Pseudo Code & Java (Chapter 1)

Eleni Mangina

Room B2.05
School of Computer Science and Informatics
University College Dublin, Ireland

Writing a Java Program

- Design
- Coding
- Testing and Debugging

Design

- Responsibilities: Divide the work into different actors, each with a different responsibility.
- Independence: Define the work for each class to be as independent from other classes as possible. Subdivide responsibilities between classes and give data to the class that has jurisdiction over the actions that require access to this data.
- Behaviours: So that the consequences of each action performed by the class will by well understood by other classes that interact with it. These behaviours will define the methods that this class performs.

Pseudo Code

- A simple (non executable) language that allows more abstract descriptions of algorithms.
  - It is not intended to be as rigorous as a programming language.
- Syntax not precisely defined.
  - Combines programming style concepts with mathematical notation and natural language.
- Pseudo code should be clear enough to provide an easy mapping to a programming language.
  - For Java, algorithms are implemented as **methods…**

Pseudo Code

- Typically, pseudo code combines the following features:
  - Operator support for expressions: **+ - * / % & | ! = < > <= >= <>**.
  - An assignment operator: ←.
  - If statements: **if … then … [else…]**
  - While loops: **while … do …**
  - For loops: **for … to … do …**
  - Repeat loops: **repeat … until …** (like do…while … loops in Java)
  - Array indexing: **A[i]**
  - Method Declarations: **Algorithm algorithmName(param1, …)**
  - Procedure Calls: **algorithmName(param1,…)**
  - Returns: **return …**

- Other possible features include:
  - Console Printing: **print(…)**
  - User input: **read()**

Example: Sum Range

Algorithm SumRange(lower, upper):

Input: Two integer numbers specifying the range

Output: The sum of the integers in the given range

sum ← 0

for every integer value i in the range [lower, upper] **do**

sum ← sum + I

return sum

Understanding Pseudo Code

- **Algorithm Tracing**: a technique that is used to study how state of the variables change as the algorithm is executed.

- To perform a trace, you:
  1. Assign a number to each line in the algorithm
  2. Create a **variable table** whose columns are variables and whose rows represent state changes
  3. Identify sample values (test cases) for arguments

- But, tracing is not an exact science…
  - In some cases, it is enough just to give the state at key points in the program…

Example: Sum Range Trace

Algorithm SumRange(lower, upper):

Input: Two integer numbers specifying the range

Output: The sum of the integers in the given range

```
01    sum ← 0
02    for every integer value i in the range [lower, upper] do
03            sum ← sum + i
04    return sum
```

Example: lower = 5, upper = 8

Line	sum	i
01	0	-
02	0	5
03	5	5
02	5	6

Line	sum	i
03	11	6
02	11	7
03	18	7
02	18	8

Line	sum	i
03	26	8
04	26	-

Sample Problem: Is Prime

NOTE: an integer, n, is prime if none of the integers in the range 2 - √n divide into n.

Algorithm IsPrime(value):

Input: An integer number to be checked

Output: true if the number is prime, false otherwise

```
if value < 4 then
        return true
for every integer value i in the range [2, √value] do
        if i % value = 0 then
                return false
return true
```

Sample Problem: Is Prime

Algorithm IsPrime(value):

Input: An integer number to be checked

Output: true if the number is prime, false otherwise

```
01 if value < 4 then
02     return true
03 for every integer value i in the range [2, √value] do
04     if i % value = 0 then
05         return false
06 return true
```

Chosen arguments: 11 and 16

Arrays and Tracing

- Main Issue: How to represent an array
- Answer: Sequence of boxes labeled with the index
- Example: an array of size 10:

0	1	2	3	4	5	6	7	8	9

Example: Sum Array

Algorithm SumArray(A, n):

Input: An integer array A of size n.

Output: The Sum of the values in A.

```
01      sum ← 0
02for k in the range 0 to n-1 do
03      sum ← sum + A[k]
04return sum
```

Sample Data: A = { 5, 12, 4, 6, 2 }; n = 5

Sample Problem: Max Array

Algorithm MaxArray(A, n):

Input: An integer array A of size n.

Output: The maximum value in A.

```
01      currentMax ← A[0]
02for k in the range 1 to n-1 do
03      If currentMax < A[k] then
04              currentMax ← A[k]
05return currentMax
```

Sample Data: A = { 5, 12, 4, 6, 2 }; n = 5

Translating Pseudo Code

- Issues:
  - No variable declaration / type checking
  - Fuzzy expressions / statements
  - Syntactic Idiosyncrasies

- Approach for Java:
  - 1 method per algorithm
  - Not associated with a DS => static method
  - Use System.out.println(…) to replicate trace
  - Use test cases to validate code!!!

Coding - Javadoc

```java
/**
 * This class defines an immutable (x,y) point in the plane.
 *
 * @author Michael Goodrich
 */
public class XYPoint {
  private double x,y; // private instance variables for the coordinates

  /**
   * Construct an (x,y) point at a specified location.
   *
   * @param xCoor The x-coordinate of the point
   * @param yCoor The y-coordinate of the point
   */
  public XYPoint(double xCoor, double yCoor) {
    x = xCoor;
    y = yCoor;
  }

  /**
   * Return x-coordinate value.
   *
   * @return x-coordinate
   */
  public double getX() { return x; }

  /**
   * Return y-coordinate value.
   *
   * @return y-coordinate
   */
  public double getY() { return y; }
}
```

Code Fragment 1.8: An example class definition using javadoc-style comments. Note that this class includes two instance variables, one constructor, and two accessor methods.

The 1st Assignment...

DO NOT MISS THE LAST 15’ OF THE LAB

HOW MANY OPERATIONS??

Algorithm Average(a, b):

Input: Two integers a and b

Output: The average of a and b

return (a + b) / 2

HOW MANY OPERATIONS??

Algorithm MaxInt(a, b):

Input: Two integers a and b

Output: The larger of the two integers

```
if a > b then
    return a
else
    return b
```

HOW MANY OPERATIONS??

Algorithm Difference(a, b):

Input: Two integers a and b

Output: The difference between a and b

```
if (a > b) then
        return a – b
else
    return b - a
```

HOW MANY OPERATIONS??

Algorithm MinValue(A, n):

Input: An integer array A of size n

Output: The smallest value in A

minValue ← A[0]

for k=1 **to** n-1 **do**

if (minValue > A[k]) **then**

minValue ← A[k]

return minValue

A little exercise….
To introduce analysis of algorithms...

What does the following algorithm do?

Algorithm Foo (a, n):

Input: two integers, a and n

Output: ?

k ← 0

b ← 1

while k < n **do**

k ← k + 1

b ← b * a

return b

Solution:

The algorithm computes a^n.

The initial assignments take constant time

Each iteration of the while loop takes constant time

There are exactly n iterations

What does the following algorithm do?

```
Algorithm Bar (a, n):
        Input: two integers, a and n
        Output: ?
    k ← n
    b ← 1
c ← a
    while k > 0  do
        if k mod 2 = 0 then
                k ← k/2
                c ← c * c
        else
                k ← k – 1
                b ← b*c
    return  b
```

Solution:

This algorithm computes a^n. Its running time is O(logn) for the following reasons:

*The initialization and the **if** statement and its contents take constant time, so we need to figure out how many times the **while** loop gets called. Since k goes down (either gets halved or decremented by one) at each step, and it is equal to n initially, at worst the loop gets executed n times. But we can (and should) do better in our analysis.*

*Note that if k is even, it gets halved, and if it is odd, it gets decremented, and halved in the next iteration. So at least every second iteration of the **while** loop halves k. One can halve a number n at most [logn] times before it becomes **≤** 1 (each time we halve a number we shift it right by one bit, and a number has [logn] bits). If we decrement a number in between halving it, we still get to halve no more than [logn] times. Since we can only decrement k in between two halving operations (unless n is odd or it is the last iteration) we get to do a decrementing iteration at most [logn] + 2 times. So we can have at most 2[logn] + 2 iterations.*