Thinking Functionally in Scala

THINKING FUNCTIONALLY WITH SCALA'S LANGUAGE CONSTRUCTS

Vitthal Srinivasan
CO-FOUNDER, LOONYCORN

www.loonycorn.com

Overview

Introduce Scala

Get Scala installed and ready for use

Distinguish between expressions and statements

View functions as named, reusable expressions

Introducing Scala

Linguistic Relativity

Linguistic relativity, also known as the Sapir–Whorf hypothesis, is a concept-paradigm in linguistics and cognitive science that holds that the structure of a language affects its speakers' cognition or world view. *(Wikipedia)*

The Same Thought in Different Languages

Hindi

Originated in India/South Asia - hot, tropical climate

English

Originated in Europe/medieval England - cold/temperate climate

The Same Thought in Different Languages

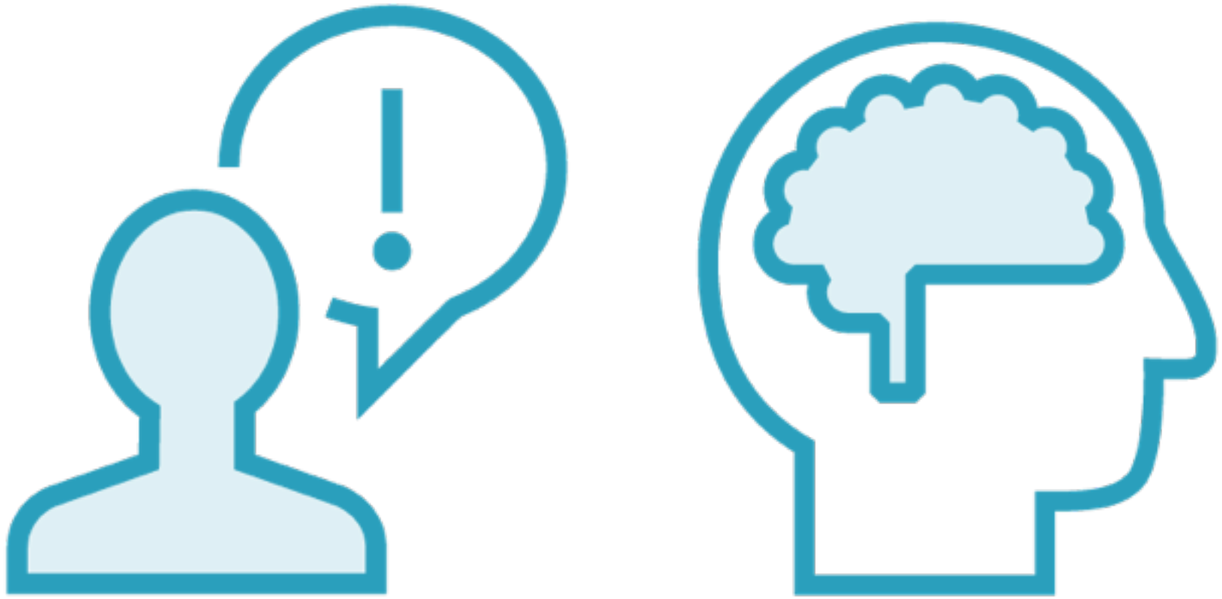
Hindi

“It cooled my heart to see her”

English

“She was a ray of sunshine”

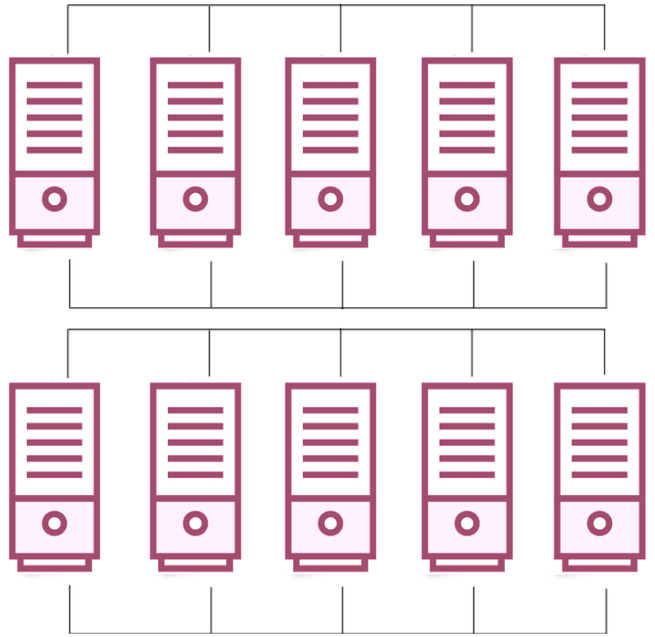
Language Shapes Thought



True in Computer Science as well as in Psychology

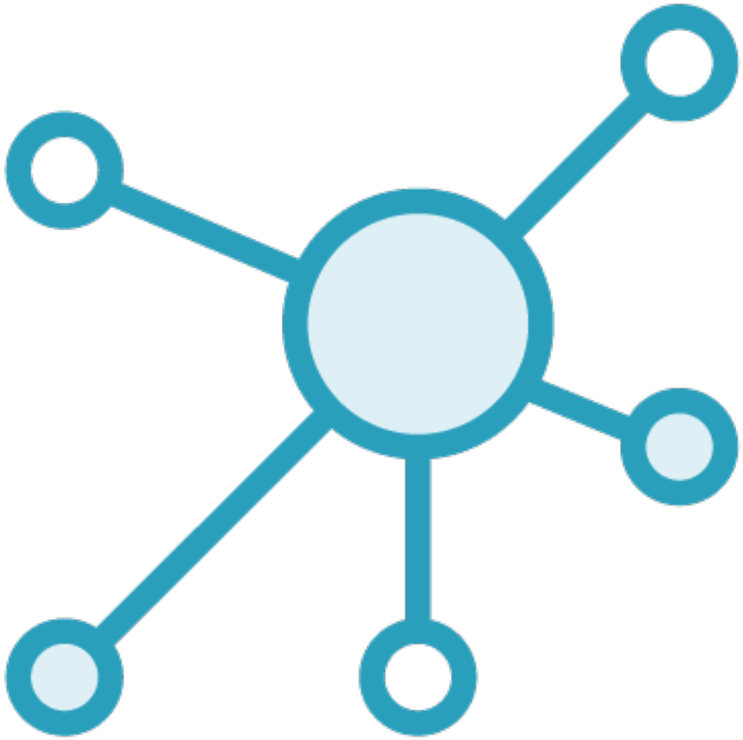
Two Ways to Build a System

Monolithic



Distributed

Language Shapes Thought



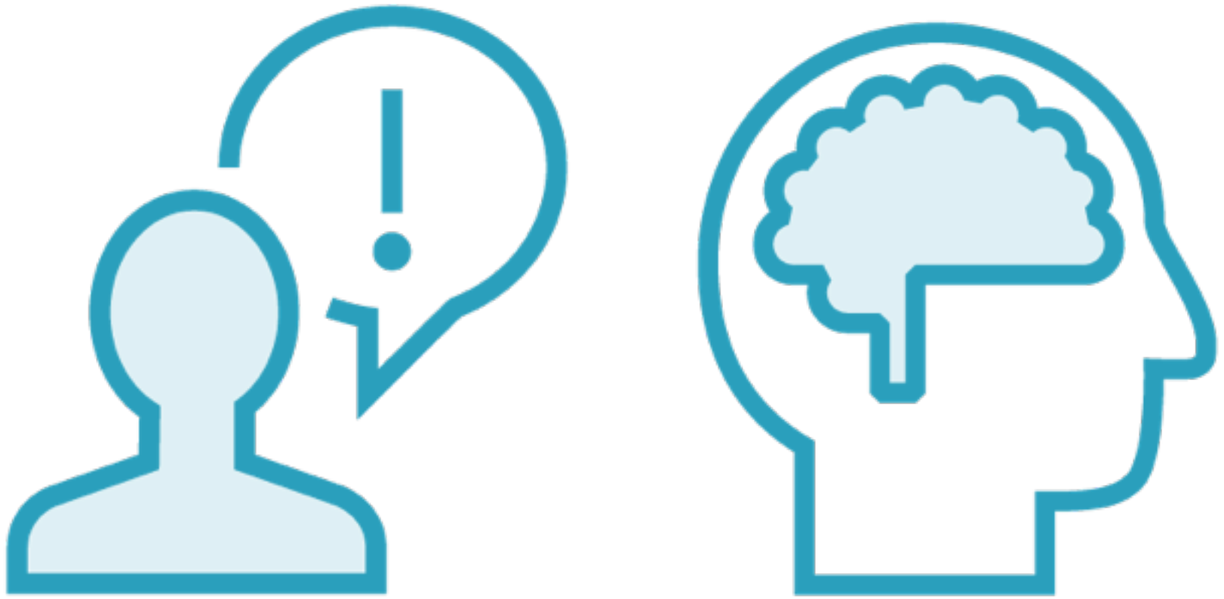
Object-Oriented

Interrelated Classes and Objects

Functional

Chained Function Calls

Language Shapes Thought



Functional languages fit well with distributed architectures

Functional Languages and Distributed Systems

First class functions

Functions can be passed to or returned from functions

Function composition

Chains of functions calling each other

Immutable data

Prevents inadvertently introduced side effects

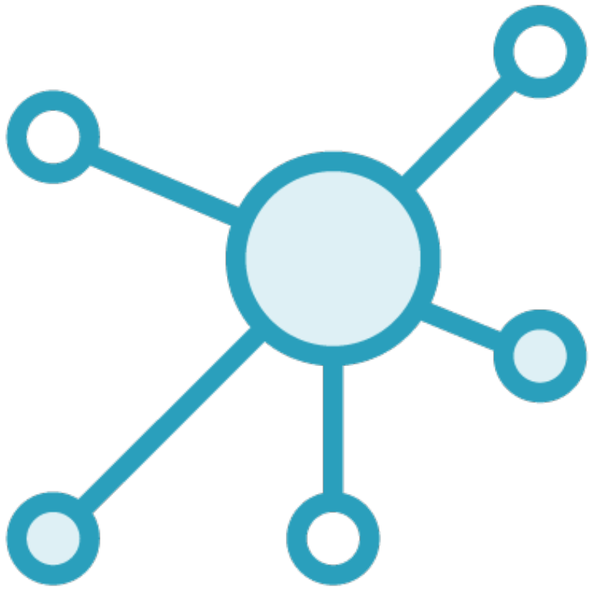
Pure functions

No side effects; same output for same input

Ease of parallelism

Pure functions acting on immutable data

Scala Straddles Paradigms



Scala is both object-oriented and functional

Scala is both object-oriented and functional

Course Objective:
Speak Scala with a Scala Accent

Course Idea: Big Ideas Behind Little Details

Little Details

Scala is packed with features that are subtly different

Big Ideas

Smart choices lie behind every one of those features

Course Outline: Think Functional, Talk Functional

Strong Basics

Simple constructs have a functional twist in Scala

Functions

Play in the big leagues in Scala, on par with objects

Collections

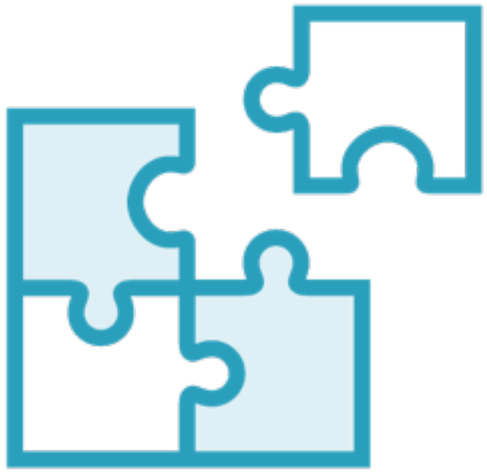
The pipes and links in functional chains

Make sure functional thought is not lost in translation

Moving on: Bigger and Better Things

This Course

Nitty-gritty of functional programming



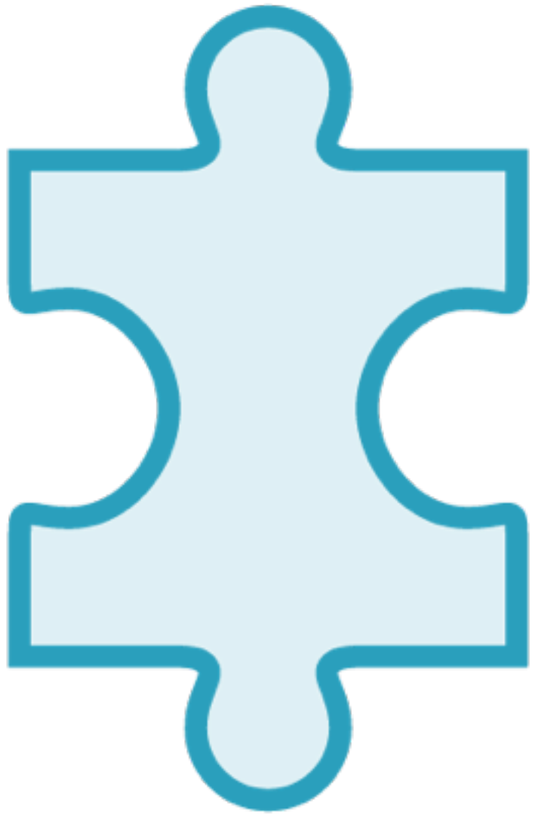
Spark

A clean, abstracted approach to Big Data (implemented in Scala)

Spark and Scala

A natural fit for large, complex data manipulation systems

This Course



Focuses on specific constructs in isolation

Intentionally does not put the pieces together

Simple, but not simplistic examples

Builds a strong foundation for bigger and better things

Speak Scala with a Scala accent

Installing and Using Scala

Demo

Download and install Scala

Use the Scala REPL environment

Compile and run Scala from the command line

Immutable Data and Pure Functions

“You could not step twice into the same river.”

Heraclitus

Can You Ever Cross the Same River Twice?

Well, in Scala you can

Can You Ever Cross the Same River Twice?

Using immutable data and pure functions

Immutable Data

Scala encourages use of immutable data

Entities are declared as immutable using ‘val’

Commonly used collections are immutable too

Can You Ever Cross the Same River Twice?

Immutable data easily leads to pure functions - and distributable code

```
val PI = 3.14;
```

Data Is Declared Immutable Using `val`

Such data can not be changed once defined

```
val PI = 3.14;
PI = 3.1415;

Error:(63, 6) reassignment to val
  PI = 3.1415;
     ^
```

Data Is Declared Immutable Using `val`

Such data can not be changed once defined

```
var radius : Double = 10.0;
```

Data Is Declared Mutable Using `var`

Use mutable data only when absolutely essential

```
val someNumbers =  List(10,20,30,40,50,60)
val stateCodes =  Map("California" -> "CA",
("Vermont","VT"))
val stateSet =  Set("California","Vermont")
```

Common Collections Are Immutable Too

Modifying an immutable collection returns a new collection, leaving the original unchanged

```
val someNumbers =
collection.immutable.List(10,20,30,40,50,60)
val stateCodes = collection.immutable.Map("California" ->
"CA",("Vermont","VT"))
val stateSet =
collection.immutable.Set("California","Vermont")
```

Common Collections Are Immutable Too

Modifying an immutable collection returns a new collection, leaving the original unchanged

Pure Functions

Pure functions have no side-effects

Pure function calls to immutable data are often easy to reorder

Pure function calls to immutable collections are often easy to parallelise

```
val weekDays = List("Mon","Tue","Wed","Thu","Fri")
weekDays.map(_ == “Mon")
```

The Map Operation

This takes in a function, applies it to each member of a collection, and returns the result

```
List[Boolean] = List(true, false, false, false, false)
```

Expressions and Statements

Expressions and Statements

Expressions

Units of code that return a value

Statements

Units of code that do not return a value

Expressions and Statements

Scala favors expressions over statements

```
“Hello World”
```

Expressions Are Ubiquitous

Anything that evaluates to a value is an expression

```
List("Mon","Tue","Wed","Thu","Fri")
```

Expressions Are Ubiquitous

Anything that evaluates to a value is an expression

```
val weekDays =
List("Mon","Tue","Wed","Thu","Fri")
```

Statements Are Less Common

Printing to screen, writing to files - these are common uses of statements

```
val weekDays =
List("Mon","Tue","Wed","Thu","Fri")
```

Expressions Can Be Used as R-values

On the right of an assignment operator used to define data

```
println(weekDays.map(_ == "Mon"))
```

Expressions Can Be Passed into Functions

Function parameters can be specified using expressions

```
val radius = 10
val area = {
  val PI = 3.14;
  PI * radius * radius
}
```

Expressions Blocks Are Enclosed in {}

The last expression in the block is the return value

```
val area = {
  val PI = 3.1
  println(s"Inside scope 1, PI = $PI");
  {
    val PI = 3.14
    println(s"Inside scope 2, PI = $PI")
    PI * radius * radius
  }
  PI * radius * radius
}
```

Expressions Blocks Can Have Nested Scopes

In the snippet above, the value of PI used is 3.1, i.e. from the outer scope

```
val area = {
  val PI = 3.1
  println(s"Inside scope 1, PI = $PI");
  {
    val PI = 3.14
    println(s"Inside scope 2, PI = $PI")
    PI * radius * radius
  }
  PI * radius * radius
}
```

Expressions Blocks Can Have Nested Scopes

In the snippet above, the value of PI used is 3.1, i.e. from the outer scope

```
val area = {
  val PI = 3.1
  println(s"Inside scope 1, PI = $PI");
  {
    val PI = 3.14
    println(s"Inside scope 2, PI = $PI")
    PI * radius * radius
  }
  PI * radius * radius
}
```

Expressions Blocks Can Have Nested Scopes

In the snippet above, the value of PI used is 3.1, i.e. from the outer scope

```
val area = {
  val PI = 3.1
  println(s"Inside scope 1, PI = $PI");
  {
    val PI = 3.14
    println(s"Inside scope 2, PI = $PI")
    PI * radius * radius
  }
  PI * radius * radius
}
```

Expressions Blocks Can Have Nested Scopes

In the snippet above, the value of PI used is 3.1, i.e. from the outer scope

```
val area = {
  val PI = 3.1
  println(s"Inside scope 1, PI = $PI");
  {
    val PI = 3.14
    println(s"Inside scope 2, PI = $PI")
    PI * radius * radius
  }
  PI * radius * radius
}
```

Expressions Blocks Can Have Nested Scopes

In the snippet above, the value of PI used is 3.1, i.e. from the outer scope

```
val area = {
  val PI = 3.1
  println(s"Inside scope 1, PI = $PI");
  {
    val PI = 3.14
    println(s"Inside scope 2, PI = $PI")
    PI * radius * radius
  }
  PI * radius * radius
}
```

Expressions Blocks Can Have Nested Scopes

In the snippet above, the value of PI used is 3.1, i.e. from the outer scope

```
val area = {
  val PI = 3.1
  println(s"Inside scope 1, PI = $PI");
  {
    val PI = 3.14
    println(s"Inside scope 2, PI = $PI")
    PI * radius * radius
  }
  PI * radius * radius
}
```

Expressions Blocks Can Have Nested Scopes

In the snippet above, the value of PI used is 3.1, i.e. from the outer scope

Functional Programming Support

Functional Programming Support

Scala's support for functional programming is hiding in plain sight

Hiding in Plain Sight

If/else

Expressions, similar to the ternary if operator (?:) in Java

For loops

Can be used either as expressions or as statements

Pattern matching

Similar to switch statements in Java, but way more power

Constructs that are statements in Java but expressions in Scala

```
val numer: Double = 22
val denom: Double = 7
val PI =
  if (denom != 0)
   {numer/denom}
  else
   {None}
```

If/else Are Expressions in Scala

Can be used as r-values, similar to ternary if expressions in Java

```
val weekDays = List("Mon","Tue","Wed","Thu","Fri")
for (day <- weekDays) {
  day match {
    case "Mon" => println("Just another Manic Monday")
    case otherDay => println(otherDay)
  }
}
```

For Loops Can Be Used as Statements

Similar to the usage in Java and other languages

```
val weekDays = List("Mon","Tue","Wed","Thu","Fri")
for (day <- weekDays) {
  day match {
    case "Mon" => println("Just another Manic Monday")
    case otherDay => println(otherDay)
  }
}
```

For Loops Can Be Used as Statements

Similar to the usage in Java and other languages

```
val weekDays = List("Mon","Tue","Wed","Thu","Fri")
val manicWeekDays = for (day <- weekDays) yield {
  day match {
    case "Mon" => "Just another Manic Monday"
    case otherDay => otherDay
  }
}
```

For Loops Become Expressions with Yield

A for-loop with yield will “yield” a collection of the return values of each iteration of the loop

```
List[String] = List(Just another Manic Monday, Tue, Wed, Thu, Fri)
```

```
day match {
  case "Mon" => "Just another Manic Monday"
  case otherDay => otherDay
}
```

Pattern Matching Is Similar to Java's Switch

But far more powerful

```
day match {
  case "Mon" => "Just another Manic Monday"
  case otherDay => otherDay
}
```

No Fall Through

Only zero or one case clauses will be executed

```
day match {
  case "Mon" => "Just another Manic Monday"
  case otherDay => otherDay
}
```

No Break Statement Either

There is a catch-all though

```
day match {
  case "Mon" => "Just another Manic Monday"
  case otherDay => otherDay
}
```

Matching Is Powerful

Matches can be on value, type or on specific conditions

Hiding in Plain Sight

If/else

Expressions, similar to the ternary if operator (?:) in Java

For loops

Can be used either as expressions or as statements

Pattern matching

Similar to switch statements in Java, but way more power

Constructs that are statements in Java but expressions in Scala

Hiding in Plain Sight

If/else

Expressions, similar to the ternary if operator (?:) in Java

For loops

Can be used either as expressions or as statements

Pattern matching

Similar to switch statements in Java, but way more power

Their being expressions provides functional programming support

Expressions Support Functional Programming

As Function Parameters

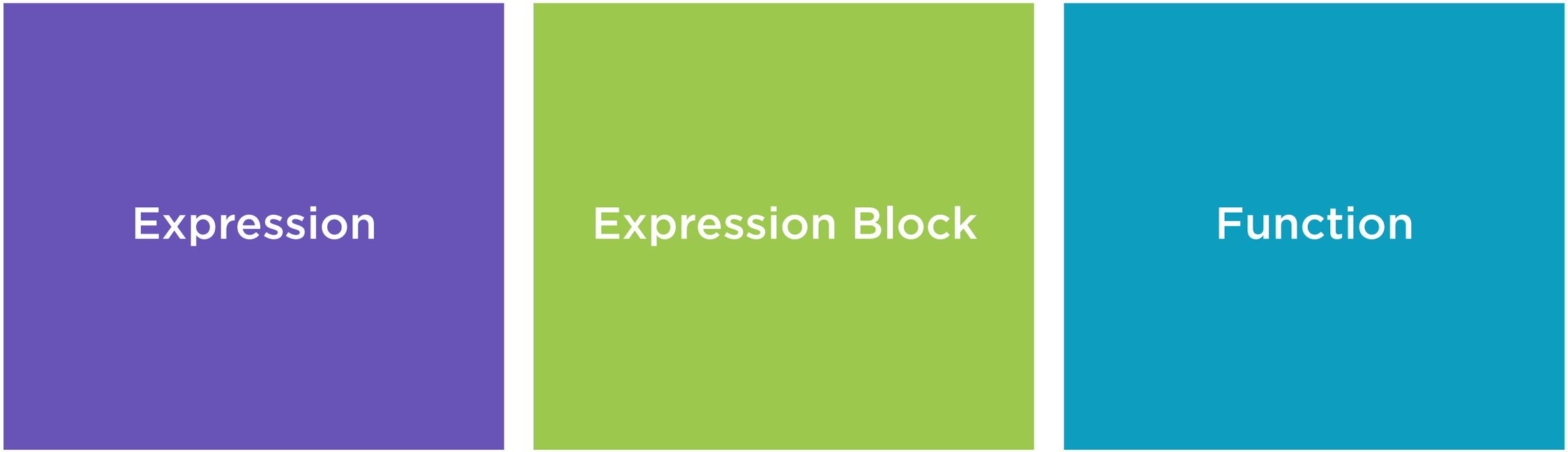
Unlike statements, expressions can be used to specify arguments to a function

As Functions

A subtle point - Scala allows expressions to be used in place of functions

Functions as Named, Reusable Expression Blocks

Hiding in Plain Sight



A function is really just a named, reusable expression block

Functions and Expression Blocks - Similarities

Expression Blocks

Units of code that return a value

Enclosed in {curly braces}

Can be passed into functions and returned from functions

Can be stored in values or variables

Can have nested scopes

Functions

Units of code that return a value

Enclosed in {curly braces}

Can be passed into functions and returned from functions

Can be stored in values or variables

Can have nested scopes

Functions and Expression Blocks - Differences

Expression Blocks

Anonymous - lack a name

Not parameterised - lack an argument list

Functions

Usually named - sometimes anonymous

Parameterised - arguments can be specified

Functions are named,
reusable expression blocks

Functions and Expression Blocks

Expression Blocks

Units of code that return a value

Enclosed in {curly braces}

Can be passed into functions and returned from functions

Can be stored in values or variables

Can have nested scopes

Functions

Units of code that return a value

Enclosed in {curly braces}

Can be passed into functions and returned from functions

Can be stored in values or variables

Can have nested scopes

Functions and Expression Blocks

Expression Blocks

Units of code that return a value

Enclosed in {curly braces}

Can be passed into functions and returned from functions

Can be stored in values or variables

Can have nested scopes

Functions

Units of code that return a value

Enclosed in {curly braces}

Can be passed into functions and returned from functions

Can be stored in values or variables

Can have nested scopes

First Class Functions

A function can be stored in a variable or value

The return value of a function can be a function

A parameter of a function can be a function

```
val compareStringsLiteral =
  (s1: String, s2: String)  =>  {
   if (s1 == s2) 0
    else if (s1 > s2)
    -1
     else 1
   }:Int
```

Functions Can Be Stored in Values or Variables

Function objects can be used as r-values

```
def getComparator(reverse:Boolean):
(String,String) => Int = {
  if (reverse == true)compareStringsDescending
  elsecompareStrings
}
```

Functions Can Be Returned from Functions

A function whose return value is a function is said to be a higher order function

```
def getComparator(reverse:Boolean):
(String,String) => Int = {
  if (reverse == true)compareStringsDescending
  else compareStrings
}
```

Functions Can Be Returned from Functions

A function whose return value is a function is said to be a higher order function

```
def
compareStrings(s1:String,s2:String):I
nt = {
  if (s1  == s2) 0
  else if (s1  > s2) -1
  else {1}
}
def
compareStringsDescending(s1:String,s2
:String):Int = {
  if (s1 == s2) 0
  else if (s1 > s2) 1
  else {-1}
}
def getComparator(reverse:Boolean):
(String,String) => Int = {
  if (reverse == true)
compareStringsDescending
  else  compareStrings
}
```

- A function that compares strings in lex order
- A function that compares strings in reverse lex order
- A third, higher order function that returns one of the two functions

```
def
compareStrings(s1:String,s2:String):I
nt = {
  if (s1  == s2) 0
  else if (s1  > s2) -1
  else {1}
}
def
compareStringsDescending(s1:String,s2
:String):Int = {
  if (s1 == s2) 0
  else if (s1 > s2) 1
  else {-1}
}
def getComparator(reverse:Boolean):
(String,String) => Int = {
  if (reverse == true)
compareStringsDescending
  else  compareStrings
}
```

- A function that compares strings in lex order
- A function that compares strings in reverse lex order
- A third, higher order function that returns one of the two functions

```
def
compareStrings(s1:String,s2:String):I
nt = {
  if (s1  == s2) 0
  else if (s1  > s2) -1
  else {1}
}
def
compareStringsDescending(s1:String,s2
:String):Int = {
  if (s1 == s2) 0
  else if (s1 > s2) 1
  else {-1}
}
def getComparator(reverse:Boolean):
(String,String) => Int = {
  if (reverse == true)
compareStringsDescending
  else  compareStrings
}
```

- A function that compares strings in lex order
- A function that compares strings in reverse lex order
- A third, higher order function that returns one of the two functions

```
def smartCompare(s1:String,
                 s2:String,
                 cmpFn:(String,String) =>
Int):Int = {
   cmpFn(s1,s2)
}
```

Functions Can Be Passed into Functions

Again, a function that takes in another function as an argument is a higher order function

```
def
compareStrings(s1:String,
s2:String):Int = {
  if (s1 == s2) 0
  else if (s1 > s2) -1
  else {1}
}

def
smartCompare(s1:String,
s2:String,
cmpFn:(String,String) =>
Int):Int = {
  cmpFn(s1,s2)
}
```

- A function that compares strings in lex order
- A higher order function that takes in the first function to compare strings
- The higher order function invokes the passed-in function

Scala's Rich Function Primitives

First class functions

Functions can be passed to or returned from functions

Closures

Functions that retain the referencing environment

Parameter groups

Logical groups of parameters to a function

Partial application

Fix some but not all parameters

Currying

Fix some but not all parameter groups

Summary

Introduced Scala - both functional and object-oriented

Spotted functional constructs - hiding in plain sight

Surveyed rich primitives for working with functions