**Lab Book**

**Ex 2-1**



**Consumer.groovy**

// insert a modified println statement

*println* "the result was: ${i}"

**Multiplier.groovy**

// write i \* factor to outChannel

outChannel.write(i \* factor)

// read in the next value of i

i = inChannel.read()

**RunMultiplier.groovy**

//insert here an instance of multiplier with a multiplication factor of 4

**new** Multiplier (inChannel: connect1.in(),

outChannel: connect2.out(),

factor: 4)

**Output**

next: 1

next: the result was: 4

2

next: the result was: 8

4

next: the result was: 16

3

next: the result was: 12

0

Finished

**Ex 2-2C:\Users\Beej\AppData\Local\Microsoft\Windows\INetCacheContent.Word\ex2.2_diagram.png**

**GenerateSetsOfThree.groovy**

//write the terminating List as per exercise definition

outChannel.write([-1,-1,-1])

**ListToStream.groovy**

// hint: output list elements as single integers

**for** ( i **in** 0 ..< inList.size)outChannel.write(inList[i])

inList = inChannel.read()

**CreateSetsOfEight.groovy**

// put v into outList and read next input

outList[i] = v

v = inChannel.read()

**Output**

Eight Object is [1, 2, 3, 4, 5, 6, 7, 8]

Eight Object is [9, 10, 11, 12, 13, 14, 15, 16]

Eight Object is [17, 18, 19, 20, 21, 22, 23, 24]

Finished

**Questions**

*What change is required to output objects containing six integers?*

**for** ( i **in** 0 .. 7 ) becomes **for** ( i **in** 0 .. 5 ) in CreateSetsOfEight.groovy

*How could you parameterise this in the system to output objects that contain any number of integers (e.g. 2, 4, 8, 12) ?*

Have the number the for loop be a variable that is read in from the console and can be decided by the user by writing it in the console.

*What happens if the number of integers required in the output stream is not a factor of the total number of integers in the input stream (e.g. 5 or 7)*

Numbers are left out and the process does not terminate because ListToStream.groovy can’t finish its for loop, and doesn’t send ‘-1’ to CreateSetsofEight.groovy

**Ex 3-1**

**Process Network Diagram for Differentiate using Minus**



**Minus.groovy**

// output one value subtracted from the other

// be certain you know which way round you are doing the subtraction!!

outChannel.write(read0.value - read1.value)

**Differentiate.groovy**

// insert a constructor for Minus

**new** Minus ( inChannel0: a.in(),

inChannel1: c.in(),

outChannel: outChannel)

**Output**

Differentiated Numbers

0

1

2

3

4

5

6

7

8

9

10

11

12

13

**Process Network Diagram for Differentiate using Negator**



**Negator.groovy**

//output the negative of the input value

outChannel.write(-(inChannel.read()))

**Differentiate.groovy**

//insert a constructor for Negator

**new** Negator ( inChannel: c.in(),

outChannel: d.out() )

**Output**

Differentiated Numbers

0

1

2

3

4

5

6

7

8

9

10

11

**Questions**

*Which is the more pleasing solution? Why?*

I prefer the Minus solution. It seems to be simpler, especially when considering the network process diagram. It directly undoes the integrate step as Minus.groovy is the oppositve to GPlus.groovy. The negator adds another process before the GPlus that I feel overcomplicates it.

**Ex 3-2**



**GSCopy.groovy**

// output the input value in sequence to each output channel

outChannel0.write(i)

outChannel1.write(i)

**GSquares.groovy**

// you will need to modify this twice

**new** GSPairsA ( inChannel: I2P.in(),

outChannel: outChannel ),

*and*

// you will need to modify this twice

**new** GSPairsB ( inChannel: I2P.in(),

outChannel: outChannel ),

**Output (with GSPairsA)**

Squares

**Output (with GSPairsB)**

Squares

1

4

9

16

25

36

49

64

81

100

121

144

169

196

**Questions**

*Determine the effect of the change. Why does this happen?*

When using GSPairsA the process halts. This is because GSCopy sequentially writes the value to a.out() then b.out() so GPlus receives the value via a.in() but doesn’t get supplied a value from c.in() yet (as GTail does not write the first value it receives to c.out()). The 2nd value from the inChannel then cannot get sent by GSCopy via the a channel to GPlus as GPlus has not run. Therefore GSCopy is unable to send a 2nd value to GTail and the process halts.

When using GSPairsB, the value is sent to GTail first so GPlus will receive the 1st value via the a channel and then the 2nd value via the c channel from the GTail and run successfully. Therefore the process executes normally, providing the correct output.

**Ex 3-3**

**Questions**

*Why was it considered easier to build* ***GParPrint*** *as a new process rather than using multiple instances of* ***GPrint*** *to output the table of results?*

Building a table using multiple instances of GPrint would require complex routing of the inChannels and complicated formatting in the println statements so that an accurate table could be formed. GParPrint, however, reduces this complexity by taking multiple inChannels at once and printing each piece of information in line such that a table can easily be formed.

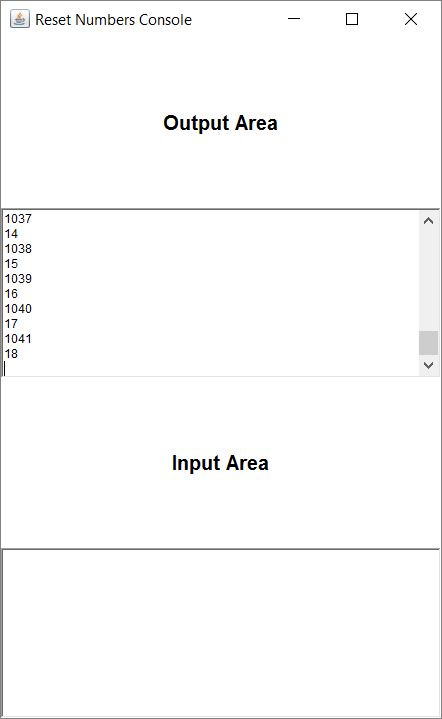
**Ex 4-1**

**Questions**

*What happens if line {25} of* ***ResetPrefix*** *Listing 4-1 is commented out? Why?*

As shown in the output image below when a reset value is inputted the output numbers oscillate between the initial sequence of numbers and the sequence starting from the reset value. If a second reset value is inputted then the process deadlocks. This is because the old values are still in the system (as the read line is commented out), when a third number is placed into the system there is no available process to accept the new number. By adding buffers more numbers could be placed into the system but you would need infinite buffers to allow the process to never deadlock, but this is impossible.

**Output**



**Ex 4-2**

**ResetNumbers.groovy**



// requires a constructor for ResetSuccessor

**new** ResetSuccessor( inChannel: b.in(),

outChannel: c.out(),

resetChannel: resetChannel )

**ResetSuccessor.groovy**

// deal with inputs from resetChannel and inChannel

// use a priSelect

**def** index = alt.priSelect()

**if** (index == 0 ) { // resetChannel input

**def** resetValue = resetChannel.read()

inChannel.read()

outChannel.write(resetValue)

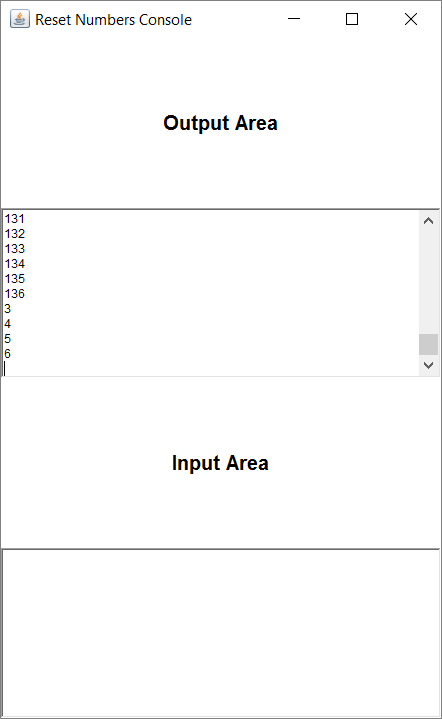
}

**else** { //inChannel input

outChannel.write(inChannel.read() + 1)

}

**Output**



**Questions**

*Does it overcome the problem identified in Exercise 1? If not, why not?*

This different formulation of ResetNumbers does not fix the problem identified in Exercise 1 for the same reasons as before.

**Ex 5-1**

**RunScaler.groovy**

**new** QProducer ( put: QP2Q.out(),

iterations: 50,

delay: 100 ),

**new** QConsumer ( get: QC2Q.out(),

receive: Q2QC.in(),

delay: 100 )

**Output**

QConsumer has started

QProducer has started

QConsumer has read 1

QConsumer has read 2

QConsumer has read 3

QConsumer has read 4

QConsumer has read 5

QConsumer has read 6

.

.

.

QConsumer has read 47

QConsumer has read 48

QConsumer has read 49

QConsumer has read 50

Q finished

QConsumer has read null

**Questions**

*By varying the delay times demonstrate that the system works in the manner expected. What do you conclude from these experiments?*

Changing the delay times did not affect operation from completeting correctly, only how long it took to finish. Increasing the delay values for either QProducer, QConsumer or for both increased the time for each line of output to print.

**Ex 5-2**



**Scale.groovy**

**while** (**true**) {

**switch** ( scaleAlt.priSelect(preCon) ) {

**case** SUSPEND :

// deal with suspend input

suspend.read()

factor.write(scaling)

suspended = **true**

println "Suspended"

preCon[INJECT] = **true**

preCon[SUSPEND] = **false**

**break**

**case** INJECT:

// deal with inject input

scaling = injector.read()

println "Injected scaling is $scaling"

suspended = **false**

timeout = timer.read() + DOUBLE\_INTERVAL

timer.setAlarm(timeout)

preCon[SUSPEND] = **true**

preCon[INJECT] = **false**

**break**

**case** TIMER:

// deal with Timer input

timeout = timer.read() + DOUBLE\_INTERVAL

timer.setAlarm ( timeout )

scaling = scaling \* multiplier

println "Normal Timer: new scaling is ${scaling}"

**break**

**case** INPUT:

// deal with Input channel

**def** inValue = inChannel.read()

**def** result = **new** ScaledData()

result.original = inValue

**if** (preCon[SUSPEND] == **true**) {

result.scaled = inValue \* scaling

}

**else** {

result.scaled = inValue

}

outChannel.write ( result )

**break**

} //end-switch

} //end-while

**Output**

Original Scaled

0 0

1 2

2 4

3 6

Normal Timer: new scaling is 4

4 16

5 20

Suspended

6 6

Injected scaling is 5

7 35

8 40

9 45

10 50

11 55

Normal Timer: new scaling is 10

12 120

Suspended

13 13

Injected scaling is 11

14 154

15 165

**Questions**

*Which is the more elegant formulation? Why?*

The scaling device that uses pre-conditions is more elegant than the solution using nested alternatives, this is because nested alternatives has multiple switch statements making it hard to follow, whereas there is only one switch when using pre-conditions and therefore improves readability.