Class 5: Scala Flow Control

New York University

Summer 2017

Agenda

1. Looping
2. Using Iterators
3. Writing Functions
4. Passing Functions as Arguments
5. Collection Iteration Methods
6. Pattern Matching
7. Processing Data with Partial Functions

- Functional programming flow control is
  - Different from both imperative and object-oriented flow control
- Imperative: program *explicitly* operates on data
- Object-oriented: program *explicitly* invokes a method
- Functional: program *implies* what needs to be done
  - Framework figures out how to satisfy requirements

- **`while`** loops are typical of imperative programming
  - Can be used in Scala, but not a best practice

```
val sorrentoPhones = List("F00L", "F01L", "F10L", "F11L",
"F20L", "F21L", "F22L", "F23L", "F24L")

var i = 0
while (i < sorrentoPhones.length){
  println(sorrentoPhones(i))
  i = i + 1
}
```


- The **<-** syntax in Scala Spark programs is an enumerator *generator*
  - You must adjust the number of iterations when using **to** because it is inclusive
  - Use **until** to avoid this extra math to adjust for length
  - The **by** keyword allows you to increment by a custom value

```
for (i <- 0 to sorrentoPhones.length - 1) {
  println(sorrentoPhones(i))
}

for (i <- 0 until sorrentoPhones.length) {
  println(sorrentoPhones(i))
}

for (i <- 0 until sorrentoPhones.length by 2) {
  println(sorrentoPhones(i))
}
```

- In this example, access to the index is necessary in order to print it, so we must use a loop form that lets us access the loop counter variable
- When possible, remove the local counting variable because it limits scalability

```
for (i <- 0 until sorrentoPhones.length) {
  println(i.toString + ": " + sorrentoPhones(i))
}
> 0: F00L
> 1: F01L
> 2: F10L
> 3: F11L
> 4: F20L
> 5: F21L
> 6: F22L
> 7: F23L
> 8: F24L
```


- This is the preferred form of explicit iteration in Scala
  - No loop counter variable
  - No bounds issues, no mutability issue to limit scalability
- The generator already knows to process each item in the collection

```
for (model <- sorrentoPhones) {
  print(model + " ")
}

> F00L F01L F10L F11L F20L F21L F22L F23L F24L
```

- Generators within the **`for()`** must be separated by semicolons
  - They are treated as if they were nested **`for`** loops, left to right

```
val phonebrands = List("iFruit", "MeToo")
val newmodels = List("Z1", "Z-Pro")

for (brand <- phonebrands; model <- newmodels) {
   println(brand + " " + model)
}

iFruit Z1
iFruit Z-Pro
MeToo Z1
MeToo Z-Pro
```

Conditional Statements

- **Use `if`** to filter out items that do not match the condition
- In this case, the loop generates *each item* and then prints those items that match the criteria

```
val sorrentoPhones = List("F00L", "F01L", "F10L", "F11L",
"F20L", "F21L", "F22L", "F23L", "F24L")

for (model <- sorrentoPhones) {
  if (model.contains("2")) print(model + " ")
}

> F20L F21L F22L F23L F24L
```

Conditional Statements *(continued)*

- A better approach when writing Scala Spark programs is to move the `if` condition inside the `for` loop
  - This is called a generator *filter*
- *Scala will only generate items that match the filter criteria*

```
val sorrentoPhones = List("F00L", "F01L", "F10L", "F11L",
"F20L", "F21L", "F22L", "F23L", "F24L")

for (model <- sorrentoPhones; if (model.contains("2"))) {
  print(model + " ")
}

> F20L F21L F22L F23L F24L
```


- **`yield`** returns a new collection of items

```
val phonebrands = List("iFruit", "MeToo")
val newmodels = List("Z1", "Z-Pro")

val newlist =
  for (brand <- phonebrands; model <- newmodels)
    yield brand + " " + model

> newlist: List[String] = List(iFruit Z1, iFruit Z-Pro,
  MeToo Z1, MeToo Z-Pro)
```


Agenda

1. Looping
2. Using Iterators
3. Writing Functions
4. Passing Functions as Arguments
5. Collection Iteration Methods
6. Pattern Matching
7. Processing Data with Partial Functions

- `Iterators` are used for iterating over elements in a collection
  - `Iterators` can refer to distributed elements
  - Iterators are scalable, making them ideal for Big Data applications

- Create an **`Iterator`** from a collection using **`toIterator`**
  - For a tuple use **`productIterator`**
- The iterator is used one time – using it is “destructive”

```
val phones = Array("iFruit", "MeToo")

val iter = phones.toIterator
> iter: Iterator[String] = non-empty iterator

iter.next
> String = iFruit

iter.next
> String = MeToo

iter.next
> java.util.NoSuchElementException: next on empty iterator
```

- This example shows the preferred use of **`while`** in Scala
  - This is preferred because there are no counting variables or I/O dependencies

```
val titanicPhones = List("1000", "2000", "3000", "Bananas")

val iter = titanicPhones.toIterator

print(iter.next)
> 1000

print(iter.next)
> 2000

while (iter.hasNext) {
  print(iter.next + " ")
}
> 3000 Bananas
```

Key methods for working with iterators

Method	Description
`size`	The remaining number of elements
`isEmpty`	`true` if there are remaining elements
`exists(`*`element`*`)`	`true` if the element exists in the list
`take(`*`n`*`)`	Returns a new `Iterator` with just the next *`n`* elements
`filter(boolean-`*`expression`*`)`	Returns a new `Iterator` with elements for which the expression is `true`
`foreach(`*`function`*`)`	Execute *`function`* for each element provided by the iterator

Agenda

1. Looping
2. Using Iterators
3. Writing Functions
4. Passing Functions as Arguments
5. Collection Iteration Methods
6. Pattern Matching
7. Processing Data with Partial Functions

- Variable types and values are evaluated immediately upon assignment
  - Contrast this with functions, where only the type is evaluated when defined
  - The value is evaluated later, when the function is called

```
val myConstant = 10

var myVariable = 24

def myFunction = myConstant + myVariable
> myFunction: Int

myFunction
> Int = 34
```



```
val myConstant = 10
var myVariable = 24

def myFunction = myConstant + myVariable
> myFunction: Int

myVariable = 9
myFunction
> Int = 19

myVariable = 20
myFunction
> Int = 30

val myConstant = 3
myFunction
> Int = 30
```

myFunction evaluates to a different result when **myVariable** is reassigned to **20** because the value is passed in by reference

However, when **myConstant** is reassigned to **3**, there is no change to the result returned by **myFunction** because **myConstant** was passed by value, not by reference

- The multi-line function definition uses curly braces
- All functions return something
  - If there is no explicit return type, Scala returns **Unit**
- Parentheses are only required if the function accepts parameters

```
def listPhones {
  println("MeToo")
  println("Titanic")
  println("iFruit")
}
> listPhones: Unit

listPhones
> MeToo
> Titanic
> iFruit
```


Agenda

1. Looping
2. Using Iterators
3. Writing Functions
4. Passing Functions as Arguments
5. Collection Iteration Methods
6. Pattern Matching
7. Processing Data with Partial Functions

```
def CtoF(celsius: Double) = {
   (celsius * 9 / 5) + 32
}
> CtoF: (celsius: Double)Double

CtoF(34.0)
> Double = 93.2

def CtoF(celsius: Double) =
   (celsius * 9 / 5 ) + 32

def CtoF(celsius: Double) =
   (celsius * 9 / 5 ) + 32 : Double
```

- Use **=** to define a function with a return value
- No **`return`** keyword
- *The result from the final expression is returned*

For simple expressions, the curly braces are not needed

Return type may be explicit or inferred

- **`convertList`** is called a higher-order function because it takes another function as a parameter
- **`convert`** is the name of the ***parameter that accepts a function***
  - **`convert`** specifies the type for the input parameter to the left of the **=>** transformation symbol
  - It specifies the return type to the right of **=>**

```
def CtoF(celsius: Double) = (celsius * 9 / 5) + 32

def convertList(myList:List[Double],
                convert:(Double) => Double) {
  for(n <- myList)
    println(n,convert(n))
}
> convertList: (myList: List[Double],
convert: Double => Double)Unit
```

```
def CtoF(celsius: Double) = (celsius * 9 / 5) + 32

def convertList(myList:List[Double],
                convert:(Double) => Double)
{
  for(n <- myList)
    println(n,convert(n))
}
> convertList: (myList: List[Double],
convert: Double => Double)Unit

val phoneCelsius = List(34.0, 23.5, 12.2)

convertList(phoneCelsius, CtoF)
> (34.0,93.2)
> (23.5,74.3)
> (12.2,53.96)
```

In this case, **`CtoF`** is the function passed into the **`convert`** parameter

- Anonymous functions are an alternate syntax for defining functions
  - They do not require a function name or label
  - Also referred to as *lambda functions*
- Anonymous functions in source code are called function literals
  - Often used when a function will be called only once

- An anonymous function is a way to define a function *inline*

```
(parameter: type) => {function_definition: type}
```

parameter names and types

code

return type

- This example converts temperature from Celsius to Fahrenheit

```
def convertList(myList:List[Double], convert:(Double) => Double)
{
  for (n <- myList)
    println(n, convert(n))
}

val phoneCelsius = List(34.0, 23.5, 12.2)

convertList(phoneCelsius, cc => (cc * 9 / 5) + 32)
> (34.0,93.2)
> (23.5,74.3)
> (12.2,53.96)
```

A function literal can be used in the call to a higher-order function as an anonymous function.

Agenda

1. Looping
2. Using Iterators
3. Writing Functions
4. Passing Functions as Arguments
5. Collection Iteration Methods
6. Pattern Matching
7. Processing Data with Partial Functions

- Commonly used collection methods include
  - **`foreach`**
  - **`map`**
  - **`filter`**
- These methods support scalability
  - They delegate control over iteration to the framework

`foreach`

- **`List`** inherits the **`foreach`** method
- The _ (underscore) is a placeholder variable
  - It is a reference to the current element being operated on by **`foreach`**

```
val phones = List("MeToo", "Titanic", "Ronin")

phones.foreach(println(_))
> MeToo
> Titanic
> Ronin

phones.foreach(println)
> MeToo
> Titanic
> Ronin
```

These two lines are equivalent

foreach

- Using _ may create ambiguity that prevents Scala from inferring the type
  - This is illustrated in the first example below
- In these cases, the type must either be specified or made more inferable
  - The second example hints to Scala that the list contains **Strings**

```
val phones = List("MeToo", "Titanic", "Ronin")

phones.foreach(println(_).toUpperCase)
> <console>:12: error: missing parameter type for expanded
function ((x$1) => x$1.toUpperCase)
       phones.foreach(println(_.toUpperCase))

phones.foreach(println(_).toString.toUpperCase)
> MeToo
> Titanic
> Ronin
```

map

```
def CtoF(celsius: Double) = celsius * 9 / 5 + 32

val phoneCelsius = List(34.0, 23.5, 12.2)
```

```
phoneCelsius.map(c => CtoF(c))
> List[Double] = List(93.2, 74.3, 53.96)
```

Passing a named function

```
phoneCelsius.map(CtoF(_))
> List[Double] = List(93.2, 74.3, 53.96)
```

Using a placeholder parameter

```
phoneCelsius.map(c => c * 9 / 5 + 32)
> List[Double] = List(93.2, 74.3, 53.96)
```

Passing an anonymous function (function literal)

```
phoneCelsius.map(_ * 9 / 5 + 32)
> List[Double] = List(93.2, 74.3, 53.96)
```

Passing an expression with a placeholder parameter

`filter`

- In this example, the underscore placeholder refers to a numeric
- Create the filter condition using relational operators
- In the example, there is an implicit conversion of the integer literal to a floating point value

```
val phoneCelsius = List(34.0, 23.5, 12.2)

phoneCelsius.filter(val1 => val1 < 23)
> List[Double] = List(12.2)

phoneCelsius.filter(_ < 23)
> List[Double] = List(12.2)
```

filter

- Since the placeholder in this case refers to a **String**, we can call string methods like **startsWith** and **length** on the placeholder

```
val phones = List("1000", "2000", "2500", "Bananas")

phones.filter(_.startsWith("2"))
> List[String] = List(2000, 2500)

phones.filter(_.length > 4 )
> List[String] = List(Bananas)
```

`sortWith`

- **`sortWith`** uses the passed in operator to compare the two elements
  - The first underscore refers to the first parameter, the second one refers to the second parameter

```
val phoneCelsius = List(34.0, 23.5, 12.2)

phoneCelsius.sortWith((val1, val2) => val1 < val2)
> List[Double] = List(12.2, 23.5, 34.0)

phoneCelsius.sortWith(_ < _)
> List[Double] = List(12.2, 23.5, 34.0)

phoneCelsius.sortWith(_ > _)
> List[Double] = List(34.0, 23.5, 12.2)
```

```
var myList: List[Int] = List(1, 5, 7, 3, 2, 1)

myList.map(_ + 10)
> List[Int] = List(11, 15, 17, 13, 12, 11)

myList.filter(_ > 4)
> List[Int] = List(5, 7)

myList.map(_ + 1).filter(_ > 4)
> List[Int] = List(6, 8)
```

```
titanicPhones.filter(_.endsWith("00")).sortWith(_ > _)
> List[String] = List(2500, 2000, 1000)
```


Agenda

1. Looping
2. Using Iterators
3. Writing Functions
4. Passing Functions as Arguments
5. Collection Iteration Methods
6. Pattern Matching
7. Processing Data with Partial Functions

- **`case`** can match any literal of any type

```
val phoneWireless = "enabled"
var msg = "Radio state Unknown"

phoneWireless match {
  case "enabled"    => msg = "Radio is On"
  case "disabled"   => msg = "Radio is Off"
  case "connected"  => msg = "Radio On, Protocol Up"
}

println(msg)
> Radio is On
```

- A **`match`** can implicitly return a value
  - **`msg`** is assigned the result of the **`match`**…**`case`**

```
val phoneWireless = "happy"
var msg = "unknown"

val msg = phoneWireless match {
   case "enabled"     => "Radio is on";
   case "disabled"    => "Radio is off";
   case "connected"   => "Radio on, protocol up";
   case default       => "Radio state unknown"
}

println(msg)
> Radio state unknown
```

- This array has a mix of types, use **match...case** to process each type
- Do you expect **'F'** to be reported as a **Char**?

```
val mixedArr = Array("11", 12, "thirteen", 14.0, 'F', null)

for (elem <- mixedArr) {
  elem match {
    case elem:String => println("String:    " + elem)
    case elem:Int    => println("Integer:   " + elem)
    case elem:Double => println("Float:     " + elem)
    case elem:AnyRef => println("Unknown:   " + elem)
    case elem:Char   => println("Char:      " + elem)
    case null        => println("Found null")
  }
}
```

- **'F'** is reported as "Unknown"

```
String:   11
Integer:  12
String:   thirteen
Float:    14.0
Unknown:  F
Found null
```

- The ordering of **case** statements within a **match** is significant
  - The first **case** that matches is executed
- Reorder the **case** statements to get the intended result
  - In this case, **elem:Char** must precede **elem:AnyRef**

- An **`Option`** is a special type with a value of **`Some(`*`n`*`)`** or **`None`**
- An **`Option`** can be used to "wrap" a function that would potentially throw an error if it produced an illegal value
- If the value is good, then it is returned wrapped in **`Some`**
- **`Option`** can be used in a **`match…case`** by the caller

- `Some(x)` contains the value, where *x* is the returned value
- `Some` and `None` can be explicitly set, as illustrated
- `getOrElse`

```
val superPhone = Some("Model 6")
> superPhone: Some[String] = Some(Model 6)

superPhone.getOrElse("Not found")
> String = Model 6

val superPhone = None
> superPhone: None.type = None

superPhone.getOrElse("Not found")
> String = Not found
```


- This example shows a common use of **`Option`** in functions
  - The function returns a value encapsulated in a **`Some`** / **`None`**

```
def str2Double(in: String): Option[Double] = {
  try {
    Some(in.toDouble)
  } catch {
    case e: NumberFormatException => None
  }
}

str2Double("35.2")
> Option[Double] = Some(35.2)

str2Double("Warm")
> Option[Double] = None
```

- Process **`Some(x)`** inputs
  - In this example, we use typed pattern matching

```
def convert2Float(x: Option[Any]) = x match {
  case Some(d: Double) => d.toFloat
  case Some(i: Int)    => i.toFloat
  case Some(f: Float)  => f
  case Some(_: Any)    => println("Invalid data provided.")
  case None => println("No data provided.")
}

convert2Float(Some(25.0))
> AnyVal = 25.0

convert2Float(Some(25F))
> AnyVal = 25.0

convert2Float(Some(25))
> AnyVal = 25.0
```

- Example to process **None** inputs and **Any** inputs

```
def convert2Float(x: Option[Any]) = x match {
  ...
  case Some(_: Any) => println("Invalid data provided.")
  case None => println("No data provided.")
}

convert2Float(Some("twenty-five"))
> Invalid data provided.
  AnyVal = ()

convert2Float(None)
> No data provided.
  AnyVal = ()
```


Agenda

1. Looping
2. Using Iterators
3. Writing Functions
4. Passing Functions as Arguments
5. Collection Iteration Methods
6. Pattern Matching
7. Processing Data with Partial Functions

- A *partial function* is used when an answer should be returned only for a subset of possible input values
  - Defines the (partial) data it can handle
  - Can be queried to determine whether a given value can be handled
- Simple examples where partial functions can be useful
  - Division by zero
  - Square root of a negative number

- Take divide by zero as an example

```
val div = (x: Int) => 24 / x
```

- Providing a zero for **x** will cause an arithmetic exception
  - Partial functions can offer a way to avoid such an exception

- Must be declared as a **`PartialFunction`**
- **`PartialFunction`** defines two methods that you must implement
  - **`apply`** performs the actual processing for your method
  - **`isDefinedAt`** evaluates whether the supplied input is valid

```
val div = new PartialFunction[Int, Int] {
  def apply(x: Int) = 24 / x
  def isDefinedAt(x: Int) = x != 0
}
```

- Partial functions allow a caller to test an input before using it as a parameter

```
val div = new PartialFunction[Int, Int] {
  def apply(x: Int) = 24 / x
  def isDefinedAt(x: Int) = x != 0
}
```

```
div.isDefinedAt(0)
> Boolean = false

div.isDefinedAt(2)
> Boolean = true

if (div.isDefinedAt(2)) div(2)
> AnyVal = 12
```


- **When a partial function includes one or more `case` statements, the `apply` and `isDefinedAt` methods are generated automatically**

```
val getThirdItem: PartialFunction[List[Int], Int] = {
  case x :: y :: z :: _ => z
}

getThirdItem.isDefinedAt(List(25))
> Boolean = false

getThirdItem.isDefinedAt(List(25, 35, 45, 85))
> Boolean = true

getThirdItem(List(25, 35, 45, 85))
> Int = 45
```

- Use complete functions whenever possible
- A partial function may compile fine, but you may experience runtime errors for unhandled values
- Partial functions are useful when you are certain that
  - An unhandled value will never be supplied
  - Values are always checked with **`isDefinedAt`** before an explicit or implicit call to the **`apply`** method

- Scala supports imperative programming and functional programming
- Scala provides iterative methods for scalability
- Scala supports higher-order functions
- If possible, use Collection methods rather than imperative programming
- Pattern matching behaves differently from “switch” in other languages

Homework

See the homework packet for details.