Chapter 4

- We learn about Control Structures
  - Structured-programming principle
  - Control structures help build and manipulate objects (Chapter 8)

1

Algorithms & Pseudocode

- Algorithm
  - Series of actions in specific order
    - The actions executed
    - The order in which actions execute
- Program control
  - Specifying the order in which actions execute
    - Control structures help specify this order
- Pseudocode
  - Informal language for developing algorithms
  - Not executed on computers
  - Helps developers “think out” algorithms

2

Control Structures

- Sequential execution
  - Program statements execute one after the other
- Transfer of control
  - Three control statements can specify order of statements
    - Sequence structure
    - Selection structure
    - Repetition structure
- Flowchart
  - Graphical representation of algorithm
    - Flowlines indicate order in which actions execute

3

Java Keywords				
`abstract`	`boolean`	`break`	`byte`	`case`
`catch`	`char`	`class`	`continue`	`default`
`do`	`double`	`else`	`extends`	`false`
`final`	`finally`	`float`	`for`	`if`
`implements`	`import`	`instanceof`	`int`	`interface`
`long`	`native`	`new`	`null`	`package`
`private`	`protected`	`public`	`return`	`short`
`static`	`super`	`switch`	`synchronized`	`this`
`throw`	`throws`	`transient`	`true`	`try`
`void`	`volatile`	`while`		
Keywords that are reserved, but not used, by Java				
`const`	`goto`			

Fig. 4.2 Java keywords.

4

Selection Structures

- Java has a sequence structure “built-in”
- Java provides three selection structures
  - **`if`**
  - **`if/else`**
  - **`switch`**
- Java provides three repetition structures
  - **`while`**
  - **`do/while`**
  - **`do`**
- Each of these words is a Java keyword

5

The if/else Selection Structure

- Perform action only when condition is **`true`**
- Perform different specified action when condition is **`false`**
- Conditional operator (**`?:`**)
- Nested **`if/else`** selection structures

6

Formulating Algorithms: Case Study 1 (Counter-Controlled Repetition)

- Counter
  - Variable that controls number of times set of statements executes
- **`Average1.java`** calculates grade averages
  - uses counters to control repetition

Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
Input the next grade
Add the grade into the total
Add one to the grade counter

Set the class average to the total divided by ten
Print the class average

7

```
1  // Fig. 4.7: Average1.java
2  // Class average program with counter-controlled repetition.
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class Average1 {
8
9     // main method begins execution of Java application
10    public static void main( String args[] )
11    {
12       int total,          // sum of grades input by user
13           gradeCounter,   // number of grades entered
14           gradeValue,     // grade value
15           average;        // average of all grades
16       String grade;       // grade typed by user
17
18       // Initialization Phase
19       total = 0;          // clear total
20       gradeCounter = 1;   // prepare to loop
21
22       // Processing Phase
23       while ( gradeCounter <= 10 ) {  // loop 10 times
24
25          // prompt for input and read grade from user
26          grade = JOptionPane.showInputDialog(
27             "Enter integer grade: " );
28
29          // convert grade from a String to an integer
30          gradeValue = Integer.parseInt( grade );
31
32          // add gradeValue to total
33          total = total + gradeValue;
34
```

`Average1.java`

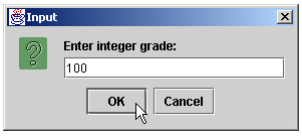
`gradeCounter`

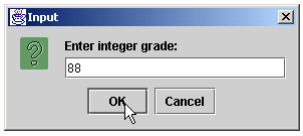
Line 23

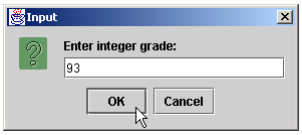
8

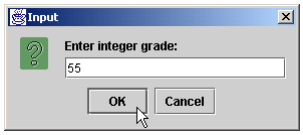
```
35          // add 1 to gradeCounter
36          gradeCounter = gradeCounter + 1;
37
38       }  // end while structure
39
40       // Termination Phase
41       average = total / 10;  // perform integer division
42
43       // display average of exam grades
44       JOptionPane.showMessageDialog( null,
45          "Class average is " + average, "Class Average",
46          JOptionPane.INFORMATION_MESSAGE );
47
48        System.exit( 0 );     // terminate the program
49
50    }  // end method main
51
52 }  // end class Average1
```

Average1.java

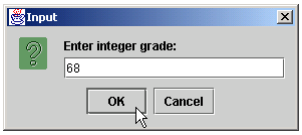


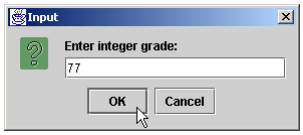


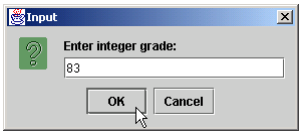


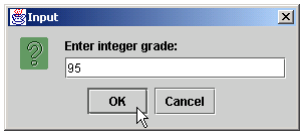


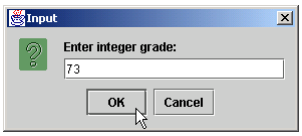
9

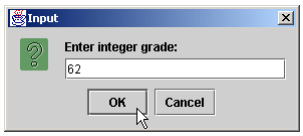


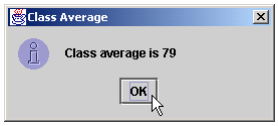










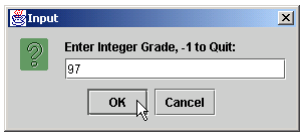


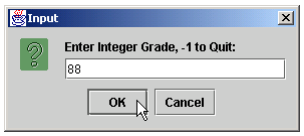
Average1.java

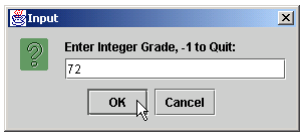
10

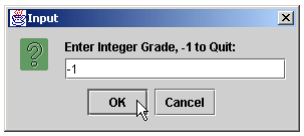
```
67    }  // end method main
68
69 }  // end class Average2
```

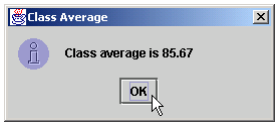
Average2.java











11

Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 2 (Sentinel-Controlled Repetition)

- Sentinel value
  - Used to indicated the end of data entry
- **`Average2.java`** has indefinite repetition
  - User enters sentinel value (**`-1`**) to end repetition

Initialize total to zero
Initialize counter to zero

Input the first grade (possibly the sentinel)

While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
else
    Print "No grades were entered"

12

```
1  // Fig. 4.9: Average2.java
2  // Class average program with sentinel-controlled repetition.
3
4  // Java core packages
5  import java.text.DecimalFormat;
6
7  // Java extension packages
8  import javax.swing.JOptionPane;
9
10 public class Average2 {
11
12    // main method begins execution of Java application
13    public static void main( String args[] )
14    {
15       int gradeCounter,  // number of grades entered
16           gradeValue,    // grade value
17           total;         // sum of grades
18       double average;    // average of all grades
19       String input;      // grade typed by user
20
21       // Initialization phase
22       total = 0;         // clear total
23       gradeCounter = 0;  // prepare to loop
24
25       // Processing phase
26       // prompt for input and read grade from user
27       input = JOptionPane.showInputDialog(
28          "Enter Integer Grade, -1 to Quit:" );
29
30       // convert grade from a String to an integer
31       gradeValue = Integer.parseInt( input );
32
```

Average2.java

13

```
33       while ( gradeValue != -1 ) {
34
35          // add gradeValue to total
36          total = total + gradeValue;
37
38          // add 1 to gradeCounter
39          gradeCounter = gradeCounter + 1;
40
41          // prompt for input and read grade from user
42          input = JOptionPane.showInputDialog(
43             "Enter Integer Grade, -1 to Quit:" );
44
45          // convert grade from a String to an integer
46          gradeValue = Integer.parseInt( input );
47       }
48
49       // Termination phase
50       DecimalFormat twoDigits = new DecimalFormat( "0.00" );
51
52       if ( gradeCounter != 0 ) {
53          average = (double) total / gradeCounter;
54
55          // display average of exam grades
56          JOptionPane.showMessageDialog( null,
57             "Class average is " + twoDigits.format( average ),
58             "Class Average", JOptionPane.INFORMATION_MESSAGE );
59       }
60       else
61          JOptionPane.showMessageDialog( null,
62             "No grades were entered", "Class Average",
63             JOptionPane.INFORMATION_MESSAGE );
64
65       System.exit( 0 );    // terminate application
66
```

Average2.java

Line 33

Line 50

14

Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 3 (Nested Control Structures)

- Nested control structures

Initialize passes to zero
Initialize failures to zero
Initialize student to one
While student counter is less than or equal to ten
Input the next exam result
If the student passed
Add one to passes
else
Add one to failures
Add one to student counter
Print the number of passes
Print the number of failures
If more than eight students passed
Print "Raise tuition"


```
1  // Fig. 4.11: Analysis.java
2  // Analysis of examination results.
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class Analysis {
8
9     // main method begins execution of Java application
10    public static void main( String args[] )
11    {
12       // initializing variables in declarations
13       int passes = 0,              // number of passes
14           failures = 0,            // number of failures
15           student = 1,             // student counter
16           result;                  // one exam result
17       String input,                // user-entered value
18              output;               // output string
19
20       // process 10 students; counter-controlled loop
21       while ( student <= 10 ) {
22
23          // obtain result from user
24          input = JOptionPane.showInputDialog(
25             "Enter result (1=pass,2=fail)" );
26
27          // convert result to int
28          result = Integer.parseInt( input );
29
30          // process result
31          if ( result == 1 )
32             passes = passes + 1;
33          else
34             failures = failures + 1;
```

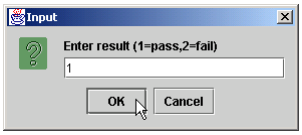
Analysis.java

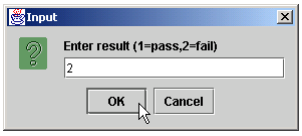
Line 21

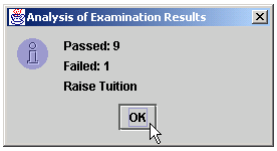
Line 31

```
35
36          student = student + 1;
37       }
38
39       // termination phase
40       output = "Passed: " + passes +
41          "\nFailed: " + failures;
42
43       if ( passes > 8 )
44          output = output + "\nRaise Tuition";
45
46       JOptionPane.showMessageDialog( null, output,
47          "Analysis of Examination Results",
48          JOptionPane.INFORMATION_MESSAGE );
49
50       System.exit( 0 );  // terminate application
51
52    }  // end method main
53
54 }  // end class Analysis
```

Analysis.java







17

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Fig. 4.12 Arithmetic assignment operators.

18

```
1  // Fig. 4.14: Increment.java
2  // Preincrementing and postincrementing
3
4  public class Increment {
5
6     // main method begins execution of Java application
7     public static void main( String args[] )
8     {
9        int c;
10
11       c = 5;
12       System.out.println( c );    // print 5
13       System.out.println( c++ ); // print 5 then postincrement
14       System.out.println( c );    // print 6
15
16       System.out.println();       // skip a line
17
18       c = 5;
19       System.out.println( c );    // print 5
20       System.out.println( ++c );  // preincrement then print 6
21       System.out.println( c );    // print 6
22
23    }  // end method main
24
25 }  // end class Increment
```

Line 13 postincrements `c`

Line 20 preincrements `c`

```
5
5
6

5
6
6
```

`Increment.java`

Line 13 postincrement

Line 20 preincrement

19

Operators	Associativity	Type
`()`	left to right	parentheses
`++ --`	right to left	unary postfix
`++ -- + - (type)`	right to left	unary
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`?:`	right to left	conditional
`= += -= *= /= %=`	right to left	assignment

Fig. 4.15 Precedence and associativity of the operators discussed so far.

20

Primitive Data Types

- Primitive types
  - "building blocks" for more complicated types
- Java is strongly typed
  - All variables in a Java program must have a type
- Java primitive types
  - portable across computer platforms that support Java

21

Type	Size in bits	Values	Standard
`boolean`	8	`true` or `false`	
`char`	16	`'\u0000'` to `'\uFFFF'` (0 to 65535) ➔ unsigned !!!	(ISO Unicode character set)
`byte`	8	–128 to +127 (-2^7 to $2^7 - 1$)	
`short`	16	–32,768 to +32,767 (-2^{15} to $2^{15} - 1$)	
`int`	32	–2,147,483,648 to +2,147,483,647 (-2^{31} to $2^{31} - 1$)	
`long`	64	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 (-2^{63} to $2^{63} - 1$)	
`float`	32	*Negative range:* –3.4028234663852886E+38 to –1.40129846432481707e–45 *Positive range:* 1.40129846432481707e–45 to 3.4028234663852886E+38	(IEEE 754 floating point)
`double`	64	*Negative range:* –1.7976931348623157E+308 to –4.94065645841246544e–324 *Positive range:* 4.94065645841246544e–324 to 1.7976931348623157E+308	(IEEE 754 floating point)

Fig. 4.16 The Java primitive data types.

22

(Optional Case Study) Thinking About Objects: Identifying Class Attributes

- Classes have attributes (data)
  - Implemented in Java programs as variables
  - Attributes of real-world objects
    - Radio (object)
      - Station setting, volume setting, AM or FM (attributes)
- Identify attributes
  - Look for descriptive words and phrases in problem statement
  - Each identified word and phrase is a candidate attribute
    - e.g., "the elevator is moving"
      - "is moving" corresponds to **`boolean`** attribute **`moving`**
    - e.g., "the elevator takes five seconds to travel between floors"
      - corresponds to **`int`** attribute **`travelTime`**
        - **`int travelTime = 5;`** (in Java)

23

Class	Descriptive words and phrases
`ElevatorModel`	number of people in the simulation
`ElevatorShaft`	[no descriptive words or phrases]
`Elevator`	moving summoned current floor destination floor capacity of only one person five seconds to travel between floors
`Person`	unique waiting / moving current floor
`Floor`	first or second; capacity for only one person
`FloorButton`	pressed / reset
`ElevatorButton`	pressed / reset
`FloorDoor`	door closed / door open
`ElevatorDoor`	door closed / door open
`Bell`	[no descriptive words or phrases]
`Light`	illuminated / turned off

Fig. 4.17 Descriptive words and phrases from problem statement.

24

Identifying Class Attributes (cont.)

- UML class diagram
  - Class attributes are place in the middle compartment
  - Attributes are written language independently
    - e.g., attribute **open** of class **ElevatorDoor**
      - **open : Boolean = false**
    - May be coded in Java as
      - **boolean open = false;**

25

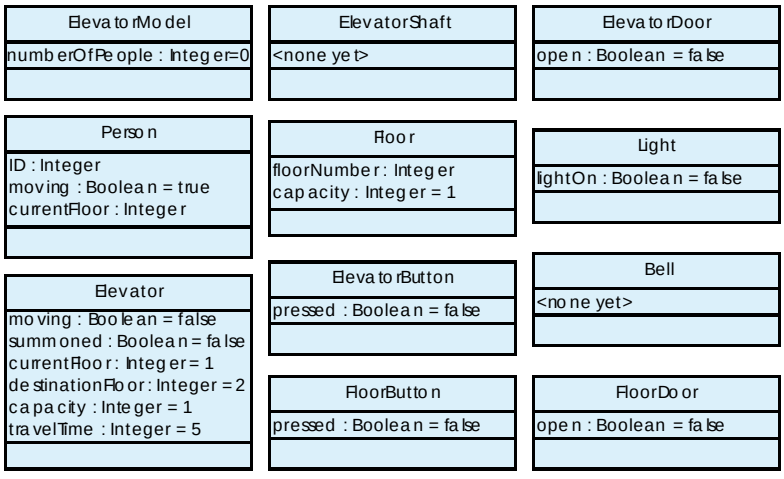


Fig. 4.18 Classes with attributes.

26