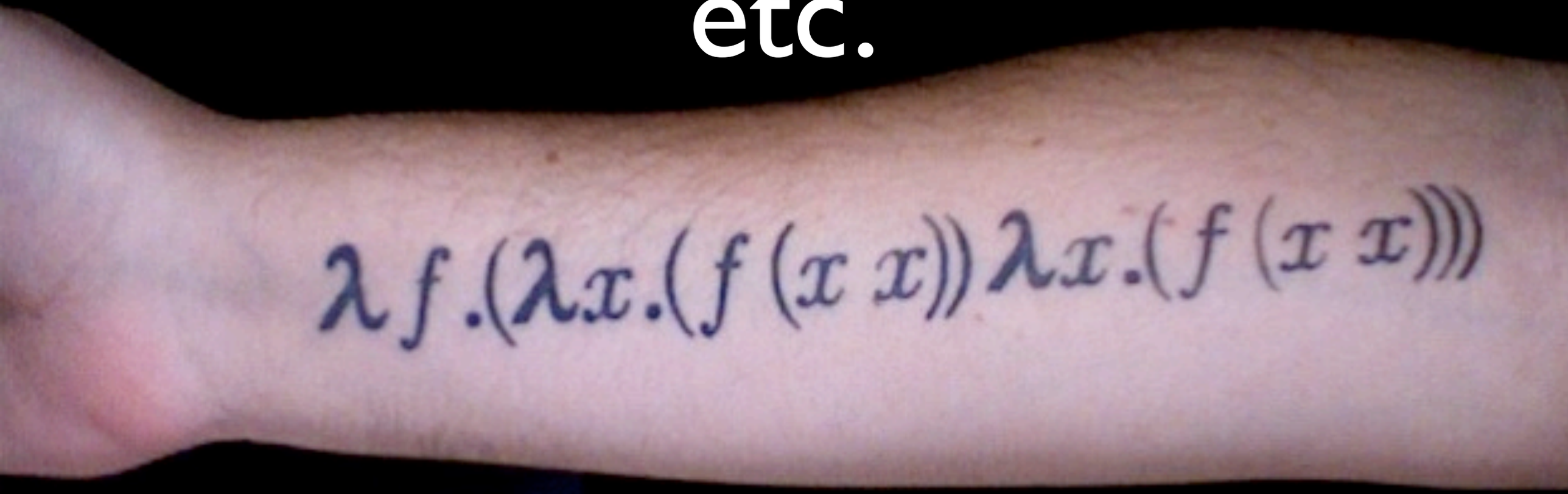
Typeclasses, Monads, etc.



Functional programming in Scala can be simple!

“The plan”

- **Implicit parameters** - readability

“The plan”

- **Implicit parameters** - readability
- **Typeclasses** - code reuse

“The plan”

- **Implicit parameters** - readability
- **Typeclasses** - code reuse
- **Monads** - composition

Implicit Parameters Demo

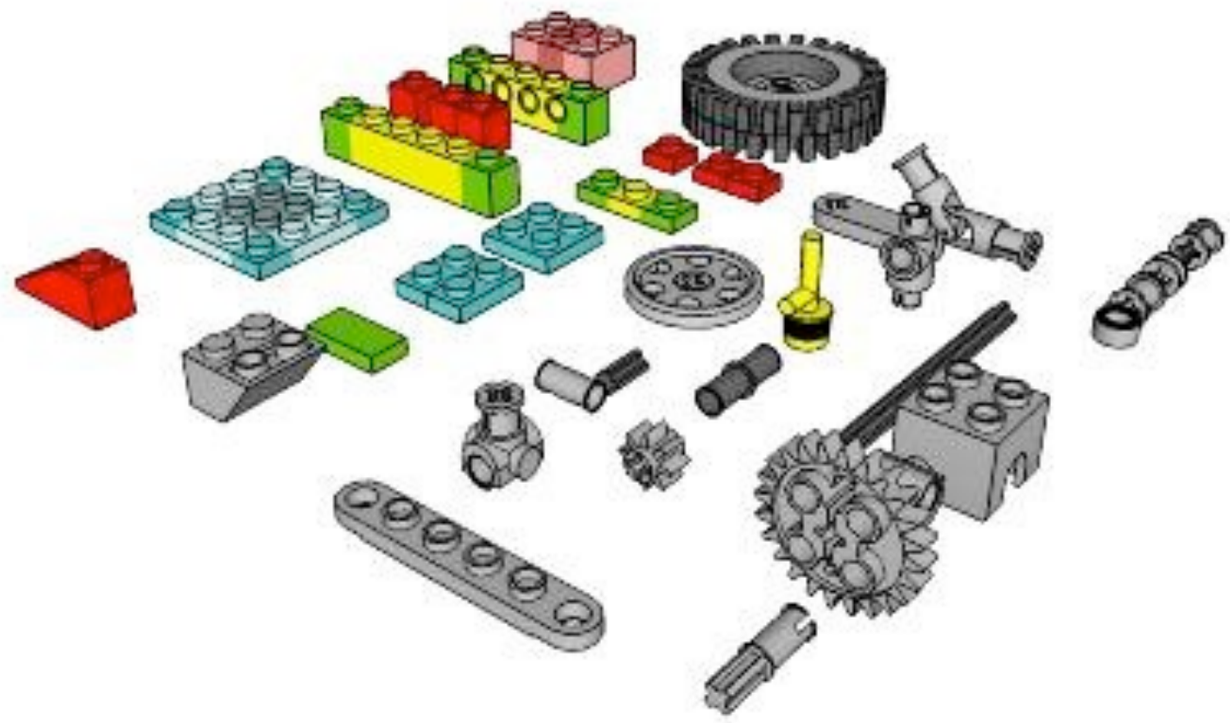
- Keyspace =~ db schema
- Column family =~ db table

Column family: Users

1	Name	Age	City
	Pete	28	San Jose

2	Name	Age	City
	Sue	25	Las Vegas

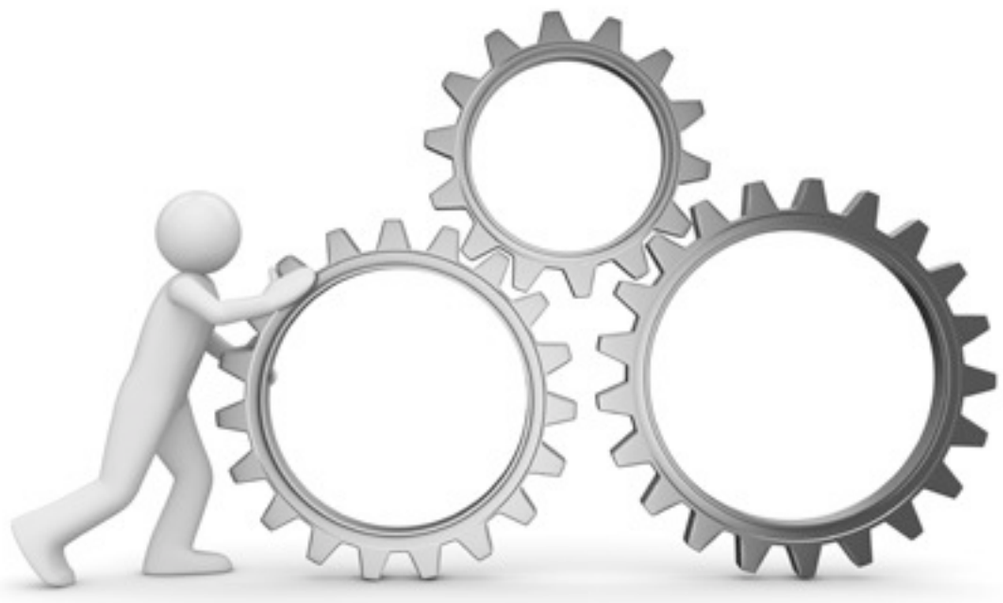
Operations on rows

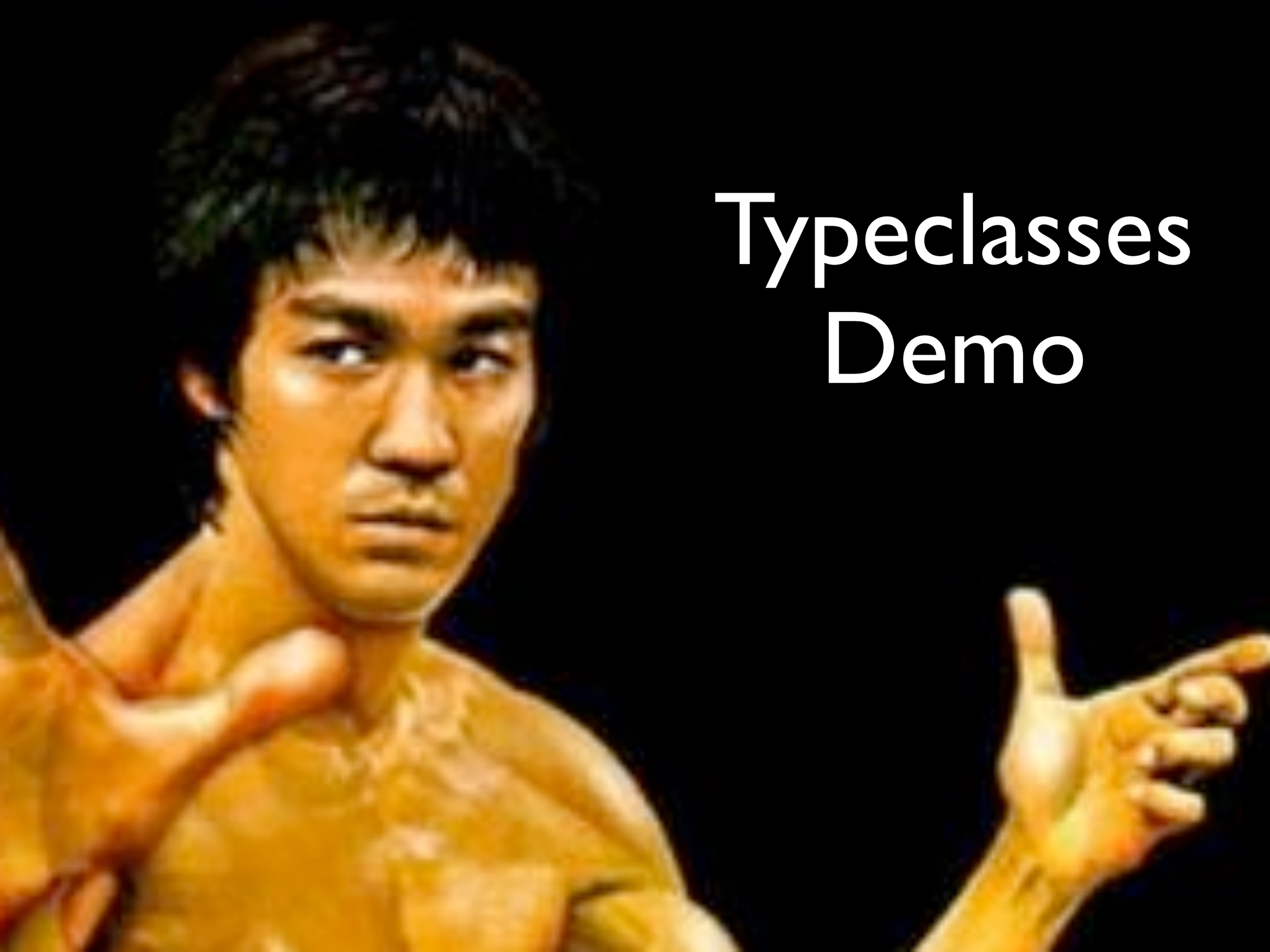


```
Put(PEOPLE, rowId,
  "name" -> "Jon Doe",
  "email" -> "jon@doe.com"
  "age" -> "29" )

val jon = Get(PEOPLE, rowId, "name", "age")

assert( jon("age") == "29" )

Remove(PEOPLE, rowId, "age")
RemoveRow(PEOPLE, rowId)

assert ( Get(PEOPLE, rowId, "name", "email") isEmpty )
```

Operations on objects

```
case class Person(id: String, name: String, email: String, age: Int)

Save(Person("joe-123", "Joe Doe", "joe@doe.com", 29))

val joe = Read[Person]("joe-123")
  .getOrElse(throw new RuntimeExcept:

Save(joe copy(name = "Joe Smith"))

Delete(joe)

DeleteById[Person](joe.id)
```





Typeclasses Demo

What is type class?

Type class == Group of types

What is type class?

Type class == Group of types

```
trait CassandraObject[T] {
  def columnFamily: String
  def columns: Seq[String]
  def rowId(o: T): String
  def marshall(o: T): List[(String, String)]
  def unmarshall(f: Map[String,String]): Option[T]
}
```

What is type class?

Type class == Group of types

```
trait CassandraObject[T] {
  def columnFamily: String
  def columns: Seq[String]
  def rowId(o: T): String
  def marshall(o: T): List[(String, String)]
  def unmarshall(f: Map[String,String]): Option[T]
}
```

T belongs to CassandraObject typeclass
if there is class X
such that X extends CassandraObject[T]

Type class instance

```scala
implicit object PersonCO extends CassandraObject[Person]{
  def marshall(p: Person): List[(String, String)] = List(
    "id" -> p.id,
    "name" -> p.name,
    "email" -> p.email,
    "age" -> p.age.toString
  )

  def rowId(p: Person): String = p.id
  def columnFamily: String = "people"
}
```

Serialization

```
def writeToFile[T](filename: String, obj: T)
                  (implicit serializer: Serializable[T]) {
  val out = new FileOutputStream(filename)
  try{
    serializer.write(obj, out)
  } finally {
    out.close()
  }
}

writeToFile("demo.txt", Address("11A", "South Colonnade", "E14 4BY"))
```

Typecalass

```
trait Serializable[T] {
  def write(obj: T, out: OutputStream)
}
```

```
def writeToFile[T](filename: String, obj: T)
                  (implicit serializer: Serializable[T]) {
  val out = new FileOutputStream(filename)
  try{
    serializer.write(obj, out)
  } finally {
    out.close()
  }
}

writeToFile("demo.txt", Address("11A", "South Colonnade", "E14 4BY"))
```

Typeclass Instance

```
trait Serializable[T] {
  def write(obj: T, out: OutputStream)
}


object JsonProtocol{
  implicit object AddressJsonSerializer extends Serializable[Address]{
    def write(obj: Address, out: OutputStream) {
      out.write( """{
          'houseNumber' : '%s',
          'street' : '%s',
          'postCode' : '%s'
          }""".format(obj.houseNumber, obj.postCode, obj.street)
        .getBytes("UTF-8"))
    }
  }
}
```

Multiple Instances

```
import JsonProtocol._
writeToFile("demo.txt", Address("11A", "South Colonnade", "E14 4BY"))
```

```
import XmlProtocol._
writeToFile("demo.txt", Address("11A", "South Colonnade", "E14 4BY"))
```

Numeric

```
trait Numeric[T] extends Ordering[T] {
  def plus(x: T, y: T): T
  def minus(x: T, y: T): T
  def times(x: T, y: T): T
  def negate(x: T): T
  def fromInt(x: Int): T
  def toInt(x: T): Int
  def toLong(x: T): Long
  def toFloat(x: T): Float
  def toDouble(x: T): Double

  def zero = fromInt(0)
  def one = fromInt(1)

  def abs(x: T): T = if (lt(x, zero)) negate(x) else x
  def signum(x: T): Int =
    ...
```

Matrix multiplication

```
def multiply[A](vector1 : List[A], vector2 : List[A])
               (implicit num: Numeric[A]): A = (
for {
  e1 <- vector1
  e2 <- vector2
} yield num.times(e1,e2)
).sum
```

Matrix multiplication

```scala
def multiply[A](vector1 : List[A], vector2 : List[A])
               (implicit num: Numeric[A]): A = (
for {
  e1 <- vector1
  e2 <- vector2
} yield num.times(e1,e2)
).sum
```

```
scala> multiply ( List(1L, 2L), List(3L, 5L) )
res0: Long = 24

scala> multiply ( List(1.2, 2.3), List(3.4, 5.3) )
res1: Double = 30.449999999999996
```

Typeclasses summary

```
trait CassandraObject[T] {
  def columnFamily: String
  def columns: Seq[String]
  def rowId(o: T): String
  def marshall(o: T): List[(String, String)]
  def unmarshall(f: Map[String,String]): Option[T]
}
```

Typeclasses summary

```
trait CassandraObject[T] {
  def columnFamily: String
  def columns: Seq[String]
  def rowId(o: T): String
  def marshall(o: T): List[(String, String)]
  def unmarshall(f: Map[String,String]): Option[T]
}

implicit object PersonCO extends CassandraObject[Person] {
  def columnFamily = "people"

  /*...*/
```

Typeclasses summary

```
trait CassandraObject[T] {
  def columnFamily: String
  def columns: Seq[String]
  def rowId(o: T): String
  def marshall(o: T): List[(String, String)]
  def unmarshall(f: Map[String,String]): Option[T]
}

implicit object PersonCO extends CassandraObject[Person] {
  def columnFamily = "people"

  /*...*/

 def Save[T](p: T)(implicit keyspace: Keyspace, pco: CassandraObject[T]) {
   Put(pco.columnFamily, pco.rowId(p), pco.marshall(p): _*)
 }

 Save(Person("1234", "Adam", "a@b.com", 34))
```

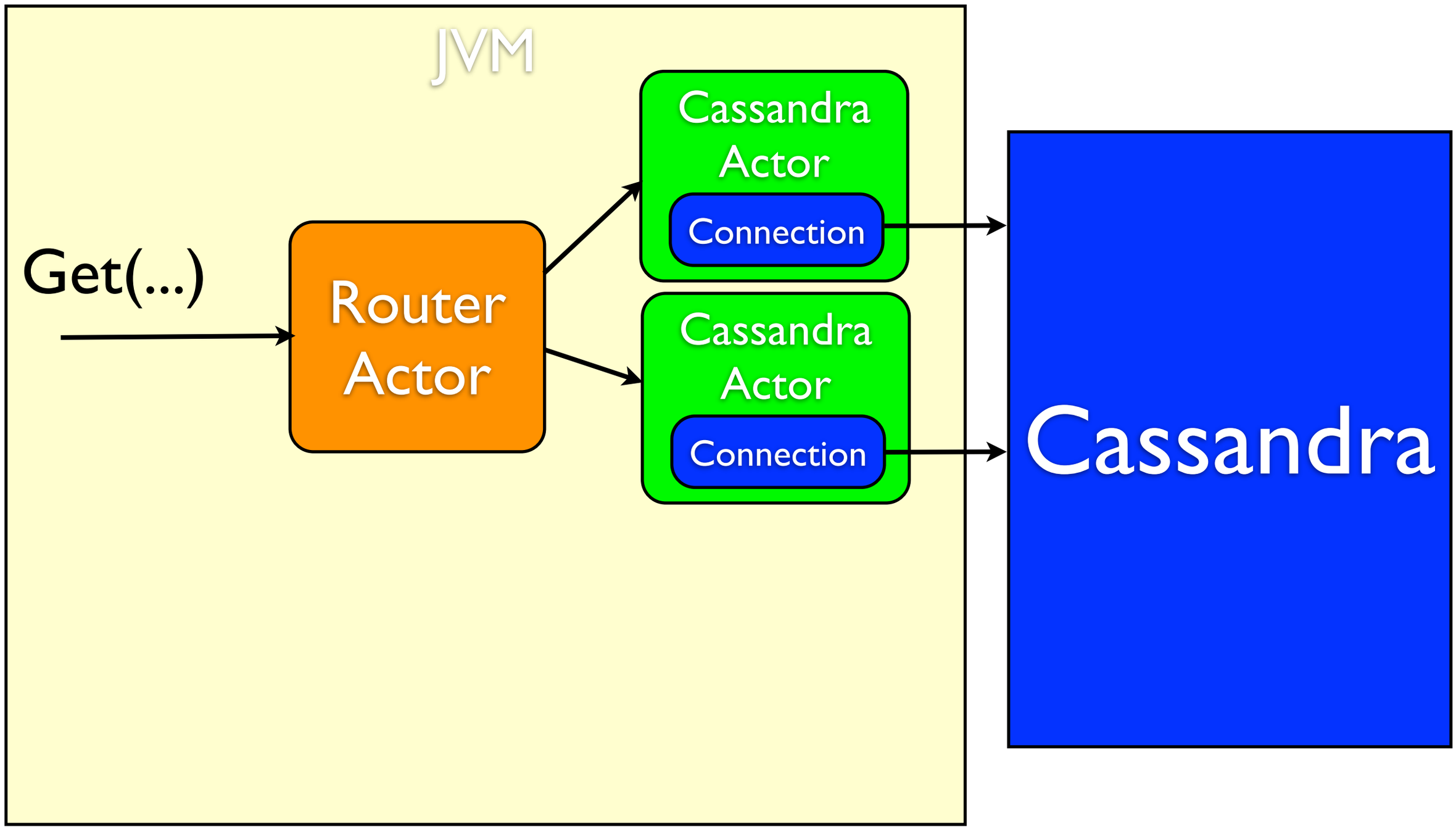
Typeclasses summary

```
trait CassandraObject[T] {
  def columnFamily: String
  def columns: Seq[String]
  def rowId(o: T): String
  def marshall(o: T): List[(String, String)]
  def unmarshall(f: Map[String,String]): Option[T]
}

implicit object PersonCO extends CassandraObject[Person] {
  def columnFamily = "people"

  /*...*/

 def Save[T](p: T)(implicit keyspace: Keyspace, pco: CassandraObject[T]) {
   Put(pco.columnFamily, pco.rowId(p), pco.marshall(p): _*)
 }

 Save(Person("1234", "Adam", "a@b.com", 34))
```

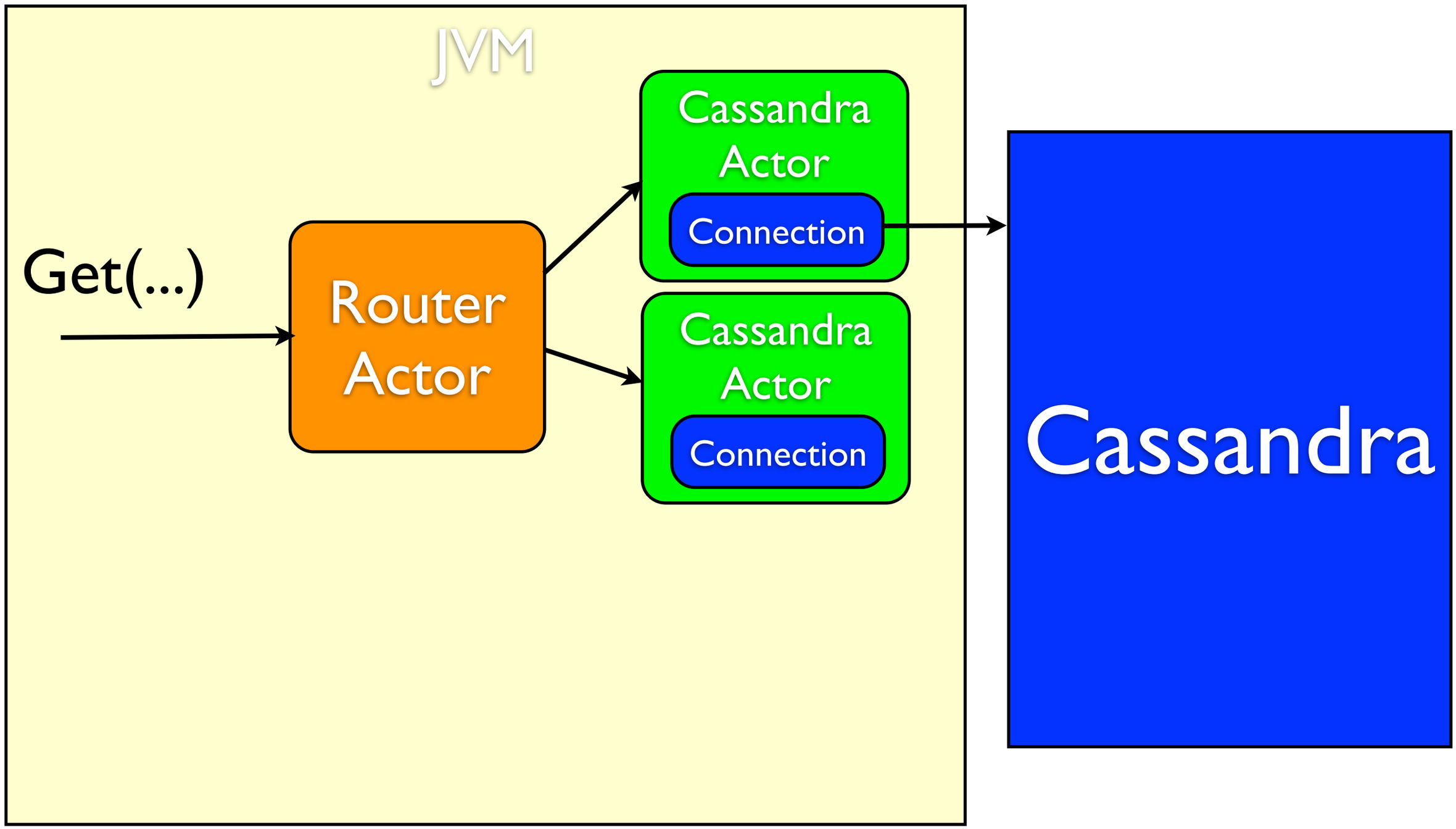
Serializable[T]

Numeric[T]

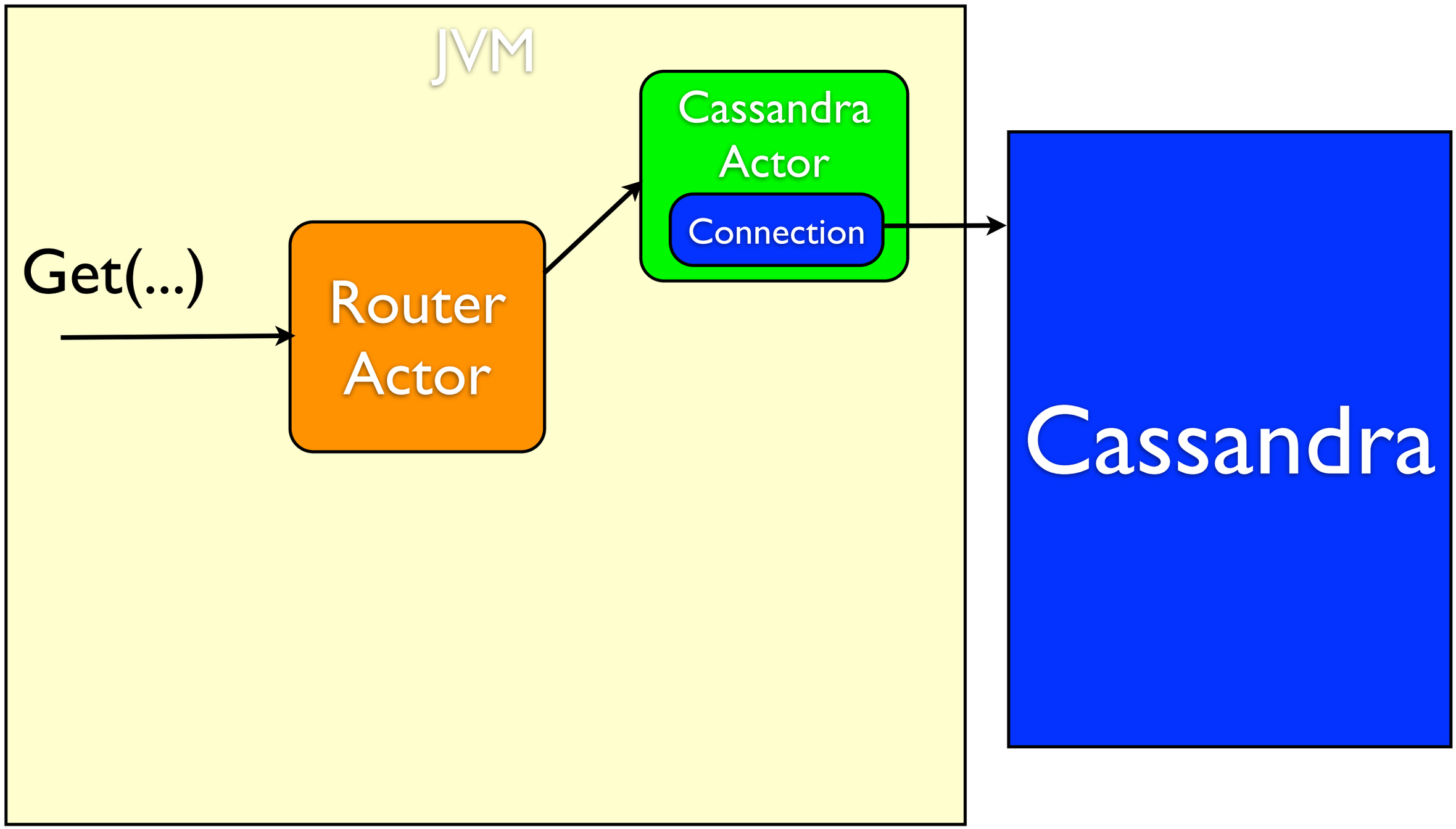
Cassandra Actor



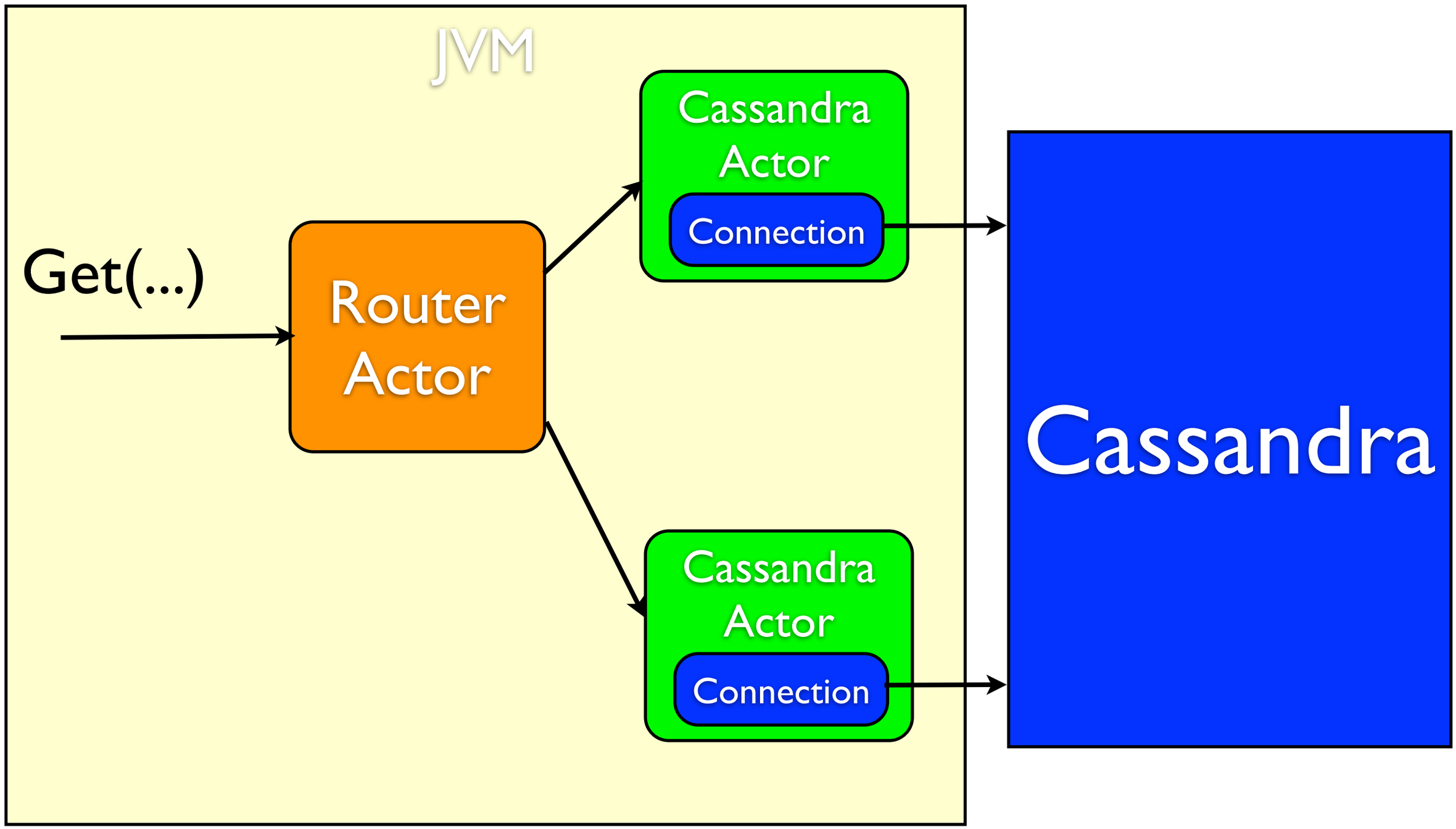
Cassandra Actor



Cassandra Actor



Cassandra Actor



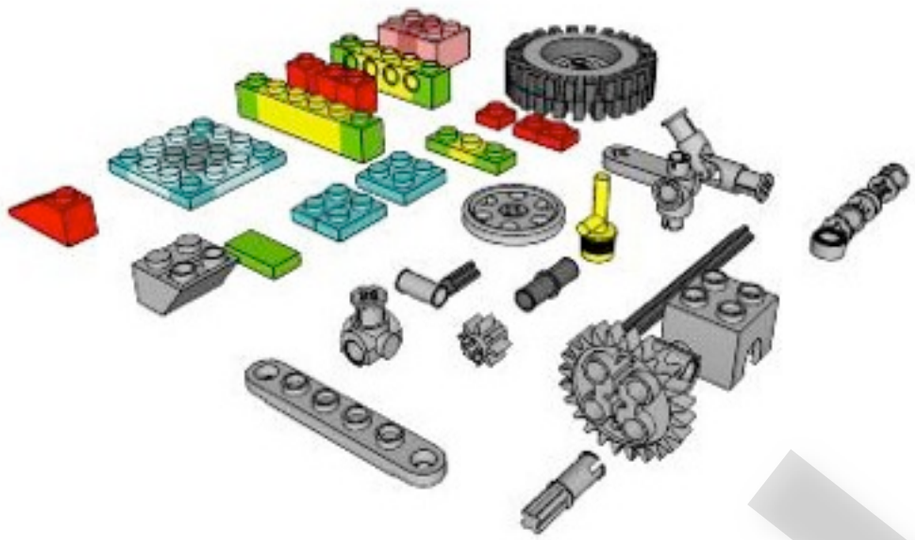
Async Action

```
trait Action[A]{ self =>
  def execute(k: Keyspace): A
}
```

Save as Async

```
trait Action[A]{ self =>
  def execute(k: Keyspace): A
}
```

```
case class Save[T](p:T)(implicit co:CassandraObject[T])
  extends Action[Unit]{

  def execute(k: Keyspace) {
    blocking.Save(p)(k, co)
  }
}
```



Composition

```
Update[Person]("joe-123", p => p.copy(name = "Mr " + p.name))
```

Composition Demo

Monad

```
trait Monad[A]{
  def unit[B](a: B) : Monad[B]
  def flatMap[B](f: A => Monad[B]) : Monad[B]
}
```

Monad laws:

```
val x,y,z : Monad[_] = _

//identity
x.flatMap(a => x.unit(a)) == x

//associativity
x.flatMap(a => y.flatMap(b => z)) == x.flatMap(a => y).flatMap(b => z)
```

Functor

```
trait Functor[A]{
  def map[B](f : A => B ): Functor[B]
}
```

Functor

```
trait Functor[A]{
  def map[B](f : A => B ): Functor[B]
}
```

Monad is a Functor:

```
trait Monad[A] extends Functor[A]{
  def unit[B](a: B) : Monad[B]
  def flatMap[B](f: A => Monad[B]) : Monad[B]

  def map[B](f: (A) => B) = flatMap(a => unit(f(a)))
}
```

For comprehension

```
case class Account(id: String, balance : Double)

def Total(id1: String, id2: String ) : CassandraMonad[Double] =
  Read[Account](id1)
    .flatMap( a1 =>
              Read[Account](id2)
                .map( a2 => a1.get.balance + a2.get.balance )
```

For comprehension

```
case class Account(id: String, balance : Double)

def Total(id1: String, id2: String ) : CassandraMonad[Double] =
  Read[Account](id1)
    .flatMap( a1 =>
              Read[Account](id2)
                .map( a2 => a1.get.balance + a2.get.balance )
```

==

For comprehension

```
case class Account(id: String, balance : Double)

def Total(id1: String, id2: String ) : CassandraMonad[Double] =
  Read[Account](id1)
    .flatMap( a1 =>
              Read[Account](id2)
                .map( a2 => a1.get.balance + a2.get.balance )
            )
```

==

```
def Total2(id1: String, id2: String ) : CassandraMonad[Double] =
  for {
    a1 <- Read[Account](id1)
    a2 <- Read[Account](id2)
  } yield a1.get.balance + a2.get.balance
```

Option is a Monad

```scala
def readBalance(id : String) : Option[Double] = {...}

val total : Option[Double] = for {
  balance1 <- readBalance("1")
  balance2 <- readBalance("2")
} yield (balance1 + balance2)
```

Future is a Monad

```
def fetchFacebookFriends(): Future[Set[String]] = {...}
def fetchTwitterFriends(): Future[Set[String]] = {...}
def fetchCurrentFriends(): Future[Set[String]] = {...}

def saveNewFriends : PartialFunction[Set[String], Unit] = {...}

val calculateNewFriends : Future[Set[String]] = for {
  facebookFriends <-  fetchFacebookFriends()
  twitterFriends <-  fetchTwitterFriends()
  currentFriends <-  fetchCurrentFriends()
} yield (facebookFriends ++ twitterFriends) -- currentFriends

calculateNewFriends onSuccess saveNewFriends
```

Collections are Monads

```
scala> List(1,2,3,4).map(a => a*a)
res2: List[Int] = List(1, 4, 9, 16)
```

```
scala> List(1,2,3).flatMap(a => List.fill(a)(a*a) )
res3: List[Int] = List(1, 4, 4, 9, 9, 9)
```

Option -> Iterable

```
def readBalance(id : String) : Option[Double] = id match {
  case "1" => Some(100)
  case "2" => Some(200)
  case _ => None
}

List("1","2","3") flatMap(id => readBalance(id) ) sum; // 300
```

Questions?

Peter Gabryanczyk
@piotrga
peter@scala-experts.com
http://blog.scala4java.com