Are you interested in
functional programming
and looking for an
alternative to Haskell?

Explore a new world...

Scala

putting the fun into functional programming

An interactive introduction to the
language with Keoni D'Souza

```scala
val training_df = sqlContext.createDataFrame(training_pca)
val test_df = sqlContext.createDataFrame(test_pca)



val tokenizer = new Tokenizer()
val hashingTF = new HashingTF()
val lr1 = new LogisticRegression()
val pipeline = new Pipeline()

val mod = pipeline.fit(training_df)

mod.transform(test_df)
  .select("id", "text", "probability", "prediction")
  .collect()
  .foreach { case Row(id: Long, text: String, prob: Vector, prediction: Do
  println(s"($id, $text) --> prob=$prob, prediction=$prediction")
}
```

Scala

```
val training_df = sqlContext.createDataFrame(training_pca)
val test_df = sqlContext.createDataFrame(test_pca)



val tokenizer = new Tokenizer()
val hashingTF = new HashingTF()
val lr1 = new LogisticRegressio
val pipeline = new Pipeline()

val mod = pipeline.fit(training_df)

mod.transform(test_df)
  .select("id", "text", "probability", "prediction")
  .collect()
  .foreach { case Row(id: Long, text: String, prob: Vector, prediction: Do
  println(s"($id, $text) --> prob=$prob, prediction=$prediction")
}
```

Scala

putting the fun

Scala

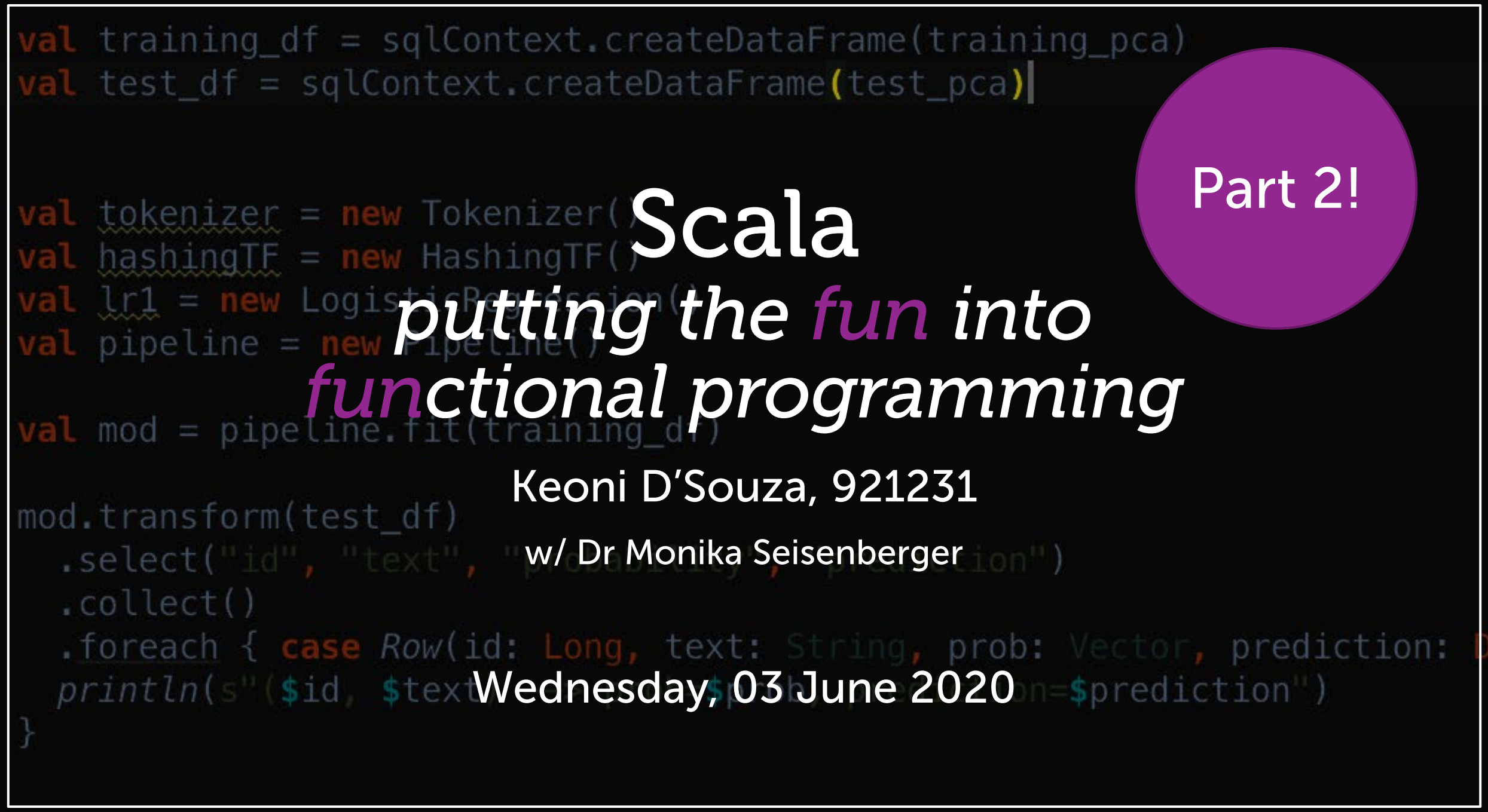
putting the fun into

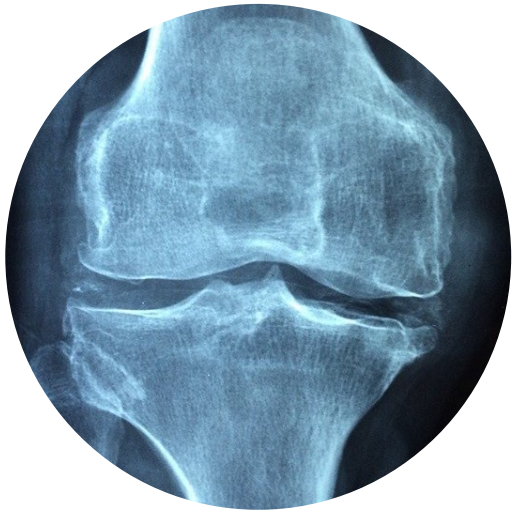
```
val training_df = sqlContext.createDataFrame(training_pca)
val test_df = sqlContext.createDataFrame(test_pca)


val tokenizer = new Tokenizer()
val hashingTF = new HashingTF()
val lr1 = new LogisticRegression()
val pipeline = new Pipeline()

val mod = pipeline.fit(training_df)

mod.transform(test_df)
  .select("id", "text", "probability", "prediction")
  .collect()
  .foreach { case Row(id: Long, text: String, prob: Vector, prediction: Do
  println(s"($id, $text) --> prob=$prob, prediction=$prediction")
}
```


Part 2!
Scala
putting the fun into functional programming
Keoni D'Souza, 921231
w/ Dr Monika Seisenberger
Wednesday, 03 June 2020



Benvenuto/a!

Willkommen!

Добро пожаловать!

Welcome!

Välkommen!

Bienvenue !

Croeso!

Witaj!

¡Bienvenido/a!

Selamat datang!

Bem-vindo/a!

Hoş geldin!

Benvenuto/a!

Willkommen!

Добро пожаловать!

Välkomm

venue !

[APPLAUSE]

Wi

a!

Selamat datang!

Bem-vindo/a!

Hoş geldin!

WHAT DID WE LOOK AT LAST TIME?

WHAT DID WE LOOK AT LAST TIME?

Recap of the introductory session

Course outline

01

Getting started with Scala

- Function declarations/definitions
- Reading user input with `io.StdIn`
- Lists: mutable `ListBuffer`, immutable `List`
- Higher order list operations: `map`, `filter`, `flatten` and `flatmap`
- Writing Scala code to interact with Java code

02

Getting more in-depth with Scala

- Pattern matching
- Traits
- Recursion
- Cats, a functional programming library
  - Type classes
  - Variance

Monoids and Semigroups

What did we look at last time?
Recap of the introductory session

- Function declarations/definitions
- Reading user input with `io.StdIn`
- Lists: mutable `ListBuffer`, immutable `List`
- Higher order list operations: `map`, `filter`, `flatten` and `flatmap`
- Writing Scala code to interact with Java code

What are we looking at this time?
Today's advanced session

- Pattern matching
- Traits
- Recursion
- Cats, a functional programming library
  - Type classes
  - Variance
  - Monoids and Semigroups

SHOW ME THE GOOD STUFF...

SHOW ME THE GOOD STUFF...

Programming functionally in Scala

Pattern matching

- Case classes can be taken deconstructed and expressions evaluated on its contents
- The syntax is:

```
expr match {
  case pattern1 => expr1
  case pattern2 => expr2
  ...
}
```

- Match compares `expr` to each pattern, finds the first match and then executes the code block

A pattern can be:

1. A name, binding any value onto it
2. An underscore, matching and ignoring anything
3. A literal
4. A case class constructor-style pattern

Pattern matching
Literal patterns

- Matching to a particular value

```
(1 + 1) match {
  case 1 => "It's one!"
  case 2 => "It's two!"
  case 3 => "It's three!"
}

Person("Jeremy", "Kyle") match {
  case Person("Trisha", "Goddard") => "It's Trisha!"
  case Person("Jeremy", "Kyle") => "It's Jezza!"
}
```

Works with all literals, except primitives, `String`s, `null`s and `()`

Pattern matching
Constant patterns

- Matching to capitalised identifiers matching a predefined value

```
val X = "Foo"
// X: String = Foo
val Y = "Bar"
// Y: String = Bar
val Z = "Baz"
// Z: String = Baz

"Bar" match {
  case X => "It's foo!"
  case Y => "It's bar!"
  case Z => "It's baz!"
}
```

Pattern matching
Alternate patterns

- Matching using vertical bars describing alternatives

```
val X = "Foo"
// X: String = Foo
val Y = "Bar"
// Y: String = Bar
val Z = "Baz"
// Z: String = Baz

"Bar" match {
  case X | Y => "It's foo or bar!"
  case Z => "It's baz!"
}
```

Pattern matching
Capturing variables

- Lowercase identifiers bind values to variables, which can be used on the right-side

```
Person("Jerry", "Springer") match {
  case Person(f, n) => f + " " + n
}
```

- Using `@` – in the form `x @ y` – lets us capture a value in `x` and match it against a pattern in `y` at the same time

```
Person("Graham", "Norton") match {
  case p @ Person(_, s) =>
    s"The person $p has the surname $s"
}
```

`x` must be a variable pattern, but `y` can be any type of pattern

Pattern matching
Wildcard patterns

- Matching using _ to ignore the value

```
Person("Stephen", "Fry") match {
  case Person("Stephen", _) => "It's Stephen!"
  case Person("Sandi", _) => "It's Sandi!"
}

Person("Stephen", "Fry") match {
  case Person(name, _) => s"It's $name!"
}

Person("Alan", "Davies") match {
  case Person("Stephen", _) => "It's Stephen!"
  case Person("Sandi", _) => "It's Sandi!"
  case _ => "It's someone else!"
}
```

_ is useful when nested inside other patterns and when acting as an `else` in the final case

Pattern matching
Type patterns

Matches any value of type Y, binding it to x

- Matching using the form x: Y, where Y is a type and x is a wildcard/variable pattern

```
val shape: Shape = Rectangle(1, 2)
// shape: Shape = Rectangle(1.0,2.0)
shape match {
  case c : Circle => s"It's a circle: $c!"
  case r : Rectangle => s"It's a rectangle: $r!"
  case s : Square => s"It's a square: $s!"
}
// res: String = It's a rectangle:
// Rectangle(1.0,2.0)!
```

Pattern matching
Tuple patterns

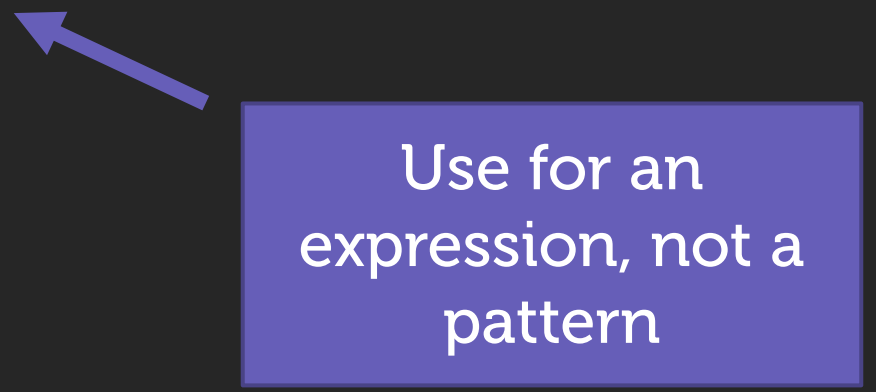
- Matching using tuples (of any arity)

```
(1, 2) match {
  case (a, b) => a + b
}
// res: Int = 3
```

Matching
Guards

- You can add a conditional in a case clause

```
123 match {
  case a if a % 2 == 0 => "even"
  case _ => "odd"
}
// res: String = odd
```



Pattern matching
Extractors

- Custom *extractor patterns* are defined with an `unapply`/`unapplySeq` method and used alongside built-in patterns in `match` expressions

Pattern matching
Extractors: case classes

- Case class companion objects have extractors creating patterns of the constructor's arity
- Variables can be used to secure fields

```
Person("Richard", "Ayoade") match {
  case Person(f, l) => List(f, l)
}
// res: List[String] = List(Richard, Ayoade)
```

Pattern matching
Extractors: regular expressions

- Regular expression objects contain a pattern letting you bind each group

```
import scala.util.matching.Regex

val r = new Regex("""(\d+)\.(\d+)\.(\d+)\.(\d+)""")

"192.168.0.1" match {
  case r(a, b, c, d) => List(a, b, c, d)
}
// res: List[String] = List(192, 168, 0, 1)
```

Pattern matching
Extractors: lists and sequences

- Companion objects to `List` and `Seq` provide patterns matching sequences of fixed length

```
List(1, 2, 3) match {
  case List(a, b, c) => a + b + c
  case Nil => 0 // empty list
}
// res: Int = 6
```

Pattern matching
Extractors: lists and sequences

- Companion objects to `List` and `Seq` provide patterns matching sequences of fixed length
- Singleton object `::` matches to the head and tail

```
List(1, 2, 3) match {
  case head :: tail => s"head $head tail $tail"
  case Nil => "empty"
}
// res: String = head 1 tail List(2, 3)
```

Binary extractor patterns can be written infix (e.g. here instead of `::(head, tail)`

Pattern matching
Extractors: lists and sequences

- Companion objects to `List` and `Seq` provide patterns matching sequences of fixed length
- Singleton object `::` matches to the head and tail
  - `::`, `Nil` and `_` can match the first element of any list length

```
List(1, 2, 3) match {
  case Nil => "length 0"
  case a :: Nil => s"length 1 starting $a"
  case a :: b :: Nil => s"length 2 starting $a $b"
  case a :: b :: c :: _ => s"length 3+ starting $a $b $c"
}
// res: String = length 3+ starting 1 2 3
```

Pattern matching
Extractors: customising length

- Objects can be used as a fixed-length extractor pattern when you provide an `unapply` method and a type signature
- An `unapplySeq` can be used for variable lengths, matching exclusively on the length of the `Seq` returned with the case class pattern length
- With variable-length extractors, the wildcard sequence pattern `_*` matches 0 or more arguments and discards the rest of the values (with `@`, you can keep them)

Traits

- Traits provide class templates (like classes are templates for objects)
- Multiple classes can be described to be the same, with the same operations implementable and sharing a supertype
- Example: registered/unregistered users on a website
  - They share common fields, but aren't identical
- There are parallels with interfaces in Java

Traits

Syntax

- Supertype

```
trait T {
  // declaration or expression
}
```

- Subtypes

```
case class S(/*params*/) extends T {
  ???
}
```

Traits
Providing limitations

- Only allow extensions in the defining file with `sealed`

```
sealed trait T {
  // declaration or expression
}
```

- Don’t allow extensions anywhere with `final`

```
final case class S(/*params*/) extends T {
  ???
}
```

Optimising recursion
Tail recursion

- Recursion can use up a lot of stack space
- Scala provides an optimisation to many recursive functions with tail calls (tail recursion)

```
def method1: Int = 1

// returns result of invoking method1 straightaway
def tailCall: Int = method1

// adds a number to the invocation
def notATailCall: Int = method1 + 2
```

Optimising recursion
Tail recursion

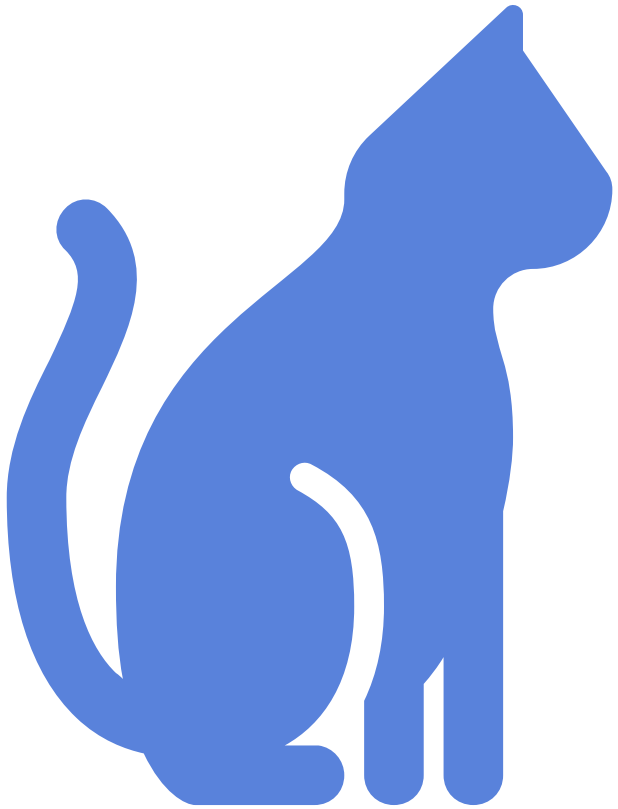
- Tail calls can be optimised to not use stack space
- JVM's limitation means Scala only optimises tail calls when callers call itself
- Using the `@tailrec` annotation makes the compiler check if the method is implementing tail recursion – it will complain otherwise
- Accumulators can provide tail-recursive equivalent functions
  - Implements heap vs stack allocation (often better)

Optimising recursion
Tail recursion

Using a wrapper function (in this example, `sum`) hides the additional parameter that can cloud up invocations (the accumulator `currentSum` is only relevant in the scope of `sumWithAccumulator`)

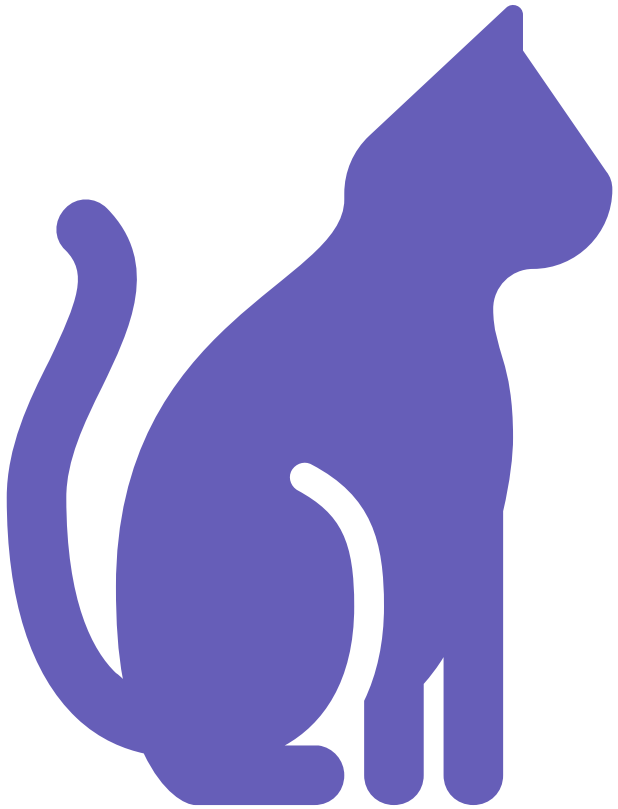
```
import scala.annotation.tailrec

def sum(list: List[Int]): Int = {
  @tailrec
  def sumWithAccumulator(list: List[Int], currentSum: Int): Int = {
    list match {
      case Nil => { // if there are no more numbers to add
        ???
      }
      case x :: xs => sumWithAccumulator(xs, currentSum + x)
    }
  }
  sumWithAccumulator(list, 0) // Initiates the function, starting at 0
}
```



Cats

- A functional programming library
- Helps us write better functionally!
- Extends existing libraries with type classes
  - Adds functionality
  - Bypasses inheritance
  - Preserves original source



Type classes in Cats
The 3 important components

1. The type class itself
2. Instances for a particular type
3. Interface methods exposed to users

Type classes in Cats
1. The type class itself

```scala
// Define a very simple JSON AST
sealed trait Json
final case class JsObject(get: Map[String, Json]) extends Json
final case class JsString(get: String) extends Json
final case class JsNumber(get: Double) extends Json
case object JsNull extends Json

// The "serialize to JSON" behaviour is encoded in this trait
trait JsonWriter[A] {
  def write(value: A): Json
}
```

supporting code

type class

A trait...

...with at least one type parameter

Type classes in Cats

2. Instances for a particular type

```
final case class Person(name: String, email: String)
object JsonWriterInstances {
  implicit val stringWriter: JsonWriter[String] =
    (value: String) => JsString(value)
  implicit val personWriter: JsonWriter[Person] =
    (value: Person) => JsObject(Map(
      "name" -> JsString(value.name),
      "email" -> JsString(value.email)
    ))
  // etc...
}
```

Defined by implementing and tagging with `implicit`

Instances → implementations (included & self-defined)

Type classes in Cats

3. Interface methods exposed to users

- Interfaces: exposed functionality
  - Generic methods
  - Accept type class instances as implicit params
- Specified through:
  - Interface objects
  - Interface syntax

Type classes in Cats

3. Interface methods exposed to users

Interface objects

Put methods in a singleton object

```
object Json {
  def toJson[A](value: A)
               (implicit w: JsonWriter[A])
  : Json = w.write(value)
}
```

Type classes in Cats

3. Interface methods exposed to users

Interface objects

Put methods in a singleton object

```
object Json {
  def toJson[A](value: A)
               (implicit w: JsonWriter[A])
  : Json = w.write(value)
}
```

Import type class instances to use it

Call the required method

```
import JsonWriterInstances._

Json.toJson(Person("Dave", "dave@example.com"))

// res4: Json = JsObject(Map(
// name -> JsString(Dave),
// email -> JsString(dave@example.com)))
```

Type classes in Cats

3. Interface methods exposed to users

Interface objects

Put methods in a singleton object

```
object Json {
  def toJson[A](value: A)
               (implicit w: JsonWriter[A])
  : Json = w.write(value)
}
```

Import type class instances to use it

Call the required method

```
import JsonWriterInstances._

Json.toJson(Person("Dave", "dave@example.com"))

// res4: Json = JsObject(Map(
// name -> JsString(Dave),
// email -> JsString(dave@example.com)))
```

Compiler sees no implicit params so looks for type class instances and inserts them

```
Json.toJson(Person("Dave", "dave@example.com"))(personWriter)
```

Type classes in Cats

3. Interface methods exposed to users

Interface syntax

Extend existing types with interface (*extension*) methods

Syntax in Cats language

```
object JsonSyntax {
  implicit class JsonWriterOps[A](value: A) {
    def toJson(implicit w: JsonWriter[A]): Json =
      w.write(value)
  }
}
```

Type classes in Cats

3. Interface methods exposed to users

Interface syntax

Extend existing types with interface (*extension*) methods

Syntax in Cats language

Import alongside other type instances needed

```
object JsonSyntax {
  implicit class JsonWriterOps[A](value: A) {
    def toJson(implicit w: JsonWriter[A]): Json =
      w.write(value)
  }
}
```

```
import JsonWriterInstances._
import JsonSyntax._
Person("Dave", "dave@example.com").toJson
// res6: Json = JsObject(Map(
// name -> JsString(Dave),
// email -> JsString(dave@example.com)))
```

Type classes in Cats

3. Interface methods exposed to users

Interface syntax

Extend existing types with interface (*extension*) methods

Syntax in Cats language

Import alongside other type instances needed

```
object JsonSyntax {
  implicit class JsonWriterOps[A](value: A) {
    def toJson(implicit w: JsonWriter[A]): Json =
      w.write(value)
  }
}
```

```
import JsonWriterInstances._
import JsonSyntax._
Person("Dave", "dave@example.com").toJson
// res6: Json = JsObject(Map(
// name -> JsString(Dave),
// email -> JsString(dave@example.com)))
```

Compiler sees no implicit params so looks for type class instances and inserts them

```
Person("Dave", "dave@example.com").toJson(personWriter)
```

Type classes in Cats

3. Interface methods exposed to users

The method `implicitly`

- Generic type class interface

```
def implicitly[A](implicit value: A): A =
  value
```

- Used to summon any value from implicit scope

```
import JsonWriterInstances._
// import JsonWriterInstances._

implicitly[JsonWriter[String]]
// res8: JsonWriter[String] = JsonWriterInstances$$anon$1@73eb1c7a
```

Most Cats type classes provide other options, but `implicitly` is good for debugging and reducing ambiguity

```
import cats.Eq
```

The Eq type class

- Supports type-safe equality
  - Alternative to `==`
  - Great for when you accidentally compare an `Int` to an `Option[Int]`.

- Comparing different types → compile error (instead of an ignored logic error)
  - `eqInt.eqv(12,12) // true`
  - `eqInt.eqv(12,"12") // error: type mismatch`

Variance
Choosing the right instance

- Variance involves subtypes
  - If we can use a value of type B where A is expected, B is a subtype of A
- You can add variance annotations to a type parameter
- This modifies:
  - The type class variance
  - How the compiler chooses instances when resolving implicits
- Variance is all about substitution

Variance
Covariance, +

- Type `F[B]` is a subtype of `F[A]` if `B` is a subtype of `A`
  - Examples: `trait List[+A]`, `trait Option[+A]`

- Collection covariance means you can substitute in collections of other types

```
sealed trait Shape
case class Circle(radius: Double) extends Shape

val circles: List[Circle] = ???
val shapes: List[Shape] = circles
```

`List[Circle]` **can substitute** `List[Shape]` **as** `Circle` **is a subtype of** `Shape`

Variance

Contravariance, -

```
trait JsonWriter[-A] {
  def write(value: A): Json
}
// defined trait JsonWriter

val shape: Shape = ???
val circle: Circle = ???
val shapeWriter:
  JsonWriter[Shape] = ???
val circleWriter:
  JsonWriter[Circle] = ???

def format[A](value: A,
              writer: JsonWriter[A])
: Json = writer.write(value)
```

- Consider two values of types `Shape` and `Circle` each with a `JsonWriter`: what combinations can be passed to format?

Variance
Contravariance, -

```
trait JsonWriter[-A] {
  def write(value: A): Json
}
// defined trait JsonWriter

val shape: Shape = ???
val circle: Circle = ???
val shapeWriter:
  JsonWriter[Shape] = ???
val circleWriter:
  JsonWriter[Circle] = ???

def format[A](value: A,
              writer: JsonWriter[A])
: Json = writer.write(value)
```

- Consider two values of types `Shape` and `Circle` each with a `JsonWriter`: what combinations can be passed to format?
  - `circle` can combine with either writer as all `Circles` are `Shapes`
  - `shape` can't combine with `circleWriter` as all `Shapes` aren't `Circles`

Variance
Contravariance, -

```
trait JsonWriter[-A] {
  def write(value: A): Json
}
// defined trait JsonWriter

val shape: Shape = ???
val circle: Circle = ???
val shapeWriter:
  JsonWriter[Shape] = ???
val circleWriter:
  JsonWriter[Circle] = ???

def format[A](value: A,
              writer: JsonWriter[A])
: Json = writer.write(value)
```

- Formally model with contravariance:
  - `JsonWriter[Shape]` is a subtype of `JsonWriter[Circle]` as `Circle` is a subtype of `Shape`
  - So, `shapeWriter` can be used wherever `JsonWriter[Circle]` is expected

Variance
Invariance

- No `+` or `-`: `trait F[A]`
- `F[A]` and `F[B]` are never subtypes of each other
  - Existing relationship with `A` and `B` doesn't matter
  - Default semantics for type constructors

Monoids
Definition

```
trait Monoid[A] {
  def combine(x: A, y: A): A
  def empty: A
}
```

- Contains:
  - a `combine` operation: `(A, A) => A`
  - an `empty` element (of type `A`)

Monoids
Definition

- As well as `combine` and `empty`, monoids have to conform to some laws
- For all values `x`, `y` and `z` in `A`, `combine` must be associative and `empty` must be an identity

```
def associativeLaw[A](x: A, y: A, z: A)
                     (implicit m: Monoid[A]): Boolean = {
  m.combine(x, m.combine(y, z)) == m.combine(m.combine(x, y), z)
}

def identityLaw[A](x: A)
                  (implicit m: Monoid[A]): Boolean = {
  (m.combine(x, m.empty) == x) && (m.combine(m.empty, x) == x)
}
```

Monoids
Examples

- Integer addition:
  - Closed binary operation: `Int + Int = Int`
  - Identity element `0` where for any `Int a`,
    `a + 0 == 0 + a == a`
  - Associative: e.g. `(1 + 2) + 3 == 1 + (2 + 3)`

Monoids
Examples

- Integer addition:
  - Closed binary operation: `Int + Int = Int`
  - Identity element `0` where for any `Int a`,
    ```
    a + 0 == 0 + a == a
    ```
  - Associative: e.g. `(1 + 2) + 3 == 1 + (2 + 3)`
- Integer multiplication:
  - Same properties as addition, but with `1` instead of `0`:
    ```
    1 * 3 == 3 * 1
    ```
  - Associative: e.g. `(1 * 2) * 3 == 1 * (2 * 3)`

Monoids
Examples

- String/sequence concatenation:
  - Add strings with the concatenation binary operator
    - e.g. `"John" ++ "Merr" = "JohnMerr"`
  - Identity element `""`
    - e.g. `"" ++ "Skoob" == "Skoob" ++ ""`
  - Associative
    - e.g. `("a" ++ "b") ++ "c" == "a" ++ ("b" ++ "c")`

Monoids
Examples

- Integer subtraction is not a monoid as it's not associative

```
(1 – 2) – 3 = –4
1 – (2 – 3) =  2
```

Monoids
Semigroups

- Contains `combine` operation: `(A, A) => A`
- Can be extended into a `Monoid` with an empty element of type `A`
- Simplified, through inheritance, provides modularity:

```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}
```

```
trait Monoid[A] extends Semigroup[A] {
  def empty: A
}
```

Monoids
The Cats type class

- `Monoid` conforms to standard Cats pattern
  - Companion object has method `apply` returning instance

```
import cats.Monoid
import cats.instances.string._ // for String Monoid

Monoid.apply[String].combine("Hi ", "there")
// res0: String = Hi there
Monoid.apply[String].empty
// res1: String = ""
```

As always, invoking `apply` is unrequired

```
import cats.Monoid; import cats.Semigroup
```

Monoids

The Cats type class

- Use imports to call the right type:

```
import cats.Monoid
import cats.instances.int._ // for Int Monoid

Monoid[Int].combine(32, 10)
// res1: Int = 42
```

```
import cats.Monoid; import cats.Semigroup
```

Monoids

The Cats type class

- Use multiple imports to combine types:

```
import cats.Monoid
import cats.instances.int._ // for Int Monoid
import cats.instances.option._ // for Option Monoid

val a = Option(22) // a: Option[Int] = Some(22)
val b = Option(20) // b: Option[Int] = Some(20)

Monoid[Option[Int]].combine(a, b)
// res0: Option[Int] = Some(42)
```

Monoids
The Cats type class

- `Monoid` extends `Semigroup`, so if empty is not needed:

```
import cats.Semigroup

Semigroup[String].combine("Hi ", "there")
// res0: String = Hi there
```

Monoids
Syntax

Coming from `Semigroup`, the `combine` method is expressed through the operation `|+|`

```
import cats.syntax.semigroup._ // for |+|
import cats.instances.string._ // for String Monoid
import cats.instances.int._    // for Int Monoid

val stringResult = "Hi " |+| "there" |+| Monoid[String].empty
// stringResult: String = Hi there

val intResult = 1 |+| 2 |+| Monoid[Int].empty
// intResult: Int = 3
```

IS THAT IT?

Rounding off the presentation

References

Special thanks to these resources which have been the basis for this presentation

[1] Welsh N and Gurnell D, “Scala with Cats”, 2017

[2] Welsh N and Gurnell D, “Essential Scala”, 2017

That's it from me – for now...!

Lab time! Log in and go to

---

keonidsouza.com/scala