Essential Scala
Six Core Concepts for Learning Scala

Noel Welsh, @noelwelsh
Introduction
Overview
1. Expressions, types, & values
2. Objects and classes
3. Algebraic data types
4. Structural recursion
5. Sequencing computation
6. Type classes
1. Expressions, types, & values
2. Objects and classes
3. Algebraic data types
4. Structural recursion
5. Sequencing computation
6. Type classes
Motivation
There are simple patterns in effective use of Scala
Improve the teaching of Scala
Huge thanks to the PLT team
http://racket-lang.org/people.html
Expressions, Types, and Values
Goal: model of evaluation
1 + 1  Expression
Int Type

1 + 1 Expression
1 + 1

Expression

2

Value

Int

Type

Int
Types exist at compile-time, values at run-time
Basic language features, syntax
Objects and Classes
Goal: familiarity with syntax
Objects and classes!
Case classes and pattern matching
Largely practicing syntax
Algebraic Data Types
Goal: translate data descriptions into code
Model data with logical ors and logical ands
A website visitor is:
• logged in; or
• anonymous
A logged in user has:
• an ID; \textit{and}
• an email address
Structure of the code follows the structure of the data
Two patterns:
• product types (and)
• sum types (or)
Product type: A has a B and C
final case class A(b: B, c: C)

A has a B and C
final case class A(b: B, c: C)

A has a B and C
final case class A(b: B, c: C)

A has a B and C
final case class A(b: B, c: C)

A has a B and C
Sum type: A is a B or C
sealed trait A
final case class B() extends A
final case class C() extends A

A is a B or C
sealed trait A
final case class B() extends A
final case class C() extends A

A is a B or C
sealed trait A
final case class B() extends A
final case class C() extends A

A is a \textcolor{green}{B} or \textcolor{green}{C}
sealed trait A
final case class B() extends A
final case class C() extends A

A is a B or C
Sum and product together make algebraic data types.
Examples
A website visitor is:
• logged in; or
• anonymous
sealed trait Visitor
final case class Anonymous()
  extends Visitor
final case class User()
  extends Visitor
A logged in user has:
• an ID; \textit{and}
• an email address

An anonymous user has:
• an ID
sealed trait Visitor {
  id: Id
}
final case class Anonymous(id: Id)
  extends Visitor
final case class User(id: Id, email: Email)
  extends Visitor
A calculation is a success or failure
sealed trait Calculation
final case class Success()
  extends Calculation
final case class Failure()
  extends Calculation
A success has an value. 
A failure has an error message
sealed trait Calculation

final case class Success(value: Int) extends Calculation

final case class Failure(msg: String) extends Calculation
Summary

• Structure data with logical ands and ors
• These are called algebraic data types
• Code follows immediately from structure of the data
Structural Recursion
Goal: transform algebraic data types
sealed trait Calculation
final case class Success(value: Int) extends Calculation
final case class Failure(msg: String) extends Calculation
Implement on Calculation

def add(value: Int): Calculation = ???
Structure of the code follows structure of the data
Two (sub-)patterns: pattern matching and polymorphism
A is a B or C
B has a D and E
C has a F and G
sealed trait A
final case class B(d: D, e: E) extends A
final case class C(f: F, g: G) extends A
Pattern matching
sealed trait A {
    def doSomething: H = {
        this match {
            case B(d, e) => doB(d, e)
            case C(f, g) => doC(f, g)
        }
    }
}

final case class B(d: D, e: E) extends A
final case class C(f: F, g: G) extends A
Polymorphism
sealed trait A {
    def doSomething: H
}
final case class B(d: D, e: E) extends A {
    def doSomething: H =
    doB(d, e)
}
final case class C(f: F, g: G) extends A {
    def doSomething: H =
    doC(f, g)
}
Example
sealed trait Calculation
final case class Success(value: Int) extends Calculation
final case class Failure(msg: String) extends Calculation
Add an Int to a Calculation
sealed trait Calculation {
  def add(value: Int): Calculation = ???
}

final case class Success(value: Int) extends Calculation

final case class Failure(msg: String) extends Calculation
sealed trait Calculation {
    def add(value: Int): Calculation =
        this match {
            case Success(v) => ???
            case Failure(msg) => ???
        }
}

final case class Success(value: Int) extends Calculation

final case class Failure(msg: String) extends Calculation
sealed trait Calculation {
  def add(value: Int): Calculation =
  this match {
    case Success(v) =>
      Success(v + value)
    case Failure(msg) =>
      Failure(msg)
  }
}

final case class Success(value: Int)
  extends Calculation

final case class Failure(msg: String)
  extends Calculation
Summary

• Processing algebraic data types immediately follows from the structure of the data

• Can choose between pattern matching and polymorphism

• Pattern matching (within the base trait) is usually preferred
Sequencing
Computation
Goal: patterns for sequencing computations
Functional programming is about transforming values
That is all you can do without introducing side-effects
A => B => C
This is sequencing computations
Three patterns: fold, map, and flatMap
Fold

A => B
Abstraction over structural recursion
sealed trait A {
  def doSomething: H = {
    this match {
      case B(d, e) => doB(d, e)
      case C(f, g) => doC(f, g)
    }
  }
}

final case class B(d: D, e: E) extends A
final case class C(f: F, g: G) extends A
sealed trait A {
  def doSomething: H = {
    this match {
      case B(d, e) => doB(d, e)
      case C(f, g) => doC(f, g)
    }
  }
}

final case class B(d: D, e: E) extends A
final case class C(f: F, g: G) extends A
sealed trait A {
    def fold(doB: (D, E) => H, doC: (F, G) => H): H = {
        this match {
            case B(d, e) => doB(d, e)
            case C(f, g) => doC(f, g)
        }
    }
}

final case class B(d: D, e: E) extends A
final case class C(f: F, g: G) extends A
Example
A Result is a Success or Failure
sealed trait Result
final case class Success() extends Result
final case class Failure() extends Result
Success contains a value of type A
sealed trait Result[A]

final case class Success[A](value: A) extends Result[A]

final case class Failure[A]() extends Result[A]
(This just an invariant Option)
Implement fold
Start with structural recursion pattern
sealed trait Result[A] {
  def fold[B]: B =
    this match {
      Success(v) => ???
      Failure()  => ???
    }
}

final case class Success[A](value: A)
  extends Result[A]
final case class Failure[A]()
  extends Result[A]
Abstract out arguments
sealed trait Result[A] {
  def fold[B](s: A => B, f: B): B =
    this match {
      Success(v) => s(v)
      Failure() => f
    }
}

final case class Success[A](value: A)
  extends Result[A]

final case class Failure[A]()
  extends Result[A]
Fold is a generic transform for any algebraic data type.
Fold is not always the best choice
Not all data is an algebraic data type
Sometimes other methods are easier to use.
Result[A]
Get user from database (might not be a user)
Convert user to JSON

Result[User]
Result[User] => Json

User => Json

Result[Json]
Map

F[A]  map  A  =>  B  =  F[B]
Get user from database (might not be a user)
Get order for user (might not be an order)
FlatMap

Example
getOrder(id: UserId): HttpResponse
Order => Json

UserId

UserId => Result[User]

User => Result[Order]

Order => Json

Result[Json] => HttpResponse
 UserId

 UserId => Result[User]

 User => Result[Order]

 Order => Json

 Result[Json] => HttpResponse
UserId => Result[User]

User => Result[Order]

flatMap

Order => Json

Result[Json] => HttpResponse
UserId => Result[User]

User => Result[Order]

Order => Json

Result[Json] => HttpResponse
Order => Json
UserId => Result[User]
User => Result[Order]
Order => Json
UserId

UserId => Result[User]

User => Result[Order]

Order => Json

Result[Json] => HttpResponse
UserId

UserId => Result[User]

User => Result[Order]

Order => Json

Result[Json] => HttpResponse
UserId => Result[User]

fold

User => Result[Order]

Order => Json

Result[Json] => HttpResponse
Summary

• Standard patterns for sequencing computations

• $F[A] \text{ map } (A \Rightarrow B) = F[B]$

• $F[A] \text{ flatMap } (A \Rightarrow F[B]) = F[B]$

• $\text{fold}$ is general transformation for algebraic data types

• You can teach monads in an introductory course!
Type Classes
Ad-hoc polymorphism
Break free from your class oppressors!
Conclusions
Scala is simple
3 patterns are 90% of code
4 patterns are 99% of code
Program design in Scala is systematic
Be like keyboard cat!
underscore.io/training/courses/essential-scala/