Scala for C# Developers

preconditions

1. you know how to computer
2. different syntax isn't taxing
3. you'll bug me afterwards

why would a C# developer want to learn Scala?

1. curiosity

2. necessity

1. curiosity

2. necessity

Curiosity allows developers to discover new ways of doing things and broaden their ability

1. curiosity

2. necessity

As the technological landscape grows niche skills and specialisms become less important

but why scala?

why not ____

clojure | f# | java

kotlin | elixir | go

ruby | haskell | ...

or one of the other ~794 languages*

* excluding BASIC dialects & esoteric languages [WIKIPEDIA]

1. i don't know

2. ½ way up the learning curve

3. trends in industry

4. data science

so, scala...

java/.net

2004(ish)

martin odersky

epfl

typesafe

v 2.10.3

object oriented

functional

statically typed

type inference

immutable by default

interop

concurrency baked in

compile times

complexity

esoteric

mixing paradigms

syntax teardown

HelloWorld.cs

```
public class Hello1
{
    public static void Main()
    {
        string message = “Hello World”;
        System.Console.WriteLine(message);
    }
}
```

HelloWorld.scala

```
object HelloWorld extends App {
   val message = “Hello World”
   println(message)
}
```

HelloWorld.scala

```
object HelloWorld {
  def main(args: Array[String]): Unit  = {
    val message: String = “Hello World”
    println(message)
  }
}
```

HelloWorld.scala

```
object HelloWorld {
  def main(args: Array[String]) : Unit = {
    val message :String = “Hello World”
    println(message) no semicolons
  }
}
```

```
class Person
{
    private readonly string _name;
    public string name { get { return _name; } }
    public int age { get; set; }

    public Person(string name, int age)
    {
        this._name = name
        this.age = age
    }
}
```

Person.cs

```
class Person(val name: String, var age: Int)
```

Person.scala

```
class Person(val name: String, var age: Int)
```

Person.scala

```
class Person(val name: String, var age: Int)
```

Person.scala

language features

default values for functions

options vs nulls

```
def sayHello(name: String = "World") {
  println("Hello, " + name)
}

def sayHello(name: String = null) {
  println("Hello," + if(name == null) "" else "World")
}

def sayHello(name: Option[String] = None) {
  println("Hello," + name.getOrElse("World"))
}
```

```
def sayHello(name: String = “World”) {
  println(”Hello, ” + name)
}

def sayHello(name: String = null) {
  println(”Hello,” + if(name == null) ”” else ”World”)
}

def sayHello(name: Option[String] = None) {
  println(”Hello,” + name.getOrElse(”World”))
}
```

collection support

lambdas and shorthand

for comprehension

```
List(1,2,3,4)
  .map(number => number + 1)
  .filter(_ % 2 == 0)
  .foldLeft(0)(_ + _)
```

```
List(1,2,3,4)
    .map(number => number + 1)
    .filter(_ % 2 == 0)
    .foldLeft(0)(_ + _)
```

```
(1 :: 2 :: 3 :: 4 :: Nil)
  .map(number => number + 1)
  .filter(_ % 2 == 0)
  .foldLeft(0)(_ + _)
```

```
Map(
   “name”    -> ”James Hughes”,
   “handle” -> ”@kouphax”
)
```

```
Map(
   “name”    -> ”James Hughes”,
   “handle” -> ”@kouphax”
)
```

```
val attendees = List(
  Person("james", List("scala", ".net")),
  Person("will",  List("java", ".net")),
  Person("luke",  List("ios", "drinking redbull"))
)

val coolPeople = for {
  attendee <- attendees if attendee.name.startsWith(“j”);
  skills   <- attendee.skills if skills.contains("scala")
} yield attendee.name
```

```
val attendees = List(
  Person("james", List("scala", ".net")),
  Person("will",  List("java", ".net")),
  Person("luke",  List("ios", "drinking redbull"))
)

val coolPeople = for {
  attendee <- attendees if attendee.name.startsWith(“j”);
  skills   <- attendee.skills if skills.contains("scala")
} yield attendee.name
```

traits

dynamic composition

```
trait Logging {
  def log(message: String) = println(message)
}

class Worker extends TheHadoops with Logging {
  def onFinish = {
    log("Finished working")
   }
}

trait BetterLogging extends Logging {
  override def log(message: String) =
    println("[INFO] " + message)
}
```

```
trait Logging {
  def log(message: String) = println(message)
}

class Worker extends TheHadoops with Logging {
  def onFinish = {
    log("Finished working")
   }
}

trait BetterLogging extends Logging {
  override def log(message: String) =
    println("[INFO] " + message)
}
```

```
val worker = new Worker
val betterWorker = new Worker with BetterLogging

worker.work
> Finished Working

betterWorker.work
> [INFO] Finished Working
```

```
val worker = new Worker
val betterWorker = new Worker with BetterLogging

worker.work
> Finished Working

betterWorker.work
> [INFO] Finished Working
```

pattern matching

case classes and extraction

```
def wordify(number: Int) = {
  number match {
    case 1                => "One"
    case 2                => "Two"
    case n if n > 100 => "More than one hundred"
    case _                => "Between two and one hundred"
  }
}
```

```
def wordify(number: Int) = {
  number match {
    case 1                    => "One"
    case 2                    => "Two"
    case n if n > 100 => "More than one hundred"
    case _                    => "Between two and one hundred"
  }
}
```

```
def length(list: List[_]) : Int = {
  list match {
    case Nil          => 0
    case _ :: tail => 1 + length(tail)
  }
}
```

```
def length(list: List[_]) : Int = {
  list match {
    case Nil         => 0
    case _ :: tail => 1 + length(tail)
  }
}
```

```
trait Role
case class Anon() extends Role
case class User(val name: String, admin: Boolean) extends Role

def isAdmin(user: Role) = {
  user match {
    case Anon() =>
      println("anons aren't admins"); false
    case User(name, true) =>
      println(name + " is admin"); true
    case User(name, false) =>
      println(name + " is not an admin"); false
  }
}
```

```
trait Role
case class Anon() extends Role
case class User(val name: String, admin: Boolean) extends Role

def isAdmin(user: Role) = {
  user match {
    case Anon() =>
      println("anons aren't admins"); false
    case User(name, true) =>
      println(name + " is admin"); true
    case User(name, false) =>
      println(name + " is not an admin"); false
  }
}
```

implicits as extension methods

implicit scope

implicit type conversions

```
implicit class FancyString(val s: String) {
  def makeChristmasy = "***" + s + "***"
}

"CHRISTMAS".makeChristmasy
```

```
implicit class FancyString(val s: String) {
  def makeChristmasy = "***" + s + "***"
}

"CHRISTMAS".makeChristmasy
```

```
trait Logger { def log(s: String) }

class LoggerImpl extends Logger {
  def log(s: String) = println(s)
}

def inspect(s: String)(implicit logger: Logger) {
  logger.log(s)
}

implicit val defaultLogger = new LoggerImpl

inspect("Hello")
```

```scala
trait Logger { def log(s: String) }

class LoggerImpl extends Logger {
  def log(s: String) = println(s)
}

def inspect(s: String)(implicit logger: Logger) {
  logger.log(s)
}

implicit val defaultLogger = new LoggerImpl

inspect("Hello")
```

```
def log(s: String) = {
  println(s)
}

log(1) // Type error

implicit def int2String(i: Int): String = i.toString

log(1) // success
```

```
def log(s: String) = {
  println(s)
}

log(1) // Type error

implicit def int2String(i: Int): String = i.toString

log(1) // success
```

multiline strings

string interpolation

string context

```
val plain = """
This is a
multiline string
"""

val stripped = """
        | This is a
        | multiline string
        """.stripMargin
```

```
val plain = """
This is a
multiline string
"""

val stripped = """
    | This is a
    | multiline string
    """.stripMargin
```

```
def sayHi(name: String) {
  println(s"Hi $name")
}

sayHi("james")
```

```
def sayHi(name: String) {
  println(s”Hi $name”)
}

sayHi(”james”)
```

```
implicit class SQLHelper(val sc: StringContext) extends AnyVal {
  def sql(args: Any*): PreparedStatement = {
    SQLEngine.prepare(sc.s(args))
  }
}

val id = 1

val query = sql"select * from users where id = $id"
```

```
implicit class SQLHelper(val sc: StringContext) extends AnyVal {
  def sql(args: Any*): PreparedStatement = {
    SQLEngine.prepare(sc.s(args))
  }
}

val id = 1

val query = sql"select * from users where id = $id"
```

inline XML

```
def get("/:name") {

  contentType="text/html"

  <html>
   <head>
     <title>Test</title>
    </head>
   <body>
     <h1> Hello { request.param("name") }</h1>
    </body>
  </html>
}
```

```
def get("/:name") {

  contentType="text/html"

  <html>
   <head>
    <title>Test</title>
   </head>
   <body>
    <h1> Hello { request.param("name") }</h1>
   </body>
  </html>
}
```

duck typing

dynamic

```
class Logger1 { def log(s: String) = println(s) }
class Logger2 { def log(s: String) = println(s) }

def log(s: String, l: { def log(s: String) }) {
  l.log(s)
}

log("same", new Logger1)
log("same", new Logger2)
```

```
class Logger1 { def log(s: String) = println(s) }
class Logger2 { def log(s: String) = println(s) }

def log(s: String, l: { def log(s: String) }) {
  l.log(s)
}

log("same", new Logger1)
log("same", new Logger2)
```

```
import scala.language.dynamics

class Dynamap extends Dynamic {

  var map = Map.empty[String, Any]

  def selectDynamic(name: String) =
    map get name getOrElse sys.error("method not found")

  def updateDynamic(name: String)(value: Any) {
    map += name -> value
  }

  def applyDynamic(name: String)(args: Any*) = name match {
    case "clear" => map = Map.empty[String, Any]
  }
}
```

```
import scala.language.dynamics

class Dynamap extends Dynamic {

  var map = Map.empty[String, Any]

  def selectDynamic(name: String) =
    map get name getOrElse sys.error("method not found")

  def updateDynamic(name: String)(value: Any) {
    map += name -> value
  }

  def applyDynamic(name: String)(args: Any*) = name match {
    case "clear" => map = Map.empty[String, Any]
  }
}
```

```
val d = new Dynamap

d.foo = 10 // updateDynamic

d.foo      // selectDynamic

d.clear()  // applyDynamic

d.foo      // ERROR!
```

just because you

can doesn't mean

you should...

```
object SquareRoot extends Baysick {
  def main(args:Array[String]) = {

    10 PRINT "Enter a number"
    20 INPUT 'n
    30 PRINT "√ of " % "'n is " % SQRT('n)
    40 END

    RUN
  }
}
```

```scala
implicit def CokleisliCategory[M[_]: Comonad]:
  Category[({type λ[α, β]=Cokleisli[M, α, β]})#λ] =
    new Category[({type λ[α, β]=Cokleisli[M, α, β]})#λ] {
      def id[A] = ★(_ copure)

      def compose[X, Y, Z](
          f: Cokleisli[M, Y, Z],
          g: Cokleisli[M, X, Y]) = {
        f =<= g
      }
    }
```

╯(°□°)╯ ɐlɐɔs

```
def °□° = “”
def ɐlɐɔs = “”

object ノノ {
  def ノ (s: String) = "ɐlɐɔs"
}

object ノ {
  def apply(s: String) = ノノ
}
```

```
scala> ヽ(°□°)ヽ  ɐlɐɔs

res0: String = ɐlɐɔs
```

practical scalaing

1. simple app

1.1 compiled

1.2 as a script

2. sbt

2.1 scala & sbt are just jars

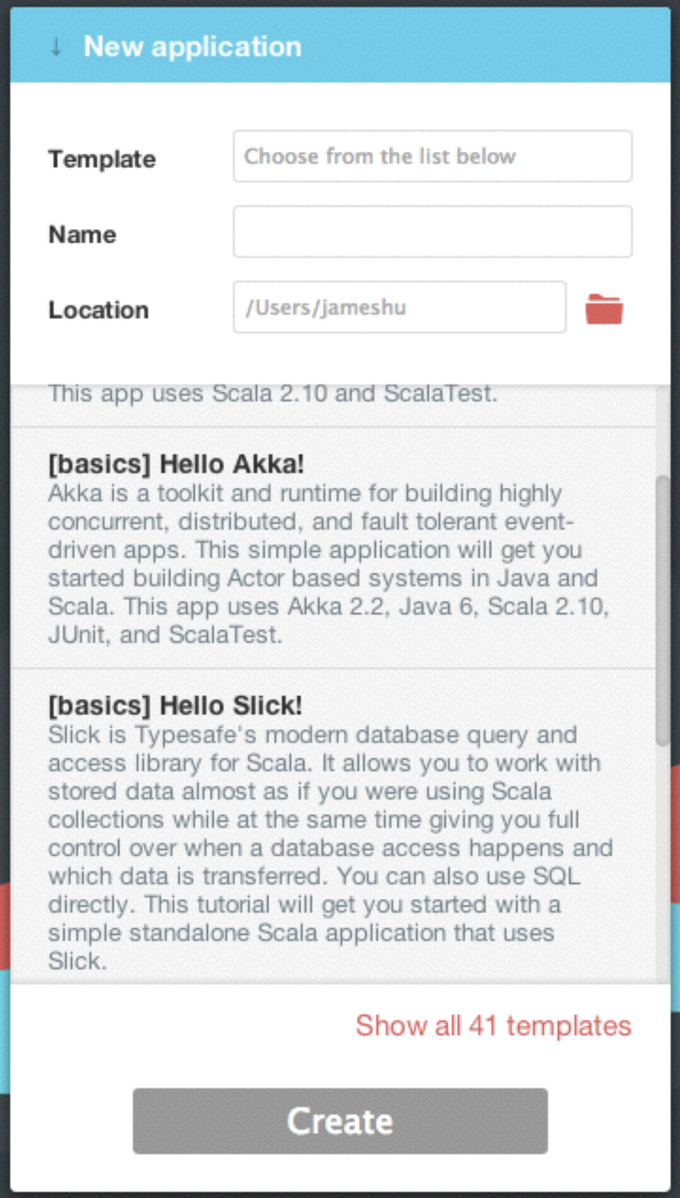
2.2 maven, gradle fine

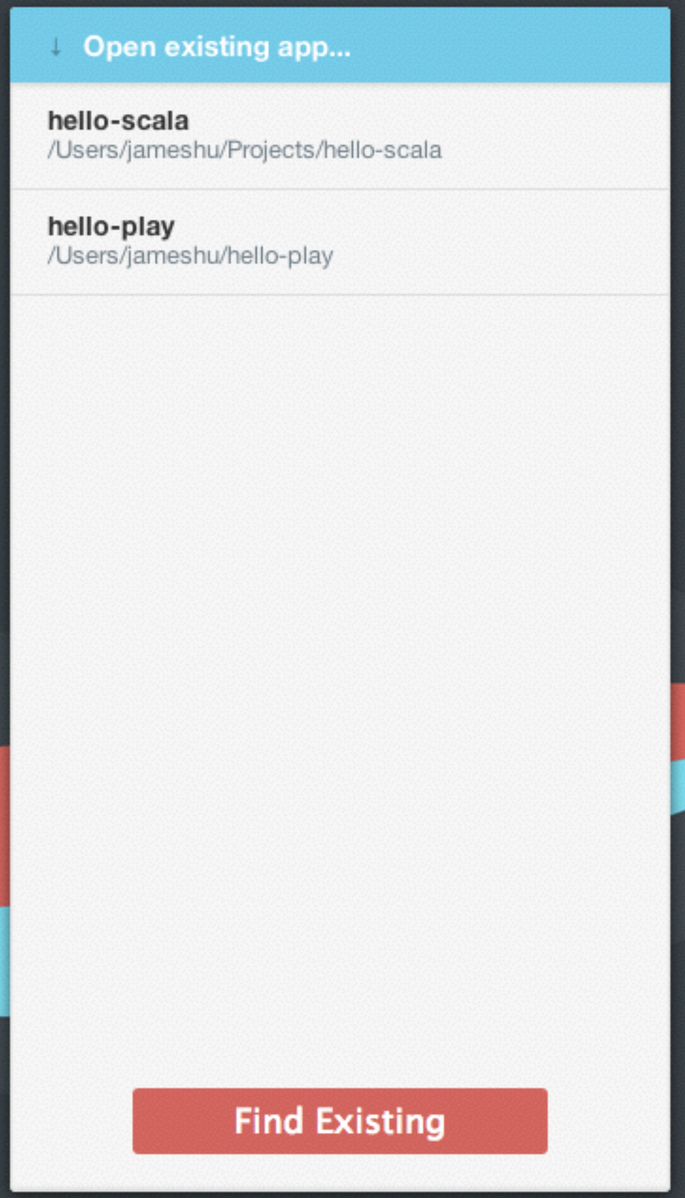
activator

http://typesafe.com/activator

Typesafe Activator

Build apps in a snap! Typesafe Activator is a local-running web application. Create new apps from templates. Browse & edit code. Run apps and tests.





Your application is being opened!

This will just take a minute...

```
Template is cloned, compiling project definition...
Loading global plugins from /Users/jameshu/.sbt/0.13/plugins/project
Loading global plugins from /Users/jameshu/.sbt/0.13/plugins
```

Home | Code | Compile | Test | Run

hello-play

/Users/jameshu/hello-play + Open

app/controllers/MainController.java delete revert save

/ app / controllers

java / 240.00 b

- MainController.java
- MessageController.scala

```java
package controllers;

import play.mvc.Controller;
import play.mvc.Result;

public class MainController extends Controller {

    public static Result index() {
        return ok(views.html.index.render("Hello from Java"));
    }

}

```

TUTORIAL

View the App

You've just created a simple Play Framework application! Now lets explore the code and make some changes.

View the App

Once the application has been compiled and the server started, your application can be accessed at: http://localhost:9000
Check in Run to see the server status.
When you make an HTTP request to that URL, the Play server figures out what code to execute in order to handle the request and return a response. In this application the request handler for requests to the root URL (e.g. "/") are handled by a Java Controller. You can use Java and Scala to create your controllers.
Controllers asynchronously return HTTP responses of any content type (i.e. HTML, JSON, binary). To see a JSON response produced by a Scala controller, click the "Get JSON Message" button. This uses AJAX via jQuery to get data from the server and then display that on the web page.

technologies

web

play! 2

scalatra

finatra

finagle

spray

skinny

lift

socko

unfiltered

play! 2

App.scala

```
object Application extends Controller {
  def index = Action {
    Ok(views.html.index(“Hello World”))
  }
}
```

routes

```
GET  /  controllers.Application.index
```

main.scala.html

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
    <head>
        <title>@title</title>
    </head>
    <body>
        @content
    </body>
</html>
```

index.scala.html

```
@(message: String)

@main {
    <h1>@message</h1>
}
```

play! 2

finatra

```
class HelloWorld extends Controller {
  get("/") { request =>
    render.plain("Hello World").toFuture
  }
}

object App extends FinatraServer {
  register(new HelloWorld())
}
```

database

anorm

slick

scalalikejdbc

squeryl

sorm

activate

reactivemongo

casbah

anorm

```
SQL(
  """
    SELECT * FROM country c
    JOIN   CountryLanguage l ON l.CountryCode = c.Code
    WHERE  c.code = {countryCode};
  """
).on("countryCode" -> “FRA”)()
 .map(_[String](“code”) -> _[String](“name”))
 .toList
```

activate

```
class Person(var name: String) extends Entity

transactional {
  new Person(“James Hughes”)
}

val james = transactional {
  select[Person].where(_.name :== “James Hughes”).head
}

transactional {
  all[Person].foreach(_.delete)
}
```

testing

scalatest

specs2

scalacheck

scalautils

functional suite

```
test(“Empty set size == 0”) {
  assert(Set.empty.size == 0)
}
```

functional spec

```
describe("A Set") {
    it("should have size 0") {
      assert(Set.empty.size == 0)
    }
}
```

flat spec

```
“Empty Set” should "have size 0" in {
  assert(Set.empty.size == 0)
}
```

scalatest

akka actors

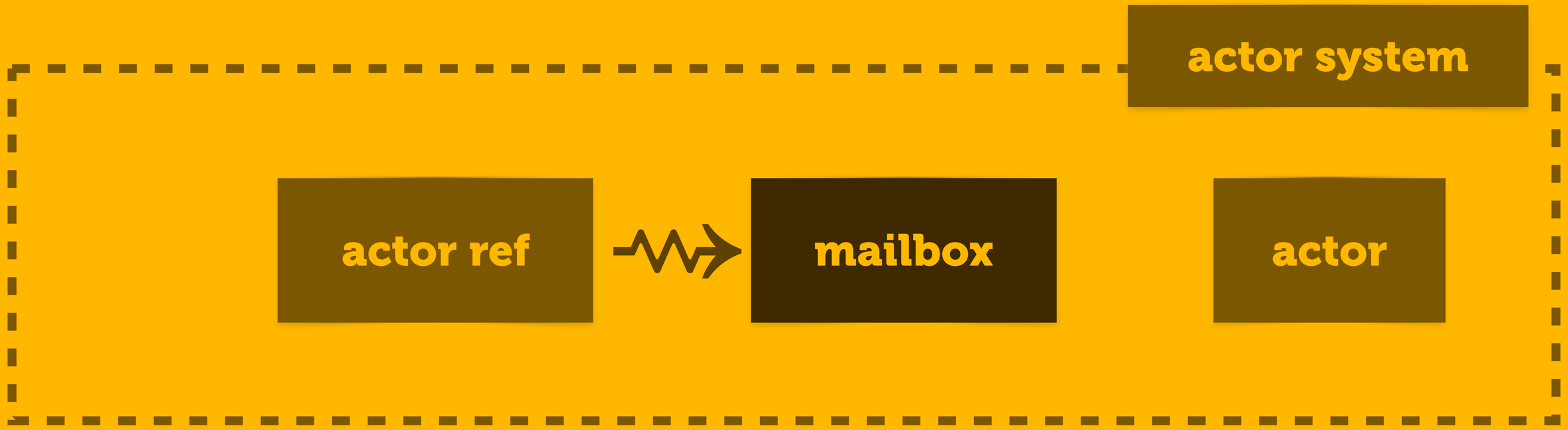
https://speakerdeck.com/rayroestenburg/akka-in-action

actor


actor

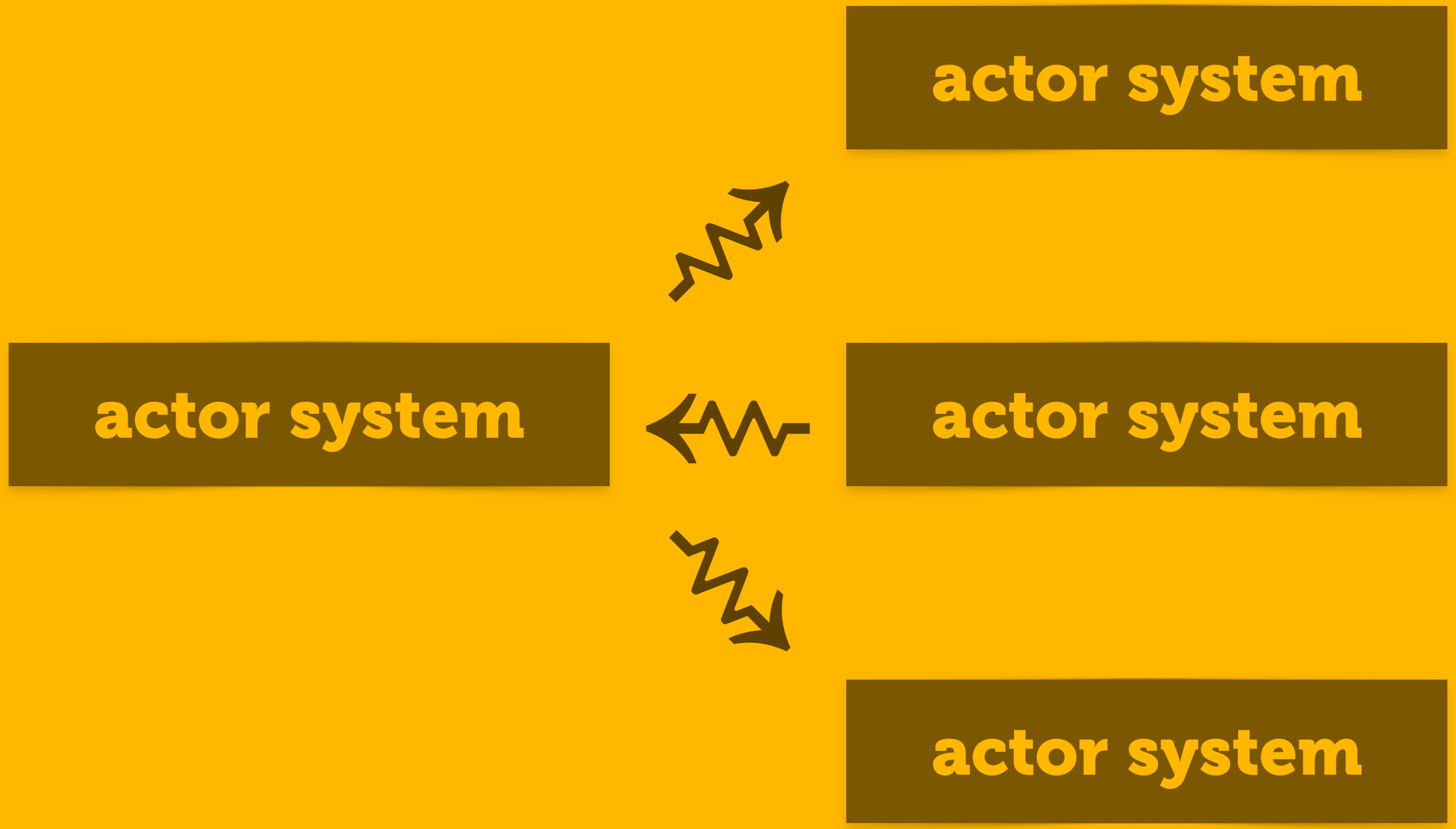

actor system
actor ref
mailbox
actor


actor system
actor ref
mailbox
actor


actor system
actor ref
mailbox
actor


actor system
actor ref
mailbox
actor


actor system
actor ref
mailbox
actor


actor system
actor system
actor system
actor system

Scala for C# Developers