Reference


Reading

Chapters 7 & 8
Control Structures
If

def example(x: Int, y: Int): Int {
    var min = 0;
    if (x < y)
        min = x
    else
        min = y
    return min
}

def example(x: Int, y: Int): Int = {
    val min =
        if (x < y)
            x
        else
            y
    return min
}
if else Problems

OK

val x = 2;
val y = 3;
var min = y;
if (x < y)
    min = x

if (x < y) min = x else min = y

{if (x < y)
    min = x
else
    min = y}

Compile Error

val x = 2;
val y = 3;
var min = 0;
if (x < y)
    min = x
else
    min = y
Recommended Style

val x = 2;
val y = 3;
val min = if (x < y) x else y
While

```java
var a = 10
var b = 6;
while (a != 0) {
    val temp = a
    a = b % a
    b = temp
}
println(b)
```
do - while

```java
var line = ""
do {
    line = readLine()
    println("read: " + line)
} while (line != ")
```
C/C++/Java while idom does not work

```java
var line = ""
while ((line = readline()) != "")
    println("read: " + line)
```

```java
line: java.lang.String =
<console>:6: warning: comparing values of types Unit and java.lang.String using `!' will always yield true
    while ((line = readline()) != "")
```
for (k <- 5 to 2 by -1)
    println(k)

for (k <- 2 to 5)
    println(k)

for (k <- 2 until 5 by 2)
    println(k)

val names = Array("Roger", "Jim", "Sam")

for (name <- names)
    println(name)

val localFiles = (new java.io.File(".")).listFiles

for (file <- localFiles)
    println(file)
for { k <- 5 to 2 by -1}
    println(k)

for {k <- 2 to 5}
    println(k)
val names = Array("Roger", "Jim", "Sam", "Randy")

for (name <- names
    if name.startsWith("R")
  )
  println(name)
Multiple Filters

val names = Array("Roger", "Jim", "Sam", "Randy")

for {
    name <- names
    if name.startsWith("R")
    if name.endsWith("r")
} println(name)

Output
Roger
val names = Array("RoGeR", "Jim", "Sam", "RAndy")

for {
  name <- names
  if name.startsWith("R")
    character <- name.toCharArray()
    if character.isUpperCase
      println(name + " : " + character)
}
val names = Array("Roger", "Jim", "Sam", "Randy")

val rNames = for {
  name <- names
  if name.startsWith("R")
} yield name

Result
rNames  = Array(Roger, Randy)
val input = readLine

input match {
  case "cat" => println("mouse")
  case "dog" => println(5)
  case _ => println("all others")
}

val result = input match {
  case "cat" => "mouse"
  case "dog" => 5
  case _ => "all others"
}
break & continue

Not in Scala
def sum(items: List[Int]) = {
  var sum = 0
  for (item <- items)
    sum += item
  sum
}

def average(items: List[Int]) = {
  if (items.length == 0)
    throw new RuntimeException("empty list")
  sum(items)/items.length
}

try {
  average(List(1,2,3))
  average(List())
} catch {
  case except: java.io.IOException => println(except)
  case exception: RuntimeException => println(exception)
}
```scala
val result = try {
    average(List(1,2,3))
} catch {
    case except: java.io.IOException => println(except)
    case exception: RuntimeException => println(exception)
} finally {
    println("always done")
}
```
Using the return value

```java
import java.net.URL
import java.net.MalformedURLException

def urlFor(path: String) =
    try {
        new URL(path)
    } catch {
        case e: MalformedURLException =>
            new URL("http://www.google.com")
    }
```
Declaring throws & Check Exceptions

Scala does not require
  catching exceptions
  declaring that a method throws an exception
Functions
Parameter Passing

All function/method parameters are val

Scala does not specify if
  pass by value
  pass by reference

def function(x: Array[Int]) {
  x(0) = 10
}

var y = Array(1,2,3)
function(y)
println(y(0))

But once parameters are constant the semantics are the same. Java only has pass by value so Scala likely uses pass by value
vals verses vars

```scala
val y = Array(1,2,3)
y(0) = 10
y = Array(3,2)    //compile error

var y = Array(1,2,3)
y(0) = 10
y = Array(3,2)    //OK
```
Nesting Functions

def average(items: List[Int]) = {
    def sum(items: List[Int]) = {
        var sum = 0
        for (item <- items)
            sum += item
        sum
    }

    if (items.length == 0)
        throw new RuntimeException( "empty list")
    sum(items)/items.length
}

Really you can nest Functions

for (k <- 1 to 5) {
    def sum(items: List[Int]) = {
        var sum = 0
        for (item <- items)
            sum += item
        sum
    }
    println( k)
}


First Class Functions

```
var next = (x: Int) => x + 1
val previous = (x: Int) => x - 1

next(4)
previous(3)

next = x => x + 2
next(4)
```

```scala
def example(test: (Int => Int)) {
    println( test(4))
}

example (previous)
example (next)
```

"a programming language is said to support first–class functions (or function literals) if it treats functions as first–class objects. Specifically, this means that the language supports constructing new functions during the execution of a program, storing them in data structures, passing them as arguments to other functions, and returning them as the values of other functions." http://en.wikipedia.org/wiki/First–class_function
Types & First Class Functions

```typescript
var next: (x: Int) => x + 1
next(4)

next = (x: String) => x + "ing"  //Compile Error
```

```typescript
var next: (Int => Int) = (x: Int) => x + 1
```
Scala Verses Java

var next = (x: Int) => x + 1

val previous = (x: Int) => x - 1

var result = next(4)
result = previous(3)

interface Operation {
    int dolt(int x);
}

class Adder implements Operation {
    public int dolt(int x) { return x + 1;}
}

public class Example {
    public static void main(String[] args) {
        Operation next = new Adder();

        Operation previous = new Operation() {
            public int dolt(int x) {
                return x - 1;
            }
        };
        int result = next.dolt(4);
        result = previous.dolt(3);
    }
}

Java does not have first-class functions. One can create Anonymous classes that have one method, which is not the same a first-class function.
filter

Returns all the elements of collection that satisfy the argument

val numbers = List(-5, 5, 10, -2, -3, 8)
val result = numbers.filter((x:Int) => x >0)

result: List[Int] = List(5, 10, 8)
Other Forms

numbers.filter((x: Int) => {x > 0})
numbers.filter(x => x > 0)
numbers.filter(_ > 0)
More Complex Example

val numbers = List(-5, 5, 10, -2, -3, 8)
var maxSoFar = numbers(0) -1;

val result = numbers.filter((x:Int) => {
  val isLarger = x > maxSoFar
  if (isLarger) maxSoFar = x
  isLarger
})

Result
result: List[Int] = List(-5, 5, 10)
val numbers = List(-5, 5, 10, -2, -3, 8)
var maxSoFar = numbers(0) - 1;

val increasingNumbers = (x:Int) => {
  val isLarger = x > maxSoFar
  if (isLarger) maxSoFar = x
  isLarger
}

val result = numbers.filter(increasingNumbers)
Some Other Useful Collection Methods

- foreach
- map
- reduceLeft
foreach

foreach (f : (A) => Unit) : Unit
Apply the given function f to each element

val numbers = List(1,2,3)
var sum = 0
numbers.foreach(sum += _)
println(sum)

numbers.foreach((x: Int) => {sum += x})
numbers.foreach(x => sum += x)
numbers.foreach(sum += _)
map

map [B](f : (A) => B) : List[B]
Returns the list resulting from applying the given function f to each element

val numbers = List(1,2,3)  
val result = numbers.map(_ + 1)  
println(result)

Output
List(2, 3, 4)

numbers.map((x: Int) => {x + 1})  
numbers.map(x => x + 1)  
numbers.map(_ + 1)
reduceLeft

Combines the elements of this list together using the binary operator

```
val numbers = List(1,2,3,4)
val result = numbers.reduceLeft(_ + _)
println(result)
```

Output

```
10
```
Closures

```
var more = 1;
val adder = (x: Int) => {more + x}
more = 2
val result = adder(2)
println( result)
```

```
def test(op: Int => Int) {
  var more = 10;
  val result = op(1)
  println(result)
}
```

```
var more = 1;
val adder = (x: Int) => {more + x}
test(adder)
```

Output
4

Output
2
def test(): Int => Int = {
    var more = 10;
    val testAdder = (x: Int) => {more + x}
    return testAdder
}

def overWriteMore() {
    var more = 5
    more += 2
}

var more = 1;
val adder = test()
overWriteMore()
val result = adder(2)
println( result)
Bound verse Free Variables

def example(x: Int) {
    var y = 1;
    return y + x + outside
}

Bound
x
y

Free
outside
Closure

a first-class function with free variables that are bound in the lexical environment

The environment for the free variables exists as long as the first-class function exists
Tail Recursion

```scala
def boom(n: Int) : Int = {
  if (n == 0)
    throw new Exception("boom")
  else n + boom(n -1)
}

boom(3)
```

```java
java.lang.Exception: boom
    at .boom(<console>:6)
    at .boom(<console>:7)
    at .boom(<console>:7)
    at .boom(<console>:7)
    at .<init>(<console>:6)
    at .<clinit>(<console>)
```

```scala
def bang(n: Int) : Int = {
  if (n == 0)
    throw new Exception("bang")
  else bang(n -1)
}

bang(3)
```

```java
java.lang.Exception: bang
    at .bang(<console>:6)
    at .<init>(<console>:6)
    at .<clinit>(<console>)
```
Special Syntax for One argument

def next(x: Int) = \{x + 1\}
val a = next(1)
val b = next \{1 \}