

Session 15:
SCALA BASICS 2
Assignment 1
- Prachi Mohite

Note: We will be using IDEA IntelliJ Scala Projects with scala worksheet to complete this assignment

Task 1

Create a Scala application to find the GCD of two numbers

Solution Approach

1. Create a program to get the GCD of two number by writing custom math Define a function named as gcd

- Here the function 'gcd' is recursive function
 - recursive function is a function which calls itself. That's why it's important to have some condition for stopping. Of course, that is if you don't want to create an infinite loop.
 - In this example the condition is when remainder is ZERO stop calling recursive function



```
//Task 1 - find out GCD of two numbers
//define a function named as gcd
//a - First number
//b - second number
def gcd (a:Int,b:Int):Int ={
  if(b ==0) a else {
    gcd(b, a % b)
  }
}

gcd(25,15)
gcd(12,45)
gcd(36,48)
gcd(33,121)
```

This condition stops the recursive program (pointing to the `if(b == 0) a` condition)

```
gcd: gcd[(val a: Int, val b: Int) => Int]
res0: Int = 5
res1: Int = 3
res2: Int = 12
res3: Int = 11
c: Int = 33
```

Task 2

Fibonacci series (starting from 1) written in order without any spaces in between, thus producing a sequence of digits. Write a Scala application to find the Nth digit in the sequence.

2.1 Write the function using standard for loop

```
//Task 2 - Find Nth digit in Fibonacci Series
//Print Fibonacci Series
//Assumption - we will print Fibonacci series starting from 1
//Get the Fibonacci series for nth number with for loop

def fiboNonRectursive(n: Int): Int = {
  var f = 0
  var a = 0
  var b = 0
  if (n <= 1) n
  else {
    a = 0
    b = 1

    for (i <- 2 to n) {
      f = a+b
      a = b
      b = f
    }
  }
  return f
}

fiboNonRectursive(4)
fiboNonRectursive(10)
```

```
fiboNonRectursive: fiboNonRectursive[(val n: Int) => Int]

res4: Int = 3
res5: Int = 55
```

4th number of fibonacci series is 3 (1,1,2,3,...)

10th number of fibonacci series is 55 (1,1,2,3,5,8,13,21,34,55,...)

2.2 Write the function using recursion

```
//Get the Fibonacci series for nth number with recursion

def fibowithRecursion(n : Int):Int={
  if (n <= 1) return n
  else {
    return fibowithRecursion(n-1) + fibowithRecursion(n -2)
  }
}

fibowithRecursion(4)
fibowithRecursion(10)
fibowithRecursion(5)
fibowithRecursion(1)
fibowithRecursion(2)
```

```
fibowithRecursion: fibowithRecursion[(val n: Int) => Int]

Function with recursion

res6: Int = 3
res7: Int = 55
res8: Int = 5
res9: Int = 1
res10: Int = 1
```

4th Number in sequence

10th number in sequence

5th number in sequence

1st number in sequence

2nd number in sequence

Task 3

Find square root of number using Babylonian method.

1. Start with an arbitrary positive start value x (the closer to the root, the better).
2. Initialize y = 1.
3. Do following until desired approximation is achieved.
 - a) Get the next approximation for root using average of x and y
 - b) Set y = n/x

Here we have used while to loop to get to the next approximation for root using average of x and y

While loop will continue till the difference is greater than the accuracy level we have defined.

```
// starting with any positive value
def getSquareRoot(number:Double):Double={
  var x:Double = number
  var y:Double = 1
  var e :Double = 0.00001

  while(x - y > e)
  {
    x = (x + y)/2
    y = number/x
  }
  return x
}

getSquareRoot( number = 20)
getSquareRoot( number = 36)
getSquareRoot( number = 50)
```

```
getSquareRoot: getSquareRoot[(val number: Double) => Double]

res11: Double = 4.4721402170657
res12: Double = 6.000000005333189
res13: Double = 7.071067984011346
```

Output

Below is the recursive implementation of BabyLonian method of the finding square root

```
def recursiveSquareRoot(n:Double,guess:Double,err:Double):Double={
  var newGuess:Double = 0

  if ((guess*guess) - n <= err)
    return guess
  else
  {
    //recursive case
    newGuess = (guess + (n/guess))/2
    return recursiveSquareRoot(n, newGuess,err)
  }
}

recursiveSquareRoot(20, guess = 10, err = 0.00001)
recursiveSquareRoot(36, guess = 10, err = 0.00001)
recursiveSquareRoot(50, guess = 10, err = 0.00001)
```

```
recursiveSquareRoot: recursiveSquareRoot[(val n: Double, val guess: Double, val err: Double) => Double]

res14: Double = 4.472135954999956
res15: Double = 6.000000002793968
res16: Double = 7.07106781187345
```

Recursive Call

Respective Output