

### 3. Concurrency in Java and its memory model

---

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:

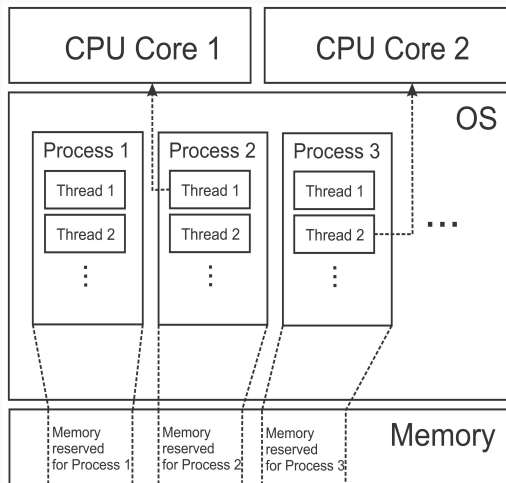
- What is Scala?
- Concurrency in Java and its memory model
- Basic concurrency blocks and libraries
- Futures and promises
- Actor model

## Synchronisation

- Coordination of multiple executions in a concurrent system
- Mechanisms to **order** concurrent executions
- Mechanisms to **exchange** information

## Exchanging information

- **Concurrent programs: shared memory communication**
- Distributed programs: message passing communication



[ in "Learning Concurrent  
Programming in Scala", pg. 32 ]

Starting a new JVM instance always creates **only one** process.

In that process, **multiple threads** can run simultaneously.

Unlike runtimes (e.g. Python), the JVM:  
does not implement its custom threads,  
maps each **Java thread** to an **OS thread**

# Managing threads

---

```
object ThreadsMain extends App {  
  val t: Thread =  
    Thread.currentThread  
  val name = t.getName  
  println(s"I am the thread $name")  
}
```

Using SBT, this prints:

[info] I am the thread sbt-bg-threads-1

In SBT do “set fork := true”  
(or add “fork := true” to build.sbt)

It will then it prints:

[info] I am the thread main

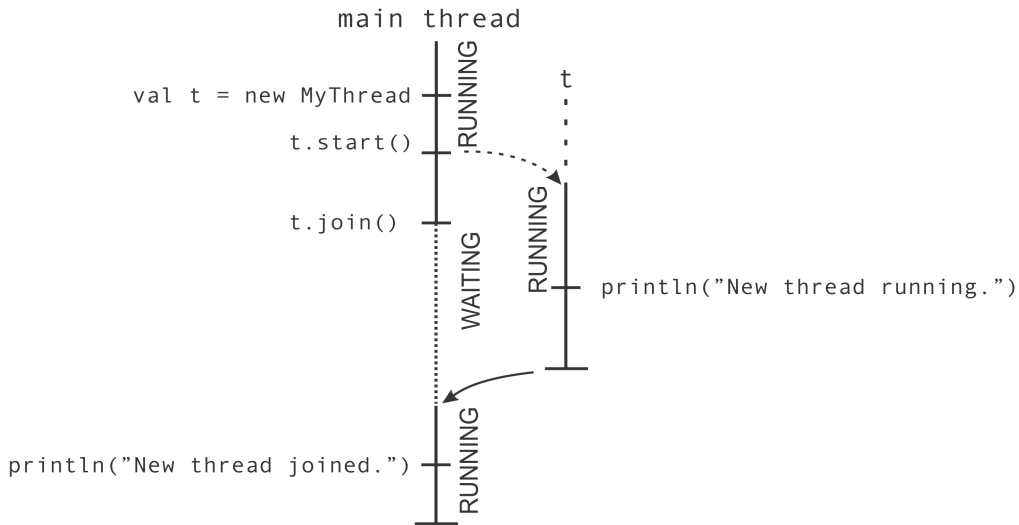
```
object ThreadsCreation extends App {  
  class MyThread extends Thread {  
    override def run(): Unit = {  
      println("New thread running.")  
    }  
  }  
  val t = new MyThread  
  t.start()  
  t.join()  
  println("New thread joined.")  
}
```

`start` eventually causes  
`run` to execute in a new thread;

the OS decides when;

`join` puts the main thread in a `waiting`  
`state`, and allows the OS to re-assign the  
processor.





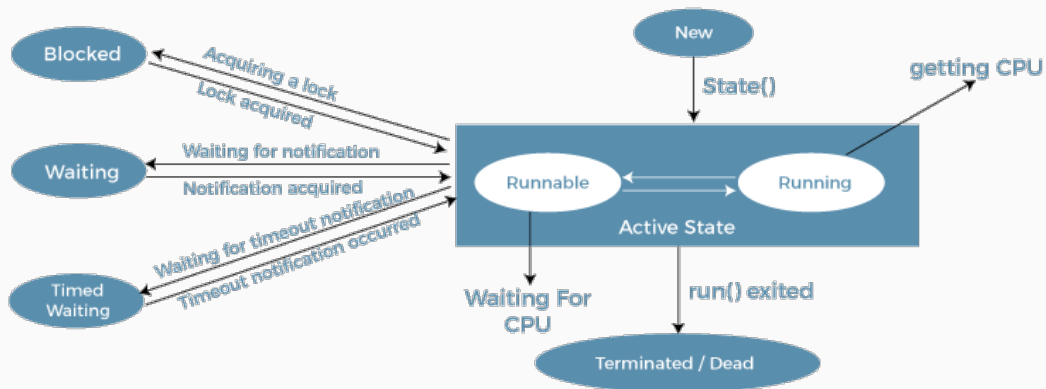
[in *"Learning Concurrent Programming in Scala"*, pg. 35]

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

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

## Using the thread function

```
object ThreadsSleep extends App {  
  val t = thread {  
    Thread.sleep(1000)  
    log("New_thread_running.")  
    Thread.sleep(1000)  
    log("Still_running.")  
    Thread.sleep(1000)  
    log("Completed.")  
  }  
  t.join()  
  log("New_thread_joined.")  
}
```



## Life Cycle of a Thread

[in <https://static.javatpoint.com/core/images/life-cycle-of-a-thread.png>]

## What to expect?

```
object ThreadsNondeterminism
  extends App {
    val t = thread {
      log("New thread running.")
    }
    log("...")
    log("...")
    t.join()
    log("New thread joined.")
  }
```

## What to expect?

```
object ThreadsNondeterminism
  extends App {
    val t = thread {
      log("New thread running.")
    }
    log("...")
    log("...")
    t.join()
    log("New thread joined.")
  }
```

- "New thread joined" printed always at the end
- Other prints not always in the same order – nondeterministic execution

```
object ThreadsNondeterminism
  extends App {
    val t = thread {
      log("New thread running.")
    }
    log("...")
    log("...")
    t.join()
    log("New thread joined.")
  }
```

## What to expect?

- "New thread joined" printed always at the end
- Other prints not always in the same order – nondeterministic execution
- Common in concurrent applications – what makes it so hard
- Note: join also forces all memory writes from the threads before proceeding

## Control of the execution order

---



action  $\alpha$  happens-before (HB) action  $\beta$   
means action  $\beta$  sees the memory writes of action  $\alpha$

- **Program order:**  $\alpha$  in a thread HB every subsequent  $\beta$  in that program and thread
- **Thread start:** calling `thrd.start()` HB any actions of `thrd`
- **Thread termination:**  $\alpha$  in a thread HB a `join()` on that thread.
- **Transitivity:** if  $\alpha$  HB  $\beta$  and  $\beta$  HB  $\gamma$ , then  $\alpha$  HB  $\gamma$
- ...

action  $\alpha$  happens-before (HB) action  $\beta$   
means action  $\beta$  sees the memory writes of action  $\alpha$

- **Program order:**  $\alpha$  in a thread HB every subsequent  $\beta$  in that program and thread
- **Thread start:** calling `thrd.start()` HB any actions of `thrd`
- **Thread termination:**  $\alpha$  in a thread HB a `join()` on that thread.
- **Transitivity:** if  $\alpha$  HB  $\beta$  and  $\beta$  HB  $\gamma$ , then  $\alpha$  HB  $\gamma$
- ...

**Data race:** when a write to memory does not happen-before its *intended* read.

- `join` provides guarantees that other threads terminated
- Not enough – we may want to inform other threads without terminating

## Example 1: shared counter for unique IDs

```
object ThreadsUnprotectedUid extends App {  
  var uidCount = 0L  
  def getUniqueId() = {  
    val freshUid = uidCount + 1  
    uidCount = freshUid  
    freshUid  
  }  
  ...  
}
```

What can go wrong?

```
...  
def printUniqueIds(n: Int): Unit = {  
  val uids = for (i<- 0 until n)  
    yield getId()   
  log(s"Generated uids: $uids")  
}  
val t = thread { printUniqueIds(5) }  
printUniqueIds(5)  
t.join()  
...
```

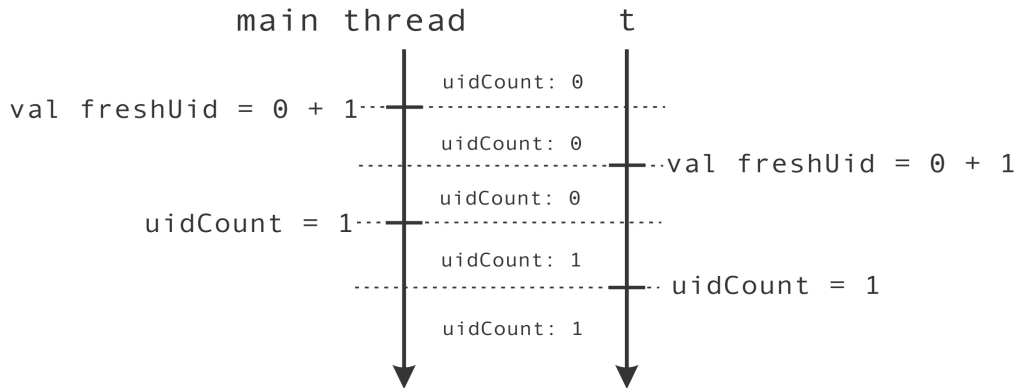
What do you expect?

```
...  
def printUniqueIds(n: Int): Unit = {  
  val uids = for (i<- 0 until n)  
    yield getId()   
  log(s"Generated uids: $uids")  
}  
val t = thread { printUniqueIds(5) }  
printUniqueIds(5)  
t.join()  
...
```

## Race Condition

when the output of a concurrent program depends on how the statements are scheduled.

```
val freshUid = uidCount + 1 ; uidCount = freshUid ; freshUid
```



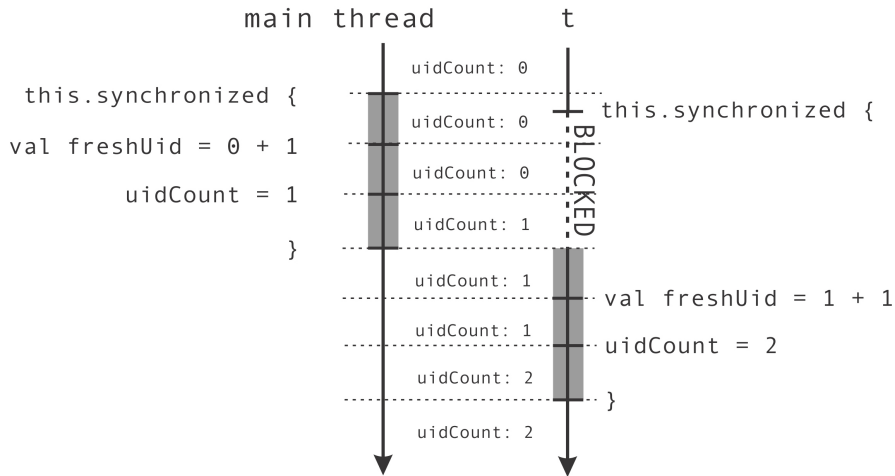
[in "Learning Concurrent Programming in Scala", pg. 40]

```
def getId() =  
  this.synchronized {  
    val freshId = uidCount + 1  
    uidCount = freshId  
    freshId  
  }
```

synchronized is:

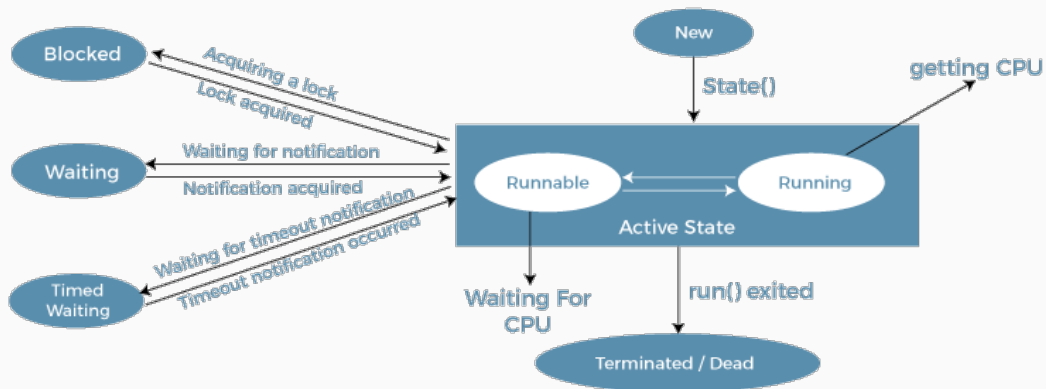
- a fundamental Scala/Java construct for atomic executions
- can be called in **any object** (or instance of a class)
- ensures atomic execution wrt the object
- we say `obj.synchronized`
  - **acquires** the **lock/monitor** of **obj** at the start
  - **releases** the **lock/monitor** of **obj** at the end

# Updating counter in parallel atomically



[in *"Learning Concurrent Programming in Scala"*, pg. 41]





### Life Cycle of a Thread

[in <https://static.javatpoint.com/core/images/life-cycle-of-a-thread.png>]

- using the synchronized statement has some (not too large) overhead
- not using synchronized can easily lead to errors, even if all seems correct

Find the bug in the next slide...

```
object ThreadSharedStateAccessReordering extends App {  
  for (i <- 0 until 100000) {  
    var a = false  
    var b = false  
    var x = -1  
    var y = -1  
    val t1 = thread {  
      a = true  
      y = if (b) 0 else 1  
    }  
    val t2 = thread {  
      b = true  
      x = if (a) 0 else 1  
    }  
    t1.join()  
    t2.join()  
    assert(!(x==1 && y==1), s"x=$x, y=$y")  
  }  
}
```

- The previous code can raise an error: both x and y can become 1!
- JVM can reorder statements in a thread when they seem to be independent.
- Because some processors do not always execute instructions in the expected order, to increase performance.
- (Known as “weak memory model”)
- A synchronized block would solve this:
  - also enclosing each assignment in a synchronized block
  - synchronized sets up a *memory barrier*

- every object has a *lock*
- a running **thread** can **acquire** multiple locks from different objects

## Example 2: Logging Bank Transfers

```
object SynchronizedNesting extends App {  
  import scala.collection._  
  
  private val transfers = mutable.ArrayBuffer[String]()  
  def logTransfer(name: String, n: Int) = transfers.synchronized {  
    transfers += s"transfer to account '$name' = $n"  
  }  
  class Account(val name: String, var money: Int)  
  def add(account: Account, n: Int) = account.synchronized {  
    account.money += n  
    if (n > 10) logTransfer(account.name, n)  
  }  
  ...  
}
```

```
private val transfers = mutable.ArrayBuffer[String]()
def logTransfer(name: String, n: Int) = transfers.synchronized {
  transfers += s"transfer to account '$name' = $n"
}

class Account(val name: String, var money: Int)
def add(account: Account, n: Int) = account.synchronized {
  account.money += n
  if (n > 10) logTransfer(account.name, n)
}

val jane = new Account("Jane", 100)
val john = new Account("John", 200)
val t1 = thread { add(jane, 5) }
val t2 = thread { add(john, 50) }
val t3 = thread { add(jane, 70) } // will not corrupt Jane's account
t1.join(); t2.join(); t3.join()
log(s"---transfers---\n$transfers")
```

action  $\alpha$  happens-before (HB) action  $\beta$   
means action  $\beta$  sees the memory writes of action  $\alpha$

- **Program order:**  $\alpha$  in a thread HB every subsequent  $\beta$  in that program and thread
- **Thread start:** calling `thrd.start()` HB any actions of `thrd`
- **Thread termination:**  $\alpha$  in a thread HB a `join()` on that thread.
- **Transitivity:** if  $\alpha$  HB  $\beta$  and  $\beta$  HB  $\gamma$ , then  $\alpha$  HB  $\gamma$
- ...
- **Monitor locking:** unlocking HB every subsequent locking (of the same lock)

# Deadlocks

---



## Deadlock

when two or more executions wait for each other before proceeding

- Studied in the first module with prof. Nelma Moreira
- Often caused by locks that are not released at the right time

```
object SynchronizedDeadlock extends App {  
  import SynchronizedNesting.Account  
  def send(a: Account, b: Account, n: Int) = a.synchronized {  
    b.synchronized {  
      a.money -= n  
      b.money += n  
    }  
  }  
  ... // can this go wrong?  
}
```

```
def send(a: Account, b: Account, n: Int) = a.synchronized {  
  b.synchronized {  
    a.money -= n  
    b.money += n  
  }  
}  
  
val l = new Account("Lucy", 1000)  
val j = new Account("Jim", 2000)  
val t1 = thread { for (i<- 0 until 100) send(l, j, 1) }  
val t2 = thread { for (i<- 0 until 100) send(j, l, 1) }  
t1.join(); t2.join()  
log(s"a_ = ${a.money}, b_ = ${b.money}")
```

```
def send(a: Account, b: Account, n: Int) = a.synchronized {  
  b.synchronized {  
    a.money -= n  
    b.money += n  
  }  
}  
  
val l = new Account("Lucy", 1000)  
val j = new Account("Jim", 2000)  
val t1 = thread { for (i<- 0 until 100) send(l, j, 1) }  
val t2 = thread { for (i<- 0 until 100) send(j, l, 1) }  
t1.join(); t2.join()  
log(s"a = ${a.money}, b = ${b.money}")
```

It works but...

```
def send(a: Account, b: Account, n: Int) = a.synchronized {  
  b.synchronized {  
    a.money -= n  
    b.money += n  
  }  
}  
  
val l = new Account("Lucy", 1000)  
val j = new Account("Jim", 2000)  
val t1 = thread { for (i<- 0 until 100) send(l, j, 1) }  
val t2 = thread { for (i<- 0 until 100) send(j, l, 1) }  
t1.join(); t2.join()  
log(s"a = ${a.money}, b = ${b.money}")
```

It works but... it can deadlock

- always acquire locks in the same order
- need a total order on locks
- we can use the `getUniqueId` (Example 1)

```
import SynchronizedProtectedUid.getUniqueId
class Account(val name: String, var money: Int) {
  val uid = getUniqueId()
}
```

- always acquire locks in the same order
- need a total order on locks
- we can use the `getUniqueId` (Example 1)

```
import SynchronizedProtectedUid.getUniqueId
class Account(val name: String, var money: Int) {
  val uid = getUniqueId()
}
```

```
def send(a1: Account, a2: Account, n: Int) {
  def adjust() {
    a1.money -= n
    a2.money += n
  }
  if (a1.uid < a2.uid) a1.synchronized{ a2.synchronized{ adjust() }}
  else                 a2.synchronized{ a1.synchronized{ adjust() }}
}
```

## Guarded blocks

---

## Guarded block (for us)

a **block of code** that **waits for a condition** before running in a thread

### Example 3: Thread pool with a queue of tasks

- Creating **new threads** in Java is **expensive** and **avoidable**
- Usually we re-use threads, by maintaining a set of waiting threads
- This set is called a thread pool
  - Scala already provides thread pools
  - We first create our own



```

import scala.collection._
object SynchronizedBadPool extends App {
  // our set of tasks
  private val tasks = mutable.Queue[()=>Unit]()

  // our single working thread
  val worker = new Thread {
    def poll(): Option[()=>Unit] =
      tasks.synchronized {
        if (tasks.nonEmpty) Some(tasks.dequeue())
        else None
      }
    // keep on trying to run forever!
    override def run() = while (true)
      poll() match {
        case Some(task) => task()
        case None =>
      }
  }
}

```

```

// starting the worker as
  a daemon
worker.setName("Worker")
worker.setDaemon(true)
worker.start()

def asynchr(body: =>Unit) =
  tasks.synchronized {
    tasks.enqueue(()=>body)
  }

asynchr{ log("Hello") }
asynchr{ log("└world!") }
Thread.sleep(5000)
}

```

## Daemon thread

- not the default
- have lower priority
- terminated automatically when JVM terminates
- in other words, do not prevent the JVM from terminating
- (the JVM terminates when 'normal' tasks terminate)

## Busy-waiting is bad

- needlessly uses processor power (and drains the battery)
- after executing the previous code the worker will keep on running (unless you set in SBT `set fork := true,`)
- in general, we want the worker to enter a waiting state

## synchronized + wait + notify

- these are methods that every Java/Scala object has
- **wait**:
  - needs the lock
  - puts the thread in a **waiting** state
  - releases the lock until activation
- **notify**:
  - needs the lock
  - **activates** all waiting threads

## synchronized + wait + notify

- these are methods that every Java/Scala object has
- **wait**:
  - needs the lock
  - puts the thread in a **waiting** state
  - releases the lock until activation
- **notify**:
  - needs the lock
  - **activates** all waiting threads
- Note that the JVM can decide to call **wait** on its own – **spurious wakeups** – needing to re-enter the *wait*

```
object SynchronizedGuardedBlocks extends App {  
  val lock = new AnyRef  
  var message: Option[String] = None  
  val greeter = thread {  
    lock.synchronized {  
      while (message == None) lock.wait() // non-busy waiting for a message  
      log(message.get) // it will eventually log!  
    }  
  }  
  lock.synchronized {  
    message = Some("Hello!")  
    lock.notify() // awakes the (possibly) locked thread  
  }  
  greeter.join()  
}
```

## Example 3 – without busy-waiting



```
import scala.collection._

object SynchronizedPool extends App {
  private val tasks = mutable.Queue[()=>Unit]()

  object Worker extends Thread {
    setDaemon(true)
    def poll() = tasks.synchronized {
      while (tasks.isEmpty) tasks.wait()
      // now using wait

      tasks.dequeue()
    }
    override def run() = while (true) {
      val task = poll()
      task()
    }
  }
}
```

```
Worker.start()

def asynchr(body: =>Unit) =
  tasks.synchronized {
    tasks.enqueue(()=>body)
    // now notifying
    tasks.notify()
  }

asynchr{ log("Hello") }
asynchr{ log("_world!") }
Thread.sleep(500)
}
```

- Our Worker can run forever (`while-true`)
- Terminates when the JVM terminates (daemon)
- Worker can be terminated earlier while waiting with `Worker.interrupt()`



- Our Worker can run forever (while-true)
- Terminates when the JVM terminates (daemon)
- Worker can be terminated earlier while waiting with `Worker.interrupt()`
- This triggers:
  - If it was waiting: an `InterruptedException` that can be handled
  - If it was not waiting: no exception and a flag `Worker.isInterrupted` becomes true

- Our Worker can run forever (while-true)
- Terminates when the JVM terminates (daemon)
- Worker can be terminated earlier while waiting with `Worker.interrupt()`
- This triggers:
  - If it was waiting: an `InterruptedException` that can be handled
  - If it was not waiting: no exception and a flag `Worker.isInterrupted` becomes true
- Interrupts are needed if a thread does not awake with `notify` (e.g., it is doing blocking I/O)

```
object Worker extends Thread {  
  var terminated = false  
  // "manually" terminate when asked  
  def poll(): Option[() => Unit] = tasks.synchronized {  
    while (tasks.isEmpty && !terminated) tasks.wait()  
    if (!terminated) Some(tasks.dequeue()) else None  
  }  
  
  import scala.annotation.tailrec  
  @tailrec override def run() = poll() match {  
    case Some(task) => task(); run()  
    case None =>  
  }  
  // "manually" ask to terminate  
  def shutdown() = tasks.synchronized {  
    terminated = true  
    tasks.notify()  
  }  
}
```

- using the `@volatile` annotation
- can be [atomically read] and [atomically modified]
- mostly used as status flag
- are never reordered in a thread
- writes are immediately visible to other threads
- very cheap to read
- not enough in many situations (e.g., `getUniqueID`)
- enough for previous example – Slide 19

```
object Volatile extends App {  
  class Page(val txt: String, var position: Int)  
  
  val pages = for (i<- 1 to 5) yield  
    new Page("Na" * (100 - 20 * i) + "␣Batman!", -1)  
  @volatile var found = false  
  for (p <- pages) yield thread {  
    var i = 0 //  
    while (i < p.txt.length && !found) //  
      if (p.txt(i) == '!') { // Separate  
        p.position = i // thread  
        found = true //  
      } else i += 1 //  
  }  
  while (!found) {}  
  log(s"results:␣${pages.map(_.position)}")  
}
```

# The Java Memory Model overview

---

action  $\alpha$  happens-before (HB) action  $\beta$   
means action  $\beta$  sees the memory writes of action  $\alpha$

- **Program order:**  $\alpha$  in a thread HB every subsequent  $\beta$  in that program and thread
- **Thread start:** calling `thrd.start()` HB any actions of `thrd`
- **Thread termination:**  $\alpha$  in a thread HB a `join()` on that thread.
- **Transitivity:** if  $\alpha$  HB  $\beta$  and  $\beta$  HB  $\gamma$ , then  $\alpha$  HB  $\gamma$
- **Monitor locking:** unlocking HB every subsequent locking (of the same lock)
- **Volatile fields:** writing to a volatile field HB every of its subsequent read

## Data race

When a write to memory does not happen-before its *intended* read.

```
class Foo( final val a: Int,
          val b: Int,
          c: Int)

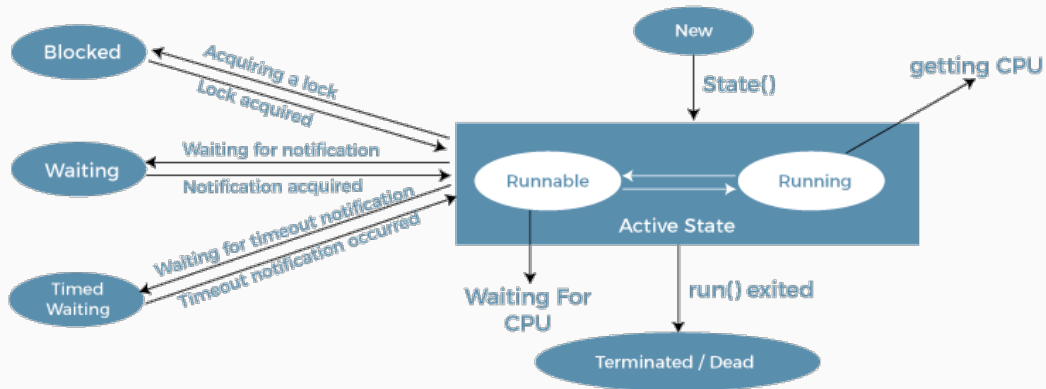
// Encoding as Java:
class Foo {
    final private int a$;
    final private int b$;
    final private int c$;
    final public int a()
    { return a$; }
    public int b()
    { return b$; }
    public Foo(int a,
               int b,
               int c) {
        { a$ = a; b$ = b; c$ = c; }
    }
}
```

- **Final** fields: cannot be **overridden**
- **val**: cannot be **updated**
- **vals** are **final**
- Objects with only final fields
  - **do not need synchronisation** when shared (after constructed)
- Some collections are immutable (e.g. List), but contain non-final fields
  - **need synchronisation** when shared



- `Thread.sleep`
- `thr.start`
- `thr.join`
- `lock.synchronized`
- `lock.wait`
- `lock.notify`
- `thr.interrupt()`
- `thr.isInterrupted`
- `@volatile var x`

# Recall the Life-cycle of a thread



Life Cycle of a Thread

[in <https://static.javatpoint.com/core/images/life-cycle-of-a-thread.png>]