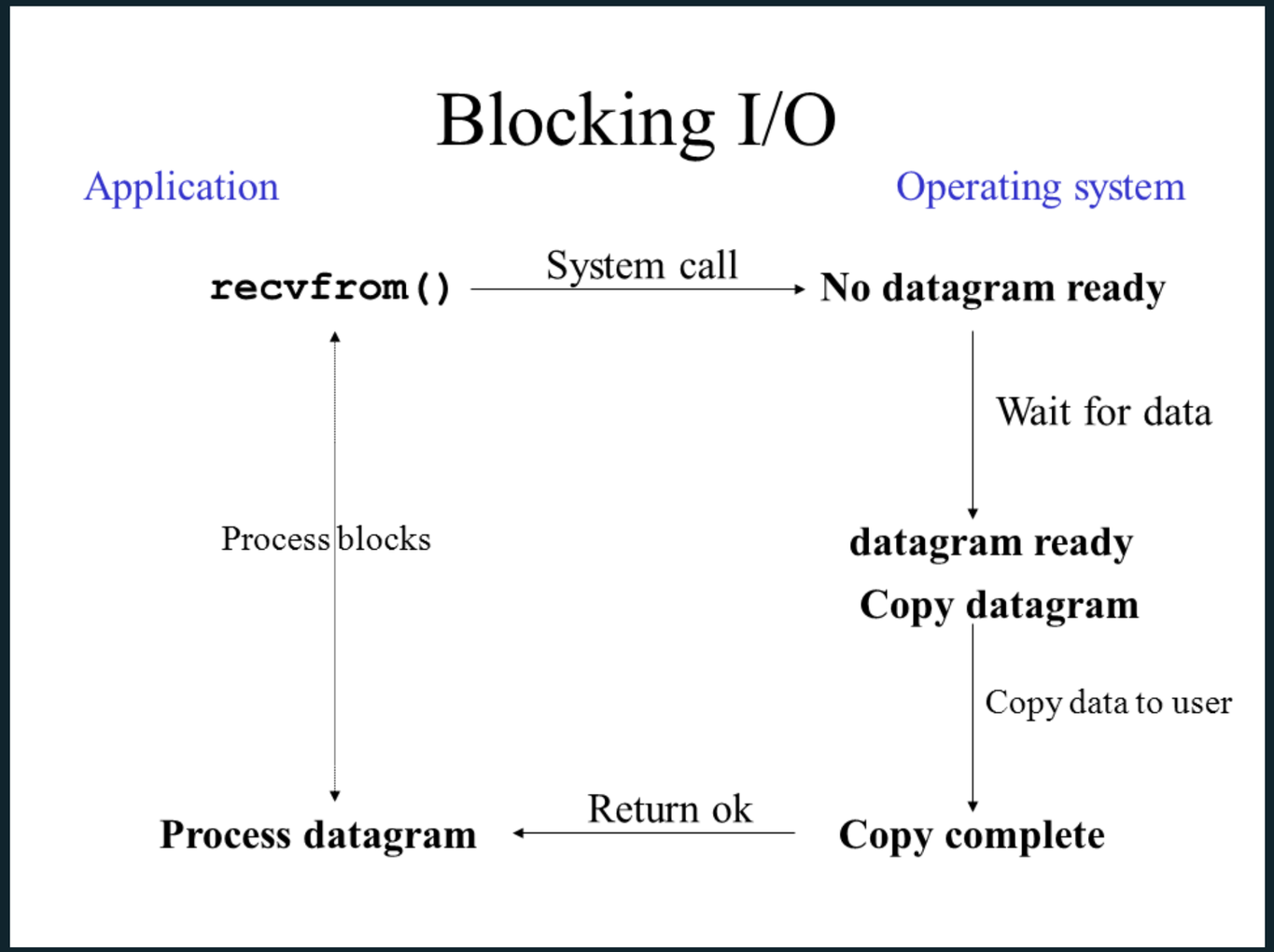
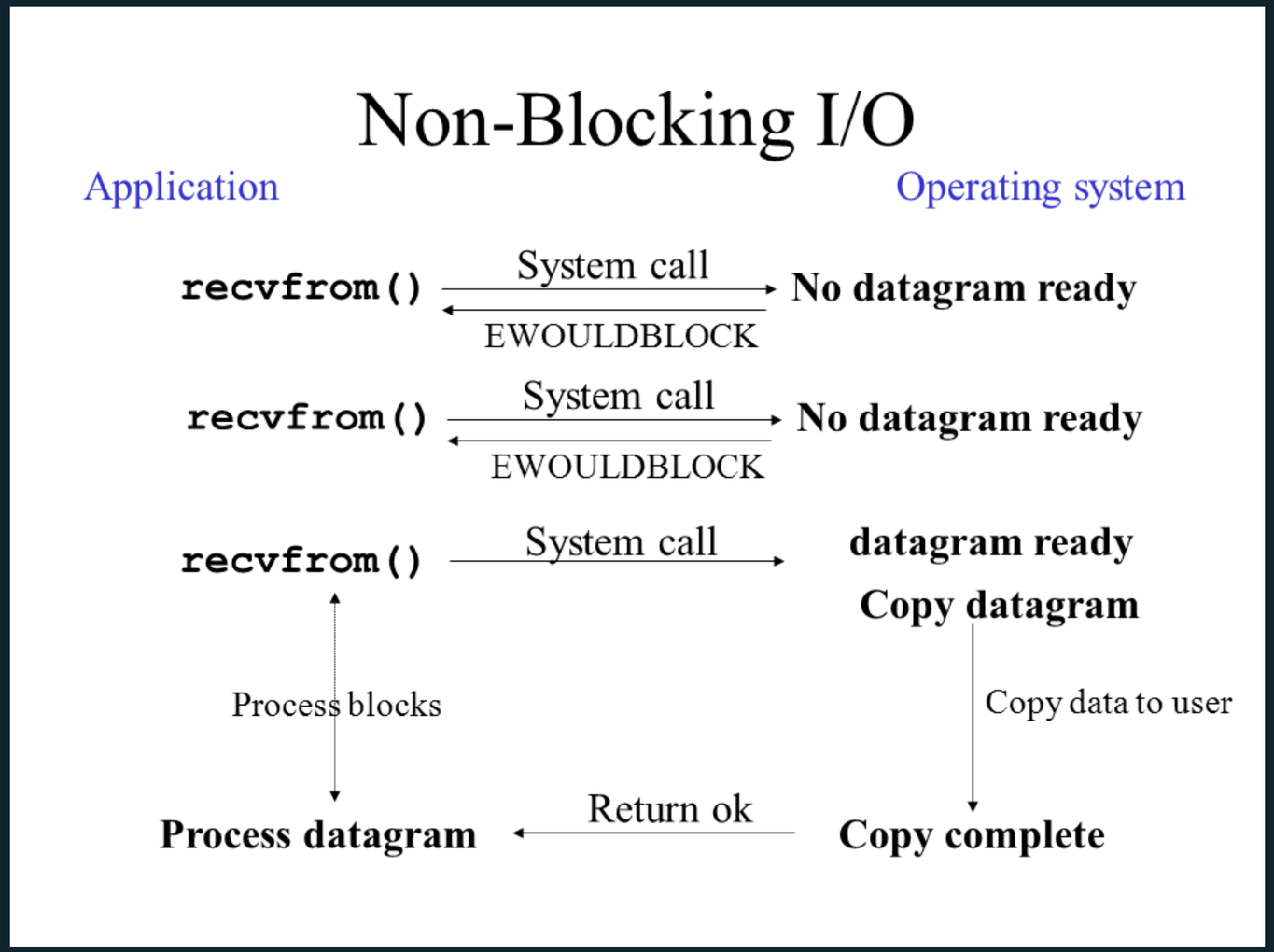
Miserable Future

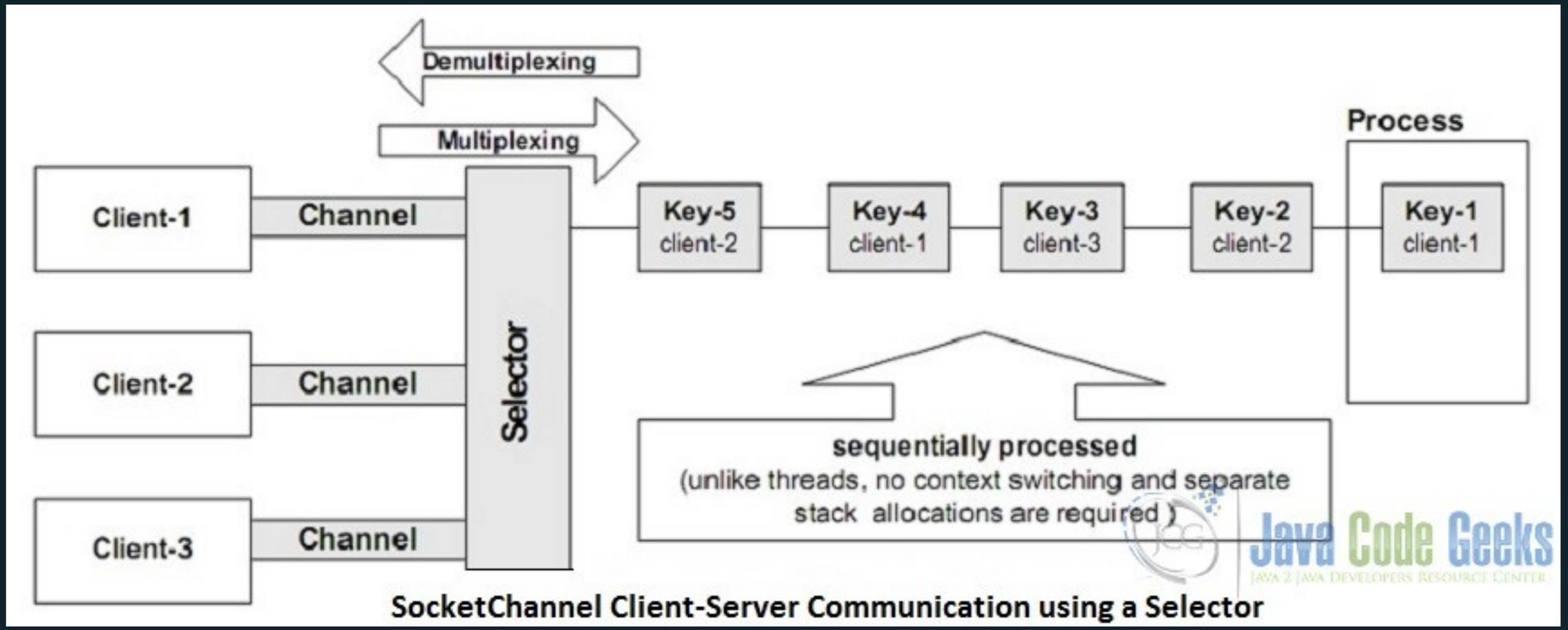
Synchronous I/O



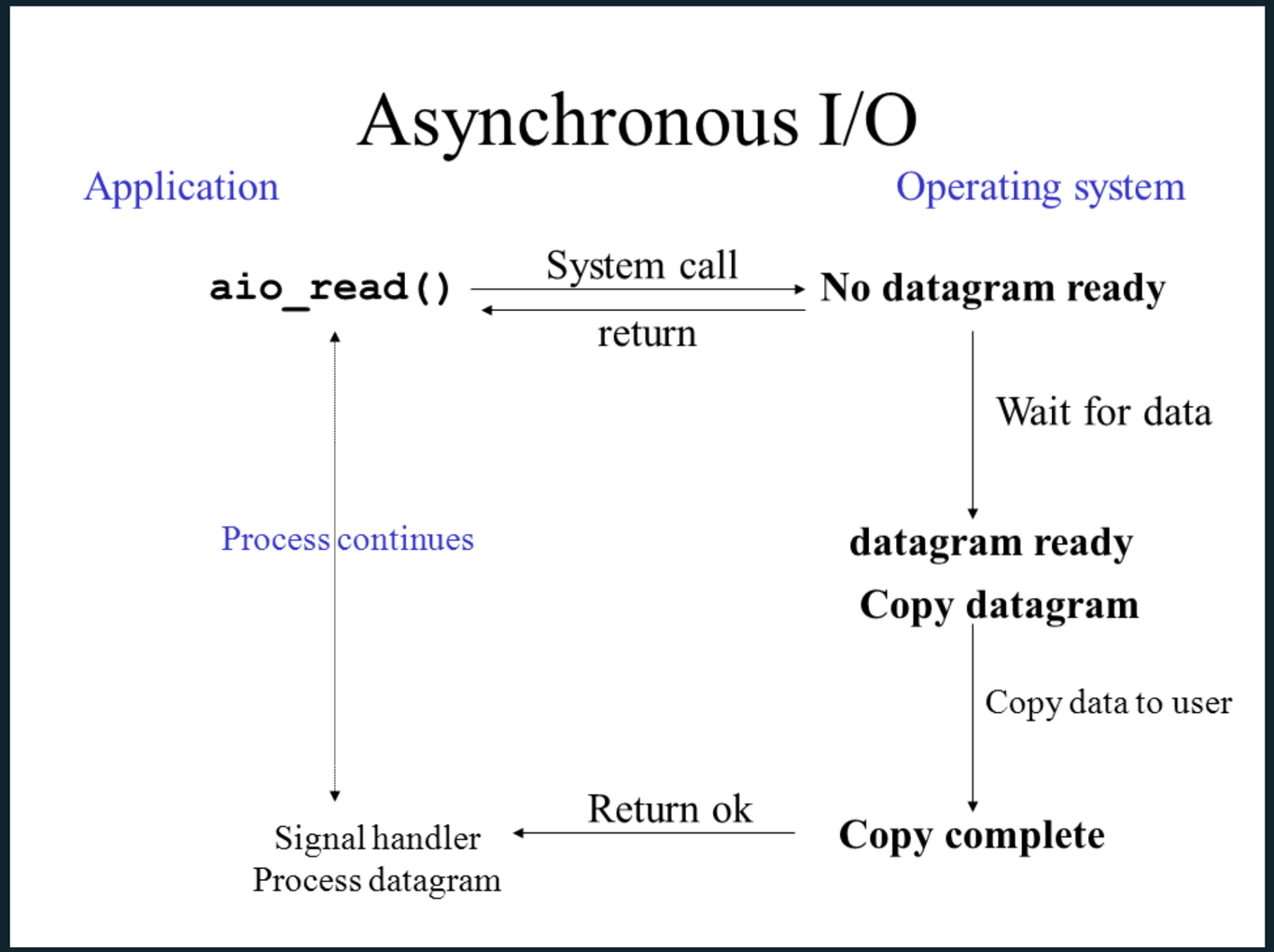
Non-Blocking I/O



Java Non-Blocking I/O



Asynchrous I/O



Jave 7

```
Future<Integer> futureOne = executorService.submit(() -> {
  Thread.sleep(1000);
  return 1;
});

Future<Integer> futureTwo = executorService.submit(() -> {
  Thread.sleep(2000);
  return 2;
});

System.out.println(futureOne.get(5000, TimeUnit.MILLISECONDS) + futureTwo.get(5000, TimeUnit.MILLISECONDS));
```

Java 8

```
CompletableFuture<Integer> futureOne = CompletableFuture
.runAsync(() -> {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
})
.thenApplyAsync(v -> 1);

CompletableFuture<Integer> futureTwo = CompletableFuture
.runAsync(() -> {
  try {
    Thread.sleep(2000);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
})
.thenApplyAsync(v -> 2);

CompletableFuture<Integer> result = futureOne.thenCombineAsync(futureTwo, (i1, i2) -> i1 + i2);
System.out.println(result.get(5000, TimeUnit.MILLISECONDS));
```

What is Future ?

A Future is an object holding a value which may become available at some point.

— When a Future is completed with a **value**.

— When a Future is completed with an **exception** thrown by the computation.

Future trait

```
trait Future[+T] {
  def onComplete[U](f : scala.Function1[scala.util.Try[T], U])(implicit executor : scala.concurrent.ExecutionContext) : Unit

  def foreach[U](f : scala.Function1[T, U])(implicit executor : scala.concurrent.ExecutionContext) : Unit

  def flatMap[S](f: T => Future[S])(implicit executor: ExecutionContext): Future[S]

  def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S]
}
```

Callback Method 1/3

```
val futureOne = Future {
  Thread.sleep(1000)
  1
}

val futureTwo = Future {
  Thread.sleep(2000)
  2
}

futureOne.onComplete {
  case Success(s1) =>
    futureTwo.onComplete {
      case Success(s2) => println(s1 + s2)
      case Failure(f2) => println(s"error, $f2")
    }
  case Failure(f1) => println(s"error, $f1")
}

// focus on happy path
futureOne.foreach { one =>
  futureTwo.foreach { two =>
    println(one + two)
  }
}

// focus on unhappy path
futureOne.failed.foreach { f1 =>
  futureTwo.failed.foreach { f2 =>
    println(f1)
    println(f2)
  }
}
```

Callback Method 2/3

— Callback methods are called asynchronously when a future completes.

— The order in which callbacks are executed is **not guaranteed**, the callback is executed eventually.

— `onComplete` have the result type `Unit`, so they can't be chained(callbacks registered on the **same** future are **unordered**).

Callback Method 3/3

```
@volatile var result = 0

val value = Future {
  123
}

value foreach { v =>
  if(v == 123)
     result += 1
}

value foreach { v =>
  if(v == 123)
     result += 2
}
```

Functional Composition

```
val futureOne = Future {
  Thread.sleep(1000)
  1
}

val futureTwo = Future {
  Thread.sleep(2000)
  2
}

val result = for {
  r1 <- futureOne
  r2 <- futureTwo
} yield {
  r1 + r2
}
```

Global implicit ExecutionContext 1/2

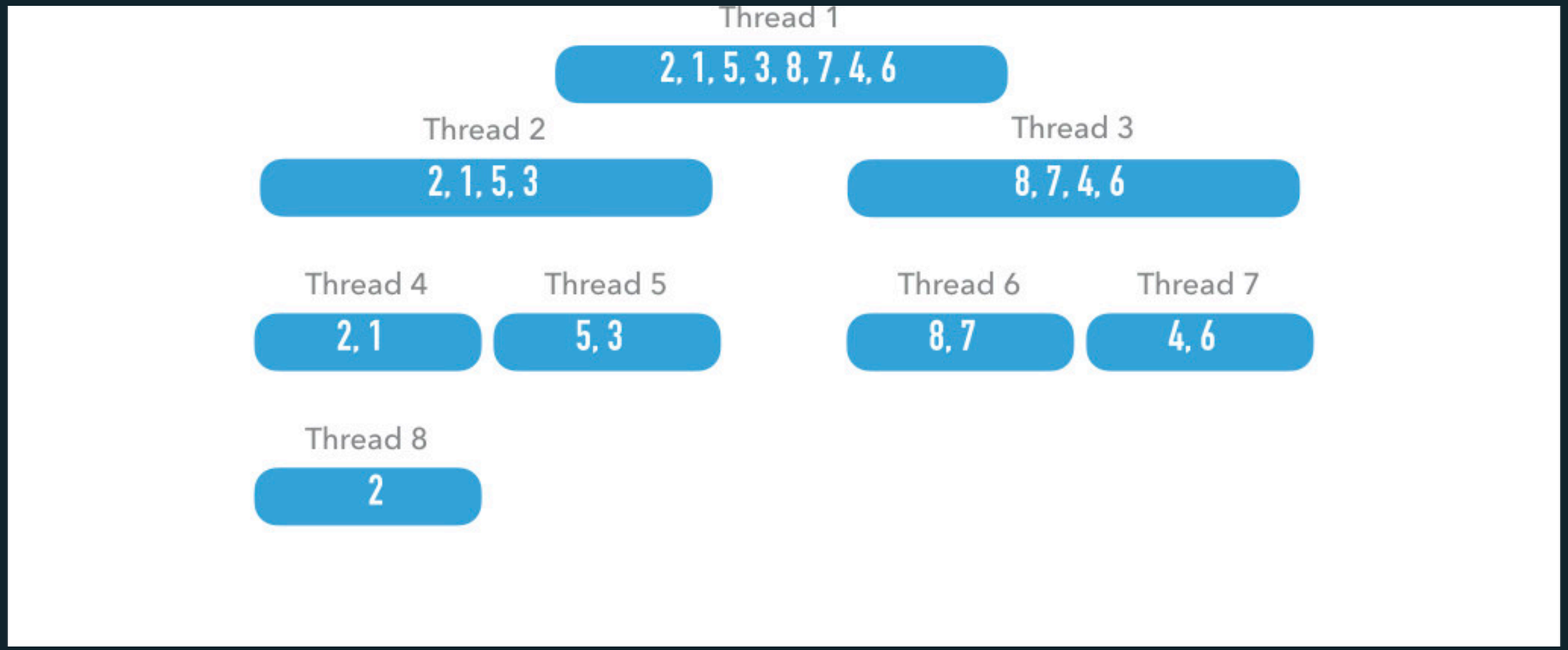
Cannot find an implicit ExecutionContext.
You might pass an (implicit ec: ExecutionContext) parameter to your method or import scala.concurrent.ExecutionContext.Implicits.global.

Global implicit ExecutionContext 2/2

- An `ExecutionContext` is similar to an Executor
- `ExecutionContext.global` is an ExecutionContext backed by a ForkJoinPool.
- Number of threads
  - `scala.concurrent.context.minThreads`
  - `scala.concurrent.context.numThreads`
  - `scala.concurrent.context.maxThreads`
  - `scala.concurrent.context.maxExtraThreads`

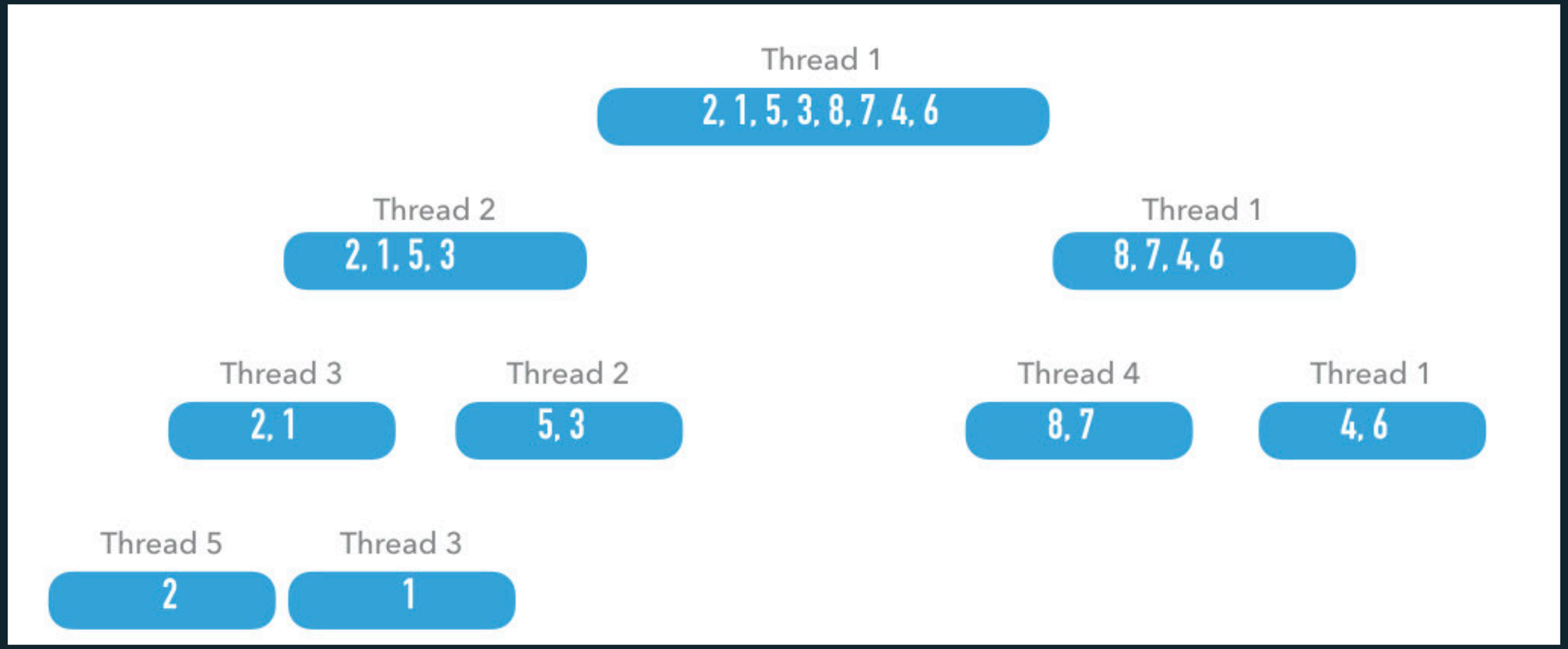
ForkJoinPool 1/2

Example: Sorting a list with merge sort and using a fixed thread pool of 8 cores



ForkJoinPool 2/2

Using a fork join pool of 8 cores



Java thread pools

- `FixedThreadPool`
  n threads will process tasks at the time, when the pool is saturated, new tasks will get added to **a queue** without a limit on size.
- `CachedThreadPool`
  not put tasks into a queue. When all current threads are busy, it creates another thread to run the task.
- `ForkJoinPool`
  uses a **work-stealing** algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

Promise 1/2

As a **writable**, **single-assignment container**, which completes a future. That is you can finish a future **manually**.

The Promise and Future are complementary concepts.

Promise 2/2

```
def httpClient = {
  val promise = Promise[Response]
  val asyncHttpClient = new DefaultAsyncHttpClient()

  asyncHttpClient.prepareGet("http://www.example.com/").execute(new AsyncCompletionHandler<Response>(){
    @Override
    def onCompleted(response: Response) = {
        // Do something with the Response
        // ...

        promise.complete(response)
        response
    }

    @Override
    def onThrowable(t:Throwable) = {
        // Something wrong happened.
        promise.failure(t)
    }
  })

  promise.future
}
```

What is the features of future 1/3?

— Future is a **eager** evaluation.
— Future **momorize** results

What is the features of future 2/3?

```
val r = for {
  a <- Future{
         Thread.sleep(1)
         "a"
       }
  b <- Future{
         Thread.sleep(2)
         "b"
       }
} yield{
  a + b
}

Await.result(r, 3 second)
```

What is the features of future 3/3?

```
val fa = Future{
  Thread.sleep(1)
  "a"
}

val fb = Future{
  Thread.sleep(2)
  "b"
}

val r = for {
  a <- fa
  b <- fb
} yield{
  a + b
}

Await.result(r, 2 second)
```

Can I put any code blocks into Future? 1/3

This is in general an **anti-pattern**:

```
def add(x: Int, y: Int) = Future { x + y }
```

— If you want to initialize a Future[T] with a constant, always use `Future.successful()`.

— If you want to initialize a Future[T] with a exception, always use `Future.failed()`.

Can I put any code blocks into Future? 2/3

```
def future(x: Int): Future[Int] =
  for {
    r1 <- Future(x + Random.nextInt())
    r2 <- Future(r1 - Random.nextInt())
    r3 <- Future(r2 * Random.nextInt())
    r4 <- Future(r3 / Random.nextInt())
  } yield {
    r4
  }

def futureWithSuccessful(x: Int): Future[Int] =
  for {
    r1 <- Future.successful(x + Random.nextInt())
    r2 <- Future.successful(r1 - Random.nextInt())
    r3 <- Future.successful(r2 * Random.nextInt())
    r4 <- Future.successful(r3 / Random.nextInt())
  } yield {
    r4
  }
```

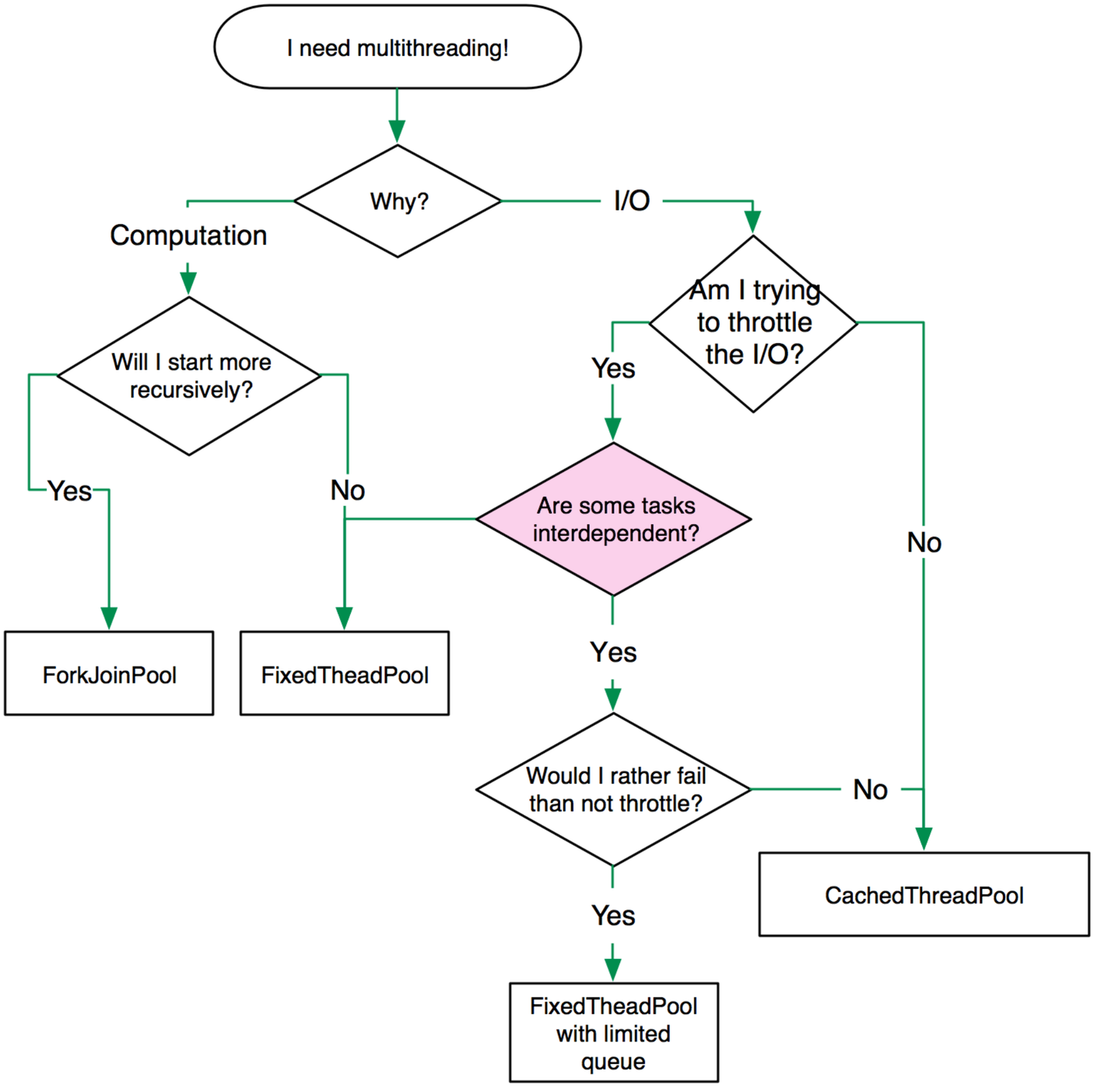
Can I put any code blocks into Future? 3/3

::Benchmark Future.future::
cores: 8
Parameters(size -> 3000): 0.46817 ms
Parameters(size -> 6000): 0.826904 ms
Parameters(size -> 9000): 1.107552 ms
Parameters(size -> 12000): 1.762273 ms
Parameters(size -> 15000): 2.176588 ms

::Benchmark Future.futureWithSuccessful::
cores: 8
Parameters(size -> 3000): 0.310014 ms
Parameters(size -> 6000): 0.605951 ms
Parameters(size -> 9000): 0.923088 ms
Parameters(size -> 12000): 1.201882 ms
Parameters(size -> 15000): 1.456783 ms

How to handle blocking I/O

How to choose a thread-pool


I need multithreading!
Why?
Computation
I/O
Will I start more recursively?
Yes
No
ForkJoinPool
FixedTheadPool
Am I trying to throttle the I/O?
Yes
No
Are some tasks interdependent?
Yes
Would I rather fail than not throttle?
No
Yes
CachedThreadPool
FixedTheadPool with limited queue

How do I execute a bunch of Future concurrently ?

```
object Stock {
  private def getStockPrice(id: String): Future[Double] = Future {
    val price = Random.nextDouble()
    price
  }

  def mapThenSequence(): Future[List[Double]] = {
    val stockIds: List[String]                = List.fill(Random.nextInt(1000))(Random.nextInt(1000).toString)
    val mapResults: List[Future[Double]]      = stockIds.map(getStockPrice)
    val sequenceResults: Future[List[Double]] = Future.sequence(mapResults)
    sequenceResults
  }

  def traverse(): Future[List[Double]] = {
    val stockIds: List[String]                = List.fill(Random.nextInt(1000))(Random.nextInt(1000).toString)

    // This is useful for performing a parallel map. For example, to apply a function to all items of a list
    // in parallel
    val traverseResults: Future[List[Double]] = Future.traverse(stockIds)(getStockPrice)
    traverseResults
  }
}
```

Future is so intricate, do we have another choice?

Yes !!!

— Monix

— cats-effect

Wish you have a better future

References:

— FUTURES AND PROMISES
— Is non-local return in Scala new?
— FixedThreadPool, CachedThreadPool, or ForkJoinPool? Picking correct Java executors for background tasks
— Fork/Join
— scala-best-practices
— What are the use cases of scala.concurrent.Promise?
— Scala, promises, futures, Netty and Memcached get together to have monads
— I'm purr-e pics
— Thread Pools
— Choosing an ExecutorService