

3

Adding Structure

Self-review Questions

3.1 *Consider a television set. What abstractions does it present and how is encapsulation used?*

The way a television set works is complex. It has to receive a signal, decode it, create images on a frame-by-frame basis and display those images on the screen. With digital television, high definition pictures and new display technologies, the complexity is increasing. In addition, a television set is physically complex, containing many parts, some of which are dangerous due to requiring high voltages and getting very hot.

The user of a television is primarily interested in watching a TV programme and has no interest in monitoring how the television displays pictures and plays sound. The user just requires a limited number of basic abstractions or operations: turn on/off, select channel, raise/lower sound level. These are provided by a limited number of controls on the television, or more likely, on a remote control. The controls are typically buttons, a concept that everyone is familiar with and are easy to use. Hence, the complex operation of a television set is reduced to a small number of controls sufficient to let the typical user watch television with ease.

Of course, a common complaint about remote controls is that they provide too many controls, most of which the user does not need or does not understand. In addition, related devices such as video recorders are notoriously hard to program. However, these are faults of the design of the user interface, not of the principle of using abstraction to reduce the complexity of a device down to a manageable and (ideally) straightforward set of controls.

Encapsulation in the form of hard to open cases and cabinets is used to protect the user from physical harm. Cases also protect the insides of a television from physical damage.

3.2 *Which of the following are valid method names:*

`convert`, `2times`, `add one`, `add_two`, `time/space`, `AddUp`, `dRaawcIRClE`, `class`

`convert`, `add_two`, `AddUp` and `dRaawcIRClE` are all valid—they start with an alphabetic character and comprise only alphabetic and numeric character and the character `'_'` as required by the rules of Java.

`2times`, `add one`, `time/space` and `class` are all invalid. `2times` starts with a numeric, `add one` contains a space, `time/space` contains a `/`. `class` is a Java keyword and cannot be used as a method name.

3.3 *Can a method return more than one value at a time?*

No.

However, the value returned could be structured and so itself contain more than one value. This will arise in a later chapter.

3.4 What type checking is done on methods?

The statements in a method body are type checked to detect any type errors, such as trying to assign a value of one type to a variable of an incompatible type.

For a method call, the type of each of the parameter value is checked against the type of each parameter variable specified in the parameter list. For example, given a method declared like this:

```
int f(int i, String s) { ... }
```

the method call `f(1,"hello")` would be valid as the values `1` and `"hello"` match the types declared in the method's parameter list. In contrast, the method call `f("hello",1)` would be invalid as the values `1` and `"hello"` do not match those declared — even though there is one `String` and one `int` they are not in the correct order.

When a method returns after having been called it can return a value, the type of which is specified in the method declaration (the type before the method name). The type returned has to be compatible with what was expected where the method call was made. For example, in the statement:

```
int n = f(2,"world");
```

The value returned by the method call must be of type `int` so that it can be used to initialise the variable `n`. Type checking ensures that it is.

In most cases, type checking is performed by the Java compiler. If a type error is found, the code will not compile successfully and has to be corrected before it will.

3.5 Why are there no values of type void?

A type is defined by a set of values of the type and the operations that can be performed on those values. For example, for type `int` the values are the positive and negative integers that can be represented using 32 bits and the operations are `+`, `-`, `*`, `/`, etc.

`void` is the type with an empty set of values and, hence, no operations as there are no values to apply them to. In Java, `void` is the type used to denote that a method has no return value. It has no other use and cannot be used to declare a variable of type `void` as there are no values that can be represented and stored in a variable container.

3.6 Write a summary of the scope rules for compound statements.

A compound statement is a sequence of zero or more statements bounded by braces (curly brackets). Compound statements can be nested within each other to any depth. A method body is also a compound statement.

The scope rules are:

- a compound statement defines a local scope.
- a variable declared in a compound statement is a local variable, accessible within the compound statement and any nested compound statements.
- a local variable is valid from the point of its declaration to the end of the compound statement.
- the lifetime of a local variable is limited to the lifetime of the compound statement it is declared in.
- once a local variable is declared in a compound statement no other local variable with the same name can be declared in the same local scope, including any scopes declared by nested compound statements.

The last item is worth noting as it means that the following code is invalid:

```
...
{
    int n = 1;
    ...
    {
        int n = 2 ; // ERROR - the variable n is already declared in the enclosing scope.
    }
}
...
```

Unlike some programming languages (notably C and C++) Java does not allow a local variable to hide another local variable in an enclosing scope.

Local variables with the same name can be declared in disjoint scopes, as disjoint scopes have no overlap, so there is no ambiguity about which variable is being used in each scope.

3.7 What are the values of these expressions:

```
22 / 7
4 + 3 / 6.1
1.3 * 2.2
34 % 55
2.1 % 3
2147483647 + 1
```

We think the values should be:

3
 4.49180327868852459016 (to 20 decimal places)
 2.86
 34
 2.1
 -2147483648

but to make sure we weren't thinking anything silly, we wrote the program:

```
public class CheckExpressionValues {
    private void printValues ( ) {
        System.out.println ( "22 / 7 = " + ( 22 / 7 ) );
        System.out.println ( "4 + 3 / 6.1 = " + ( 4 + 3 / 6.1 ) );
        System.out.println ( "1.3 * 2.2 = " + ( 1.3 * 2.2 ) );
        System.out.println ( "34 % 55 = " + ( 34 % 55 ) );
        System.out.println ( "2.1 % 3 = " + ( 2.1 % 3 ) );
        System.out.println ( "2147483647 + 1 = " + ( 2147483647 + 1 ) );
    }
    public static void main ( final String[] args ) {
        ( new CheckExpressionValues ( ) ).printValues ( );
    }
}
```

and it generated the result:

```
22 / 7 = 3
4 + 3 / 6.1 = 4.491803278688525
1.3 * 2.2 = 2.8600000000000003
34 % 55 = 34
2.1 % 3 = 2.1
2147483647 + 1 = -2147483648
```

The apparent discrepancy in the second value is an issue of display. The value is not a finite fraction and Java is only displaying 16 significant digits whereas we showed 20 decimal places in the earlier list.

The definite error in the Java output for the third value shows clearly the problems of rounding errors in the double arithmetic implemented by computers. If we had restricted the number of displayed decimal digits to say 6, we would not have seen this.

The apparent bizarreness of the final value shows that integer arithmetic on computers is finite. We are seeing here that integers 'wrap around'—overflow in the jargon— so that adding two numbers that result in a value too large to represent in the computer results in an unexpected value.

3.8 *Is the method times2 (see page 64) referentially transparent?*

Yes. No matter how many time you call the method with the same parameter, it gives the same result.

3.9 *Devise a test plan for a modified version of program MultiplicationTable (see page 68) that allows any multiplication table from 1 to 100,000,000 to be displayed.*

A test plan consists of a collection of test cases where each case should provide the following information:

- the purpose of the test, what error(s) it is looking for,
- a procedure for carrying out the test,
- the input data,
- the expected output.

Exhaustive testing of all tables the program could generate is not possible. To be anywhere near manageable we must select a very limited number of tables where we believe there is a chance that errors can be found and then create tests based on those tables. If the tests fail to find errors in those cases we gain confidence that the program works well enough but cannot say with absolute certainty that it contains no errors.

Testing a table means running the program, inputting the number of the table and then comparing the output table with the expected table (which has to be generated some other way). A range of tables to test would be:

- the first table, which would be for 1.
- the last table, for 100,000,000.
- for boundary conditions such as tables 9, 10, 11 or 999, 1000, 1001.
- for boundary conditions where arithmetic overflow might occur (a multiplication results in a number that cannot be represented in the integer representation chosen).

Also test cases could be added for tables outside the range (less than 1, including negative numbers or greater than 100,000,000) to confirm that the program displays the correct error message.

3.10 *Execute program ConvertBinaryToDecimal by hand to convert the binary numbers 1001, 11100101 and 1000101010001011 to decimal.*

Learning to execute a program by hand (or in your head) is an essential skill for a programmer. A programmer needs to be able to read through code, especially more complex code using

structures like loops, and have a clear and accurate idea of what the code does. If code is not working correctly manual execution is an important technique for tracking down the bugs and confirming that a proposed bug-fix actually works.

For simple code or code with a straightforward linear sequence it is possible to just read the code, following the flow of control. For more complex code, loops, arithmetic expressions, parameterised method calls, arrays and data structures, you will need to make notes as you go in order to keep track of the details and, especially, the values of variables.

A common technique for manually executing code is to create a table of the values of variables. Each row shows the result of executing one or more statements, with the variable values updated as necessary. Often the table is drawn up using pen and paper, but alternatively a spreadsheet can be used with the advantage that it is easy to edit. To illustrate the idea the `String` based `ConvertBinaryToDecimal` program will be used to convert 1001.

Starting with the `main` method the flow of control is initially linear and can be understood just by reading through the code. The `doOutput` method is called with the result returned by a call to `convertBinaryToDecimal` passed as a parameter. As `doOutput` simply consists of a print statement it is obvious what happens and we can focus on the `convertBinaryToDecimal` method. `convertBinaryToDecimal` takes as an argument a `String` input from the user by a call to `getBinaryNumberString`. Again, however, `getBinaryNumberString` can be understood by just quickly reading the method body, so doesn't require detailed attention. With the basic flow of control through the program clear, we can now focus on using a table to see how `convertBinaryToDecimal` works.

<i>comment</i>	value	power	position	position > -1	charAt(position)
start of method	0	1			
start of first iteration	0	1	3	true	'1'
case '1'	1	1	3	true	'1'
start of second iteration	1	2	2	true	'0'
case '0'	1	2	2	true	'0'
start of third iteration	1	4	1	true	'0'
case '0'	1	4	1	true	'0'
start of fourth iteration	1	8	0	true	'1'
case '1'	9	8	0	true	'1'
start of fifth iteration	9	8	-1	false	
loop terminates, return value	9				

As can be seen the first column of the table is a comment stating what has just been executed. The rest of the columns are either the values of relevant variables or the values of expressions. In this case, the two expressions `position > -1` and `charAt(position)` are most helpful in keeping track of what is happening. Each line of the table corresponds to a useful step in the algorithm, not just a single statement. Exactly the same process can be used to show the execution of 11100101 and 1000101010001011, although 1001 is really enough to gain an understanding of what is going on.

An alternative to tracking the execution by hand would be to use a debugging tool and go

through the execution step-by-step. The standard Java tools include a debugger called jdb that can be run from the command line. However, it is far better to use the debugger in an integrated programming environment like BlueJ, Netbeans, IDEA or Eclipse. The command line jdb is at best awkward to use and typically more or less impossible!

3.11 Convert this for loop to a while loop:

```
for ( int i = 10 ; i > -2 ; i -= 2 ) {
    ...
}
```

A for loop can be considered as “syntactic sugar” for a while loop, meaning that it is simply a better way of writing the same thing that is easier to understand in some situations. Any for loop can be translated into a while loop and vice versa following a straightforward set of rules.

The for loop structure:

```
for ( initialisation ; boolean expression ; end of loop expression ) { loop body }
```

maps to the while loop structure:

```
initialisation
while ( boolean expression ) {
    loop body
    end of loop expression
}
```

The while equivalent of the given for loop is:

```
int i = 10 ;
while ( i > -2 ) {
    ...
    i -= 2 ;
}
```

Programming Exercises

3.1 Write methods to do the following:

- Convert from feet to centimetres.
- Convert from yards to metres.
- Convert from miles to kilometres.

Include the methods in an interactive program that lets the user select which conversion to perform.

```

public class E_3_1 {
    private double feetToCentimetre ( final double x ) { return x * 30.48 ; }
    private double yardsToMetres ( final double x ) { return x * 0.9144 ; }
    private double milesToKilometers ( final double x ) { return x * 1.609 ; }
    private void processConversion ( ) {
        final Input input = new Input ( ) ;
        while ( true ) {
            System.out.print ( "\nConversion Program\n 1 -- Feet to centimetres\n 2 -- Yards to metres\n 3 -- Miles to kilometres\nSelect Conversion: " ) ;
            final int selection = input.nextInt ( ) ;
            if ( ( selection < 1 ) || ( selection > 3 ) ) {
                System.out.println ( "Selection not a known conversion, stopping." ) ;
                System.exit ( 0 ) ;
            }
            System.out.print ( "Value to convert: " ) ;
            final double value = input.nextDouble ( ) ;
            double result = 0.0 ;
            switch ( selection ) {
                case 1 : result = feetToCentimetre ( value ) ; break ;
                case 2 : result = yardsToMetres ( value ) ; break ;
                case 3 : result = milesToKilometers ( value ) ; break ;
            }
            System.out.println ( "Converted value: " + result ) ;
        }
    }
    public static void main ( final String[] args ) {
        new E_3_1 ( ) .processConversion ( ) ;
    }
}

```

3.2 Write a program that counts the number of times a specified character appears in a line of text.

This exercise is about using a loop to access each of the characters in the string in turn and if it is the same as the specified character increment a count. In this answer we take the opportunity to use a switch statement to make the output wording appropriate for the resulting count.

```

public class E_3_2 {
    private void processLine ( ) {
        final Input input = new Input ( ) ;
        System.out.print ( "Which letter to count : " ) ;
        final char c = input.nextChar ( ) ; input.nextLine ( ) ;
        System.out.print ( "Enter text to scan : " ) ;
        final String s = input.nextLine ( ) ;
        int count = 0 ;
        for ( int i = 0 ; i < s.length ( ) ; ++i ) {
            if ( s.charAt ( i ) == c ) { ++count ; }
        }
        final StringBuilder sb = new StringBuilder ( ) ;
        sb.append ( "There " ) ;
        switch ( count ) {
            case 0 :
                sb.append ( "were no " + c + "s " ) ;
                break ;
            case 1 :

```

```

        sb.append ( "was 1 " + c + " " );
        break ;
    default :
        sb.append ( "were " + count + " " + c + "s " );
        break ;
    }
    sb.append ( "in the text." );
    System.out.println ( sb );
}
public static void main ( final String[] args ) {
    new E_3_2 ( ) .processLine ( ) ;
}
}

```

3.3 Write a program to read in a decimal integer and print out the binary equivalent.

This exercise is really about reading the manual page for `Integer` or `Long` and spotting that there is a `toString` method that can do all the work for us.

```

public class E_3_3 {
    private void processValue ( ) {
        final Input input = new Input ( ) ;
        System.out.print ( "Enter value to convert : " );
        final long value = input.nextLong ( ) ;
        System.out.println ( "Decimal value " + value + " in binary is " + Long.toString ( value , 2 ) ) ;
    }
    public static void main ( final String[] args ) {
        new E_3_3 ( ) .processValue ( ) ;
    }
}

```

3.4 Write a program that uses methods to display the following:

```

    *
   **
  ***
 ****
*****

```

writing a single character at a time.

A simple program to display this shape works by iterating through each row and using two nested loops to display the correct number of spaces and stars. The number of spaces and stars to display is determined directly from the row number. This gives the following program:

```

public class E_3_4 {
    private void drawTriangle ( ) {
        for ( int row = 0 ; row < 6 ; ++row ) {
            for ( int spaces = 0 ; spaces < 6 - row - 1 ; ++spaces ) { System.out.print ( ' ' ) ; }
            for ( int stars = 0 ; stars < row + 1 ; ++stars ) { System.out.print ( '*' ) ; }
        }
    }
}

```

```

        System.out.println ( );
    }
}
public static void main ( final String[] args ) {
    new E_3_4 ( ) .drawTriangle ( );
}
}

```

While this answers the specific question the program is limited to displaying triangles of one size only. A more satisfying approach would be to pass the height (number of rows) of the triangle as a method parameter, giving this:

```

public class E_3_4b {
    private void drawTriangle ( final int height ) {
        for ( int row = 0 ; row < height ; ++row ) {
            for ( int spaces = 0 ; spaces < height - row - 1 ; ++spaces ) { System.out.print ( ' ' ) ; }
            for ( int stars = 0 ; stars < row + 1 ; ++stars ) { System.out.print ( '*' ) ; }
            System.out.println ( ) ;
        }
    }
    public static void main ( final String[] args ) {
        new E_3_4b ( ) .drawTriangle ( 6 ) ;
    }
}

```

The program still contains two nested for loops, which are essentially doing the same thing — outputting zero or more characters. This is an example of duplication of code and it is always good design practice to eliminate duplication. Hence a further revision is:

```

public class E_3_4c {
    private void displayChars ( final char character , final int count ) {
        for ( int n = 0 ; n < count ; ++n ) { System.out.print ( character ) ; }
    }
    private void drawTriangle ( final int height ) {
        for ( int row = 0 ; row < height ; ++row ) {
            displayChars( ' ', height - row - 1 ) ;
            displayChars( '*', row + 1 ) ;
            System.out.println ( ) ;
        }
    }
    public static void main ( final String[] args ) {
        new E_3_4c ( ) .drawTriangle ( 6 ) ;
    }
}

```

This is a good example of finding duplication by looking for the same pattern of code rather than simply identical code.

In contrast to the programs above, here is a further answer that uses recursion rather than for loops to provide the repetition.

```

public class E_3_4d {

```

```

private void displayChars ( final char character , final int count ) {
    if ( count < 1 ) return ;
    System.out.print ( character ) ;
    displayChars ( character , count - 1 ) ;
}
private void displayTriangle ( final int spaces , final int stars ) {
    if ( spaces < 0 ) return ;
    displayChars( ' ', spaces ) ;
    displayChars( '*', stars ) ;
    System.out.println ( ) ;
    displayTriangle ( spaces - 1 , stars + 1 ) ;
}
private void drawTriangle ( final int height ) {
    displayTriangle ( height - 1 , 1 ) ;
}
public static void main ( final String[] args ) {
    new E_3_4d ( ) .drawTriangle ( 6 ) ;
}
}

```

The `drawTriangle` method sets up the initial call to the recursive `displayTriangle` method. `displayTriangle` takes two parameters giving the number of spaces and stars to output. If the number of spaces is less than zero the method returns ending the sequence of recursive calls, otherwise the required number of spaces and stars are output, and the method recursively calls itself with updated parameter values. The recursive `displayChars` method displays a sequence of zero or more characters.

3.5 Write a program, using methods, that displays triangles of the following form:

```

*
***
*****
*****

```

The user should input how many lines to display. You may only display one character at a time. Use the method in a test program to display triangles of various sizes.

There are a number of strategies for displaying shapes made up from characters. This answer uses two variables to keep a count of how many spaces and stars to display on each line. At the start of the `drawTriangle` method the variables are initialised to one star and the required number of spaces. The number of spaces is determined by the height of the triangle, which is passed as a parameter. For a given height, the width of the base of the triangle is $(2 * height) - 1$, so the number of spaces on the first line is $width/2$ (remember this will do integer division). A loop is used to count through each line, the spaces and stars are output, and the count variables are updated at the end of each iteration.

The rest of the program simply deals with the input and calling the `drawTriangle` method.

```

public class E_3_5 {
    private void drawTriangle ( final int height ) {

```

```

final int width = 2 * height - 1 ;
int spaces = width / 2 ;
int stars = 1 ;
for ( int row = 0 ; row < height ; ++row ) {
    for ( int space = 0 ; space < spaces ; ++space ) { System.out.print ( ' ' ) ; }
    for ( int star = 0 ; star < stars ; ++star ) { System.out.print ( '*' ) ; }
    System.out.println ( ) ;
    --spaces ;
    stars += 2 ;
}
}
private int inputHeight ( ) {
    final Input in = new Input ( ) ;
    System.out.print ( "Enter height of triangle: " ) ;
    return in.nextInt ( ) ;
}
public void inputAndDraw ( ) {
    final int height = inputHeight ( ) ;
    drawTriangle ( height ) ;
}
public static void main ( final String[] args ) {
    new E_3_5 ( ).inputAndDraw ( ) ;
}
}

```

3.6 Write a method to display rectangles with the following form:

```

*****
*****
*****
*****

```

The method parameters should give the number of rows and columns of the rectangle. Use the method in a test program to display various rectangles of different sizes.

A straightforward way to display a rectangle shape is to use two loops, one nested inside the other. The outer loop counts through each row, while the inner loop counts through each column in each row, displaying a star at each iteration.

```

public class E_3_6 {
    private void drawRectangle ( final int rows , final int columns ) {
        for ( int row = 0 ; row < rows ; row++ ) {
            for ( int column = 0 ; column < columns ; column++ ) { System.out.print ( '*' ) ; }
            System.out.println ( ) ;
        }
    }
    public static void main ( final String[] args ) {
        new E_3_6 ( ).drawRectangle ( 5 , 4 ) ;
        new E_3_6 ( ).drawRectangle ( 1 , 1 ) ;
        new E_3_6 ( ).drawRectangle ( 3 , 7 ) ;
        new E_3_6 ( ).drawRectangle ( 1 , 4 ) ;
        new E_3_6 ( ).drawRectangle ( 5 , 1 ) ;
        new E_3_6 ( ).drawRectangle ( 3 , 0 ) ;
        new E_3_6 ( ).drawRectangle ( 0 , 3 ) ;
    }
}

```

```

    new E_3_6 ( ) .drawRectangle ( -2 , 3 );
    new E_3_6 ( ) .drawRectangle ( 3 , 3 );
}
}

```

The main method demonstrates a number of calls of the `drawRectangle` method that can be made to informally test that the right output is displayed. Note that displaying a rectangle of three rows and zero columns, simply outputs three newlines, which seems reasonable on reflection. Trying to display a rectangle with -2 rows does nothing as the outer loop terminates immediately without the loop body being executed.

3.7 Write a program using methods to display your name, or any other message, in the middle of a line 80 characters wide.

Arguably the most straightforward way of programming this is to output the right number of spaces and then the message. Calculating the correct number of spaces to output is relatively straightforward – subtracting the message length from 80 says how many of the total characters on the line are spaces and half of those need to go before the message:

```

public class E_3_7 {
    private String getInput ( ) {
        System.out.print ( "Enter the string to display: " );
        return ( new Input ( ) ).nextLine ( ) ;
    }
    private void writeLine ( final String input ) {
        for ( int i = 0 ; i < ( 80 - input.length ( ) ) / 2 ; ++i ) { System.out.print ( ' ' ) ; }
        System.out.println ( input ) ;
    }
    public static void main ( final String[] args ) {
        final E_3_7 application = new E_3_7 ( ) ;
        application.writeLine ( application.getInput ( ) ) ;
    }
}

```

3.8 Modify program `ConvertBinaryToDecimal` to create a program, called `ConvertOctalToDecimal`, that converts from base 8 numbers to base 10 numbers.

The most straightforward answer to this question is a modified version of the first convert binary to decimal program presented in chapter 3. This relies on the `nextInt(final int radix)` method in `class Input` to convert from binary to decimal. By specifying the `radix` as 8 (octal) the `nextInt` method will do octal to binary conversion. This gives the following program:

```

/**
 * Program to convert a <code>String</code> representing an octal number to an <code>int</code> to
 * be output in decimal.
 *
 * @author Russel Winder

```

```

* @version 2006-07-29
*/
public class ConvertOctalToDecimal_1 {
    private void doInputConvertAndOutput () {
        final Input in = new Input ();
        System.out.print ( "Enter octal number: " );
        final int result = in.nextInt ( 8 );
        System.out.println ( "Value as decimal is: " + result );
    }
    public static void main ( final String[] args ) {
        ConvertOctalToDecimal_1 theObject = new ConvertOctalToDecimal_1 ();
        theObject.doInputConvertAndOutput ();
    }
}

```

In production code we would normally use a Java class library method to do number base conversions, as we can assume that such a method is fully debugged and reliable so there is no need to spend time re-inventing it. However, for these example answers it is more instructive to write our own convert octal to decimal method to understand how it can be done. The next program, therefore, presents a modified version of the second `ConvertBinaryToDecimal` program that converts an octal integer represented as a `String` into a decimal value.

Following the strategy of the binary version, the conversion is done by multiplying each digit with the power of 8 corresponding to the position of the digit in the input number. For example:

$$\begin{aligned}
 123_8 &= 1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0 \\
 &= 64 + 16 + 3 \\
 &= 83_{10}
 \end{aligned}$$

```

/**
 * Program to convert a <code>String</code> representing a octal number to an <code>int</code> to
 * be output in decimal.
 *
 * @author Graham Roberts and Russel Winder
 * @version 2006-07-20
 */
public class ConvertOctalToDecimal_2 {
    private String getOctalNumberString () {
        final Input in = new Input ();
        System.out.print ( "Enter octal number: " );
        return in.nextLine ();
    }
}
/**
 * Attempt to convert the string argument representing an octal number to a decimal integer.
 * If any invalid characters are found in the string stop and return the partially converted
 * value.
 */
public int convertOctalToDecimal ( final String s ) {
    int value = 0 ;
    int power = 1 ;
    for ( int position = s.length () - 1 ; position > -1 ; --position , power *= 8 ) {

```

```

switch ( s.charAt ( position ) ) {
    case '0' :
        break ;
    case '1' :
        value += power ;
        break ;
    case '2' :
        value += 2 * power ;
        break ;
    case '3' :
        value += 3 * power ;
        break ;
    case '4' :
        value += 4 * power ;
        break ;
    case '5' :
        value += 5 * power ;
        break ;
    case '6' :
        value += 6 * power ;
        break ;
    case '7' :
        value += 7 * power ;
        break ;
    case '-' :
        if ( position == 0 ) { value = -value ; break ; }
        else { return value ; }
    default :
        return value ;
}
}
return value ;
}
private void doOutput ( final int result ) {
    System.out.println ( "Value as decimal is: " + result ) ;
}
public static void main ( final String[] args ) {
    ConvertOctalToDecimal_2 object = new ConvertOctalToDecimal_2 ( ) ;
    object.doOutput (
        object.convertOctalToDecimal (
            object.getOctalNumberString ( )
        )
    ) ;
}
}
}

```

When reviewing the program above, the duplication of the pattern of the code in the cases in the switch statement stands out. Duplicated code or patterns of code should always be removed, so the program is modified to give the following version.

```

/**
 * Program to convert a <code>String</code> representing an octal number to an <code>int</code> to
 * be output in decimal.
 *
 * @author Graham Roberts and Russel Winder
 * @version 2006-09-04

```

```

*/
public class ConvertOctalToDecimal_3 {
    private String getOctalNumberString ( ) {
        final Input in = new Input ( );
        System.out.print ( "Enter octal number: " );
        return in.nextLine ( );
    }
    /**
     * Attempt to convert the string argument representing an octal number to a decimal integer.
     * If any invalid characters are found in the string stop and return the partially converted
     * value.
     */
    public int convertOctalToDecimal ( final String s ) {
        int value = 0 ;
        int power = 1 ;
        for ( int position = s.length ( ) - 1 ; position > -1 ; --position , power *= 8 ) {
            char digit = s.charAt ( position ) ;
            switch ( digit ) {
                case '0' :
                case '1' :
                case '2' :
                case '3' :
                case '4' :
                case '5' :
                case '6' :
                case '7' :
                    value += Character.digit(digit,10) * power ;
                    break ;
                case '-':
                    if ( position == 0 ) { value = -value ; break ; }
                    else { return value ; }
                default :
                    return value ;
            }
        }
        return value ;
    }
    private void doOutput ( final int result ) {
        System.out.println ( "Value as decimal is: " + result ) ;
    }
    public static void main ( final String[] args ) {
        ConvertOctalToDecimal_3 object = new ConvertOctalToDecimal_3 ( ) ;
        object.doOutput (
            object.convertOctalToDecimal (
                object.getOctalNumberString ( )
            )
        ) ;
    }
}

```

As a further exercise, replace the switch statement in the code above with one or more if statements.

3.9 Write a program that uses a recursive method to calculate the product of a sequence of numbers specified by

the user. For example, if the user specifies 4 to 8, the method calculates $4*5*6*7*8$.

In the following we have a recursive method `doCalculation` that is an answer to the question. It is worth noting that in the recursion we change both the start and end point values so we have to be careful about how we terminate the recursion – it is important to deal with both termination cases, `start == end` and `start < end`:

```
public class E_3_9 {
    private int getIntegerValue ( final String prompt ) {
        System.out.print ( prompt );
        return ( new Input ( ) ).nextInt ( );
    }
    private int doCalculation ( final int start , final int end ) {
        if ( start > end ) { return 1 ; }
        else if ( start == end ) { return start * end ; }
        else { return start * end * doCalculation ( start + 1 , end - 1 ) ; }
    }
    public static void main ( final String[] args ) {
        final E_3_9 application = new E_3_9 ( );
        System.out.println ( "Result is : " +
            application.doCalculation (
                application.getIntegerValue ( "Enter start number: " ) ,
                application.getIntegerValue ( "Enter end number: " )
            )
        );
    }
}
```

There is a ‘bug’ with this code of course. If the user enters a start number that is larger than the end number the program writes 1 whereas it should either:

1. Do the calculation as though the start and end numbers were swapped.
2. Put out an error message.

This doesn’t affect the `doCalculation` method, it just means we need to change the way we call it initially: we need to do more work before calling the `doCalculation` method.

3.10 Write a program that reads in a line of text and displays the number of characters and words it contains. Spaces and tabs should not be counted as characters.

Lots of questions to answer with this question before we can write a program. The most obvious of which is ‘What is a word?’

If we assume that words are sequences of characters that are not spaces or tabs, then we can write:

```
public class E_3_10 {
```

```

private String getInput () {
    System.out.print ( "Enter a line of text: " );
    return ( new Input () ).nextLine ();
}
private int skipSpace ( final String line , int i ) {
    for ( char c ; ( i < line.length () ) && ( ( c = line.charAt ( i ) ) == ' ' ) || ( c == '\t' ) ) ; ++i ) {}
    return i ;
}
private void process ( final String line ) {
    int wordCount = 0 ;
    int characterCount = 0 ;
    if ( line.length () > 0 ) {
        int i = skipSpace ( line , 0 ) ;
        if ( i < line.length () ) {
            do {
                final char c = line.charAt ( i ) ;
                if ( ( c == ' ' ) || ( c == '\t' ) ) {
                    i = skipSpace ( line , i ) ;
                    ++wordCount ;
                }
                else {
                    ++i ;
                    ++characterCount ;
                }
            } while ( i < line.length () ) ;
            ++wordCount ;
        }
    }
    System.out.println ( "Number of characters is " + characterCount ) ;
    System.out.println ( "Number of words is " + wordCount ) ;
}
public static void main ( final String[] args ) {
    final E_3_10 application = new E_3_10 () ;
    application.process ( application.getInput () ) ;
}
}

```

as one of the many possible solutions. If, however, we want to take into account that punctuation symbols separate words then we have to be a little bit more sophisticated:

```

public class E_3_10_a {
    private String getInput () {
        System.out.print ( "Enter a line of text: " );
        return ( new Input () ).nextLine ();
    }
    private int skipSpace ( final String line , int i ) {
        for ( char c ; ( i < line.length () ) && ( ( c = line.charAt ( i ) ) == ' ' ) || ( c == '\t' ) ) ; ++i ) {}
        return i ;
    }
    private void process ( final String line ) {
        int wordCount = 0 ;
        int characterCount = 0 ;
        if ( line.length () > 0 ) {
            int i = skipSpace ( line , 0 ) ;
            boolean inWord = false ;
            while ( i < line.length () ) {

```

```

final char c = line.charAt ( i );
switch ( c ) {
  case ' ':
  case '\t':
    i = skipSpace ( line , i );
    ++wordCount ;
    inWord = false ;
    break ;
  case ' ':
  case ' ':
  case ' ':
  case ' ':
  case ' ':
    if ( inWord ) {
      ++wordCount ;
      inWord = false ;
    }
    /* FALLTHROUGH */
  default :
    ++ i ;
    ++characterCount ;
    break ;
}
}
if ( inWord ) { ++wordCount ; }
}
System.out.println ( "Number of characters is " + characterCount ) ;
System.out.println ( "Number of words is " + wordCount ) ;
}
public static void main ( final String[] args ) {
  final E_3_10_a application = new E_3_10_a ( ) ;
  application.process ( application.getInput ( ) ) ;
}
}

```

3.11 Consider displaying a large letter formed from stars:

```

*   *
*   *
*****
*   *
*   *

```

Write a method that displays one line of the large character H, where the line to display is given as a parameter (e.g. `bigH(3)` would display the third line).

Then write a program to display six large H's in a row, with one space between each H:

```

*   *   *   *   *   *   *
*   *   *   *   *   *   *
***** ***** ***** ***** *****
*   *   *   *   *   *   *
*   *   *   *   *   *   *

```

The core of the answer to this question is a method that can display one slice or line of a large H at a time — the `bigH` method suggested in the question. This can then be used by another

method to display a row of large H's, where each line output is a sequence of the same slice of a large H separated by a space. The following program gives a basic working example:

```
public class E_3_11 {
    private void bigH ( final int line ) {
        switch ( line ) {
            case 0 :
            case 1 :
            case 3 :
            case 4 :
                System.out.print ( '*' );
                for ( int spaces = 0 ; spaces < 4 ; ++spaces ) { System.out.print ( ' ' ); }
                System.out.print ( '*' );
                break ;
            case 2 :
                for ( int stars = 0 ; stars < 6 ; ++stars ) { System.out.print ( '*' ); }
                break ;
            default ; ;
        }
    }
    private void displayRowOfBigH ( ) {
        for ( int line = 0 ; line < 5 ; ++line ) {
            for ( int count = 0 ; count < 6 ; ++count ) {
                bigH ( line );
                System.out.print ( ' ' );
            }
            System.out.println ( );
        }
    }
    public static void main ( final String[] args ) {
        final E_3_11 application = new E_3_11 ( );
        application.displayRowOfBigH ( );
    }
}
```

There is room for improvement in the code shown above. Duplication can be removed and literal values ('magic numbers') can be replaced by method parameters, to make the code more general purpose — for example, to easily change the size and number of large H characters displayed.

```
public class E_3_11a {
    private void displayChars ( final char character , final int count ) {
        for ( int n = 0 ; n < count ; ++n ) { System.out.print ( character ); }
    }
    private void bigH ( final char character , final int height ,
        final int width , final int line ) {
        final int middle = height / 2 ;
        if ( line == middle ) { displayChars ( character , width ); }
        else {
            displayChars ( character , 1 );
            displayChars ( ' ' , width - 2 );
            displayChars ( character , 1 );
        }
    }
    private void displayRowOfBigH ( final char character , final int height ,
        final int width , final int count , final int gap ) {
        for ( int line = 0 ; line < height ; ++line ) {
```

```

    for ( int column = 0 ; column < count ; ++column ) {
        bigH ( character , height , width , line );
        displayChars ( ' ', gap );
    }
    displayChars ( '\n', 1 );
}
}
}
public static void main ( final String[] args ) {
    final E_3_11a application = new E_3_11a ( );
    application.displayRowOfBigH ( '*', 5 , 5 , 6 , 1 );
    application.displayRowOfBigH ( '@', 8 , 6 , 6 , 2 );
    application.displayRowOfBigH ( '^', 11 , 5 , 8 , 3 );
}
}
}

```

In this version, method parameters have been introduced to specify the width and height of a large H, along with the character used to display it. The number of spaces between each large H can also be set. The `bigH` method has been modified to display any size H (assuming reasonable parameter values are supplied — no checking is done for negative or very large numbers). The `main` method shows some examples of how the `displayRowOfBigH` method can be called. This results in the following output:

```

* * * * *
* * * * *
*****
* * * * *
* * * * *
@ @ @ @ @ @ @ @ @ @ @
@ @ @ @ @ @ @ @ @ @ @
@ @ @ @ @ @ @ @ @ @ @
@ @ @ @ @ @ @ @ @ @ @
@@@@@ @@@@@ @@@@@ @@@@@ @@@@@ @@@@@
@ @ @ @ @ @ @ @ @ @ @ @ @
@ @ @ @ @ @ @ @ @ @ @ @ @
@ @ @ @ @ @ @ @ @ @ @ @ @
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^

```

Note that a method called `displayChars` has been introduced to do the work of actually outputting characters, so that it is the only place in the program to contain a `System.out.print` statement. A further refinement of the program could replace the `System.out.print` statement with code to create a `String`, write to a file or some other way of capturing the output for future use. The strategy behind adding the `displayChars` method is one of reducing the dependency of code on specific resources such as direct output to the screen via `System.out`.

3.12 Write a method to test if an integer of type `long` is a prime number. The method should return a `boolean`

value. Test your method by writing a test program that reads an integer typed at the keyboard and states whether the integer was prime. Using your prime method, write a program that finds all the prime numbers that can be represented by an integer of type long.

A prime number is a positive integer that has two divisors only, the number 1 and the prime number itself. The basic test for a prime number is to divide a candidate number by all the integers from 2 up to the square root of the candidate number. If any divisions give a remainder of 0 then the number is not prime. The following method performs this test:

```
public boolean isPrime ( final long n ) {
    if ( n < 2L ) { return false ; }
    if ( n == 2L ) { return true ; }
    if (( n % 2L ) == 0 ) { return false ; }
    long divisor = 3L ;
    final long limit = new Double( Math.sqrt ( n ) ).longValue ( ) ;
    while ( divisor <= limit ) {
        if ((n % divisor) == 0 ) { return false ; }
        divisor += 2L ;
    }
    return true ;
}
```

Several small optimisations have been made. Firstly any even number is immediately rejected by checking to see if dividing by two leaves a remainder of zero. Second, the search performed by the while loop only tries to divide using odd numbers, as any multiple of an even number must be an even number and hence not prime.

The method above performs a brute force test. There are a number of much more efficient ways of testing if a number is prime but they require the use of data structures not covered until a later chapter.

Using the isPrime method a program can be written to let the user type in a number and check whether it is prime or not.

```
public class CheckForPrime {
    public boolean isPrime ( final long n ) {
        if ( n < 2L ) { return false ; }
        if ( n == 2L ) { return true ; }
        if (( n % 2L ) == 0 ) { return false ; }
        long divisor = 3L ;
        final long limit = new Double( Math.sqrt ( n ) ).longValue ( ) ;
        while ( divisor <= limit ) {
            if ((n % divisor) == 0 ) { return false ; }
            divisor += 2L ;
        }
        return true ;
    }
    public void checkPrime ( ) {
        Input input = new Input ( ) ;
        System.out.print( "Enter a number: " ) ;
        long candidate = input.nextLong ( ) ;
    }
}
```

```

    if ( isPrime ( candidate ) ) { System.out.println ( candidate + " is a prime number!" ); }
    else { System.out.println ( candidate + " is not a prime number." ); }
}
public static void main ( final String[] args ) {
    new CheckForPrime ( ).checkPrime ( );
}
}

```

The following program will find all the prime numbers that can be represented by a long but will take some time to run!

```

public class FindAllLongPrimes {
    public boolean isPrime ( final long n ) {
        if ( n < 2L ) { return false ; }
        if ( n == 2L ) { return true ; }
        if ( ( n % 2L ) == 0 ) { return false ; }
        long divisor = 3L ;
        final long limit = new Double( Math.sqrt ( n ) ).longValue ( ) ;
        while ( divisor <= limit ) {
            if ( ( n % divisor ) == 0 ) { return false ; }
            divisor += 2L ;
        }
        return true ;
    }
    public void findPrimes ( ) {
        long count=0L;
        for ( long candidate = 3L ; candidate < Long.MAX_VALUE ; candidate++ ) {
            if ( isPrime ( candidate ) ) {
                System.out.println ( candidate + " is Prime" );
            }
        }
    }
    public static void main ( final String[] args ) {
        new FindAllLongPrimes ( ).findPrimes ( ) ;
    }
}

```

See http://en.wikipedia.org/wiki/Prime_number For a starting point to find out more about prime numbers.

- 3.13** Write a program that acts as a simple desktop calculator, which allows you to type in a sequence of values and operators and displays the results of your calculation. For example, the following might be typed in (<return> indicates the return key being pressed):

```

5 <return>
+ <return>
6 <return>
= <return>
11

```


Hint: Write methods to:

- Take a single digit and return a string between 'zero' and 'nine'.
- Take a number between 10 and 19 and return a string between 'ten' and 'nineteen'.
- Take a number between 0 and 99 and, using the other methods as necessary, 'verbalize' the number.