Functions and Closures

Agenda

1. Private and Nested Methods
2. Function Literals
3. How Function Literals Work
4. Higher Order Functions
5. Placeholder Syntax
6. Partial Application of Functions
7. Closures
8. Partial Functions
9. Var Args in Methods
10. Argument Expansion
11. Named and Default Parameters

Private Methods

```scala
object FactSeq {

  def factSeq(n: Int): List[Long] = {
    factSeqInner(n, List(1L), 2)
  }

  @tailrec
  private def factSeqInner(n: Int, acc: List[Long], ct: Int): List[Long] = {
    if (ct > n) acc else
      factSeqInner(n, ct * acc.head :: acc, ct + 1)
  }
}

FactSeq.factSeq(8)
// List[Long] = List(40320, 5040, 720, 120, 24, 6, 2, 1)
```

- In, say, Java, implementation methods are hidden by making them private

Nested Methods

- In Scala, we have another alternative: methods may be nested in other methods

```
object FactSeqNested {

  def factSeq(n: Int): List[Long] = {
    @tailrec
    def factSeqInner(n: Int, acc: List[Long], ct: Int): List[Long] = {
      if (ct > n) acc else
        factSeqInner(n, ct * acc.head :: acc, ct + 1)
    }

    factSeqInner(n, List(1L), 2)
  }
}

FactSeqNested.factSeq(8)
// List[Long] = List(40320, 5040, 720, 120, 24, 6, 2, 1)
```

- Note that the `n: Int` in the `factSeqInner` hides the `n` from the outer `factSeq` method

Nested Method Scoping

- Because `n: Int` is in scope throughout the `factSeq` method, we can drop it from the `factSeqInner` definition:

```scala
object FactSeqScoped {

  def factSeq(n: Int): List[Long] = {
    @tailrec
    def factSeqInner(acc: List[Long], ct: Int): List[Long] = {
      if (ct > n) acc else
        factSeqInner(ct * acc.head :: acc, ct + 1)
    }

    factSeqInner(List(1L), 2)
  }
}

FactSeqScoped.factSeq(8)
// List[Long] = List(40320, 5040, 720, 120, 24, 6, 2, 1)
```

Function Literals

- Nested methods are handy, but they still need to be named
- A function literal (or lambda) is just a function (like a method) that may not have a name
- From the point of view of the caller, the syntax is interchangeable, e.g.

```
def multiplyMethod(a: Int, b: Int): Int = a * b
// multiplyMethod[](val a: Int,val b: Int) => Int

val multiplyFunction: (Int, Int) => Int = (a, b) => a * b
// (Int, Int) => Int = $Lambda$1281/148080390@6a84a9dd

multiplyMethod(2, 3)
// res0: Int = 6

multiplyFunction(2, 3)
// res1: Int = 6
```

- Note how the method has a name `multiplyMethod` but the Lambda just has a type. We assign it to a value so that we do have a name for it, but that is not required to use a function literal, only to identify it.

Using an Anonymous Function Literal

- E.g. if you call the `map` method on a list, there is no need to name the function passed:

```
val nums = (1 to 5).toList

nums.map(x => x * x)
// List[Int] = List(1, 4, 9, 16, 25)

nums.map(x => x * 3)
// List[Int] = List(1, 4, 9, 16, 25)

nums.map(x => x % 2 == 0)
// List[Boolean] = List(false, true, false, true, false)
```

- We use the `map` method for all three different functions. They are never assigned a name.

- Notice also that the third usage converts an `Int` to a `Boolean`, and the result of the `map` is then a `List[Boolean]`

How Function Literals Work

- Although they use Java 8 Lambdas now, behind the scenes the details are the same as they have always been for Scala function literals

```
val fn1: (Int, Int) => Int = (a, b) => a + b

val fn2 = new Function2[Int, Int, Int] {
  override def apply(a: Int, b: Int) = a + b
}

fn1(2, 3) // 5
fn1.apply(2, 3) // 5
fn2(2, 3) // 5
fn2.apply(2, 3) // 5
```

- Scala calls the `apply` method on any object or instance followed immediately by parens

- Therefore if we make a class or instance that overrides apply, that will be invoked by a *function call*

- When you create a new instance of a function, that is called a *function value*

Other Methods on Function

- In addition to an auto-generated `apply` method, when you define a function you also get:
- `.curried` (we'll look at currying a little later in the course)

```
val fn1curried = fn1.curried
fn1curried(2)(3)  // 5
```

- `.tupled`

```
val fn1tupled = fn1.tupled
val tup = (2, 3)

// fn1(tup) // won't compile

fn1tupled(tup) // 5
```

Higher Order Functions

e.g. map, filter, span, partition and more:

```
val nums = (1 to 10).toList

nums.map(x => x * x)
// List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)

nums.filter(x => x < 4)
// List(1, 2, 3)

nums.span(x => x % 4 != 0)
// (List(1, 2, 3),List(4, 5, 6, 7, 8, 9, 10))

nums.partition(x => x % 4 != 0)
// (List(1, 2, 3, 5, 6, 7, 9, 10),List(4, 8))
```

- Higher order functions are just functions (or methods) that take or return other functions
- If a method or function does not take or return another function, it is called a *first order function*

Writing a Higher Order Function

E.g. compare neighbors within a list using a function:

```
def compareNeighbors(xs: List[Int], compare: (Int, Int) => Int): List[Int] = {
  for (pair <- xs.sliding(2)) yield {
    compare(pair(0), pair(1))
  }
}.toList

compareNeighbors(nums, (a, b) => a + b)
// List(3, 5, 7, 9, 11, 13, 15, 17, 19)

compareNeighbors(List(4, 1, 7, 3, 4, 8), (a, b) => (a - b).abs)
// List(3, 6, 4, 1, 4)
```

- The `compare: (Int, Int) => Int)` is syntactic sugar for `Function2[Int, Int, Int]` and is the idiomatic Scala way to write a function literal type

Placeholder Syntax

- So far we have seen function definitions like `(a, b) => a + b` and `x => x * 2` but for very simple definitions, there is a syntactic shortcut

```
val nums = (1 to 10).toList

nums.filter(_ < 4)          // placeholder style for x => x < 4
nums.span(_ % 4 != 0)       // placeholder style for x => x % 4 != 0
nums.partition(_ % 4 != 0)

compareNeighbors(nums, _ + _) // placeholder style for (a, b) => a + b
```

- Placeholder can only be used where each parameter is used **exactly once in order**
- E.g. `_ * _` cannot be used in instead of `x => x * x` as x is used twice
- The _s cannot be inside parens either (that means something different), so

```
_ - _  // can be substituted for (a, b) => a - b
// but
(_ - _).abs // cannot be substituted for (a, b) => (a - b).abs
```

Placeholders With Types

- If you have a function definition like:

```
def compareNeighbors(xs: List[Int], compare: (Int, Int) => Int)
```

- When you call it, only `Int` params will compile, so Scala will infer those types, though we could also type the params explicitly:

```
compareNeighbors(nums, _ + _)  // can infer types
compareNeighbors(nums, (_: Int) + (_: Int))  // explicit types
```

- If you define a function literal where Scala has nothing to infer from, the types are mandatory:

```
val addPair = (_: Int) + (_: Int)
compareNeighbors(nums, addPair)

val addPair2 = (a: Int, b: Int) => a + b
compareNeighbors(nums, addPair2)
```

- `val addPair = _ + _` and `val addPair2 = (a, b) => a + b` will not compile, since Scala does not have enough information to infer the types

Partial Application of Functions

- Not to be confused with **Partial Functions** coming up shortly

```
val add3Nums = (a: Int, b: Int, c: Int) => a + b + c
// (Int, Int, Int) => Int = $Lambda$1701/51817638@2ecbf5a8

val add6and3 = add3Nums(6, (_: Int), 3)
// Int => Int = $Lambda$1702/435985129@4db04cd7

add6and3(5)  // 14
```

- The type is not optional on the placeholder in this case
- This also works with methods:

```
def add3Method(a: Int, b: Int, c: Int) = a + b + c
// add3Method[](val a: Int,val b: Int,val c: Int) => Int

val add4and7 = add3Method(4, _: Int, 7)
// Int => Int = $Lambda$1718/831478568@22e3cec7

add4and7(2) // 13
```

You Can Partially Apply All the Parameters

```scala
def add3Method(a: Int, b: Int, c: Int) = a + b + c

val add3Functionv1 = add3Method(_, _, _)
add3Functionv1(1,2,3)  // 6

val add3Functionv2 = add3Method _  // alternative when replacing all params
add3Functionv2(1,2,3)  // 6
```

```scala
def compareTriplets(xs: List[Int], compare: (Int, Int, Int) => Int): List[Int] = {
  for (triplet <- xs.sliding(3)) yield {
    compare(triplet(0), triplet(1), triplet(2))
  }
}.toList

val nums = (1 to 10).toList

compareTriplets(nums, add3Functionv1)
// List(6, 9, 12, 15, 18, 21, 24, 27)
compareTriplets(nums, add3Functionv2)
// List(6, 9, 12, 15, 18, 21, 24, 27)
compareTriplets(nums, add3Method)  // eta expansion
// List(6, 9, 12, 15, 18, 21, 24, 27)
```

Closures

- All closures are function literals, but not all function literals are closures
- A closure is so-called because it encloses around some other state than that passed in to the function as parameters

```
val incBy1 = (x: Int) => x + 1        // not a closure
val more = 10
val incByMore = (x: Int) => x + more  // a closure

incBy1(10)    // 11
incByMore(10) // 20
```

- In Scala, closures can be made over vars!

```
var more = 10
val incByMore = (x: Int) => x + more

incByMore(12)  // 22
more = 100
incByMore(12)  // 122
```

- This is confusing, and usually unintentional. Don't do that! Take a defensive `val` copy of any state before using it

Partial Functions

- Not to be confused with *partially applied functions*
- A `PartialFunction[T, R]` extends `Function1[T, R]` (which is idiomatically written `T => R`)
- It can therefore be used in place of any `Function1[T, R]`
- Any block of code with `case` inside of `{}`s is a Partial Function:

```scala
val pf1: PartialFunction[Int, Int] = {
  case x: Int if x > 0 => x + x
  case x => x * -1
}

val fn1: Int => Int = pf1  // upcast

val nums = (-5 to 5).toList

nums.map(pf1)
// List(5, 4, 3, 2, 1, 0, 2, 4, 6, 8, 10)
```

Partial Functions

- In the previous example, the function was complete for all inputs, but it needn't be:

```
val pf2: PartialFunction[Int, Int] = {
  case x: Int if x > 0 => x + x
}

nums.map(pf2)  // MatchError!
```

- You get a `MatchError` thrown if there is no case to handle the input

- You can also ask `PartialFunctions` if they are defined for an input:

```
pf2.isDefinedAt(5)   // true
pf2.isDefinedAt(-5)  // false
```

Partial Functions

- `map` may not be safe with a `PartialFunction`, but `collect` is:

```
nums.map(pf1)

val pf2: PartialFunction[Int, Int] = {
  case x: Int if x > 0 => x + x
}

nums.map(pf2)      // MatchError!

nums.collect(pf2)  // List(2, 4, 6, 8, 10)
```

- `match` and `catch` use `PartialFunctions`

```
a match {
  case 4 => "It's four!"
}

try (1 / 0)
catch {
  case ae: ArithmeticException => 0
}
```

Var Args in Methods

- Ever wonder how:

```
val xs = List(1,2,3)
val ys = List(1,2,3,4,5)
```

  works for different numbers of params?

- We know Scala re-writes to:

```
val xs = List.apply(1,2,3)
val ys = List.apply(1,2,3,4,5)
```

  so are there just lost of overridden apply methods?

Var Args

```
def sayHello(names: String*): Unit = {
  for (name <- names) println(s"Hello, $name")
}

sayHello()
sayHello("Fred")
sayHello("Fred", "Julie", "Kim")

def greet(greeting: String, names: String*): Seq[String] = {
  for (name <- names) yield s"$greeting $name"
}

greet("Hi", "Fred", "Julie", "Kim")
```

- The var args parameter has a * after it
- It must always be the last parameter
- The parameter comes in as `Array[T]` for a parameter defined `item: T*`

Calling var args with a Collection

- What if we want to greet an existing collection of names?

```
// greet a seq of names:
val names = List("Fred", "Julie", "Kim")

greet("Hi", names)  // does not compile
```

- For this, we use the **expansion operator**:

```
greet("Hi", names: _*)  // expansion operator
// List(Hi Fred, Hi Julie, Hi Kim)
```

- Note that if using expansion operator, the original collection type is retained (in this case, `List`) instead of converting to `Array`
- The expansion operator is occasionally useful, particularly for recursion over var-args methods

Named and Default Parameters

- All method parameters have names:

```
def greet(name: String): String =
  "hello" + name   // here we use the parameter name

greet("Fred")
```

- We can also use that `name` outside of the method when we call it:

```
greet(name = "Fred")
```

- This is considered best practice for `Boolean` parameters in particular:

```
def thingy(isCold: Boolean, isBroken: Boolean): Unit = {}

thingy(true, false) // doesn't tell us much
thingy(isCold = true, isBroken = false) // is much more readable
```

- And if you use the names, you can choose any order:

```
thingy(isBroken = false, isCold = true) // exactly the same meaning as above
```

Couple With Default Parameters

E.g.

```
def gravity = 9.81  // meters/sec

def force(mass: Double = 1, acceleration: Double = gravity) =
  mass * acceleration

force()      // 9.81
force(12)    // 117.72

force(acceleration = 2 * gravity)    // 19.62
force(acceleration = gravity / 13.0, mass = 100) // 75.46153846153847
```

or for recursion:

```
def factSeq(n: Int, acc: List[Long] = List(1L), ct: Int = 2): List[Long] = {
  if (ct > n) acc else factSeq(n, acc = ct * acc.head :: acc, ct = ct + 1)
}
```