



13. An Example Program

To illustrate some of what we have covered so far about programming in Java – starting with a problem you have already met and should have tried to program

1



Extending the practice project

- n Will look at developing an application that extends the practice programming project to go over some programming concepts and methods
- n Description of the problem based on the lecture last year is on the next few slides....

2



The Towers of Hanoi Puzzle

(as described by Baldwin & Scragg, 2004)

- n *Legend tells of an ageless temple in the city of Hanoi. Within this temple are three poles of diamond piercing 64 golden disks of varying sizes. At the moment of creation, the disks were stacked smaller-on-top-of-larger on the leftmost pole.*
- n *Since then, the monks of the temple have been moving the disks from the left pole to the right pole, subject to the following rules:*
 - n *The monks only move one disk at a time.*
 - n *Every disk taken off a pole has to be put back on some pole before moving another disk.*
 - n *The monks never put a disk on top of a smaller one.*
- n *When all the disks have been moved to the right-hand pole, the purpose of creation will have been fulfilled and the universe will end.*

3



The Project

- n You were to develop a program to validate potential solutions to this puzzle.
- n To facilitate the solution to this puzzle, you were to write a simulation program to explore the various possibilities. To keep the program simple, n was limited to 3.
 - n The restriction to three disks was to allow a solution that did not need to use an array as we had not covered arrays in any detail at the time

4



The Required Program Functionality

- n When started, the program should display a description of the problem
- n Do, while the problem is not solved
 - n Display where each of the three disks are located and prompt the user to indicate which disk to move and (if the disk can be moved) to which pole
 - n If the user input is valid, the program should update the program state (i.e. move the disk and display the new locations for the disks) and add one to the count of moves
 - n If the user input is invalid, the program should display an error message and not update the program state
- n On completing the puzzle the program should display a congratulations message and indicate whether the problem was solved with the optimal number of moves

5

An Outline Solution

- n Initialise the program state
 - n Set small, medium and large disk locations to pole A
 - n Set count of moves to zero
- n Display problem description
- n do
 - n Display the location of the three disks
 - n Prompt the user to enter which disk to move
 - n Get the user's choice and check that the disk can move
 - n if (the disk can move)
 - n Prompt the user to enter which pole to move the disk to
 - n Get the user's choice and check that the disk can move to that pole
 - n if (the disk can move to the chosen destination pole)
 - n Move the disk and add one to the count of moves
 - n else report that the chosen destination pole contains a smaller disk
 - n else report that the disk cannot move as there is a disk on top of it
- n while (the three disks are not on pole C)
- n Display congratulations message and count of moves

6



Tracking the disk locations

- n The presentation before the break suggested that with three disks it would be enough just to track where each disk was
 - n You would not need to maintain a list of the disks on each pole
- n Maintain a variable for each disk indicating which pole it is on
 - n Solutions submitted differed in whether they made these variables `char`, `int` or `String`

7



Using methods (subroutines)

- n Some of the steps in the outline solution can be delegated to subroutines, for example
 - n Displaying the problem description
 - n Get the user's choice of disk to move
 - n Get the user's choice of pole to move to
 - n Displaying the congratulations message
- n Depending on the choice of type for the three disk variables, other helper subroutines are possible

8



The main() method

```
public static void main(String[] args) {

    char smallDisk = POLE_A; // Current location of the small disk
    char mediumDisk = POLE_A; // Current location of the medium disk
    char largeDisk = POLE_A; // Current location of the large disk

    char chosenDisk; // To record which disk it is requested to move
    char chosenPole; // To record which pole to move the disk to

    boolean diskValid = false; // Can disk move from current pole?
    boolean poleValid = false; // Can disk move to the chosen pole?
    int moves = 0; // Used to count the number of moves
    char currentPole = POLE_A; // Used to record where a disk is
                                // moving from, to help validate
                                // that the requested move is to
                                // a different pole.

    displayInstructions();
```

9

```
do {

    // Display the current location of each disk
    TextIO.putln("The small disk is on pole " + smallDisk);
    TextIO.putln("The medium disk is on pole " + mediumDisk);
    TextIO.putln("The large disk is on pole " + largeDisk);
    TextIO.putln();

    // Get the details of the disk to move
    chosenDisk = getDisk();

    // Check that the disk is the top disk on its pole
    switch(chosenDisk) {
    case SMALL_DISK :
        diskValid = true;
        currentPole = smallDisk;
        break;
    case MEDIUM_DISK :
        diskValid = (mediumDisk != smallDisk);
        currentPole = mediumDisk;
        break;
    case LARGE_DISK :
        diskValid = (smallDisk != largeDisk &&
                     mediumDisk != largeDisk);
        currentPole = largeDisk;
        break;
    }

}
```

10

```
if (diskValid) {
    // Get and validate the details of the pole.
    chosenPole = getPole();
    if (chosenPole == currentPole) {
        TextIO.putln("The disk is already on this pole!");
    } else {
        switch(chosenDisk) {
        case SMALL_DISK :
            // The small disk can always move to another pole.
            poleValid = true;
            smallDisk = chosenPole;
            break;
        case MEDIUM_DISK :
            // The medium disk can move to the chosen pole if the pole
            // does not contain the small disk.
            poleValid = (chosenPole != smallDisk);
            if (poleValid) mediumDisk = chosenPole;
            break;
        case LARGE_DISK :
            // Check that neither the small or medium disk are on the
            // chosen pole.
            poleValid = (chosenPole != smallDisk &&
                        chosenPole != mediumDisk);
            if (poleValid) largeDisk = chosenPole;
            break;
        }
    }
}
```

11

```
        if (!poleValid) {
            TextIO.putln("Cannot move to " + chosenPole +
                        " as it contains a smaller disk.");
        } else {
            moves++; // a disk was moved
        }
    }
    else { // !diskValid
        TextIO.putln("Cannot move " + chosenDisk +
                    " as there are disks on top of it.");
    }
} while (!(smallDisk==POLE_C &&
           mediumDisk==POLE_C &&
           largeDisk==POLE_C));

displayPuzzleSolved(moves);

} // end main()
```

12

The helper methods

```
private static char getChar() { // returns input char in uppercase
    char result = TextIO.getlnChar();
    result = Character.toUpperCase(result);
    return result;
}

private static char getDisk() {
    int attempts = 0;
    char disk;
    do {
        if (attempts > 0) {
            TextIO.putln("There is no such disk!");
        }
        TextIO.put("Which disk would you like to move? " +
            "(S=Small, M=Medium, L=Large): ");
        disk = getChar();
        attempts++;
    } while (disk != SMALL_DISK && disk != MEDIUM_DISK &&
        disk != LARGE_DISK);
    return disk;
}
```

13

The helper methods cont'd

```
private static char getPole() {
    int attempts = 0;
    char pole;

    do {
        if (attempts > 0) {
            TextIO.putln("There is no such pole!");
        }
        TextIO.put("Which pole do you want to move the disk to?" +
            "(A=Pole A, B=Pole B, C=Pole C): ");
        pole = getChar();
        attempts++;
    } while (pole != POLE_A && pole != POLE_B && pole != POLE_C);
    return pole;
}
```

14

The helper methods cont'd

```
private static void displayPuzzleSolved(int moves) {
    TextIO.putln("Well done, all three disks are on pole C " +
        "and you have solved the puzzle of the Towers of Hanoi!");
    if (moves > OPTIMAL_MOVES) {
        TextIO.putln("It took you " + moves + " moves," +
            " so try again as it can" +
            " be solved in fewer moves than that!");
    } else {
        TextIO.putln("You solved the puzzle in the minimum " +
            "number of moves: " + moves + "!");
    }
}

private static void displayInstructions() {
    // Just lots of TextIO.putln() statements in here!
}
```

15

Generalising the solution

- n We have now covered arrays so we can lift the restriction that the number of disks is limited to three
- n Allow the user to specify how many disks they want
 - n Will now need to represent the relative sizes of the disks
 - n Probably the simplest solution is to represent the disks as integers, with a larger value implying a larger disk
- n There will always be just three poles, so amend the solution to keep track of what is on each pole
 - n The number of disks per pole will change as the disks are moved
 - n We could keep a count of the number of disks on each pole and use an array of int for each, but it will be simpler to use an ArrayList<Integer> as its size() method will do the counting for us

16



Adding disks to a pole

- n As the towers are built, the larger disks are always on the bottom with smaller disks on the top
- n As we add disks to a pole (an `ArrayList<Integer>`) the larger disks will be at the beginning of the list (as we add those first) and the smaller disks will be at the end of the list (as we add those last)
 - n The largest disk on a non-empty pole will be at position 0
 - n The smallest disk on a non-empty pole will be at position `size()-1`
 - n To determine if we can add a disk to a non-empty pole we only need look at the top disk to check that it is larger than the disk being added

17



Dealing with the input

- n In our original problem we could ask the user to specify which disk to move but as there are now potentially a large number of disks it will be simpler to just deal in poles
 - n Ask the user which pole they want to move the top disk from and to which pole they want to move it

18



Validating the input

- n We need to check that the user inputs a valid pole name (i.e. one of the three)
- n In validating the pole to move *from* we check that the pole is not empty
- n In validating the pole to move *to* we check that if it is not empty the top disk is larger than the disk that user is trying to place on it

19



Moving the disk

- n To move a disk we need to remove it from the pole it is on...
 - n This means calling the `remove()` method of the `ArrayList`, *from*, which returns the object being removed
- n ... and add it to the top of the pole it is being moved to
 - n This just means calling the `add()` method of the `ArrayList`, *to*, as the end of the list is the top of the pole

20



Moving the disk continued

- n We can do the move in a few steps
 - n Remove the disk from the pole it is moving from:

```
int topOfPoleIndex = from.size()-1;
int disk = from.remove(topOfPoleIndex);
```

- n Add the disk to the top of the pole it is moving to
- ```
to.add(disk);
```

- n Or combine the steps to do it all at once:

```
to.add(from.remove(from.size()-1));
```

21



## How many moves?

- n With 3 disks, the minimum number of moves needed to solve the puzzle is 7
- n With each additional disk, the number of moves needed doubles (+1 more)
  - n 15 moves for four disks, 31 for five disks, 63 for six disks, 127 for seven disks etc.
- n For  $N$  disks at least  $2^N - 1$  moves are needed

22

## An Outline Solution

- n Display problem description
- n Prompt the user for the number of disks and get the response
- n Initialise the program state
  - n Create the three ArrayLists to represent the three poles
  - n Add all the disks to pole A
  - n Set count of moves to zero
- n do
  - n Display what is on each of the three poles
  - n Prompt the user to enter which pole to move the top disk from
  - n Get the user's choice and check that the pole is not empty
  - n if (the pole is not empty)
    - n Prompt the user to enter which pole to move the disk to
    - n Get the user's choice and check that the disk can move to that pole
    - n if (the disk can move to the chosen destination pole)
      - n Move the disk and add one to the count of moves
    - n else report that the chosen destination pole contains a smaller disk
  - n else report that there is no disk on the chosen pole to move
- n while (not all the disks are on pole C)
- n Display congratulations message and count of moves

23



## Refining the outline solution

- n Again we can use some helper methods
- n The logic needs to accommodate any number of disks but the changes to the previous logic are quite small
  - n A disk can move to another pole if the pole is empty (whatever size the disk is) or if it is smaller than the top disk on the other pole

24

## How do we compare the disk sizes?

- n We are representing each disk by a number
  - n E.g. if there are 10 disks the numbers are 1 to 10, with 10 representing the largest disk and 1 the smallest
- n So to compare the sizes of two disks we just compare them
  - $x < y$  means that  $x$  is smaller than  $y$

25

## Some observations

- n The program is on the next few slides
- n You might be surprised to see that it is slightly smaller and simpler than the original solution with just 3 disks
  - n because ArrayList does some of the work for us
  - n and because once you explicitly capture what is on each pole the problem is easier to describe and translate into code!

26

## The main() method

```
public static void main(String[] args) {

 displayInstructions();
 int numberOfDisks = getNumberOfDisks();

 // Always have three poles, so use an array
 ArrayList<Integer>[] poles = new ArrayList[3];

 // Create an ArrayList<Integer> for each pole
 for (int i = 0; i < poles.length; i++) {
 poles[i] = new ArrayList<Integer>();
 }

 // Use 1 to numberOfDisks to represent the disk sizes
 // Add disks in order of decreasing size, largest first
 for (int disk = numberOfDisks; disk > 0; disk--) {
 poles[0].add(disk);
 }
}
```

27

```
int moves = 0;
final int MIN_MOVES = (int) Math.pow(2, numberOfDisks) - 1;
do {
 display(poles);
 TextIO.putln("Remove disk from which pole?");
 int fromPole = getPole();
 ArrayList<Integer> from = poles[fromPole];
 if (from.size() > 0) {
 TextIO.putln("Move disk to which pole?");
 int toPole = getPole();
 ArrayList<Integer> to = poles[toPole];
 if (to.size() == 0 ||
 to.get(to.size()-1) > from.get(from.size()-1)) {
 to.add(from.remove(from.size()-1));
 moves++;
 } else {
 TextIO.putln("Disk is too large to place on this pole");
 }
 } else {
 TextIO.putln("There are no disks on this pole to move!");
 }
} while (poles[2].size() < numberOfDisks);
display(poles);
displayPuzzleSolved(moves, MIN_MOVES);
}
```

28



## Yes, main() is just two slides!

- n The main() method is half the size of the one to solve the mini-Towers problem with three disk variables
- n To save typing the array and index names several times the variables from and to are declared and refer to the relevant pole's ArrayList
  - n As these are reference types these are just aliases, the ArrayLists have not been copied
- n The getPole() method is modified so that it returns the index of the pole in the array of poles – but the rest of the method is the same
- n The solution uses an array of String, NAME, to store the names for each pole, used in the display() method

29

## The helper methods

```
private static char getChar() { // return the input char in uppercase
 char result = TextIO.getlnChar();
 result = Character.toUpperCase(result);
 return result;
}

/* Prompts the user to enter the pole and returns the index
 of that pole in the array of poles */
private static int getPole() {
 int attempts = 0;
 char pole;
 do {
 if (attempts > 0) {
 TextIO.putln("There is no such pole!");
 }
 TextIO.put("Choose the pole (A=Pole A, B=Pole B, C=Pole C): ");
 pole = getChar();
 attempts++;
 } while (pole != POLE_A && pole != POLE_B && pole != POLE_C);
 return pole - POLE_A; // if user chooses A the method returns 0
}
```

30

## The helper methods cont'd

```
/*
 * Displays the disks on each of the poles,
 * row by row starting at pole A.
 */
private static void display(ArrayList<Integer>[] poles) {
 if (poles.length != NAME.length) {
 throw new IllegalArgumentException();
 }
 for (int current=0; current < poles.length; current++) {
 TextIO.putln("NAME[current] + ": " + poles[current]);
 TextIO.putln();
 }
}
```

31



## The helper methods cont'd

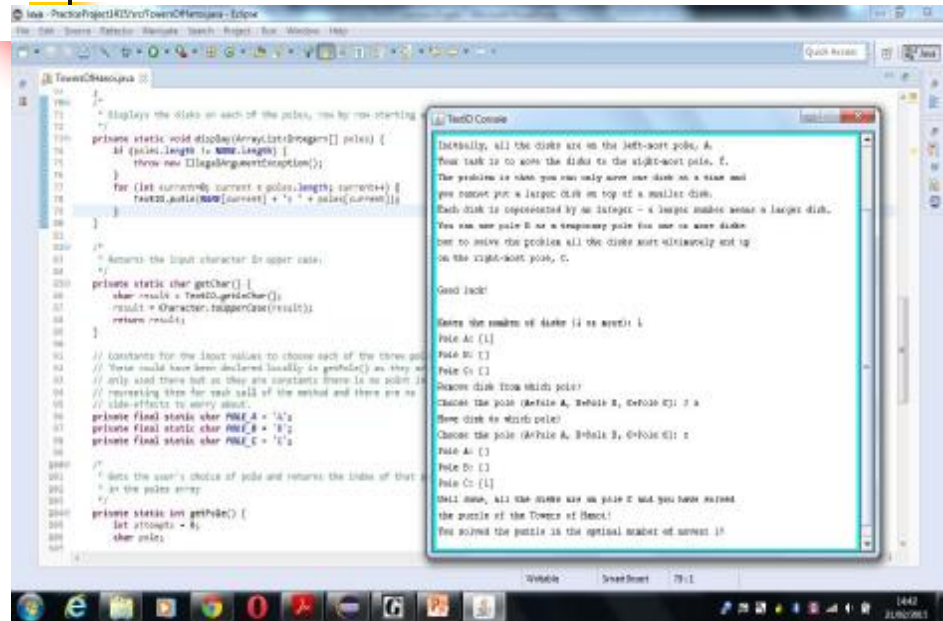
```
private static void displayPuzzleSolved(int moves,
 final int OPTIMAL_MOVES) {
 TextIO.putln("Well done, all the disks are on pole C " +
 "and you have solved the puzzle of the Towers of Hanoi!");
 if (moves > OPTIMAL_MOVES) {
 TextIO.putln("It took you " + moves + " moves," +
 " so try again as it can " +
 " be solved in fewer moves than that!");
 } else {
 TextIO.putln("You solved the puzzle in the minimum " +
 "number of moves: " + moves + "!!");
 }
}

private static void displayInstructions() {
 // Just lots of TextIO.putln() statements in here!
}
```

32



## A run of the program



The screenshot shows a Java IDE with a file named 'TowersOfHanoi.java'. The code implements the Towers of Hanoi algorithm using recursive static methods. It includes comments in English explaining the problem and the recursive steps. The 'TextIO Console' window shows the program's output, which includes instructions for the user, the state of the three poles (A, B, C) with the number of disks on each, and the sequence of moves (e.g., 'Move disk 4 to which pole?'). The console output ends with 'You solved the puzzle in the optimal number of moves!'

33

## Concluding comments

- n These two versions of the program have used
  - n an outline of the algorithm in pseudocode
  - n primitive type variables and values (local and static)
  - n TextIO for input and output
  - n do while loops
  - n for loops
  - n switch statements
  - n if statements
  - n subroutines (static methods)
    - n Both void and with return types
  - n wrapper classes
    - n Integer as well as the Character.toUpperCase() method
  - n Arrays of reference types (in this case, of ArrayList and of String)
  - n ArrayLists and their add(), remove(), get() and size() methods

34

## The two programs

- n Are both on Moodle
  - n Download them and run them
- n If there is something you do not understand in them, ask your tutor or email the module coordinator
- n The programming project will be coming in the next few weeks!

35

## Questions?

36