Miserable Future

Jave 7

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

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

```java
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

```scala
trait Future[+T] {
  @deprecated("use `foreach` or `onComplete` instead (keep in mind that they take total rather than partial functions)", "2.12.0")
  def onSuccess[U](pf: PartialFunction[T, U])(implicit executor: ExecutionContext): Unit

  @deprecated("use `onComplete` or `failed.foreach` instead (keep in mind that they take total rather than partial functions)", "2.12.0")
  def onFailure[U](@deprecatedName('callback) pf: PartialFunction[Throwable, U])(implicit executor: ExecutionContext): Unit

  def onComplete[U](@deprecatedName('func) f: Try[T] => U)(implicit executor: 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 1/2

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

  import scala.concurrent.Future
// import scala.concurrent.Future

  import scala.util.{Failure, Success}
// import scala.util.{Failure, Success}

  import scala.concurrent.duration._
// import scala.concurrent.duration._

  import scala.concurrent.{Await, Future}
// import scala.concurrent.{Await, Future}

  val futureOne = Future {
    Thread.sleep(1000)
    1
  }
// futureOne: scala.concurrent.Future[Int] = Future(<not completed>)

  val futureTwo = Future {
    Thread.sleep(2000)
    2
  }
// futureTwo: scala.concurrent.Future[Int] = Future(<not completed>)

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

  Await.result(futureTwo, 5 second)
// 3
// res1: Int = 2
```

Callback Method 2/2

- Callback methods are called asynchronously when a future completes.

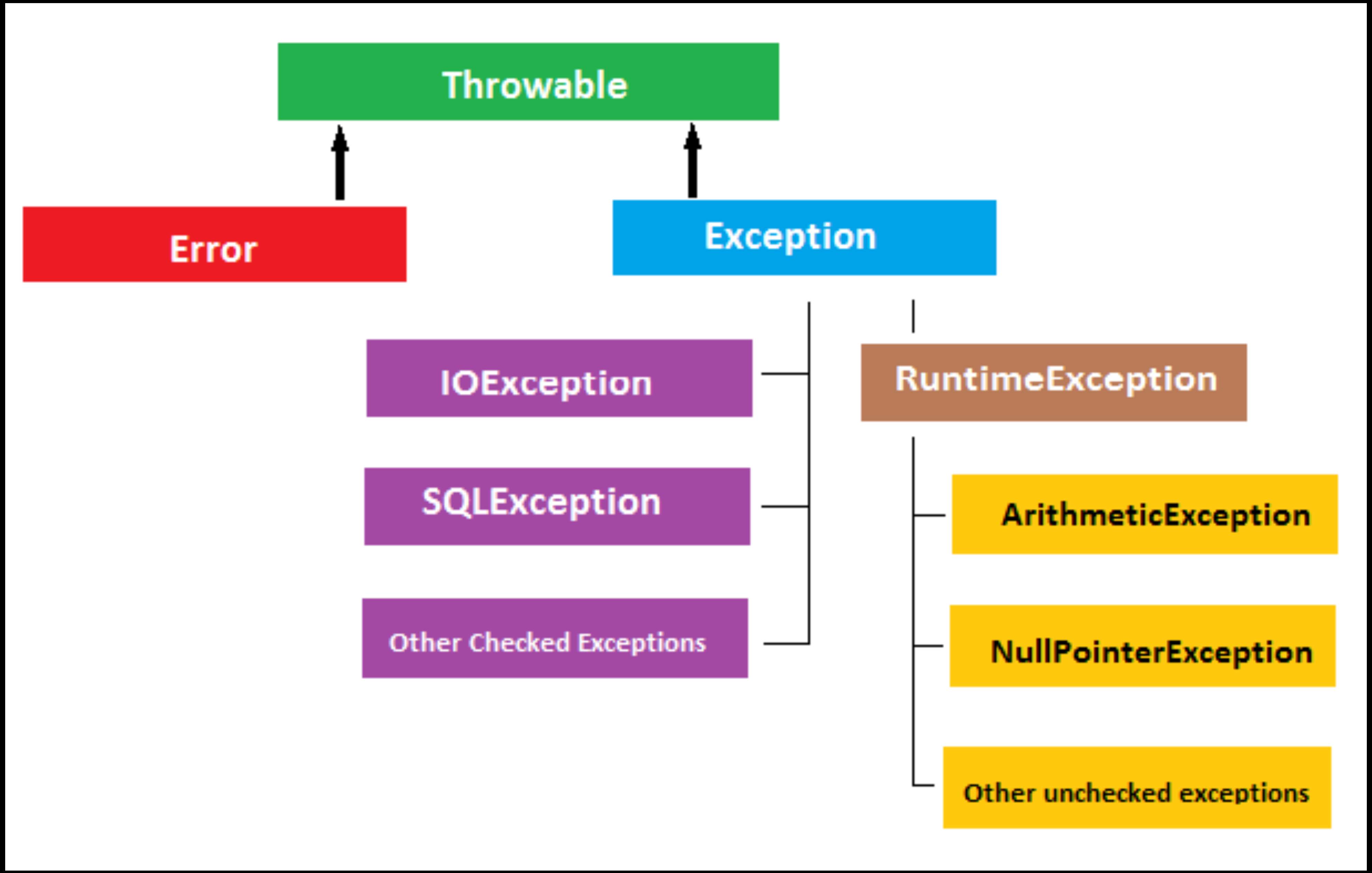
- The order in which callbacks are executed is ***not guaranteed***, the callback is executed eventually.

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

Functional Composition

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

  import scala.concurrent.duration._
// import scala.concurrent.duration._

  import scala.concurrent.{Await, Future}
// import scala.concurrent.{Await, Future}

  val futureOne = Future {
    Thread.sleep(1000)
    1
  }
// futureOne: scala.concurrent.Future[Int] = Future(<not completed>)

  val futureTwo = Future {
    Thread.sleep(2000)
    2
  }
// futureTwo: scala.concurrent.Future[Int] = Future(<not completed>)

  val result = for {
    r1 <- futureOne
    r2 <- futureTwo
  } yield {
    r1 + r2
  }
// result: scala.concurrent.Future[Int] = Future(<not completed>)

  Await.result(result, 5 second)
// res2: Int = 3
```

Java Exception Hierarchy



Exceptions

- `scala.runtime.NonLocalReturnControl[_]`
  Returning from a ***nested anonymous*** function
- `ExecutionException`
  - `InterruptedException`
  - `Error`
  - `scala.util.control.ControlThrowable`.

Fatal exceptions (as determined by `NonFatal`)
This informs the code managing the executing threads of the problem and allows it to fail fast, if necessary.

Execution Context 1/ 2

An `ExecutionContext` is similar to an Executor: it is free to execute computations in a new thread, in a pooled thread or in the current thread

`ExecutionContext.global` is an ExecutionContext backed by a ForkJoinPool.

By default the `ExecutionContext.global` sets the parallelism level of its underlying fork-join pool to ***the amount of available processors***.

- `scala.concurrent.context.minThreads`
- `scala.concurrent.context.numThreads`
- `scala.concurrent.context.maxThreads`

Execution Context 2/ 2

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


implicit lazy val global: ExecutionContext = impl.ExecutionContextImpl.fromExecutor(null: Executor)


def fromExecutor(e: Executor, reporter: Throwable => Unit = ExecutionContext.defaultReporter): ExecutionContextImpl =
new ExecutionContextImpl(Option(e).getOrElse(createDefaultExecutorService(reporter)), reporter)
```

Thread pools 1/2

- `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.

Thread pools 2/2

Task 1: A(5 sec) {a1(2 sec), a2(3 sec)}
Task 2: B(10 sec) {b1(6 sec), b2(4 sec)}

- `FixedThreadPool` (size = 2)

```
|--------------------------------------|--------------------------------------|
0                                      5                                      10
Thread 1: A                            Thread 1: empty                        Thread 1: empty
Thread 2: B                            Thread 2: B                            Thread 2: empty
```

- `ForkJoinPool` (size = 2)

```
|----------------------|-----------------------|----------------------|
0                      2                       5                      9
Thread 1(a1, a2): a1   Thread 1(a2): a2        Thread 1: b2           Thread 1: empty
Thread 2(b1, b2): b1   Thread 2(b1, b2): b1    Thread 2: b1           Thread 2: empty
```

Promise 1/3

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 2/3

```
implicit def scalaToTwitterFuture[T](f: Future[T])(implicit ec: ExecutionContext): twitter.Future[T] = {
  val promise = twitter.Promise[T]()
  f.onComplete(promise update _)
  promise
}

implicit def twitterToScalaFuture[T](f: twitter.Future[T]): Future[T] = {
  val promise = Promise[T]()
  f.respond(promise complete _)
  promise.future
}
```

Promise 3/3

```
import org.asynchttpclient.*
import scala.concurrent._

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
}
```

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()`.

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

```scala
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: 4
name: Java HotSpot(TM) 64-Bit Server VM
osArch: x86_64
osName: Mac OS X
vendor: Oracle Corporation
version: 25.144-b01
Parameters(size -> 3000): 0.530467
Parameters(size -> 6000): 1.016189
Parameters(size -> 9000): 1.494891
Parameters(size -> 12000): 2.067224
Parameters(size -> 15000): 2.341089
```

```
::Benchmark Future.futureWithSuccessful::
cores: 4
name: Java HotSpot(TM) 64-Bit Server VM
osArch: x86_64
osName: Mac OS X
vendor: Oracle Corporation
version: 25.144-b01
Parameters(size -> 3000): 0.36767
Parameters(size -> 6000): 0.693213
Parameters(size -> 9000): 0.988905
Parameters(size -> 12000): 1.442866
Parameters(size -> 15000): 1.85345
```

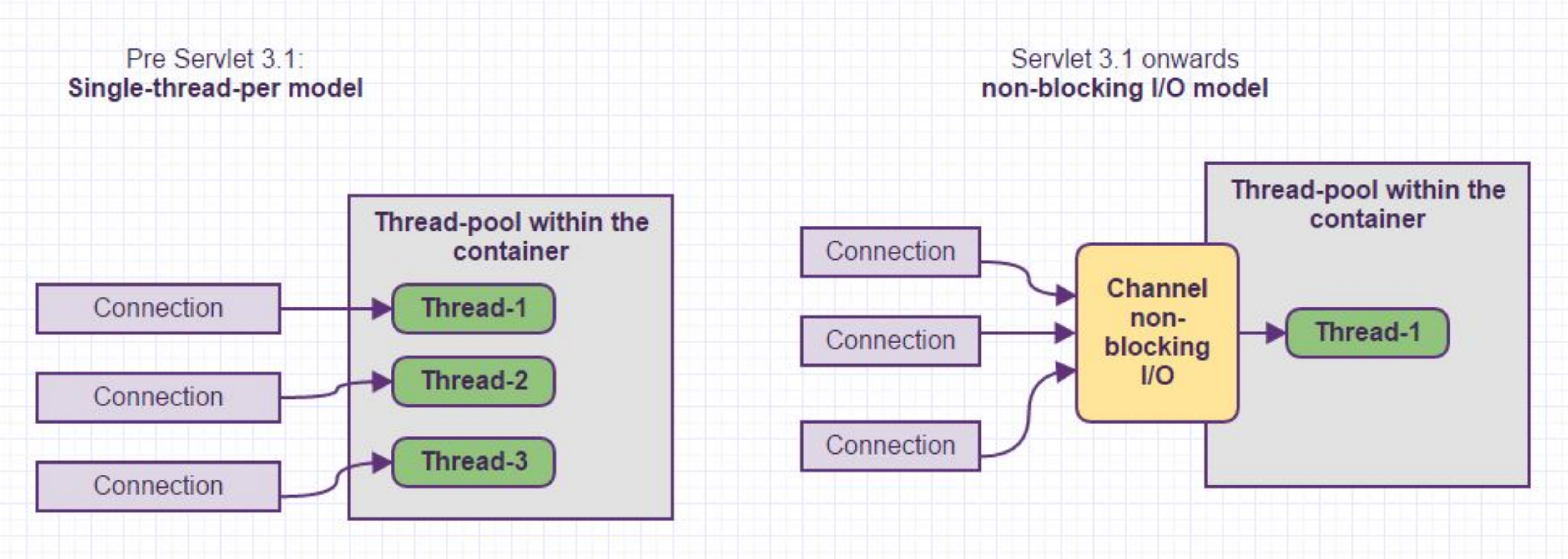
What is `blocking` ? 1/5

Blocking calls have to be marked with a `blocking` call that signals to the `BlockContext` a blocking operation.

Lets the ExecutionContext know that a blocking operation happens, such that the ExecutionContext can decide what to do about it, such as adding more threads to the thread-pool (which is what Scala's ForkJoin thread-pool does).

What is `blocking` ? 2/5

- synchronous and blocking IO
- synchronous and non-blocking IO
- asynchronous and non-blocking IO



What is `blocking` ? 3/5

```scala
object Sentinel extends Runnable {
  override def run(): Unit = {
    while (true) {
      println(s"Active threads count: ${Thread.activeCount}")
      Thread.sleep(1000)
    }
  }
}

import java.util.concurrent.{Executors, TimeUnit}

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

object NonBlock extends App {
  Executors.newSingleThreadScheduledExecutor.schedule(Sentinel, 1000, TimeUnit.MILLISECONDS)

  for (i <- 0 until 100) {
    Future {
      Thread.sleep(3000)
    }
  }
}
```

What is `blocking` ? 4/5

```scala
object Sentinel extends Runnable {
  override def run(): Unit = {
    while (true) {
      println(s"Active threads count: ${Thread.activeCount}")
      Thread.sleep(3000)
    }
  }
}

import java.util.concurrent.{Executors, TimeUnit}

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

object Block extends App {
  Executors.newSingleThreadScheduledExecutor.schedule(Sentinel, 3000, TimeUnit.MILLISECONDS)

  for (i <- 0 until 100) {
    Future {
      blocking {
        Thread.sleep(3000)
      }
    }
  }
}
```

What is blocking ? 5/5

```scala
object Sentinel extends Runnable {
  override def run(): Unit = {
    while (true) {
      println(s"Active threads count: ${Thread.activeCount}")
      Thread.sleep(3000)
    }
  }
}

import java.util.concurrent.{Executors, TimeUnit}

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

object Block extends App {
  Executors.newSingleThreadScheduledExecutor.schedule(Sentinel, 3000, TimeUnit.MILLISECONDS)

  val executorService = Executors.newFixedThreadPool(4)
  implicit val ec     = ExecutionContext.fromExecutorService(executorService)
  for (i <- 0 until 100) {
    Future {
      blocking {
        Thread.sleep(3000)
      }
    }
  }
}
```

Should use a separate thread-pool for blocking I/O ?

Yes, it's better to create a second thread-pool / execution context and execute all blocking calls on that, leaving the application's thread-pool to deal with CPU-bound stuff.

In a blocking environment, `thread-pool-executor` is better than `fork-join` because no work-stealing is possible, and a `fixed-pool-size` size should be used and set to the maximum size of the underlying resource.

How do I execute a bunch of Future concurrently 1/3?

Future is a ***eager*** evaluation.

```
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)
```

How do I execute a bunch of Future concurrently 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)
```

How do I execute a bunch of Future concurrently 2/2?

```scala
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)
    val traverseResults: Future[List[Double]] = Future.traverse(stockIds)(getStockPrice)
    traverseResults
  }
}
```

Future is so intricate, do we have another choice?

Yes !!!

- Monix
- cats-effect
- Scalaz

What about Twitter Future?

It is your turn !!!

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