Scala Language 03

# Conditionals

Like Java, conditionals can be expressed using an if/else expression. The primary difference is that in Scala, if..else is an expression and returns a value. For example

val answer = if (1 == 2) “yes” else “no”

answer

So if/else is more like the ternary ?: operator in Java than Java’s if else statement.

# Equality

In scala, == is equivalent to Java .equals, meaning the natural, logical equals of two objects. So

“abc” == “abc”

is the same as

“abc”.equals(“abc”)

If you want to test object identity, use the eq operator. For example,

“abc” eq “abc”

# Pattern Matching

The "match" operator is like "switch" on steroids. Can do simple matches on numbers like

1 match {

case 1 => "one"

case 2 => "two"

}

Can match strings

"green" match {

case "green" => "go"

case \_ => "stop"

}

"orange" match {

case "green" => "go"

case \_ => "stop"

}

Here, the \_ case is a default case.

Deconstruct a case class and pick out an element:

abstract class Person(val name: String)

class Employee(override val name: String, val salary:Int) extends Person(name)

case class Engineer(override val name: String, override val salary:Int,

languages:List[String]) extends Employee(name, salary)

val mike = Engineer("Mike", 33000, List("Scala"))

mike match {

case Engineer(name,salary,List(language)) => s"${name} is an engineer and knows ${language}!"

}

Deconstruct a list and pick out the tail

val l = List(1,2,3,4)

l match {

case Nil => Nil

case x :: y => y

}

def matchTest(x: Any) = x match {

case 1 => "one"

case "two" => 2

case y: Int if y %2 == 0 => "even"

case \_: Int => "An Int"

case \_: String => "A String"

case thing @ \_ => s"Not sure what $thing is."

}

matchTest(1)

matchTest(“two”)

matchTest(2)

matchTest(3)

matchTest(“four”)

matchTest(5L)

# Extractors

So far we’ve seen pattern matching applied to case classes to break up the structure of the classes into constituent pieces. But what if we want to apply pattern matching to objects which are not case classes? For example, what if we want to pull out a name and a domain from a string representing an email?

We can do this by defining an “unapply” method on an object. This is called an extractor. The unapply method can be thought of as the opposite of the apply method. The apply method puts an object together, and the unapply method breaks it apart. Here’s an example

object Email {

// constructor

def apply(user: String, domain: String) = user + “@” + domain

// extractor

def unapply(s: String): Option[(String, String)] = {

val parts = s split “@”

if (parts.length == 2) Some((parts(0), parts(1))) else None

}

}

we can define a function that uses pattern matching

def emailer(s: String) = s match {

case Email(user, domain) =>

println(s“sending email to $user at $domain”)

case \_ => println(s“$s is not an email address”)

}

and call it on a couple examples

email(“[emelz@linkedin.com](mailto:emelz@linkedin.com)”)

email(“blah”)

# Regexes

Regular expressions are defined by strings suffixed with a .r, for example

val Decimal = """(-)?(\d+)\.?(\d\*)?""".r

The expressions in parenthesis are called capturing groups and can participate in pattern matching.

For example

def decimalMatcher(s: String) = s match {

case Decimal(sign, whole, part) =>

s"sign=$sign, whole=$whole, part=$part"

case \_ => s"$s is not a decimal"

}

List("one", "1.23", "-2.46") map decimalMatcher

# Scala Type Hierarchy

Refer to the Type Diagram in [the slides](http://goo.gl/MpJDL9).

The top of the Hierarchy is the Any Type

AnyType has two children, AnyVal and AnyRef

AnyVal contains value types like Int, Boolean etc

AnyRef is Synonymous with java.lang.Object and contains references to objects

ScalaObject is a subtype of AnyRef and contains only the Scala classes

Iterable, Seq, and List, are descendants of ScalaObject, and show part of the collections hierarchy

String is an AnyRef - it is a Java String, not a Scala String. Java strings are used in Scala, there’s no Scala String.

At the very bottom of the hierarchy is a class called Nothing. It is a subtype of all other classes in Scala.

# Type Parameters and Variance

Scala, like java has type parameters. For example, the type List[T] indicates that the List type can be further refined by specifying the type T of elements. For example, we can have a List of String by writing

val l: List[String] = List(“one”,”two”)

If you look at the ScalaDoc for the Scala APIs, you may notice funny symbols stuck to some of the type parameters. In particular, you will see + and - quite frequently. For example if you look at the ScalaDoc for [List](http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.List), you will see that List has a type parameter of [+A]. So what does the + mean?

What the + means is that the type List is *covariant* with the type parameter A. A covariant type means that as the type parameter gets narrower, the parent type gets narrower. So for example, since Apple is a subtype of fruit, a List of Apple is a subtype of fruit.

In Scala, types may be either covariant, invariant, or contravariant with their type parameters.

There is a programming principle, known as the Liskov substitution principle which states that if type T is a subtype of a type U, you can substitute a value of T wherever a type of U is required. For example, if we have

class Fruit(name: String)

case class Apple(name: String) extends Fruit(name)

def processFruit(fruits: List[Fruit]) = fruits map { \_.name }

we can call

val apples: List[Apple] = List(Apple(“mac”), Apple(“grannysmith”)

processFruits(apples)

since apples is a subtype of List[Fruit].

Now, you might think that *all* types are covariant in their type parameters. This is not the case. Consider the mutable type cell, which is a wrapper for a single value.

class Cell[+T](init: T) {

var current = init

def get = current

def set(x: T) { current = x}

}

to demonstrate the problem with making Cell covariant with T, let’s look at the following sequence

val c1 = new Cell[String](“abc”) // Create a Cell of String with initial value “abc”

val c2: Cell[Any] = c1 // Assign c1 to supertype

c2.set(42) // Set the value of c1 to an Int

val s: String = c1.get // Get c1’s value, which should be a String,

// but…. hold on!!!??

The above example illustrates the problem with making mutable types covariant. Because of this problem, mutable types in Scala must be invariant, that is, there is no + or - symbol on the type parameter, and invariant types may only be used in places which expect exactly that type.

What about an example of a contravariant type? If you look at the type of [Function1](http://www.scala-lang.org/api/current/index.html#scala.Function1) in the ScalaDoc, you will see that a function with a single input parameter is covariant in it’s return type and contravariant in it’s parameter type.

Consider the following class hierarchy:

class Publication(val title: String)

class Book(override val title: String, val isbn: String) extends Publication(title)

class ChildrensBook(override val title: String, override val isbn: String, val minAge: Int) extends Book(title, isbn)

Now suppose we have a library which has a collection of books and a display function which takes a function for formatting the book info.

object Library {

val books: Set[Book] = Set(new Book("Scala", "123"), new Book("Java", "234"))

def printBooks(info: Book => Any) {

for (book <- books) println (info(book))

}

}

Now let’s define a formatting function which takes a Book

def getIsbn(b: Book) = b.isbn

It’s fine to call printBooks with this function, since the type signature matches perfectly

Library.printBooks(getIsbn)

Now let’s define a formatting function which takes a Publication

def getTitle(p: Publication)= p.title

it is perfectly okay to call

Library.printBooks(getTitle)

since Publication is a supertype of Book, and Function1 is contravariant in it’s parameter type. The printBooks function will know about any fields that are present in a supertype of Book and will work just fine if a super of Book is passed in to it.

What if we try calling printBooks with a function that takes a subtype of Book as a parameter?

def getMinAge(c: ChildrensBook) = c.minAge

if we try to call

Library.printBooks(getMinAge)

we will get an error, since the printBooks function can’t deal with minAge, since it’s not defined on book.

# Bounds

class Food(val name: String)

class Fruit(override val name: String, val parentPlant: String) extends Food(name)

val foods: List[Food] =

List(new Food("ham"),

new Food("carrot"),

new Fruit("apple", "apple tree"))

val fruits: List[Fruit] =

List(new Food("banana", "banana tree"),

new Fruit("apple", "apple tree"))

Foo[U >: T] means that type U has a lower bound of T

case class NonFruitCart[U >: Food](foods: List[U])

val foodCart: NonFruitCart[Food] = NonFruitCart(foods) // ok

val fruitCart: NonFruitCart[Fruit] = NonFruitCart(fruits) // error

Foo[U <: T] means that type U has an upper bound of T

case class FoodCart[U <: Food](foods: List[U])

val fruitCart: FoodCart[Fruit] = FoodCart(fruits) // ok

val intCart: FoodCart[Int] = FoodCart(List(1,2)) // error

# Implicits

Let's say we have a function which takes a string

def stringLengthPrinter(s:String) {

println(f"Length of s is ${s.length}")

}

Verify that it works

stringLengthPrinter("abcd")

What happens when we pass an integer to it?

stringLengthPrinter(1234) // bad idea

We can define an implicit type conversion from Int to String

implicit def intToString(x: Int) = x.toString

Now what happens when we pass an integer to it?

stringLengthPrinter(1234) // interesting

Implicit vals can be used to specify default parameters

implicit val x = "Eric"

def greet(implicit name: String) {

println(f"Greetings, $name!");

}

We can call our function as usual

greet("John")

We can also omit parameters

greet

# Scripting

Shell scripts can be created which used scala, which can be useful for writing non-trivial scripts. Create a file called ~/Documents/projects/hello.sh with the following contents:

#!/bin/sh  
exec scala "$0" "$@"  
!#

object HelloWorld {  
 def main(args: Array[String]) {  
 println("Hello, world! " + args.toList)  
 }  
}  
HelloWorld.main(args)

cd into ~/Documents/projects and exec

chmod 755 hello.sh

./hello.sh one two

# Imports

Use \_ instead of \*

import java.io.\_

Can import several symbols from a package in one line

import java.text.{DateFormat,SimpleDateFormat}

Can rename symbols

import java.util.{Date=>MyDate}

Can have imports inside of functions

def doIt() {

import java.util.Date

println(new Date())

}

doIt()

# Packages

Packages are similar to Java packages. Create a package (need special :paste -raw mode in REPL)

:paste -raw

package linkedin

class Profile

<ctrl-d>

Access the package

new linkedin.Profile()

You can use curly braces to delineate package boundaries

:paste -raw

package foo {

class FooClass

}

package bar {

class BarClass

}

<ctrl-d>

Accessible as before

new foo.FooClass()

new bar.BarClass()

You can nest packages

:paste -raw

package linkedin {

class Profile

package test {

class ProfileTest

}

}

<ctrl-d>

Accesibility is as expected

new linkedin.Profile()

new linkedin.test.ProfileTest()