The German University in Cairo
Dpt. for Media Engineering and Technology
Prof. Dr. Slim Abdennadher, Assoc. Prof. Dr. Georg Jung

June 23, 2012

CSEN 202: Introduction to Computer Programming

Spring 2012

Final exam

Model Solutions

Instructions. Please read carefully before proceeding.

(a) The duration of this exam is **180 minutes**.
(b) Non-programmable calculators are allowed.
(c) No books or other aids are permitted for this test.
(d) This exam booklet contains a total of **11 pages**, including this one.

Exercise 1 Arrays, loops, conditionals (10 Marks)
Two-dimensional array evaluation

(a) Given a two-dimensional, possibly ragged, array of booleans, write a method that evaluates the array such that the value of every row is the *conjunction* (logical AND) of all values in the row, and the value of the complete array is the *disjunction* (logical OR) of all row values.[1] (10 Marks)

Solution:

A correct method signature and a correct return-value will earn one Mark each. Four Marks can be reached in each dimension of the array evaluation.

```
public static boolean evaluate(boolean[][] a) {
  boolean formula = false;
  for (int i = 0; i < a.length; i++) {
    boolean clause = true;
    for (int j = 0; j < a[i].length; j++)
      clause = clause && a[i][j];
    formula = formula || clause;
  }
  return formula;
}
```

[1] This way of evaluating the array corresponds to the disjunctive normal form (DNF) which reappears in a completely different approach again in exercise 4

Exercise 2 Command line arguments (8 Marks)
Digit 2 English

(a) Write a program (complete with class and main-method) that takes a series of digits as command-line arguments and prints the English name of each digit. You can assume that all arguments are integer numerals. (8 Marks)

Hint. You may use the static method `parseInt(String s)` in the `Integer` class that takes a string and returns an integer.

Two examples.

First run console input:
```
java DigitToEnglish 5 42 7 9
```
Output:
```
five
sorry, 42 is not a digit!
seven
nine
```
Second run console input:
```
java DigitToEnglish 6 4 7
```
Output:
```
six
four
seven
```

Solution:

```
package exams;

public class Digit2English {

  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
      System.out.println(digit2English(Integer.parseInt(args[i])));
  }

  public static String digit2English(int x) {
    switch (x) {
      case 0: return "zero";
      case 1: return "one";
      case 2: return "two";
      case 3: return "three";
      case 4: return "four";
      case 5: return "five";
      case 6: return "six";
      case 7: return "seven";
      case 8: return "eight";
      case 9: return "nine";
      default: return "sorry,␣" + x + "␣is␣not␣a␣digit!";
    }
  }
}
```

Exercise 3 Recursion (6 Marks)
Lucas numbers

The n^{th} *Lucas number* is defined by the equation

$$L_n = \begin{cases} 2 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \text{ and} \\ L_{n-1} + L_{n-2} & \text{if } n > 1. \end{cases}$$

(a) Write a *recursive* method `lucas(int n)` that returns the `n`$^{\text{th}}$ Lucas number. (6 Marks)

(b) Use only the conditional operator and no local variables in your method. **(3 Marks)**

Solution:

A correct method signature and a correct return-value will earn one Mark each. There is one Mark for each base case and two Marks for the recursion case.

```
public static long lucas(int n) {
  return (n == 0) ? 2 : (n == 1) ? 1 : lucas(n - 1) + lucas(n - 2);
}
```

Exercise 4 Classes and Objects (21 Marks)
Disjunctive normal form

The *disjunctive normal form* (DNF) is a format for Boolean expressions that is constructed with the following schema:

- There is a number of *variables*:
$$x, \quad y, \quad z, \quad \ldots$$
- A single variable or a negated single variable is called a *literal*:
$$x, \quad y', \quad z', \quad \ldots$$
- The conjunction (*i. e.*, connection with an AND-operator) of a several literals is called a *clause*:
$$x \cdot y' \cdot z, \quad x' \cdot y \cdot z', \quad \ldots$$
- The disjunction (*i. e.*, connection with an OR-operator) of a several clauses is called a *formula in DNF*:
$$x \cdot y' \cdot z + x' \cdot y \cdot z' + x' \cdot y' \cdot z$$

In this exercise you will implement four classes that together represent a formula in DNF. Specifically, your classes must work with the following tester main method (Listing 1), which you should try to fully understand before you start solving this exercise:

Listing 1: Tester class for building and evaluating a DNF formula

```
public class FormulaTester {

  public static void main(String[] args) {
    Variable x = new Variable('x');
    Variable y = new Variable('y');

    Literal l1 = new Literal(x);
    Literal l2 = new Literal(x);
    l2.negate();
    Literal l3 = new Literal(y);
    Literal l4 = new Literal(y);
    l4.negate();

    Clause c1 = new Clause(new Literal[] { l1, l4 });
    Clause c2 = new Clause(new Literal[] { l2, l3 });

    Formula f = new Formula(new Clause[] { c1, c2 });

    x.setFalse();
    y.setTrue();

    System.out.println(f.evaluate());
  }
}
```

(a) Implement a class `Variable`, which contains a character (the variable name) and a boolean value (the truth value of the variable). The class must provide a constructor that takes a character (the name) and three methods: `setTrue()` and `setFalse()`, which set the variable to **`true`** and **`false`** respectively, and `evaluate()` that returns the current value as a boolean. (5 Marks)

Solution:

There is one Mark for the class header and member variables, and one for each method except for `toString`, which is not required to receive full Marks.

```
public class Variable {
  char name;
  boolean valuation;

  public Variable (char name) {
    this.name = name;
  }

  public void setTrue() {
    valuation = true;
  }

  public void setFalse() {
    valuation = false;
  }

  public boolean evaluate() {
    return valuation;
  }

  public String toString() {
    return "" + name;
  }
}
```

(b) Implement a class `Literal`, which contains a variable and a boolean value that determines if the literal is negated or not. The class must provide a constructor that takes an object of type `Variable` and two methods: `negate()` which flips the negation from positive to negative or vice versa, and `evaluate`, which returns the truth value of the literal. (4 Marks)

Solution:

There is one Mark for the class header and member variables, and one for each method except for `toString`, which is not required to receive full Marks.

```
public class Literal {
  Variable var;
  boolean negative;

  public Literal (Variable var) {
    this.var = var;
  }

  public void negate() {
    negative = !negative;
  }

  public boolean evaluate() {
    return negative ^ var.evaluate();
  }

  public String toString() {
    return var + (negative ? "'" : "");
  }
}
```

(c) Implement a class `Clause`, which contains an array of objects of type `Literal`. The class must provide a constructor that takes an array `Literal[]` and a method `evaluate()` that returns the truth value of the clause. (6 Marks)

Solution:

There is one Mark for the class header and member variable, and one for the constructor. The method `evaluate()` receives four Marks. `toString` is not required to receive full Marks.

```
public class Clause {
  Literal[] literals;

  public Clause (Literal[] literals) {
    this.literals = literals;
  }

  public boolean evaluate() {
    boolean result = true;
    for (int i = 0; i < literals.length; i++)
      result = result && literals[i].evaluate();
    return result;
  }

  public String toString() {
    String result = "";
    for (int i = 0; i < literals.length - 1; i++)
      result += literals[i] + "*";
    result += literals[literals.length - 1];
    return result;
  }
}
```

(d) Finally, implement a class `Formula`, which contains an array of objects of type `Clause`. The class must provide a constructor that takes an array `Clause[]` and a method `evaluate()` that returns the truth value of the formula. (6 Marks)

Solution:

There is one Mark for the class header and member variable, and one for the constructor. The method `evaluate()` receives four Marks. `toString` is not required to receive full Marks.

```
public class Formula {
  Clause[] clauses;

  public Formula (Clause[] clauses) {
    this.clauses = clauses;
  }

  public boolean evaluate() {
    boolean result = false;
    for (int i = 0; i < clauses.length; i++)
      result = result || clauses[i].evaluate();
    return result;
  }

  public String toString() {
    String result = "";
    for (int i = 0; i < clauses.length - 1; i++)
      result += clauses[i] + "␣+␣";
    result += clauses[clauses.length - 1];
    return result;
  }
}
```

Exercise 5 Object types (16 Marks)
Tangled references

Consider the class `Point2D` as given in the following code (Listing 2):

Listing 2: Class `Point2D` for two-dimensional points

```
public class Point2D {
  static int nextID = 0;
  int iD;
  double x;
  double y;

  public Point2D (double x, double y) {
    this.iD = nextID++;
    this.x = x;
    this.y = y;
  }

  public void move(double xOffset, double yOffset) {
    x += xOffset;
    y += yOffset;
  }

  public void display() {
    System.out.println("Point␣" + iD + ":\t␣x=␣" + x + ",\t␣y=␣" + y);
  }

  public double getX() {
    return x;
  }

  public double getY() {
    return y;
  }
}
```

Further, consider a tester class which contains a method `swap(Point2d p, Point2D q)` and the following main method (Listing 3):

Listing 3: Main method of a tester class for `Point2D`

```
  public static void main(String[] args) {
    Point2D a = new Point2D(5, 5);
    Point2D b = a;
    Point2D c = new Point2D(0, 0);
    Point2D d;

    b.move(-10, 10);

    swap(b, c);
    d = new Point2D(10, 10);

    a.display();
    b.display();
    c.display();
    d.display();
  }
```

For each of the following different implementations of `swap(Point2d p, Point2D q)` give the *exact output* of the main method. Trace carefully, the output is *not* obvious!

(a) First implementation of `swap`: (4 Marks)

```
public static void swap(Point2D p, Point2D q) {
  Point2D r = p;
  p = q;
  q = r;
}
```

Output:

Solution:

```
Point 0: x= -5.0, y= 15.0
Point 0: x= -5.0, y= 15.0
Point 1: x= 0.0, y= 0.0
Point 2: x= 10.0, y= 10.0
```

(b) Second implementation of `swap`: (4 Marks)

```
public static void swap(Point2D p, Point2D q) {
  Point2D r = new Point2D(p.getX(), p.getY());
  p = new Point2D(q.getX(), q.getY());
  q = r;
}
```

Output:

Solution:

```
Point 0: x= -5.0, y= 15.0
Point 0: x= -5.0, y= 15.0
Point 1: x= 0.0, y= 0.0
Point 4: x= 10.0, y= 10.0
```

(c) Third implementation of `swap`: (4 Marks)

```
public static void swap(Point2D p, Point2D q) {
  double xp = p.getX(), yp = p.getY();
  double xq = q.getX(), yq = q.getY();
  p.move(-xp + xq, -yp + yq);
  q.move(-xq + xp, -yq + yp);
}
```

Output:

Solution:

```
Point 0: x= 0.0, y= 0.0
Point 0: x= 0.0, y= 0.0
Point 1: x= -5.0, y= 15.0
Point 2: x= 10.0, y= 10.0
```

(d) Fourth implementation of `swap`: (4 Marks)

```
public static void swap(Point2D p, Point2D q) {
  Point2D r = new Point2D(p.getX() - q.getX(), p.getY() - q.getY());
  p.move(-r.getX(), -r.getY());
  q.move(r.getX(), r.getY());
}
```

Output:

Solution:

```
Point 0: x= 0.0, y= 0.0
Point 0: x= 0.0, y= 0.0
```

```
Point 1: x= -5.0, y= 15.0
Point 3: x= 10.0, y= 10.0
```

Bonus Exercise 6 Inheritance **Three-dimensional point** (5 Marks)

(a) Create a class `Point3D` as an extension of the class `Point2D` in Listing 2 on page 8. Use **`super`** wherever possible. **(5 Marks)**

Solution:

```
public class Point3D extends Point2D {
  private double z;

  public Point3D (double x, double y, double z) {
    super(x, y);
    this.z = z;
  }

  public void move(double xOffset, double yOffset, double zOffset) {
    super.move(xOffset, yOffset);
    z += zOffset;
  }

  public void display() {
    System.out.println("Point␣" + iD + ":\t␣x=␣" + x + ",\t␣y=␣" + y
        + ",\t␣z=␣" + z);
  }

  public double getZ() {
    return z;
  }
}
```