SCALA AND AKKA IN ONE HOUR

INGEGNERIA DEL SOFTWARE

Università degli Studi di Padova

Dipartimento di Matematica

Corso di Laurea in Informatica, A.A. 2015 – 2016

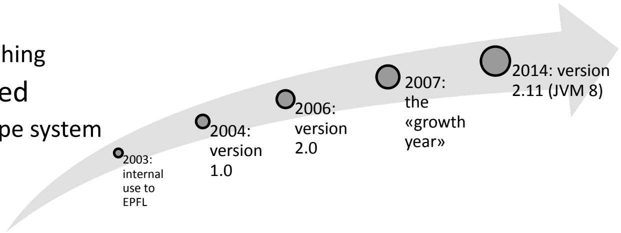
rcardin@math.unipd.it

SOMMARIO

- Introduction
- Object oriented Scala
- Actor model
- Akka

INTRODUCTION

- Object/functional programming language
  - Based on Java Virtual Machine
    - Full interoperability with Java libraries
  - Pure object orientation (no primitive types)
  - Every function is a value
    - High order functions (lambdas)
    - Promoting immutability
    - Currying
    - Pattern matching
  - Statically typed
    - Intelligent type system
  - Extensible



2003: internal use to EPFL

2004: version 1.0

2006: version 2.0

2007: the «growth year»

2014: version 2.11 (JVM 8)

INTRODUCTION

- Language main features
  - Read-Eval-Print-Loop (REPL) interpreter
    - An interactive shell to execute statements "on the fly"
  - Variables
    - UML syntax -> `name : type`
    - `val x: Int = 1 + 1` Immutable
    - `lazy val x: Int = 1 + 1` Immutable and lazy
    - `var x: Int = 1 + 1` Mutable
    - `def x: Int = 1 + 1` Executed every time
  - Very smart type inferer `val x = 1` *`x: Int`*
  - No semicolon
  - Blocks, blocks everywhere...

INTRODUCTION

- Language main features
  - Every statement is a "pure" expression
    - No `return` statement needed!
  - The `if-else` statement is a value

```
val y = 4
// x has type String
val x = if (y < 3) "minor" else "greater"
```

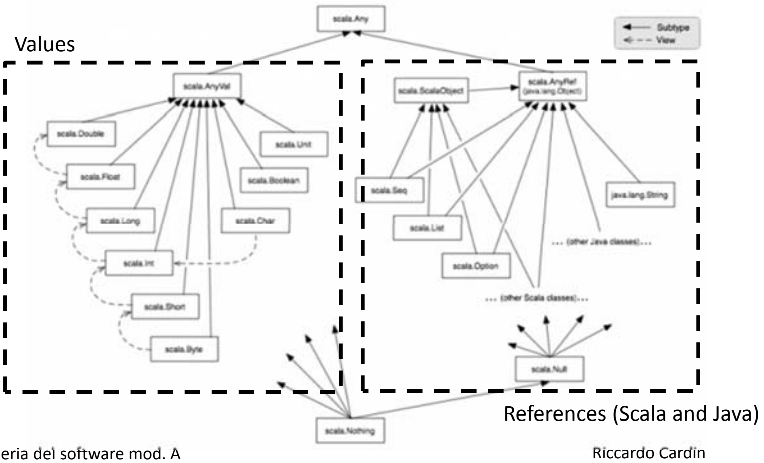
  - Imperative `for` statement is not supported
    - Use the `for-yield` statement instead

```
// Returns an array of integers between 2 and 101
for (i <- (1 to 100)) yield i + 1
```

  - Every char can be used to define names (%$#@?...)
  - Full integration with every Java library

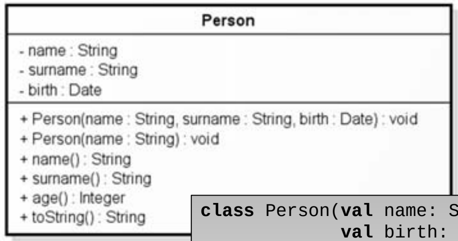
OBJECT ORIENTED SCALA

- Everything is an object



OBJECT ORIENTED SCALA

- Classes



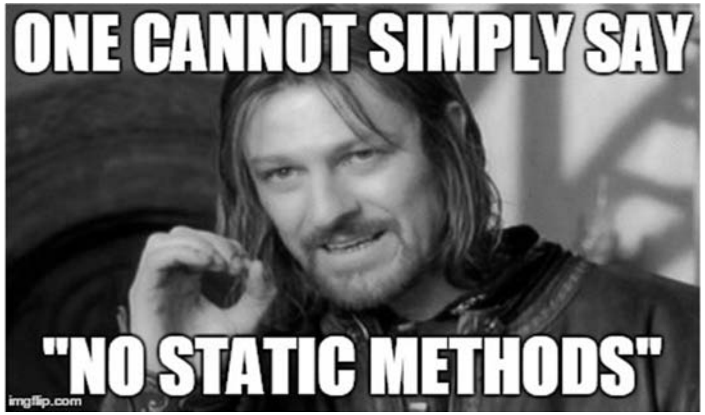
```
class Person(val name: String, val surname: String,
             val birth: Date) {
  // Main Ctor body here

  // Every secondary Ctor must call the main Ctor as
  // first statement (very restrictive)
  def this(name: String) = this(name, "", null)
  def age(): Int = { /* return the age */ }
  // Override intention must be declared
  override def toString = s"My name is $name $surname"
}
```

OBJECT ORIENTED SCALA

- Classes
  - Main constructor is declared within Class name
    - Ctor body is made of statements in class body
  - Accessor methods are build automatically
    - `val` →immutable value, getter only
    - `var` → mutable value, getter and setter
  - `def` defines methods (and functions too...)
    - Use equal "=" iff the method is a function
  - No public class
    - You can have more than a class into a `.scala` file
  - No static methods (wait a minute...)

OBJECT ORIENTED SCALA



- ...so let's introduce objects notation!

OBJECT ORIENTED SCALA

- Object
  - Native Singleton pattern implementation

```
object ChuckNorris extends Person("Chuck Norris") {
  // This is a static method
  def roundhouseKick = "Roundhouse"
}
println ChuckNorris.roundhouseKick
```

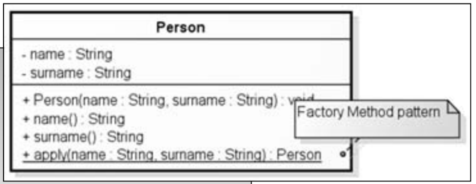
    - There can be only one instance of `ChuckNorris`
  - *Companion* object
    - An object called as an existing class
    - *Companion* object's methods are static method of the class

```
class Person(val name: String, val surname: String)
object Person {   // Companion object
  def someStupidStaticMethod = "This is a static method"
}
println Person.someStupidStaticMethod
```

OBJECT ORIENTED SCALA

- Object
  - *Companion* objects implement also Factory Method pattern natively
    - Every `apply` method creates an instance of the class
      - Syntactic sugar: `Class.apply(...)` → `Class(...)`
      - It's like a class application (...function anyone?)

There can be more than one apply!!!!


Person
- name : String
- surname : String
+ Person(name : String, surname : String) : void
+ name() : String
+ surname() : String
+ apply(name : String, surname : String) : Person

Factory Method pattern

```
class Person(val name: String,
           val surname: String)
object Person {
  // Factory method
  def apply(n: String, s: String) =
    new Person(n, s)
}
// Application of the class will return a new instance
val ricky = Person("Riccardo", "Cardin")   // No new operator
```

OBJECT ORIENTED SCALA

- Object
  - Used also to define the entry point of an application
    - The Java way

```
object Hello {
  // Every object containing a main method could be
  // Executed. There are no restriction on file name
  def main(args: Array[String]) {
    println("Hello World!")
  }
}
```

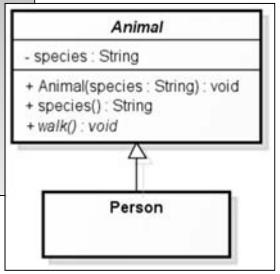
    - The Scala way

```
object Hello extends App {
  // All the code in the object's ctor body will be
  // executed as our application.
  // Arguments are accessible via an 'args' variable,
  // available in the App trait
  println("Hello World!")
}
```

OBJECT ORIENTED SCALA

- Abstract classes
  - Also defines a main constructor
    - It has to be called from the derived classes' main Ctor

```
abstract class Animal(val species: String) {
  def walk   // Abstract method
}
class Person(val name: String,
             val surname: String)
// Calling main ctor directly in class
// definition
extends Animal("Human") {
  def walk = { /* Do some stuff */ }
}
val ricky = new Person("Riccardo", "Cardin")
```



OBJECT ORIENTED SCALA

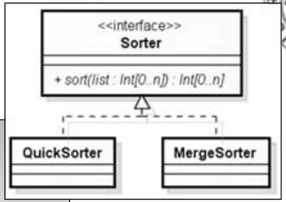
- Traits: interfaces or abstract classes?
  - Like interfaces, they have not a constructor
    - To implement a single trait, use `extends`
    - Otherwise, use `with` (*mixin*)
  - But, they can have methods with implementation and attributes
    - Similar (but more powerful) to *default* methods in Java 8
  - Multiple inheritance problem

```
trait A { override def toString = "I’m A" }
trait B { override def toString = "I’m B" }
trait C { override def toString = "I’m C" }
class D extends A with B, C
new D().toString    // Prints ‘I’m C’
```

- Class linearization: use the rightmost type (more or less...)

OBJECT ORIENTED SCALA

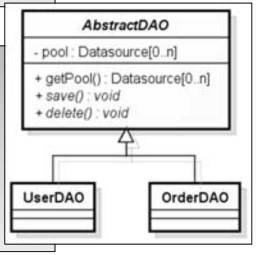
- Traits
  - Like an interface



```
// No implementation at all
trait Sorter {
  def sort(list: List[Int]): List[Int]
}
class QuickSorter extends Sorter {
 def sort(list: List[Int]) = { /* Do something */ }
}
```

- Like an abstract class

```
trait AbstractDao {   // No Ctor (wtf!)
  var pool: Array[Datasource]
  def getPool = pool // Implemented method
  def save           // abstract method
  def delete         // abstract method
}
class UserDao extends AbstractDao {
  // Has to implement save and delete
}
```



OBJECT ORIENTED SCALA

- Traits
  - Inside a *mixin* the `super` refers to the previuos trait in the linearized-type
    - Used to implement the Decorator pattern natively

```
trait Employee {
  // ...
  def whois(): String
}
class Engineer(name: String, office: String) extends Employee

trait ProjectManager extends Employee {
  abstract override def whois() {
    // super refers to the previuos trait on the left
    super.whois(buffer)
    println("and I’am a project manager too!")
  }
}
new Engineer("Riccardo","Development") with ProjectManager
```

Statically binded Decorator pattern

OBJECT ORIENTED SCALA

- Traits and Objects
  - Using the *companion* object of a trait, we can implement the Abstract Factory pattern

```
// Abstract factory
trait AbstractFactory
// Private classes’ scope is limited to the
// definition file
private class ConcreteFactory1 extends AbstractFactory { /* ... */ }
private class ConcreteFactory2 extends AbstractFactory { /* ... */ }

object AbstractFactory{
  def apply(kind: String) =
    kind match {
      case “factory1" => new ConcreteFactory1 ()
      case “factory2" => new ConcreteFactory1 () factory 2
    }
}
val factory = AbstractFactory("factory1")
```

OBJECT ORIENTED SCALA

- Case classes and pattern matching
  - Scala has a built in pattern matching mechanism

```
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case _ => "many"
}
println(matchTest(3))   // Prints ‘many’
```

Case clauses are evaulated from top to bottom

  - Case class: regular class that can be pattern-matched
    - Turns all ctor params into `val` (immutable) constants
    - Generates a *companion object* w/ `apply` methods
    - Generates `toString`, `equals` and `hashCode` methods properly

```
trait Term
case class Var(name: String) extends Term
case class Fun(arg: String, body: Term) extends Term
Var("x") == Var("x")    // Prints true
```

Checks the equality among attributes

OBJECT ORIENTED SCALA

- Case classes and pattern matching
  - Native implementation of Value Object pattern
  - Obviously, case classes can be used in pattern matching...
    - The match is done recursively on the structure of the type

```
trait Term
case class Var(name: String) extends Term
case class Fun(arg: String, body: Term) extends Term
def printTerm(term: Term) { term match {
    // The clauses uses placeholders for variables.
    // If we don’t need the value, we can use Var(_)
    case Var(n) =>
      print(“We have a variable!”)
    case Fun(x, b) =>
      print(“We have a function!”)
  }
}
```

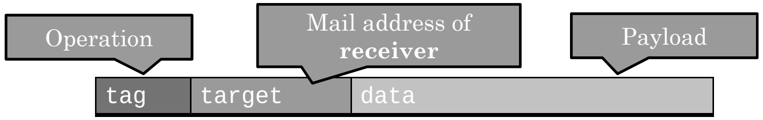
ACTOR MODEL

> *Each actor is a form of reactive object, executing some computation in response to a message and sending out a reply when the computation is done*
>
> -- John C. Mitchell

- Reactive
  - Execution are in response of external events
    - Two possible states: sleep / awake
    - There is no the explicit concept of thread
- An actor can perform basically three actions
  - Sending async messages to another actor or to itself
  - Create new actors
  - Modify its interface (no mutable state)

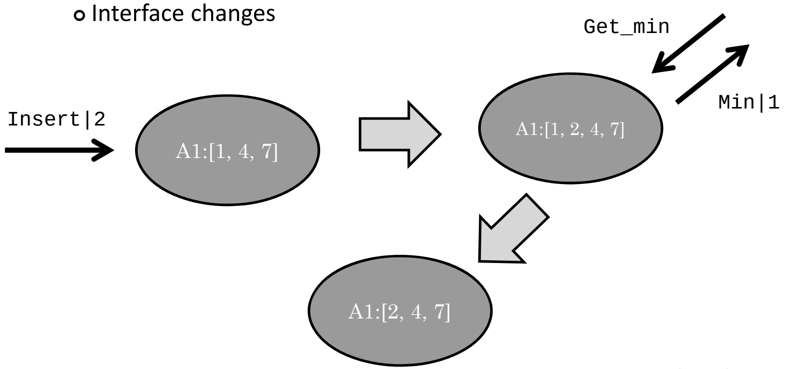
ACTOR MODEL

- Messages (tasks)
  - They represent the interface of the actor
    - Variable through time
  - Messages are read one by one
    - Each actor has an associated mail box (queue)
    - No guarantee on the order of reception of messages
  - Mail system
    - Each actor has an associated mail address
  - Messages are divided in three parts



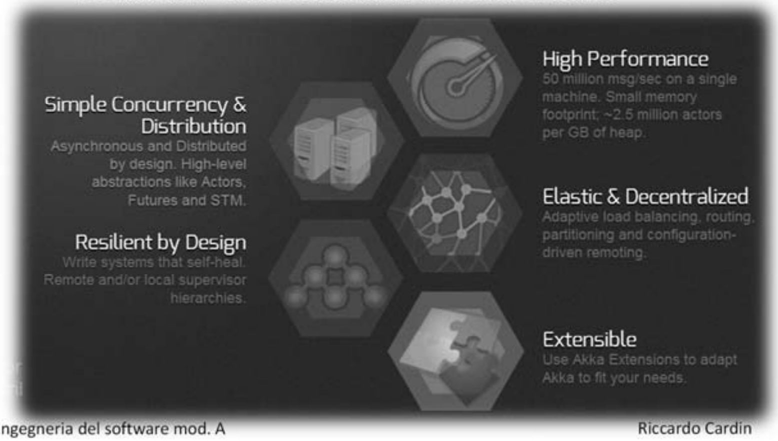
ACTOR MODEL

- Example
  - A state change may corresponds to a new actor
    - No race conditions: state of an actor in invisibile from outside
    - Interface changes



ACTOR MODEL

- Akka
  - Toolkit and runtime for actor model on JVM



Simple Concurrency & Distribution
Asynchronous and Distributed by design. High-level abstractions like Actors, Futures and STM.

Resilient by Design
Write systems that self-heal. Remote and/or local supervisor hierarchies.

High Performance
50 million msg/sec on a single machine. Small memory footprint; ~2.5 million actors per GB of heap.

Elastic & Decentralized
Adaptive load balancing, routing, partitioning and configuration-driven remoting.

Extensible
Use Akka Extensions to adapt Akka to fit your needs.

AKKA

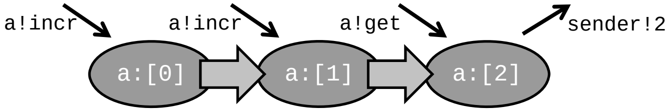
```
type Receive = PartialFunction[Any, Unit]
trait Actor {
  def receive: Receive  // Actor actual behavior
  implicit val self: ActorRef  // Reference to itself
  def sender: ActorRef  // Reference to the sender of last message
  implicit val context: ActorContext  // Execution context
}
abstract class ActorRef {
  // Send primitives
  def !(msg: Any)(implicit sender: ActorRef = Actor.noSender): Unit
  def tell(msg: Any, sender: ActorRef) = this.!(msg)(sender)
  // ...
}
trait ActorContext {
  // Change behavior of an Actor
  def become(behavior: Receive, discardOld: Boolean = true): Unit
  def unbecome(): Unit
  // Create a new Actor
  def actorOf(p: Props, name: String): ActorRef
  def stop(a: ActorRef): Unit // Stop an Actor
  // ...
}
```

Akka

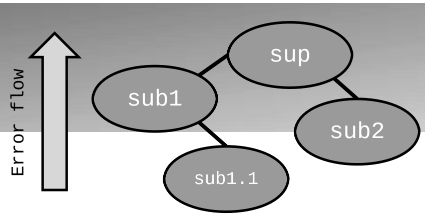
- Example
  - Distributed counter implementation using actors

```
class Counter extends Actor {
  // State == explicit behavior
  def counter(n: Int): Receive = {
    // Receive two types of messages: ‘incr’ and ‘get’
    // ‘incr’ change actor’s behavior
    case ”incr” => context.become(counter(n + 1))
    // ‘get’ returns current counter value to sender
    case ”get” => sender ! n
  }
  def receive = counter(0)  // Default behavior
}
```

Internal state can be modelled using *closures*



Akka

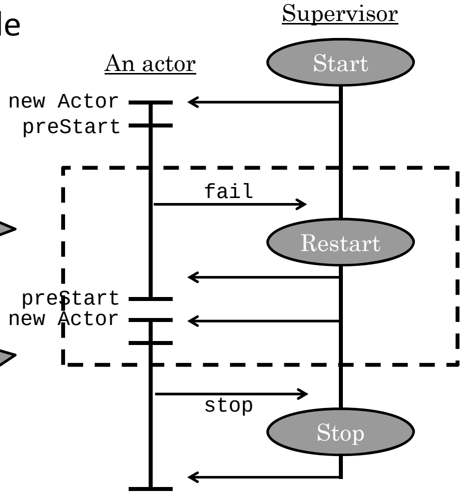


- Resiliency
  - Automatic containment and response to errors
    - Actor in a error state are killed or restarted
    - Decision is made by an actor of type supervisor
    - Actors that use supervision are organized in a tree structure
    - The supervisor create its subordinates

```
class Manager extends Actor {
  // OneForOneStrategy restarts only actor which died
  override val supervisorStrategy = OneForOneStrategy() {
    case _: DBException => Restart // reconnect to DB
    case _: ActorKilledException => Stop
    case _: ServiceDownException => Escalate
  }
  // ...
  context.actorOf(Props[DBActor], ”db”)
  // ...
}
```

Akka

- Actor lyfecycle



There can be multiple restarts

ActorRef remains valid between two different restarts

Refereces

- Functional Programming Principles in Scala (Coursera) https://www.coursera.org/course/progfun
- Principles of Reactive Programming (Coursera) https://www.coursera.org/course/reactive
- Scala School! http://twitter.github.io/scala_school/
- Scala for the Impatient, Cay Horstmann, Addison-Wesley 2012 http://www.horstmann.com/scala/index.html
- Getting Started Tutorial (Scala): First Chapter http://doc.akka.io/docs/akka/2.0/intro/getting-started-first-scala.html
- Akka: Scala Documentation http://doc.akka.io/docs/akka/2.3.14/scala.html?_ga=1.62489391.244507724.1445894179