

Essential Scala

Six Core Concepts for Learning Scala

Noel Welsh, @noelwelsh



Introduction

Scala is complex?

Self types
Type bounds
Existential types
Trait stacks
Overloading
Implicit conversions

Is doesn't have to be
this way

Essential Scala

Noel Welsh and Dave Gurnell



underscore

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

Huge thanks to the PLT team
[http://racket-lang.org/
people.html](http://racket-lang.org/people.html)

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; *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 **B** or 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; *and*
- an email address

An anonymous has:

- an ID


```
sealed trait Visitor {  
  def 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 a 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

$A \Rightarrow B \Rightarrow C$

This is sequencing
computations

Three patterns: fold,
map, and flatMap

Fold



A



\Rightarrow



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)



Result[User]



Convert user to JSON



Result[User]

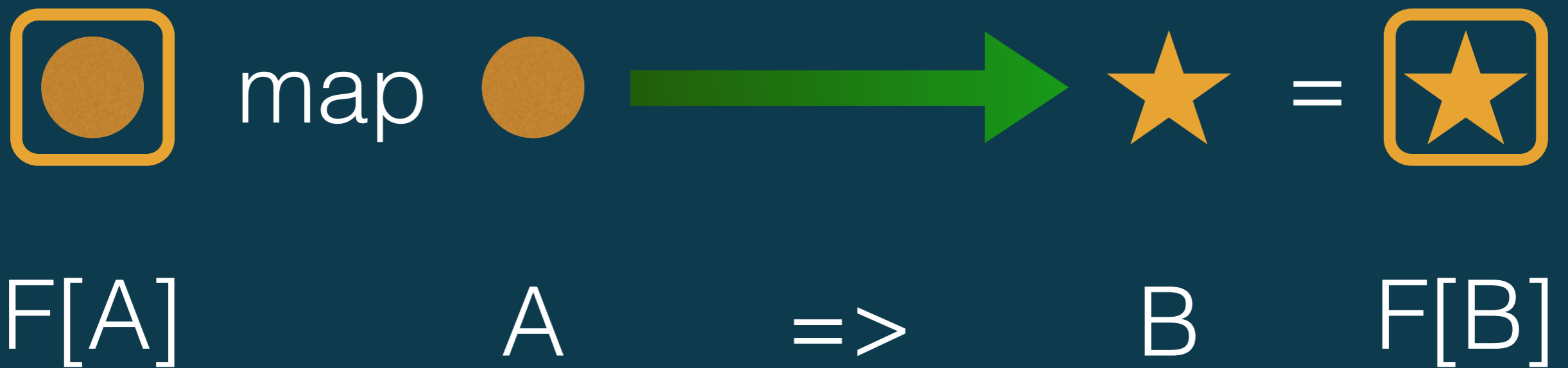


User => Json



Result[Json]

Map





Get user from database
(might not be a user)



Result[User]



Get order for user (might
not be an order)



Result[User]



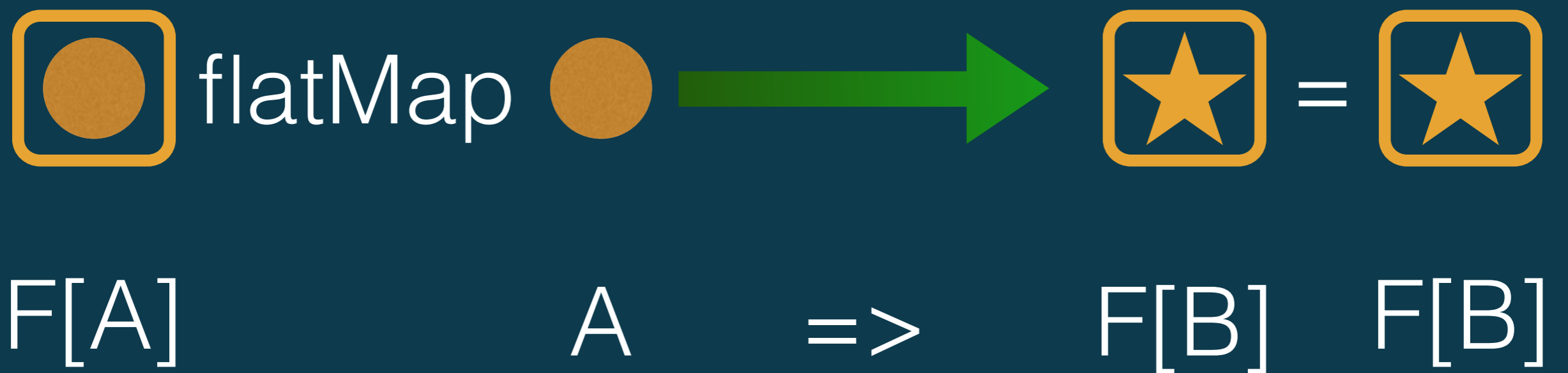
User =>

Result[Order]



Result[Order]

FlatMap



Example

```
getOrder(id: UserId):  
    HttpResponse
```



UserId



UserId => Result[User]



User => Result[Order]

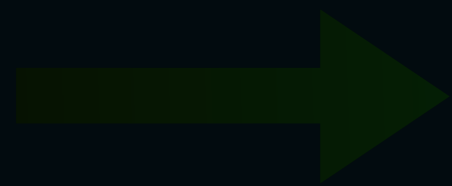


Order => Json



Result[Json] => HttpResponse

UserId



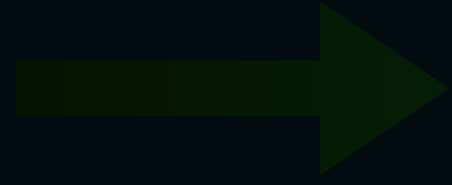
UserId => Result[User]



User => Result[Order]

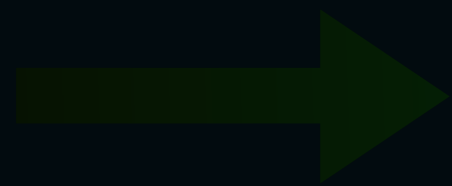


Order => Json



Result[Json] => HttpResponse

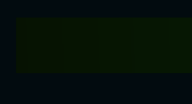
UserId



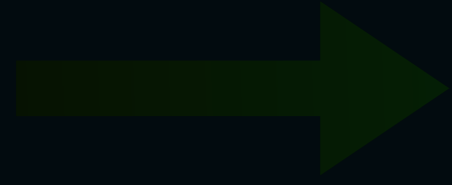
UserId => Result[User]



User => Result[Order]

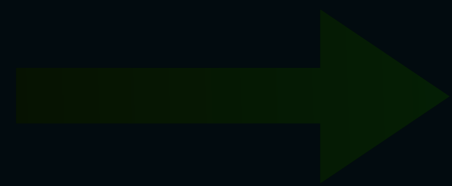


Order => Json



Result[Json] => HttpResponse

UserId



UserId => Result[User]



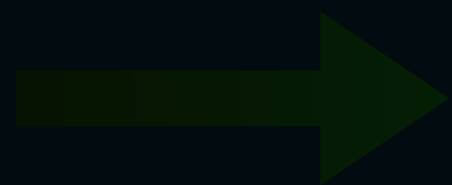
User => Result[Order]



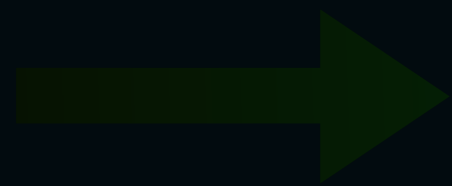
flatMap



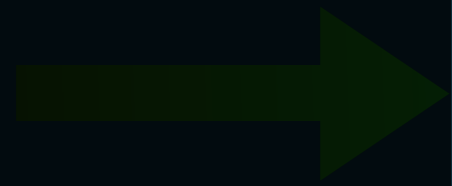
Result[Json] =>
HttpResponse



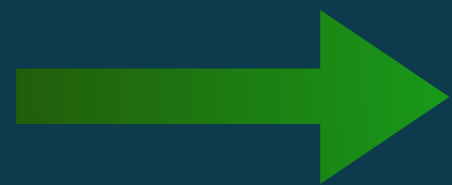
UserId



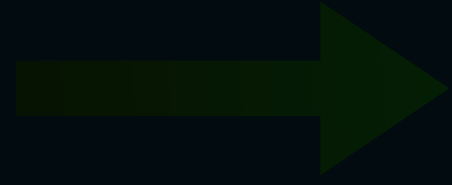
UserId => Result[User]



User => Result[Order]



Order => Json

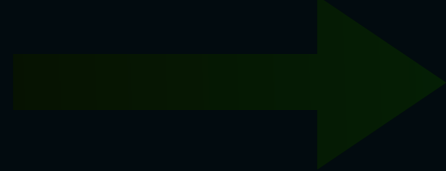


Result[Json] => HttpResponse

UserId



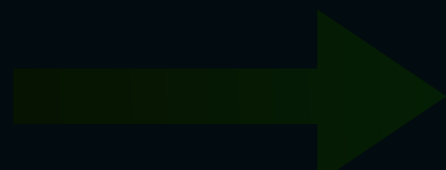
userId => Result[User]



User => Result[Order]



Order => Json



Result[Json] => HttpResponse

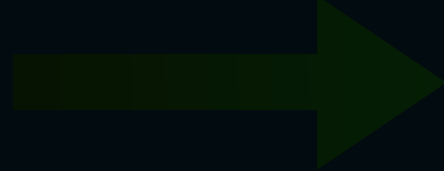
UserId



map



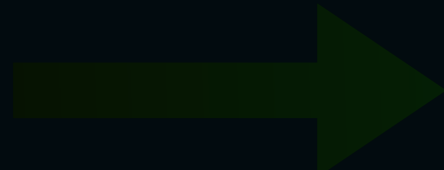
userId => Result[User]



User => Result[Order]



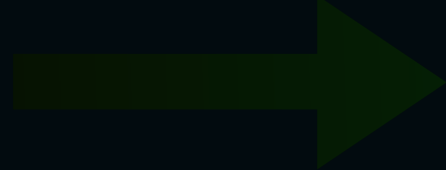
Order => Json



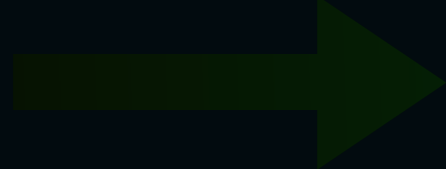
Result[Json] => HttpResponse



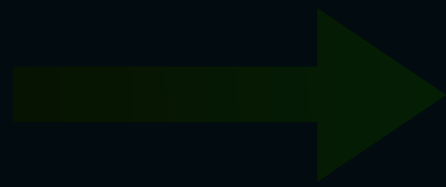
UserId



UserId => Result[User]



User => Result[Order]



Order => Json



Result[Json] =>
HttpResponse

UserId



???



UserId => Result[User]



User => Result[Order]



Order => Json



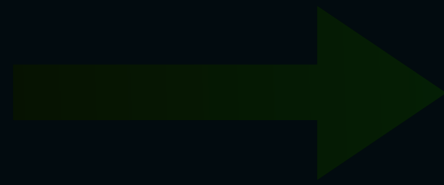
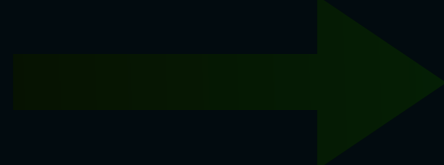
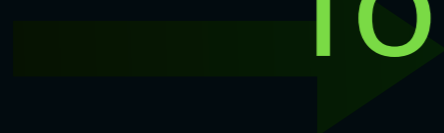
Result[Json] => HttpResponse

UserId

fold

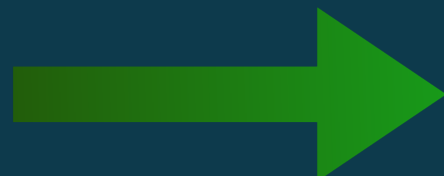


UserId => Result[User]



User => Result[Order]

Order => Json



Result[Json] => HttpResponse

Summary

- Standard patterns for sequencing computations
- $F[A] \text{ map } (A \Rightarrow B) = F[B]$
- $F[A] \text{ flatMap } (A \Rightarrow F[B]) = F[B]$
- **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!

Concerns that cross
class hierarchy

E.g. *serialize* to JSON

Common behaviour
without (useful) common
type

Abstract behaviour to
a type class

Can implement type class
instances in ad-hoc
manner

Conclusions

Scala is simple

3 patterns are 90% of
code

4 patterns are 99% of
code

Program design in
Scala is *systematic*

Essential Scala



underscore

[underscore.io/training/
courses/essential-scala/](https://underscore.io/training/courses/essential-scala/)

15% off with flatMap



Be like keyboard cat!