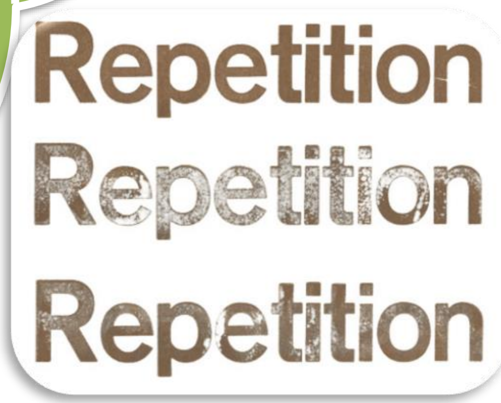
SIT105 - Critical Thinking and Problem Solving for IT
Class 08

Part 1 – Repetition Statements
Part 2 – Modularisation

A/Prof. Xiao Liu
xiao.liu@deakin.edu.au

Part 1 Content

1. Selection Concept – Reminder
2. Repetition Concept
3. DOWHILE Statement
4. REPEAT Statement
5. DO Statement

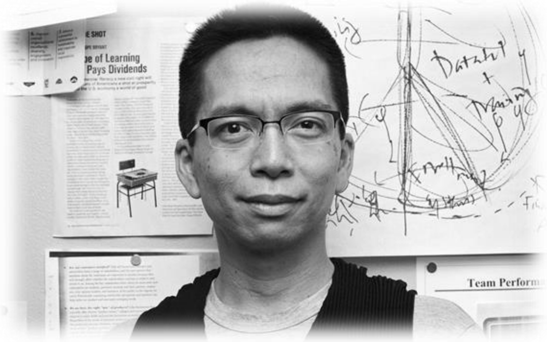


QUOTE

*There is a construct in computer programming called **'the infinite loop'** which enables a computer to do what no other physical machine can do: <u>to operate in perpetuity without tiring.</u>*

*In the same way it **doesn't know exhaustion**, it **doesn't know when it's wrong** and it can keep doing the wrong thing over and over without tiring.*

Quote by: John Maeda
Japanese-American graphic designer, computer scientist, academic, and author.

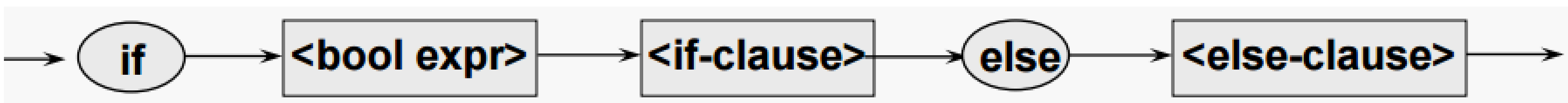


SELECTION CONCEPT – REMINDER

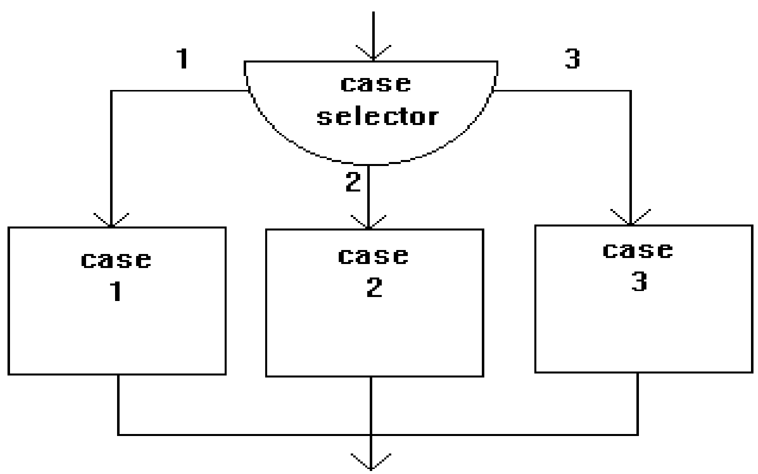
Selection statements are used to select and evaluate one embedded group of statements
and so ignore other embedded groups.

2 Possible Selection Choices

IF Statement



Case Statement



REPETITION CONCEPT

A repetition statement is used to **continually evaluate one group of statements**

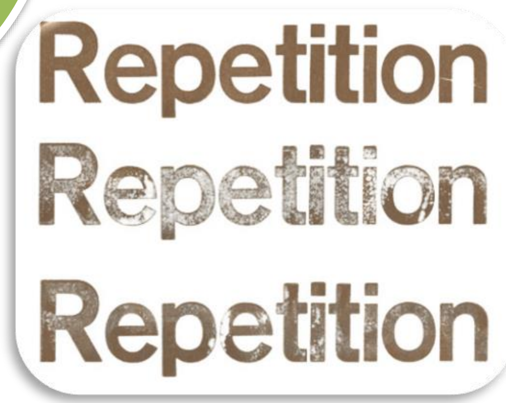
The value of an **explicit Boolean expression** is used to either:

- **Terminate** the continual evaluation (stop the loop)
- **Evaluate** the group one more time

"Something that is explicit is expressed or shown clearly and openly, without any attempt to hide anything"

Let’s take a look at the following repetition statements!

1. DOWHILE Statement
2. REPEAT Statement
3. DO Statement



DOWHILE STATEMENT

Format

DOWHILE Boolean expression

statements

ENDDO

The blue block is executed first.

The green block is executed after the blue block, but only when the blue block has a **TRUE** value.

After executing the last statement in the green block, we restart back at the blue block.

DOWHILE STATEMENT - EXAMPLE 1

```
age = 21
IF age >= 18 THEN
    PRINT "voter"
ENDIF

DOWHILE age >= 18
    PRINT age
    age = age - 1
ENDDO
PRINT "Age is ", age
```

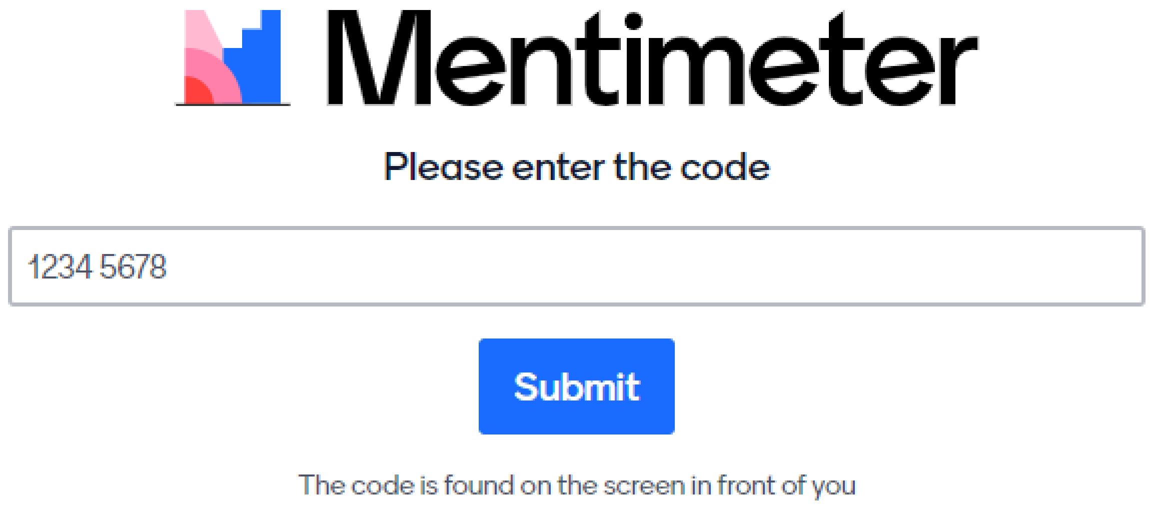
	age	condition	output
assignment	21		
IF ...	21	true	
PRINT...	21		voter
DOWHILE ...	21	true	
PRINT...	21		21
Decrement	20		
DOWHILE ...	20	true	
PRINT...	20		20
Decrement	19		
DOWHILE ...	19	true	
PRINT...	19		19
Decrement	18		
DOWHILE ...	18	true	
PRINT...	18		18
Decrement	17		
DOWHILE ...	17	false	
PRINT...	17		Age is 17

DOWHILE Statement - Example 2a

```
age = 13
DOWHILE age >= 13 AND age <= 19
    PRINT age
    age = age + 1
ENDDO
PRINT “end”
```

Let’s go to www.menti.com

- 3mins



DOWHILE Statement - Example 2a

```
age = 13
DOWHILE age >= 13 AND age <= 19
    PRINT age
    age = age + 1
ENDDO
PRINT “end”
```

age	output
~~13~~	13
~~14~~	14
~~15~~	15
~~16~~	16
~~17~~	17
~~18~~	18
~~19~~	19
20	end

DOWHILE STATEMENT - EXAMPLE 2B

```
// using constants
// for maintenance
LOWER_LIMIT = 13
UPPER_LIMIT = 19

age = LOWER_LIMIT
DOWHILE age >= LOWER_LIMIT AND age <= UPPER_LIMIT
    PRINT age
    age = age + 1
ENDDO
```

DOWHILE STATEMENT - EXAMPLE 3A

```
LOWER_LIMIT = 13, UPPER_LIMIT = 19, MINIMUM_AGE = 0
PRINT "Please type an age value: "
READ age
DOWHILE age >= MINIMUM_AGE // don't do the loop if its negative
    IF age >= LOWER_LIMIT AND age <= UPPER_LIMIT THEN
        PRINT age, " is a teenager value"
    ENDIF
    PRINT "Please type an age value: "
    READ age // get the next age value
ENDDO
```

DOWHILE STATEMENT - EXAMPLE 3B

```
LOWER_LIMIT = 13, UPPER_LIMIT = 19, MINIMUM_AGE = 0
age = MINIMUM_AGE        // ensure that the loop runs
DOWHILE age >= MINIMUM_AGE // repetition statement
    PRINT “Please type an age value: ”
    READ age // read is done inside the DOWHILE only
    IF age >= LOWER_LIMIT AND age <= UPPER_LIMIT THEN
        PRINT age, “ is a teenager value”
    ENDIF
ENDDO
```

REPEAT STATEMENT

- **Format**

REPEAT

statement

UNTIL Boolean expression

The green block is executed first.

The blue block is executed after the green block.

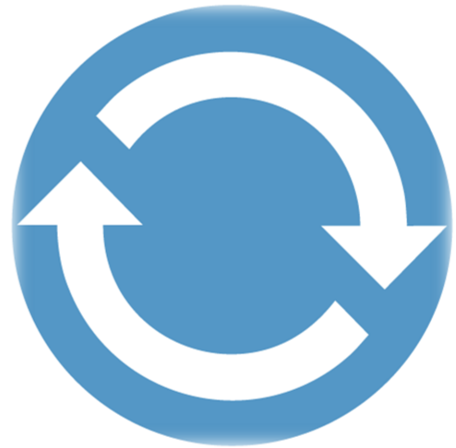
If the blue block has a **False** value, we restart back at the green block.

In this, the statement will always be executed at least once As where WHILEDO the statements may never execute.

REPEAT STATEMENT - EXAMPLE 1

```
VOTING_AGE = 18, MINIMUM_AGE = 0
age = 21
IF age >= VOTING_AGE THEN
        PRINT “voter”
ENDIF

REPEAT
        PRINT age
        age = age - 1
UNTIL age < MINIMUM_AGE  // keep doing statements until true
PRINT “Age is ”, age     // only do this once condition is true
```

What is this doing?

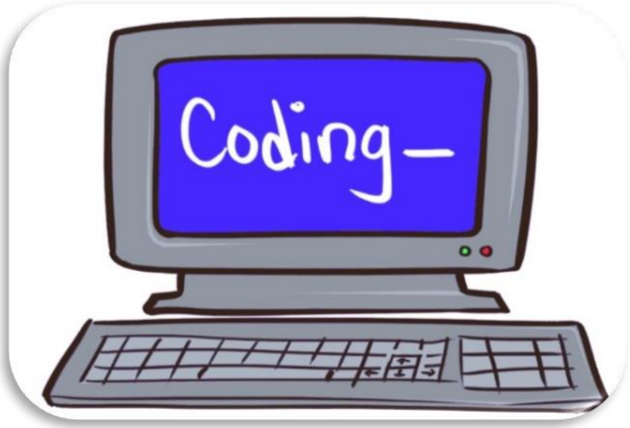


REPEAT STATEMENT - EXAMPLE 2

```
LOWER_LIMIT = 13, UPPER_LIMIT = 19

age = LOWER_LIMIT
REPEAT
        PRINT age
        age = age + 1
UNTIL age < LOWER_LIMIT OR age > UPPER_LIMIT
```

HEADS UP – “REPEAT” STATEMENT IN C

```
do
        statement
while ( expression );
```



1. Evaluate the statement
2. Evaluate the expression
3. On a true value, go back to 1
   A **false** value causes loop termination

DO STATEMENT

Value TO Value = range

The blue block is executed first.
At first the initialValue is assigned to the variable.

- <u>Format 1</u>

DO variable = initialValue TO finalValue

statement

ENDDO

The green block is executed after the blue block.

After executing the green block;
if the variable's value is less than the finalValue,
add 1 to the variable and re-execute the green block.

That 'variable' gets the 'intialValue' first. Then the next number in the range gets assigned to it and so on.

We know how many times we want to repeat.

DO STATEMENT - EXAMPLE 1A

// Display all teenage values

```
DO age = 13 TO 19
    PRINT age
ENDDO
```

	age	condition	output
DO initailise	13		
PRINT…	13		13
DO step	14		
DO condition	14	True (as 14<=19)	
PRINT…	14		14
DO step	15		
DO condition	15	True (as 15<=19)	
PRINT…	15		15
DO step	16		
DO condition	16	True (as 16<=19)	
ETC			
ETC			
PRINT…	19		19
DO step	20		
DO condition	20	False (as 20>19)	

DO STATEMENT - EXAMPLE 1B

```
// Display all teenage values
// Same thing just using our CONSTANTS
INITIAL_VALUE = 13
FINAL_VALUE = 19
DO age = INITIAL_VALUE TO FINAL_VALUE
      PRINT age
ENDDO
```

DO STATEMENT

The blue block is executed first.
At first the initialValue is assigned to the variable.

Format 2

```
DO variable = initial TO final STEP stepValue
    statement
ENDDO
```

The green block is executed after the blue block.

After executing the green block;
if the variable's value is less than the finalValue,
add stepValue to the variable and re-execute the green block.

In this statement we have 'STEP' which allows us to adjust the value we step up by.
Default is 1.

DO STATEMENT - EXAMPLE 2A

// Display 100, 105, 110, 115, 120 and 125

```
DO x = 100  TO  125  STEP  5
      PRINT  x
ENDDO
```

DO STATEMENT - EXAMPLE 2B

```
// Display 100, 105, 110, 115, 120 and 125
// Same thing just using CONSTANTS
INITIAL_VALUE = 100
FINAL_VALUE = 125
STEP_VALUE = 5
DO x = INITIAL_VALUE TO FINAL_VALUE STEP  STEP_VALUE
        PRINT  x
ENDDO
```

DO STATEMENT

Format 3

The blue block is executed first.
At first x is assigned to the variable.

```
DO variable = x DOWNTO y STEP z
      statement
ENDDO
```

The green block is executed after the blue block.

Stepping down instead of up.

Start at a higher value then count down.

After executing the green block;
if the variable's value is greater than y,
subtract z from the variable and re-execute the green block.

DO STATEMENT - EXAMPLE 3

```
// Display 20, 18, 16, 14, 12 and 10
 DO x = 20  DOWNTO  10  STEP  2
      PRINT  x
 ENDDO
```

DO STATEMENT - EXAMPLE 4 (VERSION 1)

```
// Determine and display the
// 2 times table
        PRINT “2 * 1 = “, 2 * 1
        PRINT “2 * 2 = “, 2 * 2
        PRINT “2 * 3 = “, 2 * 3
        PRINT “2 * 4 = “, 2 * 4
        …
        PRINT “2 * 12 = “, 2 * 12
```

Expected Output

```
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20
2 * 11 = 22
2 * 12 = 24
```

DO STATEMENT - EXAMPLE 4 (VERSION 2)

```
// Determine and display the 2 times table
timesTable = 2
PRINT “2 * 1 = “, timesTable * 1
PRINT “2 * 2 = “, timesTable * 2
PRINT “2 * 3 = “, timesTable * 3
PRINT “2 * 4 = “, timesTable * 4
…
PRINT “2 * 12 = “, timesTable * 12
```

DO STATEMENT - EXAMPLE 4 (VERSION 3)

```
// Determine and display the 2 times table
timesTable = 2
DO n = 1 TO 12
      product = timesTable * n
      PRINT timesTable, “ * “, n, “ = “, product
ENDDO
```

DO STATEMENT - EXAMPLE 5

```
// Determine and display the 2 and 3 times tables
  timesTable = 2
  DO n = 1 TO 12
        product = timesTable * n
        PRINT timesTable, " * ", n, " = ", product
  ENDDO

  timesTable = 3
  DO n = 1 TO 12
        product = timesTable * n
        PRINT timesTable, " * ", n, " = ", product
  ENDDO
```

Just alter this to do the 3x times table

HEADS UP – "DO" STATEMENT IN C

Do statement is a for statement in C

C

```
for(expression1; expr2; expr3)
        statement
```

Pseudocode

```
expression1
DOWHILE expr2
        statement
        expr3
ENDDO
```

C example

```
sum = 0;
for(n = 1; n <= 10; n = n + 1)
        sum = sum + n;
```

Pseudocode example

```
sum = 0
n = 1
DOWHILE n <= 10
        sum = sum + n
        n = n + 1
ENDDO
```

Part 2 Content

1. Top Down Method
2. What is a Module
3. Modular Programming
4. Good Features of a Module
5. Benefits of Good Module Design
6. The Main Module
7. Module Execution
8. Hierarchy Charts

Top Down Method (Technique)

A <u>*top down method*</u> is to design a **solution**.

- Divide a single task into several smaller subtasks
- Divide these subtasks into smaller subtasks
- Continually divide subtasks until they:
  - represent ***one*** logical / valid task
  - are at a ***manageable size***
- <u>Each task / subtask is considered to be a module</u>

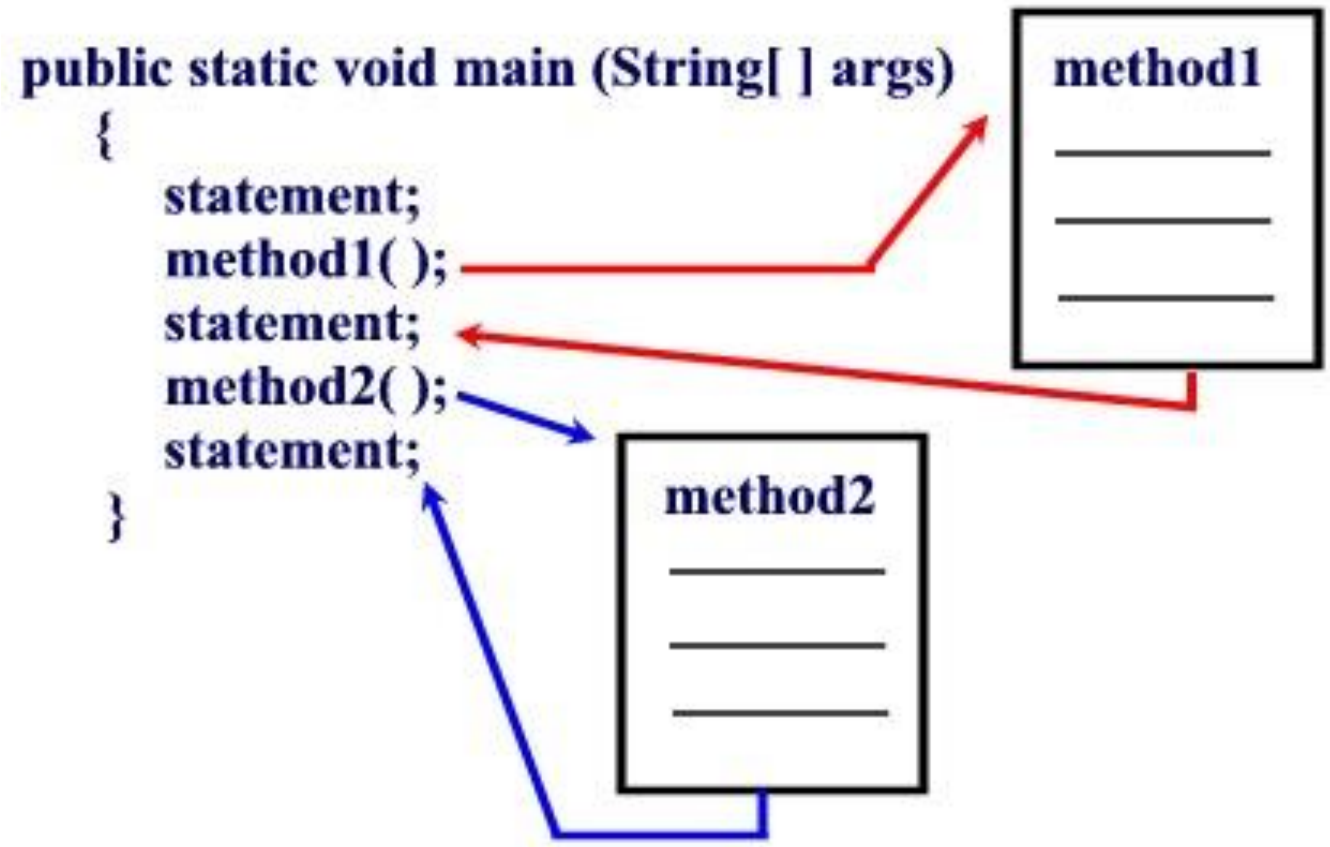
What is a Module

- Modular programming is the process of subdividing a computer program.
  - **So:** A ***module*** is a separate software component.
- A <u>module</u> can often be used in a variety of programs.
- Allows for boundaries and improves maintainability.
- It enables multiple programmers to divide up the work and debug pieces of the program independently.
- Teams can develop modules separately and do not require knowledge of all modules in the system.

Modular Programming?

- must perform its task
- must not do anything unrelated to its task
- must always start the code at its top
- should only have an exit at its end
- must have a meaningful name to describe its task
- must have appropriate:
  - cohesion (see chapter 10) //next class
    - Things that belong together should be kept together.
  - coupling (see chapter 10) //next class
    - One object doesn't directly change or modify the state or behavior of another object.

Module Example



Benefits of Good Module Design

Improves Understanding!

- Each module should be based on performing one logical task.
- E.g. login module has nothing to do with the print module.

Improves Reusability!

- A module might be used several times in a program.
  but also within other programs.
- E.g. We have been reusing the ‘PRINT’ module a lot so far in our algorithms.

Benefits of Good Module Design

Removes Redundancy

- Use one module instead of duplicating the same code several times in a program.
- E.g. just use the PRINT module and not create multiple to do the same thing.

Improves Maintenance

- Each module has little or no dependency on other modules.
- E.g. If you buy a new TV remote control, do you really want to change the TV also? You want them to be independent…. ideally.

The Main Module

```
// Main Function
void main(void)
{
    TRISB0 = 0;                  // Make RB0 pin output
    RB0    = 0;                  // Make RB0 low
```

- Where the program starts!
- Name of the first function in most coding languages is called Main()
- An algorithm shall have one main module (its where the algorithm starts).
- The main module controls other modules.
- <u>Execution of the algorithm:</u>
  - always starts at the **first line** of the main module
  - **calls** other modules and executes code statements
  - finishes at **the end** (last line of the program)

The Main Module

```
// Main Function
void main(void)
{
    TRISB0 = 0;                 // Make RB0 pin output
    RB0    = 0;                 // Make RB0 low
```

- The name of the **main** keyword highlights its the main task of the program.
- If the names of the ***other modules*** are sensible, you should be able to:
  - **read** the pseudocode of the main module, and
  - **understand** what the program does.

Example

The Main Module (example)

Loop through each record of the file.

Inside the loop we work out the bill values.

Write a patients bill and update some totals.

Close file and print.

By examining this main module one can understand its overall task.

```
CREATE_PATIENT_BILL
        INITIALISE_BILL_VARIABLES
        OPEN patient_file
        DOWHILE (READ patient_record SUCCEEDS)
                COMPUTE_ACCOMODATION_BILL
                COMPUTE_SURGERY_BILL
                COMPUTE_PATHOLOGY_BILL
                WRITE_PATIENT_BILL
                UPDATE_OVERALL_TOTALS
        ENDDO
        CLOSE patient_file
        PRINT total_patients, overall_totals
END
```

Module Execution

When executing statements in **module A** and if we come across the name of a **module B**:

1. **suspend** execution of module **A**
2. **start executing** statements of module **B**

When module **B** finishes:

1. **terminate** module **B**
2. **resume** execution of module **A**

Module Execution (example)

```
CREATE_PATIENT_BILL
    INITIALISE_BILL_VARIABLES
    OPEN patient_file
    WHILE (READ patient_record SUCCEEDS)
        COMPUTE_ACCOMODATION_BILL
        COMPUTE_SURGERY_BILL
        COMPUTE_PATHOLOGY_BILL
        WRITE_PATIENT_BILL
        UPDATE_OVERALL_TOTALS
    END
    CLOSE patient_file
    PRINT total_patients, overall_totals
END
```

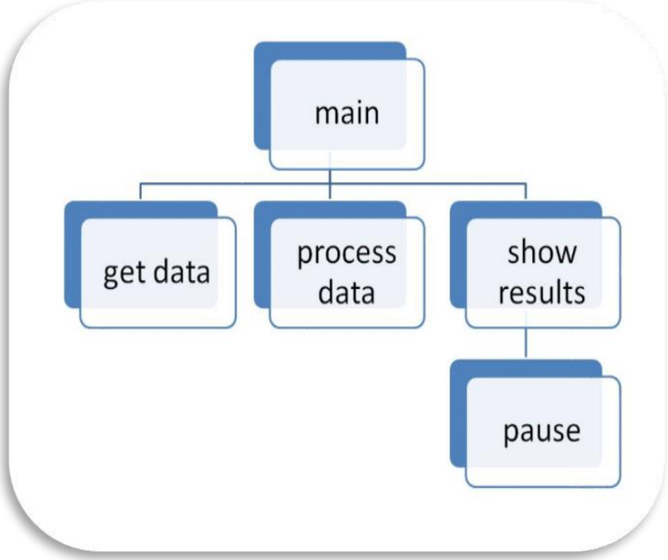
```
INITIALISE_BILL_VARIABLES
    accomodation_bill = 0
    surgery_bill = 0
    pathology_bill = 0

    total_accomodation_bill = 0
    total_surgery_bill = 0
    total_pathology_bill = 0
    overall_totals = 0
END
```

… (other modules)

Hierarchy Charts (Structure Charts)

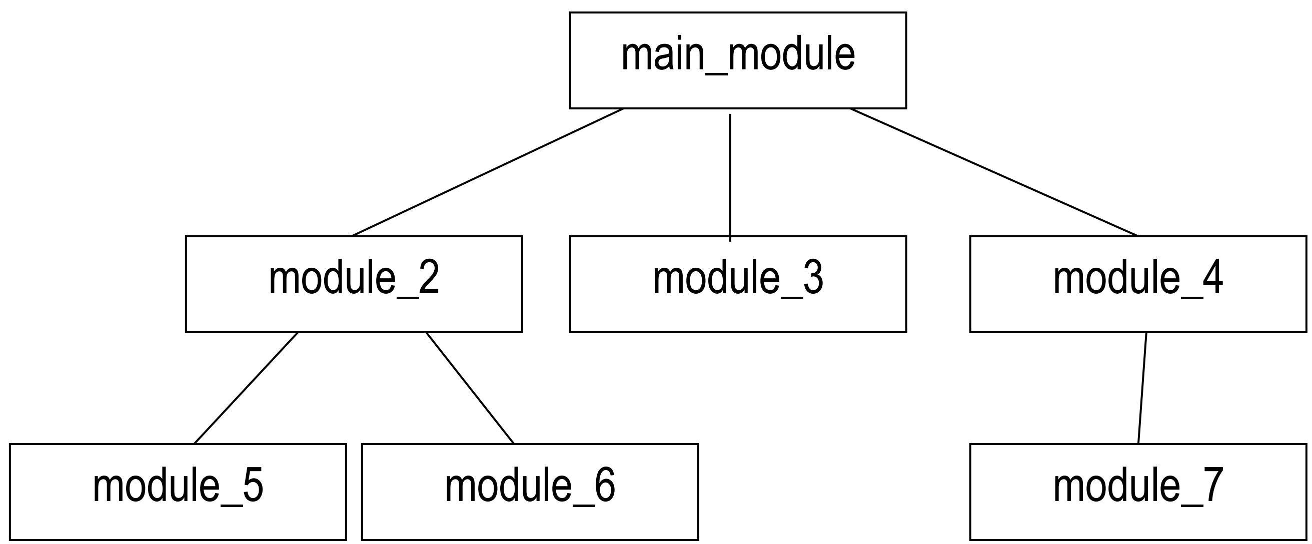
Design tool ☺

1. Show which modules (potentially) **call** other modules
2. May show parameters
   i.e., **data flow** from one module to another
3. Does NOT show algorithm details!



Hierarchy Charts

- A module in a chart is called by the module above it
  which is **connected by a line**.
- Modules at the same level are called from **left** to **right**.
- A module calls its *children* modules **before** its *sibling* is called.



Hierarchy Charts

- The previous chart can be represented by the following pseudocode.

```
main_module
    ...
    module_2
    ...
    module_3
    ...
    module_4
    ...
END
```

```
module_2
    ...
    module_5
    ...
    module_6
    ...
END
```

```
module_4
    ...
    module_7
    ...
END
```

```
module_3
    ...
END
```

```
module_5
    ...
END
```

```
module_6
    ...
END
```

```
module_7
    ...
END
```

Module 3, 5, 6, 7
Start no other
module!

Hierarchy Charts

- For an algorithm which is described by the above chart, a *possible order* in which those modules are called is:

main_module
module_2
module_5
module_6
module_3
module_4
module_7

Why is this order one possible order of many?

Depends if each module meets the right condition to get called into action.

Hierarchy Charts (example with data)

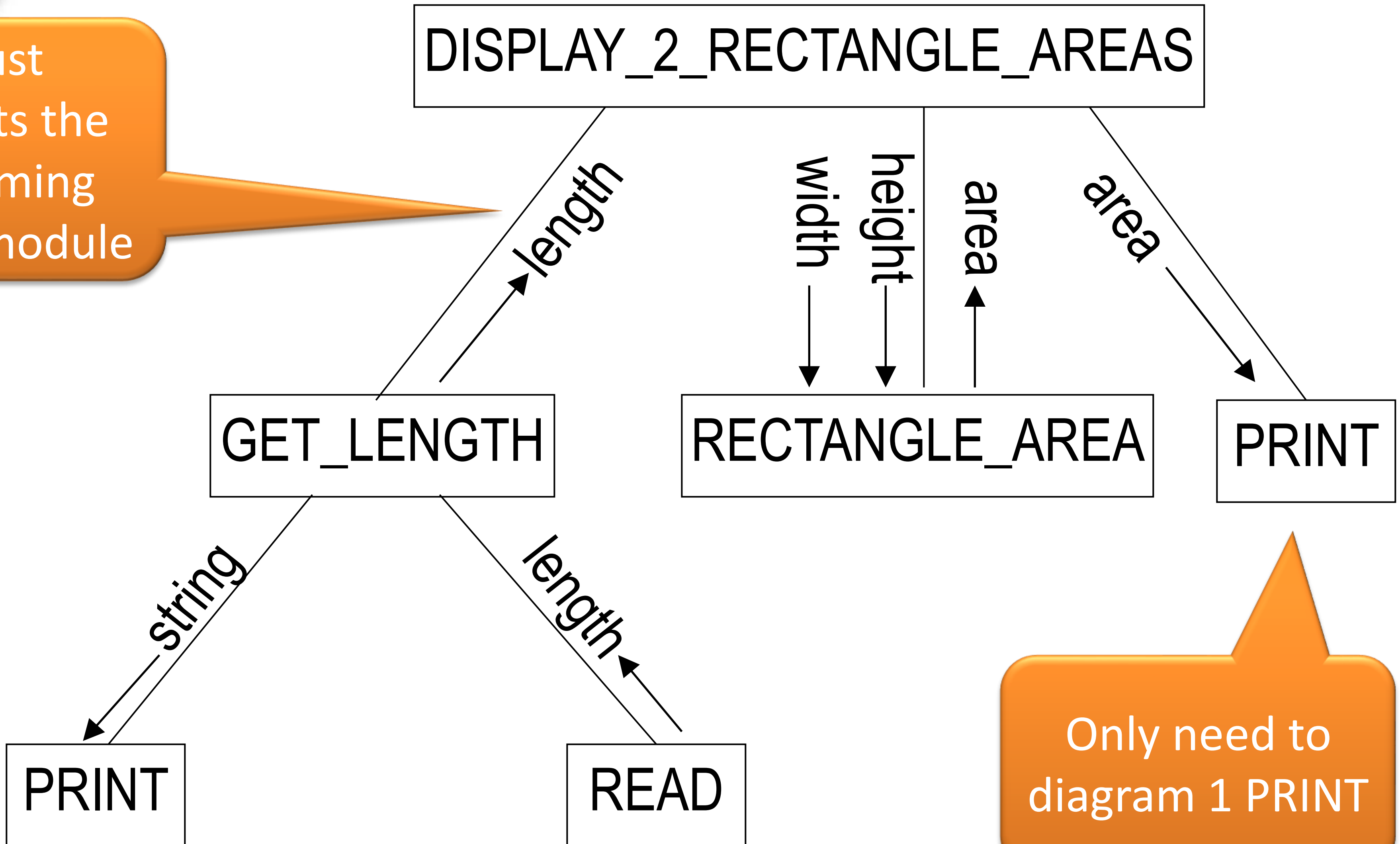
We can see here from the Main module what the program is about.

Get some numbers, work out an area and then display them.

- **Convert the following pseudocode to a hierarchy chart, include data and modules.**

```
DISPLAY_2_RECTANGLE_AREAS  //main module
    GET_WIDTH width //whatever is returned from
                                        GET_WIDTH
    GET_LENGTH height //is place inside variables
    RECTANGLE_AREA area1, width, height
    GET_WIDTH width
    GET_LENGTH height
    RECTANGLE_AREA area2, width, height
    PRINT area1
    PRINT area2
END
```

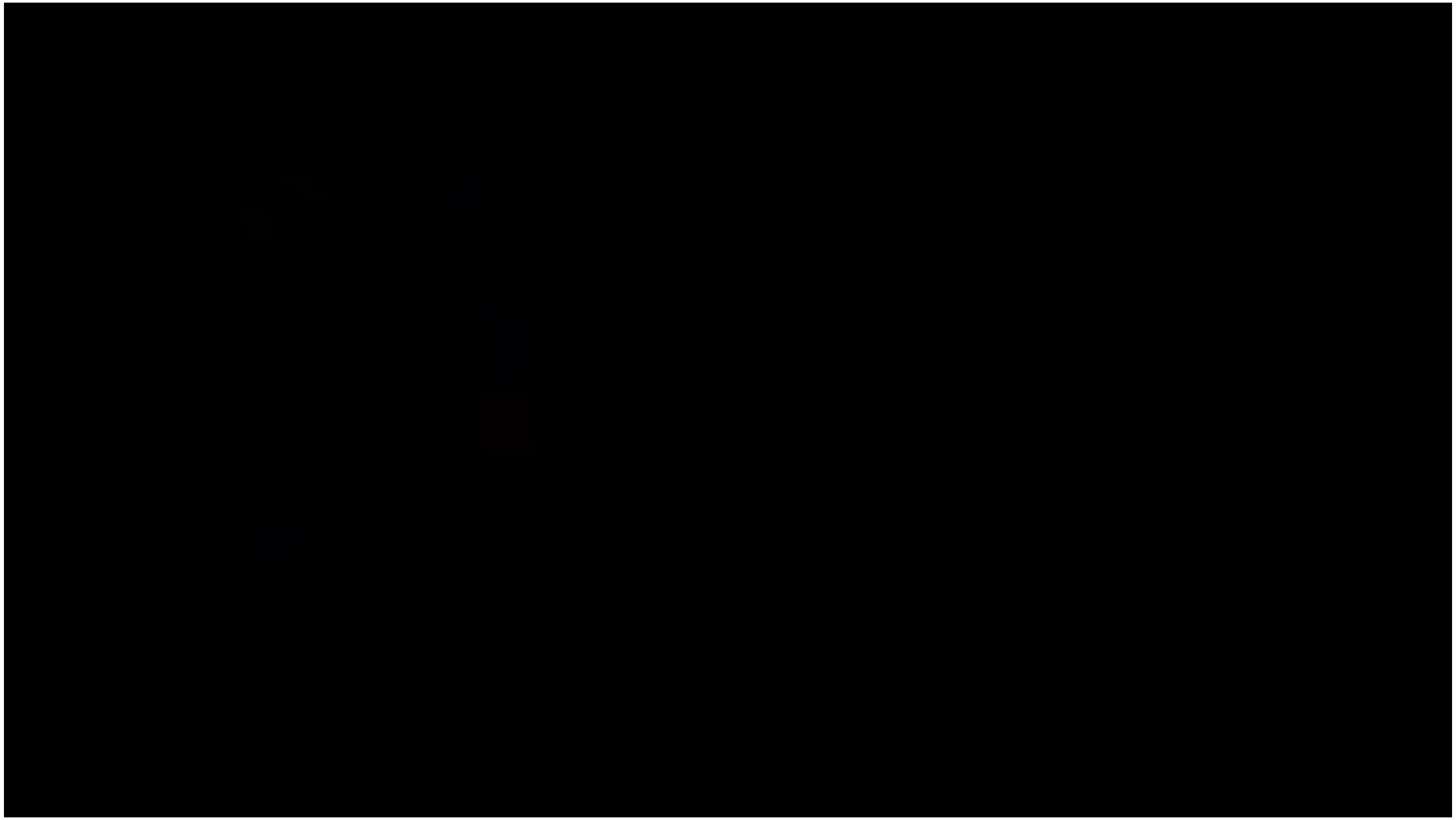
Hierarchy Charts (example with data)



This just represents the value coming from the module

Only need to diagram 1 PRINT

GAME: Can you solve the river crossing riddle



GAME: Can you solve the river crossing riddle

As a wildfire rages through the grasslands, three lions and three wildebeest flee for their lives. To escape the inferno, they must cross over to the left bank of a crocodile-infested river. Fortunately, there happens to be a raft nearby. It can carry up to two animals at a time, and needs as least one lion or wildebeest on board to row it across the river. There's just one problem. If the lions ever outnumber the wildebeest on either side of the river, even for a moment, their instincts will kick in, and the results won't be pretty. That includes the animals in the boat when it's on a given side of the river. What's the fastest way for all six animals to get across without the lions stopping for dinner?