Essential Essential Scala

Dave Gurnell, @davegurnell

underscore

Prologue

in which we learn what we're in for…

Essential Me

Dave Gurnell
dave@underscore.io

Consultant and trainer at Underscore

6 years Scala
10 years functional programming

Essential Overview

Scala is
Statically typed
Object oriented
Functional

Essential Overview

Creating data

Processing data

Sequencing computation

Abstractions on abstractions

Essential Context

Taken from our *Essential Scala* course/book:
http://underscore.io

Email me for a preview copy!

Essential Admin

Get the code now
https://github.com/underscoreio/eescala-code

Get your editor ready

Follow along as we go (it’s the only way to learn)

Part One

in which we create data…

Everything is an Object

In Scala everything is an object

We interact with objects by
calling methods
accessing fields

Everything is an Object

If everything is an object, what is…

1 + 2

???

Operator Syntax

If everything is an object, what is…

1 + 2

1 is an object
+ is a method
2 is an argument

Operator Syntax

1 + 1 is 1.+(2)

a b c is a.b(c)

a b c d is a.b(c).d

Known as *operator syntax*

Symbolic method names are fine in Scala

Expressions vs Values

Expressions are *program text*

Expressions *evaluate* to *values*

Scala is *expression oriented*
Most parts of Scala are expressions
(e.g. conditionals)

Defining Objects

We can define our own objects

```
object Name {
  // body goes here
}
```

Objects have fields and methods

Defining Fields

Syntax for defining fields

```
val name = expression
val name: Type = expression
```

Example

```
val firstName = "Mary"
val secondName: String = "Sue"
```

Defining Methods

Syntax for defining methods

```
def name(arg1: Type, arg2: Type, ...): Type = {
  // body goes here
}
```

Return type optional

Example

```
def sayHello(other: Type): String = {
  firstName + " says hello to " + other
}
```

Complete Example

```
object Mary {
  val firstName = "Mary"
  val lastName = "Sue"

  def sayHello(other: String): String = {
    firstName + " says hello to " + other
  }
}
```

Exercises

Exercises

A *cat* has a *color* and *favoriteFood*

Oswald is *black* and his favourite food is *milk*

Henderson is *ginger and white*,
and his favorite food is *chips*

Quentin is *tabby and white*,
and his favorite food is *curry*

Define objects for these cats!

Exercises

Add a method *eat* to one of your cats

eat accepts a food parameter (a *String*)
and returns a *String*

If the food is the cat's favourite, return "OMNOM",
otherwise return "Blehhh".

Pro tip: remember that *if* is an expression

Defining Classes

Classes let us abstract over objects

```
class Name(arg1: Type, arg2: Type, ...) {
  // body goes here
}
```

We create objects (instances) using the *new* keyword

```
new Name(arg1, arg2, ...)
```

Exercises

Exercises

Define a *Cat* class

Create instances for our three cats

Case Classes

We usually use *case class* rather than *class*

```
case class Name(arg1: Type, arg2: Type, ...) {
  // body goes here
}
```

Constructor parameters become fields automatically

We no longer need to use the *new* keyword

Pattern Matching

We can interact with case classes in a new way:
pattern matching

```
expression match {
  case pattern1 => expression1
  case pattern2 => expression2
  // ...
}
```

Pattern Matching

```scala
case class Person(
  firstName: String,
  lastName: String)

aPerson match {
  case Person(first, last) =>
    s"Hello, $first $last!"
}
```

first and *last* are names bound to values
use _ for values we don't care about
can also use *literals* as patterns

Exercises

Exercises

Re-define *Cat* as a case class

Create an object *ChipShop* with a method *serves*:
accepts a parameter of type *Cat*
return a *Boolean* if the cat's favourite food is chips

Use pattern matching to achieve your result

Summary

Summary

Case classes represent combinations of values

A has an *X* and a *Y*

We interact with case classes in two ways

method calls

pattern matching

Part Two

in which we create more data…

Part Two

In this part, we'll focus on modelling data

In particular, *logical **ors***

We'll be introduced to a pattern called
algebraic data types

No math skills will be required

Example

A website *visitor* is *anonymous* or a registered *user*

How do we model this in code?

We need to abstract over classes

Traits

Traits abstract over classes

```
trait Name {
  // body goes here
}
```

Traits are like classes except

no constructor

can contain abstract methods

Example

```
trait Visitor {
  // abstract methods
  def id: String
  def createdAt: Date

  // concrete methods and fields
  def age: Long = {
    new Date().getTime - createdAt.getTime
  }
}
```

Traits

We can *extend* traits

```
case class Name(...) extends SomeTrait {
  // body goes here
}
```

Extending a trait establishes an *is a* relationship

We can extend multiple traits if we like

`A extends B with C with D`

Algebraic Data Types

A is a B or C

```
trait A
case class B(...) extends A
case class C(...) extends A
```

Case classes at the *leaves* of the hierarchy
Traits (or perhaps classes) for *parent* elements

Example

```scala
case class Anonymous(id: String)
    extends Visitor {
  val createdAt = new Date()
}

case class User(
  id: String,
  email: String,
  createdAt: Date = new Date()
) extends Visitor
```

Uniform Access Principle

In Scala, an abstract *def* can be implemented by a *val*

This is the *uniform access principle*

We cannot tell how a field is implemented simply by accessing it

Gives flexibility to the implementation

Pattern: define all abstract fields/methods using *def*

Exercises

Exercises

A *Shape* is either a *Rectangle* or a *Circle*

Every *Shape* has a *width and a height*

A *Circle* has a *radius*

Make it so!

Destructuring Data

How do we get data out of data?

We have done this in two ways so far
polymorphic methods
pattern matching

Polymorphic Methods

We’ve been using these without comment

```
trait A {
  def foo: X
}

case class B(...) extends A {
  def foo: X = someX
}
```

Pattern Matching

The pattern is to write one *case* for each leaf in the hierarchy

```
trait A
case class B(...) extends A
case class C(...) extends A

someA match {
  case B(...) => ...
  case C(...) => ...
}
```

Exercises

Exercises

Create an object *Area* with a method *calculate*
calculate accepts a *Shape* parameter
it returns the area as a *Double*

Math tip:
the area of a rectangle is *width * height*
the area of a circle is $pi * radius^2$

Pro tip: use pattern matching!

Summary

Summary

Traits abstract over classes

A is an *X* or a *Y*

We use combinations of traits and case classes
to implement *algebraic datatypes*

Part Three

in which we create recursive data…

Recursive Data

Algebraic datatypes are often used to model *recursive data*

```
trait IntList
case object Empty extends IntList
case class Cell(
  head: Int,
  tail: IntList
) extends IntList
```

Recursive Data

We can use *pattern matching*
or *polymorphic methods*
to implement operations

Let's see some examples…

Exercises

Exercises

Implement a *length* method on *IntList*
using polymorphic methods

Implement a variant *length2*
using pattern matching

Implement a *sum* method
using your preferred syntax

Optional Exercises

Implement the following methods
using your preferred technique

```
def get(index: Int): Int
def contains(item: Int): Boolean
def indexOf(item: Int): Int
```

Structural Recursion

When are *polymorphic methods* preferable?
What about *pattern matching*?

Structural Recursion

When are *polymorphic methods* preferable?
What about *pattern matching*?

	Polymorphic methods	Pattern matching
Adding a type		
Adding an operation		

Structural Recursion

When are *polymorphic methods* preferable?
What about *pattern matching*?

	Polymorphic methods	**Pattern matching**
Adding a type	Add new code	Change existing code
Adding an operation	Change existing code	Add new code

Summary

Summary

We implement operations on algebraic datatypes using *polymorphic methods* or *pattern matching*

The structure of the operations often resembles the structure of the types

Part Four

in which we explore new uses for types…

Types

What are they good for?

Abstracting properties of values

Imposing constraints on our code

Exercises

Exercises

A *Calculation* is an
Addition, *Multiplication*, *Division*, or *Number*

Here is the signature for Calculation

```
trait Calculation {
  def calculate: Int
}
```

Make it so!

Exercises

Division by zero is a problem!

```
Division(1, 0).calculate
// throws DivisionByZeroException
```

Types to the rescue!

Exercises

A *Result* is *Finite* or *Undefined*

```
sealed trait Result
case class Finite(value: Int) extends Result
case object Undefined extends Result
```

Redefine *calculate* to return a *Result*

Pro tip: Use pattern matching!

Summary

Summary

A formal definition of a type:

"Any property of our code that can be verified without running the code."

We use types to *restrict ourselves*,
to *document intent* and *prevent bugs*.

Part Five

in which we finally introduce functional programming…

Functions

Functions are values… they are also code

```
val func =
  (a: Int, b: Int) =>
    (a + b) / 2

func(1, 3) // returns 2
```

Functions

We write *function values* like this

```
(arg1: Type1, arg2: Type2, ...) => expression
```

We write *function types* like this

```
(Type1, Type2) => ReturnType
```

Functions

A complete example

Type

```
val func: (Int, Int) => Int =
  (a: Int, b: Int) =>
    (a + b) / 2

func(1, 3) // returns 2
```

Value

Exercises

Exercises

Write a function that calculates
Pythagoras' theorem

```
math.sqrt(a * a + b * b)
```

Store the function in a variable called *pythagoras*

Higher Order Functions

We can write methods and functions
that accept and return other functions!

```
def andThen(
  f1: (Int) => Int,
  f2: (Int) => Int) =
  (input: Int) => f2(f1(input))

val both = andThen(a => a + 1, a => a * 2)

both(10) // returns 22
```

Exercises

Exercises

Add a *filter* method to *IntList*

filter accepts a parameter *f* of type *(Int) => Boolean*

filter returns an *IntList* of items where *f(item) == true*

Use *filter* to find the even numbers in

```
Cell(1, Cell(2, Cell(3, Cell(4, Empty))))
```

Optional Exercises

Add a *map* method to *IntList*

map accepts a parameter *f* of type *(Int) => Int*

map returns a new *IntList* with *f* applied to all items

Use *map* to double the items in the list

```
Cell(1, Cell(2, Cell(3, Cell(4, Empty))))
```

Summary

A function is code. It is also data.

We can build *higher order functions and methods*.

Examples include *filter* and *map* on *IntList*.

Part Six

in which we reach new heights of abstraction…

Generic Types

Type parameters allow us to abstract over types

```
trait Name[A] {
  // body goes here
}

case class Name[A](arg: Type, ...) {
  // body goes here
}
```

Generic Types

We can use type parameters to create *generic types*

```
case class Box[A](value: A)

val box1 = Box("Gary Stu")
val box2 = Box(12345)

val str: String = box1.get
val num: Int    = box2.get
```

Exercises

Exercises

Convert the example *IntList*
to a generic type *LinkedList[A]*

Pro tip: You will need to convert *Empty* to a class

```
case class Empty[A]() extends IntList
```

Optional Exercises

Implement the following methods

```
def contains(item: Int): Boolean
def indexOf(item: A): Boolean
def reverse: LinkedList[A]
```

Generic Methods

We can also create *generic methods*

```
def methodName[A, B, ...](
  arg1: Type1,
  arg2: Type2,
  ...): ReturnType = expression
```

Generic Methods

A concrete example

```
def andThen[A, B, C](
  f1: (A) => B,
  f2: (B) => C): (A) => C =
  (input: A) => f2(f1(input))

val both = andThen(
  (a: Int)    => a * 2.5,
  (a: Double) => "the answer is " + a)

both(3) // returns "the answer is 7.5"
```

Exercises

Exercises

Add a *map* method to *LinkedList*

map accepts a function *A => B*

map applies the function to all members of the list

```
trait LinkedList[A] {
  def map[B](item: A => B): LinkedList[B]
}
```

Use *map* to halve the items in the list

```
Cell(1, Cell(2, Cell(3, Cell(4, Empty))))
```

Optional Exercises

Define an *append* method to concetenate two lists

Use *append* to define a *flatMap* method
flatMap accepts a function *A => LinkedList[B]*
flatMap maps the function and appends the results

```
trait LinkedList[A] {
  def flatMap[B](
    func: A => LinkedList[B]
    ): LinkedList[B]
}
```

Summary

Type parameters allow us to abstract over types.

We can build *generic types and methods*.

We can now build collections and many other tools.

Epilogue

in which we take a deep breath and… relax

Essential Scala

Scala is
Statically typed
Object oriented
Functional

Essential Scala

Creating data
objects and classes

Processing data
algebraic datatypes, case classes, traits

Sequencing computation
polymorphism, pattern matching, structural recursion

Abstractions on abstractions
classes, traits, functions, generics, etc...

Essential Scala

This is just a taste… there's loads more

monads, for comprehensions

collections

async

type classes

the list goes on…

Essentially Done

Thanks!

We hope you enjoyed it!