Reference Guide for Stack Tracing Scala

February 28, 2022

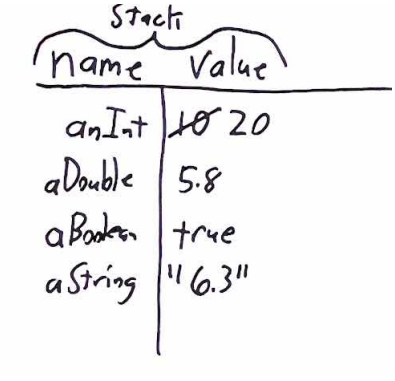
FYI

- Only a single color is required for memory diagrams, different colors are used (and order written into code) for greater clarity in this document

1 Basic variables

```
var anInt: Int = 10
var aDouble: Double = 5.8
var aBoolean: Boolean = true
var aString: String = "6.3"
anInt=20
```

- variable changes result in previous values being crossed out and new ones written in so that progression can be seen



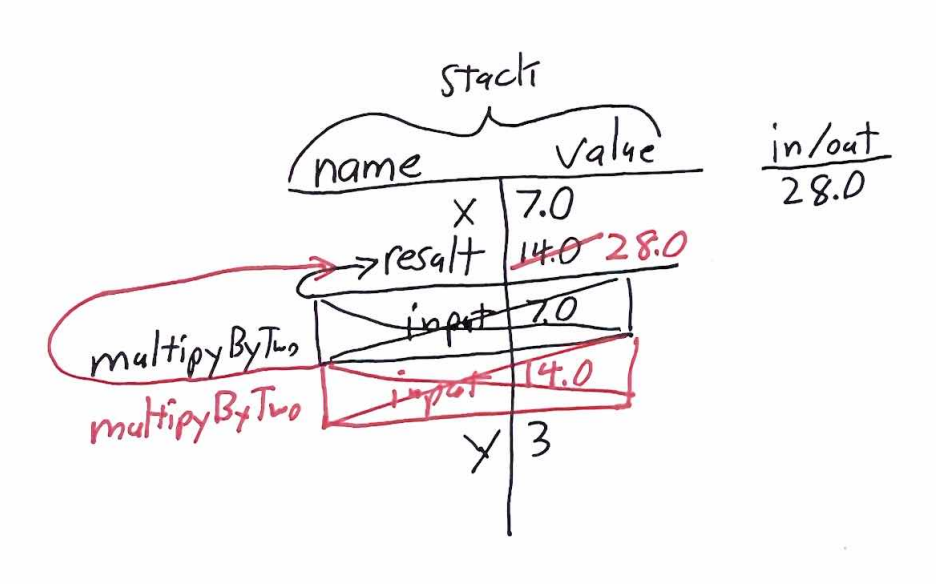
2 Function calls

2.1 Return value

```
def multipyByTwo(input: Double): Double={
  input * 2.0
}

def main(args: Array[String]): Unit = {
  var x: Double = 7.0
  var result = multipyByTwo(x)
  result = multipyByTwo(result)//done in red
  println(result)
  var y: Int = 3
}
```

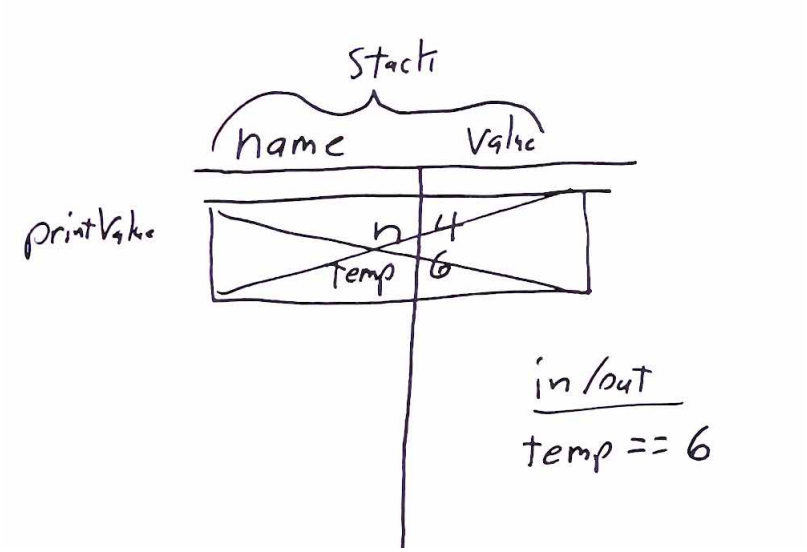
- each function call is put in its own stack frame
- variables created after the function call appear further down the stack
- in/out stands for input/output and is where any command line user input or outputs in terms of print statements appear



2.2 No (Unit) return value

```
def printValue(n: Int): Unit = {
  var temp: Int=n+2
  println("temp == " + temp)
}

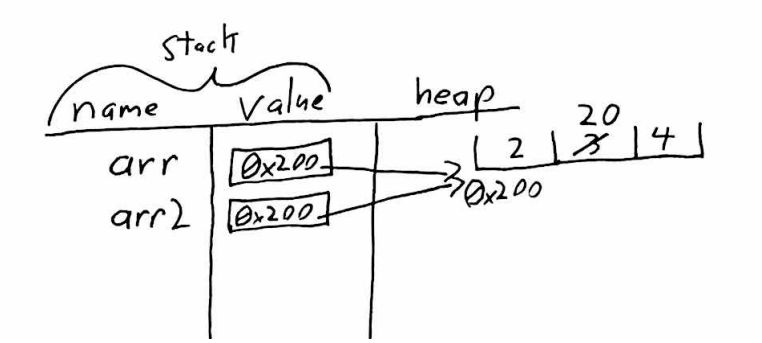
def main(args: Array[String]): Unit = {
  printValue(4)
}
```



3 Arrays

```
val arr: Array[Int] = Array(2, 3, 4)
val arr2: Array[Int] = arr
arr(1) = 20
```

- not that the assignment statement on the second line assigns the memory address and does not do a deep copy
  - this point is emphasized for newer programmers
- memory addresses all start with 0x to indicate that they are hexadecimal numbers.
  - heap addresses are typically given 3 digit numbers
  - numbers are typically written in decimal as it is easier for students to grasp at first (as they are not familiar with hexadecimal, this detail will be corrected in later courses)
  - numbers are randomly generated and just must agree on the stack and heap

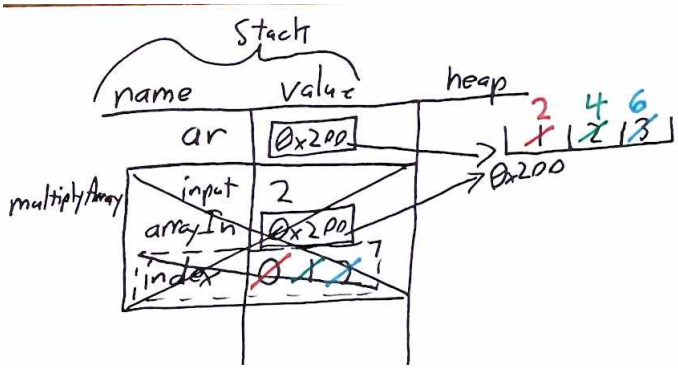


3.1 Passed to functions

```
def multiplyArray(input: Int,arrayIn: Array[Int]): Unit={
    for (index <-0 to (arrayIn.length -1)) {
      arrayIn(index)*=input //passes are red,green,blue in trace
    }
  }

  def main(args: Array[String]): Unit = {
    var ar : Array[Int]=Array(1,2,3)
    multiplyArray(2,ar)
  }
```

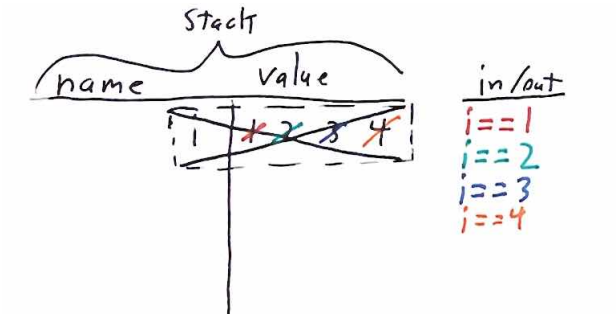
- multiplyArray is a function that utilizes sides effects
- when passing variables to functions as arguments the value on the stack associated with the variable name is the argument to the function that sets the parameter
- the dashed lines around index indicate that it is a scoped variable that only exists while the loop exists



4 For loops

4.1 basic

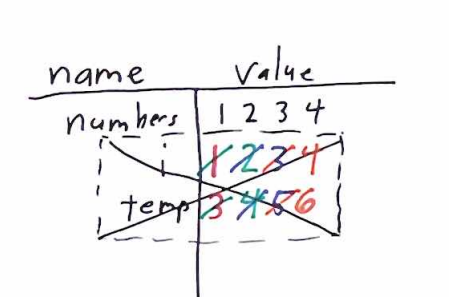
```
for(i <- 1 to 4){
  println("i == " + i)//passes are red,green,blue,orange in trace
}
```



4.2 over range

```
val numbers: Range = 1 to 4
for (i <- numbers) //passes are red,green,blue,orange in trace{
  var temp: Int=i+2
}
```

- range is a special case but we will simplify to draw it as a series of values on the stack
- scoped variables within the loop are crossed out after the loop completes



5 Lists

```
var list: List[Int] = List(2, 3, 4)

val x: Int = list.head

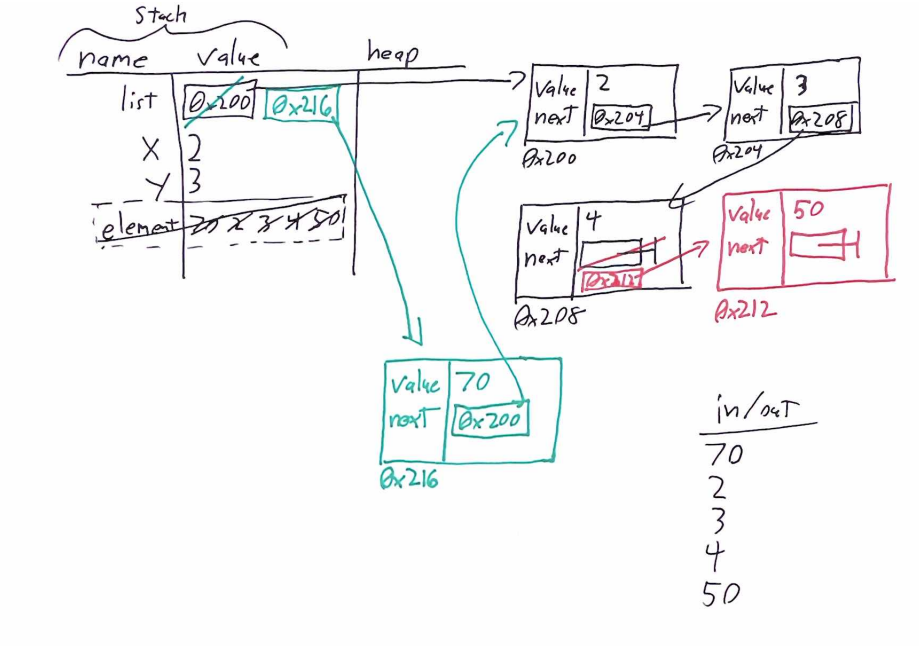
val y: Int = list.apply(1)

list = list :+ 50 //red

list = 70 :: list //green

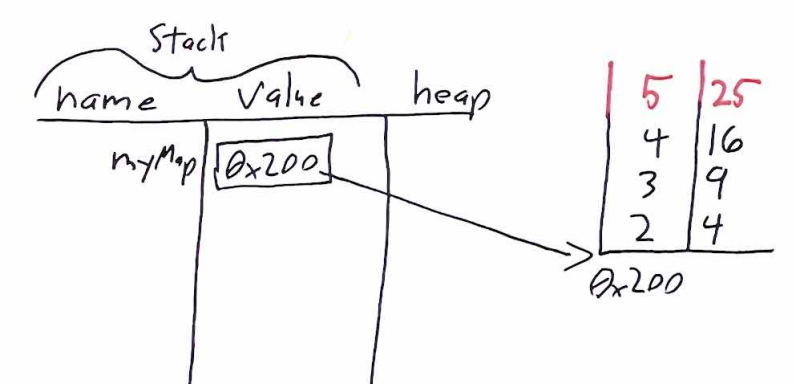
for(element <- list){
  println(element)
}
```

- input/output (in/out) can be anywhere on the page
- the positions and layout of material on the heap does not matter



6 Maps

```
var myMap: Map[Int, Int] = Map(2 -> 4, 3 -> 9, 4 -> 16)
myMap = myMap + (5 -> 25)//red
```

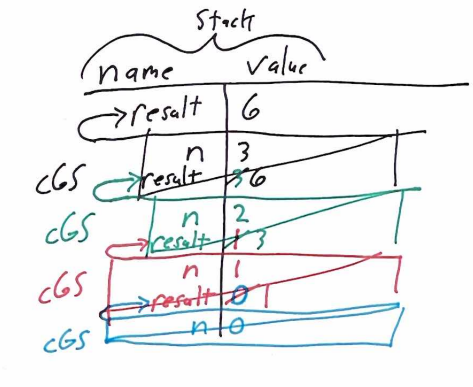


7 Recursion

```
def computeGeometricSum(n: Int): Int ={ //cGS in trace
  if(n>0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  }else{
    0
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

- each new call of cGS is performed in a new color
  - returned values are kept in the color of the method that returns it



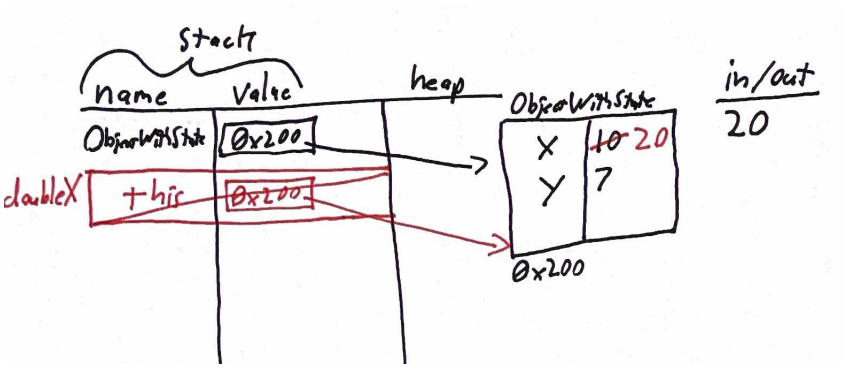
8 Objects-singleton

```
object ObjectWithState {
  // State of the object
  var x: Int = 10
  var y: Int = 7

  // Behavior of the object
  def doubleX(): Unit = {
    this.x *= 2
  }

  def main(args: Array[String]): Unit = {
    ObjectWithState.doubleX()
    println(ObjectWithState.x)
  }
}
```

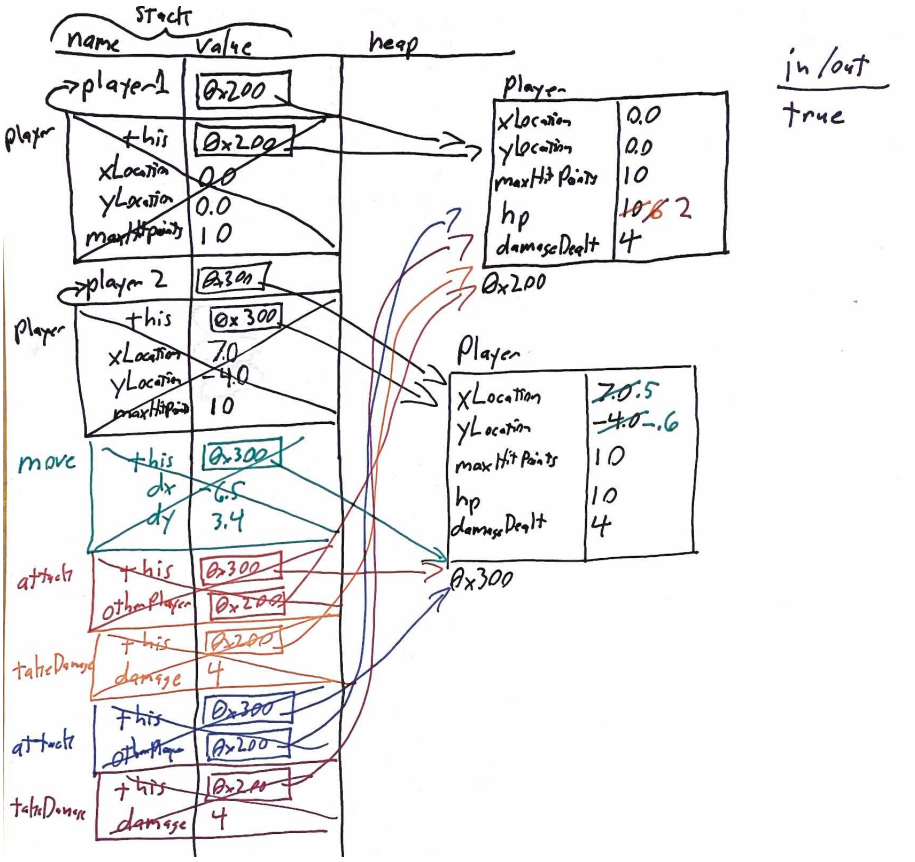
- objects that are singleton are created on the heap when first called and stack reference is made to them



9 Classes

```
class Player (var xLocation: Double, var yLocation: Double, val maxHitPoints: Int) {
  var hp: Int = this.maxHitPoints
  val damageDealt: Int = 4
  def takeDamage(damage: Int): Unit = {
    this.hp -= damage
  }
  def attack(otherPlayer: Player): Unit ={
    otherPlayer.takeDamage(this.damageDealt)
  }
  def conscious(): Boolean = {
    this.hp > 0
  }
  def move(dx: Double, dy: Double): Unit = {
    this.xLocation += dx
    this.yLocation += dy
  }
}
```

- object RunPlayer {
  def main(args: Array[String]): Unit = {
    val player1: Player = new Player(0.0, 0.0, 10)
    val player2: Player = new Player(7.0, -4.0, 10)
    player2.move(-6.5, 3.4)
    player2.attack(player1)
    player2.attack(player1)
    println(player1.hp == 2)
  }
}



10 Inheritance

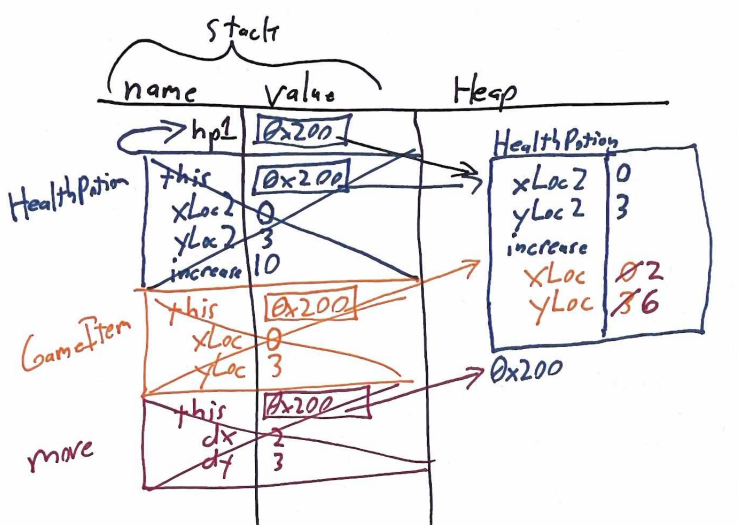
```
abstract class GameItem (var xLoc:Double, var yLoc:Double){
  def use(player: Player): Unit

  def move(dx:Double, dy:Double):Unit={
    this.xLoc+=dx
    this.yLoc+=dy
  }
}
```

- ```
  class HealthPotion (xLoc2:Double, yLoc2:Double, var increase:Int)
        extends GameItem (xLoc2, yLoc2){

    override def use(player: Player): Unit = {
      player.health+=this.increase
    }
  }

  object RunPlayer {
    def main(args: Array[String]): Unit = {
      var hp1:HealthPotion=
          new HealthPotion(0,3,10)
      hp1.move(2,3)
    }
  }
  ```


  ```