Les acteurs

Scala par l'exemple

F. Dagnat

3e année, Filière SLR – UV IDL – CX

année 2016-2017

Plan

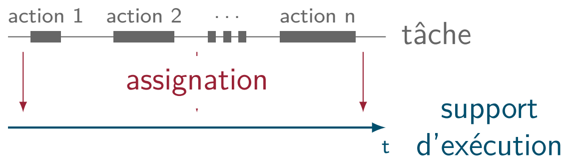
1. Concurrence à la Java
2. Les acteurs
3. Conclusion

Avancement

1. Concurrence à la Java
2. Les acteurs
3. Conclusion

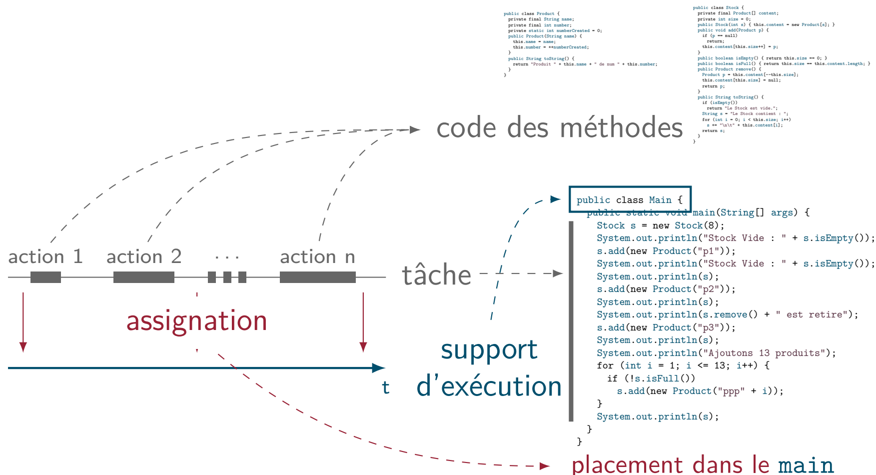
De la spécification à l'exécution

Un unique support d'exécution



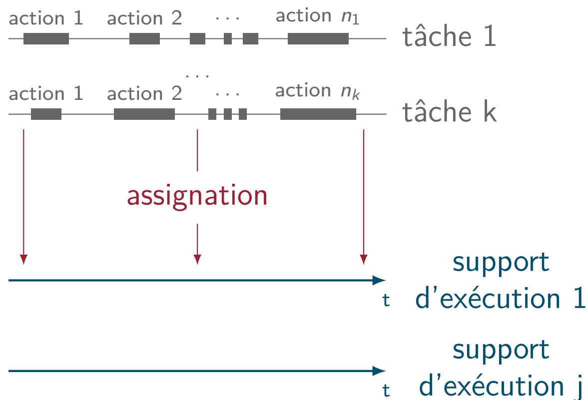
De la spécification à l'exécution

Un unique support d'exécution



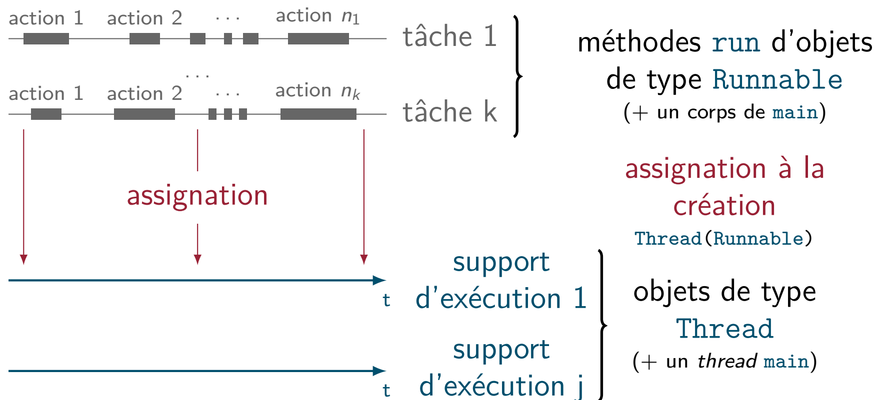
De la spécification à l'exécution

Plusieurs supports d'exécution

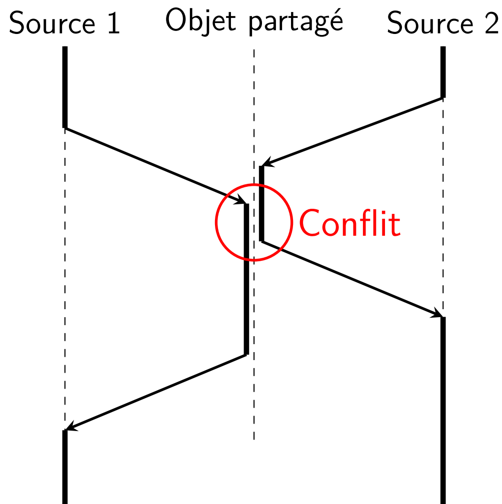


De la spécification à l'exécution

Plusieurs supports d'exécution

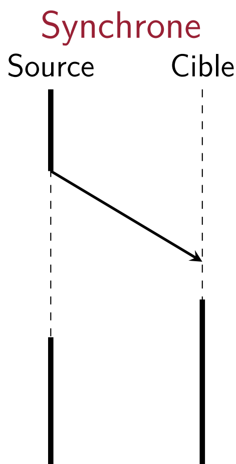


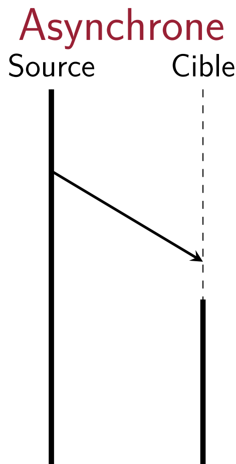
Les objets et la concurrence



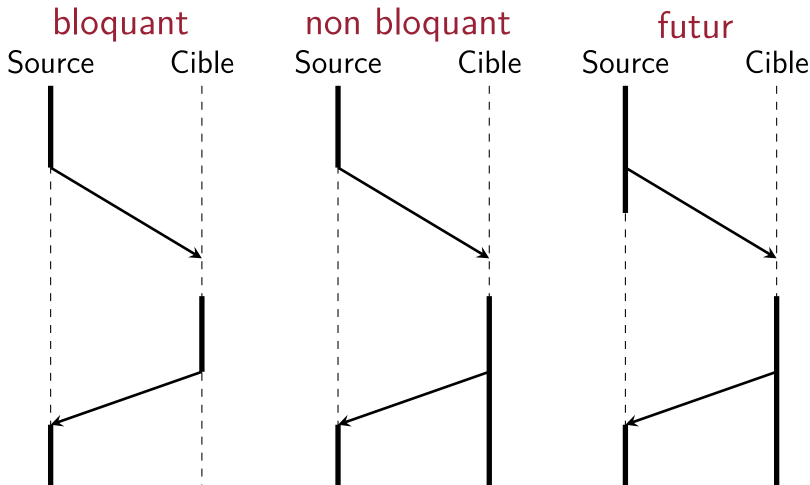
- Deux choix
  1. Les objets protégés
  2. Les objets actifs

Les différentes formes de communications





RPC – *Remote Procedure Call*



Thread, comme d'hab

```
import Math._
class SimpleThread(name: String) extends Thread(name) {
  override def run() = {
    // Je m'endors 10 fois entre 0 et 1 seconde...
    for (i <- List.range(1, 11)) {
      println(getName() + " : " + i)
      Thread.sleep((random * 1000).toLong)
    }
    println(getName() + " a fini!")
  }
}
object MainForSTE extends App {
  for (i <- List.range(1, 6))
    new SimpleThread("Thread " + i).start()
}
```

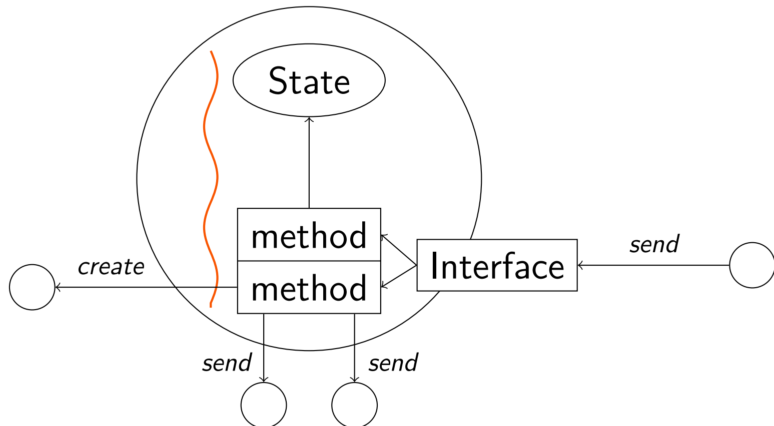
Avancement

1 Concurrence à la Java

2 Les acteurs

3 Conclusion

Les acteurs (objet actif asynchrone)



- Comportement réactif à la reception de messages
- Les acteurs ont des boîtes aux lettres

Un premier exemple

```
import akka.actor.Actor
import akka.actor.ActorDSL
        }
}
case class Inc(amount: Int)
case object Value
class Counter extends Act {
  var counter: Int = 0;
  override def receive = {
        case Inc(amount) => counter += amount
        case Value => println("Value is " + counter); context.stop(self)
  }
}
object ActorTest extends App {
        val system = ActorSystem("test")
        val counter = ActorDSL.actor(system)(new Counter)
        for (i <- 0 until 100000) counter ! Inc(1)
        counter ! Value // prints: Value is 100000
}
```


TELECOM Bretagne

Un deuxième exemple

```
import akka.actor.ActorDSL._
import akka.actor.ActorSystem
class Bof extends Act {
    override def receive = {
        case s: String => println("I receive a String: " + s)
        case i: Int => println("I receive an Int: " + i.toString)
        case _ => println("Unknown")
      }
}
object TestActors {
        val system = ActorSystem("test")
        val fussyActor = ActorDSL.actor(system)(new Bof)
        def main(args: Array[String]): Unit = {
                fussyActor ! "hello"; fussyActor ! -5; fussyActor ! 'a'
```

- `react` ne possède pas un *thread* mais en utilise un quelconque libre (ou attend)

Envoyeur et réponse

- Accès à l'envoyeur du message reçu (`sender`)
- Il peut être modifié par `send(msg,replyTo)`
- On peut répondre à un message

```
receive {
  case Msg(value) =>
    val r = process(value)
    reply(Response(r))
}
```

- Du coup l'envoyeur

```
myService ! Msg(value)
receive {
  case Response(r) => ...
}
```

```
myService !? Msg(value) match {
  case Response(r) => ...
}
```

Avancement

1. Concurrence à la Java
2. Les acteurs
3. Conclusion

Bilan sur les acteurs

- Les acteurs assurent l'isolation
  - attention à ne pas partager des éléments mutables (état de l'art, les messages)
- La synchronisation se fait par des protocoles (des formes de message)
- Rien n'empêche les famines ou les interblocages !
- Deux formes d'acteur : les objets actifs (`receive`) et ceux qui sont basés évenement (`react`)
  - objet actif lorsque l'acteur a un état
- Un *framework* plus sophistiqué : Akka
  - utilisable en Java ou scala avec des meilleures performances
  - supporte les acteurs typés (objets actifs) et la distribution