Contents

1 Programming with Objects and Classes 1

1 Introduction 3

2 Programming Fundamentals 5

Self-review Questions .................................................. 6
Programming Exercises ............................................. 10
Challenges ............................................................... 19

3 Adding Structure 23

Self-review Questions .................................................. 24
Programming Exercises ............................................. 30
Challenges ............................................................... 47

4 Introducing Containers 49

Self-review Questions .................................................. 50
Programming Exercises ............................................. 53
Challenges ............................................................... 65
<table>
<thead>
<tr>
<th>Chapter Title</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Drawing Pictures</td>
<td></td>
</tr>
<tr>
<td>Self-review Questions</td>
<td>71</td>
</tr>
<tr>
<td>Programming Exercises</td>
<td>72</td>
</tr>
<tr>
<td>Challenges</td>
<td>74</td>
</tr>
<tr>
<td>6 Classes and Objects</td>
<td></td>
</tr>
<tr>
<td>Self-review Questions</td>
<td>83</td>
</tr>
<tr>
<td>Programming Exercises</td>
<td>84</td>
</tr>
<tr>
<td>Challenges</td>
<td>86</td>
</tr>
<tr>
<td>7 Class Relationships</td>
<td></td>
</tr>
<tr>
<td>Self-review Questions</td>
<td>105</td>
</tr>
<tr>
<td>Programming Exercises</td>
<td>106</td>
</tr>
<tr>
<td>Challenges</td>
<td>110</td>
</tr>
<tr>
<td>8 Exceptions</td>
<td></td>
</tr>
<tr>
<td>Self-review Questions</td>
<td>113</td>
</tr>
<tr>
<td>Programming Exercises</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>115</td>
</tr>
<tr>
<td>9 Introducing Concurrency with Threads</td>
<td></td>
</tr>
<tr>
<td>Self-review Questions</td>
<td>117</td>
</tr>
<tr>
<td>Programming Exercises</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td>119</td>
</tr>
<tr>
<td>10 User Interfaces</td>
<td></td>
</tr>
<tr>
<td>Self-review Questions</td>
<td>129</td>
</tr>
<tr>
<td></td>
<td>130</td>
</tr>
</tbody>
</table>
2 The Process of Programming

11 The Programming Process

Self-review Questions .................................................. 140
Programming Exercises ............................................... 143
Challenges .................................................................. 143

12 Unit Testing

Self-review Questions .................................................. 146
Programming Exercises ............................................... 151
Challenges .................................................................. 156

13 Test-driven Programming Strategies

Self-review Questions .................................................. 158
Programming Exercises ............................................... 159
Challenges .................................................................. 159

14 Programming Tools

Self-review Questions .................................................. 162
Programming Exercises ............................................... 162
Challenges .................................................................. 163
3 Case Studies in Developing Programs 165

15 Introducing the Case Studies 167

16 Contacts Book 169

Programming Exercises ........................................... 170
Challenges .......................................................... 170

17 Pedestrian Crossing Simulation 171

Programming Exercises ........................................... 172
Challenges .......................................................... 172

4 The Java Programming Language in Detail 173

18 A Java Language Reference 175

19 Variables Types and Expressions 177

Self-review Questions ............................................ 178
Programming Exercises ........................................... 181

20 Flow Control 185

Self-review Questions ............................................ 186
Programming Exercises ........................................... 186

21 Classes and Packages 189

Self-review Questions ............................................ 190
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 Inheritance and Interfaces</td>
<td>193</td>
</tr>
<tr>
<td>Self-review Questions</td>
<td>194</td>
</tr>
<tr>
<td>Programming Exercises</td>
<td>194</td>
</tr>
<tr>
<td>23 Exception Handling</td>
<td>197</td>
</tr>
<tr>
<td>Self-review Questions</td>
<td>198</td>
</tr>
<tr>
<td>24 Threads and Concurrency</td>
<td>199</td>
</tr>
<tr>
<td>Self-review Questions</td>
<td>200</td>
</tr>
<tr>
<td>Programming Exercises</td>
<td>200</td>
</tr>
</tbody>
</table>
Part 1

Programming with Objects and Classes
Introduction
Chapter 1: Introduction

There are no self-review questions, exercises or challenges in this chapter.
2
Programming Fundamentals
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?

```plaintext
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 $5 \times 5$, 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 an 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.

```java
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?
**Self-review Questions**

**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?

```java
int i = 1.2;
double d = 1,200.46;
char c = "hello";
String s = "word;"
double j = 2.3×10^4;
```
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 $^4$—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:**

\[
\text{true} || \text{false} \&\& \text{true} \\
\text{true} \&\& \text{false} || \text{true}
\]

|| 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.

\[
\text{true — true} || \text{false is true, true} \&\& \text{true is true.} \\
\text{true — true} \&\& \text{false is false, false} || \text{true is true.}
\]

## 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" ) ; }
    }
}
Chapter 2: Programming Fundamentals

```java
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.

```java
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.)

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

```java
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`.

```java
public class E_2_7a {
    private void doComputation ( ) {
```
Chapter 2: Programming Fundamentals

```java
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.

```java
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.

```java
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[i].compareTo(names[j+1]) > 0) {
                    final String tmp = names[i];
                    names[i] = names[j+1];
                    names[j+1] = tmp;
                }
            }
        }
    }
```
Programming Exercises

```java
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.

```java
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 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
Programming Exercises

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.

```java
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.

```java
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.

```java
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.

```
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.

```java
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 ( "Number of characters (excluding space and tabs): " ) ;
        System.out.println ( charCount ) ;
        System.out.print ( "Number of words: " ) ;
        System.out.println ( wordCount ) ;
    }
}
```
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.
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> varaibles 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();
}
3
Adding Structure
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, dRaawcIRCe, class

convert, add_two, AddUp and dRaawcIRCe 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:

```java
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:

```java
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:

```java
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 to large to represent in the computer results in an unexpected value.

3.8 Is the method times2 (see page 64) referentially transparent?
Chapter 3: Adding Structure

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
Self-review Questions

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.

<table>
<thead>
<tr>
<th>comment</th>
<th>value</th>
<th>power</th>
<th>position</th>
<th>position &gt; -1</th>
<th>charAt(position)</th>
</tr>
</thead>
<tbody>
<tr>
<td>start of method</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>start of first iteration</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>true</td>
<td>'1'</td>
</tr>
<tr>
<td>case '1'</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>true</td>
<td>'1'</td>
</tr>
<tr>
<td>start of second iteration</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>true</td>
<td>'0'</td>
</tr>
<tr>
<td>case '0'</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>true</td>
<td>'0'</td>
</tr>
<tr>
<td>start of third iteration</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>true</td>
<td>'0'</td>
</tr>
<tr>
<td>case '0'</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>true</td>
<td>'0'</td>
</tr>
<tr>
<td>start of fourth iteration</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>true</td>
<td>'1'</td>
</tr>
<tr>
<td>case '1'</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td>true</td>
<td>'1'</td>
</tr>
<tr>
<td>start of fifth iteration</td>
<td>9</td>
<td>8</td>
<td>-1</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>loop terminates, return value</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

```java
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:

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

maps to the while loop structure:

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

The while equivalent of the given for loop is:

```java
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 ( "Conversion 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 :
Chapter 3: Adding Structure

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.

```java
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:

```java
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 ( "*"); }
        }
    }
    public static void main ( final String[] args ) {
        new E_3_4 ( ) .drawTriangle ( ) ;
    }
}
```
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:

```java
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:

```java
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.

```java
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 one 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.

```java
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.

```java
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);
    }
}
```
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:

```java
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:

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

\[
123_8 = 1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0 \\
= 64 + 16 + 3 \\
= 83_{10}
\]
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.

```java
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.

```java
//**
* 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 2006-09-04
```
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
Chapter 3: Adding Structure

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

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`:

```java
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.getDoCalculation("
Enter start number: ", application.getIntegerValue("Enter start 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:

```java
public class E_3_10 {
```
Programming Exercises

```java
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:

```java
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 ( ) ) {
```
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:

```java
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.

```java
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:

```java
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.

```java
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();
```
Chapter 3: Adding Structure

```java
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!

```java
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](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
```
3.1 Write a program to display the multiplication tables from 2 to 12 displayed as four rows of three tables, i.e.:  

1 \times 2 = 2  
1 \times 3 = 3  
1 \times 4 = 4  
2 \times 2 = 4  
2 \times 3 = 6  
2 \times 4 = 8  
3 \times 2 = 6  
3 \times 3 = 9  
3 \times 4 = 12  

3.2 Write a program that includes a recursive method to calculate the greatest common divisor of two numbers.

3.3 Write methods to display a large E, L and O as below. Use the methods to display HELLO in large letters in the following arrangement:

```
* *
* * *****
****** *
* * ******
* *  * ******
* *  *  * ******
* *  *  *  * ******
* *  *  *  *  * ******
```

3.4 Consider a table that shows temperature conversions from Celsius to Fahrenheit, like this:

<table>
<thead>
<tr>
<th>Temperature Conversion</th>
</tr>
</thead>
</table>
| \hline
<table>
<thead>
<tr>
<th>C</th>
<th>F</th>
<th>C</th>
<th>F</th>
<th>C</th>
<th>F</th>
<th>C</th>
<th>F</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>1</td>
<td>33</td>
<td>2</td>
<td>35</td>
<td>3</td>
<td>37</td>
<td>4</td>
<td>39</td>
</tr>
<tr>
<td>5</td>
<td>41</td>
<td>6</td>
<td>42</td>
<td>7</td>
<td>44</td>
<td>8</td>
<td>46</td>
<td>9</td>
<td>48</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
<td>11</td>
<td>51</td>
<td>12</td>
<td>53</td>
<td>13</td>
<td>55</td>
<td>14</td>
<td>57</td>
</tr>
</tbody>
</table>

and so on up to 100C.

Write a program that asks for the number of columns to display (one column is one C/F pair) and displays a conversion table. Make sure the columns of numbers line up neatly.

Hint: Write a method to convert from Celsius to Fahrenheit.

Write a method that displays one line of the table, given the line number and the number of columns.

Write a method that displays the table one line at a time.

Separate the columns of C/F pairs using tab characters. A tab is represented using $'\t'$, for example `System.out.println ('\t').`

3.5 Write a program that reads an integer between 0 and 99 and ‘verbalizes it’. For example, if the program is given 23 it would display ‘twenty three’.
Chapter 3: Adding Structure

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.
4

Introducing Containers
Self-review Questions

4.1 Which of the following are valid array declarations?

```java
int x = new int[10] ;
int[][] = new int[][10] ;
double []d = new double[10] ;
char[] s = "Hello" ;
```

The declaration:

```java
double []d = new double[10] ;
```

is fine, although it would be much more usual to write it as:

```java
double[] d = new double[10] ;
```

Whitespace is not significant in Java so the spacing between non-space elements is of no consequence. This does mean it is important to space things to aid human readability which is why the second form is preferred to the first form as it visually associates the [] with the double as the compiler does.

The declaration:

```java
int x = new int[10] ;
```

causes a compilation error:

```java
found : int[]
required: int
   int x = new int[10] ;
     ^
2 errors
```

The problem here is that the variable is not of array type so the initialization is not possible.

```java
int[][] = new int[][][10] ;
```

The first problem here is that there is no variable name so the declaration is not well formed, it leads to an error like:

```java
Trial.java:4: not a statement
   int[][] = new int[][][10] ;
     ^
Trial.java:4: ';' expected
   int[][] = new int[][][10] ;
     ^
2 errors
```
If we correct this to give:

```java
int[][] xx = new int[][10];
```

then we still have an error:

```java
Trial.java:4: ']' expected
    int[][] xx = new int[][10];
           ^
Trial.java:4: array dimension missing
    int[][] xx = new int[][10];
                      ^
2 errors
```

The problem here is that we can only ever leave the last size blank. So:

```java
int[][] xx = new int[10][];
```

compiles fine and on execution make `xx` a reference to a size 10 array of references to `int` arrays. The declaration:

```java
char[] s = "Hello";
```

results in the error message:

```java
Trial.java:6: incompatible types
    found : java.lang.String
    required: char[]
        char[] s = "Hello";
               ^
1 error
```

The problem is that the types on left and right hand side of the initialization are different. On the right we have a `String` on the left we have a `char[]` and the two are different. To correct this we have to use one of:

```java
String s = "Hello";
char[] s = {'H', 'e', 'l', 'l', 'o', '\0'};
```

### 4.2 What is array bounds checking?

It is a run time check that any index used in an index expression is greater than 0 and less than the maximum size of the array being indexed into.

### 4.3 What is the maximum index that can be used with an array of size 7?
Chapter 4: Introducing Containers

6 because arrays are always 0-origin—a 7 item array has elements indexed by 0, 1, 2, 3, 4, 5 and 6.

4.4 How is a 3D array represented in terms of array objects?

4.5 Which of these are valid array indexing expressions for an array of size 10?

\[ n[2.5] \]
\[ n[0] \]
\[ n[3-7] \]
\[ n[2*3] \]
\[ n[n[1]] \]

The expressions:

\[ n[0] \]
\[ n[2*3] \]
\[ n[n[1]] \]

are all valid. The last expression however will lead to an `ArrayIndexOutOfBoundsException` if `n[1]` is negative or greater than 9.

The expression:

\[ n[2.5] \]

is not valid as the index value is a `double` and all indexes must be positive and integral.

The expression:

\[ n[3-7] \]

is not valid as the `int` expression results in -4 and negative indexes are not permitted.

4.6 When is it an advantage to use an `ArrayList` rather than an array?

Whenever there is any doubt about the exact size of the array at compile time.

4.7 Explain how array parameters work. What happens if an element of an array passed as a parameter to a method is changed by assignment? Do `ArrayLists` behave the same or differently?
The array elements are stored somewhere in memory and a reference to the array is passed from the caller to the callee.

As the caller has passed a reference to the array, the caller is using the same array elements which means the callee is changing the data that the caller had.

ArrayLists behave the same.

**4.8 What do opening and closing a file do?**

Opening a file causes the JVM and the underlying operating system to set up connections between the program and the file. These connections must always be set up before a program can read and/or write a file. Closing a file causes the connections to be terminated and all the JVM and operating system resources set up during opening to be released.

**4.9 How does **eof** work?**

**Programming Exercises**

**4.1 Write a program to read in 10 integers and store them in an array. Then display the contents of the array.**

```java
public class E_4_1 {
    private void execute () {
        final int[] data = new int[10];
        final Input input = new Input();
        System.out.println("Enter "+data.length+" int values :");
        for (int i = 0 ; i < data.length ; ++i) {
            data[i] = input.nextInt();
        }
        System.out.println("Values input were :");
        System.out.println();
    }
    public static void main (final String[] args) {
        new E_4_1().execute();
    }
}
```

**4.2 Write a program to read in a sequence of integers until 'stop' is entered. Store the integers in an array. Then display the average of the numbers entered.**

```java
public class E_4_2 {
    private int[] doInput () {
        final Input in = new Input();
        int[] buffer = new int[2000];
```
System.out.println ("Type in up to " + buffer.length + " integers, and the input with the word stop.");
for (int i = 0; i < buffer.length; ++i) {
    final String s = in.next();
    if (s.compareTo("stop") == 0) {
        // buffer is too large for the amount of data input so create a new array of the right
        // size for the number of items input and replace the current buffer.
        final int[] data = new int[i];
        System.arraycopy(buffer, 0, data, 0, i);
        buffer = data;
        break;
    }
    buffer[i] = Integer.parseInt(s);
}
return buffer;
}

private double calculateMean (final int[] data) {
    int total = 0;
    for (int i = 0; i < data.length; ++i) { total += data[i]; }
    return ((double) total) / data.length;
}

public static void main (final String[] args) {
    final E_4_2 object = new E_4_2();
    System.out.println("Mean of entered data is " + object.calculateMean(object.doInput()));
}

4.3 Repeat Exercise 4.2 but use an ArrayList instead of an array.

import java.util.ArrayList;
public class E_4_3 {
    private ArrayList<Integer> doInput () {
        final Input in = new Input();
        final ArrayList<Integer> buffer = new ArrayList<Integer>();
        System.out.println("Type in integers and the input with the word stop.");
        while (true) {
            final String s = in.next();
            if (s.compareTo("stop") == 0) { break; }
            buffer.add(Integer.parseInt(s));
        }
        return buffer;
    }
    private double calculateMean (final ArrayList<Integer> data) {
        int total = 0;
        for (int i = 0; i < data.size(); ++i) { total += data.get(i); }
        return ((double) total) / data.size();
    }
    public static void main (final String[] args) {
        final E_4_3 object = new E_4_3();
        System.out.println("Mean of entered data is " + object.calculateMean(object.doInput()));
    }
}

4.4 A matrix can be represented using a 2D array. Write methods to perform matrix addition, subtraction and
multiplication. Each method should take two 2D arrays as a parameter and return a new 2D array containing the result. Use the methods in a test program to verify that they work correctly.

The following is a potential solution:

```java
/**
 * Some methods for doing things with 2D rectangular arrays representing matrices. The data has to be of some
 * form of numeric type so for now we make doubles, later we can do this better.
 *
 * @author Russel Winder
 * @version 2006-09-10T18:58
 */

public class E_4_4 {

    public static boolean isRectangular ( final double[][] a ) {
        for ( int i = 1 ; i < a.length ; ++i ) {
            if ( a[i].length != a[0].length ) { return false ; }
        }
        return true ;
    }

    public static void checkSameDimensions ( final double[][] a , final double[][] b ) {
        if ( ! isRectangular ( a ) ) { throw new RuntimeException ( "a is not a rectangular array." ) ; }
        if ( ! isRectangular ( b ) ) { throw new RuntimeException ( "b is not a rectangular array." ) ; }
        if ( a.length != b.length ) { throw new RuntimeException ( "a and b do not have the same number of rows." ) ; }
        if ( a[0].length != b[0].length ) { throw new RuntimeException ( "a and b do not have the same number of columns." ) ; }
    }

    public static double[][] add ( final double[][] a , final double[][] b ) {
        checkSameDimensions ( a , b ) ;
        final double[][] r = new double[a.length][a[0].length] ;
        for ( int i = 0 ; i < a.length ; ++i ) {
            for ( int j = 0 ; j < a[0].length ; ++j ) {
                r[i][j] = a[i][j] + b[i][j] ;
            }
        }
        return r ;
    }

    public static double[][] subtract ( final double[][] a , final double[][] b ) {
        checkSameDimensions ( a , b ) ;
        final double[][] r = new double[a.length][a[0].length] ;
        for ( int i = 0 ; i < a.length ; ++i ) {
            for ( int j = 0 ; j < a[0].length ; ++j ) {
                r[i][j] = a[i][j] - b[i][j] ;
            }
        }
        return r ;
    }

    public static double[][] multiply ( final double[][] a , final double[][] b ) {
        if ( ! isRectangular ( a ) ) { throw new RuntimeException ( "a is not a rectangular array." ) ; }
        if ( ! isRectangular ( b ) ) { throw new RuntimeException ( "b is not a rectangular array." ) ; }
        if ( a[0].length != b.length ) { throw new RuntimeException ( "a and b are not multiplication compatible." ) ; }
        final double[][] r = new double[a.length][b[0].length] ;
        for ( int i = 0 ; i < a.length ; ++i ) {
            for ( int j = 0 ; j < b[0].length ; ++j ) {
                for ( int k = 0 ; k < b.length ; ++k ) {
                    r[i][j] += a[i][k] * b[k][j] ;
                }
            }
        }
    }
}
```
The following is a short program to test some of the features of the program, there needs to be a lot more to test it properly.

```java
public class E_4_4_Test {
    private void assertEqual (final double[][] expected, final double[][] actual) {
        E_4_4.checkSameDimensions(expected, actual);
        for (int i = 0; i < expected.length; ++i) {
            for (int j = 0; j < expected[0].length; ++j) {
                if (expected[i][j] != actual[i][j]) {
                    System.err.println("Got a " + actual[i][j] + " expected a " + expected[i][j]);
                    System.exit(1);
                }
            }
        }
    }
    public void addTest () {
        final double[][] a = new double[][] {{1, 1}, {1, 1}, {1, 1}};
        final double[][] b = new double[][] {{1, 1}, {1, 1}, {1, 1}};
        final double[][] r = new double[][] {{2, 2}, {2, 2}, {2, 2}};
        assertEqual(r, E_4_4.add(a, b));
    }
    public void subtractTest () {
        final double[][] a = new double[][] {{1, 1}, {1, 1}, {1, 1}};
        final double[][] b = new double[][] {{1, 1}, {1, 1}, {1, 1}};
        final double[][] r = new double[][] {{0, 0}, {0, 0}, {0, 0}};
        assertEqual(r, E_4_4.subtract(a, b));
    }
    public void multiplyTest () {
        double[][] a = {{1, 1}};
        double[][] b = {{1}, {1}};
        double[][] r = {{2}};
        assertEqual(r, E_4_4.multiply(a, b));
        a = new double[][] {{1, 1}};
        b = new double[][] {{1, 1}};
        r = new double[][] {{1, 1}, {1, 1}};
        assertEqual(r, E_4_4.multiply(a, b));
        a = new double[][] {{1, 1}, {1, 1}, {1, 1}};
        b = new double[][] {{1, 1, 1}, {1, 1, 1}};
        r = new double[][] {{2, 2, 2}, {2, 2, 2}, {2, 2, 2}};
        assertEqual(r, E_4_4.multiply(a, b));
    }
    public static void main (final String[] args) {
        final E_4_4_Test tester = new E_4_4_Test();
        tester.addTest();
        tester.subtractTest();
        tester.multiplyTest();
    }
}
```
4.5 Extend program AddUpColumns in Section 4.2.10, page 110, to add a method that ensures the 2D array is a square array and employs this new method to ensure adding up the columns actually makes sense.

Although the question as printed said ‘square’, it should have said ‘rectangular’. In the following we have created a method ensureRectangular that terminates execution if the array is not rectangular. The method is called as the first action in totalColumns since that is the most appropriate place of responsibility, i.e. it is the responsibility of the totalColumns method to ensure the array that it is given is a reasonable array to work with. We have added an extra test case to ensure that the method is called and traps a non-rectangular array. It has to be the last test case though as it terminates execution.

```java
public class E_4_5 {
    private void ensureRectangular ( final double[][] array ) {
        for ( int i = 1 ; i < array.length ; ++i ) {
            if ( array[i].length != array[0].length ) {
                System.out.println ( "Array is not rectangular." ) ;
                System.exit ( 1 ) ;
            }
        }
    }
    private double[] totalColumns ( final double[][] array ) {
        ensureRectangular ( array ) ;
        final int nColumns = array[0].length ;
        final double[] totals = new double[nColumns] ;
        for ( int column = 0 ; column < nColumns ; ++column ) {
            totals[column] = 0.0 ;
            for ( int row = 0 ; row < array.length ; ++row ) {
                totals[column] += array[row][column] ;
            }
        }
        return totals ;
    }
    private void displayTotals ( final double[] totals ) {
        System.out.print ( "Totals are: " ) ;
        for ( int column = 0 ; column < totals.length ; ++column ) {
            System.out.print ( totals[column] + " " ) ;
        }
        System.out.println ( ) ;
    }
    public static void main ( final String[] args ) {
        final E_4_5 object = new E_4_5 ( ) ;
        double[][] testArray = {
            { 2.1 , 3.4 , 5.6 , 9.6 , 5.5 } ,
            { 4.2 , 3.5 , 6.6 , 1.9 , 3.2 } ,
            { 1.1 , 5.8 , 8.2 , 4.5 , 2.8 } }
        ;
        object.displayTotals ( object.totalColumns ( testArray ) ) ;
        testArray = new double[][] {
            { 2.1 , 3.4 , 5.6 , 9.6 , 5.5 } ,
            { 4.2 , 3.5 , 6.6 , 1.9 } }
        ;
        object.displayTotals ( object.totalColumns ( testArray ) ) ;
    }
}
```
4.6 **Rewrite program** AddUpColumns in Exercise 4.5 to use ArrayLists instead of arrays.

```java
import java.util.Arrays;
import java.util.ArrayList;

public class E_4_6 {
    private ArrayList<Double> totalColumns ( final ArrayList<ArrayList<Double>> array ) {
        final int nColumns = array.get ( 0 ).size ( ) ;
        final ArrayList<Double> totals = new ArrayList<Double> ( nColumns ) ;
        for ( int column = 0 ; column < nColumns ; ++column ) {
            totals.add ( 0.0 ) ;
            for ( int row = 0 ; row < array.size ( ) ; ++row ) {
                totals.set ( column , totals.get ( column ) + array.get ( row ).get ( column ) ) ;
            }
        }
        return totals ;
    }

    private void displayTotals ( final ArrayList<Double> totals ) {
        System.out.print ( "Totals are: " ) ;
        for ( Double d : totals ) { System.out.print ( d + " " ) ; }
        System.out.println ( ) ;
    }

    public static void main ( final String[] args ) {
        final E_4_6 object = new E_4_6 ( ) ;
        final ArrayList<ArrayList<Double>> testArray = new ArrayList<ArrayList<Double>> ( ) ;
        testArray.add ( new ArrayList<Double> ( Arrays.asList ( 2.1 , 3.4 , 5.6 , 9.6 , 5.5 ) ) ) ;
        testArray.add ( new ArrayList<Double> ( Arrays.asList ( 4.2 , 3.5 , 6.6 , 1.9 , 3.2 ) ) ) ;
        testArray.add ( new ArrayList<Double> ( Arrays.asList ( 1.1 , 5.8 , 8.2 , 4.5 , 2.8 ) ) ) ;
        object.displayTotals ( object.totalColumns ( testArray ) ) ;
    }
}
```

4.7 **Rewrite program** DisplayTextFile to read and display strings instead of characters.

It seems that this question is a bit redundant in that the version of DisplayTextFile on page 132 is actually the answer to this question. It appears that in updating the book, the original point behind this question was taken into the text itself, and we did not update the question.

```java
/**
 * A program to display the contents of a text file.
 *
 * @author Graham Roberts
 * @author Russel Winder
 * @version 2004-11-03
 */

public class DisplayTextFile {
    public String getFileName ( ) {
        final Input in = new Input ( ) ;
        System.out.print ( "Enter name of file to display: " ) ;
        final String name = in.nextLine ( ) ;
        return name ;
    }

    public void display ( final String fileName ) {
        final FileInputStream in = new FileInputStream ( fileName ) ;
```
Programming Exercises

```java
while (in.hasNextLine()) { System.out.println(in.nextLine()); }
in.close();

public static void main (final String[] args) {
    final DisplayTextFile object = new DisplayTextFile();
    object.display(object.getFileName());
}
```

4.8 Write a program to find the largest sized array you can use on your computer system.

The 'brute force' method might seem to be:

```java
public class E_4_8_brute {
    private void findLargestArray () {
        int i = 1;
        while (true) {
            System.out.print("Trying " + (i) + ". . .");
            final byte[] array = new byte[i];
            System.out.println("done.");
            ++i;
        }
    }
}
public static void main (final String[] args) {
    final E_4_8_brute object = new E_4_8_brute();
    object.findLargestArray();
}
```

but this is not actually a solution because it results (on a 3GHz, 1GB, Ubuntu machine under normal workstation usage) in a program that is indistinguishable from an infinite loop. It is not, in fact, an infinite loop, but it would take many, many, many hours to get to the point where the array size was too big.

The most obvious way of actually answering this question is to use an interactive program. By involving a human being we can avoid our program having to make any guesses about the maximum size of array. The user can apply sophisticated algorithms and heuristics to make the search relatively quick – the hint here is to employ binary chop search. So, using the program:

```java
public class E_4_8 {
    private void findLargestArray () {
        final Input input = new Input();
        while (true) {
            System.out.print("Enter a value to try: ");
            final int i = input.nextInt();
            final byte[] array = new byte[i];
            System.out.println("Succeeded.");
        }
    }
}
public static void main (final String[] args) {
    final E_4_8 object = new E_4_8();
}
```
we found that, on the aforementioned 3GHz, 1GB, Ubuntu machine under normal workstation usage, using JDK 1.5.0_06, the largest array was around 61860980 bytes. There was a certain amount of non-determinism – sometimes numbers would succeed and sometimes they would fail. Almost certainly something to do with the exact memory available at the instant the JVM was allocating the array.

Actually we didn’t use the above program, we used:

```java
public class E_4_8_exceptions {
    private void findLargestArray () {
        final Input input = new Input ();
        while ( true ) {
            System.out.print ( "Enter a value to try: " ) ;
            final int i = input.nextInt ( ) ;
            try {
                final byte[] array = new byte[i] ;
            } catch ( OutOfMemoryError oome ) {
                System.out.println ( "Failed." ) ;
                continue ;
            }
            System.out.println ( "Succeeded." ) ;
        }
        System.out.println ( "Succeeded." ) ;
    }
    public static void main ( final String[] args ) {
        final E_4_8_exceptions object = new E_4_8_exceptions ( ) ;
        object.findLargestArray ( ) ;
    }
}
```

so as to avoid having to restart the Java system for every failure. However at this point we haven’t covered exceptions so we you may not have come up with this. This reflects the difficulties of sequencing material. It would have been good to put exceptions earlier but . . .

4.9 Create a program that writes a data file containing 1000 random integers between 1 and 25. Use the JDK class Random to create random numbers, or implement your own random number algorithm. Then write a second program that reads the integers from the file and counts how many of each integer is in the file.

Writing our own random number generator is almost certainly a bad idea. Generating random numbers on a computer is a complicated issue – both in terms of the mathematics and in terms of the actual programming – and the standard Java library has some good solutions. We will just use them. From the documentation we see there is a class Random that we could use. Alternatively, we could use the method Math.random, but this is just a utility method for using Random.nextDouble in a certain way and as the question asks about random integers, and there is a method Random.nextInt ( int n ) which seems to do more of what we want, let us use that.

So now we are in a position to create the writer of numbers:
import java.util.Random;

public class E_4_9_writer {
    private void writeNumbers() {
        final Random generator = new Random();
        final FileOutputStream output = new FileOutputStream("random_numbers");
        for (int i = 0; i < 1000; ++i) {
            // NB Random.nextInt generates in the range 0 <= x < n and the question requires the numbers to be in
            // the range 1 <= x <= 25.
            output.writeInt(generator.nextInt(25) + 1);
            output.writeEndOfLine();
        }
        output.close();
    }
    public static void main(final String[] args) {
        final E_4_9_writer object = new E_4_9_writer();
        object.writeNumbers();
    }
}

public class E_4_9_reader {
    private void displayCounts() {
        final int maxNumber = 25;
        final FileInputStream input = new FileInputStream("random_numbers");
        final int[] counts = new int[maxNumber];
        while (input.hasNextInt()) { ++counts[input.nextInt() - 1]; }
        input.close();
        for (int i = 0; i < maxNumber; ++i) { System.out.println("counts [ " + (i + 1) + "] = " + counts[i]); }
    }
    public static void main(final String[] args) {
        final E_4_9_reader object = new E_4_9_reader();
        object.displayCounts();
    }
}

Creating the reader is a question of reading in the numbers from the file and then counting them. We do this by having an array where the index of the array is the number being counted and the value stored in that element of the array is the count. The hard bit is that arrays have index starting with 0 but the data does not use that value. We cheat a bit and subtract 1 from each data item in order to know which element of the array to use as the counter. The really hard bit is, of course, to remember to add one when we print things out!

public class E_4_9_reader {
    private void displayCounts() {
        final int maxNumber = 25;
        final FileInputStream input = new FileInputStream("random_numbers");
        final int[] counts = new int[maxNumber];
        while (input.hasNextInt()) { ++counts[input.nextInt() - 1]; }
        input.close();
        for (int i = 0; i < maxNumber; ++i) { System.out.println("counts [ " + (i + 1) + "] = " + counts[i]); }
    }
    public static void main(final String[] args) {
        final E_4_9_reader object = new E_4_9_reader();
        object.displayCounts();
    }
}

You will see when you run this, that although the counts of the numbers are all reasonable close together they are not the same. Herein lies the problem with computer generating random numbers: they aren’t actually random and it is hard with only a small number (1000 is small in this sort of situation) to get a good distribution. However, to go further is to diverge to far from the purpose of this book. There is a mass of literature on pseudo-random and quasi-random numbers. If you are interested it is well worth hunting out.

4.10 Using the data file created in the last question, write a drawing program that plots a bar chart showing how many of each integer between 1 and 25 appears in the file.
4.11 Write a method that takes two character array parameters and returns true if the sequence of characters stored in the second array is a subsequence of those characters stored in the first array.

We can create an answer to this question using indexing and iteration:

```java
import java.util.Arrays;

public class E_4_11 {
    private boolean containsSequence (final char[] a, final char[] b) {
        if (a.length >= b.length) {
            for (int i = 0; i <= a.length - b.length; ++i) {
                boolean areTheSame = true;
                for (int j = 0; j < b.length; ++j) {
                    if (a[i + j] != b[j]) { areTheSame = false; break; }
                }
                if (areTheSame) { return true; }
            }
        }
        return false;
    }
    private void assertTrue (final boolean b, final String message) {
        if (!b) { System.out.println ("Test failed: " + message); }
    }
    private void runTest () {
        char[] a = { 'a', 'b', 'c', 'd', 'e', 'f' };
        char[] b = { 'a', 'b', 'c', 'd' };
        assertTrue (containsSequence (a, a), "a == a");
        assertTrue (containsSequence (b, b), "b == b");
        assertTrue (containsSequence (a, b), "b is in a");
        assertTrue (containsSequence (b, a), "b is longer than a");
        char[] c = { 'c', 'd' };
        assertTrue (containsSequence (c, c), "c == c");
        assertTrue (containsSequence (a, c), "c is in a");
        assertTrue (containsSequence (c, a), "a is longer than c");
        assertTrue (containsSequence (b, c), "c is in b");
        assertTrue (containsSequence (c, b), "b is longer than c");
        char[] d = { 'b', 'z' };
        assertTrue (containsSequence (d, d), "d == d");
        assertTrue (containsSequence (a, d), "d is not in a");
        assertTrue (containsSequence (b, d), "d is not in b");
        assertTrue (containsSequence (c, d), "d is not in c");
    }
    public static void main (final String[] args) {
        (new E_4_11()).runTest();
    }
}
```

Note how we are written lots of tests for the method. In fact, these proved crucial since the first version of the method that was written failed the test labelled “Position3”. The problem was that we had written `i <= a.length - b.length` instead of `i <= a.length - b.length`. A minor slip, but a major error nonetheless. The tests found this and so stopped us issuing a program with a bug in it.

There is also a ‘trick’ solution for this question:

```java
private boolean containsSequence (final char[] a, final char[] b) {
    return true;
}
```
This works by converting each character array to a String and then using the method `String.contains` which tests to see if the parameter String is a subsequence of the String the method is called on.

Which of these two answers is better is a moot point. Without a context in which to compare and contrast the solutions, asking the question 'Which is better?' is fairly meaningless.

4.12 Write a method that takes two character array parameters and returns true if both arrays contain the same characters but not necessarily in the same order.

If the arrays contain the same characters then it implies that they are the same length and that one array is just a permutation of the other other array. For two sequences to be permutations of one another, the count of each value present in either sequences must be the same in both sequences. The 'brute force' method is to loop over one of the sequences checking the counts in both sequences of the each value in the chosen sequence. This involved an inner loop to pass over each item in both sequences creating a count.

```java
import java.util.Arrays;
public class E_4_12 {
    private boolean isPermutation ( final char[] a , final char[] b ) {
        if ( a.length != b.length ) { return false; }
        for ( int i = 0 ; i < a.length ; ++i ) {
            int countA = 0;
            int countB = 0;
            char c = a[i];
            for ( int j = 0 ; j < a.length ; ++j ) {
                if ( c == a[j] ) { ++countA ; }
                if ( c == b[j] ) { ++countB ; }
            }
            if ( countA != countB ) { return false; }
        }
        return true;
    }
    private void assertTrue ( final boolean b , final String message ) {
        if ( ! b ) { System.out.println ( "Test failed: " + message ) ; }
    }
    private void runTest ( ) {
        char[] a = { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' } ;
        char[] b = { 'a' , 'b' , 'c' , 'd' } ;
        assertTrue ( isPermutation ( a , a ) , "a is permutation of a" ) ;
        assertTrue ( isPermutation ( b , b ) , "b is permutation of b" ) ;
        assertTrue ( ! isPermutation ( a , b ) , "a is different from b" ) ;
        assertTrue ( ! isPermutation ( b , a ) , "b is different from a" ) ;
        char[] c = { 'e' , 'd' , 'c' , 'b' , 'a' } ;
        assertTrue ( isPermutation ( c , c ) , "c is permutation of c" ) ;
        assertTrue ( isPermutation ( a , c ) , "a is permutation of c" ) ;
        assertTrue ( isPermutation ( c , a ) , "c is permutation of a" ) ;
        char[] d = { 'a' , 'a' , 'c' , 'b' } ;
        char[] e = { 'a' , 'b' , 'b' , 'b' } ;
    }
}
```
Chapter 4: Introducing Containers

```java
public static void main ( final String[] args ) {
(new E_4_12 ( ) ).runTest ( ) ;
}

private boolean isPermutation ( final char[] a , final char[] b ) {
    if ( a.length != b.length ) { return false ; }
    char[] aa = (char[]) a.clone ( ) ;
    Arrays.sort ( aa ) ;
    char[] bb = (char[]) b.clone ( ) ;
    Arrays.sort ( bb ) ;
    return Arrays.equals ( aa , bb ) ;
}
```

This is actually quite inefficient if there are repeated values since it implies recalculating the count for the same value.

An alternative approach if the two sequences are permutations is to sort them in which case they should now be equal. Well we can’t sort the originals since that would disturb the original data, and that is unacceptable. So, we have to create copies (clone) the original and then we can sort the clones.

```java
import java.util.Arrays ;
import java.util.HashSet ;
```

```java
public class E_4_12_trick {
private boolean containsTheSame ( final Character[] a , final Character[] b ) {
    final HashSet<Character> hA = new HashSet<Character> ( Arrays.asList ( a ) ) ;
    final HashSet<Character> hB = new HashSet<Character> ( Arrays.asList ( b ) ) ;
    return hA.equals ( hB ) ;
}
```

Many people would consider this less good because of the resources required to make the clones and sort them. In fact, this may be a false position. The counting method is an \(O(n^2)\) algorithm, we have two nested loops both of which loop over the entire sequence. Cloning an array is \(O(n)\), comparing arrays is \(O(n)\), and sorting them is \(O(n\log_2(n))\) for a total of \(O(n(2 + \log_2(n)))\) which is \(O(n\log_2(n))\). So this method may actually be more efficient! For small arrays other factors dominate, but for very large arrays the difference could well be significant.

If it turns out that the number of items is not actually an issue then there is a ‘trick’ solution. If the number of times a character appears is not relevant, then we can use a set data structure (we will use HashSet) and ask if the two sets are equal:

```java
import java.util.Arrays ;
import java.util.HashSet ;
```

```java
public class E_4_12_trick {
private boolean containsTheSame ( final Character[] a , final Character[] b ) {
    final HashSet<Character> hA = new HashSet<Character> ( Arrays.asList ( a ) ) ;
    final HashSet<Character> hB = new HashSet<Character> ( Arrays.asList ( b ) ) ;
    return hA.equals ( hB ) ;
}
```

```java
private void runTest ( ) {
    Character[] a = {'a' , 'b' , 'c' , 'd' , 'e' , 'f'} ;
```
Character[] b = {'a', 'b', 'c', 'd'};
assertTrue (containsTheSame(a, a), "a contains the same as a");
assertTrue (containsTheSame(b, b), "b contains the same as b");
assertTrue (!containsTheSame(a, b), "a does not contain the same as b");
assertTrue (!containsTheSame(b, a), "b does not contain the same as a");
Character[] c = {'a', 'e', 'd', 'c', 'b', 'a'};
assertTrue (containsTheSame(c, c), "c contains the same as c");
assertTrue (containsTheSame(a, c), "a contains the same as c");
assertTrue (containsTheSame(c, a), "c is permutation of a");
Character[] d = {'a', 'a', 'a', 'b'};
Character[] e = {'a', 'b', 'b', 'b'};
assertTrue (containsTheSame(d, d), "d contains the same as d");
assertTrue (containsTheSame(e, e), "e contains the same as e");
assertTrue (containsTheSame(d, e), "d does not contain the same as e");
assertTrue (containsTheSame(e, d), "e does not contain the same as d");
}
public static void main (final String[] args) {
( new E_4_12_trick()).runTest();
}

Notice that we have made the data Character arrays in this version. We do this because we are making use of Collections classes and these cannot use primitive types. The trick here is to use the method Arrays.asList to create a Collections view of the array so that we can use them as the initializing data for the newly created HashSets. This creates two sets that can be compared. Sets have a single instance of any value so creating the sets from the arrays removes any duplicates. Comparing the sets for equality asks the question 'Do the two sets contain one or more instances of the same values?' which is close the question we were asked to answer.

4.13 Amend the AverageUsingArrayList program (Section 4.3.4, page 119) to take its input from a file instead of the keyboard. (It may well be best to start with the final version of the program discussed in that section!)

4.14 Amend the AddUpColumns program (Section 4.2.10, page 110) and the solution to Exercise 4.6 so as to read the numbers to total from a file.

4.15 Write a program that reads a text file and counts the number of times a selected word appears in the file.

4.16 Write a program that reads a text file containing Java source code and verifies that there is a matching number of opening and closing braces (curly brackets). Beware of braces appearing in comments, or in character literals and constants.
4.1 Write a program that sorts an array of integers into ascending order. Include a sort method that you write yourself, rather than using one from the JDK.

4.2 Write a program to act as a simple address book, storing names and addresses in a data file.

The following program is a basic one-class program implementing an address book storing a list of names and addresses. It provides a core set of operations to add, remove and search entries, along with the ability to load and save the entries from and to a text file. A simple text based user interface is implemented.

Two ArrayList<String>s are used to store the name and addresses, relying on the corresponding element in each ArrayList to represent the same address book entry. An entry is stored in a data file as two lines, the first containing the name string and the second the address string. Loading a data file appends the contents of the file to the current address book.

```java
import java.util.ArrayList;

public class C_4_2 {
    private static final int SEARCH = 1;
    private static final int LIST = 2;
    private static final int ADD = 3;
    private static final int REMOVE = 4;
    private static final int SAVE = 5;
    private static final int APPEND = 6;
    private static final int QUIT = 7;

    private Input in = new Input();
    private ArrayList<String> names = new ArrayList<String>();
    private ArrayList<String> addresses = new ArrayList<String>();

    public void go() {
        boolean quit = false;
        while (!quit) {
            displayMenu();
            int item = getMenuOption();
            if (item == QUIT) { quit = true; }
            else { doOption(item); }
        }
    }

    public void displayMenu() {
        System.out.println("Address Book");
        System.out.println(SEARCH + ". Search for an address");
        System.out.println(LIST + ". List all entries in the address book");
        System.out.println(ADD + ". Add an entry into the address book");
        System.out.println(REMOVE + ". Remove an entry from the address book");
        System.out.println(SAVE + ". Save the address book to a file");
        System.out.println(APPEND + ". Append the address book to a file");
        System.out.println(QUIT + ". Quit this program");
    }

    private void doOption(int item) {
        switch (item) {
            case SEARCH:
                // Search for an address
                break;
            case LIST:
                // List all entries in the address book
                break;
            case ADD:
                // Add an entry into the address book
                break;
            case REMOVE:
                // Remove an entry from the address book
                break;
            case SAVE:
                // Save the address book to a file
                break;
            case APPEND:
                // Append the address book to a file
                break;
            case QUIT:
                quit = true; // Exit the program
                break;
            default:
                System.out.println("Invalid option!");
                break;
        }
    }
}
```
System.out.println ( ADD + ". Add a new name and address" ) ;
System.out.println ( REMOVE + ". Remove a name and address" ) ;
System.out.println ( SAVE + ". Save address book to a file" ) ;
System.out.println ( APPEND + ". Append an address book from a file" ) ;
System.out.println ( QUIT + ". Quit" ) ;

public int getMenuOption ( ) {
    while ( true ) {
        System.out.print ( "Enter selection: " ) ;
        if ( in.hasNextInt ( ) ) {
            int temp = in.nextInt ( ) ;
            in.nextLine ( ) ;
            return temp ;
        }
        in.nextLine ( ) ;
        System.out.println( "Menu option not recognised, please try again" ) ;
    }
}

public void doOption ( int item ) {
    switch ( item ) {
    case SEARCH :
        searchForEntry ( ) ;
        break ;
    case LIST :
        listAllEntries ( ) ;
        break ;
    case ADD :
        addEntry ( ) ;
        break ;
    case REMOVE :
        removeEntry ( ) ;
        break ;
    case SAVE :
        saveAddressBook ( ) ;
        break ;
    case APPEND :
        appendAddressBook ( ) ;
        break ;
    default :
        System.out.println( "\nSorry - don't recognise that selection, try again" ) ;
    }
}

public void searchForEntry ( ) {
    System.out.println( "\nSearch address book" ) ;
    System.out.print ( "Enter a name: " ) ;
    String name = in.nextLine ( ) ;
    int index = names.indexOf(name) ;
    if (index != -1) { System.out.println( "The address is: " + addresses.get ( index ) ) ; }
    else { System.out.println( "Sorry - nothing found" ) ; }
}

public void listAllEntries ( ) {
    if ( names.size ( ) == 0 ) {

Chapter 4: Introducing Containers

System.out.println( "There are no entries in the address book" ) ;
    return ;
}
for ( int i = 0 ; i < names.size ( ) ; i++ ) {
    String name = names.get ( i ) ;
    String number = addresses.get ( i ) ;
    System.out.println( "Entry " + ( i + 1 ) + ": " ) ;
    System.out.println( " Name: " + name ) ;
    System.out.println( " Address: " + number ) ;
}

public void addEntry ( ) {
    System.out.println ( "Add entry to address book" ) ;
    System.out.print ( "Enter a name: " ) ;
    String name = in.nextLine ( ) ;
    System.out.print ( "Enter an address: " ) ;
    String address = in.nextLine ( ) ;
    names.add ( name ) ;
    addresses.add ( address ) ;
}

public void removeEntry ( ) {
    System.out.println ( "Remove entry from address book" ) ;
    System.out.print ( "Enter a name: " ) ;
    String name = in.nextLine ( ) ;
    remove( name ) ;
}

public void remove ( String name ) {
    int index = names.indexOf ( name ) ;
    if ( index != -1 ) {
        names.remove ( index ) ;
        addresses.remove ( index ) ;
        System.out.println ( "The entry for: " + name + " has been removed" ) ;
    }
    else { System.out.println ( "Name not found - nothing removed" ) ;
    }
}

public void saveAddressBook ( ) {
    System.out.println ( "Save address book to a file" ) ;
    System.out.print ( "Enter the file name: " ) ;
    String fileName = in.nextLine ( ) ;
    FileOutputStream out = new FileOutputStream ( fileName ) ;
    writeToFile ( out ) ;
    out.close ( ) ;
}

public void writeToFile ( FileOutputStream file ) {
    for ( int i = 0 ; i < names.size ( ) ; i++ ) {
        file.writeString ( names.get ( i ) ) ;
        file.writeEndOfLine ( ) ;
        file.writeString ( addresses.get ( i ) ) ;
        file.writeEndOfLine ( ) ;
    }
}
public void appendAddressBook () {
    System.out.println ( "
Append address book from a file" ) ;
    System.out.print ( "Enter the file name: " ) ;
    String fileName = in.nextLine ( ) ;
    FileInput fileIn = new FileInput ( fileName ) ;
    readFromFile ( fileIn ) ;
    fileIn.close ( ) ;
}

public void readFromFile ( FileInput file ) {
    while ( file.hasNextLine ( ) ) {
        String name = file.nextLine ( ) ;
        String address = file.nextLine ( ) ;
        names.add ( name ) ;
        addresses.add ( address ) ;
    }
}

public static void main(String[] args) {
    new C_4_2 ( ) .go ( ) ;
}

4.3 Write a program that reads a text file and builds an ArrayList containing each distinct word found in the file, in sorted order, along with a count of how many times each word appears.
5

Drawing Pictures
5.1 Use a Web browser to find the documentation for classes: Line2D, Line2D.Double, Graphics2D, BasicStroke and Color.

This is more of a ‘did you try’, type question, so trying to provide an answer is a bit fatuous.

5.2 Find class Arc2D.Double in the JDK documentation and work out how to draw arcs.

Another ‘did you try’ question, but this time there is a concrete deliverable. We should write a program to draw an arc. Basing things on the line drawing program on page 148, we came up with:

```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Arc2D;
import java.awt.geom.Arc2D.Double;

public class S_5_2 extends DrawPanel {
    public void paint ( final Graphics g ) {
        final Graphics2D g2d = (Graphics2D) g ;
        final Arc2D aLine = new Arc2D.Double ( 10 , 10 , 160 , 160 , 45 , 135 , Arc2D.OPEN ) ;
        g2d.draw ( aLine ) ;
    }
}

public static void main ( final String[] args ) {
    DrawFrame.display ( "S_5_2" , new S_5_2 ( ) ) ;
}
```

When executed we got the frame:

5.3 What needs to be done to rename class Drawing to class MyPicture?
Edit the source code to change the 'Drawing' to 'MyPicture' wherever it occurs. This can be done quickly by using the Find and Replace command in your editor.

5.4 How can the size of the drawing grid be changed?

Drawing programs are created by copying the class Drawing template and editing it. The template defines a main method, which looks like this:

```java
DrawFrame.display ( "Drawing" , new Drawing ( ) ) ;
```

The expression `new Drawing ( )` creates a new drawing with the default size of 300 by 300 pixels, as the parameter list following `Drawing` is empty. If two size parameter values are added to the parameter list then a drawing can be created with specified size. For example, this:

```java
DrawFrame.display ( "Drawing" , new Drawing ( 500 , 400 ) ) ;
```

will create a window with a grid 500 pixels in width and 400 pixels in height. You need to copy the entire class Drawing template for this to work correctly.

5.5 Where is the origin (position (0, 0)) of a drawing grid?

The origin is at the top left corner of a grid or panel. This follows from the way that the Java 2D graphics system defines its coordinate system.

5.6 How is the size and shape of an ellipse specified?

An ellipse is drawn using the `Ellipse2D` class from the Java class libraries. The size and position of an ellipse is specified by giving the rectangular boundary that it should be drawn in, with the width and height of the ellipse determined by the size of the rectangle. For example:

```java
Ellipse2D circle = new Ellipse2D.Double ( x , y , width , height ) ;
```

This displays an ellipse with a rectangular boundary of width by height in size, with the top left corner of the boundary at (x,y).
Note that *Ellipse2D* declares two nested subclasses *Ellipse2D.Float* and *Ellipse2D.Double*, which are used to instantiate ellipse objects, as *Ellipse2D* itself is abstract.

Once an ellipse object has been created it is drawn by calling methods like *draw* or *fill* from class *Graphics2D*.

### 5.7 How can you draw a circle?

A circle should be seen as a kind of ellipse and drawn using the *Ellipse2D* class (see previous question). If a square is specified as the boundary, the ellipse will actually be a circle. For example:

```java
e llipse2D circle = new Ellipse2D.Double ( 10 , 10 , 25 , 25 ) ;
```

Here, 10, 10 specifies the top left corner of the square boundary of the circle, with 25, 25 giving the width and height to give the size of the square (both should, of course, be equal).

### 5.8 What is the path of a dodecahedron?

#### Programming Exercises

### 5.1 Write a program to draw a hexagon using lines.

```java
import java.awt.Graphics ;
import java.awt.Graphics2D ;
import java.awt.geom.Line2D ;
import java.awt.geom.Line2D.Double ;
public class E_5_1 extends DrawPanel {
    public void paint ( final Graphics g ) {
        final Graphics2D g2d = (Graphics2D) g ;
        Line2D aLine = new Line2D.Double ( 100 , 100 , 200 , 100 ) ;
        g2d.draw ( aLine ) ;
        aLine = new Line2D.Double ( 200 , 100 , 250 , 150 ) ;
        g2d.draw ( aLine ) ;
        aLine = new Line2D.Double ( 250 , 150 , 200 , 200 ) ;
        g2d.draw ( aLine ) ;
        aLine = new Line2D.Double ( 200 , 200 , 100 , 200 ) ;
        g2d.draw ( aLine ) ;
        aLine = new Line2D.Double ( 100 , 200 , 50 , 150 ) ;
        g2d.draw ( aLine ) ;
        aLine = new Line2D.Double ( 50 , 150 , 100 , 100 ) ;
        g2d.draw ( aLine ) ;
    }
    public static void main ( final String[] args ) {
        DrawFrame.display ( "E_5_1" , new E_5_1 ( ) ) ;
    }
}
```
Programming Exercises

5.2 Write a program to draw a hexagon using class GeneralPath.

```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.GeneralPath;
public class E_5_2 extends DrawPanel {
    public void paint ( final Graphics g ) {
        final Graphics2D g2d = (Graphics2D) g;
        GeneralPath path = new GeneralPath ( ) ;
        path.moveTo ( 100, 100 ) ;
        path.lineTo ( 200 , 100 ) ;
        path.lineTo ( 250 , 150 ) ;
        path.lineTo ( 200 , 200 ) ;
        path.lineTo ( 100 , 200 ) ;
        path.lineTo ( 50 , 150 ) ;
        path.closePath ( ) ;
        g2d.draw ( path ) ;
    }
    public static void main ( final String[] args ) {
        JFrame.display ( "E_5_2" , new E_5_2 ( ) ) ;
    }
}
```

This seems a whole lot more sensible that using lines. In particular, we do not have to repeat the coordinates of points that are end points of one line and start points of another line. But then that is the whole point of a path.

5.3 Write a program to draw a collection of hexagons of different sizes and in different positions, using the GeneralPath you created in the last question.

The code:

```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.GeneralPath;
public class E_5_3 extends DrawPanel {
    public void paint ( final Graphics g ) {
        final Graphics2D g2d = (Graphics2D) g;
        GeneralPath path = new GeneralPath ( ) ;
        path.moveTo ( 100, 100 ) ;
        path.lineTo ( 200 , 100 ) ;
        path.lineTo ( 250 , 150 ) ;
        path.lineTo ( 200 , 200 ) ;
        path.lineTo ( 100 , 200 ) ;
        path.lineTo ( 50 , 150 ) ;
        path.closePath ( ) ;
        g2d.draw ( path ) ;
        g2d.scale ( 0.5 , 0.5 ) ;
        g2d.draw ( path ) ;
        g2d.translate ( -20 , -20 ) ;
        g2d.draw ( path ) ;
        g2d.translate ( 120 , 200 ) ;
```
5.4 Write a program to draw a picture of a house.

A basic house can be drawn using mostly rectangles with two lines to form the roof. This is primarily an exercise in determining the correct coordinates for drawing everything.

```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Line2D;
import java.awt.geom.Rectangle2D;
public class E_5_4 extends DrawPanel {
    public void paint ( final Graphics g ) {  
        final Graphics2D g2d = (Graphics2D) g ;
        g2d.draw ( new Line2D.Double ( 50 , 100 , 150 , 25 ) ) ;
        g2d.draw ( new Line2D.Double ( 150 , 25 , 250 , 100 ) ) ;
        g2d.draw ( new Rectangle2D.Double ( 130 , 190 , 40 , 60 ) ) ;
        g2d.draw ( new Rectangle2D.Double ( 50 , 100 , 200 , 150 ) ) ;
        g2d.draw ( new Rectangle2D.Double ( 70 , 195 , 40 , 30 ) ) ;
        g2d.draw ( new Rectangle2D.Double ( 190 , 195 , 40 , 30 ) ) ;
        g2d.draw ( new Rectangle2D.Double ( 70 , 125 , 40 , 30 ) ) ;
        g2d.draw ( new Rectangle2D.Double ( 190 , 125 , 40 , 30 ) ) ;
    }
    public static void main ( final String[] args ) {
        DrawFrame.display ( "E_5_4" , new E_5_4 ( ) ) ;
    }
}
```

The house looks like this:
5.5 Write a program that uses drawString to display your name and a message, using a large font.

5.6 Write a program to draw a sine wave.

5.7 As suggested in Section 5.2.4, page 147, write a method called createTriangle that can be used to create triangles of any size at any position. Use the method in a program to demonstrate that triangle drawing works correctly.

5.8 Repeat the last question but additionally allow the fill colour and outline width of triangles to be specified as well. (This will require some thinking about what actually goes in the createTriangle method.)

5.9 Rewrite program Chessboard so that it can display a board of any size in any position, as suggested in the design review section.

5.10 Write a program to display a shape like this one:

5.11 Write a program that displays a graph showing the curves $y = \sin x$, $y = \cos x$ and $y = \tan x$. 
Chapter 5: Drawing Pictures

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Line2D;
import java.awt.geom.Rectangle2D;

public class E_5_11 extends DrawPanel {
    public E_5_11 (final int w, final int h) {
        super (w, h);
    }

    public void paint (final Graphics g) {
        final Graphics2D g2d = (Graphics2D) g;
        final int width = getWidth();
        final int height = getHeight();
        final int x_zero = width / 2;
        final int y_zero = height / 2;
        final double y_max = 1.2;

        // Draw and label the x axis.
        g2d.draw(new Line2D.Double(0, y_zero, width, y_zero));
        for (int x = 60; x < x_zero; x += 60) {
            g2d.draw(new Line2D.Double(x_zero + x, y_zero, x_zero + x, y_zero + 5));
            g2d.draw(new Line2D.Double(x_zero - x, y_zero, x_zero - x, y_zero + 5));
            g2d.drawString("" + x, x_zero + x - 10, y_zero + 20);
            g2d.drawString("-" + x, x_zero - x - 15, y_zero + 20);
        }

        // Draw the and label the y axis.
        g.drawLine(x_zero, 0, x_zero, height);
        for (double y = 0.2; y < y_max; y += 0.2) {
            final int y_pixel = (int) (y * (y_zero / y_max));
            g2d.draw(new Line2D.Double(x_zero, y_pixel, x_zero + 5, y_pixel));
            g2d.draw(new Line2D.Double(x_zero, y_pixel + y_zero, x_zero + 5, y_pixel + y_zero));
            String label = ("" + y).substring(0, 3);
            g.drawString(label, x_zero + 8, y_zero - y_pixel + 5);
            g.drawString(label, x_zero + 8, y_zero + y_pixel + 5);
        }

        // Draw the curves.
        for (double x = -x_zero; x < x_zero; x += 0.2) {
            g2d.setColor(Color.red);
            double y = Math.cos(Math.toRadians(x));
            int y_pixel = y_zero - (int) (y * (y_zero / y_max));
            g2d.draw(new Rectangle2D.Double((int) x + x_zero, y_pixel, 1, 1));
            g2d.setColor(Color.blue);
            y = Math.sin(Math.toRadians(x));
            y_pixel = y_zero - (int) (y * (y_zero / y_max));
            g2d.draw(new Rectangle2D.Double((int) x + x_zero, y_pixel, 1, 1));
            g2d.setColor(Color.green);
            y = Math.tan(Math.toRadians(x));
            y_pixel = y_zero - (int) (y * (y_zero / y_max));
            g2d.draw(new Rectangle2D.Double((int) x + x_zero, y_pixel, 1, 1));
        }
    }

    public static void main (final String[] args) {
        DrawFrame.display("Curves", new E_5_11(700, 480));
    }
}

This program displays the following:
Most of the detail in this program is about getting the visual presentation neat and tidy, so that the displayed graph fills the window whatever size the window is set to. The size specified in the main method to 700 by 400, but this can be changed to any sensible size.

5.12 Extend the program Graph allowing the graph to be placed in any position and to be of any size, with multiple lines given data from multiple files.

Challenges

5.1 Write a program to draw a random collection of shapes, with different fills and outline widths.

Hint: The JDK has a class that generates random numbers, which can be employed to determine the properties of the shapes drawn.

5.2 Write a program that draws a series of curves using the classes CubicCurve2D.Double and QuadCurve2D.Double.

5.3 Draw a picture of a Mandelbrot set.

The Mandelbrot set is a kind of fractal that has an appealing visual representation. The drawing program listed below displays this basic Mandelbrot:
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Line2D;
import java.awt.geom.Line2D.Double;

public class C_5_3 extends DrawPanel {
    public C_5_3 () {
    }
    public C_5_3 (final int w, final int h) {
        super (w, h);
    }
    @Override
    public void paint (final Graphics g) {
        final Graphics2D g2d = (Graphics2D) g;
        int numberOfColours = 8;
        Color colours[] = new Color[] {
            Color.black, Color.red, Color.green, Color.blue,
            Color.cyan, Color.magenta, Color.yellow, Color.white
        };
        int columns = getWidth();
        int rows = getHeight();
        int iterations = 256;
        int size = 4;

        for (int column = 0; column < columns; column++) {
            for (int row = 0; row < rows; row++) {
                double x = 0.0;
                double y = 0.0;
                int colour;
                for (colour = 0; colour < iterations; colour++) {
                    if ((x * x + y * y) > size) {
                        break;
                    }
                    double newy = 2 * x * y + ((row * 0.01) - 1.5);
                    x = x * x - y * y + ((column * 0.01) - 2.0);
                    y = newy;
                }
                g2d.setColor(colours[colour % numberOfColours]);
                final Line2D point = new Line2D.Double(column, row, column, row);
                g2d.draw(point);
            }
        }
    }
    public static void main (final String[] args) {
        DrawFrame.display("Mandelbrot", new C_5_3 ());
    }
}

5.4 Write a program to create an array of squares where every square is a different colour.
6 Classes and Objects
6.1 Why is an abstract data type abstract?

Because there is no concept of representation involved – an abstract data type specifies the behaviour of values of the type, it does not say how the type should be represented in a specific implementation.

6.2 How is the state of an object represented?

By the values associated with fields declared in the class.

6.3 What is object identity?

The identity of an object is the way that it is uniquely identified. For the JVM, references to objects are the identities of the objects. Thus, \( a == b \), where \( a \) and \( b \) are variables that are references to objects, is only true if \( a \) and \( b \) refer to the same object, i.e. the identities of the objects are the same.

6.4 What determines the public interface of an object?

The set of public methods.

6.5 Define an ADT for a set data type.

6.6 What are the rules for method overloading?

If two or more methods in a class have the same name but different signatures then the method is said to be overloaded. The signature of the methods is determined by the number and type of the parameters to the method. The return type of the methods is not considered to be part of the signature for the purposes of overloading.

6.7 Why should instance variables not be made public?
If any of the instance variables are public then the representation of the type is being exposed to code using the class. This means that it can become impossible to make changes to the representation without having to rewrite code that uses the class. The strategy of providing accessor methods separates use and representation so that changes of representation are easy and have no consequences outside the class itself.

**6.8 How is object encapsulation enforced?**

By ensuring that nothing of the state of an object is visible from outside the object except via the public interface.

**6.9 How many ways can an instance variable be initialized?**

Three:

1. By direct initialization of the field.
2. By assignment in an instance initializer.
3. By assignment in a constructor.

**6.10 What would a private constructor mean?**

Only methods within the class can create objects of the class. This is a commonly used technique: The class has static methods (called factory methods) that return newly created objects. This design means that the class itself can keep track of all created objects and/or apply specific sorts of creation strategies. Design patterns such as Factory Method, Abstract Factory, etc. make use of this technique.

**6.11 What is a shared object?**

A shared object is one that has more than one variable holding a reference to it, i.e. more than one reference to the shared object is being used at the same time in the program.

**6.12 What is an unreachable object?**

An unreachable object is one for which there is no chain of references from the main entry that end up referring to the object. This is an important concept since it drives garbage collection: any unreachable object is garbage and can be collected.

**6.13 How do object references work when passing parameters and performing assignment?**
References are passed by value and create shared objects.

6.14 Why can’t static methods access instance variables and instance methods?

Instance variables and instance methods require there to be an object to access those variables and methods. Static methods can be called without a particular object being referred to and so cannot access instance variables and methods.

6.15 What is this?

**this** is a keyword that is either:

1. the name of the reference to the current object; or
2. the name of an overloaded constructor.

6.16 Devise a comprehensive test plan for class Matrix.

6.17 Outline the differences between procedural and object-oriented programming.

This is actually a deep and complicated issue when addressed fully. The following is the (relatively glib) short answer.

Procedural programming is about creating a hierarchy of function calls that act on the local and global state. Object oriented programming is about creating a system of interacting objects where objects encapsulate state and offer services via their public interface.

### Programming Exercises

6.1 Write and test a class to represent phone numbers. To support generality and future expansion, international codes, area codes and the phone number within the area should be stored separately from each other.

We chose to represent phone number as a trio of character arrays. Of course, this means checking that we only have decimal digits but that is straightforward. There are in fact rules (specified by the ITU) of what constitutes a phone number, so we work with those.
import java.util.Arrays;
/**
 * A class to represent a telephone number. ITU-T E164 requires phone numbers to contain at most 15
 * digits, with a maximum of 3 digits for the country code. All digits must be decimal digits, i.e. in the
 * range [0-9]. Allowed print formats of phone number are specified by ITU-T E123.
 *
 * @author Russel Winder
 * @version 2006-11-06
 */
public class PhoneNumber {
    // Store the components of the number as arrays of characters. This means though that we must ensure we
    // only have decimal digit characters.
    private char[] country;
    private char[] area;
    private char[] subscriber;
    private void checkDigits (final char[] array) {
        for (char c : array) {
            if ((c < '0') || (c > '9')) { throw new RuntimeException("Digit not in the range [0-9]."); }
        }
    }
    private void checkCorrectness () {
        if (country.length > 3) {
            throw new RuntimeException("Phone numbers can contain at most 3 decimal digits in the country code.");
        }
        if (country.length + area.length + subscriber.length > 15) {
            throw new RuntimeException("Phone numbers can contain at most 15 decimal digits.");
        }
        checkDigits (country);
        checkDigits (area);
        checkDigits (subscriber);
    }
    public PhoneNumber (final char[] country, final char[] area, final char[] subscriber) {
        this.country = country;
        this.area = area;
        this.subscriber = subscriber;
        checkCorrectness ();
    }
    public PhoneNumber (final String country, final String area, final String subscriber) {
        this.country = new char[country.length()];
        country.getChars(0, country.length(), this.country, 0);
        this.area = new char[area.length()];
        area.getChars(0, area.length(), this.area, 0);
        this.subscriber = new char[subscriber.length()];
        subscriber.getChars(0, subscriber.length(), this.subscriber, 0);
        checkCorrectness ();
    }
    public String toStringNationalFormat () {
        final StringBuilder sb = new StringBuilder();
        sb.append('(');
        for (char c : area) { sb.append(c); }
        sb.append(') ');  
        for (char c : subscriber) { sb.append(c); }
      return sb.toString();
    }
    public String toStringInternationalFormat () {
        final StringBuilder sb = new StringBuilder();
        sb.append('+');
Chapter 6: Classes and Objects

```java
sb.append ( '+' ) ;
for ( char c : country ) { sb.append ( c ) ; }
sb.append ( '' ) ;
for ( char c : area ) { sb.append ( c ) ; }
sb.append ( '' ) ;
for ( char c : subscriber ) { sb.append ( c ) ; }
return sb.toString () ;
}
public String toString () { return toStringInternationalFormat () ; }
}

Notice that we provide a way of specifying phone numbers using strings as well as arrays of characters. It is important to give users of the class easy ways to use it. Specifying everything with character arrays is awkward, so we provide the ability to use strings. The awkwardness of character arrays is exemplified in our test program.

```java
public class PhoneNumber_Test {
    private void assertTrue ( final String message , final String expected , final String actual ) {
        if ( ! expected.equals ( actual ) ) {
            System.out.println ( "assertTrue failed: " + message + "; expected " + expected + ", but got " + actual ) ;
        }
    }
    private void assertRuntimeException ( final String country , final String area , final String subscriber ) {
        try {
            final PhoneNumber pn = new PhoneNumber ( country , area , subscriber ) ;
        } catch ( final RuntimeException re ) {
            return ;
        }
        System.out.println ( "assertRuntimeException failed: " + country + " + area + " + subscriber ) ;
    }
    private void runTests ( ) {
        PhoneNumber pn = new PhoneNumber ( new char [] { '4' , '4' } ,
                                                new char [] { '2' , '0' } ,
                                                new char [] { '7' , '5' , '8' , '5' , '2' , '2' , '0' , '0' } ) ;
        assertTrue ( "String constructor, international format", "+44 20 75852200" , pn.toStringInternationalFormat () ) ;
        assertTrue ( "String constructor, national format", "(20) 75852200" , pn.toStringNationalFormat () ) ;
        pn = new PhoneNumber ( "44" , "20" , "75852200" ) ;
        assertRuntimeException ( "String constructor, international format", "+44 20 75852200" , pn.toStringInternationalFormat () ) ;
        assertRuntimeException ( "String constructor, national format", "(20) 75852200" , pn.toStringNationalFormat () ) ;
        assertTrue ( "String constructor, international format", "+44 20 75852200" , pn.toStringInternationalFormat () ) ;
        assertRuntimeException ( "1234" , "5678" , "9012345" ) ;
        assertRuntimeException ( "123" , "456" , "7890123456" ) ;
    }
    public static void main ( final String[] args ) {
        ( new PhoneNumber_Test ( ) ).runTests ( ) ;
    }
}

6.2 Write and test a class to represent dates. (We know there is a Date class in the JDK and that we normally advocate using the JDK whenever possible, but this is an exercise in building a class and understanding the issues in representing dates — you would of course use the JDK class in a production program; or more likely GregorianCalendar.)

6.3 Extend the Chessboard class (Section 5.5.1, page 156) so that it can be used to create objects for drawing
This is really a question of adding fields to store the row and column count and constructors to allow them to be set. We ensure that the default constructor creates an $8 \times 8$ chessboard:

```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Color;

/**
 * A program to draw a chessboard.
 *
 * @author Graham Roberts
 * @author Russel Winder
 * @version 2005-08-12
 */
public class Chessboard extends DrawPanel {
    private final int rows;
    private final int columns;
    public Chessboard() {
        this(8, 8);
    }
    public Chessboard(final int rows, final int columns) {
        this.rows = rows;
        this.columns = columns;
    }
    public Chessboard(final int rows, final int columns, final int w, final int h) {
        super(w, h);
        this.rows = rows;
        this.columns = columns;
    }
    public void paint(final Graphics g) {
        final Graphics2D g2d = (Graphics2D) g;
        final int squareSize = getWidth() / rows;
        Color colour = Color.green;
        for (int row = 0; row < rows; ++row) {
            for (int column = 0; column < columns; ++column) {
                g2d.setColor(colour);
                g2d.fillRect(row * squareSize, column * squareSize, squareSize, squareSize);
                if ((column != (columns - 1)) || ((columns % 2) != 0)) { colour = (colour == Color.red) ? Color.green : Color.red; }
            }
        }
    }
    public static void main(final String[] args) {
        DrawFrame.display("Chessboard", new Chessboard(9, 9));
    }
}
```

Other changes of note:

- We had to change the size of each square from being fixed to being calculated according to the size of the drawing area.
- We had to change the algorithm for swapping colours since previously it had assumed that the number of columns was an even number.
Chapter 6: Classes and Objects

The chessboard can now be used for Shogi, which uses a 9×9 board. Or even the more extreme forms of draughts (aka checkers) where boards as big as 12×12 are used.

6.4 Write a comprehensive test class for the Stack<T> class. In particular provide tests for Stack<Double>, Stack<Boolean> and Stack<Stack<Integer>>.

6.5 Write and test a pie-chart panel class to display pie-charts, following the structure shown by class GraphPanel.

6.6 Implement matrix subtraction and multiplication methods for class Matrix. Extend the test class so as to provide suitable tests for the new code.

We add methods subtract and multiply to the class:

```java
/**
 * A basic matrix class.
 * @author Graham Roberts and Russel Winder
 * @version 2006-11-18
 */
public class Matrix {
    private double[][] elements ;
    /**
     * Create a <code>Matrix</code> with a given size and each element initialized to 0.0. If a
     * number less than one is given for either or both of the parameters then create a 0x0
     * representation.
     */
    public Matrix (final int rows , final int columns ) {
        elements = ( rows < 1 ) || ( columns < 1 )?
        new double[0][0] :
        new double[rows][columns] ;
    }
    /**
     * Conversion constructor that takes a 2D array of <code>double</code>s and creates a
     * <code>Matrix</code>. Assumes that the array that is the parameter is a rectangular structure,
     * i.e. all rows have the same number of columns -- perhaps ought to check this?
     */
    public Matrix (final double[][] elements ) {
        this.elements = (double[][]) elements.clone ( ) ;
    }
    public int getNumberOfRows ( ) {
        return elements.length ;
    }
    public int getNumberOfColumns ( ) {
        return elements[0].length ;
    }
    public double getElement (final int row , final int column ) {
        return isValidElement (row , column ) ?
        elements[row][column] : 0.0 ;
    }
    public void setElement (final int row , final int column , final double value ) {
        if ( isValidElement (row , column ) ) {
            elements[row][column] = value ;
        }
    }
    /**
     * @return a new matrix that is the sum of this matrix and the parameter matrix.
     */
    public Matrix add (final Matrix m ) {
        if ( ( (getNumberOfRows ( ) != m.getNumberOfRows ( ) ) ||
            (getNumberOfColumns ( ) != m.getNumberOfColumns ( ) ) ) ) {
            return null ;
        }
```

```java
```
**Programming Exercises**

```java
final Matrix result = new Matrix ( getNumberOfRows () , getNumberOfColumns () ) ;
for ( int row = 0 ; row < getNumberOfRows () ; ++row ) {
    for ( int column = 0 ; column < getNumberOfColumns () ; ++column ) {
        result.setElement ( row , column , getElement ( row , column ) + m.getElement ( row , column ) ) ;
    }
}
return result ;
/**
 * @return a new matrix that is the difference of this matrix and the parameter matrix.
 */
public Matrix subtract ( final Matrix m ) {
    if ( ( getNumberOfRows () != m.getNumberOfRows () ) ||
         ( getNumberOfColumns () != m.getNumberOfColumns () ) ) { return null ; }
    final Matrix result = new Matrix ( getNumberOfRows () , getNumberOfColumns () ) ;
    for ( int row = 0 ; row < getNumberOfRows () ; ++row ) {
        for ( int column = 0 ; column < getNumberOfColumns () ; ++column ) {
            result.setElement ( row , column , getElement ( row , column ) - m.getElement ( row , column ) ) ;
        }
    }
    return result ;
/**
 * @return a new matrix that is the product of this matrix and the parameter matrix.
 */
public Matrix multiply ( final Matrix m ) {
    if ( ( getNumberOfRows () != m.getNumberOfColumns () ) ||
         ( getNumberOfColumns () != m.getNumberOfRows () ) ) { return null ; }
    final Matrix result = new Matrix ( getNumberOfRows () , m.getNumberOfColumns () ) ;
    for ( int row = 0 ; row < result.getNumberOfRows () ; ++row ) {
        for ( int column = 0 ; column < result.getNumberOfColumns () ; ++column ) {
            double sum = 0.0 ;
            for ( int i = 0 ; i < getNumberOfColumns () ; ++i ) {
                sum += getElement ( row , i ) * m.getElement ( i , column ) ;
            }
            result.setElement ( row , column , sum ) ;
        }
    }
    return result ;
}
public String toString ( ) {
    final StringBuilder sb = new StringBuilder ( ) ;
    for ( int row = 0 ; row < getNumberOfRows () ; ++row ) {
        sb.append ( "| " ) ;
        for ( int column = 0 ; column < getNumberOfColumns () ; ++column ) {
            sb.append ( getElement ( row , column ) + " " ) ;
        }
        sb.append ( "|\n" ) ;
    }
    return sb.toString ( ) ;
}
private boolean isValidElement ( final int row , final int column ) {
    return ( row > -1 ) && ( row < getNumberOfRows () ) && ( column > -1 ) && ( column < getNumberOfColumns () ) ;
}
```
and then create a framework for testing the code:

```java
/**
 * A test for the <code>Matrix</code> class. This is not really a proper unit test it just ensures the code is
 * not obviously broken.
 *
 * @author Russel Winder
 * @version 2006-11-18
 */

class Matrix_Test {
    private void assertEquals ( final String m , final Matrix a , final Matrix b ) {
        final int rowCount = a.getNumberOfRows ( ) ;
        final int columnCount = a.getNumberOfColumns ( ) ;
        if ( ( rowCount != b.getNumberOfRows ( ) ) || ( columnCount != b.getNumberOfColumns ( ) ) ) {
            System.out.println ( m + " -- matrices not of same shape." ) ;
        } else {
            for ( int row = 0 ; row < rowCount ; ++row ) {
                for ( int column = 0 ; column < columnCount ; ++column ) {
                    if ( ( a.getElement ( row , column ) - b.getElement ( row , column ) ) > 0.00000000000001 ) {
                        System.out.println ( m + " -- values not equal." ) ;
                        System.out.println ( "\ta = " + a ) ;
                        System.out.println ( "\tb = " + b ) ;
                        return ;
                    }
                }
            }
        }
    }

    private void testAdd ( ) {
        final Matrix m1 = new Matrix ( new double[][] { { 1.2 , 2.5 , 4.5 } , { 3.9 , 4.2 , 0.9 } } ) ;
        final Matrix m2 = new Matrix ( new double[][] { { 1.3 , 7.5 , 5.2 } , { 4.8 , 8.3 , 9.1 } } ) ;
        final Matrix expected = new Matrix ( new double[][] { { 2.5 , 10.0 , 9.7 } , { 8.7 , 12.5 , 10.0 } } ) ;
        Matrix result = m1.add ( m2 ) ;
        assertEquals ( "Add 1" , result , expected ) ;
        result = m2.add ( m1 ) ;
        assertEquals ( "Add 2" , result , expected ) ;
    }

    private void testSubtract ( ) {
        final Matrix m1 = new Matrix ( new double[][] { { 1.2 , 2.5 , 4.5 } , { 3.9 , 4.2 , 0.9 } } ) ;
        final Matrix m2 = new Matrix ( new double[][] { { 1.3 , 7.5 , 5.2 } , { 4.8 , 8.3 , 9.1 } } ) ;
        Matrix result = m1.subtract ( m2 ) ;
        assertEquals ( "Subtract 1" , result , new Matrix ( new double[][] { { -0.1 , -5.0 , -0.7 } , { -0.9 , -4.1 , -8.2 } } ) ) ;
        result = m2.subtract ( m1 ) ;
        assertEquals ( "Subtract 2" , result , new Matrix ( new double[][] { { 0.1 , 5.0 , 0.7 } , { 0.9 , 4.1 , 8.2 } } ) ) ;
    }

    private void testMultiply ( ) {
        final Matrix m1 = new Matrix ( new double[][] { { 1.2 , 2.5 , 4.5 } , { 3.9 , 4.2 , 0.9 } } ) ;
        final Matrix m2 = new Matrix ( new double[][] { { 1.3 , 7.5 , 5.2 } , { 4.8 , 8.3 , 9.1 } } ) ;
        Matrix result = m1.multiply ( m2 ) ;
        assertEquals ( "Multiply 1" , result , new Matrix ( new double[][] { { 51.91 , 61.95 } , { 34.38 , 57.60 } } ) ) ;
        result = m2.multiply ( m1 ) ;
        assertEquals ( "Multiply 2" , result , new Matrix ( new double[][] { { 30.81 , 34.75 , 12.60 } , { 24.96 , 53.16 , 27.72 } , { 45. } } ) ;
    }

    public static void main ( final String[] args ) {
        final Matrix_Test mt = new Matrix_Test ( ) ;
        mt.testAdd ( ) ;
```
This is not really a comprehensive test, we should try with many more values and especially we should test lots of the error conditions. However, we avoid doing this now since error handling should involve exceptions and the TestNG framework should be used for testing anyway.

6.7 Write and test a sparse matrix class which stores only values explicitly added to a matrix. All other values are assumed to be zero. The class should not allocate any variables or objects to represent matrix elements that have not been added.

There are two answers presented here. The first is an answer that can be created with the material we have covered so far. The problem is that this is a far from good solution, but features of Java not yet covered are required for the second, better solution. We present this second solution here, even though it uses features not covered till the next two chapters, to ensure that the point about ‘good algorithm’ and ‘good use of library’ is made. Feel free to ignore the second solution initially, but do return to it after having covered inheritance and exceptions.

What the question is asking us to do is to store the data on an as needed basis. We assume that the entry in the matrix is 0.0 unless a specific value is held in some data structure. Clearly this is leading us towards having a container of entries where each entry has integers for row and column position and a double for the non-zero value. So we need a class to represent the entries and a container to hold them in. For a container, we can use an ArrayList. When we need to find an item, we search the ArrayList and if we find an entry with the right row and column, we return the value of the datum, otherwise we return 0.0. Something along the lines of:

```java
import java.util.ArrayList;
import java.util.Iterator;
public class SparseMatrix {
    private static class Entry {
        private final int row;
        private final int column;
        private double datum;
        public Entry (final int row, final int column, final double datum) {
            this.row = row;
            this.column = column;
            this.datum = datum;
        }
        public int getRow () { return row; }
        public int getColumn () { return column; }
        public double getDatum () { return datum; }
        public void setDatum (final double datum) { this.datum = datum; }
    }
    private final ArrayList<Entry> entries = new ArrayList<Entry> ( );
    private final int rows ;
    private final int columns ;
    public SparseMatrix ( final int rows , final int columns ) {
```
Chapter 6: Classes and Objects

```java
this.rows = rows;
this.columns = columns;
}
public int getNumberOfRows() { return rows; }
public int getNumberOfColumns() { return columns; }
public double getElement(final int row, final int column) {
    if (isValidElement(row, column)) {
        final Entry e = getEntry(row, column);
        if (e != null) { return e.getDatum(); }
    }
    return 0.0;
}
public void setElement(final int row, final int column, final double value) {
    if (isValidElement(row, column)) {
        final Entry e = getEntry(row, column);
        if (e != null) {
            if (value != 0) { e.setDatum(value); }
            else { entries.remove(e); }
        }
        else { if (value != 0.0) { entries.add(new Entry(row, column, value)); } }
    }
}
public SparseMatrix add(final SparseMatrix m) {
    if ((getNumberOfRows() != m.getNumberOfRows()) ||
        (getNumberOfColumns() != m.getNumberOfColumns())) { return null; }
    final SparseMatrix result = new SparseMatrix(getNumberOfRows(), getNumberOfColumns());
    for (int row = 0; row < getNumberOfRows(); ++row) {
        for (int column = 0; column < getNumberOfColumns(); ++column) {
            result.setElement(row, column, getElement(row, column) + m.getElement(row, column));
        }
    }
    return result;
}
public SparseMatrix subtract(final SparseMatrix m) {
    if ((getNumberOfRows() != m.getNumberOfRows()) ||
        (getNumberOfColumns() != m.getNumberOfColumns())) { return null; }
    final SparseMatrix result = new SparseMatrix(getNumberOfRows(), getNumberOfColumns());
    for (int row = 0; row < getNumberOfRows(); ++row) {
        for (int column = 0; column < getNumberOfColumns(); ++column) {
            result.setElement(row, column, getElement(row, column) - m.getElement(row, column));
        }
    }
    return result;
}
public SparseMatrix multiply(final SparseMatrix m) {
    if ((getNumberOfRows() != m.getNumberOfColumns()) ||
        (getNumberOfColumns() != m.getNumberOfRows())) { return null; }
    final SparseMatrix result = new SparseMatrix(getNumberOfRows(), m.getNumberOfColumns());
    for (int row = 0; row < result.getNumberOfRows(); ++row) {
        for (int column = 0; column < result.getNumberOfColumns(); ++column) {
            double sum = 0.0;
            for (int i = 0; i < m.getNumberOfColumns(); ++i) {
                sum += getElement(row, i) * m.getElement(i, column);
            }
            result.setElement(row, column, sum);
        }
    }
    return result;
}
```

and we can test that this is not totally broken with a small test program. To use this usefully, assertions must be set to on (-ea option to the java command) on when running this or it will seem like tests pass even if they do not. This is why we put the always failing assertion at the end just to ensure that we have. Not really the best thing to do, agreed, but, we haven’t covered using TestNG as our test framework yet so this will do for now.

```java
public class SparseMatrix_Test {
    private void assertEquals ( final String m , final SparseMatrix a , final SparseMatrix b ) {
        final int rowCount = a.getNumberOfRows ( ) ;
        final int columnCount = a.getNumberOfColumns ( ) ;
        if ( ( rowCount != b.getNumberOfRows ( ) ) || ( columnCount != b.getNumberOfColumns ( ) ) ) {
            System.out.println ( m + " -- matrices not of same shape." ) ;
        } else {
            for ( int row = 0 ; row < rowCount ; ++row ) {
                for ( int column = 0 ; column < columnCount ; ++column ) {
                    if ( Math.abs ( a.getElement ( row , column ) - b.getElement ( row , column ) ) > 0.00000000000001 ) {
                        System.out.println ( m + " -- values not equal." ) ;
                        System.out.println ( "\ta = " + a ) ;
                        System.out.println ( "\tb = " + b ) ;
                        return ;
                    }
                }
            }
        }
    }

    private void testCreate ( ) {
        final SparseMatrix m = new SparseMatrix ( 10000 , 20000 ) ;
    }
}
```
m.setElement (200, 300, 4.5);
assert m.getElement (0, 0) == 0.0;
assert m.getElement (1000, 2000) == 0.0;
assert m.getElement (200, 300) == 4.5;
}

private void testAdd () {
    final SparseMatrix m1 = new SparseMatrix (3, 3);
m1.setElement (0, 1, 2.0);
    final SparseMatrix m2 = new SparseMatrix (3, 3);
m2.setElement (1, 0, 2.0);
    final SparseMatrix expected = new SparseMatrix (3, 3);
    expected.setElement (0, 1, 2.0);
    expected.setElement (1, 0, 2.0);
    SparseMatrix result = m1.add (m2);
    assertEquals ("Add 1", result , expected);
    result = m2.add (m1);
    assertEquals ("Add 2", result , expected);
}

private void testSubtract () {
    final SparseMatrix m1 = new SparseMatrix (3, 3);
m1.setElement (0, 1, 2.0);
    final SparseMatrix m2 = new SparseMatrix (3, 3);
m2.setElement (1, 0, 2.0);
    final SparseMatrix expected = new SparseMatrix (3, 3);
    expected.setElement (0, 1, 2.0);
    expected.setElement (1, 0, -2.0);
    SparseMatrix result = m1.subtract (m2);
    assertEquals ("Subtract 1", result , expected);
    expected.setElement (0, 1, -2.0);
    expected.setElement (1, 0, 2.0);
    result = m2.subtract (m1);
    assertEquals ("Subtract 2", result , expected);
}

private void testMultiply () {
    final SparseMatrix m1 = new SparseMatrix (3, 3);
m1.setElement (0, 1, 2.0);
    final SparseMatrix m2 = new SparseMatrix (3, 3);
m2.setElement (1, 0, 2.0);
    final SparseMatrix expected = new SparseMatrix (3, 3);
    expected.setElement (0, 0, 4.0);
    SparseMatrix result = m1.multiply (m2);
    assertEquals ("Multiply 1", result , expected);
    expected.setElement (0, 0, 0.0);
    expected.setElement (1, 1, 4.0);
    result = m2.multiply (m1);
    assertEquals ("Multiply 2", result , expected);
}

private void run () {
    testCreate ();
    testAdd ();
    testSubtract ();
    testMultiply ();
}

public static void main (final String[] args) {
( new SparseMatrix_Test () ).run ();
assert 1 == 0;
Of course, storing a sparse matrix is all very well but as soon as operations are added, the result may not actually be sparse!

Note that the `add`, `subtract`, and `multiply` operations are basically the same as the class `Matrix` since they were coded up in a way that is dependent only on the public interface and not on the representation. This is leading up to material in the next chapter about inheritance – we can create a class to hold the operations and then inherit from it for the different representations. More on this issue later – in the next chapter, in fact.

Although, the above answers the question as set, it is not a good way of implementing a sparse matrix. Using an array to store the items is a very inefficient way of doing things. What we really need is a container that maps the pairs `(row, column)` to the datum. Such a data structure is called a `map`: a matrix is a map between pairs of integers and values, the integers are the row and column index values. So for an $m \times n$ matrix the map keys are the pairs $(a, b)$ where $0 < a < m$ and $0 < b < n$. Thus a sparse matrix is a map with keys that are pairs of integers but where the vast majority of values are 0. So instead of storing the whole array, we create a map that contains only the non-zero items.

So how are we going to represent this map? If we look at the JDK documentation, we see that there are a number map types already implemeted in the standard library – in particular `TreeMap` and `HashMap`. It seems sensible therefore to spend the time learning to use the standard types rather than devising our own implementation – a map of this sort we want is exactly a general purpose map, we have no special requirements.

Of course, it is at this point that we need to know more that we have covered so far. To use a `TreeMap`, or `HashMap`, we need to know a bit about inheritance and overloading, and also a bit about exceptions. So you may want to wait reading the rest of this answer till after you have studied those features of the language.

Since we have to store a pair of integers as a single key value, we need to spend time thinking about how to manage pairs of values and provide all the methods needed for the pairs to be used in the map type. The JDK has no predefined pair type so we have to create our own. We chose to do this as a top-level nested class so as to minimize the visibility of this specialized type.

To use a `TreeMap`, we have to ensure that our pair type has a *natural order*, i.e. a total order. The type must implement the interface `Comparable`, and we must implement the methods `equals` and `compareTo`, defining a total order on our pair of integer types. Unfortunately, it is not possible to define a good total order on a pair of integers – we could try but it would be something of a hack.

So what about using a `HashMap`? To use this type we have to implement `equals` and `hashCode` in our pair type. `equals` is easy, but the `hashCode` function has a quite strict requirement, see the online manual on `Object.hashCode`. In this particular case things are relatively straightforward, we can map
the (row, column) to a unique integer: we can flatten the integer pair to a sequence of integers, and whilst this is good for hash codes it is not a good way of creating a total order. So we use the expression row * columns + column to ‘linearize’ the integer pair, and can therefore use `HashMap`:

```java
import java.util.HashMap;
public class SparseMatrix {
    private class Index {
        private final int row;
        private final int column;
        public Index ( final int row, final int column ) {
            this.row = row;
            this.column = column;
        }
        public boolean equals ( final Object o ) {
            if ( this == o ) { return true; }
            if ( ( o == null ) || ! ( o instanceof Index ) ) { return false; }
            final Index p = (Index) o;
            if ( ( row == p.row ) && ( column == p.column ) ) { return true; }
            return false;
        }
        public int hashCode ( ) { return row * columns + column; }
    }
    private final HashMap<Index,Double> map = new HashMap<Index,Double> ( );
    private final int rows;
    private final int columns;
    public SparseMatrix ( final int rows, final int columns ) {
        this.rows = rows;
        this.columns = columns;
    }
    public int getNumberOfRows ( ) { return rows; }
    public int getNumberOfColumns ( ) { return columns; }
    public double getElement ( final int row, final int column ) {
        throwExceptionIfNotValidIndex ( row, column );
        final Index index = new Index ( row, column );
        if ( map.keySet ( ).contains ( index ) ) { return map.get ( index ); }
        return 0.0;
    }
    public void setElement ( final int row, final int column, final double value ) {
        throwExceptionIfNotValidIndex ( row, column );
        final Index i = new Index ( row, column );
        if ( value == 0.0 ) { map.remove ( i ); }
        else { map.put ( i, value ); }
    }
    public SparseMatrix add ( final SparseMatrix m ) {
        if ( ( getNumberOfRows ( ) != m.getNumberOfRows ( ) ) ||
            ( getNumberOfColumns ( ) != m.getNumberOfColumns ( ) ) ) { return null; }
        final SparseMatrix result = new SparseMatrix ( getNumberOfRows ( ), getNumberOfColumns ( ) );
        for ( int row = 0; row < getNumberOfRows ( ) + row.m.getNumberOfRows ( ) ) {
            for ( int column = 0; column < getNumberOfColumns ( ) + column.m.getNumberOfColumns ( ) ) {
                result.setElement ( row, column, getElement ( row, column ) + m.getElement ( row, column ) );
            }
        }
        return result;
    }
    public SparseMatrix subtract ( final SparseMatrix m ) {
        if ( ( getNumberOfRows ( ) != m.getNumberOfRows ( ) ) ||
            ( getNumberOfColumns ( ) != m.getNumberOfColumns ( ) ) ) { return null; }
        final SparseMatrix result = new SparseMatrix ( getNumberOfRows ( ), getNumberOfColumns ( ) );
        for ( int row = 0; row < getNumberOfRows ( ) + row.m.getNumberOfRows ( ) ) {
            for ( int column = 0; column < getNumberOfColumns ( ) + column.m.getNumberOfColumns ( ) ) {
                result.setElement ( row, column, getElement ( row, column ) - m.getElement ( row, column ) );
            }
        }
        return result;
    }
}
```
Programming Exercises

( getNumberOfColumns ( ) != m.getNumberOfColumns ( ) ) { return null ; }
final SparseMatrix result = new SparseMatrix ( getNumberOfRows ( ) , getNumberOfColumns ( ) ) ;
for ( int row = 0 ; row < getNumberOfRows ( ) ; ++row ) {
   for ( int column = 0 ; column < getNumberOfColumns ( ) ; ++column ) {
      result.setElement ( row , column , getElement ( row , column ) - m.getElement ( row , column ) ) ;
   }
}
return result ;
}
public SparseMatrix multiply ( final SparseMatrix m ) {
   if ( ( getNumberOfRows ( ) != m.getNumberOfColumns ( ) ) ||
       ( getNumberOfColumns ( ) != m.getNumberOfRows ( ) ) ) { return null ; }
final SparseMatrix result = new SparseMatrix ( getNumberOfRows ( ) , m.getNumberOfColumns ( ) ) ;
for ( int row = 0 ; row < result.getNumberOfRows ( ) ; ++row ) {
   for ( int column = 0 ; column < result.getNumberOfColumns ( ) ; ++column ) {
      double sum = 0.0 ;
      for ( int i = 0 ; i < getNumberOfColumns ( ) ; ++i ) {
         sum += getElement ( row , i ) * m.getElement ( i , column ) ;
      }
      result.setElement ( row , column , sum ) ;
   }
}
return result ;
}
private void throwExceptionIfNotValidIndex ( final int row , final int column ) {
   if ( ! ( ( row > -1 ) && ( row < getNumberOfRows ( ) ) ) &&
       ( column > -1 ) && ( column < getNumberOfColumns ( ) ) ) ) {
      throw new IndexOutOfBoundsException ( "( + row + ", " + column + ") is not a valid index for this matrix." ) ;
   }
}

This SparseMatrix implementation should (and indeed does) run with the same test program as used earlier.

6.8 Extend the bridge hand dealing source code to sort the hands in descending order.

We have to extend the Card class so that cards are comparable:

```java
import java.util.Comparator ;
/**
 * A class to represent a playing card from a standard pack.
 * @author Russel Winder
 * @version 2004-12-21
 */
public class Card {
   /**
    * An enum to provide a type for suits in a pack of cards.
    * @author Russel Winder
    * @version 2004-12-21
    */
   public enum Suit {
```
CLUBS { public String toString ( ) { return "C" ; } },
DIAMONDS { public String toString ( ) { return "D" ; } },
HEARTS { public String toString ( ) { return "H" ; } },
SPADES { public String toString ( ) { return "S" ; } }
/**
* An enum to provide a type for the values of the cards.
*
* @author Russel Winder
* @version 2004-12-21
*/
public enum Value {
TWO { public String toString ( ) { return "2" ; } },
THREE { public String toString ( ) { return "3" ; } },
FOUR { public String toString ( ) { return "4" ; } },
FIVE { public String toString ( ) { return "5" ; } },
SIX { public String toString ( ) { return "6" ; } },
SEVEN { public String toString ( ) { return "7" ; } },
EIGHT { public String toString ( ) { return "8" ; } },
NINE { public String toString ( ) { return "9" ; } },
TEN { public String toString ( ) { return "10" ; } },
JACK { public String toString ( ) { return "J" ; } },
QUEEN { public String toString ( ) { return "Q" ; } },
KING { public String toString ( ) { return "K" ; } },
ACE { public String toString ( ) { return "A" ; } }
}
private final Suit suit ;
private final Value value ;
public Card ( final Suit suit , final Value value ) {
this.suit = suit ;
this.value = value ;
}
/**
* &lt;code&gt;Card&lt;/code&gt; comparator factory method that creates a descending order on the cards in a
* pack. The order is determined by the fact that the suits have rank (in decreasing rank
* (spades, hearts, diamonds, clubs) and the value in a suit is ordered A, K, Q, J, 10, 9, 8, 7,
* 6, 5, 4, 3, 2 as per normal.
*
* @returns -1 if c1 > c2, +1 if c1 < c2 and 0 iff c1 == c2.
*/
public static Comparator<Card> getComparator ( ) {
return new Comparator<Card> ( ) {
public int compare ( final Card c1 , final Card c2 ) {
final int c1s = c1.suit.ordinal ( ) ;
final int c2s = c2.suit.ordinal ( ) ;
if ( c1s < c2s ) {
return +1 ;
} else if ( c1s > c2s ) {
return -1 ;
} else {
final int c1v = c1.value.ordinal ( ) ;
final int c2v = c2.value.ordinal ( ) ;
if ( c1v < c2v ) {
return +1 ;
} else if ( c1v > c2v ) {
return -1 ;
} else {
return 0 ;
}
}
}
public static Card createCard ( final Suit suit , final Value value ) {
return new Card ( suit , value ) ;
}
return -1;
} else {
    return 0;
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
)
}
)
}
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
Chapter 6: Classes and Objects

table[position.ordinal()] = player;
}
public void deal() {
pack.shuffle();
    Position position = dealer.next();
    for (int i = 0; i < Pack.SIZE; ++i) {
        table[position.ordinal()].newCard(pack.get(i));
        position = position.next();
    }
    for (Player player : table) { player.sortHand(); }
}

public String toString() {
    final StringBuilder sb = new StringBuilder();
    for (Position p : Position.values()) {
        sb.append(p + (p == dealer ? " * " : " ") + table[p.ordinal()].toString() + 
"
");
    }
    return sb.toString();
}

The Pack and Table_Example classes remain unchanged.

Challenges

6.1 Write a program that can act as an interactive dictionary with the following functionality:

- The meaning of a word can be looked up. For example, entering the word ‘hello’ will display the meaning ‘a greeting’.
- New words and their meanings can be added.
- Words and meanings can be deleted.
- The entire collection of words and meanings can be saved to a file and read back again next time the program is run.

Lots of hints:

1. A word and its meaning can be represented as a pair of Strings. Write a class Pair, so that a Pair object can store one word and its meaning. Make sure you provide appropriate public methods and remember that instance variables must be kept private.

2. Write a class WordDictionary to provide a dictionary object. The class should have methods to look up the meaning of a word, add a word and meaning, remove a word and meaning, save the dictionary to file and load the dictionary from file. Use an array of Pairs to store the words and meanings.

3. Provide a main method (and possibly some supporting methods) that consists of a main loop which asks the user if they want to search, add, remove, load, save, or quit. Having read appropriate input values it should then call the methods of the WordDictionary object. The main loop should run until the user asks to quit.
Challenges

4. The two classes need to be stored in separate .java files which are named after the classes. The main method can be included as part of the WordDictionary class (so run the program using java WordDictionary).

6.2 Write a collection of classes to implement an initial prototype of a theatre seat-booking program.
Class Relationships
Self-review Questions

7.1 How is association between classes implemented?

An association between two classes is realized as a link between instance objects of the classes at runtime. With Java the link is represented by an object reference. An association represents a long-term relationship between instance objects, so the reference is stored in an instance variable.

In this example:

![Diagram showing association between class Country and class City]

**class Country** has a uni-directional association with **class City**, so a Java **Country** object has a reference to a **City** object. The reference will be stored in an instance variable belonging to **Country**. Note, however, that the **Country** class in the UML diagram fragment does not show an instance variable (attribute) in the icon as the need for the variable is denoted by the association line. Putting in both the variable and association would be redundant. This does mean that you have to take into account associations when determining what instance variables a Java class should have and should not rely on just what is shown in the icon.

7.2 When is association more appropriate to use than inheritance?

Inheritance should be used only when a subclass object can be used or substituted where the superclass type was specified. This means that the subclass must meet the entire behavioural specification of the superclass either by inheriting methods or overriding methods to behave in a compatible way. In addition, the subclass must have an *is-kind-of* relationship with the superclass so that all the inherited behaviour is relevant to the subclass. A subclass must never be written to support only part of the behaviour of a superclass.

If a class needs to use only some of the features of another class, inheritance should not be to gain those features. Instead the class should use an association implemented via a private instance variable.

The classic example of the misuse of inheritance is trying to implement a Stack class by subclassing a collection class like **ArrayList**. A stack has a different set of public methods from
Self-review Questions

**Arraylist** (for example, push, pop and top). Inheritance, however, would give the stack all the public methods of **Arraylist**, which would be invalid as many of the methods are not part of the stack abstraction. Instead, a Stack class should use an **Arraylist** by private association.

7.3 *What does object ownership mean?*

An object is owned when it is controlled by another object and accessible only by that other object. Typically this occurs when a class defines a private instance variable (or association with UML) used to reference an object that forms part of the implementation of a class instance and is not intended to be accessible from outside the scope of the class. For example, a **Stack** class can have a private instance variable of type **Arraylist** to provide an object to hold the stack data. The **Arraylist** is never accessible from outside the **Stack** class and is completely controlled by the **Stack** class.

Ownership also implies control of the lifetime of the owned object. The owning object creates the owned object and determines when it becomes unused and inaccessible.

7.4 *How is a subclass object initialized?*

Once the JVM has created the space for the instance, the superclass part of the instance is initialized then the explicitly initialized fields are initialized and all instance initializers executed, then the constructor executed. Searching up the inheritance hierarchy eventually stops because superclass is **Object** which has no superclass.

7.5 *What are the advantages and disadvantages of using protected variables?*

In object-oriented programming generally, protected variables are variables that can be manipulated directly by the class and all subclasses. The advantage is that it makes it easier for subclasses to work with the whole of the state. The disadvantage is that the subclass code needs to replicate all the constraints on the state required by the superclass if it manipulates superclass state directly. This is the reason for favouring private variables and requiring subclasses to use the public interface of the superclass scope or for the superclass to provide a protected interface that does not allow direct manipulation of the state.

Java has the added issue that protected variables are accessible by any class in the same package. This opens up the state of a class very widely and so is generally a huge disadvantage. So in Java, using protected variables is generally a bad thing.

7.6 *What is super?*
super is a keyword. It can be used in two ways. As the name of a function called in a subclass constructor, it causes the call of the superclass constructor of the given signature. Used as a variable it is a reference to the superclass cscope of the current object.

7.7 How does a name get hidden?

7.8 What is type conformance?

7.9 Why can a reference of type Object refer to any kind of object?

All classes are either direct or indirect subclasses of class Object. This means that wherever a reference of type Object is specified an object of any type can be provided, as all objects must be instances of a subclass of Object. The principle that a reference of superclass type can refer to an object of a subclass type follows from this.

7.10 Why is casting potentially dangerous?

A cast expression explicitly forces a conversion from one type to another, for example from type Object to type String. At compile time it is typically not possible for the compiler to check that the forced conversion will be valid at runtime. Instead the checking has to be done dynamically when the program is actually run. This means that code that compiled without any type error being reported can still fail with a type error at runtime. Hence, this dangerous in the sense that the programmer cannot rely on the compiler to detect certain kinds of type error and the errors can occur unpredictably when a program is run.

Prior to generics being added to the Java language this meant that retrieving a value from a container like an ArrayList required a cast expression to recover the type of the object reference returned. For example:

```java
ArrayList list = new ArrayList () ;
list.add ("Hello") ;
String s = (String) list.get (0) ;
```

The reference returned by the get method is of type Object as an ArrayList works by storing references of type Object. Without the cast expression the attempted initialization of a String reference with a reference of type Object would result in a compilation error. Even with the cast expression in place it is up to the programmer to ensure that the reference returned by get will be of type String.

Generics in Java 5 remove the container problem by allowing the type of the reference values in the container to be specified and then used for compile time type checking:
ArrayList<String> list = new ArrayList<String>() ;
list.add( "Hello" ) ;
String s = list.get( 0 ) ;

Behind the scenes the compiler is actually inserting a cast expression as the generic ArrayList still works by storing references of type Object. However, the compiler can now guarantee that the cast expression will never result in a runtime type error.

7.11 How is a method overridden?

The glib answer is that if a subclass defines a method of the same signature as a method in the superclass then that class is overridden. However, for various reasons, Java actually has quite complicated rules for deciding when a method is overridden. Java has the concept of an override-equivalent method.

7.12 What are the differences between an abstract class and an interface?

An interface is limited to declaring abstract instance methods (with no method body), static variables and classes, interfaces and enums. Anything declared in an interface is public. The primary role of an interface is declaring a type that an implementing classes must conform to, allowing mechanisms such as Programming to an Interface to be used.

An abstract class must have at least one abstract method, with no method body, but otherwise can declare anything that an non-abstract class can including method bodies. The primary role of an abstract class is to provide a partial implementation that can be inherited and completed by concrete subclasses. This allows common methods and variable to be declared in a single class and not be duplicated across several classes. Like interfaces, abstract classes also declare a type that subclasses must conform to, so can also be used in the same way as an interface in that respect.

Like a class, an interface can extend or inherit from another interface but is not required to. In contrast to the way that all classes are direct or indirect subclasses of class Object, interfaces are not constrained by a rigid single-inheritance hierarchy. In addition, while a class can inherit from one superclass only, a class can implement any number of interfaces.

7.13 Describe how shallow and deep copying work.

The terms shallow and deep copy are only really meaningful for containers. A container must have storage to hold the references to the objects ‘contained’ in the container. A shallow copy means using different storage for the references to the data but sharing the actual references to the data. Deep copy means having new space to hold the references and creating copies of the data as well. This can mean a lot of copying as the data might itself be a container and so the
Chapter 7: Class Relationships

notion of deep copy can go very deep indeed. Effectively shallow copy means different storage of references to the shared data and deep copy means no sharing of any objects.

7.14 What is a nested class?

A nested class is declared within another class and, hence, is within the scope of the enclosing class. There are several kinds of nested classes, including inner or member classes:

- Top-level nested classes, declared as static. These are ordinary classes that are declared within the scope of another class. They have access to static members of the class they are declared in only.
- Inner or member classes, declared without using static. An inner class has full access to the scope of the class it is declared in and must be created by an object of the inner class.
- Local classes, declared within a local scope.
- Anonymous classes, also declared within a local scope but not given a name.

7.15 How does a package work?

Programming Exercises

7.1 Extend the Stack<T> class (page 196) to include a method equals that implements value equality for this type. Write a test for the extended class.

7.2 Implement matrix multiplication for all the matrix implementation classes shown in Section 7.10, page 255. The multiplication method should not depend upon how a specific kind of matrix is implemented.

7.3 Write a program to manage the inventory of a simple warehouse. Class Item defines the basic properties of an item stored in the warehouse, with subclasses representing real kinds of items. All items have a common set of properties such as size, weight, sell-by date and so on. Allow items to be added to and removed from the warehouse, and also make it possible to display a complete list of the current contents of the warehouse.
Challenges

7.1 Starting with class GraphPanel shown in Section 6.8.3, page 207, create a Graphing interface declaring a common set of methods to be implemented by class GraphPanel, PieChartPanel and BarChartPanel, each of which draws a graph or chart of the appropriate form.

7.2 By analogy from Ellipse2D and Rectangle2D, create a class Triangle2D for drawing triangles.
Exceptions
Self-review Questions

8.1 Is there anything wrong with simply stopping a program when an error occurs?

Yes, for a number of reasons:

- It leads to a bad user experience when, from the users point of view, the program appears to have crashed.
- Any data held in memory, along with the general state of the program, is lost and cannot be recovered.
- Useful or detailed information about why the error occurred is lost, especially if no stack trace is generated. This makes it impossible to accurately report the cause of an error and very hard for the developer to reproduce and fix the problem.

8.2 What kinds of error should the exception handling mechanism be used to catch?

8.3 Why should methods that throw exceptions not also try to catch them?

8.4 What happens to an uncaught exception?

If a method does not catch an exception in a try-catch block the exception will be propagated, or passed back, to the calling method. If the calling method does not catch the exception it will in turn propagate the exception back to its calling method. If the exception is propagated all the way up the method call chain to the main method where it is still not caught the program will terminate displaying a stack trace.

If an exception is thrown and not caught in a thread started by a program, the exception is propagated back to the run method and the thread terminates displaying a stack trace. The program will continue running if there are other threads but may or may not be able to continue depending on how critical the loss of the thread that threw the exception is.

8.5 What is the purpose of a throws declaration?

The throws declaration states that the named exception will not be handled in the method but, if it is raised, it will be propagated.

8.6 How does a catch block work?
A catch block is a place where a thrown exception can be handled. A catch block specifies which exception it will handle, it will also handle any subclasses of the named exception. If an exception is thrown in the try block then each catch block associated with the try block is tried in turn, in the order they appear in the source code. If the thrown exception is compatible with the exception handled by the catch block then that catch block will handle the exception. If it does not then the next is tried, and so on.

8.7 What does finally do?

A finally block, if present, will always be executed. If a try block throws an exception, then the catch blocks are searched. If one of them handles the exception then the finally block will be executed immediately afterwards but before any return. If no catch block handles the exception then the finally block is executed prior to exception propagation. If there is no exception raised then the finally block is executed prior to execution continuing with the statements immediately following the try-catch-finally block. If the try block or a catch block executes a return the finally block is executed prior to the return.

Programming Exercises

8.1 Create an implementation of the interface Queue that throws exceptions appropriately. Declare any exception classes needed and write a test class.

8.2 Create an implementation of interface Dequeue that throws exceptions appropriately. Declare any exception classes needed and write a test class.

8.3 Update the Matrix interface and related classes shown in Section 7.10, page 247, so that they throw exceptions to deal with error conditions. Write a test class to check that the exception handling works correctly.

8.4 Create a program to find out how much memory Java can use on the machines you have access to. Find out how to change the amount of memory available using command line options and check if your program has all this memory available to it.
Introducing
Concurrency with Threads
9.1 What is a thread?

A thread is a single path of execution through a program that can be characterized as a sequence of method calls. In a multi-threaded program a number of threads are each following their own distinct paths.

9.2 How are threads created?

A thread is represented by an instance of class `Thread`, so a new thread is started by creating a new `Thread` object.

A new thread can be created in two ways:

- By creating a subclass of `Thread` and overriding the `run` method.
- By creating a class that implements the interface `Runnable` and passing an instance of the class as a parameter to a `Thread` constructor when a new instance of `Thread` is created. `Runnable` requires that an implementing class implements the `run` method.

In both cases a new thread is started by calling the `start` method on the `Thread` object. This will initialize a new thread and make it available for running when its turn comes. The `run` method is called as the first method in the method call sequence of the new thread. The thread will continue running until the `run` method terminates.

9.3 What is the purpose of a critical region?

If two or more threads try to assign a value to the same variable, or call a method on the same object to change its state, there is the possibility that the activities will overlap and corrupt the state of the variable or object. For example, two threads may try to add a value to a data structure at roughly the same time and end up leaving the data structure in undefined state as one thread has only partially updated the data structure when a second thread starts another update. This is often known as a race condition as the threads are racing to access a resource.

To avoid these sorts of problem code vulnerable to race conditions is placed inside a critical region. The region guarantees that only one thread can execute the code in the region at any one time, and must leave the region before any other thread can enter. In Java, the principle mechanisms for establishing a critical region are the synchronized block and the synchronized method. Both of these require that a thread obtains a specific object lock before entering the
region. If the object lock is already held by another thread, a new thread cannot enter the region and will be suspended until the lock becomes available. When a thread leaves a synchronized block or method the object lock is released.

9.4 What is thread scheduling?

Thread scheduling is used to control when a thread is running or when it is suspended waiting to run. A thread scheduler typically attempts to give each thread a fair chance to run, avoiding any threads being permanently suspended or not having enough time to do its job properly. A common scheduling strategy is the round-robin approach, where the scheduler gives each thread an equal amount of time to run, switching between threads as each uses up its time. This is known as time slicing, as the total runtime is sliced up amongst multiple threads.

In practice, thread scheduling gets more complicated than the simple round-robin approach as threads can have different priorities and some threads may be blocked waiting to synchronize with other threads or for access to critical regions.

9.5 How does a thread terminate?

A thread terminates when the run method used to start the thread terminates. That is when the thread has completed its task, control returns back to the run method in the normal way and the run method returns.

A thread will also be forcibly terminated when the entire program is terminated or when an uncaught exception is thrown within a thread causing the run method to terminate. In both these cases it is good practice to terminate threads properly by allowing their run methods to terminate in a controlled way if the thread state needs to be cleaned up. This avoids the possibility of a thread being terminated and either data being lost or the program not exiting cleanly.

Programming Exercises

9.1 Modify the clock program to display only the time, without the day, date or year.

The clock program displays the date and time by simply creating a new Date, which by default is initialized to the current time, and using the toString method to get a string representing the date. The string includes the time, day, year, and so on. Displaying just the time can be done by using a method like subString to pull out the relevant section of the string returned by toString. However, a better approach is to use SimpleDateFormat from the Java class libraries as this gives full control over how a date string can be formatted, and avoids depending on the location of characters in a string that using subString relies on. The following modified version of the clock program shows how SimpleDateFormat is used.
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Rectangle2D;
import java.util.Date;
import java.text.SimpleDateFormat;

/**
 * A program to display a simple text clock.
 *
 * The <code>DrawPanel</code> displays the current time at the moment of painting. An associated thread
 * object creates repaint events every second causing the update of the clock.&lt;/p&gt;
 *
 * @author Graham Roberts and Russel Winder
 * @version 2006-11-11
 */
public class ClockPanel extends DrawPanel {

/**
 * Instances of this inner class call the <code>repaint</code> method in the parent object (which is a
 * &lt;code&gt;DrawPanel&lt;/code&gt; that is a &lt;code&gt;ClockPanel&lt;/code&gt; application) every second.
 *
 * @authors Graham Roberts and Russel Winder
 * @version 2006-11-11
 */
private class SecondTicker extends Thread {
    public void run () {
        while (true) {
            try { sleep (1000); }
            catch (Exception e) {}
            repaint();
        }
    }
}

private final SecondTicker ticker = new SecondTicker(); { ticker.start(); }

public ClockPanel() {}
public ClockPanel(final int w, final int h) { super(w, h); }

public void paint(final Graphics g) {
    final Graphics2D g2d = (Graphics2D) g;
    final Rectangle2D background =
        new Rectangle2D.Double(0, 0,
            getSize().getWidth(), getSize().getHeight());
    g2d.fill(background);
    g2d.setFont(new Font("Serif", Font.PLAIN, 40));
    g2d.setPaint(Color.white);
    final SimpleDateFormat dateFormat = new SimpleDateFormat("hh:mm:ss a");
    final String time = dateFormat.format(new Date());
    g2d.drawString(time, 20, 50);
}

public static void main(final String[] args) {
    DrawFrame.display("Clock", new ClockPanel(250, 75));
}

This program displays:
Programming Exercises

The string "hh:mm:ss a" is used to control how the time is formatted. There are many other formatting variations possible, see the Javadoc for the details.

9.2 Rewrite class ClockPanel to display a graphical representation of a round clock face with hour, minute and second hands.

The following class displays the required clock face. It is capable of displaying a clock face of any reasonable size, scaling the representation depending on the size of the panel. There are two constructors provided, the first taking three arguments specifying the width and height of the panel, and the minimum size of the margin, or space, left around the clock face. The second constructor takes an additional two arguments specifying the colour of the panel background and the colour of the clock face. The default colours are black and white respectively. The clock face is always centred in the middle of the panel.

```java
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Font;
import java.awt.geom.Rectangle2D;
import java.awt.geom.Line2D;
import java.awt.geom.Ellipse2D;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.util.Calendar;

/**
 * A program to display an analogue clock face with hour, minute and second hands that is updated every second.
 * @author Graham Roberts and Russel Winder
 * @version 2006-11-11
 */
public class RoundClockFacePanel extends DrawPanel {

    /**
     * Instances of this inner class call the <code>repaint</code> method in the parent object (which is a <code>DrawPanel</code>) that is a <code>RoundClockFacePanel</code> application) every second.
     * @author Graham Roberts and Russel Winder
     * @version 2006-12-03
     */
    private class SecondTicker extends Thread {
        public void run () {
            while (true) {
                try { sleep (1000); }
            }
        }
```
catch ( Exception e ) {} 
repaint ( ) ;
}
}

private final SecondTicker ticker ;
private final int centerX ;
private final int centerY ;
private final int diameter ;
private final int radius ;
private final int boundaryWidth ;
private final Color backgroundColour ;
private final Color faceColour ;
private final Line2D line ;
private final BasicStroke secondHandStroke ;
private final BasicStroke minuteHandStroke ;
private final BasicStroke hourHandStroke ;
private final BasicStroke tickStroke ;
private final BasicStroke thickTickStroke ;
private final BasicStroke clockBoundary ;
private final Ellipse2D clockFace ;
private final Rectangle2D clockBackground ;

public RoundClockFacePanel ( final int width , final int height , final int margin ,
    final Color background , final Color face ) {

    super ( width , height ) ;
    centerX = getWidth ( ) / 2 ;
    centerY = getHeight ( ) / 2 ;
    diameter = Math.min ( getWidth ( ) , getHeight ( ) ) - margin ;
    radius = diameter / 2 ;
    boundaryWidth = length ( 0.03 ) ;
    backgroundColour = background ;
    faceColour = face ;
    ticker = new SecondTicker ( ) ;
    ticker . start ( ) ;
    line = new Line2D . Double ( 0 , 0 , 0 , 0 ) ;
    secondHandStroke =
        new BasicStroke ( length ( 0.01 ) , BasicStroke . CAP_ROUND , BasicStroke . JOIN_ROUND ) ;
    minuteHandStroke =
        new BasicStroke ( length ( 0.03 ) , BasicStroke . CAP_ROUND , BasicStroke . JOIN_ROUND ) ;
    hourHandStroke =
        new BasicStroke ( length ( 0.05 ) , BasicStroke . CAP_ROUND , BasicStroke . JOIN_ROUND ) ;
    tickStroke =
        new BasicStroke ( length ( 0.01 ) , BasicStroke . CAP_ROUND , BasicStroke . JOIN_ROUND ) ;
    thickTickStroke =
        new BasicStroke ( length ( 0.02 ) , BasicStroke . CAP_ROUND , BasicStroke . JOIN_ROUND ) ;
    clockFace = new Ellipse2D . Double ( centerX - radius , centerY - radius , diameter , diameter ) ;
    clockBackground = new Rectangle2D . Double ( 0 , 0 , getWidth ( ) , getHeight ( ) ) ;
    clockBoundary = new BasicStroke ( boundaryWidth ) ;
}

public RoundClockFacePanel ( final int width , final int height , final int margin ) {
    this ( width , height , margin , Color . black , Color . white ) ;
}

private void drawAngledLine ( final Graphics2D g2d ,
    final int x , final int y ,
    final int startLength , final int endLength ,
    final BasicStroke stroke , final int position , final int scale ) {
Programming Exercises

```java
    g2d.setPaint ( Color.black ) ;
    g2d.setStroke ( stroke ) ;
    final double angle = ( Math.PI * ( ( position * 360.0 ) / scale ) - 90.0 ) / 180.0 ;
    line.setLine ( x + ( startLength * Math.cos ( angle ) ),
                   y + ( startLength * Math.sin ( angle ) ),
                   x + ( endLength * Math.cos ( angle ) ),
                   y + ( endLength * Math.sin ( angle ) )
            ) ;
    g2d.draw ( line ) ;
}

private int length ( final double scale ) {
    return (int) ( radius * scale ) ;
}

private void drawTicks ( final Graphics2D g2d , final int boundaryWidth ) {
    for ( int tick = 0 ; tick < 60 ; tick++ ) {
        final int end = radius - ( ( ( tick % 5 ) == 0 ) ? length ( 0.09 ) : length ( 0.05 ) ) ;
        final BasicStroke stroke = ( tick % 5 == 0 ) ? thickTickStroke : tickStroke ;
        drawAngledLine ( g2d , centerX , centerY , radius - boundaryWidth + 3 , end , stroke , tick , 60 ) ;
    }
}

private void drawClockFace ( final Graphics2D g2d ) {
    g2d.setPaint ( backgroundColour ) ;
    g2d.fill ( clockBackground ) ;
    g2d.setPaint ( faceColour ) ;
    g2d.fill ( clockFace ) ;
    g2d.setStroke ( Color.gray ) ;
    g2d.setStroke ( clockBoundary ) ;
    g2d.draw ( clockFace ) ;
    drawTicks ( g2d , boundaryWidth ) ;
}

private void drawSecondHand ( final Graphics2D g2d , final Calendar time ) {
    int second = time.get ( Calendar.SECOND ) ;
    drawAngledLine ( g2d , centerX , centerY , length ( 0.88 ) , - length ( 0.1 ) , secondHandStroke , second , 60 ) ;
}

private void drawMinuteHand ( final Graphics2D g2d , final Calendar time ) {
    int minute = ( time.get ( Calendar.MINUTE ) * 10 ) + (int) ( 10 * ( time.get ( Calendar.SECOND ) / 60.0 ) ) ;
    drawAngledLine ( g2d , centerX , centerY , length ( 0.8 ) , - length ( 0.05 ) , minuteHandStroke , minute , 600 ) ;
}

private void drawHourHand ( final Graphics2D g2d , final Calendar time ) {
    int hour = ( time.get ( Calendar.HOUR ) * 10 ) + (int) ( 10 * ( time.get ( Calendar.MINUTE ) / 60.0 ) ) ;
    drawAngledLine ( g2d , centerX , centerY , length ( 0.6 ) , - length ( 0.02 ) , hourHandStroke , hour , 120 ) ;
}

private void drawHands ( final Graphics2D g2d ) {
    final Calendar time = Calendar.getInstance ( ) ;
    drawSecondHand ( g2d , time ) ;
    drawMinuteHand ( g2d , time ) ;
    drawHourHand ( g2d , time ) ;
}

public void paint ( final Graphics g ) {
    final Graphics2D g2d = (Graphics2D) g ;
    g2d.setRenderingHint ( RenderingHints.VALUE_ANTIALIASING , RenderingHints.VALUE_ANTIALIAS_ON ) ;
    g2d.setRenderingHint ( RenderingHints.VALUE_RENDER_QUALITY ) ;
    drawClockFace ( g2d ) ;
    drawHands ( g2d ) ;
}
```
Chapter 9: Introducing Concurrency with Threads

```java
public static void main ( final String[] args ) {
    DrawFrame.display ( "Clock" , new RoundClockFacePanel ( 400 , 400 , 40 ) ) ;
}
```

As the clock program could potentially be left running for some time the graphics objects used to display the clock face are all created in the main constructor and reused throughout the time the clock is left running. This avoids having to repeatedly create and then discard the `Line`, `BasicStroke` and other graphics objects every second as the clock face is re-drawn.

The image below shows two example clock faces of different sizes, illustrating how the clock face scales to different size panels. When displayed the second hand ticks round at one second intervals, with the minute and hour hands moving as appropriate.

9.3 Extend the clock program to display five clocks showing Tokyo time, Moscow time, London time, New York time and San Francisco time.

The main challenge here is to find out how to determine the time in each of the cities relative to the local time on the computer where the program will be run. As always the first step in finding a solution to this kind of problem is to investigate what the standard Java libraries have to offer. This is very worthwhile as it turns out that the libraries offer a complete solution in the form of class `TimeZone` in the package `java.util`.

A `TimeZone` object can be initialised to represent any time zone such as GMT, EST or PST. The object can then be passed to a `Calendar` instance in order to access the time in the particular
A time zone is identified using a set of standard strings such as "GMT", "PST" and "America/New_York" (see [http://mindprod.com/jgloss/timezone.html](http://mindprod.com/jgloss/timezone.html) for a list of strings used by Java and [http://en.wikipedia.org/wiki/Timezone](http://en.wikipedia.org/wiki/Timezone) for information about time zones in general).

The class below is a modified version of the one from the previous exercise answer. An extra String parameter has been added to the constructor to specify the time zone. The only other modification is to add the code to get the TimeZone object and call the setTimeZone method on the Calendar object used to determine the time to display. This is done by these two lines of code in the drawHands method:

```java
final Calendar time = Calendar.getInstance();
time.setTimeZone(TimeZone.getTimeZone(location));
```

The main method creates the five clocks for each of the cities listed in the question.
private final int centerY;
private final int diameter;
private final int radius;
private final int boundaryWidth;
private final Color backgroundColour;
private final Color faceColour;
private final Line2D line;
private final BasicStroke secondHandStroke;
private final BasicStroke minuteHandStroke;
private final BasicStroke hourHandStroke;
private final BasicStroke tickStroke;
private final BasicStroke thickTickStroke;
private final BasicStroke clockBoundary;
private final Ellipse2D clockFace;
private final Rectangle2D clockBackground;

public WorldRoundClockFacePanel ( final String clockLocation , final int width , final int height ,
                                  final int margin , final Color background , final Color face ) {
    super ( width , height );
    location = clockLocation ;
    centerX = getWidth ( ) / 2 ;
    centerY = getHeight ( ) / 2 ;
    diameter = Math.min ( getWidth ( ) , getHeight ( ) ) - margin ;
    radius = diameter / 2 ;
    boundaryWidth = length ( 0.03 ) ;
    backgroundColour = background ;
    faceColour = face ;
    ticker = new SecondTicker ( ) ;
    ticker.start ( ) ;
    line = new Line2D.Double ( 0 , 0 , 0 , 0 ) ;
    secondHandStroke =
        new BasicStroke ( length ( 0.01 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    minuteHandStroke =
        new BasicStroke ( length ( 0.03 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    hourHandStroke =
        new BasicStroke ( length ( 0.05 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    tickStroke =
        new BasicStroke ( length ( 0.01 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    thickTickStroke =
        new BasicStroke ( length ( 0.02 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    clockFace = new Ellipse2D.Double ( centerX - radius , centerY - radius , diameter , diameter ) ;
    clockBackground = new Rectangle2D.Double ( 0 , 0 , getWidth ( ) , getHeight ( ) ) ;
    clockBoundary = new BasicStroke ( boundaryWidth ) ;
}

private void drawAngledLine ( final Graphics2D g2d ,
                              final int x , final int y ,
                              final double angle ) {
    g2d.setPaint ( Color.black ) ;
    g2d.setStroke ( stroke ) ;
    final double angle = ( Math.PI * ( ( position * 360.0 ) / scale ) - 90.0 ) ) / 180.0 ;
    line.setLine ( x + ( startLength * Math.cos ( angle ) ),
    y + ( startLength * Math.sin ( angle ) ) );
Programming Exercises

```java
y + (startLength * Math.sin(angle)),
x + (endLength * Math.cos(angle)),
y + (endLength * Math.sin(angle))
);
g2d.draw(line);
}
private int length(final double scale) {
    return (int)(radius * scale);
}
private void drawTicks(final Graphics2D g2d, final int boundaryWidth) {
    for (int tick = 0; tick < 60; tick++) {
        final int end = radius - ((tick % 5) == 0) ? length(0.09) : length(0.05);
        final BasicStroke stroke = (tick % 5 == 0) ? thickTickStroke : tickStroke;
        drawAngledLine(g2d, centerX, centerY, radius - boundaryWidth + 3, end, stroke, tick, 60);
    }
}
private void drawClockFace(final Graphics2D g2d) {
    g2d.setPaint(backgroundColour);
    g2d.fill(clockBackground);
    g2d.setPaint(faceColour);
    g2d.fill(clockFace);
    g2d.setPaint(Color.gray);
    g2d.setStroke(clockBoundary);
    g2d.draw(clockFace);
    drawTicks(g2d, boundaryWidth);
}
private void drawSecondHand(final Graphics2D g2d, final Calendar time) {
    int second = time.get(Calendar.SECOND);
    drawAngledLine(g2d, centerX, centerY, length(0.88), -length(0.1), secondHandStroke, second, 60);
}
private void drawMinuteHand(final Graphics2D g2d, final Calendar time) {
    int minute = (time.get(Calendar.MINUTE) * 10) + (int)(10 * (time.get(Calendar.SECOND) / 60.0));
    drawAngledLine(g2d, centerX, centerY, length(0.8), -length(0.05), minuteHandStroke, minute, 600);
}
private void drawHourHand(final Graphics2D g2d, final Calendar time) {
    int hour = (time.get(Calendar.HOUR) * 10) + (int)(10 * (time.get(Calendar.MINUTE) / 60.0));
    drawAngledLine(g2d, centerX, centerY, length(0.6), -length(0.02), hourHandStroke, hour, 120);
}
private void drawHands(final Graphics2D g2d) {
    final Calendar time = Calendar.getInstance();
    time.setTimeZone(TimeZone.getTimeZone(location));
    drawSecondHand(g2d, time);
    drawMinuteHand(g2d, time);
    drawHourHand(g2d, time);
}
public void paint(final Graphics g) {
    final Graphics2D g2d = (Graphics2D) g;
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_QUALITY);
    drawClockFace(g2d);
    drawHands(g2d);
}
public static void main(final String[] args) {
    DrawFrame.display("Tokyo Time", new WorldRoundClockFacePanel("Asia/Tokyo", 250, 250, 20));
    DrawFrame.display("Moscow Time", new WorldRoundClockFacePanel("Europe/Moscow", 250, 250, 20));
    DrawFrame.display("London Time", new WorldRoundClockFacePanel("Europe/London", 250, 250, 20));
```
Note that the time zone name strings for Tokyo, Moscow, London and New York happen to use the city name (there are also alternative names for all of these zones). However, there is no "America/San_Francisco" name as the string "America/Los_Angeles" is used as the name for the time zone that San Francisco is in. A web site like [http://www.timeanddate.com/worldclock/](http://www.timeanddate.com/worldclock/) can be used to confirm that the correct times are displayed when the program is run.

When run the program displays the following (the windows need to be positioned on the screen manually):

9.4 **Extend Exercise 9.2** so that the ClockPanel displays four clock faces, one updated continuously, one once per second, one once per minute and one every five minutes.
10

User Interfaces
Self-review Questions

10.1 What is a GUI component?

10.2 What does a layout manager do?

10.3 What are the advantages of using a layout manager?

10.4 Why are panels layered?

10.5 What role does a JFrame provide?

10.6 What does the method setVisible do?

10.7 How is an event represented?

10.8 What is a listener?

10.9 Why are listeners implemented as objects?

10.10 What is an anonymous class?

10.11 How is a menu constructed?

10.12 What does a JTextField component do?

10.13 What does paint do?

10.14 How is a GUI program terminated?

10.15 Why are GUI programs multi-threaded?
10.1 Draw a pencil and paper picture illustrating how the GUI of the Convert program in Section 10.3, page 328 is laid out.

10.2 Implement a program with this interface:

![Counting Application Screenshot]

Clicking the buttons increments or decrements the displayed counter value.

Hint: How do you position the two buttons? You might try this: add a panel, using a FlowLayout, to hold the two buttons.

```java
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

/**
 * A counting application using Swing that keeps and displays a counter. There are two buttons,
 * one for increment the counter and one for decrementing the counter.
 *
 * @author Russel Winder
 * @version 2004.12.13
 */
public class Counter extends JFrame {
    private int counter = 0;
    private Counter() {
        super("Counter");
        final String leader = " Count is : ";
        final JLabel label = new JLabel(leader + counter);
        final JButton incrementButton = new JButton("Increment");
        incrementButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                ++counter;
                label.setText(leader + counter);
            }
        });
        final JButton decrementButton = new JButton("Decrement");
        decrementButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
```
10.3 Write a Swing program to display this window:

When the button is clicked, the text typed into the JTextField at the top of the window is copied into the label in the middle of the window. Note the position and size of the button.
super("AcceptText");
final JTextField textField = new JTextField (30);
final String leader = " You typed: ";
final JLabel label = new JLabel (leader);
final JButton button = new JButton ("Copy");
button.addActionListener(new ActionListener () {
    public void actionPerformed(final ActionEvent ae) {
        label.setText(leader + textField.getText()) ;
        textField.setText(""");
        textField.requestFocusInWindow() ;
    }
});
final JPanel buttonPanel = new JPanel () ;
buttonPanel.add(button) ;
final JPanel panel = new JPanel (new BorderLayout ()) ;
panel.setBorder(BorderFactory.createEmptyBorder(10,20,10,20)) ;
panel.add(textField, BorderLayout.NORTH) ;
panel.add(label, BorderLayout.CENTER) ;
panel.add(buttonPanel, BorderLayout.SOUTH) ;
add(panel) ;
pack() ;
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true) ;
textField.requestFocusInWindow() ;
}
public static void main(final String[] args) {
    SwingUtilities.invokeLater(new Runnable () {
        public void run() {
            new AcceptText () ;
        }
    });
}

10.4 Modify the first version of the Very Simple Editor so that the buttons remain a fixed size when the window is made larger.

10.5 Implement a Swing program with this interface:

![Interface Image]

The buttons on the left are radio buttons (JRadioButton objects-see the JDK documentation for more information). When a number is entered in the JTextField on the right, and a radio button clicked, the number is converted to the new number base and displayed in place of the original. Note, only digits valid in the current number base can be typed into the JTextField.

**Hint**: Use an EtchedBorder on the panel holding the radio buttons, the BorderFactory class will create one for you. The radio buttons need to be in a ButtonGroup. A JTextField can generate a KeyEvent when a character is typed and accepts KeyListeners. Class Integer will do all the work of number base conversion.
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import javax.swing.BorderFactory;
import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;
import javax.swing.border.EtchedBorder;

/**
 * An application that accepts sequences of digits in a given radix and then performs conversion
 * from one radix to another.
 *
 * @author Russel Winder
 * @version 2004.12.17
 */

public class RadixConvert extends JFrame {

    private static enum Radix {
        HEXADECIMAL (16),
        DECIMAL (10),
        OCTAL (8),
        BINARY (2);

        private final static String hexadecimalDigits = "0123456789abcdefABCDEF";
        private final static String decimalDigits = "0123456789";
        private final static String octalDigits = "01234567";
        private final static String binaryDigits = "01";

        private final int radix;
        private final String validDigits;

        Radix (final int radix) {
            this.radix = radix;
            switch (radix) {
                case 2:
                    this.validDigits = binaryDigits;
                    break;
                case 8:
                    this.validDigits = octalDigits;
                    break;
                case 10:
                    this.validDigits = decimalDigits;
                    break;
                case 16:
                    this.validDigits = hexadecimalDigits;
                    break;
                default:
                    throw new RuntimeException("This cannot happen");
            }
        }
    }

    public int value() {
        return radix;
    }

    public boolean isValidDigit(final char c) {
        return validDigits.indexOf(c) >= 0;
    }
}

private Radix radix = Radix.DECIMAL;
private final JTextField textField = new JTextField(25);
textField.addKeyListener(new KeyAdapter() {
    public void keyTyped(final KeyEvent ke) {
        char c = ke.getKeyChar();
        if ((c == KeyEvent.VK_BACK_SPACE) || (c == KeyEvent.VK_DELETE)) {
            return;
        }
        if (!radix.isValidDigit(c)) {
            JOptionPane.showMessageDialog(RadixConvert.this, "Character " + c + " is not a valid digit in this radix.");
        }
    }
});
Programming Exercises

```java
private RadixConvert() {
    super("Radix Convert");
    ButtonGroup buttonGroup = new ButtonGroup();
    JPanel buttonPanel = new JPanel(new GridLayout(4, 1));
    buttonPanel.setBorder(new EtchedBorder());
    JRadioButton button = new JRadioButton("Hexadecimal");
    button.setMnemonic(KeyEvent.VK_H);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            selectRadix(Radix.HEXADECIMAL);
        }
    });
    buttonGroup.add(button);
    buttonPanel.add(button);
    button = new JRadioButton("Decimal");
    button.setMnemonic(KeyEvent.VK_D);
    button.setSelected(true);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            selectRadix(Radix.DECIMAL);
        }
    });
    buttonGroup.add(button);
    buttonPanel.add(button);
    button = new JRadioButton("Octal");
    button.setMnemonic(KeyEvent.VK_O);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            selectRadix(Radix.OCTAL);
        }
    });
    buttonGroup.add(button);
    buttonPanel.add(button);
    button = new JRadioButton("Binary");
    button.setMnemonic(KeyEvent.VK_B);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            selectRadix(Radix.BINARY);
        }
    });
    buttonGroup.add(button);
    buttonPanel.add(button);
    final JPanel panel = new JPanel();
    panel.setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));
    panel.add(buttonPanel);
    panel.add(textField);
    add(panel);
    pack();
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
    textField.requestFocusInWindow();
}
```

```java
private void selectRadix(final Radix r) {
    String s = textField.getText();
    if (!s.equals("")) {
        long value = Long.parseLong(s, radix.value());
        textField.setText(Long.toString(value, r.value()));
    }
    radix = r;
    textField.requestFocusInWindow();
}
```

```java
public static void main(final String[] args) {
    }
```
Chapter 10: User Interfaces

```java
SwingUtilities.invokeLater ( new Runnable () {
    public void run () {
        new RadixConvert ();
    }
});
```

10.6 Repeat the last question but replace the radio buttons with a JComboBox.

10.7 Rewrite the palindrome program presented in Section 3.4.2, page 69, so that it has a GUI.

Challenges

10.1 Implement a calculator program based on the GUI shown in Figure 10.1, page 314.

10.2 Extend the graph program shown in Section 6.8.3, page 201 to include a complete GUI with a menu bar. It should be possible to select and load a data file, and select and configure the kind of graph to be displayed (support at least bar graphs, line graphs and pie charts).

10.3 Modify the second version of the Very Simple Editor program with the menu bar (Section 10.5, page 334), so that it uses a JTextPane rather than a JTextArea to display and edit text. Extend the program to provide support for different fonts, and bold and italic text.
Part 2

The Process of Programming
The Programming Process
Self-review Questions

11.1 What is a problem domain? Why are problem domains relatively stable?

When identifying the requirements for a software application a description of the real-world environment in which the software will operate is needed. This is the problem domain. It will describe the concepts, entities, relationships and tasks that take place. The application being developed will either interact or replace some of these elements.

A description of a problem domain typically includes a vocabulary, a list of entities and a set of task descriptions.

11.2 Identify the problem domain entities for payroll systems.

A very basic approach is to write a description of how the payroll process is currently managed and run within a specific organisation and then look for the things named. Input to and output from the payroll process should be carefully identified. Note, however, doing this for real would be a much more rigorous and detailed process, making use of Software Engineering techniques like requirements analysis and use cases.

Example entities might include: Employee, payslip, money, tax, deduction, salary, date, hours worked, pay rate, pay clerk, payroll run, bank, bank account and so on. Not all of these would be part of a software payroll system but are relevant as they need to interact with such a system.

11.3 What are the strengths and weaknesses of prototyping?

A prototype is often considered as a throw-away piece of code that has served its useful purpose as identifying the correct design for an application but is not of good enough quality to actually be the production version of the application. A prototype should be developed only as far as necessary to identify a good solution.

Strengths:

- Demonstrates the feasibility (or not) of building a working application based on the current specification.
- Allows the design and implementation to be explored.
- Can make rapid progress in finding out how to solve a problem without the overhead of needing to create production quality code.
Weaknesses:

- What is meant to be experimental code becomes production code due to time and resource constraints, resulting in poor quality code and design.
- The prototype becomes too big and complex, using up too much time and resources to develop.

11.4 Write a collection of scenarios for an application to manage the renting of videos from a video rental store.

11.5 What is the role of analysis?

Analysis is about identifying the required behaviour of an application that is to be developed. This typically involves specifying the requirements in detail, identifying the data the application will work on and specifying the tasks that need to be performed showing how data is input, transformed and output.

With Object-Oriented Analysis (OOA), the behaviour is modelled in terms of objects and their collaborations (method calls on each other).

11.6 How can classes be identified?

Identifying a good set of classes is a key part of object-oriented design. One way to start is to identify entities, strategies and concepts by systematically reviewing the requirements specification. This activity can be initialised by looking at the nouns in the requirements and treating the named things as potential classes.

Once a small set of potential classes have been identified, the Class, Responsibility, Collaboration method (CRC — see Appendix B in Developing Java Software) can be used to identify attributes and methods for the proposed classes, along with the relationships between classes. In addition, new classes are likely to be identified and some of the potential classes dropped as they turn out not to be relevant.

With a working set of classes, attributes and methods, a process like Test-driven Development (TDD) can be started to turn the proposed classes into code. This is highly likely to lead to the further addition and removal of classes as the design falls into place. It is also possible to use (TDD) right from the beginning as a way of identifying classes.

11.7 What is the role of design?
Chapter 11: The Programming Process

Design is about determining how a software application will be realised. It involves identifying and specifying packages, detailed classes, data structures, methods, algorithms and all the details needed to actually go ahead and implement an application.

Good design is typically an exploratory process with coding and testing forming an important part of determining whether a design will work and if it is a good solution for the current context. Attempting to completely design an application before implementation and testing invariably leads to serious difficulties as it is very hard to validate a design on its own.

Design can be viewed as a step within the development process but the label ‘design’ is best applied to the entire activity of specifying and creating a working implementation.

11.8 List the properties of a well designed class.

A class should:

- Be cohesive, meaning that it should focus on representing one entity, concept or strategy. A class that is a collection of several distinct parts should be split up.
- Not be too long (too many lines of code) or have too many methods. A class that gets too long is probably not very cohesive and should be split up.
- Make proper use of encapsulation, so that the internal implementation and data structures are not exposed to users of the class. This reduces the coupling or dependency of client code on a specific class implementation. Instance variables should always be private or protected.
- Have a thin interface, meaning that the minimal set of public methods should be available.

11.9 What is the aim of testing and what are the limitations?

The aim of testing is to find errors, so tests should be focussed on those most likely to locate an error. Exhaustive testing, that is testing every possible state, is not usually possible and, in addition, testing is subject to time and budget constraints. Hence, tests should be selected to maximise the chance of errors being found. Although developers often talk of ‘all tests passing’ meaning that no errors were found, a successful test is better characterised as one which finds an error, as that error can then be removed from the software.

Tests that do not find any errors give confidence that the code works according to the specification set by the tests but cannot demonstrate that no errors exist. Testing can not prove the absence of errors in the formal sense, and the major limitation of testing is that undetected errors can still exist as no test has been written to trigger them.

11.10 Make suggestions as to how a code review might be conducted.
First and foremost, a code review should not be a confrontation. The aim is to locate errors and make suggestions for improvements to the code quality.

11.11 What is an appropriate amount of documentation for a program? What should it include?

11.12 What is the key information that you require in order to effectively maintain code written by someone else?

11.13 Summarize the strengths and weaknesses of object-oriented development.

11.14 What is the point of reuse? What features should a reusable class have?

Programming Exercises

11.1 Write Java classes corresponding to the classes shown in this UML diagram.

11.2 The UML class diagram below represents a simple order system.

Write a Java class for each class identified in the diagram.

Challenges

11.1 Consider the following specification:

"A program is required to run the controller of a burglar alarm system. A typical system consists of a number of sensors connected by individual circuits to a central control box containing the controller. The control box has a simple keypad and display. Sensors include
switches, heat detectors and motion detectors. Each sensor has an identification code which can be read by the controller to identify the sensor.

The controller allows an operator to select which sensors are active and turn on or off the system. If a sensor is triggered when the system is active, the controller must activate the alarms (a siren and a bell) and display a message on the display panel indicating which sensor is involved. The operator must enter a security code before the system is turned on or off.

1. Identify a set of classes which might be used to model the system from an object-oriented point of view.

2. Use the classes to construct a class diagram to show the structure of the system.

3. Write a program to simulate the system, using the classes.

11.2 Create a UML class diagram showing the classes and their relationships in the JDK package java.util.
12

Unit Testing
12.1 Why is testing not just necessary but fundamental to good programming practice?

Code must work correctly in order to deliver the right result. Even a single error can cause a program to fail, either by crashing immediately or, worse, by appearing to work but producing invalid results that go unnoticed. When run, a typical program will move through many millions of states, each of which needs to be valid. The sheer number of states a program can enter, combined with the range of potential errors that can made in the program code, makes the task of checking every state effectively impossible.

A number of tools and techniques help the programmer produce correct code, such as the type checking done by the compiler and the programming language specification, but testing is the technique of primary importance in producing correct code. Mathematical and proof techniques can be of use in checking code but there is no practical mathematical approach to proving that an arbitrary program is correct (indeed, the Halting Problem shows us that there is no general purpose way of even proving that a program will terminate, see http://en.wikipedia.org/wiki/Halting_problem).

The aim of testing is to find errors by running sections of code and observing the results. While some tests are used to simply confirm that the right result is computed, the majority of tests are focussed on running the code in ways most likely to find errors. Testing is the one general purpose, and practical, way that a program can be shown to work correctly.

12.2 What is a unit in unit testing?

Unit testing is based on calling methods and checking the result. Hence, a unit is a collection of related methods that can be tested within the same test class using the same fixtures declared in the test class.

A unit can be a complete class from the application being developed, or it can be some subset of the methods declared within a class that deal with a related piece of functionality. It is also possible to form a unit from methods from several class, providing those methods are dealing with the same unit of behaviour and all use the same fixtures. An interface, being a collection of related methods, is also a potential unit.

Typically when a class is first being developed, a test class is created to test the methods as they are written. When the test methods no longer all use the same fixtures, or an interface is introduced, the test class can be split up into several test classes so that the tests in each new test class do use the same fixtures. This has the affect of changing the unit from being a class to a subset of the methods in a class. It may also be a good idea to subdivide a test class if it simply
gets too long. As the development of the application proceeds, the units will be progressively refined to match the current design.

Beginners often assume that you must have one test class for each application class, and that this is the only organisation of test class. Use refactoring to move away from this position as a program is developed.

12.3 What is a Red test?

A Red test is one that has found an error when an assertion has failed or an unexpected exception gets thrown.

This can be described as a test that has succeeded, as the test has served its purpose of locating an error. Often, however, programmers talk about a red test as a failed test, meaning failed in the sense that an assertion has failed.

A Red test should be addressed immediately in order to fix the problem found. Ideally, there should never be more than one Red test at any one time.

12.4 What is a Green test?

A Green test is one that has completed without any assertions failing or any unexpected exceptions being thrown.

12.5 What is an assertion?

An assertion asserts, or checks, that the value of a boolean expression is true. If the value is false an error is raised. The boolean expression represents some property of the code being tested. For example, `assertEquals` will check that the value of two expressions is the same, so that if a method call is meant to return the value 10 `assertEquals` will compare the expected value with the actual value returned.

12.6 Why must tests be automated?

Testing manually (that is by having a person run each test step-by-step), is tedious, slow, error prone and very, very boring. Tests are likely to be chosen arbitrarily, will not be re-run and only a few tests will actually be run.

In contrast, automated testing allows large numbers of tests to be run repeatedly at high speed. The computer does all the hard work of setting up the tests, performing the tests and checking the results, which is what computers are very good at. This makes it possible to create a comprehensive test suite and use it effectively.
12.7 Why must tests be repeatable?

A unit test should always be repeatable, meaning that it can be run whenever required under exactly the same conditions. The test can then be relied on to detect when the code being tested no longer meets the specification set by the test, throughout the lifetime of that code. This avoids arbitrary decisions being made about how to perform the test or under what conditions.

Unit tests should be run every time a change is made to code and it is recompiled. This immediately identifies any errors that were made in the last change with respect to the test specification and also gives confidence that all the code still works correctly. If an error in the code is found directly after it has been introduced then the programmer (i.e., you) is in a much better position to fix the error as all the details will be fresh. In addition, by keeping each change small (literally one line of code) the location of the error can be found quickly without having to resort to a tool like a debugger. Of course, this depends on a test being in place that will detect any new error that has been introduced. If in doubt, new tests should be added.

Repeated running of a test suite is sometimes referred to as regression testing, where regression means that an error has been introduced into previously working code. If the code "regresses" then the test suite can be used to locate the cause of the error via the test or tests that detect errors.

12.8 What is a dummy object and why is it needed?

A dummy object provides a fake implementation that an be substituted, or injected, into code when running a test. The dummy object allows the code to run without having to create real objects, which may be difficult to create and initialise when testing.

12.9 What is TestNG?

TestNG is a testing framework consisting of set of annotations used to create test classes, infrastructure and a test runner to run tests and a reporting mechanism to record the results of running tests. TestNG is written in Java, requiring Java 5. Full details can be found at the TestNG website [http://testng.org](http://testng.org).

12.10 How is a TestNG test run?

By creating a testng.xml configuration file that describes the test suite and then running the framework and supplying it with the configuration file. From the command line this looks like:

```
java org.testng.TestNG testng.xml
```
(this also requires that your classpath is set correctly).

In practice it is more convenient to run TestNG either via an Ant build file or from within an IDE like Eclipse or IntelliJ IDEA. The TestNG download includes an Ant task that provides a comprehensive set of attributes for configuring the TestNG task and Ant will also manage issues like the setting up the classpath.

12.11 What does a configuration method do?

A configuration method can be called either before or after running tests. When called before a test it is used to initialise the test fixtures and any other data needed to run the test. When called after a test, a configuration method can be used to tidy-up or release any resources held by fixtures such as open files or network connections.

The @Configuration annotation is used to mark an instance method as a configuration method. There are a number of attributes that can be added to the annotation declaration to specify exactly when a configuration method is called. For example, before or after each test method, before and after a test class and so on. A full list of annotations is given in the TestNG documentation (see http://testng.org/doc/documentation-main.html#annotations).

12.12 How is a test method annotated?

A test method is annotated with the @Test annotation. The annotation has various attributes that are used to control when and how the annotated test method is run. Further information can be found at: http://testng.org/doc/documentation-main.html#annotations

12.13 List the steps of the test-driven development cycle.

1. Think. Identify the next small piece of functionality to work on, moving the design forward one small step.
2. Write a test that specifies what the next small piece of code to be written should do.
   - The piece of code should be a small method, or statement(s) with in method.
3. Write enough code to let the new test compile and nothing else.
4. Run the test and see the new test detect an error. This gives the RED state.
5. Write just enough code to make the new test complete without error. Use the simplest code you can. This gives the GREEN state.
   - Keep the code really simple, even if it is messy code or temporarily creates duplication.
6. Refactor as necessary (not needed on every iteration).
Chapter 12: Unit Testing

- All tests must run unchanged before and after refactoring.
- Clean up and restructure code.
- Remove compromises and duplication made to get test passing initially.
- An important aspect of TDD is to maintain design and code quality.
- If a test must be changed, go to step 1 and treat the change as the next iteration.

7. Repeat for next small step.

12.14 Explain refactoring and give an example.

Refactoring is the process of altering the internal structure of code without changing its external behaviour, with the goal of cleaning-up and improving the detailed design of the code. The external behaviour means that the same method calls to invoke the code being refactored must work after the refactoring. An individual refactoring is typically straightforward, such as renaming a variable or method, although it may require a number of changes throughout sections of the code. A sequence of refactorings can make substantial changes to code, while preserving the external interface in terms of the public classes and methods. Refactoring is frequently used to remove code duplication.

Refactoring is an integral part of the Test Driven Development process. After each iteration potential refactorings should be considered and applied if they will result in better quality code. An absolutely essential requirement of refactoring is that all tests must run without failure before any refactoring is performed and must still run without error after the refactoring is done.

Most integrated development environments, including Eclipse, NetBeans and IDEA, include extensive support for refactoring. Automated tools are the best, quickest and most reliable way of performing refactoring and are a good argument for using one of the development environments.

Each refactoring is given a simple name allowing it to be referred to easily. As a result a common vocabulary or language of refactorings has developed and is in wide use amongst developers. The definitive work on refactoring is: "Refactoring: Improving the Design of Existing Code", by Martin Fowler with Kent Beck, John Brant, William Opdyke, and Don Roberts (June 1999), Addison-Wesley, ISBN 0201485672. This book forms a refactoring catlog, and the names of many of the refactorings are now in common use.

Example refactorings include:

- Extract Method — remove one of more lines of code from a method into a new method, replacing the code in the original method with a method call, passing parameters as necessary. This is a standard technique for reducing the size and complexity of methods.
• Extract Class — simplify a class by removing one or more instance variables and methods into a new class.
• Pull Up Method — move a duplicated method from two or more subclasses into a superclass, so only one version of the method exists.
• Rename Method (or Variable or Class or Interface or Package) — rename a method so the name better describes the purpose of the method. Choosing a good name for a method can be hard and it can take time to identify what the best name is. It is always better to rename a method than to stick with the original but inappropriate name. The automated refactoring tools give the major advantage that they will automatically rename a method throughout the source code of a program, which can save a lot of time and avoid mistakes in finding all the places the name is used.

A full catalog of common refactorings can be found at Martin Fowler’s Refactoring website: http://www.refactoring.com/

Also see the Wikipedia article on refactoring at http://en.wikipedia.org/wiki/Refactoring for more information.

12.15 Why is duplication bad?

Duplication is a frequent source of errors and a symptom of poor design. Both design and code can be duplicated, leading to the same structures being declared multiple times within a program.

Duplicated code is often created via a copy-and-paste approach, where existing code is simply copied somewhere else in a program as a quick coding solution. When an error is discovered in one copy of the code, the programmer has to find and edit all the other copies. In practice this is error prone, as it is easy to miss one or more copies, and the programmer may not know or have forgotten that the other copies exist.

Duplication does not require identical sections of code or parts of a design. Similar elements can often be abstracted into a single more general version. One of the core goals of refactoring and the Test Driven Development process is to identify and remove duplication.

Programming Exercises

12.1 Use the test-driven approach to develop a method for calculating the nth Fibonacci number. Remember to take very small steps, one test at a time, and always write the simplest code that will pass the tests.

The Fibonacci numbers are a sequence of positive numbers where each number is the sum of the two preceding numbers. The first two numbers are zero and one, giving the sequence:
Chapter 12: Unit Testing

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, and so on.

The sequence can be computed recursively using:

\[
\text{fibonacci}(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
fibonacci(n-1) + fibonacci(n-2) & \text{if } n > 1
\end{cases}
\]

Developing a Fibonacci method starts by writing the first test:

```java
@Test
public void fibonacci_zero() {
    assertEquals(fibonacci(0), 0);
}
```

This simply asserts that \( \text{fibonacci}(0) \) is 0 as defined in the formula above. The decision has been made to name the method \( \text{fibonacci} \) and make it a static method. TestNG has, of course, been used to write the test code. In order to make this compile, the simplest version of \( \text{fibonacci} \) is written:

```java
public static int fibonacci(final int n) {
    return 0;
}
```

The test now runs and no errors are found, so we have Green. As the first step is very straightforward, we have skipped the need to see the first test fail (i.e., Red). Next a test for \( \text{fibonacci}(1) \) is added, which does result in Red:

```java
@Test
public void fibonacci_one() {
    assertEquals(fibonacci(1), 1);
}
```

The simplest way to get the new test to pass is to add an if statement to the \( \text{fibonacci} \) method:

```java
public static int fibonacci(final int n) {
    if (n == 0) {
        return 0;
    } else {
        if (n == 1 || n == 2) {
            return 1;
        } else {
            return 2;
        }
    }
}
```

The second test passes and so does a third test for \( \text{fibonacci}(2) \) as that happens to return 1 as well. Writing the fourth test for \( \text{fibonacci}(3) \) is more interesting as it requires the \( \text{fibonacci} \) method to be modified. Keeping things simple, as always, gives:

```java
public static int fibonacci(final int n) {
    if (n == 0) {
        return 0;
    } else {
        if (n == 1 || n == 2) {
            return 1;
        } else {
            return 2;
        }
    }
}
```
Noting that the definition of fibonacci defines $f_{ibonacci}(3) = f_{ibonacci}(2) + f_{ibonacci}(1)$, the method can be refactored to:

```java
public static int fibonacci (final int n) {
    if (n == 0) { return 0; }
    else {
        if (n == 1 || n == 2) { return 1; }
        else { return fibonacci(2) + fibonacci(1); }
    }
}
```

This has introduced recursion into the method. A further refactoring can eliminate the literal values passed as parameters in the recursive calls, replacing them with values computed using the parameter $n$. This makes use of the fact that $f_{ibonacci}(n) = f_{ibonacci}(n-1) + f_{ibonacci}(n-2)$.

```java
public static int fibonacci (final int n) {
    if (n == 0) { return 0; }
    else {
        if (n == 1 || n == 2) { return 1; }
        else { return fibonacci(n - 1) + fibonacci(n - 2); }
    }
}
```

With this in place tests for $f_{ibonacci}(4)$ and $f_{ibonacci}(5)$ can be added, resulting in Green and building confidence that the method works:

```java
@Test
general void fibonacci_four () {
    assertEquals ( fibonacci( 4 ) , 0 ) ;
}

@Test
general void fibonacci_five () {
    assertEquals ( fibonacci( 5 ) , 0 ) ;
}
```

At this point it is becoming very obvious that there is some serious duplication occurring in the test methods. They differ only in the value passed to the $f_{ibonacci}$ method and the expected result. We could do all the testing via a single parameterised test method, where the value passed to the $f_{ibonacci}$ call and the expected result are the parameters.

It turns out that not only can a TestNG test method be parameterised but TestNG also supports a mechanism called a DataProvider that allows a parameterised test method to be repeatedly called using different parameters each time (see [http://testng.org/doc/documentation-main.html](http://testng.org/doc/documentation-main.html) for detailed information). This allows all the tests so far, and additional ones, to be run using the following code:

```java
@DataProvider ( name = "sequence" )
public Object[][] createData () {
```
return new Object[][]{
    {0,0}, {1,1}, {2,1}, {3,2}, {4,3},
    {5,5}, {6,8}, {7,13}, {8,21}, {9,34},
    {10,55}, {11,89}, {18,2584}, {21,10946}
};
}

@Test(dataProvider = "sequence")
public void fibonacci_n(final int n, final int expected) {
    assertEquals(fibonacci(n), expected);
}

The method createData is annotated with @DataProvider (name = "sequence") to denote that it returns a data structure containing test data. The annotation parameter name allows theDataProvider to be named so it can be referred to by test methods.

The test method fibonacci_n is annotated with @Test as normal but adataProvider parameter is used to name theDataProvider that should be used when the test method is run. Given that the test method takes two parameters, theDataProvider provides two parameter values for each call of the test method. This is why theDataProvider method returns an array of pairs (each pair actually being an array with 2 items). When the tests are run the test method will be automatically be called for each pair of values declared by theDataProvider.

The tests are now Green. With theDataProvider and fibonacci_n methods in place, all the other test methods can be deleted as they are now redundant. They have, however, served their purpose in getting development to this point.

Next it is time to consider any other refactoring that might take place. The test code does not need any further work now but thefibonacci method can be improved. The first step is to note thatfibonacci(2) does not need to be treated as a special case as it can be computed recursively. This gives:

```java
public static int fibonacci (final int n) {
    if (n == 0) { return 0; }
    else {
        if (n == 1) { return 1; }
        else { return fibonacci(n - 1) + fibonacci(n - 2); }
    }
}
```

A little better but we still have a nested if statement, which looks clumsy. Noting that if n<2 the method can just return n, fibonacci can be further simplified to:

```java
public static int fibonacci (final int n) {
    if (n < 2) { return n; }
    else { return fibonacci(n - 1) + fibonacci(n - 2); }
}
```

The end result of all this is the following code:
import static org.testng.Assert.assertEquals;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class Fibonacci {
    public static int fibonacci (final int n) {
        if (n < 2) { return n; }
        else { return fibonacci (n - 1) + fibonacci (n - 2); }
    }
    @DataProvider (name = "sequence")
    public Object[][] createData () {
        return new Object[][] {
            { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 },
            { 5, 5 }, { 6, 8 }, { 7, 13 }, { 8, 21 }, { 9, 34 },
            { 10, 55 }, { 11, 89 }, { 18, 2584 }, { 21, 10946 }
        };
    }
    @Test (dataProvider = "sequence")
    public void fibonacci_n (final int n, final int expected) {
        assertEquals (fibonacci (n), expected);
    }
}

For convenience the fibonacci method has been declared in the test class while being developed.

In summary, this answer has shown a step-by-step test-driven approach to developing a fibonacci method. It might be argued that in a realistic development setting many of the steps could have left out but that misses the point of deliberately taking small steps, all the time writing tests and thinking carefully about design and refactoring. We are left with a method that has been tested and that we have good confidence in.

Or do we? Doing some more research on Fibonacci numbers reveals some further issues. First there is the question of whether 0 is actually a Fibonacci number at all. Then there is the problem that Fibonacci numbers quickly get too large to be represented by values of type int or long. Also, our fibonacci method can be called with negative values, always returning zero. Should it throw an exception instead? To address these issues we need to go back and specify the method rather more carefully. We leave that as a further exercise for you.

12.2 Could the method or variable names in the extended class Person be improved to make the code even easier to understand? Suggest some new names and see if they work better.

12.3 Make the calculateAge method in the extended class Person public and provide a set of tests for it in class PersonTest. Are your new tests any different from those for the method getAgeOn? Was it worth making calculateAge public?

12.4 Follow the test-driven approach to write a basic DateValidator class for use with the Person class.
12.5 Modify the Person class getAgeOn method to perform date validation. Extend your validator class and interface as necessary, using the test-driven approach obviously!

12.6 Use test-driven development to add a getEmailAddress method to class Person. Email address must be validated to check they have the correct structure.

12.7 Evolve the classes so as to remove all the String parameters from Person.

**Challenges**

12.1 Use test-driven development to write a program using class Person that allows the user to:

- Enter a list of famous historical people (name and birth date).
- elect a person from the list and print their age on a selected date.
13
Test-driven Programming Strategies
Chapter 13: Test-driven Programming Strategies

Self-review Questions

13.1 What are the aims of a well-designed unit test?

13.2 What are the advantages and disadvantages of using real File objects when testing?

13.3 When does the design of application code have to be modified to facilitate testing?

13.4 What is a dummy object?

A dummy object provides a fake implementation of a real object. A dummy object can be substituted for the real object when running a test, avoiding the need for the test to depend on creating and using a real object. This is particularly useful in situations where the real object is difficult to create, as it depends on creating other objects, and also where the real object makes use of resources such as files.

A dummy object is an instance of a dummy class that either implements an interface or subclasses the real class. A reference variable of the interface or superclass type can refer to the dummy object, allowing it to be injected into code being testing in place of an object of the real class. Typically any inherited behaviour is ignored and the dummy class overrides methods to check they are called or that certain parameter values have been passed.

A dummy class is typically written by a programmer as required, often as a nested class within a test class so that it forms part of the infrastructure of the test class. Commonly used dummy classes can be collected into a small class library and reused in different projects.

13.5 What is a mock object?

A mock object also provides a fake implementation but the mock object is generated automatically at runtime from an interface or class. In addition, the mock object has the ability to automatically check its usage in terms of method calls made and parameter values passed.

13.6 What is ‘injection’?

Injection is used to insert a mock or dummy object into code being tested, in place of a real object. This can be done is several ways, the simplest of which is when the method being tested can be passed a mock object reference via a parameter normally used for a real object. If this cannot be done, the method being tested may need to be refactored to replace an expression with a method call for a method can that be overridden in a subclass created when testing. The overriding method can then create and inject a mock object.
13.7 Outline the advantages and disadvantages of mock compared to dummy objects.

13.8 Outline how a GUI object can be located for testing.

13.9 What features of a GUI can and cannot be unit tested?

Programming Exercises

13.1 Extend the SwingAssert class to include assert methods that check a component like a button is in the correct panel and a panel has the correct layout manager.

13.2 Extend the directory structure used when testing the `matchFiles` method to include another sub-directory and confirm that three files with the same name will be matched. Do this for both the live file test class and the dummy file test class.

13.3 Consider a test to confirm that a window is the correct size when it is displayed. Show how this test could be written and how needing to know the size affects the design of the GUI.

13.4 Use the test-driven approach to modify the searcher GUI to display a warning dialogue when the search button is clicked but no file name has been specified.

Challenges

13.1 Modify the FileNameSearch class to use regular expressions to match file names.

13.2 Use the test-driven approach to replace the JTextArea used to display the list of files found with a JList.

13.3 Extend Searcher to search the contents of files for a string specified using a regular expression.

13.4 Use test-driven development to write an application that searches websites for HTML files with a specific name.
14
Programming Tools
Self-review Questions

14.1 Identify the directories used to structure a project and what goes into each directory.

14.2 Outline the build process.

14.3 What is an Ant target?

The actions carried out by a build file, such as creating a zip file or running the tests, are organised as a set of named targets. Each target is a sequence of tasks that perform individual actions like creating a directory or compiling a file. A target can depend on other targets, so running one target such as the 'all' target results in all the actions needed to build a project being carried out.


14.4 What is an Ant task?

A task is an individual action carried out within a target. Tasks typically have a wide range of attributes and nested elements to control their behaviour.

14.5 How does Ant manage classpaths?

14.6 What are the key features of version control?

14.7 What is the difference between importing and checking out a project?

14.8 What is branching and why is it needed?

14.9 What are the key features of an IDE?

14.10 What is a debugger?

Programming Exercises
Challenges

14.1 Modify the example Ant build file to create and use separate classes and tests directories in the build directory.

14.2 Add a target to the example build file that creates a .zip file containing a .jar file with the program code and libraries, and the user documentation from the doc directory.

14.3 Create an example project and import it into version control. Then try the various commands to see how they work.

14.4 Download an IDE like Eclipse and start experimenting with it.

Challenges

14.1 Write an Ant build file to send an email reporting the result of a build.

14.2 Write Ant targets to import, update, commit and check out a project from version control.
Part 3

Case Studies in Developing Programs
Introducing the Case Studies
Chapter 15: Introducing the Case Studies

There are no self-review questions, exercises or challenges in this chapter.
16
Contacts Book
## Programming Exercises

16.1 Add the ability to sort the data into an order specified by the user at run time.

You may want to treat this as an opportunity to rewrite the whole application so that the first names and last names of the contacts are kept separately and so can be sorted on separately.

16.2 Add the ability to commit edits to the contacts data.

The basic infrastructure is all in place. The data can already be edited, what is needed is a button to press to action any edits made to the data. Editing a person’s name is of course different from editing their contact details.

You might want to consider what happens when the user quits the application but there are pending edits.

16.3 Add the ability to add and remove data fields from the set of data known about a contact.

16.4 Add the ability to add and delete new contacts.

16.5 Amend the application so that when it is started it opens the file that was open when the application was last closed.

You might also want to think about adding a list of the last three or four files opened, making them available as a menu then the top of that list (i.e. the last file opened) can be treated as the default file to open.

## Challenges

16.1 Write and integrate into the application a class VCardContactFile to provide the ability for the application to read and write vCard format files as well as XML format files.

Remember that you want to be able to read one format and write the same or a different format.
Pedestrian Crossing Simulation
Chapter 17: Pedestrian Crossing Simulation

Programming Exercises

17.1 There is a ‘bug’ in the pedestrian crossing: if a pedestrian request button is pressed whilst the pedestrian lights are green or flashing green the button lights up ‘WAIT’ but causes no sequencing. We believe that a button press on pedestrian green should do nothing and a button press on flashing should schedule a new sequencing-flashing green means pedestrians should not start crossing, they should only start on green. Implement this bug fix.

17.2 Undertake Direction 1 of the fifth round of development reworking the use of threads so as to get more accuracy to the switching on and off of the lights.

17.3 Undertake the development of a simulation of a right-hand-drive crossroads where the up traffic stream has a left filter.

Challenges

17.1 Investigate the subject of Discrete Event Simulation and do the initial design of a full simulation with animation of the vehicles in the pedestrian crossing simulation.
Part 4

The Java Programming Language in Detail
There are no self-review questions, exercises or challenges in this chapter.
Variables Types and Expressions
Self-review Questions

19.1 Which of the following are illegal variable names and why?

- aName, 2D, WIDTH, Position, HelgHt, this, x&y, whiletrue, name$, current_position, Float

The name 2D is not allowed as it starts with a numeric digit. x&y is not allowed as it contains a symbol not permitted in variable names. Float is not allowed since it is the name of a built-in type.

19.2 What is seen if the following strings are displayed on the computer screen?

- "Hello\nWorld\n!"
- "\t\t\t\tHello"
- "\\\\\\\nHello"
- "1.23e+9f"
- "Hello\b\bWorld"
- "\\Hello"

19.3 Are any of the following expressions using ++ and – legal? If not, why not?

```java
int j = 5;
j++ ++;
j(++) ;
(j++)++; 
j++-- ;
(j)++; 
```

- `j++ ++` is not allowed since the result of `j++` is a value not a variable and the `++` operator can only be applied to a variable. `(j++)++` is not allowed for exactly the same reason. Similarly `j++--`, `--` must be applied to a variable not a value.

- `(j)++` is allowed since the result of the expression `(j)` is the variable `j`.

19.4 What value is assigned to `n` in each of the following?

```java
int n = 0;
int x = 1;
n = x++ + x++ ;
n = n++ - x++ ;
n = x-- + -x++ ;
```
The value assigned for the three assignment statements are:

1. 3
2. 0
3. 1

which can be checked by executing a small program:

```java
public class S_19_4 {
    private void run () {
        int n = 0;
        int x = 1;
        n = x++ + x++;
        System.out.println ( n ) ;
        n = n++ - x++;
        System.out.println ( n ) ;
        n = x-- + -x++;
        System.out.println ( n ) ;
    }
    public static void main ( final String[] args ) {
        ( new S_19_4 ( ) ).run ( ) ;
    }
}
```

19.5 Why can't the largest integer be negated in Java?

19.6 What does the expression:

```java
  i < 0 ? 0 : i > 10 ? 1 : i > 20 ? 2 : 3
```

evaluate to for values of i between -5 and 50?

For negative values of i the value is 0.

For values of i such that 0 ≤ i ≤ 10, the value is 3.

For all values of i > 10 the value is 1.

This result can be verified by writing, compiling and executing a small program:

```java
public class S_19_6 {
    private void run () {
        for ( int i = -5 ; i < 51 ; ++i ) {
            int n = i < 0 ? 0 : i > 10 ? 1 : i > 20 ? 2 : 3;
            System.out.println ( i + " : " + n ) ;
        }
    }
    public static void main ( final String[] args ) {
    }
}
```
When executed this gives:

-5: 0
-4: 0
-3: 0
-2: 0
-1: 0
0: 3
1: 3
2: 3
3: 3
4: 3
5: 3
6: 3
7: 3
8: 3
9: 3
10: 3
11: 1
12: 1
13: 1
14: 1
15: 1
16: 1
17: 1
18: 1
19: 1
20: 1
21: 1
22: 1
23: 1
24: 1
25: 1
26: 1
27: 1
28: 1
29: 1
30: 1
31: 1
32: 1
33: 1
34: 1
35: 1
36: 1
37: 1
38: 1
39: 1
40: 1
41: 1
42: 1
43: 1
44: 1
45: 1
46: 1
47: 1
48: 1
49: 1
50: 1
19.1 Write a program to use each kind of cast between primitive types. Which casts lose information and what is lost?

```java
public class E_19_1 {
    private byte b = (byte) 5;
    private short s = (short) 5;
    private int i = 5;
    private long l = 5;
    private float f = 5.0f;
    private double d = 5.0;
    private void checkByte() {
        byte v;
        v = (byte) s;
        v = (byte) i;
        v = (byte) l;
        v = (byte) f;
        v = (byte) d;
    }
    private void checkShort() {
        short v;
        v = (short) b;
        v = (short) i;
        v = (short) l;
        v = (short) f;
        v = (short) d;
    }
    private void checkInt() {
        int v;
        v = (int) b;
        v = (int) s;
        v = (int) l;
        v = (int) f;
        v = (int) d;
    }
    private void checkLong() {
        long v;
        v = (long) b;
        v = (long) s;
        v = (long) i;
        v = (long) f;
        v = (long) d;
    }
    private void checkFloat() {
        float v;
        v = (float) b;
        v = (float) s;
        v = (float) i;
        v = (float) l;
        v = (float) d;
    }
    private void checkDouble() {
```
Chapter 19: Variables Types and Expressions

```java
double v;
v = (double) b;
v = (double) s;
v = (double) i;
v = (double) l;
v = (double) f;
}
private void run () {
    checkByte ();
    checkShort ();
    checkInt ();
    checkLong ();
    checkFloat ();
    checkDouble ();
}
public static void main ( final String[] args ) {
    ( new E_19_1 ( ) ).run ( ) ;
}
```

19.2 Write a program to investigate the effects of overflow. What happens to the result when an overflow occurs when multiplying two integers or doubles?

19.3 Write a program to illustrate how each primitive type is represented as a string. What happens to the value null?

The program:

```java
public class E_19_3 {
    private byte b = (byte) 5;
    private short s = (short) 5;
    private int i = 5;
    private long l = 5;
    private float f = 5.0f;
    private double d = 5.0;
    private void run () {
        System.out.println ( b );
        System.out.println ( s );
        System.out.println ( i );
        System.out.println ( l );
        System.out.println ( f );
        System.out.println ( d );
        System.out.println ( (String) null );
    }
    public static void main ( final String[] args ) {
        ( new E_19_3 ( ) ).run ( ) ;
    }
}
```

results in the output:
The reason for explicitly casting the `null` literal is because without it there is ambiguity: There are two overloads of the `System.println` method that could apply, `String` or `char[]`, so the cast is needed to disambiguate.

19.4 Write a program that demonstrates the effects of applying the shift operators.

19.5 Write a program that inputs an integer value and uses the bitwise operators to print out the binary representation of the integer.

The algorithm we use here is ‘peel off’ the least significant bit of a number and append a character (‘0’ or ‘1’) to a `StringBuilder` depending on whether the bit was 0 or 1 respectively. We loop for 32 bits (the size of an `int`) or until the value is zero, i.e. all bits left are zero and so don’t need printing. This results in a string representation which is in the reverse order to the way we need to print it, so we reverse it prior to printing:

```java
public class E_19_5 {
    private void run () {
        final Input input = new Input ( ) ;
        System.out.print ( "Enter and integer: " ) ;
        int number = input.nextInt ( ) ;
        StringBuilder sb = new StringBuilder ( ) ;
        for ( int i = 0 ; i < 32 ; ++i ) {
            if ( ( number & 1 ) == 1 ) { sb.append ( '1' ) ; }
            else { sb.append ( '0' ) ; }
            number >>= 1 ;
            if ( number == 0 ) { break ; }
        }
        System.out.println ( sb.reverse ( ) ) ;
    }
    public static void main ( final String[] args ) {
        ( new E_19_5 ( ) ).run ( ) ;
    }
}
```
20

Flow Control
Self-review Questions

20.1 Run the programs While1 and While2 and determine what the introduced bug is. Why do we get this behaviour? What can we do to fix it (apart from using the original, correctly behaved program)?

20.2 The repetition of the Boolean expression in Do3 can be avoided. How is this possible?

20.3 Compare and contrast the four programs, iterative and recursive implementation of factorial numbers and Fibonacci series programs, using the factors of time to implement, execution speed and memory usage.

See Exercise 12.1

Programming Exercises

20.1 Re-code the program Break1 without using the break statement. Which version of the program do you prefer? Why?

20.2 Implement the factorial function without using recursion, i.e. implementing the iteration using iteration statements. Have you tested your factorial implementation? If you have, you will have noticed a serious flaw in the implementation. What is it and why is it there?

We came up with the function:

```java
public static int factorial ( final int n ) {
    int product = 1 ;
    for ( int i = 2 ; i <= n ; ++i ) { product *= i ; }
    return product ;
}
```

Simply outputing the first 20 values, highlights the problem:

0: 1
1: 1
2: 2
3: 6
4: 24
5: 120
6: 720
7: 5040
8: 40320
9: 362880
factorial(14) is smaller than factorial(13) according to this function, but that is clearly wrong. The problem is that an integer is a 32-bit binary quantity and the value of factorial(14) is too large to be represented in an int. So overflow occurs and no expression evaluation is correct. The factorial function creates very large values very rapidly, the only way of working with numbers such as these is to use the BigDecimal class for expression evaluation.

20.3 Implement the Fibonacci function using recursion rather than implementing the iteration using iteration statements.

An example recursive method is:

```java
public static int fibonacci ( final int n ) {
    if ( n < 2 ) { return 1; } 
    else { return fibonacci ( n - 1 ) + fibonacci ( n - 2 ) ; } 
}
```

For an explanation of how this method was written see Exercise [12.1](#).
21

Classes and Packages
Self-review Questions

21.1 In the class Blah:

```java
class Blah {
    public String hello = "hello" ;
    public final String world = "world" ;
    protected int count = 0 ;
    private float length = 2.345f ;
    long size = 123432L ;
}
```

Why is the length variable completely redundant?

The field `length` is private and is not used in the public or protected interface. This means the field cannot ever been accessed. If there were methods then we would need check whether the field was used in the methods, but as there are only field defined in this class and private fields are useless.

21.2 Why are the cast expressions necessary in the first two calls of `max` in the `Methods2` program?

21.3 What happens if the following calls had existed in `Methods2`?

```java
maxObject.max ( 3.0 , 5 ) ;
maxObject.max ( 3 , 3l ) ;
maxObject.max ( 3.4f , 5.0 ) ;
```

21.4 In program `Class2` each `max` method is identical except for the type name used. Is it possible to write a single `max` method that will work for any of the integral types? If it is possible, would you use the single method?

21.5 Explain why the statement `s2.f()` in `StaticMethods2` is legal.

21.6 Work through the execution of program `MemberClass2` as a pencil and paper exercise to determine how it works. What does the program display?

21.7 What does the `MemberClass3` program display?
21.1 Write a class to represent complex numbers, where the real and imaginary parts are represented by double values. Provide at least two constructors for creating complex number objects and implement the basic operations of addition and subtraction.
22

Inheritance and Interfaces
Self-review Questions

22.1 What is the output from running Inheritance1? Why is this the output?

22.2 What is the output from running Inheritance2? Why is this the output?

22.3 What is the output from running Override1? Why is this the output?

22.4 What is the output from running Final1? Why is this the output?

22.5 Referring to the MethodCallExample4 example on page 742, what happens with the following calls:

\[
\begin{align*}
&\text{s.f ( 1.0 , 1.0 ) ;} \\
&\text{s.f ( 1.0f , 1 ) ;} \\
&\text{s.f ( 1.0f , 1.0f ) ;} \\
&\text{s.f ( 1L , 1 ) ;} \\
&\text{s.f ( 1L , 1L ) ;}
\end{align*}
\]

22.6 Looking at the Call1 program:

1. Which methods are overridden and which overloaded?
2. What does the program display?
3. What is the significance of the last statement?

22.7 What does program Lookup1 display? Why is this the output?

22.8 What does program Super2 display? Why is this the output?

22.9 What does program Abstract1 display? Why is this the output?

Programming Exercises

22.1 Write a program incorporating all the different varieties of scope that displays the variable and method search algorithm, i.e. verifies the published variable and method lookup rules.
22.2 Write a program based on the outline on page 742 to verify that the most specific rules work when a method with two or more reference type parameters is used.

22.3 Write a program to explore the interaction between member classes, containment and inheritance hierarchies, and overloaded methods and overridden methods.
23.1 What is the output of the Exception1 program? Why is this the output?

23.2 Why does the default constructor of class Exception3 appear with an empty body in the program above? Would it make any difference if it were deleted?
24

Threads and Concurrency
Chapter 24: Threads and Concurrency

Self-review Questions

24.1 Would removing the synchronized statement, and just leaving the statement body, ever make any difference to the execution of the program TestThread7?

Programming Exercises

24.1 Amend program ThreadTest4 to include a thread running at a higher priority that periodically runs and displays the status of each Calculate thread.

24.2 Amend the following program so that PrimeFilter extends Thread and each filter object runs as a separate thread. Think very carefully about how the filter objects have to communicate and how the run method might work.

```java
// Class to represent a filter object.
class PrimeFilter {
    private final int prime; // Prime number held by filter object.
    private PrimeFilter next; // Reference to next filter object.
    public PrimeFilter() { this(2); }
    private PrimeFilter(final int prime) {
        this.prime = prime;
        System.out.println(prime + " is prime");
    }
    // Process the test value to see if it is divisible by the prime held by the current object. If it is then it can be rejected, otherwise it is passed on to the next prime in the list. If there is no object to pass it on to then we have found a new prime so create a new object.
    public void process(final int i) {
        if ((i % prime) != 0) {
            if (next != null) { next.process(i); }
            else { next = new PrimeFilter(i); }
        }
    }
}
public class SieveOfEratosthenes {
    public static void main(final String[] args) {
        final PrimeFilter two = new PrimeFilter();
        for (int i = 3; i < 10000; i += 2) { two.process(i); }
    }
}
```

24.3 Write a test program to test the SimpleThreadSafeQueue class and demonstrate that it works correctly.