

6

Classes and Objects

Self-review Questions

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 ( '+' );
        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.

```

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" );
        assertTrue ( "String constructor, international format", "+44 20 75852200" , pn.toStringInternationalFormat ( ) );
        assertTrue ( "String constructor, national format", "(20) 75852200" , pn.toStringNationalFormat ( ) );
        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

any sized chessboard.

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 8×8 chessboard:

```
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.

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:

```

/**
 * A basic matrix class.
 *
 * @author Graham Roberts and Russel Winder
 * @version 2006-11-18
 */
public class Matrix {
    private double[][] elements ;
    /**
     * Create a Matrix 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 doubles and creates a
     * Matrix. 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 ; }

```

```

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:

```

/**
 * 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
 */
public 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 ) ) > 0.000000000000001 ) {
                        System.out.println ( m + " -- values not equal." );
                        System.out.println ( "\ta = " + a );
                        System.out.println ( "\tb = " + b );
                    }
                }
            }
        }
    }
    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 ( );
    }
}

```

```

    mt.testSubtract ( ) ;
    mt.testMultiply ( ) ;
}
}

```

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:

```

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 ) {

```

```

    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 < 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 ( ) ) ;
}
private Entry getEntry ( final int row , final int column ) {
    Iterator<Entry> i = entries.iterator ( ) ;
    while ( i.hasNext ( ) ) {
        final Entry e = i.next ( ) ;
        if ( ( row == e.getRow ( ) ) && ( column == e.getColumn ( ) ) ) { return e ; }
    }
    return null ;
}
}
}

```

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.

```

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 ) ) > 0.000000000000001 ) {
                        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 implemented 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 $\text{row} * \text{columns} + \text{column}$ to 'linearize' the integer pair, and can therefore use `HashMap`:

```
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 ) {
            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 < 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:

```

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 ;
}
public String toString () { return value.toString () + suit.toString () ; }
/**
 * <code>Card</code> 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 ) {

```



```

        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 ( ) + "\n" ) ;
        }
        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.

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.*