Placing Functions on Par with Objects as First Class Entities

Vitthal Srinivasan
CO-FOUNDER, LOONYCORN

www.loonycorn.com

Overview

Create and invoke functions

Use nested functions and higher order functions

Understand Scala's code reuse mechanisms

Make sense of closures

Creating and Invoking Functions

The Dual Nature of Functions

Functions are Objects

Functions are Parameterised Expression Blocks

Functions are objects,
methods are not

```
def getArea2 (radius:Double):Double =
{
  val PI = 3.14;
  PI * radius * radius
}
```

Methods Are Not Objects

The terms method and object are often used interchangeably because methods can be stored in objects quite easily

```
val getArea = (radius:Double) =>
  {
    val PI = 3.14;
    PI * radius * radius
  }:Double
```

Functions Are Objects

Function objects are first class entities on par with classes - methods are not

Functions and Methods - Differences

Functions	Methods
Value types - can be stored in val and var storage units	Not value types - can not serve as r-values - defined using def
Objects of type function0,function1,.. (traits descending from AnyRef)	Not objects, but can be converted to function objects quite easily
Slightly slower, and higher overhead	Slightly faster and better performing
Are first class entities on par with classes	Are not first class entities unless converted to functions
Do not accept type parameters or parameter default values	Work fine with type parameters and parameter default values

Methods are associated with
a class, functions are not

Demo

Functions are parameterised expression blocks

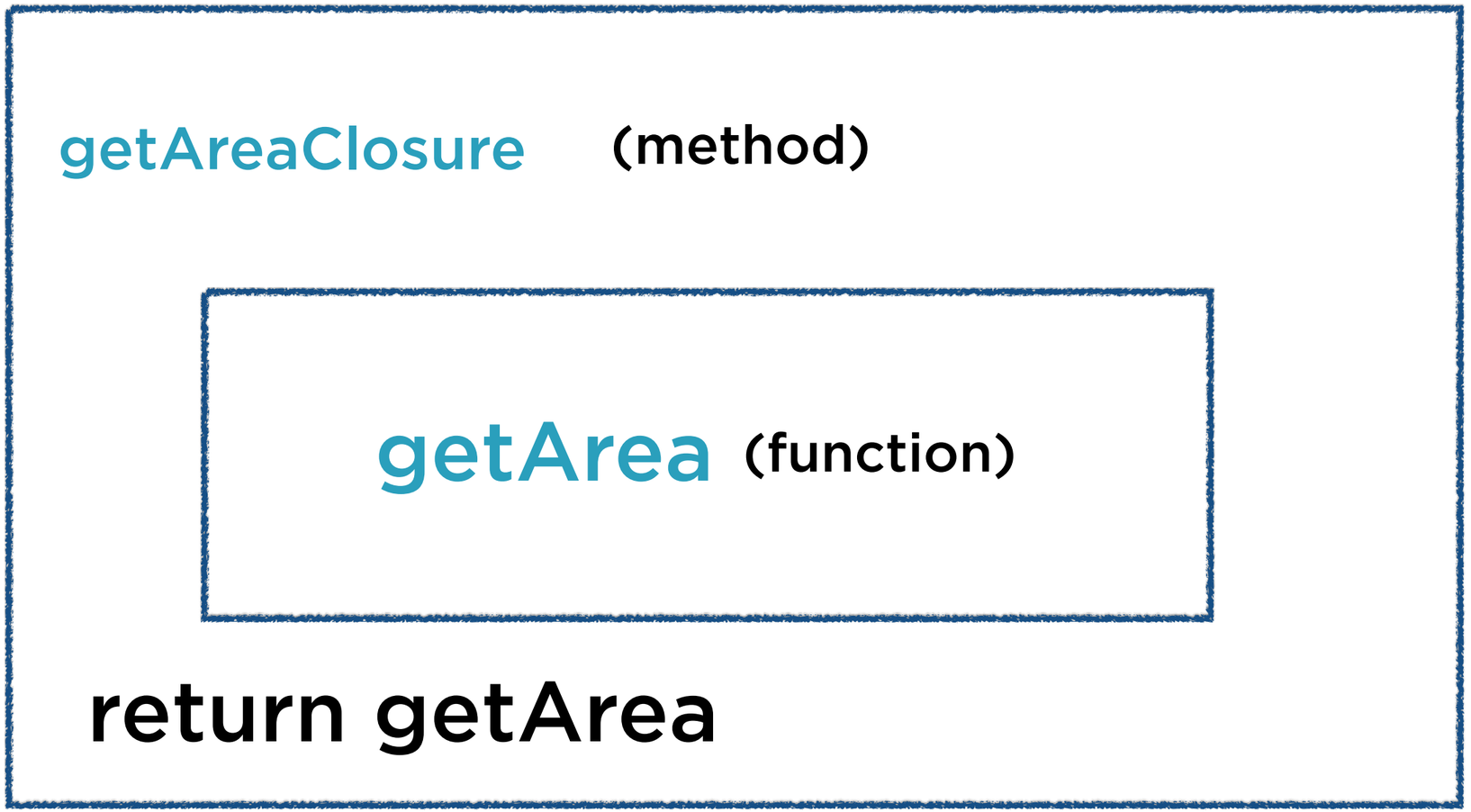
Functions are objects

Functions and methods are subtly different

Demo

Converting methods to functions

Closures

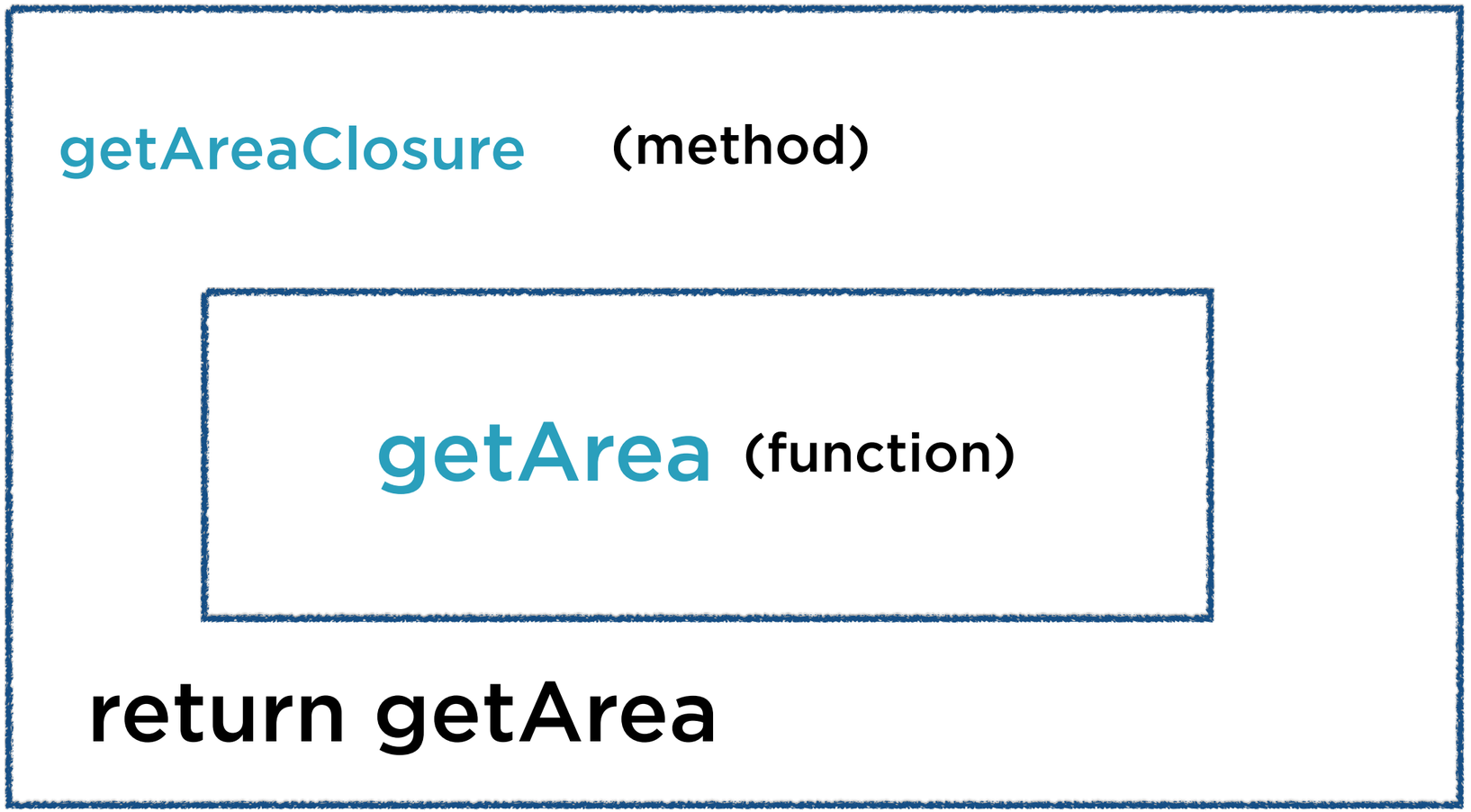


```
val areaCalculator:(Double) => Double = getAreaClosure
```

Explicitly Specifying Type Works

Method getAreaClosure is invoked and function object areaCalculator is correctly assigned return value getArea

Closures



```
val areaCalculator = getAreaClosure _
```

Using Placeholder _ Does Not Work

Method getAreaClosure is converted to function0 object and stored in areaCalculator

First Class Functions

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

Demo

Nested functions

Demo

A higher order function

- A function whose return value is a function
- A function with a parameter that is a function

Code Reuse Mechanisms

Function-level Code Reuse Mechanisms

In Java

Generic functions and function overloading; most reuse is via object-oriented design

In Scala

Rich set of primitives: default values, type parameters, partially applied functions, currying

Code Reuse of Functions

- Parameter default values
- Type parameters
- Partially applied functions
- Currying

Code Reuse of Functions

Parameter default values

Type parameters

Partially applied functions

Currying

Demo

Named function parameters

Demo

Parameter default values

Code Reuse of Functions

Parameter default values

Type parameters

Partially applied functions

Currying

```
List<String> someList = new ArrayList<>();
```

Generics in Java

A mechanism to achieve both type safety and code reuse

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o); // Correct
    }
}
```

Generics in Java

A type parameter is specified, and acts as a placeholder for the type specified by the developer

```
def printPairTypes[K,V](k:K,v:V) = {
  val keyType = k.getClass
  val valueType = v.getClass
  println(s"$k, $v are of types $keyType,$valueType")
}
```

Type Parameters in Scala

Conceptually identical to generics in Java or templates in C++

```
printPairTypes(12,"12");
```

Type Parameters in Scala

Type inference makes using them very simple

Demo

Type parameters and parametric polymorphism

Code Reuse of Functions

Parameter default values

Type parameters

Partially applied functions

Currying

Arity

In logic, mathematics, and computer science, the arity of a function or operation is the number of arguments or operands that the function takes.
(Wikipedia)

F(w,x,y,z)

A Quaternary Function

This function has arity = 4, since it takes in 4 inputs

$$F(w = W_1, x, y = Y_1, z = Z_1)$$

A Quaternary Function with Three Inputs Specified

This function has arity = 1, since it takes in just 1 input

$$G(x) = F(w = W_1, x, y = Y_1, z = Z_1)$$

A New, Partially Specified Function with Reduced Arity

G(x) is a partially specified version of F(w,x,y,z)

$$G(x) = F(w = W_1, x, y = Y_1, z = Z_1)$$

A New, Partially Specified Function with Reduced Arity

G(x) is a partially specified version of F(w,x,y,z)

Partially Applied Functions

In Java

No direct language support - can do using function overloading

In Scala

Explicitly available as language feature

Demo

Partially applied functions

Code Reuse of Functions

- Parameter default values
- Type parameters
- Partially applied functions
- Currying

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

String Comparator

Takes in two strings, a comparison function, and applies that function to the two strings

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

Conceptually Different Types of Parameters

The strings to be compared are one type of parameter, the comparison function is a different type

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

Parameter Groups

Scala allows parameters to be grouped according to semantics

```
curriedCompare(compareStrings)("abc","xyz")
```

Invoking Functions with Parameter Groups

Multiple pairs of parentheses - unusual

```
val defaultCompare = curriedCompare(compareStrings)
(_:String,_:String)
```

Currying

Partially specifying an entire parameter group to reduce the arity

```
val defaultCompare = curriedCompare(compareStrings)
(_:String,_:String)
```

Currying

Partially specifying an entire parameter group to reduce the arity

```
defaultCompare("abc","xyz")
```

Invoking a Curried Function

Now one fewer pair of parentheses, as value of one parameter group has been specified

Demo

Currying

New Semantics, New Capabilities

Fresh New Capabilities

By-name Parameters

Function parameters that auto-refresh each time they are referenced

Closures

Functions that carry around their environment with them

Declaring By-name Function Parameters

```
def sayHelloTo(name:String)
```

Regular Function Parameters

```
def sayHelloTo(name => String)
```

By-name Function Parameters

Literals and By-name Function Parameters

```
sayHelloTo("Vitthal")
```

Regular Function Parameters

Invoked with a literal

```
sayHelloTo("Vitthal")
```

By-name Function Parameters

Invoked with a literal - behave just like regular parameters

Function Calls and By-name Function Parameters

```
sayHelloTo(getTopScorer())
```

Regular Function Parameters

Invoked with a function call

```
sayHelloTo(getTopScorer())
```

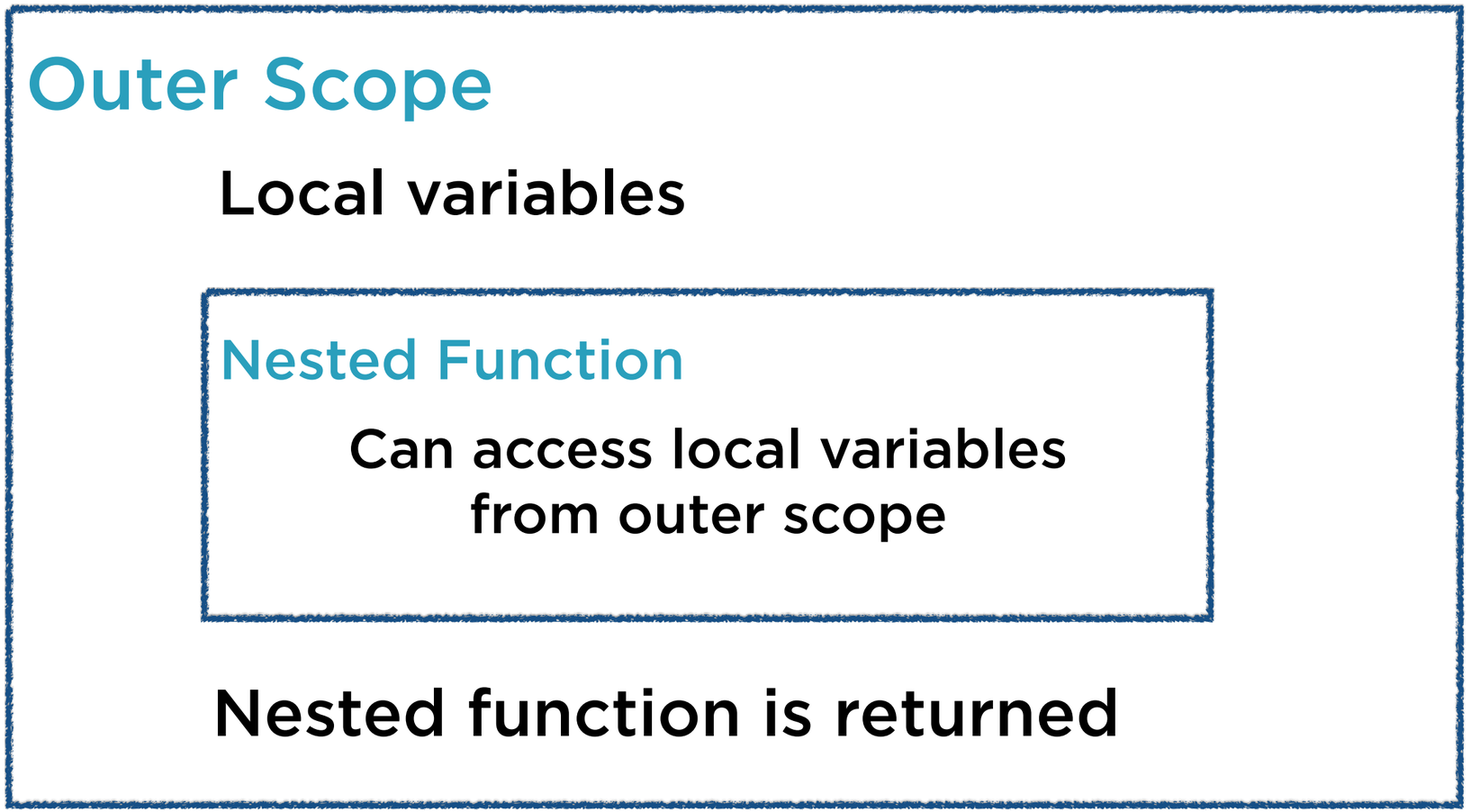
By-name Function Parameters

Invoked with a function call - each reference to this parameter will re-evaluate the function

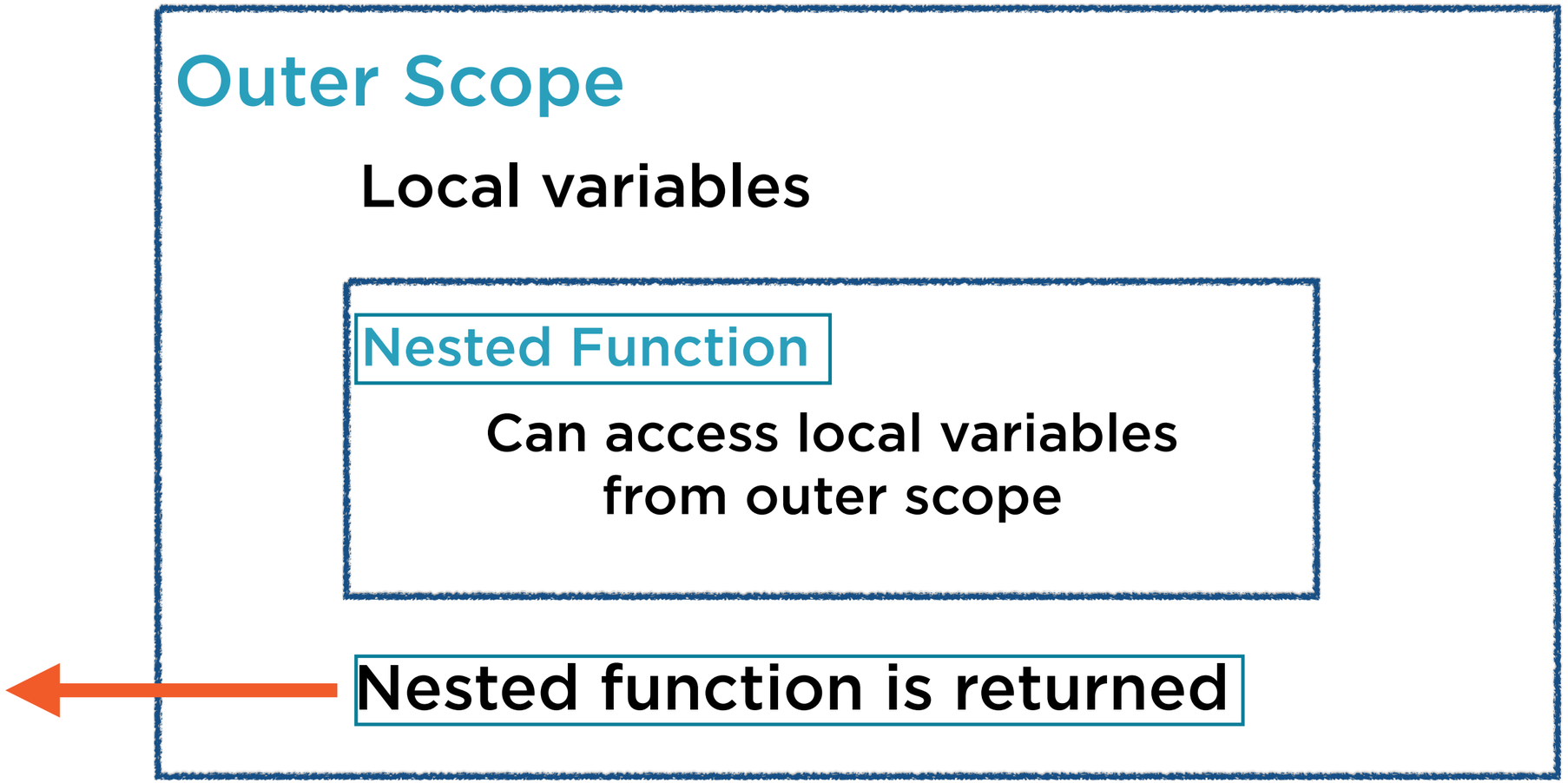
Demo

By-name parameters

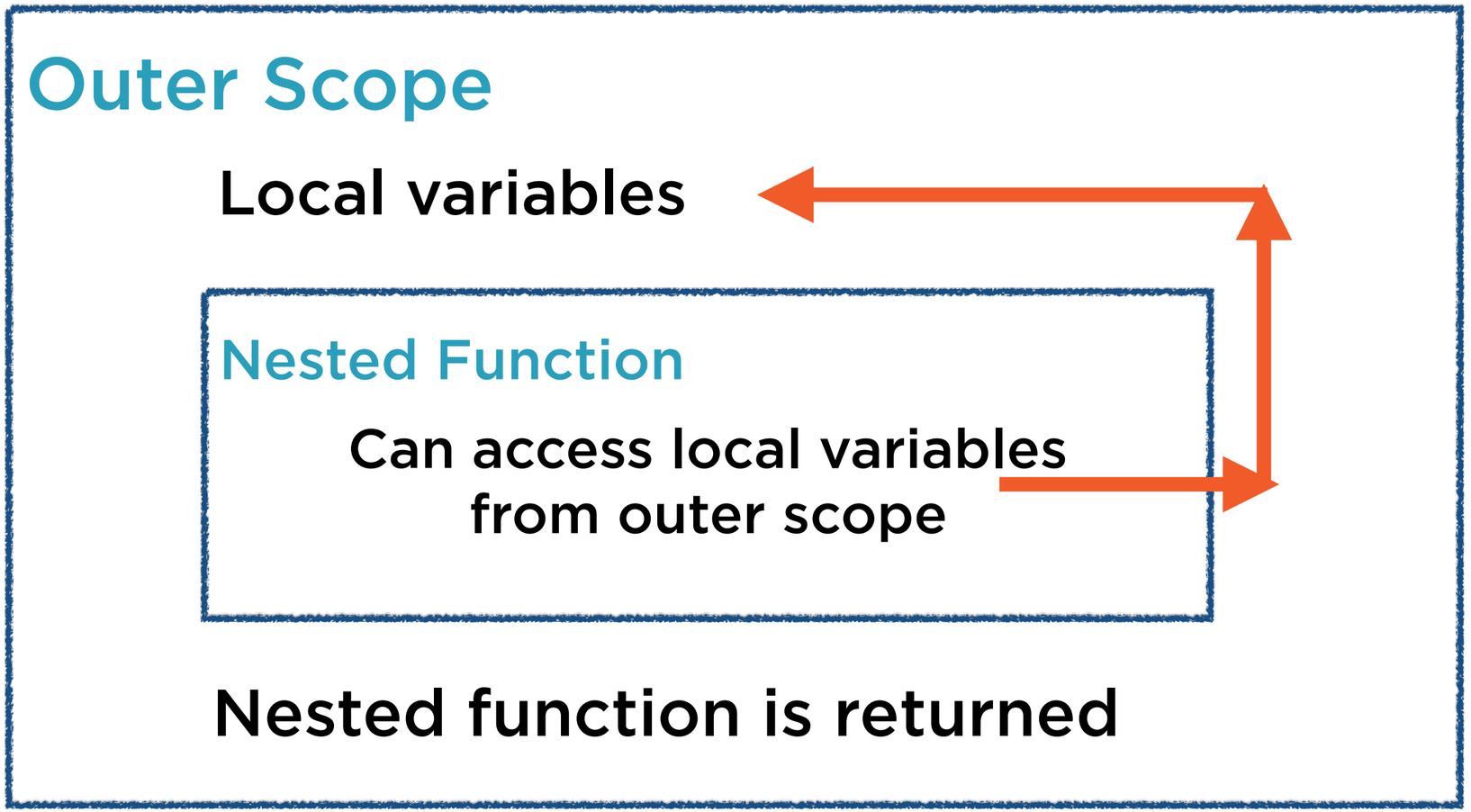
Closures


Outer Scope
Local variables
Nested Function
Can access local variables
from outer scope
Nested function is returned

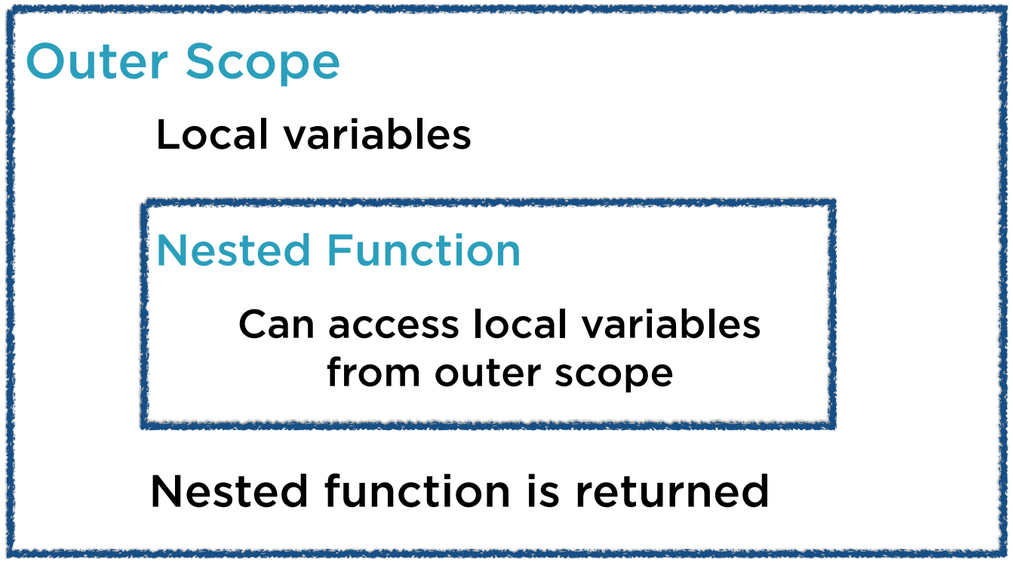
Closures


Outer Scope
Local variables
Nested Function
Can access local variables
from outer scope
Nested function is returned

Closures

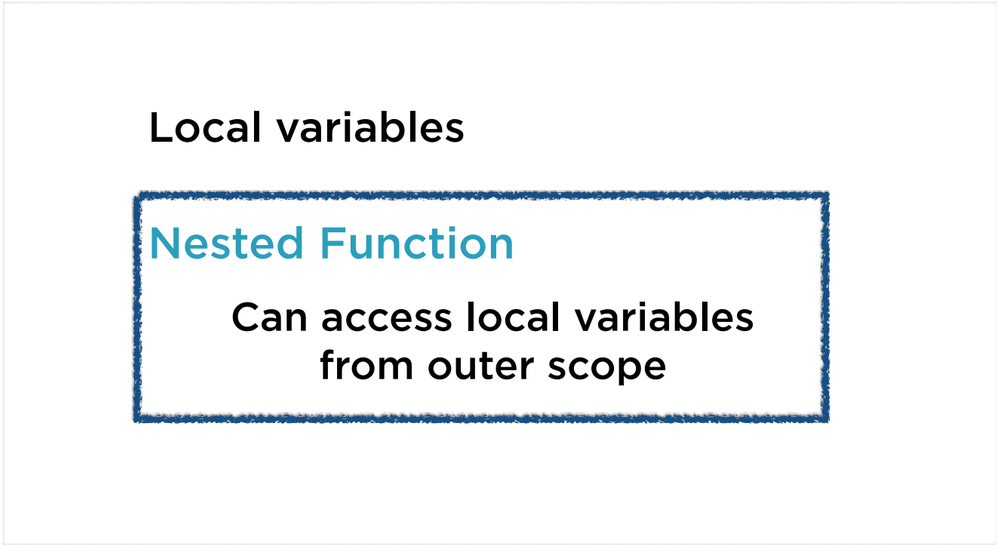

Outer Scope
Local variables
Nested Function
Can access local variables
from outer scope
Nested function is returned

Closures



The returned nested function is called a closure

Closures



The closure retains its copies of local variables - even after the outer scope ceases to exist

Demo

Closures

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

Summary

Mastered the nitty-gritty of creating and invoking functions

Understood Scala support for first class functions

Applied code reuse techniques

Used closures and by-name parameters