Contents

1. Defining and Calling Functions
2. Higher Order Functions
3. Function Literals
4. Closures
5. Partially Applied Functions and Currying
6. By-Name Parameters
1. Defining and Calling Functions

2. Higher Order Functions

3. Function Literals

4. Closures

5. Partially Applied Functions and Currying

6. By-Name Parameters
Functions and Methods

- Functions that are members of some object (class instance or singleton) are called methods.
- `private` keyword makes method visible only within this object.

```scala
import scala.io.Source

object LongLines {

  def processFile(filename: String, width: Int) {
    val source = Source.fromFile(filename)
    for (l <- source.getLines)
      procLine(filename, width, l)
  }

  private def procLine(filename: String, l: String) {
    if (l.length > 79)
      println(filename +": " + l.trim)
  }

}
```
Local Functions

- Functions can be defined within other functions.
- Functions are only visible in surrounding scope.
- Inner function can access namespace of surrounding function.

```scala
import scala.io.Source

object LongLines {
  def processFile(filename: String, width: Int) {
    def procLine(l: String) {
      if (l.length > 79) println(filename + "": " + l.trim)
    }

    val source = Source.fromFile(filename)
    for (l <- source.getLines)
      procLine(filename, width, l)
  }
}
```
Repeated Parameters

- The last parameter of a function may be repeated.
- Repeat parameter is denoted with * following the type.

```scala
def echo(args: String*) = 
  for (arg <- args) println(arg)
echo: (args: String*) Unit

def echo2(args: String*) = 
  print(args.slice(1, args.size))
echo2: (args: String*) Unit
```

Can operate on args like on an Array

```scala
def echo2(args: String*) = 
  print(args.slice(1, args.size))
```

```scala
def echo2("Scala","is","awesome")
WrappedArray(is, awesome)
```
Named Parameters

Can refer to parameter by name.

```scala
scala> def speed(distance: Double, time: Double): Double =
        distance/time
speed: (distance: Double, time: Double)Double

scala> speed(100, 20)
res0: Double = 5.0

scala> speed(time = 20, distance = 100)
res1: Double = 5.0
```
Default Parameters

- Can assign a default value to parameters in the function definition.
- Can omit default parameters when calling the function.

```scala
def printTime(out: java.io.PrintStream = Console.out) = 
  out.println("time = " + System.currentTimeMillis())

scala> printTime()
time = 1415144428335

scala> printTime(Console.err)
time = 1415144437279
```
1. Defining and Calling Functions

2. Higher Order Functions

3. Function Literals

4. Closures

5. Partially Applied Functions and Currying

6. By-Name Parameters
First Class Functions

- Functions are values
  - Can be passed as arguments to higher order functions.
  - Can be returned by other functions.
  - Can be assigned to a variable.
Higher Order Functions

- Higher order functions are functions that take function values as parameters.
- Provide a powerful abstraction over functions.

```scala
def my_map(lst : List[Int], fun : Int => Int) : List[Int] =
  for (l <- lst) yield fun(l)

val numbers = List(2,3,4,5)

def addone(n : Int) = n + 1

scala> my_map(numbers, addone)
res0 : List[Int] = List(3, 4, 5, 6)
```
Lists have methods that accept functions as arguments (sorting, filtering, mapping...) 
foreach, filter, map, fold

```scala
def addone(x : Int) : Int = x + 1
val numbers = List(-11,-10,5,0,5,10)

scala> numbers.map(addone)
res0: List[Int] = List(-10, -9, 6, 1, 6, 11)
```
1. Defining and Calling Functions

2. Higher Order Functions

3. Function Literals

4. Closures

5. Partially Applied Functions and Currying

6. By-Name Parameters
Function Literals

- use => to define a small function in place.
- called ‘lambdas’ in other languages.

```scala
scala> (x: Int) => x + 1
res0: Int => Int = <function1>

scala> val numbers = List(-11, -10, 5, 0, 5, 10)
numbers: List[Int] = List(-11, -10, 5, 0, 5, 10)

scala> numbers.map((x: Int) => x + 1)
res1: List[Int] = List(-10, -9, 6, 1, 6, 11)
```
Functions objects are values

Functions can be stored in a variable.

```scala
scala> val numbers = List(-11,-10,5,0,5,10)
names: List[Int] = List(-11, -10, 5, 0, 5, 10)

scala> var addfun = (x: Int) => x + 1
addfun: Int => Int = <function1>

scala> numbers.map(addfun)
res1: List[Int] = List(-10, -9, 6, 1, 6, 11)

scala> var addfun = (x: Int) => x * 2
addfun: Int => Int = <function1>

scala> numbers.map(addfun)
res1: List[Int] = List(-22, -20, 10, 0, 10, 20)
```
Short forms of function literals

- Target typing: If a function literal is used immediately the compiler can do type inference.

```scala
scala> numbers.map(x => x+1)
res1: List[Int] = List(-22, -20, 10, 0, 10, 20)

scala> var addfun = x => x+1
<console>:7: error: missing parameter type
  var addfun = x => x+1
```
Can skip parameter names if each parameter is used only once.

```scala
classroom> numbers.map(x => x+1)
res1: List[Int] = List(-22, -20, 10, 0, 10, 20)

classroom> numbers.map(_ + 1)
res0: List[Int] = List(-10, -9, 6, 1, 6, 11)

classroom> val fun = _ + _  // can’t do type inference here
<console>:7: error: missing parameter type for expanded
   function ((x$1, x$2) => x$1.$plus(x$2))
   val fun = _ + _

classroom> val fun = (_ : Int) + (_ : Int)
fun: (Int, Int) => Int = <function2>
```
Partially Applied Functions

Only function literals can be assigned to a variable directly.

```scala
cfg> val square = (x: Int) => x*x
foo: Int => Int = <function1>

cfg> val squareToo = square
x: Int => Int = <function1>

cfg> def squareDef (x: Int) = x*x
squareDef (x: Int) Int

cfg> val squareDefToo = squareDef
<console>:13: error: missing arguments for method squareDef; follow this method with ‘_’ if you want to treat it as a partially applied function
val squareDefToo = squareDef

cfg> val squareDefToo = squareDef _
squareDefToo: Int => Int = <function1>
```
Example: Gradient Decent

- Goal find local minimum of a function $f$.
- Move closer to the minimum by taking small steps according to the gradient of $f$ at each point.
- Important in machine learning: find model parameters that minimize some loss/error function.
- Evaluate first derivative $f'$ of $f$ to get gradient.
Example: Gradient Decent (Toy example in 1D)

Try to find the minimum of \( \text{fun}(x) = x^4 - 3x^3 + 2 \).
Derivative is \( \text{fun}'(x) = 4x^3 + 9x^2 \).

```scala
def funDeriv(x : Double) = math.pow(x, 4) + 9 * math.pow(x, 2)
def gradientDescent(f : Double => Double, init : Double, rate : Double) = {
  def rec(next : Double, old : Double) : Double = {
    if (math.abs(next - old) <= 0.00001) next
    else rec(next - rate * f(next), next)
  }
  gradientDescentInner(init, 0)
}
scala> gradientDescent(funDeriv, 1, 0.001)
res0: Double = 0.03331591669117074
```
1. Defining and Calling Functions
2. Higher Order Functions
3. Function Literals
4. Closures
5. Partially Applied Functions and Currying
6. By-Name Parameters
Closures

- **Bound variables** are defined as part of a function itself (vars, vals, parameters).
- **Free variables** are not defined within the function.
- Functions that contains free variables defined in the surrounding context are called *closures*.

```
scala> var more = 1
more: Int = 1

scala> def addMode(x : Int) = x + more
addMode: (x: Int)Int

scala> addMore(1)
res3: Int = 2
```
Closures capture variables not values

```
scala> var more = 1
more: Int = 1

scala> def addMode(x: Int) = x + more
addMode: (x: Int)Int

scala> addMore(1)
res4: Int = 1000
```
Changing captured vars

```scala
scala> var sum = 0;
sum: Int = 0

scala> def addToSum(x: Int) { sum += x}
addToSum: (x: Int) Unit

scala> addToSum(42)

scala> addToSum(23)

scala> sum
res7: Int = 65
```
Creating and Returning Closures

- Closure references the variable defined when the outer function runs.
- Can return the closure with the argument bound.

```scala
def makeIncreaser(more : Int) = (x: Int) => x + more

scala> val inc9999 = makeIncreaser(9999)
inc9999: Int => Int = <function1>

scala> inc9999(1)
res0: Int = 1000
```
1. Defining and Calling Functions
2. Higher Order Functions
3. Function Literals
4. Closures
5. Partially Applied Functions and Currying
6. By-Name Parameters
Only function literals can be assigned to a variable

```scala
scala> val square = (x: Int) => x*x
foo: Int => Int = <function1>

scala> val squareToo = square
x: Int => Int = <function1>

scala> def squareDef (x: Int) = x*x
squareDef (x: Int) Int

scala> val squareDefToo = squareDef  // Tries to evaluate squareDef
<console>:13: error: missing arguments for method squareDef; follow this method with '_’ if you want to treat it as a partially

scala> val squareDefToo = squareDef _
squareDefToo: Int => Int = <function1>
```
Partially applied functions

```
scala> val squareDefToo = squareDef _
squareDefToo: Int => Int = function1>

scala> val squareDefToo = (x: Int) squareDef(x) // equivalent
squareDefToo: Int => Int = function1>
```

- trailing `_` can be a placeholder for the entire argument list.

```
scala> def sumThree (a: Int, b: Int, c: Int) = a+b+c
sumThree: (a: Int, b: Int, c: Int) Int

scala> val sumThreeToo = sumThree _
sumThreeToo: (Int, Int, Int) => Int = function3>
```
Currying (a.k.a Schönfinkelning): Define a function that is applied to multiple parameter lists (one at a time).

```scala
scala> def curriedSum(x: Int)(y: Int) = x + y
curriedSum: (x: Int)(y: Int) Int

scala> curriedSum(1)(2)
res0: Int = 3

scala> curriedSum(1)
<console>:14: error: missing arguments for method curriedSum; follow this method with ‘_’ if you want to treat it as a partially applied function
```
Currying and Partially Applied Functions

```scala
class Sum2 {
  def curriedSum2(x: Int) = (y: Int) => x + y
  curriedSum2: (x: Int) => Int = curriedSum2(int)
}

class Sum {
  def curriedSum(x: Int)(y: Int) = x + y
  curriedSum: (x: Int)(y: Int) => Int = curriedSum(int, int)
}
```

```scala
class Sum2 {
  curriedSum2(1) // res1: Int => Int = <function1>
}

class Sum {
  curriedSum(1) // res2: Int => Int = <function1>
}
```
Higher order functions allow you to write new control structures

```scala
scala> def twice(op: Double => Double)(x: Double) = op(op(x))
twice: (op: Double => Double)(x: Double)Double

scala> twice(_ + 2)(3)
res16: Double = 7.0

scala> scala twice {
    x => x + 2
} (3)
res16: Double = 7.0
```
def curry2[A,B,C](f:(A,B) => C) : A => B => C =
  { (a: A) =>
    { (b: B) => f(a,b) }
  }

scala> val add = (x: Int , y: Int ) => x + y
add: (Int , Int ) => Int = <function2>

scala> val a = curry2 ( add )
a: Int => ( Int => Int ) = <function1>

scala> a(1)(2)
res1: Int = 3

This is actually already implemented as a method on function objects.

scala> val a = add . curried
a: Int => ( Int => Int ) = <function1>

scala> a(1)(2)
res2 : Int = 3
Currying functions

```scala
def curry2[A,B,C](f:(A,B) => C) : A => B => C = 
  { (a: A) => 
    { (b: B) => f(a,b) } 
  }

scala> val add = (x: Int, y: Int) => x + y
add: (Int, Int) => Int = <function2>

scala> val a = curry2(add)
a: Int => (Int => Int) = <function1>

scala> a(1)(2)
res1: Int = 3

This is actually already implemented as a method on function objects.

scala> val a = add.curried
a: Int => (Int => Int) = <function1>

scala> a(1)(2)
res2: Int = 3
```
1. Defining and Calling Functions

2. Higher Order Functions

3. Function Literals

4. Closures

5. Partially Applied Functions and Currying

6. By-Name Parameters
By-Name Parameters

- By default Scala is call-by-value:
  - Any expression is evaluated before it is passed as a function parameter.
- Can force call-by-name by prefixing parameter types with `=>`.
- Expression passed to parameter is evaluated every time it is used.

```scala
scala> def add(x : Int, y: Int) =
   { println("normal add: "+ x + "+" + y); x+y }
add: (x: Int, y: Int)Int

scala> def lazyAdd(x: =>Int, y: => Int) =
   { println("lazy add"); x+y }
```

```scala
def add(x : Int, y: Int) =
   { println("normal add: "+ x + "+" + y); x+y }
add : (x: Int, y: Int)Int

def lazyAdd(x: =>Int, y: => Int) =
   { println("lazy add"); x+y }
```
By-Name Parameters

- By default Scala is call-by-value:
  - Any expression is evaluated before it is passed as a function parameter.
- Can force call-by-name by prefixing parameter types with `=>`.
- Expression passed to parameter is evaluated every time it is used.

```scala
scala> def add(x : Int, y: Int) =  
    { println("normal add: "+ x + "+" + y); x+y }  
add : (x: Int, y: Int)Int

scala> def lazyAdd(x: =>Int, y: => Int) =  
    { println("lazy add"); x+y }  
lazyAdd : (x: => Int, y: => Int)Int

scala> add(add(1,2), add(2,3))  
normal add: 1+2  
normal add: 2+3  
normal add: 3+5  
res0: Int = 8
```
By-Name Parameters

- By default Scala is call-by-value:
  - Any expression is evaluated before it is passed as a function parameter.
- Can force call-by-name by prefixing parameter types with `=>`.
- Expression passed to parameter is evaluated every time it is used.

```scala
scala> def add(x : Int, y : Int) = {
  println("normal add: "+ x + "+" + y); x+y }
add: (x: Int, y: Int) Int

scala> def lazyAdd(x: =>Int, y: => Int) = {
  println("lazy add"); x+y }

scala> add(add(1,2),add(2,3))
normal add: 1+2
normal add: 2+3
normal add: 3+5
res0: Int = 8

scala> lazyAdd(add(1,2),add(2,3))
lazy add
normal add: 1+2
normal add: 2+3
res1: Int = 8
```
By-Name Parameters

- We often want to evaluate the passed expression only once.

```scala
scala> def lazyAdd(x: =>Int, y: =>Int) =  
    { println("lazy add:" + x + " + " + y); x+y }

scala> lazyAdd(add(1,2), add(2,3))
normal add: 1+2
normal add: 2+3
lazy add 3 + 5
normal add: 1+2
normal add: 2+3
res1: Int = 8
```
By-Name Parameters

- We often want to evaluate the passed expression only once.

```scala
scala> def lazyAdd(x: =>Int, y: =>Int) = {
    println("lazy add:" + x + "+" + y); x+y
}

scala> lazyAdd(add(1,2), add(2,3))
normal add: 1+2
normal add: 2+3
lazy add 3 + 5
normal add: 1+2
normal add: 2+3
res1: Int = 8

scala> def lazyAdd(x: =>Int, y: =>Int) = {
    val a = x; val b = y;
    println("lazy add:" + a + "+" + b); a+b
}

scala> lazyAdd(add(1,2), add(2,3))
normal add: 1+2
normal add: 2+3
lazy add 3 + 5
normal add: 1+2
normal add: 2+3
res2: Int = 8
```
Control Structures with By-Name Parameters

Can use By-Name parameters to build control structures that look like built-in control structures.

```scala
var assertionsEnabled = true

def myAssert(predicate: () => Boolean) = 
  if (assertionsEnabled && !predicate())
    throw new AssertionError

scala> myAssert(() => 5 > 3)

val myAssert2: (=> Boolean) => Unit = 
  def myAssert2(predicate: => Boolean) = 
    if (assertionsEnabled && !predicate)
      throw new AssertionError

scala> myAssert2(5 > 3)
```
def makeString = 
  {println("in makeString"); "hello "+"world");}

def printString = {
  lazy val s = makeString
  print("in printString")
  println(s)
  println(s)
  println(s)
}

scala> printString
in printString
in makeString
hello world
hello world