"Do not block threads!"

a blessing in disguise or a curse?

@sadache

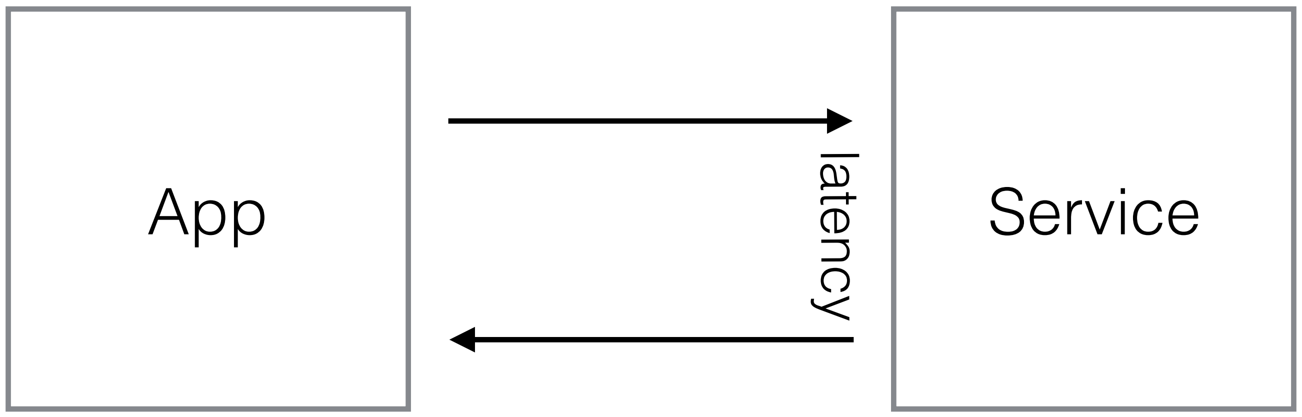
prismic.io co-founder, Play framework co-creator

Modern Applications

- Spend a considerable time talking to internet
- Internet means latency
- How does your runtime integrate this latency?

MyFunnyPics.net

A Typical Request



We should not waste scarce resources

while waiting for work to be done on other machines

- Memory, CPU, …
- Threads/processes?
  - lightweight (millions on a single machines)
  - heavyweight? …

© Loeden Media

BUD LIGHT

JVM and co

- Threads are scarce resources (or are they?)
- We should not hold to threads while doing IO (or internet call)
- “Do not block threads!”

POLICE NATIONALE
SECURITE PUBLIQUE
POLICE

Copy that! what CAN I do?

Do not block threads!

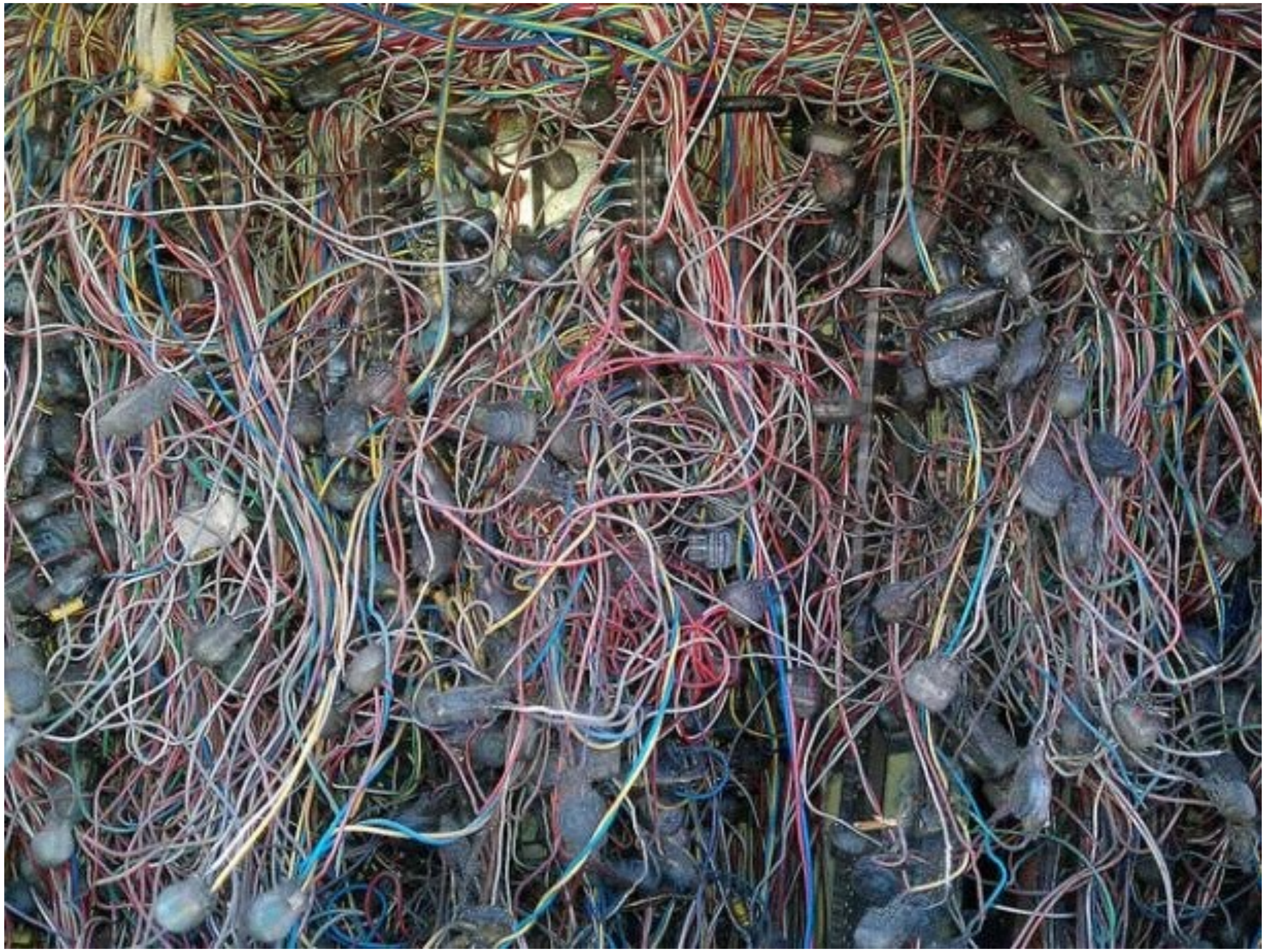
- Then what should I do?

- Non blocking IO and Callbacks

-
  ```
  ws.get(url, { result =>
    println(result)
  })
  ```

- What happens if I want to do another call after?

- Callback hell!



Futures!
(Tasks, Promises, …)

- Future[T] represents a result of type T that we are eventually going to get (at the completion of the Future)

- Doesn’t block the thread

- But how can I get the T inside?

- // blocking the current thread until completion of the future?
  Result.await(future)

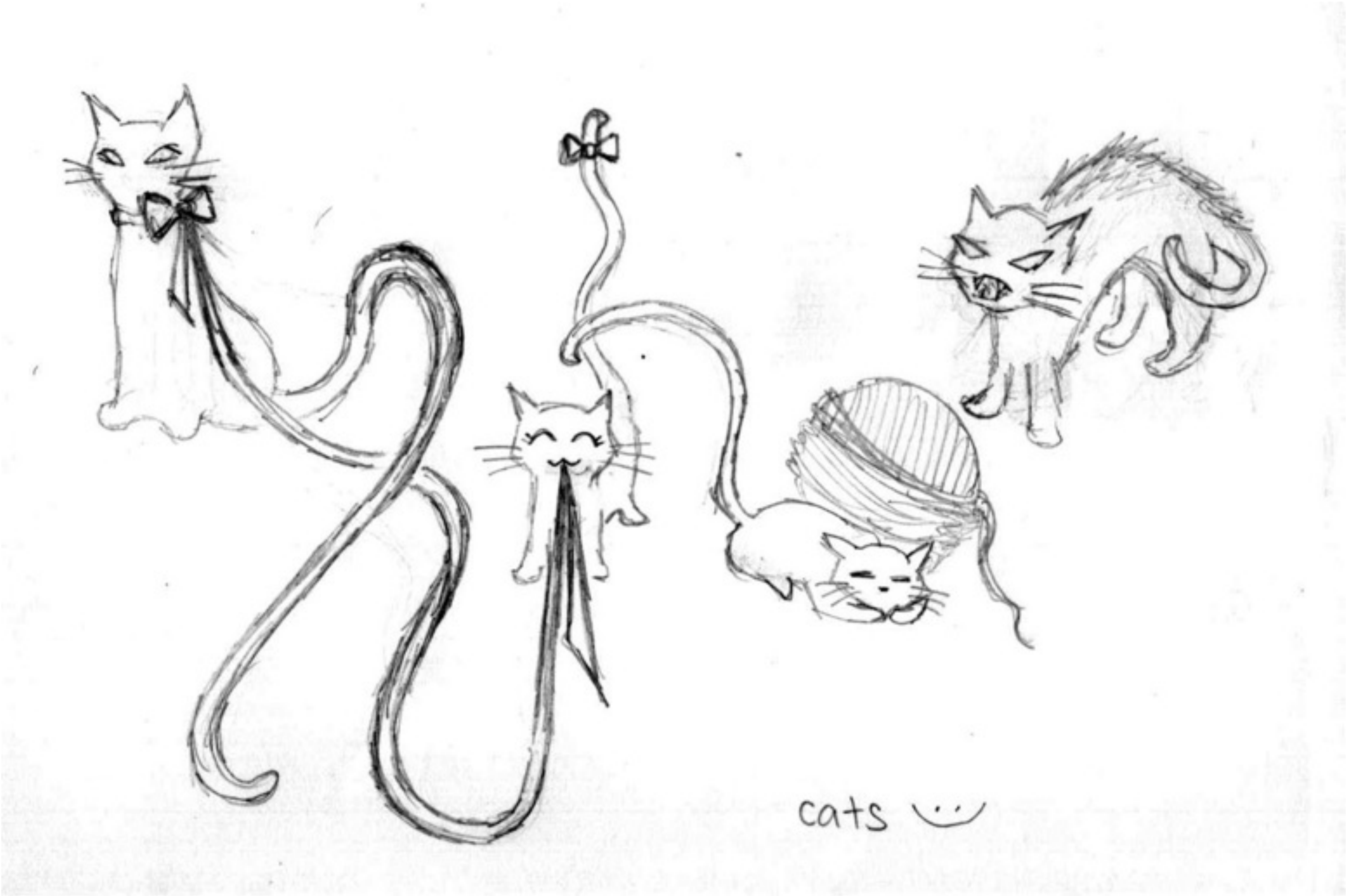
Examples of Future composition

```
val eventuallyTweet: Future[String] = …

val et: Future[Tweet] = eventuallyTweet.map(t => praseTweet(t))



val tweets: Seq[Future[Tweet]] = …

val ts: Future[Seq[Tweet]] = Future.sequence(tweets)
```

Future composition

Future composition

Future composition



Some syntax sugar

```
// for comprehensions

for {

  t <- getTweet(id)

  k <- getKloutScore(t.user)

} yield (t,k)
```

Futures are elegant

- all, any, monads, applicatives, functors
- do all the scheduling and synchronisation behind the scenes

Future is not satisfactory

Futures are not completely satisfactory

- Manage execution on completion (who is responsible of executing the code?)

- Additional logic complexity (adding one level of indirection)

- Has a big impact on your program (refactorings)

- Ceremony, or am I doing the compiler/runtime work?

- Stacktrace gone!

Who runs this code?

```
val eventuallyTweet: Future[String] = …

val et: Future[Tweet] = eventuallyTweet.map(t => praseTweet(t))
```

Futures are not completely satisfactory

- Manage execution on completion (who is responsible of executing the code?)
- Additional logic complexity (adding one level of indirection)
- Has a big impact on your program (refactorings)
- Ceremony, or am I doing the compiler/runtime work?
- Stacktrace gone!

Scala’s solution to execution management (on completion)

- Execution Context
- def map[S](f: (T) ⇒ S)(**implicit executor: ExecutionContext**): Future[S]
- Just import the appropriate EC
- Very tough to answer the question (developers tend to chose the default EC, can lead to contentions)
- import scala.concurrent.ExecutionContext.global
- Contention?

Futures are poor man's lightweight threads

- You might be stuck with them if you're stuck with heavyweight threads…
- **Scala async**
- Why not an async for the whole program?

Futures are poor man’s lightweight threads

```
val future = async {
  val f1 = async { ...; true }
  val f2 = async { ...; 42 }
  if (await(f1)) await(f2) else 0
}
```

Futures are poor man’s lightweight threads

- You might be stuck with them if you’re stuck with heavyweight threads…
- Scala async
- Why not an async for the whole program?

Inversion of control (Reactive)

- Future but for multiple values (streams)
- Just give us a Function and we call you each time there is something to do
- Mouse.onClick { event => println(event) }

Inversion of control (Reactive)

- What about maintaining state across calls
- Composability and tools
- Iteratees, RX, Streams, Pipes, Conduits, … etc

Iteratees

<a quick introduction>

Iteratees

- What about maintaining state between calls
- Composability and tools
- Iteratees, RX, Streams, Pipes, Conduits, … etc

Iteratees

```
trait Step

case class Cont( f:E => Step) extends Step

case class Done extends Step
```

Iteratees

```
trait Step[E,R]

case class Cont[E,R]( f:E => Step[E,R]) extends Step[E,R]

case class Done(r: R) extends Step[Nothing, R]
```

Iteratees

```
// A simple, manually written, Iteratee
val step = Cont[Int, Int]( e => Done(e))
//feeding 1
step match {
  case Cont(callback) => callback(1)
  case Done(r) => // shouldn’t happen
}
```

Counting characters

```
// An Iteratee that counts characters

def charCounter(count:Int = 0): Step[String, Int] = Cont[String, Int]{

  case Chunk(e) => charCounter(count + e.length)

  case EOF => Done(count)

}
```

Iteratees

```
trait Input[E]

case class Chunk[E](e: E)

case object EOF extends Input[Nothing]



trait Step[E,R]

case class Cont[E,R]( f:E => Step[E,R]) extends Step[E,R]

case class Done(r: R) extends Step[Nothing, R]
```

Counting characters

Counting characters

```
// An Iteratee that counts characters

def charCounter(count:Int = 0): Step[String, Int] = Cont[String, Int]{

  case Chunk(e) => step(count + e.length)

  case EOF => Done(count)

}
```

Same principle

- count, getChunks, println, sum, max, min, etc
- progressive stream fold (fancy fold)
- Iteratee is the reactive stream consumer

Enumerators

- Enumerator[E] is the source, it iteratively checks on the Step state and feeds input of E if necessary (Cont state)
- Enumerators can generate, or retrieve, elements from anything
- Files, sockets, lists, queues, NIO
- Helper constructors to build different Enumerators

Enumeratees

- Adapters
- Apply to Iteratees and/or Enumerators to adapt their input
- Create new behaviour
- map, filter, buffer, drop, group, … etc

Iteratees

</ a quick introduction>

Iteratees

Inversion of controls: Enumerators chose when to call the Iteratees continuation

They chose on which Thread to run continuation

What if an Iteratee (or Enumeratee) decided to do a network call?

Block the thread waiting for a response?

Counting characters

```
// An Iteratee that counts characters

def sumScores(count:Int = 0): Step[String, Int] = Cont[String, Int]{

  case Chunk(e) =>

    val eventuallyScore: Future[Int] = webcalls.getScore(e)

    step(count + Result.await(eventuallyScore)) // seriously???

  case EOF => Done(count)

}
```

Reactive all the way

```
// An Iteratee that counts characters

def sumScores(count:Int = 0): Step[String, Int] = Cont[String, Int]{

  case Chunk(e) =>

    val eventuallyScore: Future[Int] = webcalls.getScore(e)

    step(count + Result.await(eventuallyScore)) // seriously???

  case EOF => Done(count)

}
```

Iteratees

```
trait Step[E,R]

case class Cont[E,R]( f:E => Step[E,R]) extends Step[E,R]

case class Done(r: R) extends Step[Nothing, R]
```

Iteratees

```
trait Step[E,R]

case class Cont[E,R]( f:E => Future[Step[E,R]]) extends Step[E,R]

case class Done(r: R) extends Step[Nothing, R]
```

Reactive all the way

```
// An Iteratee that counts characters
def sumScores(count:Int = 0): Step[String, Int] = Cont[String, Int]{
  case Chunk(e) =>
    val eventuallyScore: Future[Int] = webcalls.getScore(e)
    eventuallyScore.map( s => step(count + s))
  case EOF => Future.successful(Done(count))
}
```

Seamless integration between Futures and Iteratees

Seq[Future[E]] is an Enumerator[E]

Iteratees can integrate any Future returning call

Back-pressure for free

Suffer from the same drawbacks of Futures

- Manage execution on completion (who is responsible of executing the code?)
- Everything becomes a Future
- Stacktrace gone!

Elegant, help manage complexity of asynchronous multiple messages

Composable

Builders and helpers

Modular

Recap

- Stuck with heavyweight threads?
- NIO and Callback hell
- Futures
- Composable Futures
- Iteratees and co
- Developer suffering from what the runtime/compiler couldn’t provide

Asynchronous Programming

is the price you pay, know what you’re paying for

The price is your productivity

Asynchronous Programming

calculate your cost effectiveness

Questions