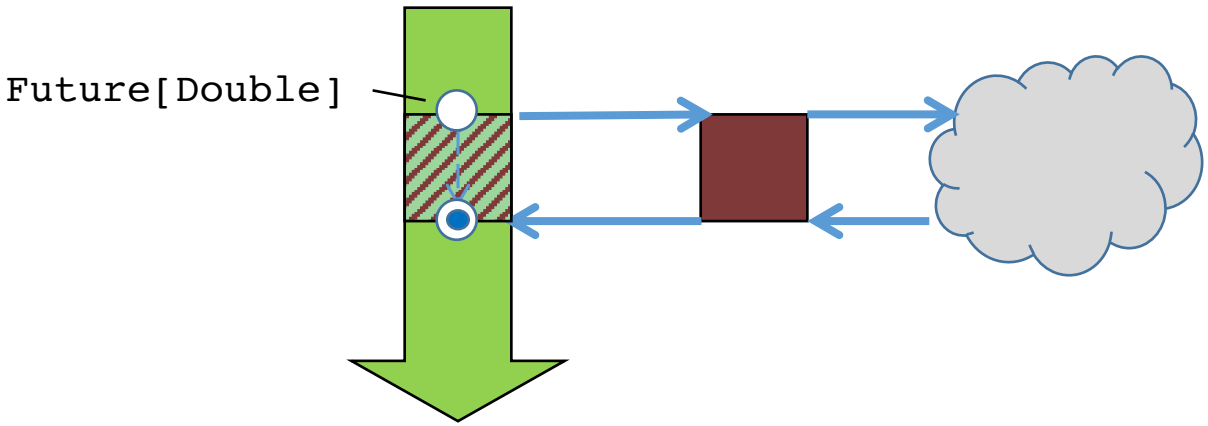


Futures

- Using a Future

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

val pf = future {
  PricingService.getPrice("AAPL")
}
```



Futures – Completion

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

```
val pf = future { PricingService.getPrice(…) }

…

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

pf onFailure {
  case ex: Throwable => println(s"There was a problem: $ex")
}
```

```
The price is 500.1000
```

```
There was a problem: java.util.NoSuchElementException: key not found: MSFT
```

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")
}
…
```

Working with Futures

- `Future[T]` is a container type
- Higher order functions available
  - `map`, `filter`, `flatMap`, …
  - build pipelines of processing on `Futures`

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

action onSuccess {
  case p: String => println(s"$p")
}
```

sell

Working with Futures

- `Future[T]` is a container 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 = future {
               PricingService.getPrice("AAPL")
             } flatMap {
               asStrategy(_)
             }

pf onSuccess {
  case p: String => println(s"$p")
}
```

sell

Working with Futures

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

```
…
val action = future {
              PricingService.getPrice("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
```

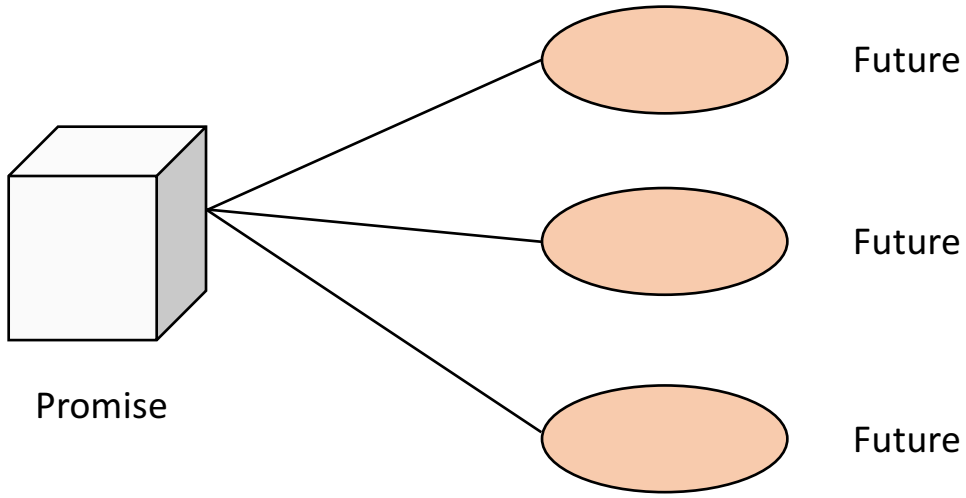
Working with Futures

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

Promises

- Future is a read-only value
  - value written from another context
- Promise used to represent write side
  - one `Promise` can be "watched" by many `Futures`
  - strictly "write once"



Promises

```
…
val vow = promise[Int]

val p1 = vow.future
val p2 = vow.future

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
```