A Gentle Introduction to Scala and FP

Hi, I'm Anton!

- Functional programmer
- Scala enthusiast
- Applications engineer at Rakuten

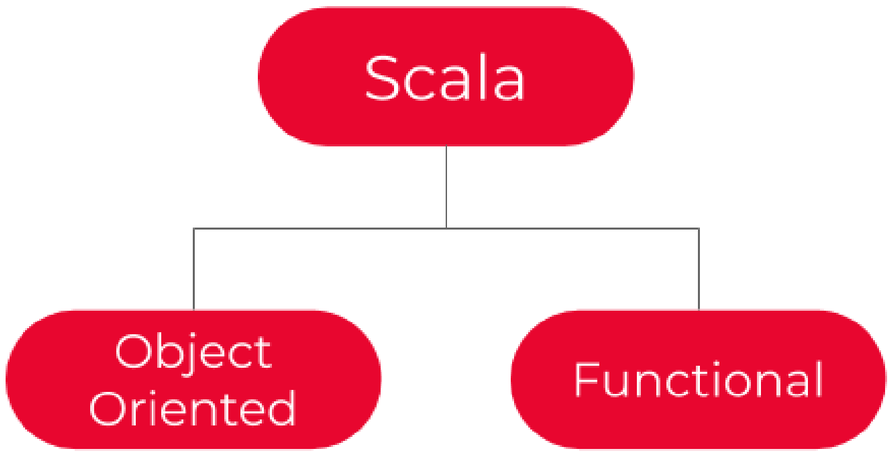
@a7emenov

Find me on Twitter, Telegram and Github

Agenda

- Introduce Scala as a language
- Give examples of universal FP constructs
- Show benefits of those constructs
- Compare Scala to other languages

What is Scala as a language?



Scala combines object-oriented and functional programming in one concise, high-level language.

What lies at the core of FP?

What lies at the core of FP?

- Absence of side-effects in computations

What lies at the core of FP?

- Absence of side-effects in computations
- Function composition

What lies at the core of FP?

- Absence of side-effects in computations
- Function composition
- Immutable data

What lies at the core of FP?

- Absence of side-effects in computations
- Function composition
- Immutable data
- Dark magic

What lies at the core of FP?

- Absence of side-effects in computations
- Function composition
- Immutable data
- Dark magic

What lies at the core of FP?

- Absence of side-effects in computations
- Function composition
- Immutable data
- Dark magic

All of it

Functional programming is a way of solving problems which centers around combining side-effect-free constructs in a declarative manner

Declarative vs. imperative

Imperative

```
for (int i = 1; i ⩽ 5; i++) {
  int temp = i * 2;
  if (temp < 6) {
    System.out.println(temp);
  }
}
```

Declarative

```
(1 to 5)
  .map(_ * 2)
  .filter(_ < 6)
  .foreach(println)
```

Deep dive

FP & Scala

- Immutability
- Pure functions
- Referential transparency
- Algebraic data types
- Pattern matching
- Recursion

Value mutation

Variables

- Can be changed anytime
- Can be changed from any place that has a reference
- Can lead to unintentional shared mutable state

Immutable values

- Can't be changed
- Create new values instead of changing the old ones
- No mutable state

Value mutation

Variables

Immutable values

Immutable values

Benefits

- Reduce concurrency issues
- No shared mutable state
- Easier local reasoning

Drawbacks

- Extra memory allocations

Pure functions

Side-effect free - everything that a function does can be observed in its return value:

- **Total** - return values for every possible input
- **Deterministic** - given same inputs, return same outputs
- **Locally scoped** - no effect on the state outside of the function's scope

Referential transparency

All calls to a function
can be substituted
with its return value

Referential transparency

Not transparent

```
def getInt(l: Int, r: Int): Int =
  Random.between(l, r)

def increment(x: Int): Int = {
  val res = x + 1
  destroyTheWorld()
  res
}
```

Referential transparency

Not transparent

```
def getInt(l: Int, r: Int): Int =
  Random.between(l, r)

def increment(x: Int): Int = {
  val res = x + 1
  destroyTheWorld()
  res
}
```

Transparent

```
def fact(n: Int): Int = {
  var res = 1
  var i = 1
  while (i ⩽ n) {
    res = res * i
    i = i + 1
  }
  res
}
```

Business value with pure functions

Can we create a useful program without any side-effects?

Business value with pure functions

- Describe side-effecting computations instead of instantly running them

- Combine all side-effects to produce a single description

- Run the resulting computation in the `main` method

Business value with pure functions

```
IO[A]
```

A description of a computation
that will produce `A`.

Business value with pure functions

```scala
def saveValue(key: String, value: Int): IO[Unit] = ???

def getValue(key: String): IO[Option[Int]] = ???

def rateLimit[A](program: IO[A]): IO[A] = ???

val program: IO[Option[Int]] =
  ratelimit(
    saveValue("key", 1) >> getValue("key")
  )

val result: Option[Int] = program.runSyncUnsafe()
```

Business value with pure functions

```scala
def saveValue(key: String, value: Int): IO[Unit] = ???

def getValue(key: String): IO[Option[Int]] = ???

def rateLimit[A](program: IO[A]): IO[A] = ???

val program: IO[Option[Int]] =
  ratelimit(
    saveValue("key", 1) >> getValue("key")
  )

val result: Option[Int] = program.runSyncUnsafe()
```

Business value with pure functions

```scala
def saveValue(key: String, value: Int): IO[Unit] = ???

def getValue(key: String): IO[Option[Int]] = ???

def rateLimit[A](program: IO[A]): IO[A] = ???

val program: IO[Option[Int]] =
  ratelimit(
    saveValue("key", 1) >> getValue("key")
  )

val result: Option[Int] = program.runSyncUnsafe()
```

Business value with pure functions

```scala
def saveValue(key: String, value: Int): IO[Unit] = ???

def getValue(key: String): IO[Option[Int]] = ???

def rateLimit[A](program: IO[A]): IO[A] = ???

val program: IO[Option[Int]] =
  ratelimit(
    saveValue("key", 1) >> getValue("key")
  )

val result: Option[Int] = program.runSyncUnsafe()
```

Business value with pure functions

```
def saveValue(key: String, value: Int): IO[Unit] = ???

def getValue(key: String): IO[Option[Int]] = ???

def rateLimit[A](program: IO[A]): IO[A] = ???

val program: IO[Option[Int]] =
  ratelimit(
    saveValue("key", 1) >> getValue("key")
  )

val result: Option[Int] = program.runSyncUnsafe()
```

Pure functions

Benefits

- Ultimate local reasoning
- Parallelizable execution
- Easy testing

Drawbacks

- Extra object allocations
- Require an effect system or a library to operate mutable state

Algebraic data types

- **Product types** - define how to create a value
  ("has a field" relationship)

- **Sum types** - define the range of values
  ("is one of" relationship)

Product types + Sum types =
Algebraic data types

Product types

Traditional classes

```
public class Person {
  // Which is the primary constructor?
  // Are the constructors related?
  public Person(Age age,
                Year birthYear)
    { /* ... */ }

  public Person(Year currentYear,
                Age age)
    { /* ... */ }

  public Person(Year currentYear,
                Year birthYear)
    { /* ... */ }
}
```

Product types

Traditional classes

```
public class Person {
  // Which is the primary constructor?
  // Are the constructors related?
  public Person(Age age,
                Year birthYear)
    { /* ... */ }

  public Person(Year currentYear,
                Age age)
    { /* ... */ }

  public Person(Year currentYear,
                Year birthYear)
    { /* ... */ }
}
```

Product types

```
// Primary constructor defines the product
case class Person(age: Age,
                  birthYear: Year) {

  // Supplementary constructors
  // must use the primary constructor
  def this(currentYear: Year, age: Age)
    { this(age, currentYear - age) }


  def this(currentYear: Year, birthYear: Year
    { this(currentYear - birthYear, birthYear
}
```

Sum types

Traditional interfaces

```
public interface User
  { /* Common methods */ }

public class Client implements User
  { /* Client-specific methods */ }

public class Admin implements User
  { /* Admin-specific methods */ }

void process(User user) {
  // How can this function reason about
  // possible implementations of User?
}
```

Sum types

Traditional interfaces

```
public interface User
  { /* Common methods */ }

public class Client implements User
  { /* Client-specific methods */ }

public class Admin implements User
  { /* Admin-specific methods */ }

void process(User user) {
  // How can this function reason about
  // possible implementations of User?
}
```

Sum types

```
// "Sealed" allows the compiler to find
// all descendants of User
sealed trait User
  { /* Common methods */ }

case class Client extends User
  { /* Client-specific methods */ }

case class Admin extends User
  { /* Admin-specific methods */ }

def process(user: User): IO[Unit] = {
  // The function now has knowledge that
  // there are only 2 options for "user"
}
```

Pattern matching

Data construction

```
sealed trait User

case class Client(email: String
                  role: String)
  extends User

case class Admin(name: String,
                 scopes: List[String])
  extends User

val client: User =
  Client("test@email.com", "manager")

val admin: User =
  Admin("Jane", List("emails", "orders")
```

Data deconstruction

```
def process(u: User): IO[Unit] = u match {
  case Client(email, "manager") ⇒
    logManager(email)

  case Client(email, _) ⇒
    logNonManager(email)

  case Admin(name, Nil) ⇒
    logAdmin(name, "No scopes")

  case Admin(name, List("emails")) ⇒
    logAdmin(name, "Only 'emails' scope")

  case Admin(name, scopes) ⇒
    logAdmin(name, "Has 'orders' scope")
}
```

ADTs and pattern matching

Benefits

- Enhanced code readability
- Reduced scope of thinking
- Compile-time checks

Drawbacks

- Extra memory allocations

Recursion

- Declarative
- Native mapping to many real-world problems
- Can be as efficient as a loop*

```scala
import scala.annotation.tailrec

@tailrec
def foldLeft[R](list: List[Int])
               (acc: R)
               (f: (R, Int) ⇒ R): R =
  list match {
    case Nil ⇒
      acc
    case head :: tail ⇒
      foldLeft(tail)(f(acc, head))(f)
  }

// "12345"
foldLeft(List(1, 2, 3, 4, 5))(acc = "") {
  (acc, num) ⇒ acc + num.toString
}
```

FP in Scala - covered points

- Immutability
- Pure functions
- Referential transparency
- Algebraic data types
- Pattern matching
- Recursion

Scala language features

- For-comprehensions
- Higher-kinded types
- Syntax extensions
- Macros
- Implicit values

For comprehensions

```
def getProfile(userName: String): Option[Profile] = ???

def getComments(profile: Profile): Option[List[Comment]] = ???

def getBonus(profile: Profile): Option[Int] = ???

val updatedComments = for {
  profile  ← getUserProfile("Jack")
  comments ← getComments(profile)
  bonus    ← getBonus(profile)
} yield comments.take(10)
                .map(_.increaseRating(bonus))
```

For comprehensions

- Define coherent syntax for sequential composition
- Syntactic sugar for every type with `map`, `flatMap` and `withFilter` methods
- Can be used with any type supplying these methods
- Take advantage of pattern matching

Higher-kinded types

- **Simple types:**
  Int, String, User
- **Unary type constructors:**
  List[A], Future[A]
- **Binary type constructors:**
  Either[A, B], Tuple[A, B]

Higher-kinded types:

```
trait Transformer[L[_]] {
  def run[A, B](l: L[A])
               (f: A ⇒ B): L[B]
}

val listT = new Transformer[List] {
  def run[A, B](l: List[A])
               (f: A ⇒ B): List[B] =
    l.map(f)
}

// 0, 1, 2
listT.run(List(1, 2, 3))(_ - 1)
```

Macros

- Eliminate boilerplate code
- Provide implementations at compile time

```
import io.circe._
import io.circe.generic.semiauto._

case class Foo(a: Int, b: String)

val fooEncoder: Encoder[Foo] =
  deriveEncoder

fooEncoder.encode(Foo(1, "b"))
```

Implicit values

- Provide automatic value forwarding
- Resolved at compile-time
- Can be created by the compiler

```
case class Foo(a: Int, b: String)

object Foo {
  implicit val e: Encoder[Foo] =
    deriveEncoder
}

def process(foos: List[Json])
  (implicit e: Encoder[Foo]) =
    foos.map(d.decodeJson)

/// Other part of the program:
process(List(Foo(1, "a")))
```

Syntax extensions

- Allow to create ad-hoc methods for code reuse and readability

- Can be used for types imported from other projects/libraries

```
implicit class StringSyntax
                  (s: String) {
  def asciiLetters: String =
    s.filter(c ⇒ c < 256 && c.isLetter
}

// "123"
"1ab23c".asciiLett
```

Scala language - covered points

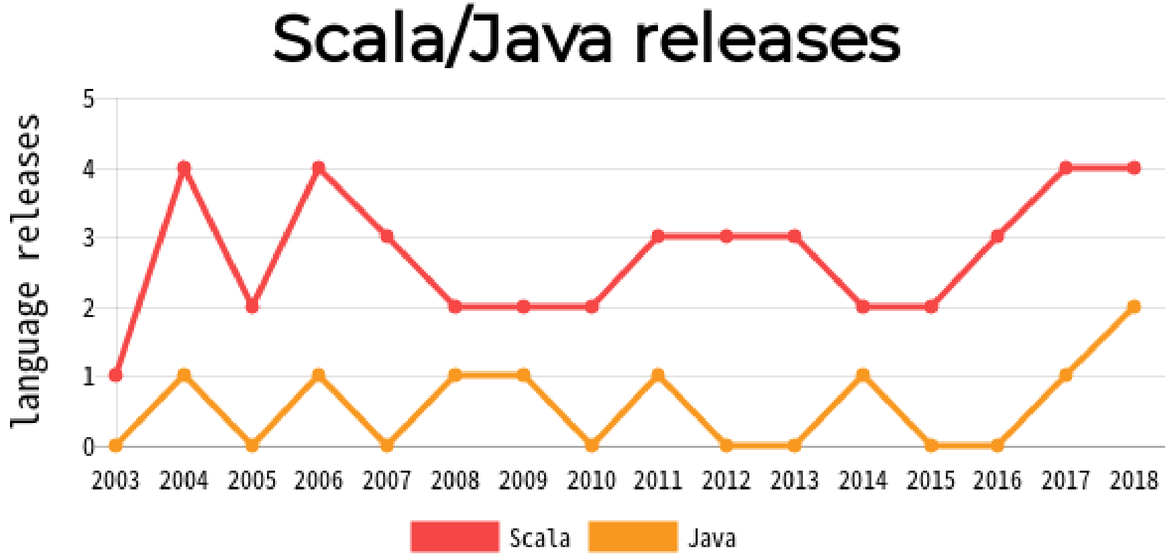
- For-comprehensions
- Higher-kinded types
- Macros
- Implicit values
- Syntax extensions

Scala in numbers

- Development and adoption rates
- Effect on productivity
- Performance

Scala adoption rates

- Maintained intense release schedule for over 15 years
- Development of the language is supported by the European government and enterprise investors[1]
- Almost as popular as Kotlin for backend development[2] and is 15% more widespread in big companies[3]



1. bit.ly/2Xr3bnH
   bit.ly/2rPlUgW
2. api.github.com
3. stackshare.io/scala

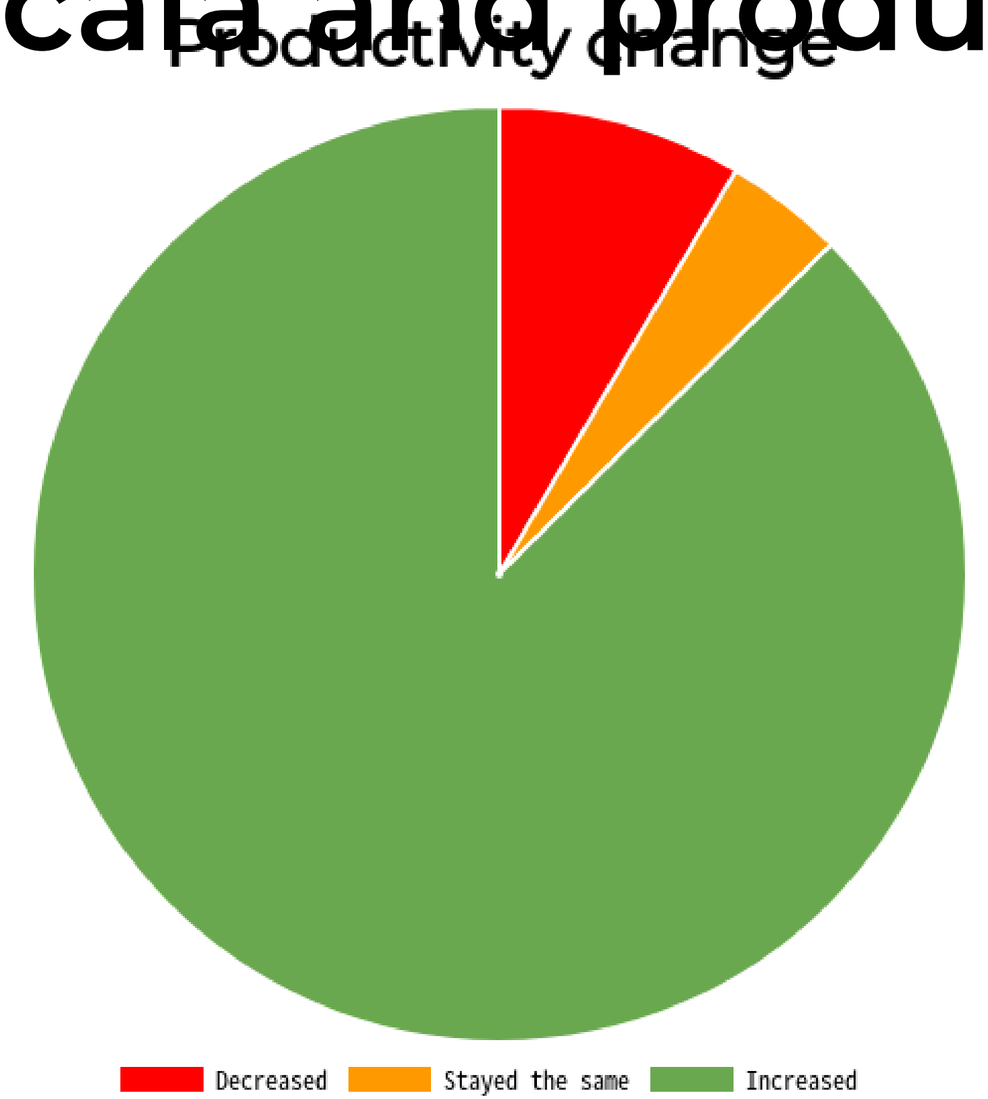
Scala adoption rates

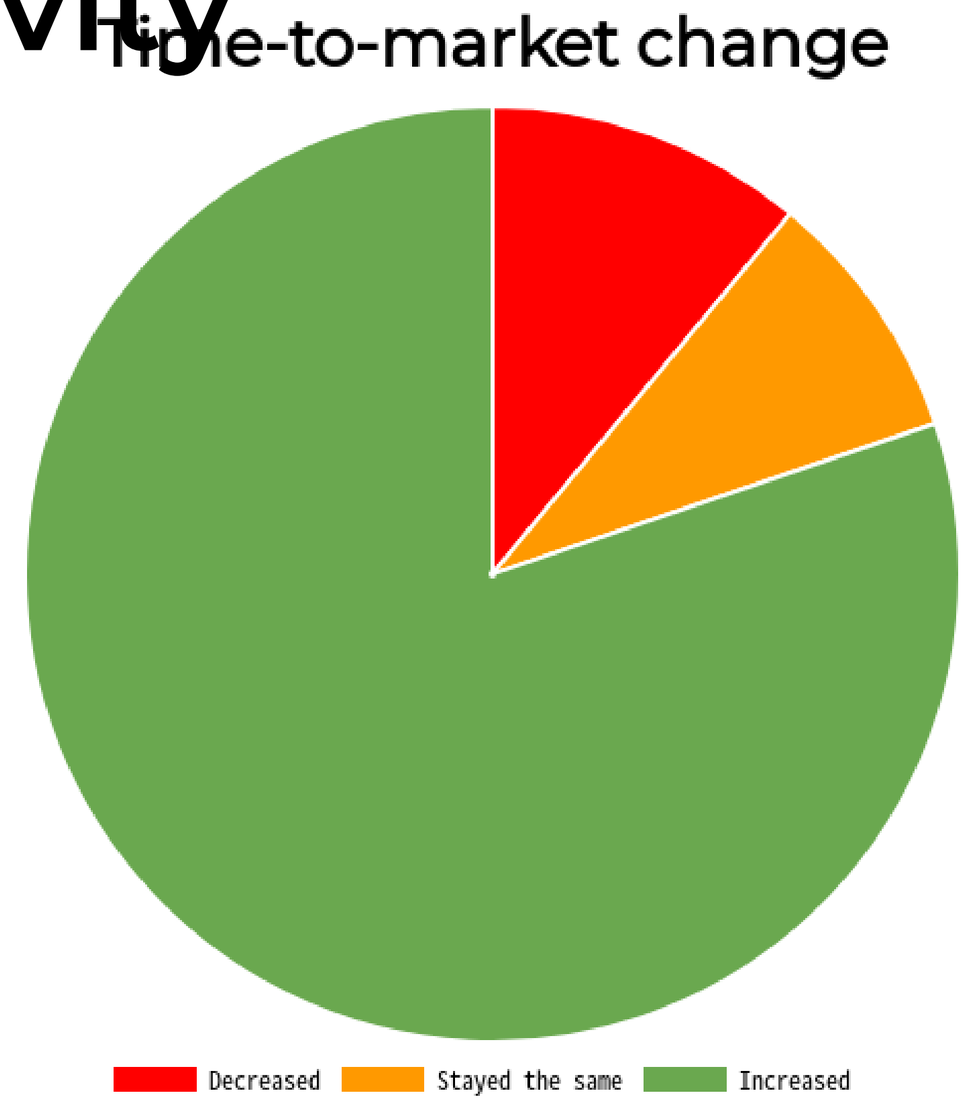
- StackOverflow survey of 2019 claims that almost 60% of respondents have tried Scala and are interested in working with it

- Approximately 4% of developers are use Scala as their main language at work

Source: insights.stackoverflow.com/survey/2019

Scala and productivity





Source: bit.ly/2qombqM

Scala's functional programming style allows writing 2 times less code[1]

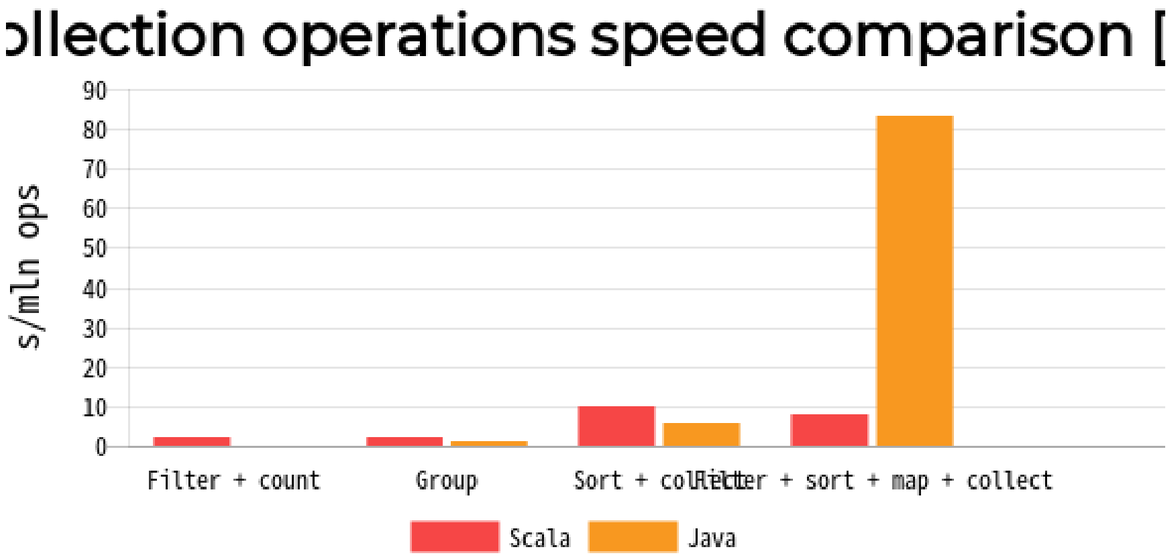
Reading Scala code is 2 times faster on average[1]

Research data[2] shows that Scala projects need ~30% less bug fixes

1. bit.ly/37eiWTH
2. arxiv.org/pdf/1901.10220.pdf

Scala's performance

- Uses JVM as the main runtime platform
- Can be compiled to a native binary
- Up to 35% performance increase with GraalVM[1]
- Utilizes incremental compilation for only a few seconds delay per build



1. graalvm.org/scala
2. bit.ly/330Ugue

Scala as a language

- Rapidly developed
- Adopted by many big companies
- Offers significant productivity benefits
- Keeps reasonable performance

Scala in real world

- Areas of application
- Scala vs. Java & Kotlin
- Scala vs. Python
- Scala vs. Golang
- Future prospects

Areas of application

Data pipelines

Small or simple systems

Resilient applications

Low-level manipulation

Distributed systems

UI-centered applications

Scala vs. Java & Kotlin

Java & Kotlin

- Vast number of libraries for different needs
- Reasonable performance for most tasks
- OOP approach
- JVM as runtime platform

Scala

- Fully compatible with Java and Kotlin libraries
- Performance competitive with Java/Kotlin
- Allows for both FP and OOP
- Target JVM, browser and native environments

Scala vs. Python

Python

- Ad-hoc Spark & Hadoop integration
- Easy to prototype ideas
- Great number of math and visualisation libraries
- Relatively slow on big volumes on data

Scala

- Native Spark & Hadoop integration
- Compile-time verification
- Mainly ML-oriented libraries
- Performs well on terabytes of data

Scala vs. Golang

Golang

- Fixed imperative style
- Blazing fast speed
- Concurrency primitives on the language level

Scala

- Extensible declarative style
- Reasonable performance
- Libraries for scalability

Future prospects

Next major version of Scala is coming at the end of 2020.

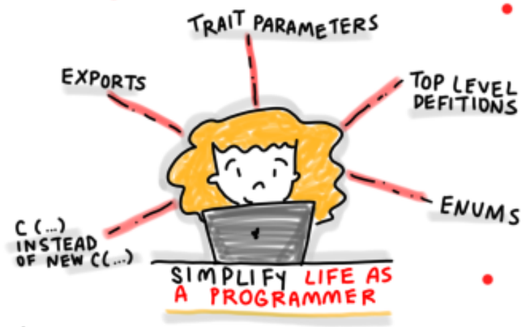
Take a look at:
dotty.epfl.ch

A SCALA 3 UPDATE

MARTIN ODERSKY
@odersky
CREATOR OF SCALA

GOAL FOR 3.0
WORK OUT ESSENCE
ELIMINATE PITFALLS
IMPROVE PREDICTABILTY
MORE APPEALING & EASY TO USE
IMPROVE TOOLING
DROP BAGGAGE
NATURAL & BEAUTIFUL
CONNECT WITH FOUNDATIONS
MORE OPINIONATED

-C·O·N·N·E·C·T-
WITH FOUNDATIONS
- INTERSECTION & UNION TYPE
- TYPE LAMBDAS
- MORE FUNCTION TYPES: DEPENDENT, IMPLICIT, POLYMORPHIC



FEB 2020
CONCENTRATES ON FLEXIBLE ABSTRACTIONS

SCALA IS A UNIFER
SIMPLIFY FOR USERS
PRIORITIZE FOUNDATIONS
DON'T NEED TO MIGRATE EVERYTHING
CAN MIGRATE WITH CONFIDENCE
LOTS OF BENEFITS
YES
A NEW LANGUAGE?
NO
A PROCESS ... WILL BE AMAZING

OPTIONAL BRACES
NO PARENTS AT ALL!
THERE'S CODE VARIATION TODAY
NO BRACES AT ALL!

A NEW FOUNDATION FOR META PROGRAMMING
CONVERSION AS A TYPECLASS
INLINE QUOTE & SPLICES FOR SAFER MACROS

A NEW WAY TO EXPRESS TERM INFERENCE
REPLACE "IMPICIT" AS A MODIFIER
DON'T HAVE NAMES
"GIVENS"
FIRST CLASS TYPECLASSES
A TRAIT
NO ADDITIONAL IMPORTS NECESSARY

PATTERN DEFINITIONS & GENERATORS THAT CAN FALL SILENTLY
RESTRICT UNNECESSARY OR UNSAFE FEATURES
FORBID UNSOUND TYPE PROJECTIONS
METHOD CALLS ALWAYS USE UNLESS DECLARED @infix
EQUALS
OPT IN FEATURE

LIST – CONTAINS
has-takes a list type class
contains

@mindseyecrf
MIND'S EYE CREATIVE

#scalalove #scalaloveconf
Scala Love

Thank you for your attention!

Any questions?

@a7emenov

Find me on Twitter, Telegram and Github