

4. Basic building blocks of concurrency

Nelma Moreira & **José Proença**

Concurrent programming (CC3040) 2024/2025

CISTER – U.Porto, Porto, Portugal

<https://fm-dcc.github.io/cp2425>



Overview

Blocks of sequential code running concurrently and sharing memory:

- What is Scala?
- Concurrency in Java and its memory model
- Basic concurrency blocks and libraries
- *Futures and Promises*
- *Data-Parallel Collections*
- *Reactive Programming (Concurrently)*
- *Software Transactional Memory*
- Actor model

- Thread pools: `Executor` and `ExecutionContext`
- Non-blocking synchronisation – compare-and-set (CAS)
- Lazy (concurrent) values
- Concurrent collections
- Running OS processes
- *Futures and Promises*

Existing thread pools in Scala

```
Executor executor = anExecutor;  
executor.execute(new RunnableTask1());  
executor.execute(new RunnableTask2());  
...
```

```
import scala.concurrent._  
import java.util.concurrent.ForkJoinPool  
  
object ExecutorsCreate extends App {  
  val executor = new ForkJoinPool  
  executor.execute(new Runnable {  
    def run() = log("This task is run  
                    asynchronously.")  
  })  
  Thread.sleep(500) // not needed with  
                    fork:=false in SBT  
}
```

- **Executor**: can start a new thread, an existing one, or the current one
- Abstracts from the management of threads
- **ExecutorService**: API that extends Executor with shutdown
 - `executor.shutdown` → executes all tasks and then stops working threads
 - `executor.awaitTermination(...)` → force termination if, after a given time, the tasks are not completed

```
import scala.concurrent._  
  
object ExecutionContextGlobal extends App {  
  val ectx = ExecutionContext.global  
  ectx.execute(new Runnable {  
    def run() = log("Running on the execution context.")  
  })  
  Thread.sleep(500)  
}
```

```
object ExecutionContextCreate extends App {  
  val pool = new forkjoin.ForkJoinPool(2)  
  val ectx = ExecutionContext.fromExecutorService(pool)  
  ectx.execute(new Runnable {  
    def run() = log("Running on the execution context  
    again.")  
  })  
  Thread.sleep(500)  
}
```

- `scala.concurrent:` has `ExecutionContext`
- Similar to `Executor` but more Scala specific
- often used as implicit parameter
- `global`: default execution context (internally uses a `ForkJoinPool`)
- `fromExecutorService`: creates `ExecutionContext` from `ExecutorService`

Similar to `threads`:

```
def thread(body: =>Unit): Thread
  = {
    val t = new Thread {
      override def run() = body
    }
    t.start()
    t
  }
```

We now define `execute`

```
def execute(body: =>Unit) =
  ExecutionContext.global.execute(
    new Runnable { def run() = body }
  )
// For example:
object ExecutionContextSleep extends App {
  for (i<- 0 until 32) execute {
    Thread.sleep(2000)
    log(s"Task_$i_completed.")
  }
  Thread.sleep(10000)
}
```


- **Expected:** all executions terminate after 2s
- **Result:** only some execute after 2s

```
object ExecutionContextSleep
  extends App {
    for (i<- 0 until 32) execute {
      Thread.sleep(2000)
      log(s"Task_$i_completed.")
    }
    Thread.sleep(10000)
  }
```

```
object ExecutionContextSleep
  extends App {
    for (i<- 0 until 32) execute {
      Thread.sleep(2000)
      log(s"Task_$i_completed.")
    }
    Thread.sleep(10000)
  }
```

- **Expected:** all executions terminate after 2s
- **Result:** only some execute after 2s
- Using quad-core CPU with hyper threading
- global has 8 threads in the thread pool

```
object ExecutionContextSleep
  extends App {
    for (i<- 0 until 32) execute {
      Thread.sleep(2000)
      log(s"Task_$i_completed.")
    }
    Thread.sleep(10000)
  }
```

- **Expected:** all executions terminate after 2s
- **Result:** only some execute after 2s
- Using quad-core CPU with hyper threading
- global has 8 threads in the thread pool
- executes tasks in batches of 8
- after 2s, 8 tasks print "completed"
- after 2s more, 8 more print "completed"
- **sleep:** all enter a **timed waiting state**

```
object ExecutionContextSleep
  extends App {
    for (i<- 0 until 32) execute {
      Thread.sleep(2000)
      log(s"Task_$i_completed.")
    }
    Thread.sleep(10000)
  }
```

- **Expected:** all executions terminate after 2s
- **Result:** only some execute after 2s
- Using quad-core CPU with hyper threading
- global has 8 threads in the thread pool
- executes tasks in batches of 8
- after 2s, 8 tasks print "completed"
- after 2s more, 8 more print "completed"
- **sleep:** all enter a **timed waiting state**
- if T1 **waits** for T10 to **notify:** **blocks indefinitely**

Lock-free programming

- **atomic variable**: memory location that supports **complex linearizable operations**
- ... i.e., **appears to occur atomically**
- write of a volatile operation:
simple linearizable operation
- at least two reads and/or writes:
complex linearizable operation

- **atomic variable**: memory location that supports **complex linearizable operations**
- ... i.e., **appears to occur atomically**
- write of a volatile operation:
simple linearizable operation
- at least two reads and/or writes:
complex linearizable operation
- **java.util.concurrent.atomic** supports some complex ones:
 - AtomicBoolean
 - AtomicInteger
 - AtomicLong
 - AtomicReference

Variation of Example 1 (getUniqueId)

```
import
    java.util.concurrent.atomic._

object AtomicUid extends App {
    private val uid =
        new AtomicLong(0L)

    def getUniqueId(): Long =
        uid.incrementAndGet()
    execute {
        log(s"Uid asynchronously: {getUniqueId()}")
    }
    log(s"Got a unique id: {getUniqueId()}")
}
```

- CAS can be used to implement others:
 - getAndSet
 - decrementAndGet
 - addAndGet
- available in all atomic variables
- including AtomicReference[T]

Long-CAS conceptually equivalent to:

```
def compareAndSet(ov: Long, nv: Long):  
    Boolean = this.synchronized {  
        if (this.get != ov) false else {  
            this.set(nv)  
            true  
        }  
    }
```

Ref-CAS conceptually equivalent to:

```
def compareAndSet(ov: T, nv: T):  
    Boolean = this.synchronized {  
        if (!(this.get eq ov)) false else {  
            this.set(nv)  
            true  
        }  
    }
```


- Back to Example 1 (getUniqueId)
- Need to keep-on-trying
- Looks like busy-waiting, but it is much better
- Here: using (cheap) recursion instead of a loop

```
@tailrec
def getIdUnique(): Long = {
  val oldUid = uid.get
  val newUid = oldUid + 1
  if (uid.compareAndSet(oldUid,
    newUid)) newUid
  else getIdUnique()
}
```

- Lock-free programs: without locks
(with synchronized)
- Achieved using atomic variables (and some re-trying)
- No locks, no deadlocks...

- Lock-free programs: without locks
(with synchronized)
- Achieved using atomic variables (and some re-trying)
- No locks, no deadlocks...
- (almost):
 - lock-free \Rightarrow use atomic variables
(for atomicity)
 - use atomic variables \nRightarrow lock-free

- Lock-free programs: without locks (with synchronized)
- Achieved using atomic variables (and some re-trying)
- No locks, no deadlocks...
- (almost):
 - lock-free \Rightarrow use atomic variables (for atomicity)
 - use atomic variables \nRightarrow lock-free

```
object AtomicLock extends App {  
  private val lock = new  
    AtomicBoolean(false)  
  def mySynchronized(body: =>Unit):  
    Unit = {  
    while (!lock.compareAndSet(false,  
      true)) {}  
    try body finally lock.set(false)  
  }  
  var count = 0  
  for (i<- 0 until 10) execute {  
    mySynchronized { count += 1 } }  
  Thread.sleep(1000)  
  log(s"Count is: $count")  
}
```

Lock-freedom

Given a set of threads and an operation **OP**.

OP is **lock-free** if at least one thread always completes **OP** after a finite number of steps, regardless of the speed at which different threads progress.

- **Example 1:** getUniqueld()
- **Example 2:** Logging Bank Transfers
- **Example 3:** Thread pool
- **Example 4:** Batman
- **Example 5:** Concurrent filesystem

Filesystem API

T1 is creating F:

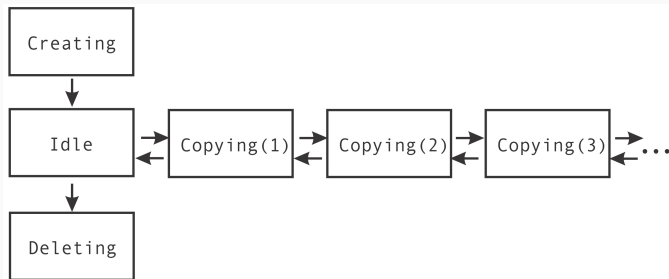
T2 cannot copy or delete F

T1 & T2 are copying F:

T3 cannot delete F

T1 is deleting F:

T2 cannot copy nor delete F



[in "Learning Concurrent Programming in Scala", pg. 79]

Filesystem API

T1 is creating F:

T2 cannot copy or delete F

T1 & T2 are copying F:

T3 cannot delete F

T1 is deleting F:

T2 cannot copy nor delete F

```
class Entry(val isDir: Boolean) {  
  val state = new AtomicReference[State](new Idle)  
}
```

```
sealed trait State  
class Idle extends State  
class Creating extends State  
class Copying(val n: Int) extends State  
class Deleting extends State
```


Deleting: prepare (checks for permission) then delete (perform delete)

Copying: acquire (get permission); copy (perform action); then release (give permission)

```
@tailrec
private def prepareForDelete(entry: Entry): Boolean = {
  val s0 = entry.state.get
  s0 match {
    case i: Idle =>
      if (entry.state.compareAndSet(s0, new Deleting)) true
      else prepareForDelete(entry)
    case c: Creating =>
      logMessage("File currently created, cannot delete."); false
    case c: Copying =>
      logMessage("File currently copied, cannot delete."); false
    case d: Deleting =>
      false
  }
}
```

`logMessage`: presented later – similar to `log`, but stores the log message

“ABA” problem: two readings of the same value **A** lead to believe that **B** was never present (type of race condition)

Illustrated by a bad acquire-release for **Copying**, using a mutable **n** in:

```
Copying(var n: Int)
```

```
def releaseCopy(e: Entry): Copying = e.state.get match {  
  case c: Copying =>  
    val nstate = if (c.n == 1) new Idle else new Copying(c.n - 1)  
    if (e.state.compareAndSet(c, nstate)) c // returns the old Copying state  
    else releaseCopy(e)  
}
```

```
def acquireCopy(e: Entry, c: Copying) = e.state.get match {  
  case i: Idle =>  
    c.n = 1 // updating 'n' in 'c'  
    if (!e.state.compareAndSet(i, c)) acquireCopy(e, c)  
  case oc: Copying =>  
    c.n = oc.n + 1 // updating 'n' in 'c'  
    if (!e.state.compareAndSet(oc, c)) acquireCopy(e, c)  
}
```

Optimization: reusing previous `Copying` if possible

T1 releases
(**T2 starts rel.**)

⇒

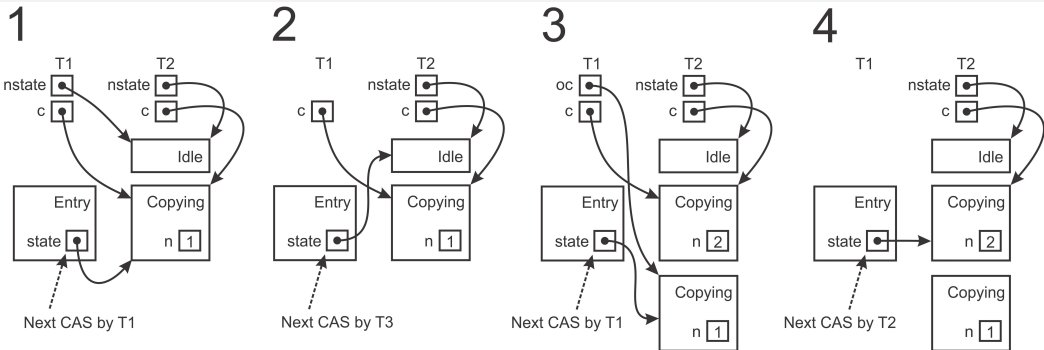
T3 acquires

⇒

T1 acquires

⇒

T2 releases
(**wrongly**)



[in "Learning Concurrent Programming in Scala", pg. 82]

- use fresh objects in `AtomicReference`
- use immutable objects in `AtomicReference`
- avoid re-assigning the same value to an atomic variable
- only increment or decrement values of numeric atomic variables (if possible)

Creating and handling processes

- **So far:** run in a single JVM
- **Now:** run processes outside JVM
- **Why:**

- **So far:** run in a single JVM
- **Now:** run processes outside JVM
- **Why:**
 - Some programs do not exist in Scala/Java

- **So far:** run in a single JVM
- **Now:** run processes outside JVM
- **Why:**
 - Some programs do not exist in Scala/Java
 - Want to sandbox untrusted code

- **So far:** run in a single JVM
- **Now:** run processes outside JVM
- **Why:**
 - Some programs do not exist in Scala/Java
 - Want to sandbox untrusted code
 - Performance (running independent code)

- **So far:** run in a single JVM
- **Now:** run processes outside JVM
- **Why:**
 - Some programs do not exist in Scala/Java
 - Want to sandbox untrusted code
 - Performance (running independent code)
- Using the `scala.sys.process` package

```
import scala.sys.process._  
object ProcessRun extends App {  
  val command = "ls"  
  val exitcode = command.! // run process (with side effects)  
  log(s"command░exited░with░status░$exitcode") }  

```

```
def lineCount(filename: String): Int = {  
  val output = s"wc░$filename".!! // run and retrieve stdout  
  output.trim.split("░").head.toInt }  

```

```
object ProcessAsync extends App {  
  val lsProcess = "ls░-R░/".run() // run and returns a Process object  
  Thread.sleep(1000)  
  log("Timeout░-░killing░ls!")  
  lsProcess.destroy() } // kill a slow process  

```

<https://www.scala-lang.org/api/2.13.x/scala/sys/process/ProcessBuilder.html>

Lazy values

- **lazy values**: initialized when read for the first time
- these should not depend-on/modify state (non-determinism)
- code in **singleton objects**: lazy execution
- under the hood: first write uses a lock – to ensure at most a thread initialises a lazy value
- **stack overflow** (sequential code) can become **deadlock** (concurrent code)

```
object LazyValsCreate extends App {  
  var x = 5  
  lazy val y = x+2  
  execute { log(s"Wrk: ␣y␣=␣$y")}  
  x = 10  
  log(s"Main: ␣y␣=␣$y")  
  // y = 7 or 12 in both cases  
  Thread.sleep(500)  
}
```

```
object LazyValsDeadlock extends App {  
  object A { lazy val x: Int = B.y }  
  object B { lazy val y: Int = A.x }  
  execute { B.y }  
  A.x  
}
```

Concurrent (mutable) collections

- Naive code: arbitrarily returns different results and exceptions
- Corruption of the internal state
- Possible fixes:
 - immutable collections + atomic variables
 - mutable collections + synchronized
 - dedicated libraries

```
import scala.collection._
object CollectionsBad extends App {
  val buffer =
    mutable.ArrayBuffer[Int]()
  def asyncAdd(numbers: Seq[Int]) =
    execute {
      buffer += numbers
      log(s"buffer = $buffer")
    }
  asyncAdd(0 until 10)
  asyncAdd(10 until 20)
  Thread.sleep(500)
}
```

- Naive code: arbitrarily returns different results and exceptions
- Corruption of the internal state
- Possible fixes:
 - immutable collections + atomic variables
(does not scale)
 - mutable collections + synchronized
(assuming collections do not block; may not scale)
 - dedicated libraries
(far better performance and scalability)

```
import scala.collection._
object CollectionsBad extends App {
  val buffer =
    mutable.ArrayBuffer[Int]()
  def asyncAdd(numbers: Seq[Int]) =
    execute {
      buffer += numbers
      log(s"buffer = $buffer")
    }
  asyncAdd(0 until 10)
  asyncAdd(10 until 20)
  Thread.sleep(500)
}
```

- Concurrent queues
 - `java.util.concurrent.BlockingQueue` **interface**
 - `...ArrayBlockingQueue` **class** (bounded)
 - `...LinkedBlockingQueue` **class** (unbounded)
- Concurrent Sets and Maps
 - `scala.collection.concurrent.Map` **trait**
 - `java.util.concurrent.ConcurrentHashMap` **class**

Operation	Exception	Special value	Timed	Blocking
Dequeue	remove(): T	poll(): T	poll(t: Long, u: TimeUnit): T	take(): T
Enqueue	add(x: T)	offer(x: T): Boolean	offer(x: T, t: Long, u: TimeUnit)	put(x: T)
Inspect	element: T	peek: T	N/A	N/A

[in "Learning Concurrent Programming in Scala", pg. 90]

We will compile a queue of messages when **logging** messages in our file system

```
class FileSystem(...) {  
  ...  
  private val messages = new LinkedBlockingQueue[String]  
  val logger = new Thread {  
    setDaemon(true)  
    override def run() = while (true) log(messages.take())  
  }  
  logger.start()  
  def logMessage(msg: String): Unit = messages.offer(msg)  
}
```

```
...  
val fileSystem = new FileSystem(".") // to be defined later  
fileSystem.logMessage("Testing log!")
```

- `concurrentQueue.iterator`
- can produce inconsistent results
- while traversing and modifying, the iterator can be updated
- (heavier) exceptions create a copy when producing an iterator

Example 5: files as a Map in our FileSystem



```
import scala.collection.convert.decorateAsScala._
import java.io.File
import org.apache.commons.io.FileUtils // needs "commons-io" in build.sbt

class FileSystem(val root: String) {
  val rootDir = new File(root)
  val files: concurrent.Map[String, Entry] =
    new ConcurrentHashMap().asScala
  for (f <- FileUtils.iterateFiles(rootDir, null, false).asScala)
    files.put(f.getName, new Entry(false))

  ...
}
```

Recall the `prepareForDelete(entry)`

```
...  
def deleteFile(filename: String): Unit = {  
  files.get(filename) match {  
    case None =>  
      logMessage(s"Path '$filename' does not exist!")  
    case Some(entry) if entry.isDir =>  
      logMessage(s"Path '$filename' is a directory!")  
    case Some(entry) => execute {  
      if (prepareForDelete(entry))  
        if (FileUtils.deleteQuietly(new File(filename)))  
          files.remove(filename)  
    }  
  }  
}
```


Signature	Description
<code>putIfAbsent (k: K, v: V): Option[V]</code>	This atomically assigns the value <code>v</code> to the key <code>k</code> if <code>k</code> is not in the map. Otherwise, it returns the value associated with <code>k</code>.
<code>remove (k: K, v: V): Boolean</code>	This atomically removes the key <code>k</code> if it is associated to the value equal to <code>v</code> and returns <code>true</code> if successful.
<code>replace (k: K, v: V): Option[V]</code>	This atomically assigns the value <code>v</code> to the key <code>k</code> and returns the value previously associated with <code>k</code> .
<code>replace (k: K, ov: V, nv: V): Boolean</code>	This atomically assigns the key <code>k</code> to the value <code>nv</code> if <code>k</code> was previously associated with <code>ov</code> and returns <code>true</code> if successful.

[in *"Learning Concurrent Programming in Scala"*, pg. 95]

- These use “equals” instead of the reference (which CAS does)
- Avoid `null` as key or value (often used as special values)
- Methods `+=`, `-=`, `put`, `update`, `get`, `apply`, `remove` are (non-complex) linearizable

Wrapping up our Filesystem (Example 5)

Recall our **broken** **acquireCopy**/**releaseCopy** methods (**ABA problem**) – slide19

```
@tailrec
private def acquire(entry: Entry): Boolean = {
  val s0 = entry.state.get
  s0 match {
    case _: Creating | _: Deleting =>
      logMessage("File␣inaccessible,␣cannot␣copy."); false
    case i: Idle =>
      if (entry.state.compareAndSet(s0, new Copying(1))) true
      else acquire(entry)
    case c: Copying =>
      if (entry.state.compareAndSet(s0, new Copying(c.n+1))) true
      else acquire(entry)
  }
}
```

Same CAS retry-approach for releasing.

```
@tailrec
private def release(entry: Entry): Unit = {
  val s0 = entry.state.get
  s0 match {
    case c: Creating =>
      if (!entry.state.compareAndSet(s0, new Idle)) release(entry)
    case c: Copying =>
      val nstate = if (c.n == 1) new Idle else new Copying(c.n-1)
      if (!entry.state.compareAndSet(s0, nstate)) release(entry)
  }
}
```

Finally: wrapper for copying a file.

```
def copyFile(src: String, dest: String): Unit = {  
  files.get(src) match {  
    case Some(srcEntry) if !srcEntry.isDir => execute {  
      if (acquire(srcEntry)) try {  
        val destEntry = new Entry(isDir = false)  
        destEntry.state.set(new Creating)  
        if (files.putIfAbsent(dest, destEntry) == None) try {  
          FileUtils.copyFile(new File(src), new File(dest))  
        } finally release(destEntry)  
      } finally release(srcEntry)  
    }  
  }  
}
```

- `executor.execute(...)`
- lock-free programming with atomic variables
- `av.compareAndSet(...)`
- ABA problem
- Lazy values & “lazy” objects
- `java.util.concurrent.BlockingQueue`
- `scala.collection.concurrent.Map`
- *weakly consistent iterators*
- *custom concurrent data structures*
- Filesystem example
- Processes outside JVM

- `executor.execute(...)`
- lock-free programming with atomic variables
- `av.compareAndSet(...)`
- ABA problem
- Lazy values & “lazy” objects
- `java.util.concurrent.BlockingQueue`
- `scala.collection.concurrent.Map`
- *weakly consistent iterators*
- *custom concurrent data structures*

- Filesystem example
- Processes outside JVM

Next

- *Futures and Promises*
- *Data-Parallel Collections*
- *Reactive Programming (Concurrently)*
- *Software Transactional Memory*
- **Actors**

Extra: Quick guide to the Future

- High-level asynchronous constructs
- Used very often by libraries that use concurrent tasks
- **Future**: read-only placeholder for the result of an ongoing computation
- **Promise**: writable, single-assignment container that completes a Future

```
object LazyValsCreate extends App {  
  import scala.concurrent._  
  import Future  
  import ExecutionContext.Implicits.global  
  
  val futFive = Future {  
    log("Computation started.")  
    val result = {  
      Thread.sleep(5000)  
      5  
    }  
    log("Computation finally finished.")  
    result // : Future[Int]  
  }  
}
```

Small notes

- Using our
ExecutionContext.global
- import of Implicits.global \Leftrightarrow
implicit val ec =
ExecutionContext.global

```
object LazyValsCreate extends App {  
  import scala.concurrent._  
  import Future  
  import ExecutionContext.Implicits.global  
  
  val futFive = Future {  
    log("Computation started.")  
    val result = {  
      Thread.sleep(5000)  
      5  
    }  
    log("Computation finally finished.")  
    result // : Future[Int]  
  }  
}
```

Small notes

- Using our
ExecutionContext.global
- import of Implicits.global \Leftrightarrow
implicit val ec =
ExecutionContext.global
- Future.apply[T]
(body: => T)
(implicit ec: ExecutionContext)
: Future[T]

```
import scala.util.{Try, Success, Failure}
...
val futFive = Future { ... }

def getFive = futFive.onComplete {
  case Success(n) =>
    log(s"Got number: $n")
  case Failure(exception) =>
    log(s"Failed to get number: $exception")
}

getFive
Thread.sleep(6000)
getFive // What to expect?
```

A future has 2 stages

- **Not completed** (ongoing computation)
- **Completed** (finished computation)
 - Success(res) (no errors)
 - Failure(err) (got errors)

Method 1: onComplete

```
futFive.onComplete[U]  
  (callback: Try[Int] => U)  
  (implicit exec: ...)  
  : Unit
```

```
import scala.concurrent.duration._
...
val futFive = Future { ... }

def getFive = // : Int
  Await.result(futFive, Duration.inf)
def doneFive = // : Future[Int]
  Await.ready (futFive, 6.seconds)

val r1 = getFive // doneFive
val r2 = getFive // doneFive
Thread.sleep(6000)
val r3 = getFive // doneFive
```

A future has 2 stages

- **Not completed** (ongoing computation)
- **Completed** (finished computation)
 - Success(res) (no errors)
 - Failure(err) (got errors)

Method 2: **Await.result/ready**

- Can throw `TimeoutException`

```
import scala.concurrent.duration._
...
val futFive = Future { ... }

def checkFive = // : Boolean
  futFive.isCompleted
def maybeFive = // : Option[Try[Int]]
  futFive.value

val r1 = checkFive // maybeFive
val r2 = checkFive // maybeFive
Thread.sleep(6000)
val r3 = checkFive // maybeFive
```

A future has 2 stages

- **Not completed** (ongoing computation)
- **Completed** (finished computation)
 - Success(res) (no errors)
 - Failure(err) (got errors)

Polling: **isCompleted/value**

How to check the status without blocking

```
import scala.concurrent.duration._
...
val futTwo  = Future { ... } // takes 2s
val futFive = Future { ... } // takes 5s

val r1 = futTwo.map(x => x*3)
val r2 = for (
  x <- futFive
  y <- r1
) yield
  x+y
log(s"a")_␣${r1.value}/${r2.value}")
Thread.sleep(3000)
log(s"b")_␣${r1.value}/${r2.value}")
Thread.sleep(6000)
log(s"c")_␣${r1.value}/${r2.value}")
```

- read and update futures “in advance”
- map, flatMap, for-loops

```
import scala.concurrent.duration._
...
val futTwo  = Future { ... } // takes 2s
val futFive = Future { ... } // takes 5s

val r1 = futTwo.map(x => x*3)
val r2 = for (
  x <- futFive
  y <- r1
) yield
  x+y
log(s"a")_␣${r1.value}/${r2.value}")
Thread.sleep(3000)
log(s"b")_␣${r1.value}/${r2.value}")
Thread.sleep(6000)
log(s"c")_␣${r1.value}/${r2.value}")
```

- read and update futures “in advance”
- map, flatMap, for-loops
- very common mechanism to run tasks in parallel
- used in many libraries

A promise p can...

- p.future
- p.success(res)
- p.failure(exception)
- p.complete(Success(res))

```
import scala.concurrent.{Future, Promise}
import scala.concurrent...Implicits.global

val prom = Promise[Int]()
val alice = Future { // execute/thread
  val res = {Thread.sleep(2000); 2}
  prom.success(res)
  log(s"gave_ '$res'")
  Thread.sleep(2000) // more work
}
val fut = prom.future
val bob = Future { // execute/thread
  Thread.sleep(3000)
  fut.foreach { r =>
    log("got_ '$r'")
  }
}
```

- `executor.execute(...)`
- lock-free programming with atomic variables
- `av.compareAndSet(...)`
- ABA problem
- Lazy values & “lazy” objects
- `java.util.concurrent.BlockingQueue`
- `scala.collection.concurrent.Map`
- *weakly consistent iterators*
- *custom concurrent data structures*

- Filesystem example
- Processes outside JVM
- **Futures and Promises (overview)**

Next

- *Data-Parallel Collections*
- *Reactive Programming (Concurrently)*
- *Software Transactional Memory*
- **Actors**