

# 2

## *Programming Fundamentals*

## Self-review Questions

### 2.1 Explain the meaning of `piece.forward()`. Why is the semicolon present?

The object referred to by `piece` is instructed to move forward. Actually, to be more precise: the method `forward` is called for the object referenced by `piece`. The actual movement made by the piece depends on how it interprets an invocation of the method `forward`—in the examples given in the chapter a piece will move one square forward, but different kinds of piece object could move differently.

The dot associates the method call (`forward`) with the object that will undertake the method (the object referred to by `piece`), while the parentheses (`()`) cause the method call to take place.

The semicolon is a bit of syntax to mark the end of the method call statement. It can also be seen as a separator, separating this statement from the next. One reason for having the semicolon present is to simplify the implementation of the Java compiler by precisely marking the end of a statement. The semicolon also improves the readability of the code for people and can help remove ambiguities that otherwise might occur in more complicated sections of code. The semicolon in Java is directly analogous to the use of full stop in natural language, it marks the end of a sequence of syntactic features.

### 2.2 How many different interpretations of the following statement sequence are there?

```
piece.forward ( ) ;  
piece.forward ( ) ;  
piece.right ( ) ;  
piece.forward ( ) ;  
piece.left ( ) ;
```

The result of executing these statements depends on the behaviour of the methods called. Hence, the number of interpretations depends on the number of different versions of the methods—at the extreme, the number of interpretations is infinite (for example, a new version of `forward` can always be created that moves one square more than the previous version).

In practice, of course, the behaviour of each method will be fixed for a given program, so there will only be a single interpretation of the statement sequence. This makes the statement sequence *deterministic*, meaning that it will always perform the same movement actions when executed. The location of where the piece ends up will depend on its starting position.

### 2.3 Write down a sequence of statements that describes making a telephone call. Can it be done without using control statements?

One of the simplest sequences might be:

1. Pick up handset.
2. Dial number
3. Speak
4. Hang-up

This requires no control statements but is not robust nor particularly realistic. For example, it assumes the number being called is not engaged and that the conversation is limited to speaking a message and then hanging up. As soon as we want to manage errors or exceptions, not to mention actual human interaction (!) then control flow features are required.

**2.4** Does the pseudocode below (from Section 2.5, Page 23) describe a general-purpose solution to starting from and moving to any square on a chessboard?

```
while ( not at destination )
  if ( destination is to the left )
    piece moves left
  else
    if ( destination is to the right )
      piece moves right;
    piece moves forward
```

The way to approach this question is to attempt to find a start position that does *not* lead to a solution which will immediately show that the algorithm is not a solution to the problem. Pencil and paper sketches can be used to try out ideas, while working with a small-sized board, say 5×5, would help speed up the search.

Following this approach, you would hopefully notice that the algorithm has a serious flaw—the piece can move forward, left or right but not backwards. Hence, for example, if the destination square is behind the starting square the forward direction of travel is away from the destination and the destination cannot be reached. Moreover, once the piece reaches the edge of the board the algorithm fails, as the piece cannot move forward but there is no provision to terminate the loop. Altogether a really unsatisfactory algorithm!

To fully investigate this algorithm you could construct a program that conducts an exhaustive search. This would systematically try all combinations of starting and destination squares until one pair failed to work. However, using exhaustive search to test algorithms in general is not useful as there are all too often so many possible combinations to examine that the program would simply take too long to run (literally years in the case of most real-world algorithms). An alternative is to attempt to construct a mathematical proof, but for most algorithms this becomes very (if not too) complex.

**2.5** Write, in pseudocode, a program to move from one corner of a chessboard to the corner diagonally opposite, assuming that some of the squares are blocked and cannot be used. Note that additional test conditions will be needed, possibly something like 'is the square ahead blocked'. Does allowing diagonal moves make the program easier to write?

To solve this problem a search algorithm needs to be developed. This will continually try different paths across the board until one is found that reaches the destination. An important feature of a search algorithm is some memory of which paths have already been searched so as to avoid repeating what has already been done. In the case of this problem, it also avoids going round in circles when particular patterns of blocked squares occur.

It turns out to be quite hard to give a satisfactory solution to this problem using just loops and if statements. Here is one possible answer that assumes some memory is available to remember where the piece has already been.

```
while (not at far corner) {  
  if (can move forward) {  
    move forward  
  } else if (can go left) {  
    move left  
  } else if (can go right) {  
    move right  
  } else {  
    return to previous square  
  }  
}
```

Allowing diagonal moves might lead to a solution being found faster, or a shorter route being found, but won't simplify the code. In fact, the code would have to include more if statements to test for possible diagonal moves.

**2.6** How might a real number be stored in a variable? How does the fixed size of a variable affect the storage of real numbers?

A binary representation for the real number is required. Such representations are fairly complex but there are a number of widely used standards defining suitable representations. Java uses the IEEE-754 format floating point representation, 32 bits in size for type float and 64 bits in size for type double.

The size of a variable container determines the number of bits available to represent a real number and, hence, the range of possible values. Values outside the range cannot be represented as they require more bits than are available.

**2.7** Why can't a value of type int be stored in a variable of type double?

**ints** and **doubles** are stored using different binary representations, and indeed different numbers of bits (32 for **ints** and 64 for **doubles**). If the binary representation of an **int** were stored in a **double** it would not represent the same value when treated as a **double**, due to the different representations. It is clear that this is an highly abstract (indeed almost glib) solution to this question but more detailed (arguably more correct) solutions only add detail they do not add any actually crucial factors.

**2.8** Compare the idea of types with the units used for measuring size or weight. Are such units actually types?

A type defines how a value is to be understood and used, much the same way that a unit of size or weight does. For example, 1Kg, denotes that the value 1 is to be treated as a measurement of weight. In doing so, the rules for how the weight value should be used are clearly defined, along with how weight values can and cannot be combined with size values. Hence, units of size or weight are effectively types as, like types, they specify how to interpret and use values.

**2.9** How would the following strings be ordered if compared using the `compareTo` method: `program`, `proGram`, `PROGRAM`, `Program`, `ProGram`, `programs`, `program3`.

One way to determine the order is to write a program and find out (see Exercise 2.9 later). This is often a good strategy for testing how things work and getting more programming practice. However, it is also wise to consult the programming language manual and to read through the relevant documentation (the javadoc generated documentation). To answer the question you need to know that characters are represented by integers and so a character ordering can be determined based on the integer values. The strings can then be compared character-by-character to find the ordering.

Given that the alphabetic characters and the digits are ordered 0-9, A-Z, a-z, the strings are ordered: `PROGRAM`, `ProGram`, `Program`, `proGram`, `program`, `program3`, `programs`.

**2.10** Find the list of Java keywords and familiarize yourself with them. What happens if a keyword is used as a variable name?

The Java compiler would report an error, as it would recognize the keyword and determine that its use where a variable name is expected violates the grammar rules of the Java language.

**2.11** Why is each of these variable declarations invalid?

```
int i = 1.2 ;
double d = 1,200.46 ;
char c = "hello" ;
String s = "word;"
double j = 2.3×104 ;
```

1.2 is of type double and cannot be stored in an int variable.

1,200.46 is a floating point value but is not formatted in a way that is legal in Java. Specifically, the comma is not allowed. Even though it might be normal to use it this way in some cultures, Java requires you to use 1200.46. In many cultures this value would be written 1 200,46 but again Java does not recognize this as a legal way of representing the value.

"hello" is a string not a character and cannot be stored in a variable of type **char** (a **char** can only store a single character not a sequence as in a string).

The semicolon that should mark the end of the declaration is missing. The semicolon present is part of the string not the Java statement.

Although  $2.3 \times 10^4$  shows a widely used way of representing a real number, the Java language requires a different syntax, in this example 2.3E4. Also, of course, neither the symbol  $\times$  nor superscripts—like the <sup>4</sup>—appear on the keyboard and moreover they are usually impossible to create in the sort of text editors used to edit source code.

**2.12** *What are the values of these expressions:*

```

true || false && true
true && false || true

```

`||` is the boolean or operator, while `&&` is the boolean and operator. Both have equal precedence and are left associative, i.e. associate left to right, so the expressions are evaluated left to right.

```

true — true || false is true, true && true is true.
true — true && false is false, false || true is true.

```

## Programming Exercises

**2.1** *Type in and run DisplayOneToFive and BasicInputOutput on your computer system.*

Do just that! You will need to find out how to edit and compile Java programs on your particular computer.

**2.2** *Write a program to keep inputting integer values until  $-1$  is entered.*

```

public class E_2_2 {
    private void doComputation () {
        final Input in = new Input ();
        int value = 0 ;
        do {
            System.out.print ( "Type an int: " );
            value = in.nextInt ();
        } while ( value != -1 );
    }
    public static void main ( final String[] args ) {
        E_2_2 object = new E_2_2 ();
        object.doComputation ();
    }
}

```

Don't forget that to compile a program, the source code has to be saved into a file whose name matches the class name. In the case of this program, it should be saved as `E_2_2.java`. It is then compiled, using a command such as `javac E_2_2.java` or `jikes E_2_2.java` to create `E_2_2.class` so that it can be executed using a command such as `java E_2_2`.

**2.3** Write a program using a while loop to display a message 10 times. Each message should be on a separate line using the following format with numbering starting from 1:

```

1: A message
2: A message
3: A message
...

```

```

public class E_2_3 {
    private void doComputation () {
        int n = 0 ;
        while ( n < 10 ) {
            ++n ;
            System.out.print ( n );
            System.out.print ( ": " );
            System.out.println ( "Hello World" );
        }
    }
    public static void main ( final String[] args ) {
        E_2_3 object = new E_2_3 ();
        object.doComputation ();
    }
}

```

By combining several statements, the program above can be reduced to this:

```

public class E_2_3a {
    private void doComputation () {
        int n = 0 ;
        while ( n++ < 10 ) { System.out.println ( n + ": Hello World" ); }
    }
}

```

```

public static void main ( final String[] args ) {
    E_2_3a object = new E_2_3a ( );
    object.doComputation ( );
}
}

```

2.4 Repeat Exercise 2.3 using a do loop.

```

public class E_2_4 {
    private void doComputation ( ) {
        int n = 1 ;
        do { System.out.println ( n + " : Hello World" ) ; } while ( n++ < 10 ) ;
    }
    public static void main ( final String[] args ) {
        E_2_4 object = new E_2_4 ( ) ;
        object.doComputation ( ) ;
    }
}

```

2.5 Write a program using loops to display the following with the constraint that only a single character at a time may be output:

```

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

```

(We appreciate that restricting output to one character at a time is somewhat artificial but the point is to work with loops rather than using sophisticated input-output mechanisms.)

```

public class E_2_5 {
    private void doComputation ( ) {
        for ( int m = 0 ; m < 4 ; ++m ) {
            for ( int n = 0 ; n < 4 ; ++n ) { System.out.print ( '*' ) ; }
            System.out.println ( ) ;
        }
    }
    public static void main ( final String[] args ) {
        E_2_5 object = new E_2_5 ( ) ;
        object.doComputation ( ) ;
    }
}

```

2.6 Write a program using loops to display the following with the constraint that only a single character at a time may be output:

```

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

public class E_2_6 {
    private void doComputation () {
        for ( int m = 0 ; m < 6 ; ++m ) {
            for ( int n = 0 ; n < m + 1 ; ++n ) { System.out.print ( '*' ) ; }
            System.out.println() ;
        }
    }
    public static void main ( final String[] args ) {
        E_2_6 object = new E_2_6 () ;
        object.doComputation () ;
    }
}

```

2.7 Write a program to input 10 integers, add them up and display their sum and average.

```

public class E_2_7 {
    private void doComputation () {
        final Input in = new Input () ;
        int sum = 0 ;
        for ( int n = 0 ; n < 10 ; ++n ) {
            System.out.print ( "Enter int " + ( n+1 ) + ": " ) ;
            int val = in.nextInt () ;
            sum = sum + val ;
        }
        System.out.println ( "Sum = " + sum ) ;
        System.out.println ( "Average = " + sum/10 ) ;
    }
    public static void main ( final String[] args ) {
        E_2_7 object = new E_2_7 () ;
        object.doComputation () ;
    }
}

```

Note the use of the running total stored in the variable `sum`. The individual integers input by the user do not need to be kept once added to `sum`, so we don't have to worry about creating separate variables to store each input value.

The calculation of the average (`sum/10`) is done using integer arithmetic, producing an integer result. If a real number is needed (i.e. a floating point number) then the sum has to be stored as a value of type **double**.

```

public class E_2_7a {
    private void doComputation () {

```

```

final Input in = new Input ( ) ;
double sum = 0.0 ;
for ( int n = 0 ; n < 10 ; ++n ) {
    System.out.print ( "Enter integer " + ( n+1 ) + ": " ) ;
    final int val = in.nextInt ( ) ;
    sum = sum + val ;
}
System.out.println ( "Sum = " + sum ) ;
System.out.println ( "Average = " + sum/10 ) ;
}
public static void main ( final String[] args ) {
    E_2_7a object = new E_2_7a ( ) ;
    object.doComputation ( ) ;
}
}

```

**2.8** Repeat Exercise 2.6 but first ask the user how many integers will be typed in and then input that number.

```

public class E_2_8 {
    private void doComputation ( ) {
        final Input in = new Input ( ) ;
        System.out.print ( "How many numbers? " ) ;
        final int count = in.nextInt ( ) ;
        int sum = 0 ;
        for ( int n = 0 ; n < count ; ++n ) {
            System.out.print ( "Enter integer " + ( n+1 ) + ": " ) ;
            int val = in.nextInt ( ) ;
            sum = sum + val ;
        }
        System.out.println ( "Sum = " + sum ) ;
        System.out.println ( "Average = " + sum/count ) ;
    }
    public static void main ( final String[] args ) {
        E_2_8 object = new E_2_8 ( ) ;
        object.doComputation ( ) ;
    }
}

```

**2.9** Write a program to confirm that your answer to Self-review 2.9 is correct.

```

public class E_2_9 {
    private void doComputation ( ) {
        final String[] names = { "program", "proGram", "PROGRAM", "Program", "ProGram", "programs", "program3" } ;
        final int size = names.length ;
        for ( int i = 0 ; i < size-1 ; ++i ) {
            for ( int j = 0 ; j < size-1 ; ++j ) {
                if ( names[j].compareTo ( names[j+1] ) > 0 ) {
                    final String tmp = names[j] ;
                    names[j] = names[j+1] ;
                    names[j+1] = tmp ;
                }
            }
        }
    }
}

```

```

    }
    for ( int i = 0 ; i < size ; ++i ) {
        System.out.println ( names[i] );
    }
}
public static void main ( final String[] args ) {
    E_2_9 object = new E_2_9 ( ) ;
    object.doComputation ( ) ;
}
}

```

This example answer makes use of an *array* (introduced in detail in a later chapter).

**2.10** *Modify TriangleCalculation so that it repeatedly asks for input until the user wants to stop.*

```

public class E_2_10 {
    private void doComputation ( ) {
        final Input in = new Input ( ) ;
        String tryAgain = "n" ;
        do {
            System.out.print ( "Enter length of first side: " ) ;
            final double side1 = in.nextDouble ( ) ;
            System.out.print ( "Enter length of second side: " ) ;
            final double side2 = in.nextDouble ( ) ;
            System.out.print ( "Enter length of third side: " ) ;
            final double side3 = in.nextDouble ( ) ;
            // Test to see if the input describes an invalid triangle by seeing if the sum of the
            // lengths of any two sides is less than the length of the third.
            if ( ( ( side1 + side2 ) < side3 ) || ( ( side2 + side3 ) < side1 ) || ( ( side3 + side1 ) < side2 ) ) {
                System.out.println ( "The input does not describe a triangle." ) ;
            }
            else {
                final double perimeter = side1 + side2 + side3 ;
                final double semiperimeter = 0.5 * perimeter ;
                final double temp = semiperimeter * ( semiperimeter - side1 ) * ( semiperimeter - side2 ) * ( semiperimeter - side3 ) ;
                final double area = Math.sqrt ( temp ) ;
                System.out.println ( "Perimeter is: " + perimeter ) ;
                System.out.println ( "Area is: " + area ) ;
            }
            System.out.print ( "\nTry another triangle (y/n)? " ) ;
            tryAgain = in.nextLine ( ) ;
        } while ( tryAgain.equals ( "y" ) ) ;
    }
    public static void main ( final String[] args ) {
        E_2_10 object = new E_2_10 ( ) ;
        object.doComputation ( ) ;
    }
}

```

The original program has to be modified to include a new outer loop that controls whether a new triangle is input or the program terminates.

**2.11** *Write a program to input 10 words and then display the words that are first and last in alphabetical order.*

```

public class E_2_11 {
    private void doComputation () {
        final Input in = new Input ();
        String first = "";
        String last = "";
        for ( int i = 0 ; i < 10 ; ++i ) {
            System.out.print ( "Enter string " + ( i+1 ) + ": " );
            final String s = in.nextLine ();
            if ( first.equals ( "" ) || ( first.compareTo ( s ) > 0 ) ) { first = s ; }
            if ( last.compareTo ( s ) < 0 ) { last = s ; }
        }
        System.out.println ( "First word in alphabetical order is: " + first );
        System.out.println ( "Last word in alphabetical order is: " + last );
    }
    public static void main ( final String[] args ) {
        E_2_11 object = new E_2_11 ();
        object.doComputation ();
    }
}

```

The key to writing this program is to have two variables, `first` and `last`, to keep a record of the alphabetically first and last strings. The variables are updated as necessary as each string is input by the user by testing to see if the input string comes before `first` or after `last`. There is no need to store all the strings, only the two strings needed for the answer.

One complication is deciding how to initialize the variables holding the first and last strings, as we don't want the initial values to be confused with the strings being tested, and then, depending on what the user actually inputs, being given as one of the answers. For example, we could try to initialize `first` to "a" and `last` to "z". However, if the user never types in a string that alphabetically precedes "a", then none of the user input strings will be selected as being first and the wrong answer will be given. A similar line of reasoning can be followed for any initialization string, including the empty string.

The program above uses the empty string to initialize `first` and `last`. However, the empty string comes alphabetically before any other string, so `first` will end up with the wrong value (it won't change) unless the user happens to enter an empty string by just pressing return at the input prompt. The first if statement attempts to address this problem by testing to see if `first` holds an empty string and if so setting `first` to be the value of the user input string. Unfortunately this doesn't fully solve the problem as the user may enter the empty string but it won't be recorded as being alphabetically first—run the program and do some tests to verify this.

You might argue that this is good enough, or that the user should not enter an empty string as it is not a word, or that the problem is sufficiently rare not to matter. However, we want to follow good practice and get the program so that it works properly with all input. Moreover, when writing real programs, rather than answering exercise questions, it becomes very important to get these things right, as long experience has shown that sooner or later the flaws in an algorithm will cause a program to be unreliable.

One solution to the initialization problem is to initialize both `first` and `last` with the first string

typed in by the user. This effectively 'unwraps' the loop by one iteration, moving that iteration before the loop. Doing that gives the following program.

```
public class E_2_11a {
    private void doComputation () {
        final Input in = new Input ();
        System.out.print ( "Enter string 1: " );
        String first = in.nextLine ();
        String last = first ;
        for ( int i = 2 ; i < 11 ; ++i ) {
            System.out.print ( "Enter string " + i + ": " );
            final String s = in.nextLine ();
            if ( first.compareTo(s) > 0 ) { first = s ; }
            if ( last.compareTo(s) < 0 ) { last = s ; }
        }
        System.out.println ( "First word in alphabetical order is: " + first );
        System.out.println ( "Last word in alphabetical order is: " + last );
    }
    public static void main ( final String[] args ) {
        E_2_11a object = new E_2_11a ();
        object.doComputation ();
    }
}
```

An objection to moving the input of the first string outside the loop, although not really very serious in this case, is that there is results in some duplication of code. For more complicated programs, however, this can be a serious issue, as duplicate code can be difficult to maintain.

It turns out that with a bit of extra knowledge the first version of the answer can be made to work without too much extra complication. A string can be initialized to the value null (this is explained further in a later chapter of the book), not to be confused with the string "null". The user cannot type in this value so there is no chance of confusing it with any user input. Using **null** gives the following program.

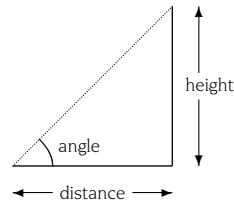
```
public class E_2_11b {
    private void doComputation () {
        final Input in = new Input ();
        String first = null ;
        String last = null ;
        for ( int i = 0 ; i < 10 ; ++i ) {
            System.out.print ( "Enter string " + ( i+1 ) + ": " );
            final String s = in.nextLine ();
            if ( ( first == null ) || ( first.compareTo ( s ) > 0 ) ) { first = s ; }
            if ( ( last == null ) || ( last.compareTo ( s ) < 0 ) ) { last = s ; }
        }
        System.out.println ( "First word in alphabetical order is: " + first );
        System.out.println ( "Last word in alphabetical order is: " + last );
    }
    public static void main ( final String[] args ) {
        E_2_11b object = new E_2_11b ();
        object.doComputation ();
    }
}
```

**2.12** Write a program that inputs the radius of a circle and displays its circumference and area.

```
public class E_2_12 {
    private void doComputation () {
        final Input in = new Input ();
        System.out.print ( "Enter radius of circle: " );
        final double radius = in.nextDouble ();
        System.out.println ( "Circumference is: " + ( 2 * Math.PI * radius ) );
        System.out.println ( "Area is: " + ( Math.PI * radius * radius ) );
    }
    public static void main ( final String[] args ) {
        E_2_12 object = new E_2_12 ();
        object.doComputation ();
    }
}
```

The library class `Math` conveniently provides a variable initialized to the value of  $\pi$ , called `Math.PI`.

**2.13** Write a program that determines the height of a building, given the angle of the top of the building and the distance from the building at which the angle was measured.



*Hint: The methods `Math.sin`, `Math.cos` and `Math.tan` are available but they work in radians. The method `Math.toRadians` will convert from degrees to radians.*

```
public class E_2_13 {
    private void doComputation () {
        final Input in = new Input ();
        System.out.print ( "Enter distance from building: " );
        final double distance = in.nextDouble ();
        System.out.print ( "Enter angle (degrees) to top of building: " );
        final double angle = in.nextDouble ();
        final double height = distance * Math.tan ( Math.toRadians ( angle ) );
        System.out.println ( "Height of building is: " + height );
    }
    public static void main ( final String[] args ) {
        E_2_13 object = new E_2_13 ();
        object.doComputation ();
    }
}
```

The main point of this question is to spend some time looking at the documentation for class `Math`, to start to become familiar with what the documentation looks like and how to use it.

## Challenges

We shall assume you are already in the habit of constructing test plans and running them to create test logs, so we won't labour the point any more.

- 2.1** *Modify `TriangleCalculation` so that it tests itself against a reasonable sized collection of tests. Do this by putting together the tests, complete with the expected correct answers. Then perform each test and compare the results with the expected answers.*
- 2.2** *Write a program to read in a line of text as a `String` and output the number of characters and words it contains. Spaces and tabs should not be counted as characters.*

*Hint: Look in the JDK documentation for information about methods provided by the `String` class.*

The key to answering this question is to firstly realize that a `String` can be examined character by character (using the `charAt` method). Then the task is to come up with a simple algorithm for determining whether a character is in a word or not. The program below uses a simple **boolean** variable that is toggled between **true** and **false** as each character is examined, to record whether the character is in a word or not. If the last character was not in a word (the flag is **false**) but the current character is (it is a letter), then the flag can be toggled to **true**. Similarly if the flag is **true** it is toggled to **false** when the current character is a tab or space. When the flag is toggled to **true** the word count can be incremented.

```
public class C_2_2 {
    private void doComputation () {
        final Input in = new Input ();
        System.out.print ( "Enter a line of text: " );
        final String text = in.nextLine ();
        int charCount = 0 ;
        int wordCount = 0 ;
        boolean inWord = false ;
        for ( int n = 0 ; n < text.length () ; ++n ) {
            char c = text.charAt ( n ) ;
            if ( ( c != ' ' ) && ( c != '\t' ) ) {
                ++charCount ;
                if ( ! inWord ) {
                    inWord = true ;
                    ++wordCount ;
                }
            }
            else { inWord = false ; }
        }
        System.out.print ( "\nNumber of characters (exluding space and tabs): " ) ;
        System.out.println ( charCount ) ;
        System.out.print ( "Number of words: " ) ;
        System.out.println ( wordCount ) ;
    }
}
```

```
public static void main ( final String[] args ) {  
    C_2_2 object = new C_2_2 ( );  
    object.doComputation ( );  
}  
}
```

**2.3** Write a program to read in a day, month and year, create a `Date` object and output the result.

*Hint: You can't create a `Date` directly. Look at the JavaSE documentation for the `Calendar` class.*

The purpose of this challenge is to spend some time studying the JavaSE documentation and practice using it to find the classes and methods needed to solve programming problems. In this case the first thing to discover is that although it is possible to directly create a class `Date` object using a `new Date ( )` expression, doing so is not going to help. A `Date` object is only meant to provide the basic representation of a date (as the number of milliseconds since 1970-01-01 00:00:00 GMT—the documentation provided for class `Date` explains why in detail). Programs that need to use dates should make use of various other more useful classes to actually manipulate dates.

The example program below makes use of a calendar object, which is obtained by calling the static method `getInstance` belonging to `Calendar`. The object returned will actually be an instance of a subclass of `Calendar` called `GregorianCalendar` that is specialized to represent the familiar calendar and date system used by many countries in the world.

The reason for obtaining a calendar object in this way is to allow different calendar and date systems to be used, although no alternatives are directly supported by the at the JDK time of writing. However, if alternatives were available a calendar object could be returned based on the calendar system that the computer running the program is configured for. A properly written program would then be able to work with any supported calendar system without having to be altered for a specific system.

Once the calendar object is obtained, the `set` method is used to store the date input by the user. To verify that the date has been correctly stored in the calendar object the final two statements output the stored date. As the format in which a date is displayed differs depending on the country or *locale* the user's computer is configured for, class `DateFormat` is asked for a formatting object using the `getDateInstance` method. The method `format` is then called on the formatting object with the argument of `calendar.getTime ( )`, and returns a string representing the date in the desired format. The `getTime` method actually returns a `Date` object, so it is at this point in the program that a `Date` object appears.

```
import java.util.Calendar ;  
import java.text.DateFormat ;  
public class C_2_3 {  
    private void doComputation ( ) {  
        final Input in = new Input ( );  
        System.out.print ( "Enter Day (1-7): " );
```

```

final int day = in.nextInt ( ) ;
System.out.print ( "Enter Month (1-12): " ) ;
final int month = in.nextInt ( ) ;
System.out.print ( "Enter Year: " ) ;
final int year = in.nextInt ( ) ;
final Calendar calendar = Calendar.getInstance ( ) ;
// Set the date -- note that the month is stored as an integer in the range 0-11.
calendar.set ( year , month - 1 , day ) ;
final String myDate = DateFormat.getDateInstance ( ).format ( calendar.getTime ( ) ) ;
System.out.println ( "The date entered was: " + myDate ) ;
}
public static void main ( final String[] args ) {
    C_2_3 object = new C_2_3 ( ) ;
    object.doComputation ( ) ;
}
}

```

#### 2.4 Modify TriangleCalculation to use the BigDecimal class and so avoid arithmetic overflow.

Hint: The `BigDecimal` class cannot use operators like `+`, it only has method calls like `add`, so expression evaluation is the same but very different.

Hint: You will need to investigate how to calculate square roots since `java.util.Math` works with doubles not `BigDecimals`. The Newton–Raphson iteration method is probably the most appropriate to use.

It is likely that you will come up with something along the lines of:

```

import java.math.BigDecimal ;
import java.math.BigInteger ;
import java.math.MathContext ;
import java.math.RoundingMode ;
/**
 * Program to input the lengths of the sides of a triangle and output the perimeter and area of
 * the triangle using Heron's Formula to calculate the area. This implementation uses
 * <code>BigDecimal</code> variables so as to ensure no overflows and therefore calculate the
 * correct numbers always. The finite size of <code>double</code> representation may be fine
 * for most situations but in testing we tried bizarre values and came across Infinity. This
 * program was written to find the real values that should have been.
 *
 * @author Graham Roberts and Russel Winder
 * @version 2005-07-26
 */
public class C_2_4 {
    /**
     * Get initial approximation for the square root.
     *
     * @return a representation  $10^n$  for some  $n$  that gives something in the same order of
     * magnitude as the expected result.
     */
    private static BigDecimal getInitialSqrtApproximation ( final BigDecimal n ) {
        final BigInteger integerPart = n.toBigInteger ( ) ;
        int length = integerPart.toString ( ).length ( ) ;
        if ( ( length % 2 ) == 0 ) { --length ; }
    }
}

```

```

length /= 2 ;
return BigDecimal.ONE.movePointRight ( length ) ;
}
/**
 * Calculate the square root of a <code>BigDecimal</code> value using a specified maximum
 * number of iterations. This implementation of <code>sqrt</code> makes use of Newton–Raphson
 * formula (which is equivalent to Newton’s Method, which is the same as Heron’s Method).
 */
public static BigDecimal sqrt ( final BigDecimal n , final int maxIterations ) throws IllegalArgumentException {
    if ( n.compareTo ( BigDecimal.ZERO ) <= 0 ) { throw new IllegalArgumentException ( ) ; }
    final MathContext mc = MathContext.DECIMAL64 ;
    BigDecimal x_np1 = getInitialSqrtApproximation ( n ) ;
    BigDecimal x_n = BigDecimal.ZERO ;
    int iterationCount = 0 ;
    do {
        x_n = x_np1 ;
        x_np1 = x_n.add ( n.divide ( x_n , mc ) , mc ).multiply ( new BigDecimal ( "0.5" ) , mc ) ;
        final BigDecimal error = n.subtract ( x_np1.multiply ( x_np1 , mc ) , mc ) ;
    } while ( ( ++iterationCount < maxIterations ) && ( x_np1.compareTo ( x_n ) != 0 ) ) ;
    return x_np1 ;
}
private void doComputations ( ) {
    final Input in = new Input ( ) ;
    System.out.print ( "Enter length of first side: " ) ;
    final BigDecimal a = in.nextBigDecimal ( ) ;
    System.out.print ( "Enter length of second side: " ) ;
    final BigDecimal b = in.nextBigDecimal ( ) ;
    System.out.print ( "Enter length of third side: " ) ;
    final BigDecimal c = in.nextBigDecimal ( ) ;
    // Test to ensure that the input describes a valid triangle by checking the triangle
    // inequality holds before attempting calculations.
    if ( ( a.add ( b ) .compareTo ( c ) > 0 ) && ( b.add ( c ) .compareTo ( a ) > 0 ) && ( c.add ( a ) .compareTo ( b ) > 0 ) ) {
        // Avoid division with BigDecimals if at all possible.
        final BigDecimal perimeter = a.add ( b ) .add ( c ) ;
        final BigDecimal s = perimeter.multiply ( new BigDecimal ( 0.5 ) ) ;
        final BigDecimal t = s.multiply ( s.subtract ( a ) ) .multiply ( s.subtract ( b ) ) .multiply ( s.subtract ( c ) ) ;
        final BigDecimal area = sqrt ( t , 100 ) ;
        System.out.println ( "Perimeter is: " + perimeter ) ;
        System.out.println ( "Area is: " + area ) ;
    }
    else {
        System.out.println ( "The input values do not describe a triangle." ) ;
    }
}
public static void main ( final String[] args ) {
    C_2_4 object = new C_2_4 ( ) ;
    object.doComputations ( ) ;
}
}

```