**Scala : Scalable language**

Scala is a statically typed programming language that incorporates both functional and object-oriented programming to increase scalability of applications.

It offers many types of functions for users so it is a functional language.

It is object-oriented and has a lot of functional programming features providing a lot of flexibility to the developers to code in a way they want.

A scala program contains the below:

Object − Objects have states and behaviors. An object is an instance of a class.

Class − A class can be defined as a template/blueprint that describes the behaviors/states that are related to the class.

Methods − A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.

Fields − Each object has its unique set of instance variables, which are called fields. An object's state is created by the values assigned to these fields.

Closure − A closure is a function, whose return value depends on the value of one or more variables declared outside this function.

Traits − A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Traits are used to define object types by specifying the signature of the supported methods.

A simple scala program demo:

Data types in scala: Byte, Short, Int, Long, Float, Double, Char, String, Boolean, Unit, Null, Nothing, Any, AnyRef

Mutable variable: var v1 : String = “ABC”

Immutable Variable: val v1 : String = “ABC”

Multiple assignments: val (myVar1: Int, myVar2: String) = Pair(20, "ABC")

Conditonal Statements demo:

Functions in scala:

Different types of functions:

| |
|---|
| • Functional Literals and Closures |
| • Recursion and tail-recursion |
| • Tail Calls |
| • Functional Data Structures |
| • Implicit Function Parameters |
| • Call by Name |
| • Call by Value |
| • Pure functions |
| • Pure Functions and Combinators |
| • Higher order functions |
| • Lambda (anonymous) functions |
| • Closures |
| • Currying |
| • Curried methods, partially applied functions |
| • Combining functions |
| • Function overloading |
| • Passing values by position |
| • Passing values by names |
| • Lazy Evaluation and partially applied functions |
| • Partial functions and partial function literals |
| • Lifting methods into functions |

**Functional Literals**: is also known as anonymous function:

Example of functional literals:

```
List(1,2,3,4,5).filter((x: Int)=> x > 3)
```

**Anonymous functions:**

Which has no name, and users right arrow operator to say move everything on the left to the right.

Acts as an inline function.

Anonymous function can also passed be as a parameter.

Scala provides a relatively lightweight syntax for defining anonymous functions. Anonymous functions in source code are called function literals and at run time, function literals are instantiated into objects called function values.

Scala supports first-class functions, which means functions can be expressed in function literal syntax, i.e., (x: Int) => x + 1, and that functions can be represented by objects, which are called function values.

**Closure function**:

A function whose return value depends on variable(s) declared outside it, is a closure.

A closure is a function, whose return value depends on the value of one or more variables declared outside this function.

A Closure function reference to the non-local variables of that function has access for all the variables that is outside its immediate lexical scope. So it takes the variable that was referenced earlier in the code and use that variable within the function to give the result. Even if that value is changed it will take the updated value and give the updated result. Closures are basically a combination of functions that have access to the surrounding state, this surrounding state is also called as lexical Environment. So closure gives us access of outer function from the scope of the inner one.

<u>The benefit of Using Closures Functions</u>

A major benefit of having closure function is the concept of data encapsulation plus data persistence. Since the variables defined have a scope and if they are defined inside a function they will have a local scope, but with the help of closure function, we have defined a global variable and can use it inside the function also.

**Recursion function:**

Recursion is a method which breaks the problem into smaller sub problems and calls itself for each of the problems. That is, it simply means **function calling itself**. Recursion means a function can call itself repeatedly.

**Tail Recursion:**

A tail-recursive function is just a function whose very last action is a call to itself.

**Tail function**:

The tail function is applicable to both Scala's **Mutable and Immutable collection data structures**. The tail method returns a collection consisting of all elements except the first one.

**Functional data structures:**

Data Structure facilitates the storing, organization, and retrieval of your data. It is important to note that Scala makes a clear distinction on **immutable** and **mutable** collection data types. Immutable collections reside in the **scala.collection.immutable** package, and mutable collections are organized under the **scala.collection.mutable** package, respectively.

**Primary data structures provided by Scala:** Array, Map, List, tree, set and queue.

**Demo:**

**Implicit function parameters:**

**Implicit parameters** are the parameters that are passed to a function with **implicit** keyword in Scala, which means the values will be taken from the context in which they are called. When **implicit** keyword used in the parameter scope of the function, all the parameters are marked as implicit. A method can only contain one implicit keyword. This helsps us by not writing too much of code and many things are internally handled by the compiler.

**Call by Name function: Rocket operator**

**call-by-Name function** in Scala calls the expression and recompute the passed-in expression's value every time it gets accessed inside the function. A call-by-name mechanism passes a code block to the call and each time the call accesses the parameter, the code block is executed and the value is calculated.

**Call by value function:**

**CBN and CBV are the 2 evaluation strategies we have for a function.**

**Pure functions:**

A function is called pure function if it always returns the same result for same argument values and it has no side effects like modifying an argument (or global variable) or outputting something. A pure function is **side-effect free** i.e it doesn't change the value of the variable implicitly and thus doesn't end up altering the values of the input.

**Pure function and combinator and High Order function (HOF):**

In Scala, a combinator is a Higher-Order Function and also a pure function.

As they are pure functions, we can easily compose them together very well as flexible, fine-grained building blocks for constructing larger, more complex programs.

Some of the frequently used Scala combinators are as follows:

Map, flatMap, reduce, fold, recover, filter, fallbackTo, zip

A combinator is simply a **higher order function which combines two functions into a new function**.

A function is called Higher Order Function if it contains other functions as a parameter or returns a function as an output i.e, the functions that operate with another functions are known as Higher order Functions.

**Currying/ Curried methods:**

**Currying** in Scala is simply a technique or a process of transforming a function. This function takes multiple arguments into a function that takes single argument.

Currying transforms a function that takes multiple parameters into a chain of functions, each taking a single parameter.

**Combining functions:**

**Function composition** is a way in which a function is mixed with other functions. During the composition the one function holds the reference to another function.

**Function overloading:**

Method overloading can be done by changing:

- The number of parameters in two methods.
- The data types of the parameters of methods.
- The Order of the parameters of methods.
- Return type is not considered

**Default parameter values:**

Scala lets you specify default values for function parameters. The argument for such a parameter can optionally be omitted from a function call, in which case the corresponding argument will be filled in with the default.

**Named Arguments: Demo**

**Lazy Evaluation: Demo**

Lazy evaluation or call-by-need is a evaluation strategy where an expression isn't evaluated until its first use i.e to postpone the evaluation till its demanded.

**Traits:**

A trait is like an interface with a partial implementation. In scala, trait is a collection of abstract and non-abstract methods. You can create trait that can have all abstract methods or some abstract and some non-abstract methods.

**Option:**

The **Option** in Scala is referred to a carrier of single or no element for a stated type. When a method returns a value which can even be null then Option is utilized i.e, the method defined returns an instance of an Option, in place of returning a single object or a null.