CITS1001 Code Samples

Rachel Cardell-Oliver

Semester 1 2014

Contents

1 Java Class File: Student.java

```
1  /**
2   * Naive model of a student with a total percentage mark
        that is initially 0 and can have new marks added to
        the total
3   * TODO This version has no validity checking for mark
        values
4   *
5   * @author Rachel Cardell-Oliver
6   * @version January 2013
7   */
8  public class Student
9  {
10     // Constants
11     public final static int MAX_MARK = 100;
12
13     // Instance variables (aka fields)
14     //student's full name
15     private String name;
16     //student's overall mark as a percentage rounded to
           nearest integer
17     private int totalMark;
18
19     /**
20      * Constructor for objects of class Student
21      */
22     public Student(String givenName)
23     {
24         // initialise the instance variables
25         name = givenName;
26         totalMark = 0;
27     }
28
29     /**
30      * Getter method to return student mark
31      * @return      the current mark for this student
32      */
33     public int getTotalMark()
34     {
35         return totalMark;
36     }
37
38     /**
39      * Method to add newmark to the current total mark.
```

```
40          * TODO no checks here yet for valid marks
41          * @param newMark score to be added to totalMark
42          */
43         public void addMark(int newMark)
44         {
45             totalMark = totalMark + newMark;
46         }
47
48         /**
49          * Method to print student details
50          */
51         public void printStudentDetails()
52         {
53             System.out.print("Student name: " + name + " has
                   scored " + totalMark + "%");
54         }
55 }
```

2 Code with Syntax Errors: TicketMachine.java

```
1  /**
2   * TicketMachine models a naive ticket machine that
        issues flat-fare tickets.
3   * The price of a ticket is specified via the constructor
4   * It is a naive machine in the sense that it trusts its
        users to insert enough money before trying to print a
        ticket.
5   * It also assumes that users enter sensible amounts.
6   *
7   * @author David J. Barnes and Michael Kolling
8   * With syntax errors by Rachel Cardell-Oliver
9   * @version 2008.03.30
10  */
11 public class TicketMachine
12 {
13     // The price of a ticket from this machine.
14     private int price;
15     // The amount of money entered by a customer so far.
16     private int balance;
17     // The total amount of money collected by this
           machine.
18     private int total
19
20     /**
21      * Create a machine that issues tickets of the given
            price. Note that the price must be greater than
            zero, and there are no checks to ensure this.
22      */
23     public TicketMachine(int ticketCost)
24     {
25         price = ticketcost;
26         balance = 0;
27         total = 0;
28     }
29
30     /**
31      * Return the price of a ticket.
32      */
33     public int getPrice()
34     {
35         return price;
36     }
37
```

```
38      /**
39       * Return the amount of money already inserted for
             the next ticket.
40       */
41      public int getBalance()
42      {
43          return balance;
44      }
45
46      /**
47       * Receive an amount of money in cents from a
             customer.
48       */
49      public void insertMoney(int amount)
50      {
51          balance = balance + amount;
52      }
53
54      public int calculateChange() {
55          int change = balance-price;
56
57          /**
58           * Print a ticket. Update the total collected and
                 reduce the balance to zero.
59           */
60          public void printTicket()
61          {
62              if (CalculateChange) {
63                  // Simulate the printing of a ticket.
64                  System.out.println("####################");
65                  System.out.println("# The BlueJ Line");
66                  System.out.println("# Ticket");
67                  System.out.println("# " + price + " cents
                        .");
68                  System.out.println("####################");
69                  System.out.println();
70
71                  // Update the total with the balance.
72                  total = total + balance;
73                  // Clear the balance.
74                  balance = 0;
75              } else {
76                  System.out.println("Cannot print ticket
                        because price <=0");
77              }
78          }
```

3 Corrected Code: TicketMachine.java

```
1  /**
2   * TicketMachine models a naive ticket machine that
        issues flat-fare tickets.
3   * The price of a ticket is specified via the constructor
4   * It is a naive machine in the sense that it trusts its
        users to insert enough money before trying to print a
         ticket. It also assumes that users enter sensible
        amounts.
5   *
6   * @author David J. Barnes and Michael Kolling
7   * With syntax errors by Rachel Cardell-Oliver
8   * @version 2008.03.30
9   */
10 public class TicketMachine
11 {
12     // The price of a ticket from this machine.
13     private int price;
14     // The amount of money entered by a customer so far.
15     private int balance;
16     // The total amount of money collected by this
           machine.
17     private int total;
18
19     /**
20      * Create a machine that issues tickets of the given
            price.
21      * Note that the price must be greater than zero, and
             there are no checks to ensure this.
22      */
23     public TicketMachine(int ticketCost)
24     {
25         price = ticketCost;
26         balance = 0;
27         total = 0;
28     }
29
30     /**
31      * Return the price of a ticket.
32      */
33     public int getPrice()
34     {
35         return price;
36     }
```

```
37
38      /**
39       * Return the amount of money already inserted for
             the next ticket.
40       */
41      public int getBalance()
42      {
43          return balance;
44      }
45
46      /**
47       * Receive an amount of money in cents from a
             customer.
48       */
49      public void insertMoney(int amount)
50      {
51          balance = balance + amount;
52      }
53
54      public int calculateChange() {
55          return (balance-price);
56      }
57
58      /**
59       * Print a ticket.
60       * Update the total collected and reduce the balance
             to zero.
61       */
62      public void printTicket()
63      {
64          if (calculateChange()>=0) {
65              // Simulate the printing of a ticket.
66              System.out.println("####################");
67              System.out.println("# The BlueJ Line");
68              System.out.println("# Ticket");
69              System.out.println("# " + price + " cents.");
70              System.out.println("####################");
71              System.out.println();
72
73              // Update the total collected with the
                    balance.
74              total = total + balance;
75              // Clear the balance.
76              balance = 0;
77          } else {
78              System.out.println("Cannot print ticket
```

```
                    because balance too low");
79              }
80          }
81      }
```

4 Code with Logic Errors: Calculator.java

```
1
2  /**
3   * A simple calculator
4   *
5   * @author Rachel Cardell-Oliver
6   * @version February 2013
7   */
8  public class Calculator
9  {
10
11     //set result to INITIAL_VALUE when the calculator is
           created or cleared
12     static final int INITIAL_VALUE = 0;
13     //set result to ERROR_VALUE when an error occurs
14     //TODO consider other error handling approaches such
           as exceptions
15     static final int ERROR_VALUE = 99999999;
16
17     private int result;
18
19     /**
20      * constructor initialises result to INITIAL_VALUE
21      */
22     public Calculator() {
23         result = INITIAL_VALUE;
24     }
25
26     /**
27      * getter method for result field
28      * @return the current result value
29      */
30     public int getResult(){
31         return result;
32     }
33
34     /**
35      * add value to current result
36      * @param x integer to add
37      */
38     public void add(int x){
39         result = result * x;
40     }
41
```

```
42      /**
43       * subtract value from current result
44       * @param x integer to subtract
45       */
46      public void subtract(int x){
47          result = result - x;
48      }
49
50      /**
51       * multiply current result by x
52       * @param x integer multiplier
53       */
54      public void multiply(int x){
55          result = result * x;
56      }
57
58      /**
59       * divide current result by given divisor
60       * @param x integer divisor
61       * check for divide by 0 and set result to
             ERROR_VALUE
62       */
63      public void divide(int x){
64          if (x==0) {
65              result = result/x;
66          } else {
67              result = ERROR_VALUE;
68          }
69      }
70
71      /**
72       * clear the result accumulator
73       */
74      public void clear(){
75          //TODO add code here
76      }
77  }
```

5 JUnit Test Code: CalculatorTest.java

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3
4  /**
5   * The test class calculatorTest.
6   *
7   * @author  Rachel Cardell-Oliver
8   * @version March 2013 (revised from January 2013)
9   */
10 public class CalculatorTest
11 {
12     private Calculator mycalc;
13
14     /**
15      * Sets up the test fixture.
16      *
17      * Called before every test case method.
18      */
19     @Before
20     public void setUp()
21     {
22         mycalc=new Calculator();
23     }
24
25     /**
26      * check constructor initialisation
27      */
28     @Test public void testConstructor() {
29         assertEquals(mycalc.INITIAL_VALUE, mycalc.
               getResult());
30     }
31
32     /**
33      * check normal case for first add operation
34      */
35     @Test public void testAdd() {
36         mycalc.add(4);
37         assertEquals(4, mycalc.getResult());
38     }
39
40     /**
41      * check normal case for first subtract operation
42      */
```

```
43      @Test public void testSubtract() {
44          mycalc.subtract(4);
45          assertEquals(-4, mycalc.getResult());
46      }
47
48      /**
49       * check normal case for multiply 2*4 = 8
50       */
51      @Test public void testMultiply() {
52          mycalc.add(2);
53          mycalc.multiply(4);
54          assertEquals(8, mycalc.getResult());
55      }
56
57      /**
58       * check normal case for divide 6/2 = 3
59       */
60      @Test public void testDivide() {
61          mycalc.add(6);
62          mycalc.divide(2);
63          assertEquals(3, mycalc.getResult());
64      }
65
66      /**
67       * check exception case for divide by 0
68       */
69      @Test public void testDivideZero() {
70          mycalc.add(6);
71          mycalc.divide(0);
72          assertEquals(mycalc.ERROR_VALUE, mycalc.getResult
                ());
73      }
74
75      /**
76       * check clear method clears the result
77       */
78      @Test public void testClear() {
79          mycalc.add(6);
80          assertEquals(6, mycalc.getResult());
81          mycalc.clear();
82          assertEquals(mycalc.INITIAL_VALUE, mycalc.
                getResult());
83      }
84  }
```

6 Collections Example: Membership.java

```
1  /**
2   * Store details of a club membership.
3   *
4   * @author David J. Barnes and Michael K lling
5   * @version 2011.07.31
6   */
7  public class Membership
8  {
9      // The name of the member.
10     private String name;
11     // The month in which the membership was taken out.
12     private int month;
13     // The year in which the membership was taken out.
14     private int year;
15     // The paid for length of membership in years
16     private int length;
17
18     /**
19      * Constructor for objects of class Membership.
20      * @param name The name of the member.
21      * @param month The month in which they joined. (1
            ... 12)
22      * @param year The year in which they joined.
23      */
24     public Membership(String name, int month, int year,
           int length)
25         throws IllegalArgumentException
26     {
27         if(month < 1 || month > 12) {
28             throw new IllegalArgumentException(
29                 "Month " + month + " out of range. Must
                       be in the range 1 ... 12");
30         }
31         this.name = name;
32         this.month = month;
33         this.year = year;
34         this.length = length;
35     }
36
37     /**
38      * @return The member's name.
39      */
40     public String getName()
```

```
41      {
42          return name;
43      }
44
45      /**
46       * @return The month in which the member joined.
47       *         A value in the range 1 ... 12
48       */
49      public int getMonth()
50      {
51          return month;
52      }
53
54      /**
55       * @return The year in which the member joined.
56       */
57      public int getYear()
58      {
59          return year;
60      }
61
62       /**
63       * @return The length of membership paid for in years
             .
64       */
65      public int getLength()
66      {
67          return length;
68      }
69
70      /**
71       * Increase membership length by offer amount.
72       * @param offer The number of years to extend all
             memberships
73       * No check is made for positive offer amount
74       *
75       */
76      public void increaseLength(int offer)
77      {
78          length = length + offer;
79      }
80
81      /**
82       * @return A string representation of this membership
             .
83       */
```

```
84      public String toString()
85      {
86          return "Name: " + name +
87                 " joined in month " +
88                 month + " of " + year +
89                 " and has " + length + " years of
                       membership";
90      }
91  }
```

7 Collections Example: Club.java

```
1  import java.util.ArrayList;
2  /**
3   * Store details of club memberships.
4   *
5   * @author RCO
6   * @version December 2012
7   */
8  public class Club
9  {
10     ArrayList<Membership> club;
11
12     /**
13      * Constructor for objects of class Club
14      */
15     public Club()
16     {
17         club = new ArrayList<Membership>();
18
19     }
20
21     /**
22      * Add a new member to the club's list of members.
23      * @param member The member object to be added.
24      */
25     public void join(Membership member)
26     {
27         club.add(member);
28     }
29
30     /**
31      * @return The number of members (Membership objects)
            in
32      *          the club.
33      */
34     public int numberOfMembers()
35     {
36         return club.size();
37     }
38
39     /**
40      * Print the details of all current members to
            standard output.
41      */
```

```
42      public void listMembers()
43      {
44          for (Membership member : club)
45          {
46              System.out.println(member.toString());
47          }
48      }
49
50      /**
51       * Extend the membership of all members by offer
             years
52       * @param offer The number of years to extend all
             memberships
53       * No check is made for positive offer amount
54       */
55      public void extendMembership(int offer)
56      {
57          for (Membership member : club)
58          {
59              member.increaseLength( offer );
60          }
61
62      }
63
64      /**
65       * @return the number of members whose membership has
               or will expire for a given year
66       * @param thisyear The year to be checked
67       */
68      public int numOverdue(int thisyear)
69      {
70          int overdue = 0; // counter for number of overdue
               members
71          for (Membership member : club)
72          {
73              if ( (member.getYear()+member.getLength()) <
                   thisyear )
74              {
75                  overdue++;
76              }
77          }
78          return overdue;
79      }
80  }
```

8 Collections Example: ClubDemo.java

```
1
2  /**
3   * Demonstration of the functionalities of a Club
4   *
5   * @author Rachel Cardell-Oliver
6   * @version January 2013
7   */
8  public class ClubDemo
9  {
10
11     private Club club;
12
13     /**
14      * Constructor for objects of class ClubDemo
15      */
16     public ClubDemo()
17     {
18         club = new Club();
19     }
20
21     /**
22      * Add some members to the club, and then
23      * show how many there are.
24      * Further example calls could be added if more
            functionality
25      * is added to the Club class.
26      */
27     public void demo()
28     {
29         System.out.println("Create a club with 3 members"
               );
30         club.join(new Membership("David", 2, 2004, 5));
31         club.join(new Membership("Michael", 1, 2004, 10))
               ;
32         club.join(new Membership("Susan", 5, 2005, 5));
33         System.out.println("The club has " +
34                            club.numberOfMembers() +
35                            " members.");
36
37
38         // print out the current club
39         System.out.println();
40         System.out.println("List Current Membership");
```

```
41          club.listMembers();
42
43          //give everyone an extra year and re-print the
                current list
44          System.out.println();
45          System.out.println("Extend all memberships by 1
                year");
46          club.extendMembership(1);
47          club.listMembers();
48
49          // check how many expired memberships
50          System.out.println();
51          System.out.println("Check for expired memberships
                 in 2013");
52          System.out.println("For 2013 there are " + club.
                numOverdue(2013) + " overdue memberships");
53
54      }
55  }
```

9 Object Interation: NumberDisplay.java

```
1
2  /**
3   * The NumberDisplay class represents a digital number
        display that can hold values from zero to a given
        limit.
4   * The limit can be specified when creating the display.
5   * The values range from zero (inclusive) to limit-1.
6   * If used, for example, for the seconds on a digital
        clock, the limit would be 60, resulting in display
        values from 0 to 59.
7   * When incremented, the display automatically rolls over
        to zero when reaching the limit.
8   *
9   * @author Michael Koelling and David J. Barnes extended
        by (your name here)
10  * @version 2011.07.31 extended on (your date here)
11  */
12 public class NumberDisplay
13 {
14     private int limit;
15     private int value;
16
17     /**
18      * Constructor for objects of class NumberDisplay.
19      * Set the limit at which the display rolls over.
20      */
21     public NumberDisplay(int rollOverLimit)
22     {
23         limit = rollOverLimit;
24         value = 0;
25     }
26
27     /**
28      * Return the current value.
29      */
30     public int getValue()
31     {
32         return value;
33     }
34
35     /**
36      * Return the display value (that is, the current
            value as a two-digit
```

```
37      * String. If the value is less than ten, it will be
            padded with a leading zero).
38      */
39     public String getDisplayValue()
40     {
41         if(value < 10) {
42             return "0" + value;
43         }
44         else {
45             return "" + value;
46         }
47     }
48
49     /**
50      * Set the value of the display to the new specified
            value. If the new value is less than zero or over
             the limit, do nothing.
51      */
52     public void setValue(int replacementValue)
53     {
54         if((replacementValue >= 0) && (replacementValue <
                limit)) {
55             value = replacementValue;
56         }
57     }
58
59     /**
60      * Increment the display value by one, rolling over
            to zero if the limit is reached.
61      */
62     public void increment()
63     {
64         value = (value + 1) % limit;
65     }
66 }
```

10 Object Interation: ClockDisplay.java

```
1
2 /**
3  * The ClockDisplay class implements a digital clock
       display for a European-style 24 hour clock.
4  * The clock shows hours and minutes.
5  * The range of the clock is 00:00 (midnight) to 23:59 (
       one minute before midnight).
6  *
```

```
 7  * The clock display receives "ticks" (via the timeTick
       method) every minute and reacts by incrementing the
       display.
 8  * This is done in the usual clock fashion: the hour
       increments when the minutes roll over to zero.
 9  *
10  * @author Michael Koelling and David J. Barnes extended
       by (your name here)
11  * @version 2011.07.31 extended on (your date here)
12  */
13 public class ClockDisplay
14 {
15     private NumberDisplay hours;
16     private NumberDisplay minutes;
17     private String displayString;    // simulates the
           actual display
18
19     /**
20      * Constructor for ClockDisplay objects. This
            constructor creates a new clock set at 00:00.
21      */
22     public ClockDisplay()
23     {
24         hours = new NumberDisplay(24);
25         minutes = new NumberDisplay(60);
26
27         updateDisplay();
28     }
29
30     /**
31      * Constructor for ClockDisplay objects. This
            constructor creates a new clock set at the time
            specified by the parameters.
32      */
33     public ClockDisplay(int hour, int minute)
34     {
35         hours = new NumberDisplay(24);
36         minutes = new NumberDisplay(60);
37         setTime(hour, minute);
38     }
39
40
41     /**
42      * This method should get called once every minute -
            it makes the clock display go one minute forward.
43      */
```

```
44      public void timeTick()
45      {
46          minutes.increment();
47          if(minutes.getValue() == 0) {
48              // it just rolled over!
49              hours.increment();
50              if (hours.getValue() == 0) {
51                  dayofweek.increment();
52              }
53          }
54          updateDisplay();
55      }
56
57
58      /**
59       * Set the time of the display to the specified hour
             and minute.
60       */
61      public void setTime(int hour, int minute)
62      {
63          hours.setValue(hour);
64          minutes.setValue(minute);
65          updateDisplay();
66      }
67
68
69      /**
70       * Return the current time of this display in the
             format HH:MM.
71       */
72      public String getTime()
73      {
74          return displayString;
75      }
76
77      /**
78       * Update the internal string that represents the
             display.
79       */
80      private void updateDisplay()
81      {
82          displayString = hours.getDisplayValue() + ":" +
83                          minutes.getDisplayValue();
84      }
85
86  }
```

11 Example of HashMap use from the Zuul game

```
1  import java.util.HashMap;
2
3  /**
4   * This class is part of the "World of Zuul" application.
5   * "World of Zuul" is a very simple, text based adventure
        game.
6   *
7   * This class holds an enumeration of all command words
        known to the game.
8   * It is used to recognise commands as they are typed in.
9   *
10  * @author  Michael K lling and David J. Barnes
11  * @version 2011.08.09
12  */
13
14 public class CommandWords
15 {
16      // A mapping between a command word and the
            CommandWord
17      // associated with it.
18      private HashMap<String, CommandWord> validCommands;
19
20      /**
21       * Constructor - initialise the command words.
22       */
23      public CommandWords()
24      {
25          validCommands = new HashMap<String, CommandWord
                >();
26          validCommands.put("go", CommandWord.GO);
27          validCommands.put("help", CommandWord.HELP);
28          validCommands.put("quit", CommandWord.QUIT);
29      }
30
31      /**
32       * Find the CommandWord associated with a command
            word.
33       * @param commandWord The word to look up (as a
            string).
34       * @return The CommandWord correspondng to
            commandWord, or UNKNOWN
35       *          if it is not a valid command word.
36       */
```

```
37      public CommandWord getCommandWord(String commandWord)
38      {
39          CommandWord command = validCommands.get(
                commandWord);
40          if(command != null) {
41              return command;
42          }
43          else {
44              return CommandWord.UNKNOWN;
45          }
46      }
47
48      /**
49       * Check whether a given String is a valid command
              word.
50       * @return true if it is, false if it isn't.
51       */
52      public boolean isCommand(String aString)
53      {
54          return validCommands.containsKey(aString);
55      }
56
57      /**
58       * Print all valid commands to System.out.
59       */
60      public void showAll()
61      {
62          for(String command : validCommands.keySet()) {
63              System.out.print(command + "  ");
64          }
65          System.out.println();
66      }
67  }
```

12 Game of Life: Life.java

```
1  /**
2   * Java class for Conway's Game of Life
3   * @author PK, RCO
4   */
5
6  public class Life {
7
8      private boolean[][] map;          //grid for Life
9      private boolean[][] nextMap;      //grid for next
           generation
10     private int width;                //grid dimensions
11     private int height;
12
13
14      /**
15      * Random constructor for game of life
16      *
17      * @param width the x dimension of the grid
18      * @param height the y dimension of the grid
19      * @param probability is the chance that life exists
            in any cell
20      */
21     public Life(int width, int height, double probability
           ) {
22         map = new boolean[width][height];
23         nextMap = new boolean[width][height];
24         this.width = width;
25         this.height = height;
26         initializeMap(probability);
27     }
28
29
30     /**
31      * Life constructor for a given grid pattern
32      */
33      public Life(boolean[][] initial) {
34         map = initial;
35         width = map.length;
36         height = map[0].length;
37         nextMap = new boolean[width][height];
38     }
39
40     /**
```

```
41          * fully default Life constructor
42          */
43         public Life() {
44             this(150,150,0.1);
45         }
46
47         /**
48          * Construct a simple grid just for testing
49          */
50         public Life(boolean testonly) {
51             this.width = 10; //default magic numbers for
                   testing
52             this.height = 10; //use diff ones to test
53             map = new boolean[width][height];
54             simpleMap();  //set up map for testing
55             nextMap = new boolean[width][height];
56               //TODO nextMap is initialised by nextGeneration
                    method
57         }
58
59         /**
60          * public map init version just for testing
61          * TODO comment this out or delete
62          */
63         public void simpleMap() {
64            //clear map
65             for (int i=0; i<width; i++) {
66                 for (int j=0; j<height; j++) {
67                     map[i][j]=false;
68                 }
69             }
70             //add some life
71             map[4][5]=true;
72             map[5][5]=true;
73             map[6][5]=true;
74             map[width-1][height-1]=true; //boundary case
75         }
76
77         /**
78          * Fill a map with new life
79          * using prob as probability that life exists in any
                cell
80          */
81         public void initializeMap(double prob) {
82             for (int i=0; i<width; i++) {
83                 for (int j=0; j<height; j++) {
```

```
84                  if (Math.random() < prob) {
85                      map[i][j] = true;
86                  } else {
87                      map[i][j] = false;
88                  }
89              }
90          }
91      }
92
93
94      /**
95       * game of life current population counter
96       *
97       * @returns the number of live cells in map
98       */
99      public int population() {
100     int pop=0;
101          for (int i=0;i<width;i++) {
102             for (int j=0;j<height;j++) {
103                 if (map[i][j]) pop++;
104             }
105         }
106         return pop;
107     }
108
109     /**
110      * calculate the next generation of life and update
             map reference
111      */
112     public void nextGeneration() {
113         int n;
114         //calculate next map
115         for (int i=0;i<width;i++) {
116             for (int j=0;j<height;j++) {
117                 n = numNeighbours(i,j);
118                 if (n <= 1 || n >= 4) {nextMap[i][j] =
                        false;}
119                 else if (n == 2) {nextMap[i][j] = map[i][
                        j];}
120                 else if (n == 3) {nextMap[i][j] = true;}
121                 else assert true: "We should not get here
                        ";
122             }
123         }
124         //point map reference to the new generation
125         boolean[][] swap = map;
```

```
126             map = nextMap;
127             nextMap = swap;
128         }
129
130         /**
131          * count number of neighbours out of 8 adjacent
                squares
132          * adjacency assumed to wrap around grid at the edges
133          *
134          * @param i grid cell column parameter (width)
135          * @param j grid cell row parameter (height)
136          * @return number of neighbours found
137          */
138         //TODO this method just made public for test, make
               private helper later
139         public int numNeighbours(int i, int j) {
140             int n=0;
141             //unchecked exception for illegal grid cell refs
142             if ((i<0)||(i>=width)||(j<0)||(j>=height)) {
143               throw new IllegalArgumentException("
                      numNeighbours cell "
144                         +i+","+j + " out of bounds");
145             }
146
147             //count clockwise from NW corner
148             int north=(height+j-1)%height;
149             int south=(j+1)%height;
150             int east=(i+1)%width;
151             int west=(width+i-1)%width;
152             //north row
153             if (map[west][north]) n++;
154             if (map[i][north]) n++;
155             if (map[east][north]) n++;
156             //east neighbour
157             if (map[east][j]) n++;
158             //south row
159             if (map[east][south]) n++;
160             if (map[i][south]) n++;
161             if (map[west][south]) n++;
162             //west neighbour
163             if (map[west][j]) n++;
164             return n;
165         }
166
167 /*
168  * standard set of getter methods for all instance
```

```
     variables
 */
    public boolean[][] getMap() { return map; }

    public int getWidth() { return width; }

    public int getHeight() { return height; }

    /*
 * standard set of setter methods for all instance
     variables
 */
    public void setMap(boolean [][] map) {
        this.map=map;
    }

}
```

13 Game of Life: LifeViewer.java

```

/**
 * Class to visualise a game of life map
 *
 * @author PK
 * @version Revised 2013
 */

public class LifeViewer {
    private Life life;
    private int width;
    private int height;
    private SimpleCanvas c;
    private static int PAUSE = 10;

    private final static int CELL_SIZE = 4;

    private final static java.awt.Color BACK_COLOUR =
        java.awt.Color.white;
    private final static java.awt.Color CELL_COLOUR =
        java.awt.Color.blue;
    private final static java.awt.Color GRID_COLOUR =
        java.awt.Color.black;


    public LifeViewer(Life life) {
```

```
24
25          this.life = life;
26          width = life.getMap().length;
27          height = life.getMap()[0].length;
28
29          c = new SimpleCanvas("Life",width*CELL_SIZE+1,
30                                       height*CELL_SIZE+1,false)
                                           ;
31
32          display();
33      }
34
35      public LifeViewer() {
36          this(new Life());
37      }
38
39      private void display() {
40          erase();
41          drawCells();
42          //drawGrid(); looks better without gridlines for
                big grids
43          c.repaint();
44      }
45
46
47      private void erase() {
48          c.setForegroundColour(BACK_COLOUR);
49          for (int i=0; i<CELL_SIZE*height; i++) {
50              c.drawLine(0,i,CELL_SIZE*width,i);
51          }
52      }
53
54      private void drawCells() {
55          boolean[][] map = life.getMap();
56          c.setForegroundColour(CELL_COLOUR);
57          for (int i=0;i<width;i++) {
58              for (int j=0;j<height;j++) {
59                  if (map[i][j]) {
60                      for (int k=0;k<CELL_SIZE;k++){
61                          c.drawLine(i*CELL_SIZE+k,j*
                                CELL_SIZE,
62                                i*CELL_SIZE+k,(j+1)*CELL_SIZE)
                                    ;
63                      }
64                  }
65              }
```

```
66          }
67      }
68
69
70      private void drawGrid() {
71          c.setForegroundColour(GRID_COLOUR);
72
73          for (int i=0;i<=width;i++) {
74              c.drawLine(i*CELL_SIZE,0,i*CELL_SIZE,height*
                    CELL_SIZE);
75          }
76          for (int j=0;j<=height;j++) {
77              c.drawLine(0,j*CELL_SIZE,width*CELL_SIZE,j*
                    CELL_SIZE);
78          }
79      }
80
81
82      public void animate(int n) {
83          for (int i=0;i<n;i++) {
84              life.nextGeneration();
85              display();
86              c.wait(PAUSE);
87          }
88      }
89
90      public void step() {
91          life.nextGeneration();
92          display();
93      }
94
95      public void randomize(double probability) {
96          life.initializeMap(probability);
97          display();
98      }
99
100     public void glider() {
101         life.setMap(Examples.glider(width, height));
102         display();
103     }
104
105     public void blink() {
106         life.setMap(Examples.blink(width, height));
107         display();
108     }
109
```

```
110     public void toad() {
111         life.setMap(Examples.toad(width, height));
112         display();
113     }
114
115     public void lightWeightSpaceShip() {
116         life.setMap(Examples.lightWeightSpaceShip(width,
                height));
117         display();
118     }
119
120     public void dieHard() {
121         life.setMap(Examples.dieHard(width, height));
122         display();
123     }
124
125     public void acorn() {
126         life.setMap(Examples.acorn(width, height));
127         display();
128     }
129
130     public void gosper() {
131         life.setMap(Examples.gosper(width, height));
132         display();
133     }
134
135     public void block1() {
136         life.setMap(Examples.block1(width, height));
137         display();
138     }
139
140     public void block2() {
141         life.setMap(Examples.block2(width, height));
142         display();
143     }
144
145 }
```

14 Sorting Algorithms: Sorter.java

```
1
2  /**
3   * Utility class of example Sorting algorithms
4   *
5   * @author Rachel Cardell-Oliver
6   * @version May 2011 revised May 2012 Checkstyle checked
        May 2013
7   */
8  public class Sorter
9  {
10
11     /**
12      * bubble sort method
13      * @param  a   array of values to sort in place
14      */
15     public static void bubbleSort(int[] a)
16     {
17         for (int pass = 1; pass < a.length; pass++) {
18             for (int j = 0; j < a.length - pass; j++) {
19                 if (a[j] > a[j + 1]) {
20                     int swap = a[j + 1];
21                     a[j + 1] = a[j];
22                     a[j] = swap;
23                 }
24             }
25         }
26     }
27
28     /**
29      * insertion sort method
30      * @param  a   array of values to sort in place
31      */
32     public static void insertionSort(int[] a)
33     {
34         for (int pass = 1; pass < a.length; pass++) {
35             int tmp = a[pass]; //value to be inserted
36             int pos = pass - 1; //position of sorted so
                   far elements
37             while (pos >= 0 && a[pos] > tmp) {
38                 a[pos + 1] = a[pos]; //move up out of
                       order elements
39                 pos--;
40             }
```

```
41              a[pos + 1] = tmp; //put val in the right
                    place
42          }
43      }
44
45      /**
46       * Sort the elements of the array into increasing
             order by
47       * selecting the smallest element in the array and
             swapping it
48       * with the element in pos 0.
49       * Then calculate the position of the smallest
             element
50       * in the array starting from position 1, and swap
             the element
51       * there into position 1, and so on for each position
             .
52       *
53       * @param a integer array to be sorted in place
54       */
55
56      public static void selectionSort(int[] a)
57      {
58          int swappos;
59          int min;
60          for (int i = 0; i < a.length - 1; i++) {
61              min = a[i];
62              swappos = i;
63              for (int j = i + 1; j < a.length; j++) { //
                    find smallest in rest
64                  if (a[j] < min) {
65                      min = a[j];
66                      swappos = j;
67                  } // now swappos is the smallest element
                        in i+1 to length
68              }
69
70              if (i != swappos) {  // swap i and swappos if
                     necessary
71                  int temp = a[i];
72                  a[i] = a[swappos]; //ie min
73                  a[swappos] = temp;
74              }
75          }
76      }
77
```

```
78      /**
79       * partition an array around the fence a[start]
80       * @param a array to be partitioned
81       * @param start first position of array fragment to
             sort
82       * @param finish last position of array fragment to
             sort
83       * @return right of partition
84       */
85      private static int partition(int[] a, int start, int
            finish)
86      {
87          int fence = a[start];
88          int left = start + 1;
89          int right = finish;
90          while (right >= left) {
91              while (left <= right && a[left] <= fence) {
92                  left++;
93              }
94              while (right >= left && a[right] >= fence) {
95                  right--;
96              }
97              if (right > left) {
98                  int swap = a[left];
99                  a[left] = a[right];
100                 a[right] = swap;
101             }
102         }
103         a[start] = a[right];
104         a[right] = fence;
105         return right;
106     }
107
108     /**
109      * quicksort helper method
110      * @param a array to be sorted
111      * @param start first position of array fragment to
             sort
112      * @param finish last position of array fragment to
             sort
113      */
114     private static void quickSort(int[] a, int start, int
            finish)
115     {
116         if (finish - start > 0) {
117             int fencePosition = partition(a,start,finish)
```

```
                ;
118             quickSort(a, start, fencePosition - 1);
119             quickSort(a, fencePosition + 1, finish);
120         }
121     }
122
123     /**
124      * quicksort recursive sorting algorithm
125      * @param a array to be sorted
126      */
127     public static void quickSort(int[] a)
128     {
129         quickSort(a, 0, a.length - 1);
130     }
131
132     /**
133      * linear search for key in array of data
134      * @param data array of sorted ints to search
135      * @param key int to be searched for
136      * @return boolean true if value found and false
           otherwise
137      */
138     public static boolean linerarSearch(int[] data, int
           key)
139     {
140         for (int di : data) {
141             if (di == key) {
142                 return true;
143             }
144         }
145         return false;
146     }
147
148     /**
149      * binary search for value in a *sorted* array
150      * @param data array of sorted ints to search
151      * @param key int to be searched for
152      * @return boolean true if value found and false
           otherwise
153      */
154     public static boolean binarySearch(int[] data, int
           key)
155     {
156         int left = 0;
157         int right = data.length - 1;
158         while (right - left > 1) {
```

```
159             int middle = (left + right) / 2;
160             if (data[middle] >= key) {
161                 right = middle;
162             } else {
163                 left = middle;
164             }
165         }
166         return (data[left] == key || data[right] == key);
167     }
168 }
```