19CSE313 – PRINCIPLES OF PROGRAMMING LANGUAGES

Programming in Scala

SCALA - OVERVIEW

- Scalable Language
- Hybrid functional programming language
- Created by Martin Odersky
- Smoothly integrates the features of object-oriented and functional languages
- Used to write small scripts to building large systems
- Scala is
  - Pure object-oriented language in the sense that every value is an object
  - Scala is also a functional language and every function is a value and every value is an object so ultimately every function is an object. Supports anonymous functions, higher order functions , nested functions and currying
  - Scala is statically typed – no need to specify a type or repeat it
  - Scala runs on the JVM
  - Scala can Execute Java Code
  - Scala can do Concurrent & Synchronize processing

SCALA VS JAVA

- Scala has a set of features that completely differ from Java. Some of these are –
- All types are objects
- Type inference
- Nested Functions
- Functions are objects
- Domain specific language (DSL) support
- Traits
- Closures
- Concurrency support inspired by Erlang

FIRST SCALA PROGRAM

```
object scala1
{
 def main(args:Array[String])
 {
  println("Hello Scala!")
 }
}
```

```
D:\PPL\Scala>scalac scala1.scala
warning: 1 deprecation (since 2.13.0); re-run with -deprecation
for details
1 warning

D:\PPL\Scala>scala scala1
Hello Scala!
```

FACTORIAL FUNCTION IN SCALA

```
object factorial
{
 def main(args:Array[String])
 {
   println(factorial(30));
 }

 def factorial(x: BigInt): BigInt =
     if (x == 0) 1 else x * factorial(x - 1)
}
```

```
D:\PPL\Scala>scalac factorial.scala
warning: 1 deprecation (since 2.13.0); re-run with -deprecation
for details
1 warning

D:\PPL\Scala>scala factorial
265252859812191058636308480000000
```

- BigInt looks like a built-in type because you can use integer literals and operators such as *and - withvalues of that type.
- Yet it is just a class that happens to be defined in Scala's standard library

SCALA INTERPRETER

```
D:\PPL\Scala>scala

Welcome to Scala 2.13.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_161).

Type in expressions for evaluation. Or try :help.


scala>


scala> 1 + 2

val res0: Int = 3
```

This line includes:
- an automatically generated or user-defined name to refer to the computed value (res0, which means result 0),
- a colon (:), followed by the type of the expression (Int),
- an equals sign (=),
- the value resulting from evaluating the expression (3).

- If you wish to exit the interpreter, you can do so by entering :quit or :q.

SCALA TYPES

- The type Int names the class Int in the package scala.
- Packages in Scala are similar to packages in Java:
- They partition the global namespace and provide a mechanism for information hiding.
- Values of class Int correspond to Java's int values.
- More generally, all of Java's primitive types have corresponding classes in the scala package.
- For example,
  - scala.Boolean corresponds to Java's boolean.
  - scala.Float corresponds to Java's float.
- And when you compile your Scala code to Java bytecodes, the Scala compiler will use Java's primitive types where possible to give you the performance benefits of the primitive types.

THE RESX IDENTIFIER

- Similar to Haskell's 'it
- Example

```
scala> 1+2
val res0: Int = 3
scala> res0*3
val res1: Int = 9' environment variable
```

PRINTLN

- prints the passed string to the standard output, similar toSystem.out.println in Java.

  Example:

```
scala> println("Hello, world!")
Hello, world!
```

SOME VARIABLE DEFINITIONS

- Scala has two kinds of variables, vals and vars.

Val	Var
Similar to a final variable in Java	Similar to a non-final variable in Java
Once initialized, a val can never be reassigned.	A var can be reassigned throughout its lifetime

- Example:

```
scala> val msg = "Welcome to Scala!"
val msg: String = Welcome to Scala!

scala> msg = "Bye Bye to Haskell!"
          ^
    error: reassignment to val
```

This example illustrates *type inference*, Scala's ability to figure out types you leave off

EXPLICIT TYPE ANNOTATION

- Can both ensure the Scala compiler infers the type you intend, as well as serve as useful documentation for future readers of the code.

```
scala> val msg2: java.lang.String = "Hello again, world!"
val msg2: String = Hello again, world!

scala> val msg3: String = "Hello yet again, world!"
msg3: String = Hello yet again, world!
```

since java.lang types are visible with their simple names

VAL AND VAR (CONTD.)

```
scala> var greeting = "Hello Scala!"
var greeting: String = Hello Scala!

scala> greeting = "Bye Bye Haskell"
// mutated greeting

scala> greeting
val res3: String = Bye Bye Haskell

scala>
```

MULTILINE CODE

```
scala> val multiline
     | =
     | "This is multiline example"
val multiline: String = This is multiline example
```

- To enter something into the interpreter that spans multiple lines, just keep typing after the first line.
- If the code you typed so far is not complete, the interpreter will respond with a vertical bar on the nextline.

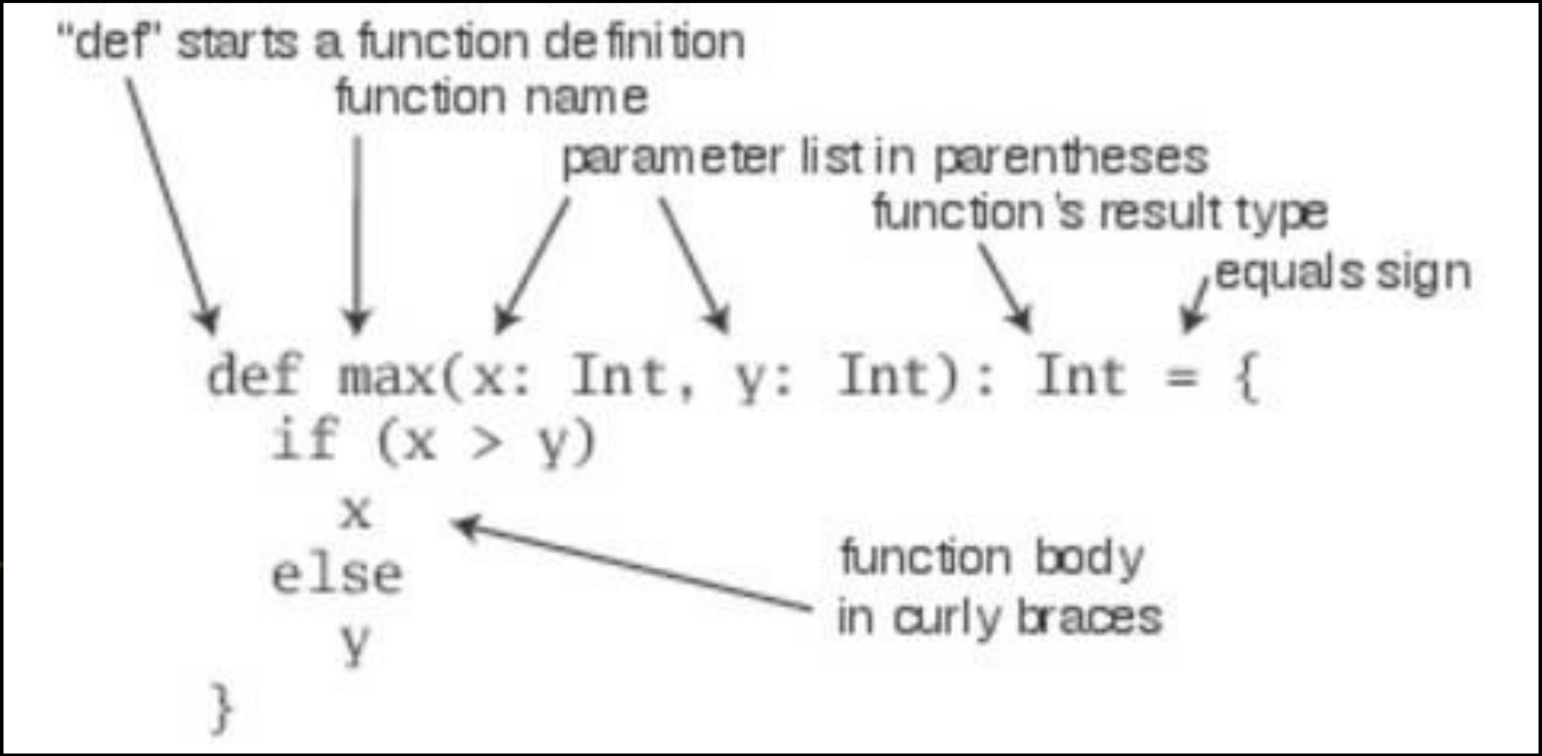
- If you realize you have typed something wrong, but the interpreter is still waiting for more input, you can escape by pressing enter twice:

```
scala> val oops =
     |
     |
You typed two blank lines.  Starting a new command.
```

SOME FUNCTION DEFINITIONS

```
scala> def max(x: Int, y: Int): Int = {
     | if (x > y) x
     | else y
     | }
def max(x: Int, y: Int): Int
```

```
scala> max(3,5)
val res4: Int = 5
```


"def" starts a function definition
function name
parameter list in parentheses
function's result type
equals sign
function body in curly braces

```
def max(x: Int, y: Int): Int = {
  if (x > y)
    x
  else
    y
}
```

OMITTING THE RESULT VALUE

- Sometimes the Scala compiler will require you to specify the result type of a function.
- If the function is *recursive*, return type must be explicitly specified
- In the case of max,however, you may leave the result type off and the compiler will infer it.
- If a function consists of just one statement, you can optionally leave off the curly braces.
- Thus, max function could alternatively be written as :

```
scala> def max(x: Int, y: Int) = if (x > y) x else y
def max(x: Int, y: Int): Int
scala> max(3,5)
val res5: Int = 5
```

A NO ARGUMENT, NO RETURN FUNCTION

```
scala> def greet() = println("Hello, world!")
def greet(): Unit
```

- A result type of Unit indicates the function returns no interesting value.
- Scala's Unit type is similar to Java's void type; in fact, every void-returning method in Java is mapped to a Unit-returning method in Scala.
- Methods with the result type of Unit, therefore, are only executed for their side effects.
- In the case of greet(), the side effect is a friendly greeting printed to the standard output.

WRITING SCALA SCRIPTS

- A script is just a sequence of statements in a file that will be executed sequentially.
- Put this into a file named hello.scala:

```
println(“This is a Hello, world, from a script!")
```

- then run:

```
> scala hello.scala
```

COMMAND LINE ARGUMENTS IN SCALA

- Command line arguments to a Scala script are available via a Scala array named args.
- In Scala, arrays are zero based, and you access an element by specifying an index in parentheses.
- So the first element in a Scala array named steps is steps(0), not steps[0], as in Java.
- To try this out, type the following into a new file named helloarg.scala:

```
// Say hello to the first argument
println("Hello, " + args(0) + "!")
```

- Now run

```
D:\PPL\Scala>scala helloarg.scala planet
Hello, planet!
```

A LOOPING EXAMPLE:

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

```
D:\PPL\Scala>scalac printargs.scala
warning: 1 deprecation (since 2.13.0); re-run with -deprecation
for details
1 warning

D:\PPL\Scala>scala printargs 1 "two" 3.5 'a'
1
two
3.5
'a'
```

- Type inference gives i the type scala.Int, because that is the type of its initial value, 0.
- args.length gives the length of the args array.
- The statement, println(args(i)), prints out the ith command line argument.
- The second statement, i += 1, increments i by one.
- **Note: Java's ++i and i++ don't work in Scala.**
- **To increment in Scala, you need to say either i = i + 1 or i += 1.**

PRINT() FUNCTION

```
var i = 0
while (i < args.length) {
if (i != 0)
print(" ")
print(args(i))
i += 1
}
println()
```

```
D:\PPL\Scala>scalac echoargs.scala
warning: 1 deprecation (since 2.13.0); re-run with -deprecation
for details
1 warning

D:\PPL\Scala>scala echoargs Scala is fun
Scala is fun
D:\PPL\Scala>
```

ITERATION WITH FOREACH FUNCTION

```
object pa
{
 def main(args:Array[String])
 {
  args.foreach(arg => println(arg));
 }
}
```

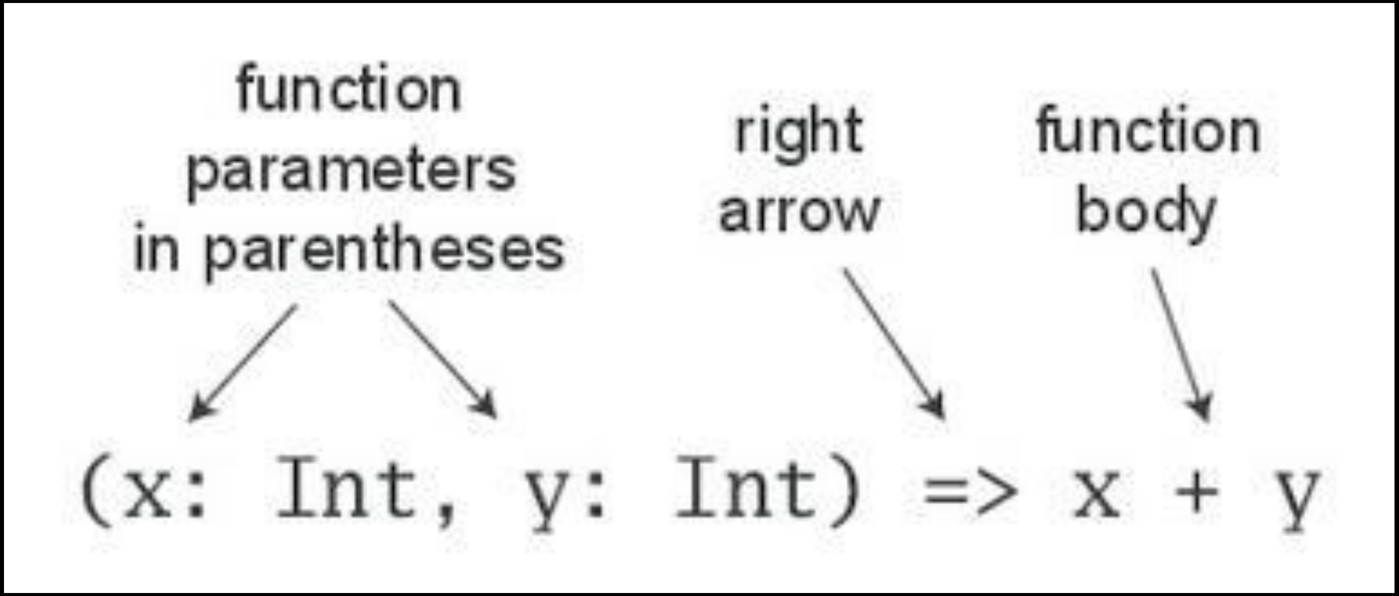
- Call the foreach method on args and pass in a function.
- In this case, a ***function literal*** *is passed* that takes one parameter named arg.
- The body of the function is println(arg).

Explicit typing (if needed)

```
args.foreach((arg: String) => println(arg))
```

```
D:\PPL\Scala>scalac pa.scala
warning: 1 deprecation (since 2.13.0); re-run with -
deprecation for details
1 warning
D:\PPL\Scala>scala pa short and sweet
short
and
sweet
D:\PPL\Scala>
```

FUNCTION LITERAL



```
(x: Int, y: Int) => x + y
```

FOREXPRESSION IN SCALA

```
object forargs
{
 def main(args:Array[String])
 {
  for (arg <- args)
   println(arg)
 }
}
```

```
D:\PPL\Scala>scalac forargs.scala
warning: 1 deprecation (since 2.13.0); re-run with -
deprecation for details
1 warning

D:\PPL\Scala>scala forargs this is a forargs example
this
is
a
forargs
example
```

- The parentheses after the "for" contain arg <- args.
- To the right of the <- symbol is the familiar args array.
- To the left of <- is "arg", the name of a val, not a var. (Because it is always a val, you just write "arg" by itself, not "val arg".)
- Although arg may seem to be a var, because it will get a new value on each iteration, it really is a val: arg can't be reassigned inside the body of the for expression.
- Instead, for each element of the args array, a *new* arg val will be created and initialized to the element value, and the body of the for will be executed.

MATCH EXPRESSIONS USING CASE

```
def main(args:Array[String])
{
 //val age1=18;
 val age1=20;
 val age="50";
 age1 match {
    case 20 => println(age1);
    case 18 => println(age1);
    case 30 => println(age1);
    case 40 => println(age1);
    case 50 => println(age1);
    case _  => println("Default");
  }
val result = age match {
        case "20" => age;
        case "18" => age;
        case "30" => age;
        case "40" => age;
        case "50" => age;
        case _  => println("Default");    }
        println("result=" + result);
     val i=7;
     i match {
      case 1 | 3 | 5 | 7 | 9 => println("odd");
      case 2 | 4 | 6 | 8 | 10 => println("even");
      }
     }
     }
```

```
D:\PPL\Scala>scala matchdemo
20
result=50
odd
```

STRING INTERPOLATION

```
object strintrp
{
 def main(args:Array[String])
 {
  val name = "john"
  val age = 21
  println(s"$name is"+ age + "years old")
  println(s"$name is $age years old")
  println(f"$name%s is $age%f years old")
  println(s"Hello \n world")
  println(raw"Hello \n world")
}
}
```

```
D:\PPL\Scala>scala strintrp
john is21years old
john is 21 years old
john is 21.000000 years old
Hello
 world
Hello \n world
```

THANK YOU