Scala Programming Language Concepts and Buzz Words

1. “val” kind of variable in Scala is equivalent of “final” kind of variable in Java.
2. “no side effect” idiom in Scala

<http://alvinalexander.com/scala/scala-idiom-methods-functions-no-side-effects>

1. “equational reasoning” : Means the time u r about to write “val” variable, u could instead write the entire “expression” assuming expression does,’t have side effects.
2. readLine() : a method which reads lines from standard inputs. In Scala a lines from standard input always results value of type “unit” i.e. (). Hence, In Scala value of type “unit” is always not equal to String type value
3. In Scala “value of type unit “ (i.e. expressed as “()”) is equal to java’s void type
4. In Scala after u used obtained Iterators of any type e.g. Iterator[String] for computing then same Iterator is not available for further computing, bcs Iterators loaded in memory was vanish once its used. In order to reuse same kind of Iterator type need be run previous expressions to get it ready in memory.
5. Difference between java.io.File.list and java.io.File.listFiles

java.io.File(“C:/a/b”).list => provides an array of a String which lists all the file names present in given path

java.io.File(“C:/a/b”).listFiles => provides an array of File objects which gets all the files (including its absolute path e.g. C:/a/b/abc.txt, C:/a/b/def(directory) ….) present in the given path.

1. Difference between toString and mkString in scala.

+ It’s a debugging method, its true purpose is to make all objects printable.

+ it is a string representation of an object, like…

val foo = scala.io.Source(“abc.txt”) its output is

foo: scala.io.BufferedSource = non-empty iterator (here foo is an object)

suppose if u apply toString on foo variable its just given a string representation of non-empty Iterator, i.e. foo.toString given res0: java.lang.String = non-empty Iterator (it doesn’t return content of a file…) OTOH, mkString() reads the iterator(that is, iterates over it) and constructs a long String based on the values read from it.

scala> foo.mkString

res6: String =

"foo

Bar

Baz

Quux

dooo"

+ The toString method is supposed to return the string representation of an object. It is often overridden to provide a meaningful representation. The mkString method is defined on collections and is a method which joins the elements of the collection with the provided string.

val a = List("a", "b", "c")

println(a.mkString(" : "))

+ the string representation of an object" sounds a little as if there where an objective way, each object is represented. In fact, there is a method ".toString ()" defined in java.lang.Object, which is used if the class or an intermediate parent did not overwrite it. In contrast, mkString is only defined in few collection classes of Scala. And they don't produce the same result, as the question points out. mkString is defined in a useful way for data: scala.io.BufferedSource = non-empty iterator, toString () not that much. –

+ They're different methods in different classes. In this case, mkString is a method in the trait GenTraversableOnce. toString is defined on Any (and is very often overridden).

1. Tab Completion :

When you're working in the Scala REPL and want to see what methods are available on a class/object, you can create an instance of an object, follow that with the "." character, and then press the [Tab] key. This process, known as "tab completion" in the REPL, gives you a preliminary list of methods that can be called on the object.

If we hit one time tab it gives preliminary methods. Then if we tried by hitting twice (tab) it gives some more methods.

1. Difference between “String” and “StringOps” class i.e. (scala.collection.immutable.StringOps) in Scala.
2. Scala’s exceptions behave just like in many other languages. Instead of returning a value in the normal way, a method can terminate by throwing an exception. The method’s caller can either catch or handle that exception, or it can itself simply terminate, in which case the exception propagates to the caller’s caller. The exception propagates in this way, unwinding the call stack, until a method handles it or there are no more methods left.
3. Difference between return a value and throw an exception by a method / function

+ Perhaps it does something like returning false if there's an error, but throwing an exception if the input is invalid.

1. An “Iterator” class offers methods called “next” and “hasNext” that allow you to iterate over the collections of elements.
2. Example for NullPointerException :

+ If we try to call a object where caller is not able to point an object like trying to call an method / definition before it defined in Scala script.

1. Example for RuntimeException:

+ def half(n: Int) = {

If (n % 2 == 0)

n/2

else

throw new RuntimeException(“”””n” shoulb be even number”)

1. FileNotFoundException:
2. IOException:
3. MalformedURLException:
4. One difference from Java that you’ll quickly notice in Scala is that unlike Java, Scala does not require you to catch checked exceptions, or declare them in a throws clause. You can declare a throws clause if you wish with the @throws annotation, but it is not required.
5. The idiomatic way to ensure a non-memory resource, such as a file, socket, or database connection is closed. First you acquire the resource. Then you start a try block in which you use the resource. Lastly, you close the resource in a finally block. This idiom is the same in Scala as in Java, however, in Scala you can alternatively employ a technique called the loan pattern to achieve the same goal more concisely.

Like…..

import java.io.FileReader

val file = new FileReader("input.txt")

try {

// Use the file

} finally {

file.close() // Be sure to close the file

}

1. Difference between java.io.FileReader and java.io.BufferedReder
2. How to edit default java.io.BufferedReader size
3. The value computed in the finally clause, if there is one, is dropped. Usually finally clauses do some kind of clean up such as closing a file; they should not normally change the value computed in the main body or a catch clause of the try.

If you’re familiar with Java, it’s worth noting that Scala’s behavior differs from Java only because Java’s tryfinally does not result in a value. As in Java, if a finally clause includes an explicit return statement, or throws an exception, that return value or exception will “overrule” any previous one

that originated in the try block or one of its catch clauses. For example, given this, rather contrived, function definition:

def f(): Int = try { return 1 } finally { return 2 }

calling f() results in 2. By contrast, given:

def g(): Int = try { 1 } finally { 2 }

calling g() results in 1. Both of these functions exhibit behavior that could surprise most programmers, thus it’s usually best to avoid returning values from finally clauses. The best way to think of finally clauses is as a way to ensure some side effect happens, such as closing an open file.

1. “switch” statements in other languages are equal to “match” expressions in scala

In “switch” The default case is specified with an underscore (\_), a wildcard symbol frequently used in Scala as a placeholder for a completely unknown value.

1. in that Scala allows you to define variables of the same name in nested scopes.
2. Difference between “function” and “method”
3. What is “first-class” function and difference between “function value” and “function literal”
4. A foreach method is defined in trait Traversable, a common supertrait of List, Set, Array, and Map. See Chapter 17 for the details.
5. Target typing : Target typing is a form of type inference that takes into account the type that’s expected. In nums.filter((x) => x > 0), for example, the Scala compiler infers type of x to be the element type of nums, because the filter method invokes the function on each element of nums.
6. Use of Placeholder as \_ :

This is why you can use this short form only if each parameter appears in the function literal at most once. Multiple underscores mean multiple parameters, not reuse of a single parameter repeatedly. The first underscore represents the first parameter, the second underscore the second parameter, the third underscore the third parameter, and so on.

1. Partially applied function:

+ A function that’s used in an expression and that misses some of its arguments. For instance, if function f has type Int => Int => Int, then f and f(1) are partially applied functions.

+ Although the previous examples substitute underscores in place of individual

parameters, you can also replace an entire parameter list with an underscore. For example, rather than writing println(\_), you could write println \_.

+ When you use an underscore in this way, you are writing a partially applied function. In Scala, when you invoke a function, passing in any needed arguments, you apply that function to the arguments. For example, given the following function:

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

sum: (a: Int,b: Int,c: Int)Int

You could apply the function sum to the arguments 1, 2, and 3 like this:

scala> sum(1, 2, 3)

res10: Int = 6

+ A partially applied function is an expression in which you don’t supply all

of the arguments needed by the function. Instead, you supply some, or none,

of the needed arguments. For example, to create a partially applied function

expression involving sum, in which you supply none of the three required arguments, you just place an underscore after “sum”. The resulting function can then be stored in a variable. Here’s an example:

scala> val a = sum \_

a: (Int, Int, Int) => Int = <function3>

1. Closed term: function literal with no free variables

Open term : Any function literal with free variable

1. Higher order functions : functions that take functions as parameters.
2. Special purpose looping methods/construct in scala : a) exists and filter in List lib
3. What is exact meaning of “feel like native language support”
4. Currying :

+ it is a functional programming technique

+

1. Difference between Statically typed and Dynamically typed language

<http://stackoverflow.com/questions/1517582/what-is-the-difference-between-statically-typed-and-dynamically-typed-languages>

<http://www.sitepoint.com/typing-versus-dynamic-typing/>

1. Difference between abstraction and encapsulation with examples
2. Referentially Transparent Methods : Methods like “replace” are called referentially transparent,which means that for any given input the method call could be replaced by

its result without affecting the program’s semantics.

Val str = “vijaykumar”

Println(“My name got changed as :” + str.replace(‘j’,’n’))

1. Predicate : it is a function who’s return type is “Boolean”

e.g. val str = “viJay”

str.exists(\_.isUpper) (here \_.isUpper is predicate function)

1. What is difference between Static type and Dynamic type languages and why do we need Unit testing for static type programming though failures are covered and resolved while compile time?