

Functions are objects, methods are not

Functions are objects, methods are not

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Functions are objects, methods are not

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Methods Are Not Objects

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

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

Functions Are Objects

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

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

Functions and Methods - Differences

Functions | Methods

Objects of type
`function0,function1,
..(traits descending
from AnyRef)`

Not objects, but can
be converted to
function objects
quite easily

Functions and Methods - Differences

Functions

Functions have both
a type and a
signature (type is
`function`,
`function1`,
`..`)

Methods

Methods have
signature, but no
independent type

Functions and Methods - Differences

Functions | Methods

Slightly slower, and
higher overhead

Slightly faster and
better performing

Functions and Methods - Differences

Functions | Methods

Are first class entities on par with classes

Are not first class entities unless converted to functions

Functions and Methods - Differences

Functions

Do not accept type parameters or parameter default values

Methods

Work fine with type parameters and parameter default values

**Methods are associated with a
class, functions are not**

Applies only to functions,
not to methods

Example 22

Functions are named,
reusable expressions

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

**Functions are named,
reusable expressions**

Let's recall an important distinction

Statements v Expressions

Applies only to functions,
not to methods

Statements v Expressions

Let's draw a distinction between these two -
Scala will make a lot more sense once we do

Applies only to functions,
not to methods

Statements

are units of code that do not return a value

```
[scala> val radius = 10  
radius: Int = 10
```

Applies only to functions,
not to methods

Statements

are units of code that do not return a value

```
scala> println("hello world")
hello world
```

Applies only to functions,
not to methods

Statements v Expressions

are units of code that do not return a value

Applies only to functions,
not to methods

Expressions

are units of code that return a value

Applies only to functions,
not to methods

Expressions

are units of code that return a value

unit of code

```
scala> "hello world"  
res51: String = hello world
```

Applies only to functions,
not to methods

Expressions

are units of code that return a value

```
scala> "hello world"  
res51: String = hello world
```

return value

Applies only to functions,
not to methods

Expressions

are units of code that return a value

```
scala> val radius = 10    expression
radius: Int = 10
```

```
scala> val area = { val PI = 3.14; PI * radius * radius}
area: Double = 314.0
```

Applies only to functions,
not to methods

Expressions

are units of code that return a value

```
scala> val radius = 10  
radius: Int = 10
```

statement

```
scala> val area = { val PI = 3.14; PI * radius * radius}  
area: Double = 314.0
```

Applies only to functions,
not to methods

Expressions

are units of code that return a value

```
scala> val radius = 10  
radius: Int = 10
```

```
scala> val area = { val PI = 3.14; PI * radius * radius}  
area: Double = 314.0
```

unit of code

Applies only to functions,
not to methods

Expressions

are units of code that return a value

```
scala> val radius = 10  
radius: Int = 10
```

```
scala> val area = { val PI = 3.14; PI * radius * radius}  
area: Double = 314.0
```

return value

Applies only to functions,
not to methods

Expressions

are units of code that return a value

```
scala> val radius = 10  
radius: Int = 10
```

```
scala> val area = { val PI = 3.14; PI * radius * radius}  
area: Double = 314.0
```

statement

Applies only to functions,
not to methods

Expressions

are units of code that return a value

```
scala> val radius = 10  
radius: Int = 10
```

```
scala> val area = { val PI = 3.14; PI * radius * radius}  
area: Double = 314.0
```

“expression block”

A bit of code enclosed in {}

Applies only to functions,
not to methods

Expressions

are units of code that return a value

```
scala> val radius = 10  
radius: Int = 10
```

The last expression in a block is the return value for the entire block

```
scala> val area = { val PI = 3.14; PI * radius * radius}  
area: Double = 314.0
```

“expression block”

A bit of code enclosed in {}

Applies only to functions,
not to methods

“Expression block”

A bit of code enclosed in {}

The last expression in a block is the return value for the entire block

Applies only to functions,
not to methods

The last expression in a block is the
return value for the entire block

Applies only to functions,
not to methods

The last expression in a block is the
return value for the entire block

Applies only to functions,
not to methods

“Expression block”

A bit of code enclosed in {}

The last expression in a block is the return value for the entire block

Applies only to functions,
not to methods

Expressions

are units of code that return a value

```
scala> val radius = 10  
radius: Int = 10
```

The last expression in a block is the return value for the entire block

```
scala> val area = { val PI = 3.14; PI * radius * radius}  
area: Double = 314.0
```

“expression block”

A bit of code enclosed in {}

Applies only to functions,
not to methods

Expressions

are units of code that return a value

```
scala> val radius = 10  
radius: Int = 10
```

```
scala> val area = { val PI = 3.14; PI * radius * radius}  
area: Double = 314.0
```

Statements

are units of code that do not return a value

```
scala> println("hello world")  
hello world
```

Applies only to functions,
not to methods

Once we think about it like that, it becomes clear that..

Functions are named,
reusable expressions

Expressions can be stored in values or variables and passed into functions..

Applies only to functions,
not to methods

Functions are named, reusable expressions

Expressions can be stored in values or
variables and passed into functions..

("First Class
Functions")

(More in a bit)

Applies only to functions,
not to methods

Functions are named, reusable expressions

Expressions have scopes and are influenced
by variables from lots of different scopes

("Closures")
(More in a bit)

Applies only to functions,
not to methods

Once we think about it like that, it becomes clear that..

Functions are named,
reusable expressions

Expressions can be stored in values or variables and passed into functions..

Applies only to functions,
not to methods

Functions are named, reusable expressions

Creating a function

```
val getRectangleArea =  
(length:Double, breadth:Double) =>  
{length * breadth}: Double
```

Invoking a function

```
val area = getRectangleArea(5,8)
```

Applies only to functions,
not to methods

Creating a function

```
val getRectangleArea =  
(length:Double, breadth:Double) =>  
{length * breadth}: Double
```

An expression

Applies only to functions,
not to methods

Creating a function

```
val getRectangleArea =  
(length:Double, breadth:Double) =>  
{length * breadth}: Double
```

A name for that
expression

An expression

Applies only to functions,
not to methods

Creating a function

```
val getRectangleArea =  
  (length:Double, breadth:Double) =>  
  {length * breadth}: Double
```

Parameters that make
the expression reusable

An expression

A name for that
expression

Applies only to functions,
not to methods

Creating a function

```
val getRectangleArea =  
(length:Double, breadth:Double) =>  
{length * breadth}: Double
```

The return type of that expression

An expression

A name for that expression

Parameters that make the expression reusable

Applies only to functions, not to methods

Creating a function

```
val getRectangleArea =  
(length:Double, breadth:Double) =>  
{length * breadth}: Double
```

Btw, we could omit the type, Scala would infer it via Type Inference

An expression

A name for that expression

Parameters that make the expression reusable

The return type of that expression

Applies only to functions, not to methods

Creating a function

```
val getRectangleArea =  
(length:Double, breadth:Double) =>  
{length * breadth}: Double
```

An expression

Parameters that make
the expression reusable

A name for that
expression

The return type of that
expression

Applies only to functions,
not to methods

Creating a function

An expression

A name for that expression

Parameters that make the expression reusable

The return type of that expression

Functions are named, reusable expressions

Applies only to functions,
not to methods

Creating a function

```
val getRectangleArea =  
(length:Double, breadth:Double) =>  
{length * breadth}: Double
```

An expression

Parameters that make
the expression reusable

A name for that
expression

The return type of that
expression

Applies only to functions,
not to methods

Functions are named, reusable expressions

Creating a function

```
val getRectangleArea =  
(length:Double, breadth:Double) =>  
{length * breadth}: Double
```

Invoking a function

```
val area = getRectangleArea(5,8)
```

Applies only to functions,
not to methods

Invoking a function

```
val area = getRectangleArea(5,8)
```

Invoking a function is pretty intuitive,
nothing strange going on here:-)

There is another way of invoking functions
- involving expression blocks - more in a bit!

Applies only to functions,
not to methods

Applies only to methods,
not to functions

Example 23

Assigning Methods to Values

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Assigning Methods to Values

Remember

Functions are named,
reusable expressions

Applies only to methods,
not to functions

Assigning Methods to Values

Functions are named,
reusable expressions

Methods are not.

Applies only to methods,
not to functions

Assigning Methods to Values

Functions are named,
reusable expressions

Methods are not.

Converting methods
to functions is easy

Applies only to methods,
not to functions

Functions are named, reusable expressions

Expressions can be stored in values or
variables and passed into functions..

("First Class
Functions")

Applies only to methods,
not to functions

Functions can be stored in values or variables

But there are some nuances worth paying attention to here

```
def getCircleArea(r:Double):Double = PI * r * r
```

Let's start with a simple method

Applies only to methods,
not to functions

This method takes in a Double, and returns a Double

```
def getCircleArea(r:Double):Double = PI * r * r
```

Let's try and simply assign it to a value

```
val calcCircleArea = getCircleArea
```

(Notice we have not specified a type for the value `calcCircleArea`)

Applies only to methods,
not to functions

This method takes in a Double, and returns a Double

```
def getCircleArea(r:Double):Double = PI * r * r
```

Let's try and simply assign it to a value

```
val calcCircleArea = getCircleArea
```

(Notice we have not specified a type for the value `calcCircleArea`)

Applies only to methods,
not to functions

This method takes in a Double, and
returns a Double

```
def getCircleArea(r:Double):Double = PI * r * r
```

This will not work.

```
[scala] val calcCircleArea = getCircleArea
<console>:14: error: missing argument list for method getCircleArea
Unapplied methods are only converted to functions when a function type is expect
ed.
You can make this conversion explicit by writing `getCircleArea _` or `getCircle
Area(_)` instead of `getCircleArea`.
    val calcCircleArea = getCircleArea
^
```

(Notice we have not specified a type
for the value calcCircleArea.)

Applies only to methods,
not to functions

```
val calcCircleArea = getCircleArea
```

This will not work.

Scala is not sure whether you meant to:

- Invoke the method `getCircleArea` and store the return value in `calcCircleArea`
- Store the method `getCircleArea` itself in the value `calcCircleArea`

Applies only to methods,
not to functions

```
val calcCircleArea = getCircleArea
```

This will not work.

Scala is not sure whether you meant to:

- Invoke the method `getCircleArea` and store the return value in `calcCircleArea`
(Had we meant the first, we also ought to have specified the parameter)
- Store the method `getCircleArea` itself in the value `calcCircleArea`

Applies only to methods,
not to functions

```
val calcCircleArea = getCircleArea
```

This will not work.

Scala is not sure whether you meant to:

- Invoke the method `getCircleArea` and store the return value in `calcCircleArea`
Clearly though, this is what we had in mind
- Store the method `getCircleArea` itself in the value `calcCircleArea`

Applies only to methods,
not to functions

Store the method `getCircleArea` itself in the value `calcCircleArea`

```
val calcCircleArea = getCircleArea
```

This will not work.

There are 2 ways to get around this:

Explicitly specify that the signatures of `calcCircleArea` and `getCircleArea` match

Use a wildcard to specify that `calcCircleArea` has whatever signature `getCircleArea` does

Applies only to methods,
not to functions

Explicitly specify that the signatures of
`calcCircleArea` and `getCircleArea` match

Err.what is the type of
`getCircleArea`?

```
def getCircleArea(r:Double):Double = PI * r * r
```

Applies only to methods,
not to functions

Err.what is the type of getCircleArea?

```
def getCircleArea(r:Double):Double = PI * r * r
```

This method takes in a Double, and
returns a Double

The signature of a method is simply the combination
of the types of its parameters and its return value

Applies only to methods,
not to functions

The signature of a method is simply the combination of the types of its parameters and its return value

Applies only to methods,
not to functions

The signature of a method is simply the combination of the types of its parameters and its return value

methods have signatures, but no independent types

Applies only to methods,
not to functions

The signature of a method is simply the combination of
the types of its parameters and its return value

(Double) goes in

```
def getCircleArea(r:Double):Double = PI * r * r
```

Applies only to methods,
not to functions

The signature of a method is simply the combination of the types of its parameters and its return value

Double comes out

```
def getCircleArea(r:Double):Double = PI * r * r
```

Applies only to methods,
not to functions

The signature of a method is simply the combination of the types of its parameters and its return value

(Double) goes in => Double comes out

```
def getCircleArea(r:Double):Double = PI * r * r
```

Applies only to methods,
not to functions

The signature of a method is simply the combination of the types of its parameters and its return value

(Double) => Double

```
def getCircleArea(r:Double):Double = PI * r * r
```

Applies only to methods,
not to functions

```
def getCircleArea(r:Double):Double = PI * r * r
```

The signature of a method is simply the combination of the types of its parameters and its return value

(Double) => Double

Applies only to methods,
not to functions

Err.what is the signature of getCircleArea?

```
def getCircleArea(r:Double):Double = PI * r * r
```

This method takes in a Double, and returns a Double

The signature of a method is simply the combination
of the types of its parameters and its return value

(Double) => Double

Applies only to methods,
not to functions

Explicitly specify that the types of
`calcCircleArea` and `getCircleArea` match

Specify the signature of `calcCircleArea`

(Double) => Double

```
[scala] val calcCircleArea: (Double) => Double = getCircleArea
calcCircleArea: Double => Double = <function1>
```

```
[scala] calcCircleArea(10)
res4: Double = 314.0
```

Applies only to methods,
not to functions

Explicitly specify that the types of
`calcCircleArea` and `getCircleArea` match

Specify the signature of `calcCircleArea`

`(Double) => Double`

```
[scala] val calcCircleArea: (Double) => Double = getCircleArea
calcCircleArea: Double => Double = <function1>
```

Applies only to methods,
not to functions

Explicitly specify that the types of
`calcCircleArea` and `getCircleArea` match

Specify the signature of `calcCircleArea`

`(Double) => Double`

```
[scala] val calcCircleArea: (Double) => Double = getCircleArea
calcCircleArea: Double => Double = <function1>
```

Applies only to methods,
not to functions

This will not work.

```
val calcCircleArea = getCircleArea
```

There are 2 ways to get around this:



Explicitly specify that the signatures of `calcCircleArea` and `getCircleArea` match

Use a wildcard to specify that `calcCircleArea` has whatever signature `getCircleArea` does

Applies only to methods,
not to functions

Use a wildcard to specify that `calcCircleArea` has whatever signature `getCircleArea` does

Actually, the error message tipped us off to this one!

```
[scala] val calcCircleArea = getCircleArea  
<console>:14: error: missing argument list for method getCircleArea  
Unapplied methods are only converted to functions when a function type is expected.
```

You can make this conversion explicit by writing `'getCircleArea _'` or `'getCircleArea(_)' instead of 'getCircleArea'.`

```
val calcCircleArea = getCircleArea  
^
```

And of course, that will work, the compiler would not lie to us :-)

Applies only to methods,
not to functions

Use a wildcard to specify that `calcCircleArea` has whatever signature `getCircleArea` does

Actually, the error message tipped us off to this one!

```
[scala] val calcCircleArea = getCircleArea
<console>:14: error: missing argument list for method getCircleArea
Unapplied methods are only converted to functions when a function type is expected.
```

You can make this conversion explicit by writing `'getCircleArea _'` or `'getCircleArea(_)' instead of 'getCircleArea'.`

```
val calcCircleArea = getCircleArea
^
```

And of course, that will work, the compiler would not lie to us :-)

Applies only to methods,
not to functions

Use a wildcard to specify that `calcCircleArea` has whatever signature `getCircleArea` does

```
val calcCircleArea = getCircleArea _
```

This is known as *eta expansion*

Eta-expansion converts an expression of method type to an equivalent expression of function type.

Applies only to methods,
not to functions

This will not work.

```
val calcCircleArea = getCircleArea
```

There are 2 ways to get around this:



Explicitly specify that the signatures of `calcCircleArea` and `getCircleArea` match



Use a wildcard to specify that `calcCircleArea` has whatever signature `getCircleArea` does

Applies only to methods,
not to functions

Applies only to methods,
not to functions

Let's do a couple more examples

A simple method that takes in no parameters, and returns a double

```
def getPI() = {PI}
```

What's the type of this method?

```
() => Double
```

How would we assign this method to a value?

```
val calcPI: () => Double = getPI //method 1
```

```
val calcPI = getPI _ //method 2
```

Applies only to methods,
not to functions

Let's do a couple more examples

A simple method that takes in 2 parameters, and returns a double

```
def getRectangleArea(l:Double,b:Double):Double = l*b
```

What's the type of this method?

(Double, Double) => Double

How would we assign this function to a method?

//method 1

```
val calcRectangleArea:(Double,Double) => Double = getRectangleArea
```

```
val calcRectangleArea = getRectangleArea _ //method 2
```

Applies only to functions,
not to methods

Example 24

Invoking Functions with Tuples as Parameters

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Invoking Functions with Tuples as Parameters

Creating a method/
function

```
def getRectangleArea  
(length:Double, breadth:Double):Double =  
{length * breadth}
```

Invoking a method/function

```
val area = getRectangleArea(5,8)
```

Applies only to functions,
not to methods

Invoking Functions with Tuples as Parameters

```
val area = getRectangleArea(5, 8)
```

Invoking a function is pretty intuitive,
nothing strange going on here:-)

We had said “There is another way of invoking functions - involving expression blocks - more in a bit!”

Applies only to functions,
not to methods

**Aim: invoke the function/method with
an expression block returning a tuple**

Applies only to functions,
not to methods

**Why? To be able to avoid
breaking chains of function calls**

Applies only to functions,
not to methods

**This can only be done with
functions, not methods**

(getRectangleArea _).tupled()

**Applies only to functions,
not to methods**

Invoking Functions with Tuples as Parameters

```
val area = getRectangleArea(5, 8)
```

Now, let's say we had a square,
whose perimeter we knew

And we wanted to calculate the area of this square, using the function `getRectangleArea`

Applies only to functions,
not to methods

Now, let's say we had a square,
whose perimeter we knew

```
val perimeterOfSquare = 20.0
```

And we wanted to calculate the area of this square, using the method `getRectangleArea`

Step 1: Find the length of the side of the square

```
val sideOfSquare = perimeterOfSquare / 4
```

Step 2: call `getRectangleArea` with this length

```
val area = getRectangleArea(sideOfSquare,  
                           sideOfSquare)
```

Applies only to functions,
not to methods

Now, let's say we had a square,
whose perimeter we knew

This will work, of course,
but its a bit long-winded..

And we wanted to calculate the area of this
square, using the method `getRectangleArea`

Step 1: Find the length of the side of the square
Let's combine the two steps
into one

Step 2: call `getRectangleArea` with this length

```
val area = getRectangleArea(sideOfSquare,  
                           sideOfSquare)
```

Applies only to functions,
not to methods

Now, let's say we had a square,
whose perimeter we knew

```
val perimeterOfSquare = 20.0
```

And we wanted to calculate the area of this square, using the method `getRectangleArea`

Step 1: Find the length of the side of the square

```
val sideOfSquare = perimeterOfSquare / 4
```

Step 2: call `getRectangleArea` with this length

```
val area = getRectangleArea(sideOfSquare,  
                           sideOfSquare)
```

Applies only to functions,
not to methods

Now, let's say we had a square,
whose perimeter we knew

```
val perimeterOfSquare = 20.0
```

And we wanted to calculate the area of this
square, using the function `getRectangleArea`

```
val area = getRectangleArea(perimeterOfSquare/4  
, perimeterOfSquare/4  
)
```

Applies only to functions,
not to methods

Now, let's say we had a square,
whose perimeter we knew

**Scala has an even cooler
way of doing this**

And we wanted to calculate the area of this
square, using the method `getRectangleArea`

**Create an expression block that
returns the length and breadth**

```
val area = getRectangleArea(perimeterOfSquare / 4  
, perimeterOfSquare / 4  
)
```

Applies only to functions,
not to methods

**Scala has an even cooler
way of doing this**

**Create an expression block that
returns the length and breadth**

**Invoke the function using the
result of this expression block**

**Applies only to functions,
not to methods**

Create an expression block that returns the length and breadth

```
{  
    val sideOfSquare = perimeterOfSquare/4;  
    (sideOfSquare, sideOfSquare)  
}
```

Notice the return type
is a tuple of 2 Doubles

```
res122: (Double, Double) = (5.0,5.0)
```

Applies only to functions,
not to methods

Create an expression block that returns the length and breadth

```
{  
    val sideOfSquare = perimeterOfSquare/4;  
    (sideOfSquare, sideOfSquare)  
}
```

Notice the return type
is a tuple of 2 Doubles

res122: (Double, Double) = (5.0,5.0)

Applies only to functions,
not to methods

Create an expression block that returns the length and breadth

```
{  
    val sideOfSquare = perimeterOfSquare/4;  
    (sideOfSquare, sideOfSquare)  
}
```

Notice the return type
is a tuple of 2 Doubles

res122: (Double, Double) = (5.0,5.0)

Applies only to functions,
not to methods

Create an expression block that returns the length and breadth

```
{  
    val sideOfSquare = perimeterOfSquare/4;  
    (sideOfSquare, sideOfSquare)  
}
```

Notice the return type
is a tuple of 2 Doubles

```
res122: (Double, Double) = (5.0,5.0)
```

Applies only to functions,
not to methods

Scala has an even cooler way of doing this

Create an expression block that returns the length and breadth

Invoke the function using the result of this expression block

Applies only to functions,
not to methods

Create an expression block that returns the length and breadth

```
{  
  val sideOfSquare = perimeterOfSquare/4;  
  (sideOfSquare, sideOfSquare)  
}
```

Invoke the function using the result of this expression block

`(getRectangleArea _).tupled()`

Applies only to functions,
not to methods

Create an expression block that returns the length and breadth

```
{  
  val sideOfSquare = perimeterOfSquare / 4;  
  (sideOfSquare, sideOfSquare)  
}
```

This is very strange syntax
- what is going on!?

(getRectangleArea _).tupled()

Applies only to functions,
not to methods

This is very strange syntax
- what is going on!?

(getRectangleArea _) . tupled()

Applies only to functions,
not to methods

Let's break it down..

```
(getRectangleArea _).tupled({val sideOfSquare =  
perimeterOfSquare/4; (sideOfSquare,sideOfSquare)})
```

Applies only to functions,
not to methods

Take an expression block

```
(getRectangleArea _).tupled {val sideOfSquare =  
perimeterOfSquare/4; (sideOfSquare, sideOfSquare)}
```

Applies only to functions,
not to methods

Take an expression block
That returns a tuple

```
(getRectangleArea _).tupled({val sideOfSquare =  
perimeterOfSquare/4; (sideOfSquare, sideOfSquare)})
```

Applies only to functions,
not to methods

Take an expression block
That returns a tuple

```
(getRectangleArea _).tupled {val sideOfSquare =  
perimeterOfSquare/4; (sideOfSquare, sideOfSquare)}
```

Applies only to functions,
not to methods

Take an expression block
That returns a tuple

```
(getRectangleArea _).tupled({val sideOfSquare =  
perimeterOfSquare/4; (sideOfSquare, sideOfSquare)})
```

Convert that tuple into a form that
a function can accept as parameters

“Packing function parameters”

Applies only to functions,
not to methods

Take an expression block
That returns a tuple

Convert our method to a function

```
(getRectangleArea _).tupled({val sideOfSquare =  
perimeterOfSquare/4; (sideOfSquare, sideOfSquare)})
```

Convert that tuple into a form that
a function can accept as parameters

“Packing function parameters”

Applies only to functions,
not to methods

Take an expression block

That returns a tuple

Pass these “**packed parameters**” into our function (converted from a method if need be)

```
(getRectangleArea _).tupled({val sideOfSquare =  
perimeterOfSquare/4; (sideOfSquare, sideOfSquare)})
```

Convert that tuple into a form that a function can accept as parameters

“**Packing function parameters**”

Applies only to functions,
not to methods

Voila, it works

```
(getRectangleArea _).tupled({val sideOfSquare =  
perimeterOfSquare/4; (sideOfSquare,sideOfSquare)})
```

```
res121: Double = 25.0
```

Applies only to functions,
not to methods

Create an expression block that returns the length and breadth

```
{  
    val sideOfSquare = perimeterOfSquare / 4;  
    (sideOfSquare, sideOfSquare)  
}
```

Invoke the function using the result of this expression block

`(getRectangleArea _).tupled()`

(If you have a method,
convert it to a function first)

Applies only to functions,
not to methods

Create an expression block that returns the length and breadth

```
{  
    val sideOfSquare = perimeterOfSquare / 4;  
    (sideOfSquare, sideOfSquare)  
}
```

Why does this matter?

Invoke the function using the result of this expression block

`(getRectangleArea _).tupled()`

Applies only to functions,
not to methods

Why does this matter?

Invoking Functions with Expression Blocks

matters not because the
code is “unreadably cool”

Applies only to functions,
not to methods

Why does this matter?

Invoking Functions with Expression Blocks

matters because this is yet another a way of chaining/composing functions

which is what functional programming is all about

Applies only to functions,
not to methods

Applies only to methods,
not to functions

Example 25

Named Method Parameters

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Named Method Parameters

While invoking a method, you can specify parameters by name

This way, the parameters need not be in the order in which they appear in the method definition

Absolutely nothing changes while defining the method btw.

Applies only to methods,
not to functions

Absolutely nothing changes while defining the method

```
def getRectangleArea(length:Double, breadth:Double) = {  
    println(s"l = $length, b = $breadth");  
    length * breadth  
}
```

Applies only to methods,
not to functions

Named Method Parameters

While invoking a function, you can specify parameters by name

This way, the parameters need not be in the order in which they appear in the function definition

Absolutely nothing changes while defining the function btw.

Applies only to methods,
not to functions

While invoking a method, you can specify parameters by name

```
getRectangleArea(breadth=5, length=10)
```

Applies only to methods,
not to functions

While invoking a method, you can specify parameters by name

```
getRectangleArea(breadth=5, length=10)
```

Applies only to methods,
not to functions

Named Method Parameters

While invoking a method, you can specify parameters by name

This way, the parameters need not be in the order in which they appear in the method definition

Absolutely nothing changes while defining the method btw.

Applies only to methods,
not to functions

```
def getRectangleArea(length:Double, breadth:Double) = {  
    println(s"l = $length, b = $breadth");  
    length * breadth  
}
```

Method Definition

This way, the parameters need not be in the order in which they appear in the method definition

```
[scala] > getRectangleArea(breadth=5, length=10)  
l = 10.0, b = 5.0  
res129: Double = 50.0
```

Method Invocation

Applies only to methods,
not to functions

```
def getRectangleArea(length:Double, breadth:Double) = {  
    println(s"l = $length, b = $breadth");  
    length * breadth  
}
```

Method Definition

This way, the parameters need not be in the order in which they appear in the method definition

```
[scala] > getRectangleArea(breadth=5, length=10)  
l = 10.0, b = 5.0  
res129: Double = 50.0
```

Method Invocation

Applies only to methods,
not to functions

```
def getRectangleArea(length:Double, breadth:Double) = {  
    println(s"l = $length, b = $breadth");  
    length * breadth  
}
```

Method Definition

This way, the parameters need not be in the order in which they appear in the method definition

```
[scala] > getRectangleArea(breadth=5, length=10)  
l = 10.0, b = 5.0  
res129: Double = 50.0
```

Method Invocation

Applies only to methods,
not to functions

```
def getRectangleArea(length:Double, breadth:Double) = {  
    println(s"l = $length, b = $breadth");  
    length * breadth  
}
```

Method Definition

This way, the parameters need not be in the order in which they appear in the method definition

```
[scala] > getRectangleArea(breadth=5, length=10)  
l = 10.0, b = 5.0  
res129: Double = 50.0
```

Method Invocation

Applies only to methods,
not to functions

```
def getRectangleArea(length:Double, breadth:Double) = {  
    println(s"l = $length, b = $breadth");  
    length * breadth  
}
```

Method Definition

This way, the parameters need not be in the order in which they appear in the method definition

```
[scala] > getRectangleArea(breadth=5, length=10)  
l = 10.0, b = 5.0  
res129: Double = 50.0
```

Method Invocation

Applies only to methods,
not to functions

```
def getRectangleArea(length:Double, breadth:Double) = {  
    println(s"l = $length, b = $breadth");  
    length * breadth  
}
```

Method Definition

This way, the parameters need not be in the order in which they appear in the method definition

```
[scala> getRectangleArea(breadth=5, length=10)  
l = 10.0, b = 5.0  
res129: Double = 50.0
```

Method Invocation

Applies only to methods,
not to functions

Named Method Parameters

This works only with
methods, not with functions

```
scala> val getRectangleArea = (length:Double, breadth:Double) => {  
|   println(s"l = $length, b = $breadth");  
|   length * breadth  
| }  
getRectangleArea: (Double, Double) => Double = <function2>
```

```
scala> getRectangleArea(5,10)  
l = 5.0, b = 10.0  
res27: Double = 50.0
```

```
scala> getRectangleArea(breadth=5,length=10)  
<console>:13: error: not found: value breadth  
      getRectangleArea(breadth=5,length=10)  
                           ^  
  
<console>:13: error: not found: value length  
      getRectangleArea(breadth=5,length=10)  
                           ^
```

Applies only to methods,
not to functions

Named Method Parameters

This works only with
methods, not with functions

```
scala> val getRectangleArea = (length:Double, breadth:Double) => {  
|   println(s"l = $length, b = $breadth");  
|   length * breadth  
| }  
getRectangleArea: (Double, Double) => Double = <function2>
```

```
scala> getRectangleArea(5,10)  
l = 5.0, b = 10.0  
res27: Double = 50.0
```

```
scala> getRectangleArea(breadth=5,length=10)  
<console>:13: error: not found: value breadth  
      getRectangleArea(breadth=5,length=10)  
                           ^  
  
<console>:13: error: not found: value length  
      getRectangleArea(breadth=5,length=10)  
                           ^
```

Applies only to methods,
not to functions

Named Method Parameters

While invoking a method, you can specify parameters by name

This way, the parameters need not be in the order in which they appear in the method definition

Absolutely nothing changes while defining the method btw.

Applies only to methods,
not to functions

Why does this matter?

Applies only to methods,
not to functions

Named Method Parameters

Why does this matter?

Named parameters help a lot when calling method
with default parameter values

which we will talk about next

Applies only to methods,
not to functions

Applies only to methods,
not to functions

Example 26

Parameter Default Values

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Parameter Default Values

A method (but not a function) can specify default values for any parameter

The method can then be invoked without specifying a value for that parameter

Of course, the method can be invoked with that parameter too

Applies only to methods,
not to functions

A method can specify default values for any parameter

```
def getCircleStats(r:Double,PI:Double = 3.14) = {  
    def getCircleArea(r:Double) = PI * r * r  
    def getCircleCircumference(r:Double) = 2 * PI * r  
    (getCircleArea(r),getCircleCircumference(r))  
}
```

Applies only to methods,
not to functions

Parameter Default Values

A method (but not a function) can specify default values for any parameter

The method can then be invoked without specifying a value for that parameter

Of course, the method can be invoked with that parameter too

Applies only to methods,
not to functions

The method can then be invoked without specifying a value for that parameter

```
scala> getCircleStats(10)  
res130: (Double, Double) = (314.0,62.80000000000004)
```

Applies only to methods,
not to functions

Of course, the method can be invoked
with that parameter too

```
scala> getCircleStats(10,3.14159)  
res131: (Double, Double) = (314.159,62.8318)00000004
```

Applies only to methods,
not to functions

Of course, the method can be invoked
with that parameter too

```
scala> getCircleStats(10,3.14159)  
res131: (Double, Double) = (314.159,62.8318)
```

Applies only to methods,
not to functions

Of course, the method can be invoked
with that parameter too

```
scala> getCircleStats(10, 3.14159)
res131: (Double, Double) = (314.159,62.8318)
```

Applies only to methods,
not to functions

Parameter Default Values

A method (but not a function) can specify default values for any parameter

The method can then be invoked without specifying a value for that parameter

Of course, the method can be invoked with that parameter too

Applies only to methods,
not to functions

Parameter Default Values

Why does this matter?

Applies only to methods,
not to functions

Parameter Default Values

Why does this matter?

Default parameters are a way to achieve code re-use

Java does not have this feature, and relies on function overloading - which leads to code bloat

C++ allows default parameter values, but with some restrictions on the order of parameters

Applies only to methods,
not to functions

C++ allows default parameter values, but with some restrictions on the order of parameters

In Scala any parameter can have a default value

This is where the ability to specify parameters by name comes in handy

Applies only to methods,
not to functions

In Scala any parameter can have a default value

```
def getCircleStats(PI:Double = 3.14,r:Double) = {  
    def getCircleArea(r:Double) = PI * r * r  
    def getCircleCircumference(r:Double) = 2 * PI * r  
    (getCircleArea(r),getCircleCircumference(r))  
}
```

Applies only to methods,
not to functions

C++ allows default parameter values, but with some restrictions on the order of parameters

In Scala any parameter can have a default value

This is where the ability to specify parameters by name comes in handy

Applies only to methods,
not to functions

This is where the ability to specify parameters by name comes in handy

```
[scala] getCircleStats(10.0)
<console>:16: error: not enough arguments for method getCircleStats: (PI: Double
, r: Double)(Double, Double).
Unspecified value parameter r.
  getCircleStats(10.0)
^
```

Attempting to invoke this method the usual way leads to an error

```
scala> getCircleStats(r=10.0)
res132: (Double, Double) = (314.0,62.80000000000004)
```

But specifying the parameter by name works fine!

Applies only to methods,
not to functions

This is where the ability to specify parameters by name comes in handy

Attempting to invoke this method the usual way leads to an error

```
scala> getCircleStats(r=10.0)
res132: (Double, Double) = (314.0,62.80000000000004)
```

But specifying the parameter by name works fine!

Applies only to methods,
not to functions

Scala tries really hard to encourage code re-use

Default Parameters

Type Parameters like Generic Functions in Java, or templates in C++

Currying

Partially Applied Functions

More on currying and partially applied fun

Applies only to methods, not to functions

Applies only to methods,
not to functions

Example 27

Type Parameters: Parametric Polymorphism

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Type Parameters: Parametric Polymorphism

In Java, it's possible to have a bunch of different types of values in a collection

```
List someList = new ArrayList();
someList.add(2);
someList.add(1);
someList.add("foo");
someList.add(null);
```

You probably don't want this, since we have lost type safety (and also type information)

Applies only to methods,
not to functions

Lost type safety

We can not rely on specific types only being passed in

Lost type information

Any type-specific operations will require dynamic casts, `instanceof`, and other clumsy syntax

Applies only to methods,
not to functions

Lost type safety

We can not rely on specific types only being passed in

In Java, to prevent code like this, simply specify a type for the array list

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

That would prevent type-unsafe code from even compiling!

Applies only to methods,
not to functions

Type Parameters: Parametric Polymorphism

In Scala, it's possible to have a bunch of different types of values in a collection

```
[scala] > val someList = 2 :: 1 :: "bar" :: "foo" :: Nil  
someList: List[Any] = List(2, 1, bar, foo)  
someList.add(1);  
someList.add("foo");  
someList.add(null);
```

In Scala, to prevent code like this, simply . specify a type for the array/list .

```
[scala] > val someList:List[String] = 2 :: 1 :: "bar" :: "foo" :: Nil  
<console>:11: error: type mismatch;  
      found   : Int  
      required: String  
           val someList:List[String] = 2 :: 1 :: "bar" :: "foo"
```

Applies only to methods,
not to functions

Type Parameters: Parametric Polymorphism

In Scala, to prevent code like this, simply specify a type for the array list

```
scala> val someList:List[String] = 2 :: 1 :: "bar" :: "foo" :: Nil
<console>:11: error: type mismatch;
          found   : Int
          required: String
                  val someList:List[String] = 2 :: 1 :: "bar" :: "foo" :: Nil
```

In exactly the same manner, we can specify type parameters for functions

Applies only to methods,
not to functions

Type Parameters: Parametric Polymorphism

In exactly the same manner, we can specify type parameters for functions

This is known as “parametric polymorphism” :-)

Applies only to methods,
not to functions

Type Parameters: Parametric Polymorphism

Let's check out an example of a method that takes in a key-value pair and prints the values and types of both items in the pair

Obviously, we need this method to work for all possible types of keys and values

So - we specify 2 type parameters

Applies only to methods,
not to functions

Let's check out an example of a method that takes in a key-value pair and prints the values and types of both items in the pair

```
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")  
}
```

Applies only to methods,
not to functions

The type parameters are in square brackets as part of the method definition

```
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")  
}
```

Applies only to methods,
not to functions

K is the type of the key, V is the type of the value

```
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")  
}
```

Applies only to methods,
not to functions

K is the type of the key, V is the type of the value

```
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")  
}
```

Applies only to methods,
not to functions

The method parameters now are expressed in terms of the type parameters

```
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")  
}
```

printPairTypes(12, "12");

12, 12 are of types class java.lang.Integer, class java.lang.String

Applies only to methods,
not to functions

The method parameters now are expressed in terms of the type parameters

```
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")  
}
```

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

12, 12 are of types class java.lang.Integer,class java.lang.String

Applies only to methods,
not to functions

The method parameters now are expressed in terms of the type parameters

```
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")  
}
```

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

12, 12 are of types class java.lang.Integer, class java.lang.String

Applies only to methods,
not to functions

The method parameters now are expressed in terms of the type parameters

```
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")  
}
```

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

12, 12 are of types class java.lang.Integer,class java.lang.String

Applies only to methods,
not to functions

The method parameters now are expressed in terms of the type parameters

```
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")  
}
```

printPairTypes(12, "12");

12, 12 are of types class java.lang.Integer,class java.lang.String

Applies only to methods,
not to functions

The method parameters now are expressed in terms of the type parameters

```
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")  
}
```

printPairTypes(12, "12");

12, 12 are of types class java.lang.Integer,class java.lang.String

Applies only to methods,
not to functions

The method parameters now are expressed in terms of the type parameters

```
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")  
}
```

printPairTypes(12, "12");

12, 12 are of types class java.lang.Integer, class java.lang.String

Applies only to methods,
not to functions

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Type Parameters work only with Methods, Not Functions

```
[scala] val printPairTypesFn = printPairTypes _  
printPairTypesFn: (Any, Any) => Unit = <function2>
```

Using the placeholder `_` causes type parameters to be converted to `Any` (i.e. lost)

Applies only to methods,
not to functions

Type Parameters work only with Methods, Not Functions

```
[scala] val printPairTypesFn = printPairTypes _  
printPairTypesFn: (Any, Any) => Unit = <function2>
```

Using the placeholder `_` causes type parameters to be converted to `Any` (i.e. lost)

Applies only to methods,
not to functions

Type Parameters work only with Methods, Not Functions

```
[scala] val printPairTypesFn = printPairTypes[Int, String] -  
printPairTypesFn: (Int, String) => Unit = <function2>
```

We need to explicitly specify the type parameter values for each function object to be properly specified

Applies only to methods,
not to functions

Applies only to methods,
not to functions

Example 28

Vararg Parameters

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Vararg Parameters

Scala, like C and Java, supports vararg parameters

This is an obscure language feature used when the number of arguments is unknown until runtime

`printf`, `String.format` are examples of C & Java functions that use varargs

Applies only to methods,
not to functions

Vararg Parameters

To specify an arg is a vararg, just add an * to the parameter specification

Applies only to methods,
not to functions

Vararg Parameters

To specify an arg is a vararg, just add an * to the parameter specification

```
def concatStrings(strings:String*) = {  
    var builtUpString = ""  
    for (s <- strings) builtUpString = builtUpString + " " + s  
    builtUpString  
}
```

Applies only to methods,
not to functions

Vararg Parameters

To specify an arg is a vararg, just add an * to the parameter specification

```
def concatStrings(strings: String*) = {  
    var builtUpString = ""  
    for (s <- strings) builtUpString = builtUpString + " " + s  
    builtUpString  
}
```

Applies only to methods,
not to functions

Vararg Parameters

To specify an arg is a vararg, just add an * to the parameter specification

```
def concatStrings(strings:String*) = {  
    var builtUpString = ""  
    for (s <- strings) builtUpString = builtUpString + " " + s  
    builtUpString  
}
```

The vararg parameter will be available inside the method as a collection

Applies only to methods,
not to functions

Vararg Parameters

```
def concatStrings(strings:String*) = {  
    var builtUpString = ""  
    for (s <- strings) builtUpString = builtUpString + " " + s  
    builtUpString  
}
```

The vararg parameter will be available inside the method as a collection

Applies only to methods,
not to functions

Vararg Parameters

To specify an arg is a vararg, just add an * to the parameter specification

```
def concatStrings(strings:String*) = {  
    var builtUpString = ""  
    for (s <- strings) builtUpString = builtUpString + " " + s  
    builtUpString  
}
```

Now, call the method with as many parameters as you like!

```
scala> concatStrings("A","B","C")  
res153: String = " A B C"
```

```
scala> concatStrings("A","B","C","D")  
res154: String = " A B C D"
```

Applies only to methods,
not to functions

Vararg Parameters

Now, call the method with as many parameters as you like!

```
scala> concatStrings("A", "B", "C")
res153: String = " A B C"
```

```
scala> concatStrings("A", "B", "C", "D")
res154: String = " A B C D"
```

Applies only to methods,
not to functions

Vararg Parameters

Now, call the method with as many parameters as you like!

```
scala> concatStrings("A","B","C")
res153: String = " A B C"
```

```
scala> concatStrings("A","B","C","D")
res154: String = " A B C D"
```

Applies only to methods,
not to functions

Vararg Parameters

Functions can't take vararg parameters, they get converted to type Seq

```
[scala] val concatStringsFn = concatStrings _  
concatStringsFn: Seq[String] => String = <function1>  
  
[scala] concatStringsFn("A", "B", "C", "D")  
<console>:14: error: too many arguments for method apply: (v1: Seq[String])String  
in trait Function1  
      concatStringsFn("A", "B", "C", "D")  
                           ^
```

Applies only to methods,
not to functions

Vararg Parameters

Functions can't take vararg parameters, they get converted to type Seq

```
[scala] val concatStringsFn = concatStrings _  
concatStringsFn: Seq[String] => String = <function1>  
  
[scala] concatStringsFn("A", "B", "C", "D")  
<console>:14: error: too many arguments for method apply: (v1: Seq[String])String  
in trait Function1  
      concatStringsFn("A", "B", "C", "D")  
           ^
```

Applies only to methods,
not to functions

Vararg Parameters

Functions can't take vararg parameters, they get converted to type Seq

```
| scala> val concatStringsFn = concatStrings _  
| concatStringsFn: Seq[String] => String = <function1>  
  
| scala> concatStringsFn("A", "B", "C", "D")  
<console>:14: error: too many arguments for method apply: (v1: Seq[String])String  
      in trait Function1  
          concatStringsFn("A", "B", "C", "D")  
          ^
```

Applies only to methods,
not to functions

Vararg Parameters

Functions can't take vararg parameters, they get converted to type Seq

```
[scala> concatStringsFn(Seq("A", "B", "C", "D"))
res38: String = " A B C D"
```

Converting to sequence kinda defeats the purpose of vararg parameters though...

Applies only to methods,
not to functions

Vararg Parameters

How does a vararg parameter know where the list of parameters ends?

It does not!!

That's why any varargs parameter must be last in the list of parameters of the method

Applies only to methods,
not to functions

Example 29

Procedures are named,
reusable statements

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Procedures are named,
reusable statements

Let's recall an important distinction

Statements v Expressions

Statements v Expressions

Let's draw a distinction between these two -
Scala will make a lot more sense once we do

Statements

are units of code that do not return a value

```
[scala> val radius = 10  
radius: Int = 10
```

Statements

are units of code that do not return a value

```
scala> println("hello world")
hello world
```

Procedures are named, reusable statements

A function/method that does not return anything is called a **procedure**

A procedure might print to screen, interact with a database or a socket etc

Applies to functions and methods

Procedures are named, reusable statements

Creating a procedure

```
scala> def printHello:Unit = print("Hellooo!")  
printHello: Unit
```

Invoking a procedure

```
| scala> printHello  
| Hellooo!
```

Procedures can be functions or methods

Creating a procedure as a method

```
scala> def printHello:Unit = print("Hellooo!")  
printHello: Unit
```

Procedures can be functions or methods

Can convert to function as usual - nothing changes

```
scala> val printHelloFn = printHello _  
printHelloFn: () => Unit = <function0>
```

Procedures can be functions or methods

Can convert to function as usual - nothing changes

```
scala> val printHelloFn = printHello _  
printHelloFn: () => Unit = <function0>
```

Creating a procedure

```
scala> def printHello:Unit = print("Hellooo!")  
printHello: Unit
```

A statement (or set of statements)

Creating a procedure

```
scala> def printHello:Unit = print("Hellooo!")  
printHello: Unit
```

A name

A statement (or set of statements)

Creating a procedure

```
scala> def printHello:Unit = print("Hellooo!")  
printHello: Unit
```

Notice the return type is
Unit

A name

This procedure has neither inputs nor outputs

A statement (or set of statements)

Creating a procedure with inputs is no different

Creating a procedure

```
scala> def printHello:Unit = print("Hellooo!")  
printHello: Unit
```

Notice the return type is
Unit

Technically it is possible to
omit the return type entirely

```
scala> def printHello = print("Hellooo!")  
printHello: Unit
```

Or even without the equal sign

```
scala> def printHello { print("Hellooo!") }  
printHello: Unit
```

Creating a procedure

```
scala> def printHello:Unit = print("Hellooo!")  
printHello: Unit
```

Notice the return type is
Unit

Technically it is possible to
omit the return type entirely

But this is considered **bad practice** - make it
explicit you are writing a procedure (and not a
function!)

Creating a procedure

```
scala> def printHello:Unit = print("Hellooo!")  
printHello: Unit
```

Notice the return type is
Unit

A name

This procedure has neither inputs nor outputs

A statement (or set of statements)

Creating a procedure with inputs is no different

Procedures are named, reusable statements

Creating a procedure

```
scala> def printHello = print("Hellooo!")
printHello: Unit
```

Invoking a procedure

```
| scala> printHello
| Hellooo!
```

Invoking a procedure

```
| scala> printHello  
| Hellooo!
```

Notice how parentheses are not needed when there are no inputs!

Example 30

Methods with No Inputs

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Methods with No Inputs

methods with no inputs can be defined in 2 slightly different ways

With parentheses

```
scala> def sayHello():String = "Hello"  
sayHello: ()String
```

Without parentheses

```
[scala> def sayHello2:String = "Hello"  
sayHello2: String
```

With Parentheses

```
scala> def sayHello():String = "Hello"  
sayHello: ()String
```

Now this method can be
invoked in 2 ways

With parentheses

```
[scala> sayHello()  
res104: String = Hello
```

Without parentheses

```
[scala> sayHello  
res103: String = Hello
```

Applies to functions and methods

With Parentheses

```
scala> def sayHello():String = "Hello"  
sayHello: ()String
```

Now this method can be
invoked in 2 ways
**All as you would expect, no
weird stuff so far**
With parentheses

```
[scala> sayHello()  
res104: String = Hello
```

Without parentheses

```
[scala> sayHello  
res103: String = Hello
```

Applies to functions and methods

With Parentheses

```
scala> def sayHello():String = "Hello"  
sayHello: ()String
```

Now this method can be
invoked in 2 ways

With parentheses

```
[scala> sayHello()  
res104: String = Hello
```

Without parentheses

```
[scala> sayHello  
res103: String = Hello
```

Applies to functions and methods

Functions with No Inputs

method with no inputs can be defined in 2 slightly different ways



With parentheses

```
scala> def sayHello():String = "Hello"  
sayHello: ()String
```

Without parentheses

```
[scala> def sayHello2:String = "Hello"  
sayHello2: String
```

Without Parentheses

```
[scala] def sayHello2:String = "Hello"  
sayHello2: String
```

Now this method can ONLY be invoked without parentheses

Invoking with parentheses leads to an error

```
scala> sayHello2()  
<console>:13: error: not enough arguments for method apply: (index: Int)Char in  
class StringOps.  
Unspecified value parameter index.  
      sayHello2()  
           ^
```

Without Parentheses

```
[scala> def sayHello2:String = "Hello"  
sayHello2: String
```

Now this method can **ONLY** be invoked without parentheses

Invoking with parentheses leads to an error

Invoking without parentheses works fine

```
scala> sayHello2  
res102: String = Hello
```

Without Parentheses

```
[scala> def sayHello2:String = "Hello"  
sayHello2: String
```

Now this method can ONLY be
invoked without parentheses

This is a little gotcha -
Invoking with parentheses leads to an error

Invoking without parentheses works fine

```
scala> sayHello2  
res102: String = Hello
```

Without Parentheses

The idea is that methods that return immutable values should look like fields

With Parentheses

The idea is that methods that have side effects should look like function calls

Functions with No Inputs

method with no inputs can be defined in 2 slightly different ways



With parentheses

```
scala> def sayHello():String = "Hello"  
sayHello: ()String
```



Without parentheses

```
[scala> def sayHello2:String = "Hello"  
sayHello2: String
```

Applies to functions and methods

Example 31

Nested Functions

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Nested Functions

There is nothing remarkable
about nested functions at all

Remember that functions are simply
named, reusable expression blocks

Nested Functions

Remember that functions are simply named, reusable expression blocks

By the same token, are expression blocks that are inside other expression blocks

Nested Functions

Let's create a simple function that returns both the area and the circumference of a circle

```
val getCircleStats = (r:Double) => {  
    val PI = 3.14  
    val getCircleArea = (r:Double) => PI * r * r  
    val getCircleCircumference = (r:Double) => 2 * PI * r  
    (getCircleArea(r),getCircleCircumference(r))  
}
```

(We will return them as a tuple of doubles)

An obvious way to modularise the function is to have separate nested functions to calculate area and circumference..

Nested Functions

Let's create a simple method that returns both the area and the circumference of a circle

(We will return them as a tuple of doubles)

An obvious way to modularise the method is to have separate nested functions to calculate area and circumference..

Let's create a simple function that returns both the area and the circumference of a circle

```
val getCircleStats = (r:Double) => {  
    val PI = 3.14  
    val getCircleArea = (r:Double) => PI * r * r  
    val getCircleCircumference = (r:Double) => 2 * PI * :  
        (getCircleArea(r),getCircleCircumference(r))  
}
```

separate nested functions to calculate area and circumference..

return them as a tuple of doubles

Applies to functions and methods

Nested Functions

Outer Function
(Or Method)

Inner Function
(or Method)



Output
Function Object

(Function only,
not method)

a simple function that returns both the area and the circumference of a circle

```
val getCircleStats = (r:Double) => {  
    val PI = 3.14  
    val getCircleArea = (r:Double) => PI * r * r  
    val getCircleCircumference = (r:Double) => 2 * PI * :  
        (getCircleArea(r),getCircleCircumference(r))  
}
```

return them as a tuple of doubles
separate nested functions to calculate area and circumference...

Applies to functions and methods

a simple function that returns both the area and the circumference of a circle

```
val getCircleStats = (r:Double) => {  
    val PI = 3.14  
    val getCircleArea = (r:Double) => PI * r * r  
    val getCircleCircumference = (r:Double) => 2 * PI * :  
        (getCircleArea(r),getCircleCircumference(r))  
}
```

separate nested functions to calculate area and circumference..

return them as a tuple of doubles

a simple function that returns both the area and the circumference of a circle

```
val getCircleStats = (r:Double) => {  
    val PI = 3.14  
    val getCircleArea = (r:Double) => PI * r * r  
    val getCircleCircumference = (r:Double) => 2 * PI * r  
    (getCircleArea(r),getCircleCircumference(r))  
}
```

return them as a tuple of doubles

Let's create a simple function that returns both the area and the circumference of a circle

```
val getCircleStats = (r:Double) => {  
    val PI = 3.14  
    val getCircleArea = (r:Double) => PI * r * r  
    val getCircleCircumference = (r:Double) => 2 * PI * :  
        (getCircleArea(r),getCircleCircumference(r))  
}
```

return them as a tuple of doubles

separate nested functions to calculate area and circumference..

Nested Functions

Let's create a simple function that returns both the **Why does this matter?**

(We will return them as a tuple of doubles)

An obvious way to modularise the function is to have separate nested functions to calculate area and circumference..

Nested Functions

Why does this matter?

It extends the idea of a function as a type like any other

Important features hinge on this..

“First Class Functions”

“Closures”

Applies to functions and methods

Example 32

Higher Order Functions

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Higher Order Functions

Remember

Functions are named,
reusable expressions

Remember

Functions are named,
reusable expressions

Expressions can be stored in values or
variables and passed into functions..

Applies to functions and methods

Functions are named, reusable expressions

Expressions can be stored in values or
variables and passed into functions..

("First Class
Functions")

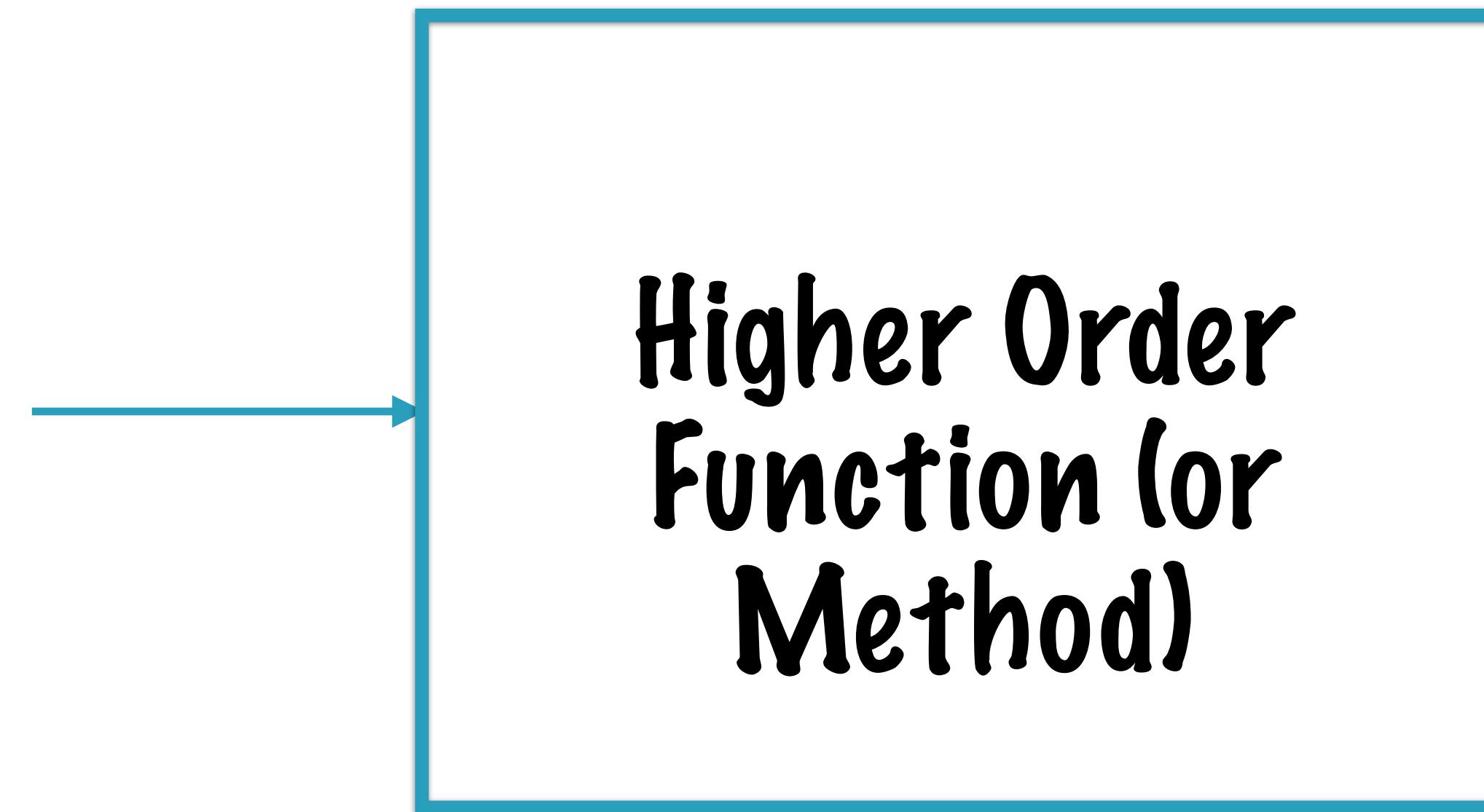
Functions can be passed into functions

A “higher order” function is one that

- Either takes in a function as one of its parameters
- Returns a function as its return value

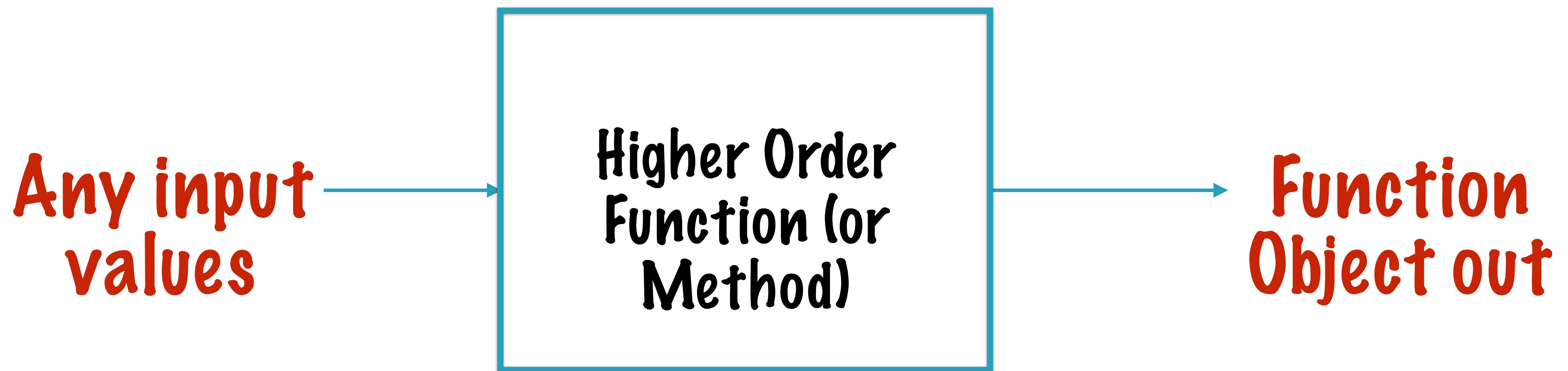
A “higher order” function

Function
object in



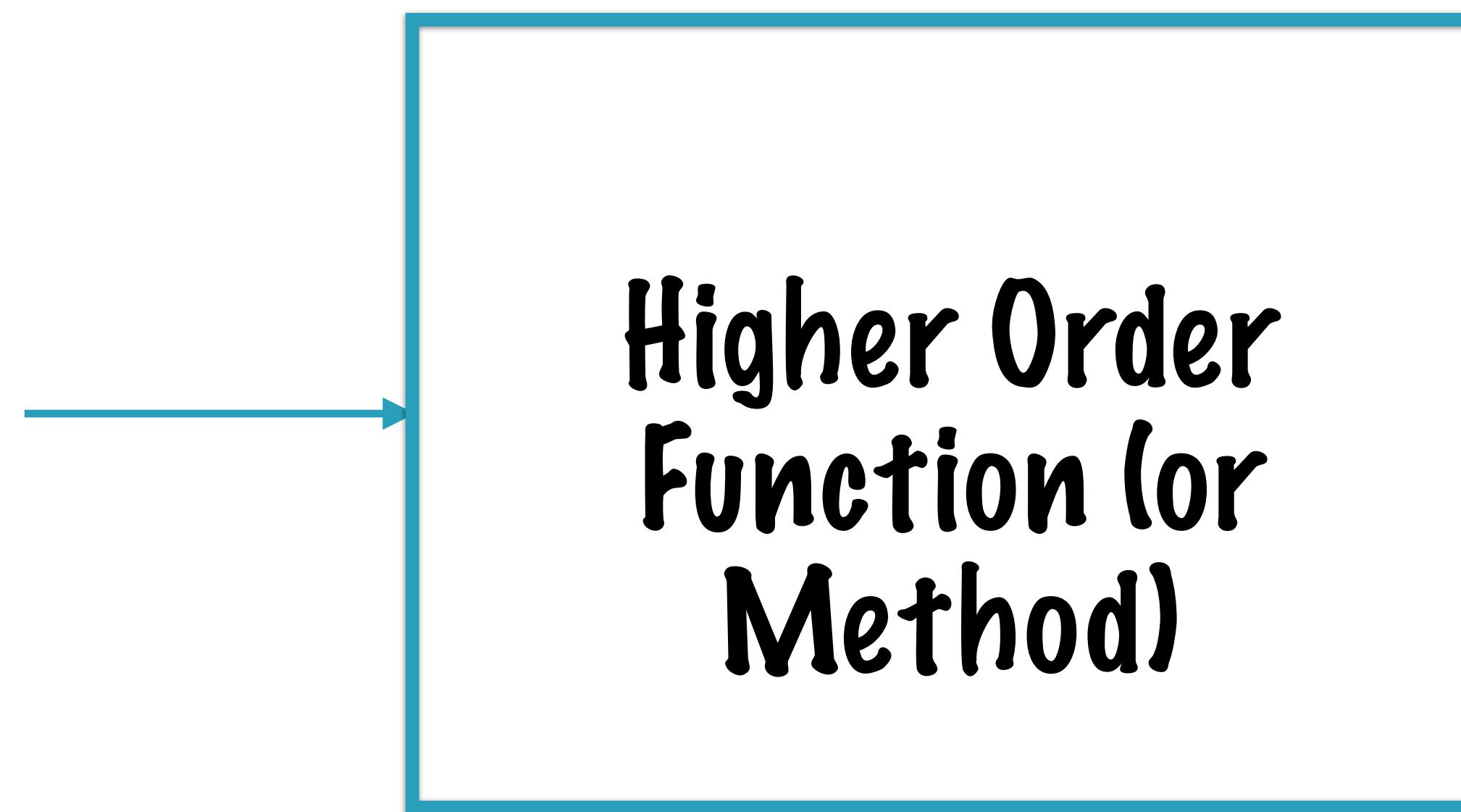
(Any output
value)

A “higher order” function



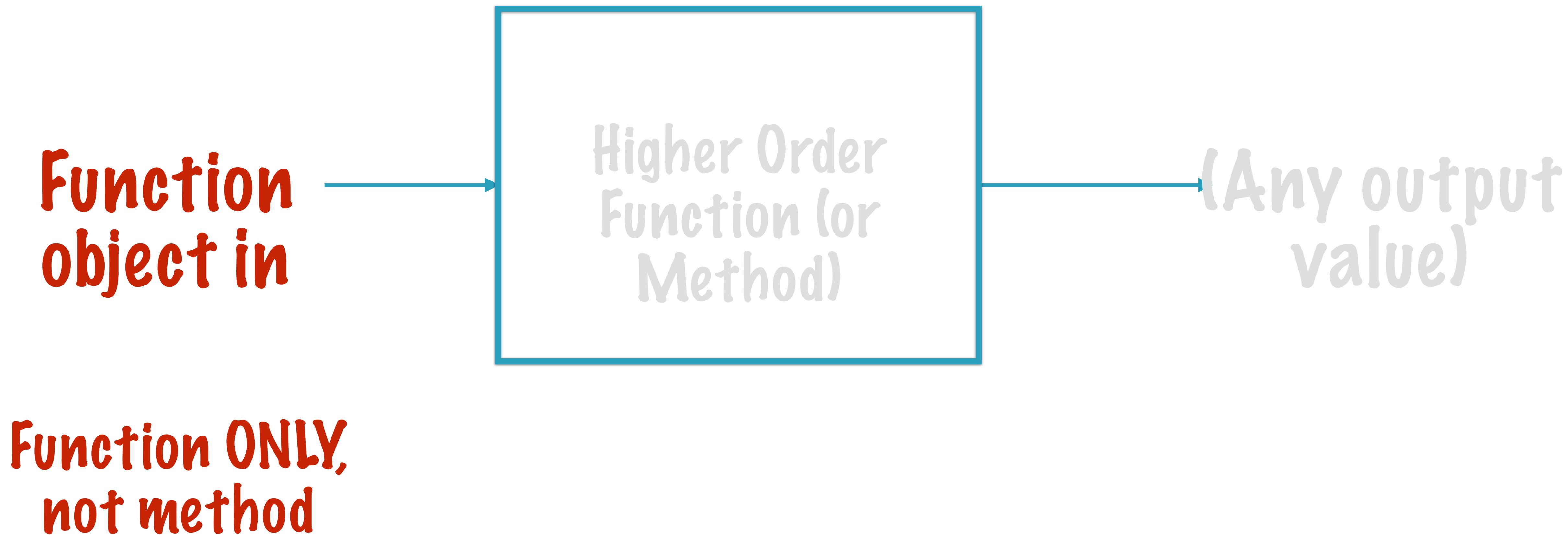
A “higher order” function

Function
object in

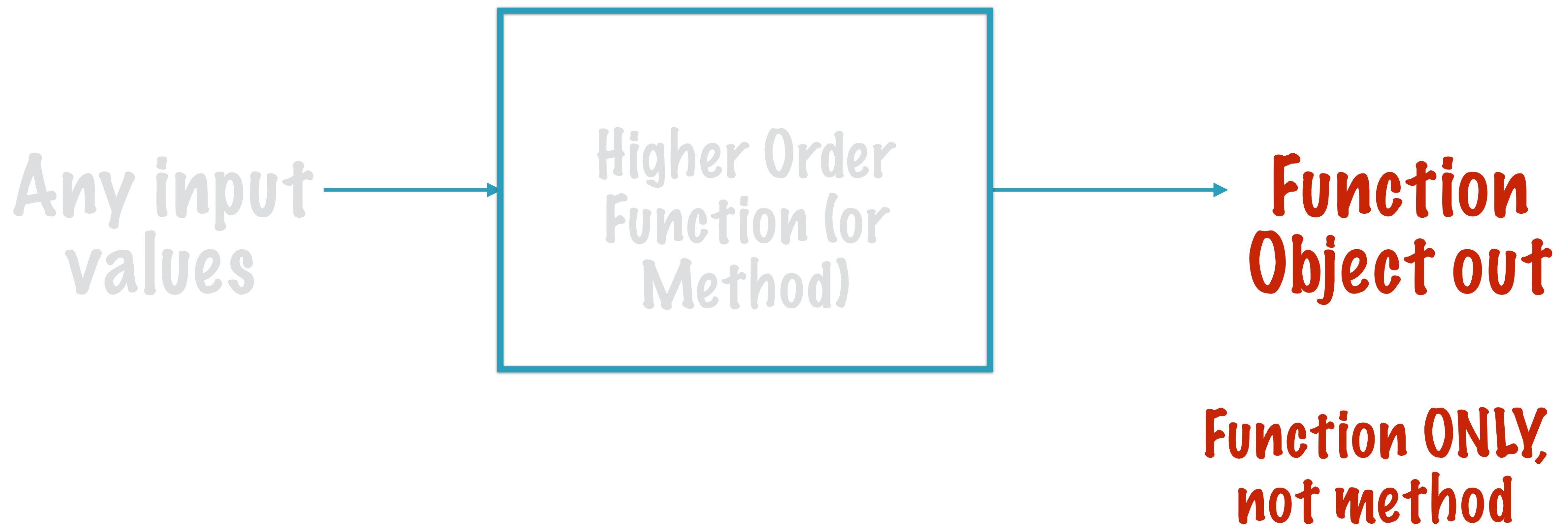


Function OR
method

A “higher order” function



A “higher order” function



Functions can be passed into functions

A “higher order” function is one that

- Either takes in a function as one of its parameters
- Returns a function as its return value

A “higher order” function
takes in a function as
one of its parameters

Let’s say we have a simple string
comparator function

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

A “higher order” function

takes in a function as one of its parameters

Let's say we have a simple string comparator function

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

```
scala> compareStrings("abc","xyz")
```

```
res9: Int = 1
```

```
scala> compareStrings("xyz","abc")
```

```
res10: Int = -1
```

```
scala> compareStrings("abc","abc")
```

```
res11: Int = 0
```

Applies to functions and methods

A “higher order” function

takes in a function as one of its parameters

Let's say we have a simple string comparator function

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

```
scala> compareStrings("abc","xyz")  
res9: Int = 1
```

```
scala> compareStrings("xyz","abc")  
res10: Int = -1
```

```
scala> compareStrings("abc","abc")  
res11: Int = 0
```

Applies to functions and methods

A “higher order” function

takes in a function as one of its parameters

Let's say we have a simple string comparator function

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

```
scala> compareStrings("abc","xyz")  
res9: Int = 1
```

```
scala> compareStrings("xyz","abc")  
res10: Int = -1
```

```
scala> compareStrings("abc","abc")  
res11: Int = 0
```

Applies to functions and methods

A “higher order” function

takes in a function as one of its parameters

Let's say we have a simple string comparator function

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

```
scala> compareStrings("abc","xyz")  
res9: Int = 1
```

```
scala> compareStrings("xyz","abc")  
res10: Int = -1
```

```
scala> compareStrings("abc","abc")  
res11: Int = 0
```

Applies to functions and methods

A “higher order” function

takes in a function as one of its parameters

Let's say we have a simple string comparator function

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

```
scala> compareStrings("abc","xyz")  
res9: Int = 1
```

```
scala> compareStrings("xyz","abc")  
res10: Int = -1
```

```
scala> compareStrings("abc","abc")  
res11: Int = 0
```

Applies to functions and methods

A “higher order” function

takes in a function as one of its parameters

Let's say we have a simple string comparator function

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

```
scala> compareStrings("abc","xyz")  
res9: Int = 1
```

```
scala> compareStrings("xyz","abc")  
res10: Int = -1
```

```
scala> compareStrings("abc","abc")  
res11: Int = 0
```

Applies to functions and methods

A “higher order” function
takes in a function as
one of its parameters

Let's say we have a simple string
comparator function

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

A “higher order” function

takes in a function as one of its parameters

Let's say we have a simple string comparator function

Now, we have another function that takes in this string comparator function

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

A “higher order” function

takes in a function as one of its parameters

Let's say we have a simple string comparator function

Now, we have another function that takes in this string comparator function

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

A “higher order” function

takes in a function as one of its parameters

Let's say we have a simple string comparator function

This is a parameter of our function - and a function itself

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

A “higher order” function

takes in a function as one of its parameters

Let's say we have a simple string comparator function

Notice the type - it takes in **(String, String)**, and returns an **Int**

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

A “higher order” function

takes in a function as one of its parameters

Let's say we have a simple string comparator function

Inside the body of our function, we call this function as usual!

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

Functions can be passed into functions

A “higher order” function is one that

- Either takes in a function as one of its parameters
- Returns a function as its return value

A “higher order” function

Returns a function as its return value

Let's say we now have 2 simple string comparator functions

```
def compareStrings(s1:String, s2:String):Int = {  
    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}  
}
```

Applies to functions and methods

A “higher order” function

Returns a function as its return value

Let's say we now have 2 simple string comparator functions

```
def compareStrings(s1:String, s2:String):Int = {  
    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}  
}
```

Applies to functions and methods

A “higher order” function

takes in a function as one of its parameters

Let's say we now have 2 simple string comparator functions

Now, we have another function that returns one or the other of these string comparator functions

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

A “higher order” function

takes in a function as one of its parameters

Let's say we now have 2 simple string comparator functions

The return value of this function is also a function

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

A “higher order” function

takes in a function as one of its parameters

Let's say we now have 2 simple string comparator functions

Within the body of the function, one or the other comparator is returned

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

A “higher order” function

takes in a function as one of its parameters

Let's say we now have 2 simple string comparator functions

Within the body of the function, one or the other comparator is returned

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

A “higher order” function

takes in a function as one of its parameters

Let's say we now have 2 simple string comparator functions

Within the body of the function, one or the other comparator is returned

```
scala> val cmpFn = getComparator(true)
cmpFn: (String, String) => Int = <function2>
```

Now call this function and use the result to compare strings!

A “higher order” function

takes in a function as one of its parameters

Let's say we now have 2 simple string comparator functions

Within the body of the function, one or the other comparator is returned

Now call this function and use the result to compare strings!

```
scala> val cmpFn = getComparator(true)  
cmpFn: (String, String) => Int = <function2>
```

```
scala> cmpFn("abc", "xyz")  
res16: Int = -1
```

```
scala> cmpFn("xyz", "abc")  
res17: Int = 1
```

```
scala> cmpFn("abc", "abc")  
res18: Int = 0
```

Applies to functions and methods

Functions can be passed into functions

A “higher order” function is one that



Either takes in a function as one of its parameters



Returns a function as its return value

Functions can be passed into functions

A “higher-order function” is one that



Either takes in a function as one of its parameters



Returns a function as its return value

Applies to functions and methods

“higher order” function

Why does this matter?

The idea that functions can be treated
exactly like other values is very
important in functional programming

“First class functions”

Example 33

Anonymous Functions (aka Function Literals)

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Anonymous Functions (aka Function Literals)

Here are 2 subtly different ways to create a function

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

#1: Method converted to function

```
compareStringsConverted = compareStrings
```

Anonymous Functions (aka Function Literals)

Here are 2 subtly different ways to create a function

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

#2: Anonymous function assigned to a value

Anonymous Functions (aka Function Literals)

Here are 2 subtly different ways to create a function

#1: Method converted to function

```
def compareStrings(s1: String, s2: String): Int = {  
}  
compareStringsConverted = compareStrings
```

#2: Anonymous function assigned to a value

```
val compareStringsLiteral = (s1: String, s2: String) => {  
} : Int
```

Here are 2 subtly different ways to create a function

#1: Method converted to function

```
def compareStrings(s1: String, s2: String): Int = {  
}  
compareStringsConverted = compareStrings _
```

#2: Anonymous function assigned to a value

```
val compareStringsLiteral = (s1: String, s2: String) => {  
}: Int
```

Method #1 uses **def**, Method #2 uses **val**

Here are 2 subtly different ways to create a function

#1: Method converted to function #2: Anonymous function assigned to a value

Despite all this, the two functions do exactly the same thing

```
[scala] > compareStringsLiteral("abc", "xyz") == compareStrings("abc", "xyz")
res9: Boolean = true
```

```
[scala] > compareStringsLiteral("xyz", "abc") == compareStrings("xyz", "abc")
res10: Boolean = true
```

```
[scala] > compareStringsLiteral("xyz", "xyz") == compareStrings("xyz", "xyz")
res11: Boolean = true
```

Example 34

Placeholder Syntax (Eta Expansion)

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Placeholder Syntax

Scala allows use of the `_` (underscore)
as a catch-all placeholder/wildcard

We have come across this a couple
of times already - recall:

Pattern
matching

Assigning
methods to values

Applies to functions and methods

Placeholder Syntax

Scala allows use of the `_` (underscore)
as a catch-all placeholder/wildcard

We have come across this a couple
of times already - recall:

 **Pattern
matching**

 **Assigning
methods to values**

Converting a method to a function using eta expansion

```
val calcCircleArea = getCircleArea _
```

This is known as eta expansion

Eta-expansion converts an expression of method type to an equivalent expression of function type.

Applies only to methods,
not to functions

Placeholder Syntax

The `_` (underscore) is also used in invoking anonymous functions

(usually through a higher order function)

This is easiest understood via an example

Applies to functions and methods

Placeholder Syntax

Consider a function that takes in a name, and determines whether that name belongs to a VIP

It also takes in a function to test if that name is a high-status one

```
def IsVIP(firstName:String, lastName:String,  
  IsHighStatus:(String, String) => Boolean):Boolean = {  
  IsHighStatus(firstName, lastName)  
}
```

Applies to functions and methods

Consider a function that takes in a name, and determines whether that name belongs to a VIP

```
def IsVIP(firstName:String, lastName:String,  
  IsHighStatus:(String, String) => Boolean):Boolean = {  
  IsHighStatus(firstName, lastName)  
}
```

It also takes in a function to test if that name is a high-status one

Consider a function that takes in a name, and determines whether that name belongs to a VIP

```
def IsVIP(firstName:String, lastName:String,  
  IsHighStatus:(String, String) => Boolean): Boolean = {  
  IsHighStatus(firstName, lastName)  
}
```

It also takes in a function to test if that name is a high-status one

Consider a function that takes in a name, and determines whether that name belongs to a VIP

```
def IsVIP(firstName:String, lastName:String,  
  IsHighStatus:(String, String) => Boolean):Boolean = {  
  IsHighStatus(firstName, lastName)  
}
```

It also takes in a function to test if that name is a high-status one

(Thus, this is a higher order function)

Applies to functions and methods

Consider a function that takes in a name, and determines whether that name belongs to a VIP

```
def IsVIP(firstName:String, lastName:String,  
  IsHighStatus:(String, String) => Boolean): Boolean = {  
  IsHighStatus(firstName, lastName)  
}
```

It also takes in a function to test if that name is a high-status one
(Thus, this is a higher order function)

Applies to functions and methods

Consider a function that takes in a name, and determines whether that name belongs to a VIP

It also takes in a function to test if that name is a high-status one

There are 2 ways to invoke this function

The long way (explicitly defining the function to be passed in)

The short way (using placeholders)

The long way (explicitly defining the function to be passed in)

Step 1: Define a function that does what we need

```
def IsSpecialName(firstName:String, lastName:String):Boolean = {  
    firstName == "Donald" && lastName == "Trump"  
}
```

Step 2: Invoke the IsVIP function with this function passed in as a parameter

```
IsVIP("Donald", "Trump", IsSpecialName)
```

The long way (explicitly defining
the function to be passed in)

Step 1: Define a function that does what we need

```
def IsSpecialName(firstName:String, lastName:String):Boolean = {  
    firstName == "Donald" && lastName == "Trump"  
}
```

Step 2: Invoke the IsVIP function with this
function passed in as a parameter

```
IsVIP("Donald", "Trump", IsSpecialName)
```

Consider a function that takes in a name, and determines whether that name belongs to a VIP

It also takes in a function to test if that name is a high-status one

There are 2 ways to invoke this function

The long way (explicitly defining the function to be passed in)

The short way (using placeholders)

The long way

```
def IsSpecialName(firstName:String, lastName:String):Boolean = {  
    firstName == "Donald" && lastName == "Trump"  
}
```

```
IsVIP("Donald", "Trump", IsSpecialName)
```

The short way (using placeholders)

```
IsVIP("Donald", "Trump", _ == "Donald" && _ ==  
"Trump")
```

Applies to functions and methods

Placeholder Syntax

Scala allows use of the `_` (underscore)
as a catch-all placeholder/wildcard

We have come across this a couple
of times already - recall:

 **Pattern
matching**

 **Assigning
functions to values**

Function Definition

```
def IsVIP(firstName:String, lastName:String,  
  IsHighStatus:(String, String) => Boolean):Boolean = {  
  IsHighStatus(firstName, lastName)  
}
```

Function Invocation

The short way (using placeholders)

```
IsVIP("Donald", "Trump", _ == "Donald" && _ ==  
"Trump")
```

Function Definition

```
def IsVIP(firstName:String, lastName:String,  
IsHighStatus:(String, String) => Boolean):Boolean = {  
    IsHighStatus(firstName, lastName)  
}
```

Function Invocation

The short way (using placeholders)

```
IsVIP("Donald", "Trump", _ == "Donald" && _ == "Trump")
```

Applies to functions and methods

Function Invocation

The short way (using placeholders)

```
IsVIP("Donald", "Trump", ___ == "Donald" && ___ ==  
"Trump")
```



Scala will blindly fill in the values of the parameters,
in the order in which they appear, into the blanks

Function Invocation

The short way (using placeholders)

```
IsVIP("Donald", "Trump", _ == "Donald" && _ ==  
"Trump")
```

Scala will blindly fill in the values of the parameters,
in the order in which they appear, into the blanks

Function Invocation

The short way (using placeholders)

Scala will blindly fill in the values of the parameters, in the order in which they appear, into the blanks

So, placeholders will only work when the parameters are used in order, and exactly once

Placeholder Syntax

Scala allows use of the `_` (underscore)
as a catch-all placeholder/wildcard

We have come across this a couple
of times already - recall:

 **Pattern
matching**

**Function
Invocation**
The short way

 **Assigning
methods to values**

Applies to functions and methods

Placeholder Syntax

Scala allows use of the `_` (underscore)
as a catch-all placeholder/wildcard

 **Pattern
matching**

 **Assigning
methods to values**

 **Function
Invocation**
The short way

Example 35

Partially Applied Functions

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Scala tries really hard to encourage code re-use



Default Parameters



Type Parameters

Currying

Partially Applied Functions

Applies to functions and methods

Partially Applied Functions

Say you have a function that takes in a bunch of parameters

$F(w, x, y, z)$

```
getOptionPrice(double underlyingPrice,  
double strike, double volatility, double interestRate)
```

(Java)

Applies to functions and methods

Partially Applied Functions

Say you have a function that takes in a bunch of parameters

$F(w, x, y, z)$

getOptionPrice(double underlyingPrice,
double strike, double volatility, double interestRate)

(Java)

Applies to functions and methods

Partially Applied Functions

Say you have a function that takes in a bunch of parameters

$F(w, x, y, z)$

`getOptionPrice(double underlyingPrice,
double strike, double volatility, double interestRate)`

(Java)

Applies to functions and methods

Partially Applied Functions

Say you have a function that takes in a bunch of parameters

$F(w, x, y, z)$

getOptionPrice(double underlyingPrice,
double strike, double volatility, double interestRate)

(Java)

Applies to functions and methods

Partially Applied Functions

Say you have a function that takes in a bunch of parameters

$F(w, x, \textcolor{blue}{y}, z)$

`getOptionPrice(double underlyingPrice,
double strike, double volatility, double interestRate)`

(Java)

Applies to functions and methods

Partially Applied Functions

Say you have a function that takes in a bunch of parameters

$F(w, x, y, z)$

`getOptionPrice(double underlyingPrice,
double strike, double volatility, double interestRate)`

(Java)

Applies to functions and methods

Partially Applied Functions

Say you have a function that takes in a bunch of parameters

$F(w, x, y, z)$

```
getOptionPrice(double underlyingPrice,  
double strike, double volatility, double interestRate)
```

(Java)

Applies to functions and methods

Partially Applied Functions

Say you have a function that

Now, say we would like to fix the
values of 3 of these 4 parameters

And convert a 4-argument function
into a 1-argument function

(Java)

Applies to functions and methods

Partially Applied Functions

Say you have a function that takes in a bunch of parameters

$F(w, x, y, z)$

```
getOptionPrice(double underlyingPrice,  
double strike, double volatility, double interestRate)
```

(Java)

Applies to functions and methods

Partially Applied Functions

Say you have a function that takes in a bunch of parameters

$F(x)$

`getOptionPrice(
 double strike)`

(Java)

Applies to functions and methods

Partially Applied Functions

In Java, the only way to do this
(while still reusing code) is..

Function Overloading

getOptionPrice(
double strike)

(Java)

Applies to functions and methods

Function Overloading

```
getOptionPrice(  
    double strike )  
{  
    getOptionPrice(100,  
    strike, 0.35, 0.05)  
  
}
```

We have specified default values
for all the other parameters

Function Overloading

```
getOptionPrice(  
    double strike)  
{  
    getOptionPrice(100,  
    strike, 0.35, 0.05)  
}
```

We have specified default values
for all the other parameters

Function Overloading

```
getOptionPrice(  
    double strike )  
{  
    getOptionPrice(100,  
    strike, 0.35, 0.05)  
  
}
```

We have specified default values
for all the other parameters

Partially Applied Functions

In Java, the only way to do this
(while still reusing code) is..

Function Overloading

getOptionPrice(
double strike)

(Java)

Applies to functions and methods

In Java, the only way to do this
(while still reusing code) is..

Function Overloading

In Scala, there are multiple ways
to do this, including

Partially Applied Functions

Applies to functions and methods

Partially Applied Functions

Say we have a string comparison function that takes in 2 strings and a comparator function

We would like to fix the comparator to a specific function

The resulting function will only take in 2 strings (no comparator function anymore)

The resulting function
is a
Partially Applied Function

Partially Applied Functions

Say we have a string comparison function that takes in 2 strings and a comparator function

We would like to fix the comparator to a specific function

The resulting function will only take in 2 strings (no comparator function anymore)

Partially Applied Functions

Say we have a string comparison function that takes in 2 strings and a comparator function

We would like to fix the comparator to a specific function

The resulting function will only take in 2 strings (no comparator function anymore)

Say we have a string comparison function that takes in 2 strings and a comparator function

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

Say we have a string comparison function that takes in **2 strings** and a comparator function

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

Say we have a string comparison function that takes in 2 strings and a comparator function

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

Applies to functions and methods

Say we have a **string comparison function** that takes in 2 strings and a comparator function

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

Say we have a string comparison function that takes in 2 strings and a comparator function

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

Partially Applied Functions

Say we have a string comparison function that takes in 2 strings and a comparator function

We would like to fix the comparator to a specific function

The resulting function will only take in 2 strings (no comparator function anymore)

Partially Applied Functions

Say we have a string comparison function that takes in 2 strings and a comparator function

We would like to fix the comparator to a specific function

The resulting function will only take in 2 strings (no comparator function anymore)

We would like to fix the comparator to a specific function

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

Partially Applied Functions

Say we have a string comparison function that takes in 2 strings and a comparator function

We would like to fix the comparator to a specific function

The resulting function will only take in 2 strings (no comparator function anymore)

Partially Applied Functions

Say we have a string comparison function that takes in 2 strings and a comparator function

We would like to fix the comparator to a specific function

The resulting function will only take in 2 strings (no comparator function anymore)

Applies to functions and methods

The resulting function will only take in 2 strings (no comparator function any more)

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

This is now a partially applied function

This is now a partially applied function

The resulting function will only take in 2 strings (no comparator function any more)

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

```
defaultCompare: (String, String) => Int = <function2>
```

This is now a partially applied function

The resulting function **will only take in 2 strings** (no comparator function any more)

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

Notice the placeholder syntax, but with the types specified!

This is now a partially applied function

The resulting function will only take in 2 strings (**no comparator function any more**)

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

The value of the comparator function
is fixed

Partially Applied Functions

Say we have a string comparison function that takes in 2 strings and a comparator function

We would like to fix the comparator to a specific function

The resulting function will only take in 2 strings (no comparator function anymore)

Applies to functions and methods

Partially Applied Functions

Say we have a string comparison function that takes in 2 strings and a comparator function

We would like to fix the comparator to a specific function

The resulting function will only take in 2 strings (no comparator function anymore)

Scala tries really hard to encourage code re-use



Default Parameters



Type Parameters

Currying



Partially Applied Functions

Applies to functions and methods

Example 36

Currying

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Let's continue with our string comparator example

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

Logically, the inputs to this function are of two types

1. the strings to be compared
2. the comparator function used to compare them

Let's continue with our string comparator example

Logically, the inputs to this function are of two types

1. the strings to be compared
2. the comparator function used to compare them

This logical distinction is lost in the definition of our function

Let's continue with our string
comparator example

Logically different things are added to the class
LogScala has the idea of **parameter groups**
to collect logically similar groups of parameters

1. the strings to be compared

Here is how we can use parameter groups
to maintain this logical separation

This logical distinction is lost in the
definition of our function

Parameter groups

Here is how we can use parameter groups
to maintain this logical separation

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

Parameter groups

Here is how we can use parameter groups
to maintain this logical separation

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

Two groups of parameters, two sets of
parantheses!

Parameter groups

Here is how we can use parameter groups
to maintain this logical separation

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

Parameter group #2: The strings to be compared

Parameter groups

Here is how we can use parameter groups
to maintain this logical separation

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

Parameter group #1: The comparator
function

Applies to functions and methods

Parameter groups

Here is how we can use parameter groups
to maintain this logical separation

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

Two groups of parameters, two sets of
parantheses!

Parameter groups

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

Two groups of parameters, two sets of parentheses!

```
scala> curriedCompare(compareStrings)("abc", "xyz")  
res26: Int = 1
```

Invoking a method with 2 parameter groups

Applies to functions and methods

Invoking a method with 2 parameter groups

```
scala> curriedCompare(compareStrings)("abc","xyz")  
res26: Int = 1
```

Two groups of parameters, two sets of parentheses!

Parameter groups

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

Two groups of parameters, two sets of parentheses!

```
scala> curriedCompare(compareStrings)("abc", "xyz")  
res26: Int = 1
```

Invoking a method with 2 parameter groups

Applies to functions and methods

Parameter groups

Two groups of parameters, two sets of parentheses!

The idea of writing a function with multiple parameter groups is called **Currying**

$$F(x, y, z) = H(G(x, y), z)$$

Parameter groups

The idea of writing a function with multiple parameter groups is called Currying

$$F(x, y, z) = H(G(x, y), z)$$

The original function F is decomposed into 2 functions, G and H

Parameter groups

The idea of writing a function with multiple parameter groups is called Currying

$$F(x, y, z) = H(G(x, y), z)$$

The original function F is decomposed into 2 functions, G and H

Parameter groups

The idea of writing a function with multiple parameter groups is called Currying

$$F(x, y, z) = H(G(x, y), z)$$

The original function F is decomposed into 2 functions, G and H

Parameter groups

The idea of writing a function with multiple parameter groups is called Currying

$$F(x, y, z) = H(G(x, y), z)$$

The original function F is decomposed into 2 functions, G and H

Parameter groups

The idea of writing a function with multiple parameter groups is called Currying

$$F(x, y, z) = H(G(x, y), z)$$

The original function F is decomposed into 2 functions, G and H

Parameter groups

The idea of writing a function with multiple parameter groups is called Currying

$$F(x, y, z) = H(G(x, y), z)$$

The original function F is decomposed into 2 functions, G and H

Parameter groups

$$F(x, y, z) = H(G(x, y), z)$$

The original function F is decomposed into 2 functions, G and H

What's more - we could specify either group of parameters to get 2 different partially specified functions!

we could specify either group of parameters to
get 2 different partially specified functions!

1. Specify the strings to be compared

```
[scala] > curriedCompare(_:(String, String)=>Int) ("abc", "xyz")
res24: ((String, String) => Int) => Int = <function1>
```

2. Specify the comparator

```
[scala] > curriedCompare(compareStrings)(_:String, _:String)
res25: (String, String) => Int = <function2>
```

we could specify either group of parameters to
get 2 different partially specified functions!

1. Specify the strings to be compared

```
[scala] > curriedCompare(_:(String, String) => Int) ("abc", "xyz")  
res24: ((String, String) => Int) => Int = <function1>
```

function that
takes 1 argument

2. Specify the comparator

```
[scala] > curriedCompare(compareStrings) (_ : String, _ : String)  
res25: (String, String) => Int = <function2>
```

function that
takes 2 arguments
Applies to functions and methods

we could specify either group of parameters to
get 2 different partially specified functions!

1. Specify the strings to be compared

```
[scala> val x = curriedCompare(_:(String, String) => Int) ("abc", "xyz")
x: ((String, String) => Int) => Int = <function1>
```

```
[scala> x(compareStrings)
res27: Int = 1
```

function that
takes 1 argument

2. Specify the comparator

```
[scala> val y = curriedCompare(compareStrings)(_:String, _:String)
y: (String, String) => Int = <function2>
```

```
[scala> y("abc", "xyz")
res28: Int = 1
```

function that
takes 2 arguments

Applies to functions and methods

Parameter groups

The idea of writing a function with multiple parameter groups is called Currying

$$F(x, y, z) = H(G(x, y), z)$$

The original function F is decomposed into 2 functions, G and H

Scala tries really hard to encourage code re-use



Default Parameters



Type Parameters



Currying



Partially Applied Functions

Applies to functions and methods

Example 37

By-Name Parameters

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

By-Name Parameters

Are a slightly strange language construct -

If you specify a function parameter as being a by-name parameter of a certain type:

- You could pass in a value of that type
- Or you could pass in a function call that returns a value of that type

By-Name Parameters

The parameter was first evaluated only when first referenced inside the function - lazy evaluation

By-Name Parameters

The parameter was subsequently re-evaluated each time it was referenced - eager re-evaluation

Let's check out a "normal" function

```
def sayHelloToMrTrump(name: String):String = {  
    println(s"Meet the President!")  
    println(s"$name");  
    name  
}
```

It takes a string as parameter - pass in a string

```
scala> sayHelloToMrTrump("Vitthal")  
Meet the President!  
Vitthal  
res8: String = Vitthal
```

Okey-dokey, nothing strange here!

Ok! now pass in a function that returns a string

```
[scala> sayHelloToMrTrump(voteForMrTrump("Vitthal"))
go D-J-T
Meet the President!
Vitthal
res12: String = Vitthal
```

Okey-dokey, nothing strange here either!

```
def voteForMrTrump(x:String):String = {
  println("go D-J-T")
  x
}           def sayHelloToMrTrump(name: String):String = {
  println(s"Meet the President!")
  println(s"$name");
  name
}
```

Applies to functions and methods

The parameter is evaluated as soon as the function was invoked - **eager evaluation**

The parameter is evaluated as soon as the function was invoked - eager evaluation

```
[scala> sayHelloToMrTrump(voteForMrTrump("Vitthal"))
go D-J-T
Meet the President!
Vitthal
res12: String = Vitthal
```

```
def voteForMrTrump(x:String):String = {
  println("go D-J-T")
  x
}
```

```
def sayHelloToMrTrump(name: String):String = {
  println(s"Meet the President!")
  println(s"$name");
  name
}
```

Applies to functions and methods

The parameter is evaluated as soon as the function was invoked - eager evaluation

```
[scala> sayHelloToMrTrump(voteForMrTrump("Vitthal"))
go D-J-T
Meet the President!
Vitthal
res12: String = Vitthal
```

```
def voteForMrTrump(x:String):String = {
  println("go D-J-T")
  x
}
```

```
def sayHelloToMrTrump(name: String):String = {
  println(s"Meet the President!")
  println(s"$name");
  name
}
```

Applies to functions and methods

The parameter was evaluated as soon as the function was invoked - eager evaluation

Now, change that function so that it has a “By-Name Parameter

```
def sayHelloToMrTrump(name: String):String = {  
    println(s"Meet the President!")  
    println(s"$name");  
    name  
}
```

Now, change that function so that it has a “By-Name Parameter

```
def sayHelloToMrTrump(name:> String):String = {  
    println(s"Meet the President!")  
    println(s"$name");  
    name  
}
```

Now, change that function so that it has a “By-Name Parameter

```
def sayHelloToMrTrump(name :=> String):String = {  
    println(s"Meet the President!")  
    println(s"$name");  
    name  
}
```

Now - if you call this with a function, that function will be evaluated only and each time the parameter is referenced!

Now, change that function so that it has a “By-Name Parameter

```
def sayHelloToMrTrump(name :=> String):String = {  
    println(s"Meet the President!")  
    println(s"$name");  
    name  
}
```

Now - if you call this with a function, that function will be evaluated **only and each time the parameter is referenced!**

```
scala> sayHelloToMrTrump(voteForMrTrump("Vitthal"))
```

```
def voteForMrTrump(x:String):String = {  
    println("go D-J-T")  
    x  
}
```

Applies to functions and methods

Now - if you call this with a function, that function will be evaluated only and each time the parameter is referenced!

```
scala> sayHelloToMrTrump(voteForMrTrump("Vitthal"))
```

```
Meet the President!
```

```
go D-J-T
```

```
Vitthal
```

```
go D-J-T
```

```
res9: String = Vitthal
```

```
def voteForMrTrump(x:String):String = {  
    println("go D-J-T")  
    x  
}
```

```
def sayHelloToMrTrump(name:> String):String = {  
    println(s"Meet the President!")  
    println(s"$name");  
    name  
}
```

Applies to functions and methods

Now - if you call this with a function, that function will be evaluated only and each time the parameter is referenced!

```
scala> sayHelloToMrTrump(voteForMrTrump("Vitthal"))
Meet the President!
go D-J-T
Vitthal
go D-J-T
res9: String = Vitthal
```

```
def voteForMrTrump(x:String):String = {
  println("go D-J-T")
  x
}
```

```
def sayHelloToMrTrump(name:> String):String = {
  println(s"Meet the President!")
  println(s"$name");
  name
}
```

Applies to functions and methods

Now - if you call this with a function, that function will be evaluated only and each time the parameter is referenced!

```
scala> sayHelloToMrTrump(voteForMrTrump("Vitthal"))
Meet the President!
go D-J-T
Vitthal
go D-J-T
res9: String = Vitthal
```

```
def voteForMrTrump(x:String):String = {
  println("go D-J-T")
  x
}
```

```
def sayHelloToMrTrump(name:> String):String = {
  println(s"Meet the President!")
  println(s"$name");
  name
}
```

Applies to functions and methods

The parameter was first evaluated only when first referenced inside the function - lazy evaluation

The parameter was subsequently re-evaluated each time it was referenced - eager re-evaluation

Applies to functions and methods

Now, change that function so that it has a “By-Name Parameter

Now - if you call this with a function, that function will be evaluated each time the parameter is referenced!

Be careful of functions
with side effects!!

Example 38

closures

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

Applies only to methods, not to functions

Applies only to functions, not to methods

Applies to functions and methods

CLOSURES

BY DEFINITION, A FUNCTION OBJECT
RESTS INSIDE A METHOD

SAY WE HAVE A NESTED
FUNCTION OBJECT

CLOSURES

OUTER METHOD

VARIABLES LOCAL TO THE OUTER METHOD

NESTED FUNCTION OBJECT
CAN BE ACCESSED FROM HERE..

Applies to functions and methods

CLOSURES

OUTER METHOD

EVEN AFTER THE OUTER
METHOD CEASES TO EXIST

CAN BE ACCESSED FROM HERE..

Applies to functions and methods

CLOSURES

OUTER METHOD

VARIABLES LOCAL TO THE OUTER METHOD

NESTED FUNCTION OBJECT
CAN BE ACCESSED FROM HERE..
EVEN AFTER THE OUTER METHOD
CEASES TO EXIST

Applies to functions and methods

CLOSURES

OUTER METHOD

VARIABLES LOCAL TO THE OUTER METHOD

NESTED FUNCTION OBJECT
CAN BE ACCESSED FROM HERE..

EVEN AFTER THE OUTER METHOD
CEASES TO EXIST

Applies to functions and methods

CLOSURE = **NESTED FUNCTION
OBJECT**
+
**VARIABLES LOCAL TO THE
OUTER METHOD**

Applies to functions and methods

CLOSURE = **NESTED FUNCTION**

+

**VARIABLES LOCAL TO THE
OUTER SCOPE**

**SCOPE IS THE MORE TECHNICAL, AND
GENERAL TERM FOR THE OUTER METHOD**

Applies to functions and methods

CLOSURE = NESTED FUNCTION

+

VARIABLES LOCAL TO THE
OUTER SCOPE

SCOPE IS THE MORE TECHNICAL, AND
GENERAL TERM FOR THE OUTER FUNCTION

Applies to functions and methods

CLOSURE = NESTED FUNCTION +
:VARIABLES LOCAL TO THE
OUTER SCOPE
"REFERENCING ENVIRONMENT"

REMEMBER THIS EQUATION, AND WE
WILL BE JUST FINE!

CLOSURE = NESTED FUNCTION +
:VARIABLES LOCAL TO THE
OUTER SCOPE
"REFERENCING ENVIRONMENT"

THIS SEEMS LIKE MAGIC - AND IT IS.

CLOSURE = NESTED FUNCTION +
VARIABLES LOCAL TO THE
OUTER SCOPE
"REFERENCING ENVIRONMENT"

MAGIC TRICK #1: THE NESTED FUNCTION CAN
ACCESS THE REFERENCING ENVIRONMENT - EVEN
THOUGH THOSE VARIABLES ARE OUTSIDE THE SCOPE

CLOSURE = NESTED FUNCTION +
VARIABLES LOCAL TO THE
OUTER SCOPE
"REFERENCING ENVIRONMENT"

MAGIC TRICK #1: THE NESTED FUNCTION CAN
ACCESS THE REFERENCING ENVIRONMENT - EVEN
THOUGH THOSE VARIABLES ARE OUTSIDE THE SCOPE

CLOSURE = NESTED FUNCTION +
VARIABLES LOCAL TO THE
OUTER SCOPE
"REFERENCING ENVIRONMENT"

MAGIC TRICK #1: THE NESTED FUNCTION CAN
ACCESS THE REFERENCING ENVIRONMENT - EVEN
THOUGH THOSE VARIABLES ARE OUTSIDE THE SCOPE

CLOSURE = NESTED FUNCTION +
VARIABLES LOCAL TO THE
OUTER SCOPE
"REFERENCING ENVIRONMENT"

MAGIC TRICK #2: THE NESTED FUNCTION CARRIES
AROUND THAT REFERENCING ENVIRONMENT EVEN
AFTER THE SCOPE HAS "GONE AWAY"!

CLOSURE = NESTED FUNCTION +
VARIABLES LOCAL TO THE
OUTER SCOPE
"REFERENCING ENVIRONMENT"

MAGIC TRICK #2: THE NESTED FUNCTION CARRIES
AROUND THAT REFERENCING ENVIRONMENT EVEN
AFTER THE SCOPE HAS "GONE AWAY"!

CLOSURE = NESTED FUNCTION +
VARIABLES LOCAL TO THE
OUTER SCOPE
"REFERENCING ENVIRONMENT"

MAGIC TRICK #2: THE NESTED FUNCTION CARRIES
AROUND THAT REFERENCING ENVIRONMENT EVEN
AFTER THE SCOPE HAS "GONE AWAY"!

CLOSURE = NESTED FUNCTION +
VARIABLES LOCAL TO THE
OUTER SCOPE
"REFERENCING ENVIRONMENT"

MAGIC TRICK #2: THE NESTED FUNCTION CARRIES
AROUND THAT REFERENCING ENVIRONMENT EVEN
AFTER THE SCOPE HAS "GONE AWAY"!

LET'S APPLY OUR EQUATION TO AN EXAMPLE

CLOSURE = NESTED FUNCTION +
: VARIABLES LOCAL TO THE
OUTER SCOPE
.....
"REFERENCING ENVIRONMENT"

FUNCTION OBJECTS IN SCALA CAN BE TREATED JUST LIKE NUMBERS OR STRINGS

1

YOU CAN STORE A
FUNCTION IN A VARIABLE

2

YOU CAN HAVE A METHOD
RETURN A FUNCTION

3

YOU CAN HAVE A METHOD
TAKE IN A FUNCTION AS
AN ARGUMENT

THESE 3 PROPERTIES COLLECTIVELY
ARE CALLED "FIRST CLASS FUNCTIONS"

Applies to functions and methods

SUCH A METHOD IS CALLED “A HIGHER ORDER METHOD”

YOU CAN STORE A
FUNCTION IN A VARIABLE

YOU CAN HAVE A METHOD
RETURN A FUNCTION

YOU CAN HAVE A METHOD
TAKE IN A FUNCTION AS
AN ARGUMENT

THESE 3 PROPERTIES COLLECTIVELY
ARE CALLED “FIRST CLASS FUNCTIONS”

Applies to functions and methods

YOU CAN HAVE A METHOD
RETURN A FUNCTION

SAY YOU WANTED TO PRINT A
GREETING TO A USER

“HELLO SWETHA”

“NAMASTE JANANI”

“BONJOUR VITTHAL”

Applies to functions and methods

YOU CAN HAVE A METHOD
RETURN A FUNCTION

YOU WANT THE GREETING TO CHANGE
BASED ON THE LANGUAGE OF THE USER

“HELLO SWETHA”

ENGLISH

“NAMASTE JANANI”

HINDI

“BONJOUR VITTHAL”

FRENCH

YOU CAN HAVE A METHOD RETURN A FUNCTION

```
def greeting(lang: String) = {  
    lang match {  
        case "English" => (x: String) => println("Hello "+x)  
        case "Hindi" => (x: String) => println("Namaste "+x)  
        case "French" => (x: String) => println("Bonjour "+x)  
        case "Spanish" => (x: String) => println("Hola "+x)  
    }  
}
```

HAVE A METHOD RETURN A FUNCTION BASED ON THE LANGUAGE!

Applies to functions and methods

YOU CAN HAVE A METHOD RETURN A FUNCTION

MATCH IS SIMILAR TO SWITCH IN JAVA

```
def greeting(lang: String) = {  
    lang match {  
        case "English" => (x: String) => println("Hello "+x)  
        case "Hindi" => (x: String) => println("Namaste "+x)  
        case "French" => (x: String) => println("Bonjour "+x)  
        case "Spanish" => (x: String) => println("Hola "+x)  
    }  
}
```



Applies to functions and methods

YOU CAN HAVE A METHOD RETURN A FUNCTION

THIS IS A FUNCTION OBJECT

```
def greeting(lang: String) = {  
  
    lang match {  
  
        case "English" => (x: String) => println("Hello "+x)  
        case "Hindi" => (x: String) => println("Namaste "+x)  
        case "French" => (x: String) => println("Bonjour "+x)  
        case "Spanish" => (x: String) => println("Hola "+x)  
    }  
}
```

YOU CAN HAVE A METHOD RETURN A FUNCTION

IT TAKES A STRING AND PRINTS A
GREETING TO SCREEN

```
def greeting(lang: String) {  
    lang match {  
        case "English" => (x: String) => println("Hello "+x)  
        case "Hindi" => (x: String) => println("Namaste "+x)  
        case "French" => (x: String) => println("Bonjour "+x)  
        case "Spanish" => (x: String) => println("Hola "+x)  
    }  
}
```

YOU CAN HAVE A METHOD RETURN A FUNCTION

FOR EACH LANGUAGE, RETURN A FUNCTION
THAT PRINTS THE APPROPRIATE GREETING

```
def greeting(lang: String) = {  
  
    lang match {  
  
        case "English" => (x: String) => println("Hello "+x)  
  
        case "Hindi" => (x: String) => println("Namaste "+x)  
  
        case "French" => (x: String) => println("Bonjour "+x)  
  
        case "Spanish" => (x: String) => println("Hola "+x)  
  
    }  
}
```



Applies to functions and methods

YOU CAN HAVE A METHOD RETURN A FUNCTION

```
def greeting(lang: String) = {  
    lang match {  
        case "English" => (x: String) => println("Hello "+x)  
        case "Hindi" => (x: String) => println("Namaste "+x)  
        case "French" => (x: String) => println("Bonjour "+x)  
        case "Spanish" => (x: String) => println("Hola "+x)  
  
    }  
  
    def main (args: Array[String]) {  
        val greetEnglish = greeting("English")  
        greetEnglish("Swetha")  
  
        val greetSpanish = greeting("Spanish")  
        greetSpanish("Janani")  
    }  
}
```

YOU CAN GET THE
APPROPRIATE
FUNCTION BASED ON
THE LANGUAGE OF
THE USER

Applies to functions and methods

YOU CAN HAVE A METHOD RETURN A FUNCTION

```
def greeting(lang: String) = {  
    lang match {  
        case "English" => (x: String) => println("Hello "+x)  
        case "Hindi" => (x: String) => println("Namaste "+x)  
        case "French" => (x: String) => println("Bonjour "+x)  
        case "Spanish" => (x: String) => println("Hola "+x)  
  
    def main (args: Array[String]) {  
        val greetEnglish = greeting("English")  
        greetEnglish("Swetha")  
  
        val greetSpanish = greeting("Spanish")  
        greetSpanish("Janani")  
    }  
}
```

GREETENGLISH IS A
FUNCTION OBJECT
FOR GREETING
ENGLISH SPEAKING
USERS

Applies to functions and methods

YOU CAN HAVE A METHOD RETURN A FUNCTION

```
def greeting(lang: String) = {  
    lang match {  
        case "English" => (x: String) => println("Hello "+x)  
        case "Hindi" => (x: String) => println("Namaste "+x)  
        case "French" => (x: String) => println("Bonjour "+x)  
        case "Spanish" => (x: String) => println("Hola "+x)  
  
    def main (args: Array[String]) {  
        val greetEnglish = greeting("English")  
        greetEnglish("Swetha")  
  
        val greetSpanish = greeting("Spanish")  
        greetSpanish("Janani")  
    }  
}
```

**CALL THE GREETING
METHOD AND STORE
THE VALUE RETURNED
IN GREETENGLISH**

Applies to functions and methods

YOU CAN HAVE A METHOD RETURN A FUNCTION

```
def greeting(lang: String) = {  
    lang match {  
        case "English" => (x: String) => println("Hello "+x)  
        case "Hindi" => (x: String) => println("Namaste "+x)  
        case "French" => (x: String) => println("Bonjour "+x)  
        case "Spanish" => (x: String) => println("Hola "+x)  
  
    def main (args: Array[String]) {  
        val greetEnglish = greeting("English")  
        greetEnglish("Swetha")  
  
        val greetSpanish = greeting("Spanish")  
        greetSpanish("Janani")  
    }  
}
```

USE IT TO PRINT THE APPROPRIATE GREETING FOR ALL ENGLISH SPEAKING USERS

Applies to functions and methods

YOU CAN HAVE A METHOD RETURN A FUNCTION

```
def greeting(lang: String) = {  
    lang match {  
        case "English" => (x: String) => println("Hello "+x)  
        case "Hindi" => (x: String) => println("Namaste "+x)  
        case "French" => (x: String) => println("Bonjour "+x)  
        case "Spanish" => (x: String) => println("Hola "+x)  
    }  
  
def main (args: Array[String]) {  
    val greetEnglish = greeting("English")  
    greetEnglish("Swetha")  
  
    val greetSpanish = greeting("Spanish")  
    greetSpanish("Janani")  
}
```

IF THE USER IS
SPANISH - GET A
DIFFERENT FUNCTION
AND USE IT

Applies to functions and methods

OK, LET'S TWEAK OUR GREETING FUNCTION A BIT

CLOSURE = NESTED FUNCTION +
VARIABLES LOCAL TO THE
OUTER SCOPE
.....
"REFERENCING ENVIRONMENT"

LET'S TIE BACK TO CLOSURES, BY
CREATING A LOCAL VARIABLE

OK, LET'S TWEAK OUR GREETING FUNCTION A BIT

```
def greeting(lang: String)= {  
    val currDate = Calendar.getInstance().getTime().toString  
    lang match {  
        case "English" => (x: String) => println("Hello "+x +". It is "+currDate)  
        case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)  
        case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)  
        case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)  
    }  
}
```

CREATE A LOCAL VARIABLE
INSIDE THIS SCOPE

CLOSURE = NESTED FUNCTION +
: VÄRIÄBLES LOCAL TO THE :
: OUTER SCOPE :
: :
: :

"REF" Applies to functions and methods

OK, LET'S TWEAK OUR GREETING FUNCTION A BIT

```
def main (args: Array[String]){

    val greetEnglish = greeting("English")
    greetEnglish("Swetha")

    val greetSpanish = greeting("Spanish")
    greetSpanish("Janani")
}

def greeting(lang: String)= {

    val currDate = Calendar.getInstance().getTime().toString

    lang match {

        case "English" => (x: String) => println("Hello "+x +". It is "+currDate)
        case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)
        case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)
        case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)
    }
}
```

WE HAVE A METHOD THAT
RETURNS A FUNCTION

CLOSURE = NESTED FUNCTION +
: VARIABLES LOCAL TO THE :
: OUTER SCOPE :
: :
: :

"REF" Applies to functions and methods

```
val greetEnglish = greeting("English")
greetEnglish("Swetha")

val greetSpanish = greeting("Spanish")
greetSpanish("Janani")
```

A METHOD THAT RETURNS A FUNCTION

```
def greeting(lang: String)= {
    val currDate = Calendar.getInstance().getTime().toString

    lang match {
        case "English" => (x: String) => println("Hello "+x +". It is "+currDate)
        case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)
        case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)
        case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)
    }
}
```

CLOSURE = NESTED FUNCTION +
: VÄRIÄBLES LOCAL TO THE :
: OUTER SCOPE :
.....

"REF" Applies to functions and methods

```
val greetEnglish = greeting("English")
greetEnglish("Swetha")

val greetSpanish = greeting("Spanish")
greetSpanish("Janani")

}
```

```
def greeting(lang: String)= {

    val currDate = Calendar.getInstance().getTime().toString

    lang match {

        case "English" => (x: String) => println("Hello "+x +". It is "+currDate)
        case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)
        case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)
        case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)
    }
}
```

CLOSURE = NESTED FUNCTION +
: VARIABLES LOCAL TO THE
: OUTER SCOPE

"REF" Applies to functions and methods

```
val greetEnglish = greeting("English")
greetEnglish("Swetha")

val greetSpanish = greeting("Spanish")
greetSpanish("Janani")

}
```

```
def greeting(lang: String)= {

    val currDate = Calendar.getInstance().getTime().toString

    lang match {

        case "English" => (x: String) => println("Hello "+x +". It is "+currDate)
        case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)
        case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)
        case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)
    }
}
```

CLOSURE = NESTED FUNCTION +
VARIABLES LOCAL TO THE
OUTER SCOPE

"REF" Applies to functions and methods

```
val greetEnglish = greeting("English")
greetEnglish("Swetha")

val greetSpanish = greeting("Spanish")
greetSpanish("Janani")

}
```

```
def greeting(lang: String)= {

    val currDate = Calendar.getInstance().getTime().toString

    lang match {

        case "English" => (x: String) => println("Hello "+x +". It is "+currDate)

        case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)

        case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)

        case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)

    }
}
```

CLOSURE = NESTED FUNCTION +
VARIABLES LOCAL TO THE
OUTER SCOPE

REF Applies to functions and methods

```

def main (args: Array[String]){

  val greetEnglish = greeting("English")
  greetEnglish("Swetha")

  val greetSpanish = greeting("Spanish")
  greetSpanish("Janani")

}

def greeting(lang: String)= {
  val currDate = Calendar.getInstance().getTime().toString
  lang match {
    case "English" => (x: String) => println("Hello "+x +". It is "+currDate)
    case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)
    case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)
    case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)
  }
}

```

**WHEN THE GREETING METHOD
IS CALLED IT RETURNS A
FUNCTION OBJECT**

WHICH IS A CLOSURE

Applies to functions and methods

```
def main (args: Array[String]){

  val greetEnglish = greeting("English")
  greetEnglish("Swetha")

  val greetSpanish = greeting("Spanish")
  greetSpanish("Janani")

}

def greeting(lang: String)= {
  val currDate = Calendar.getInstance().getTime().toString
  lang match {
    case "English" => (x: String) => println("Hello "+x +". It is "+currDate)
    case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)
    case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)
    case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)
  }
}
```

METHOD IS CALLED HERE,
WITHIN THE METHOD THE
CURRDATE VARIABLE IS CREATED

A CLOSURE IS RETURNED AND
STORED IN GREETENGLISH

```
def main (args: Array[String]){

    val greetEnglish = greeting("English")
    greetEnglish("Swetha")

    val greetSpanish = greeting("Spanish")
    greetSpanish("Janani")
}

def greeting(lang: String)= {
    val currDate = Calendar.getInstance().getTime().toString

    lang match {

        case "English" => (x: String) => println("Hello "+x +". It is "+currDate)
        case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)
        case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)
        case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)
    }
}
```

HERE THE GREETING METHOD

HAS CEASED TO EXIST

BUT, THE CLOSURE CAN STILL
USE THE CURRDATE VARIABLE

CLOSURE = NESTED FUNCTION +
: VÄRIÄBLES LOCAL TO THE :
: ... OUTER SCOPE ... :
"REFERENCING"

WHY ARE CLOSURES SO IMPORTANT?

ITS BECAUSE CLOSURES CARRY
THE ENVIRONMENT AROUND
THEM WHEREVER THEY GO

Applies to functions and methods

CLOSURE = NESTED FUNCTION +
: VÄRIÄBLES LOCAL TO THE :
: ... OUTER SCOPE ... :
"REFERENCING"

WHY ARE CLOSURES SO IMPORTANT?

IN A DISTRIBUTED ENVIRONMENT,
CLOSURES MAKE IT EASIER TO SHIP
CODE TO DIFFERENT NODES IN A CLUSTER