Examination for TDA540
Object-Oriented Programming
(Objektorienterad Programmering)

Institutionen för Datavetenskap
CTH HT-18, TDA540

Day: 2019-01-19, *Time:* 14.00-18.00

Course responsible:	Jesper Cockx
Examinator:	Krasimir Angelov
Contact:	+46 729 74 49 08, +46 31 772 10 19, +46 733056925
Result:	Will be available via Ladok
Grade boundaries:	3:a 24 points 4:a 36 points 5:a 48 points max 60 points
Numbers in brackets:	Maximal number of points for each question
Allowed material:	Cay Horstmann: *Java for everyone* (2nd edition) or Jan Skansholm: *Java direkt med Swing* Markings and small footnotes are allowed. You can also use an English-Swedish dictionary if needed.
Please take note:	Write clearly and make sure to hand in your solutions in the right place. Start each question on a new piece of paper. Do not write on the back side of the paper.
Additional remarks:	The exam consists of 6 questions. Questions are not ordered by difficulty. Start by reading the whole exam before you start writing. All programs should be structured and easy to read. Don't forget to use indentation! For the correction of program code, logical flaws are considered more serious than minor syntax errors.

Good luck!

Question 1 (18 points)

Take a look at the two classes `Robot` and `ChattyRobot` below[1]:

```
1   class Robot {
2       public int position;
3       public boolean goingRight;
4
5       public Robot() {
6           position = 0;
7           goingRight = true;
8       }
9
10      public void move() {
11          if (goingRight)
12              position++;
13          else
14              position--;
15      }
16
17      public void move(int steps) {
18          for (int i = 0; i < steps; i++) {
19              move();
20          }
21      }
22
23      public void turn() {
24        goingRight = !goingRight;
25      }
26
27      public static void main(String[] args) {
28          Robot robot = new ChattyRobot();
29          int i = 0;
30          while (Math.abs(robot.position) < 3) {
31              robot.move(i);
32              robot.turn();
33              i++;
34          }
35      }
36  }
37
38  class ChattyRobot extends Robot {
39      public void move(int steps) {
40          int oldPos = this.position;
41          super.move(steps);
42          int newPos = this.position;
43          System.out.println("Moved from " + oldPos + " to " + newPos);
44      }
45  }
```

a) What is the meaning of the keyword **`public`** in line number 2? Would the code still work if we replace **`public`** with **`private`**? If not, indicate the line where an error would occur. (3 points)

Answer: *The keyword **`public`** on this line means that the attribute **`position`** of the **`Robot`** class is accessible from any other part of the program, and even from a different package. If the keyword is replaced by **`private`**, there would be an error on lines 40 and 42 as the **`position`** attribute would no longer be accessible from the **`ChattyRobot`** class.*

[1] Line numbers are not part of the program.

b) List all *method calls* in this program (not method declarations). For each method call, give:

- The line number
- The class in which the method is declared
- The signature of the method (including the return type and the arguments)
- Whether the method is static or not

(3 points)

Answer:

- *`move()` on line 19, which is a call to the non-static method declared in class `Robot` with signature `public void move()`.*
- *`Math.abs(robot.position)` on line 30, which is a call to the static method declared in class `Math` with signature `public int abs(int x)`*
- *`move(i)` on line 31, which is a call to the non-static method declared in class `ChattyRobot` with signature `public void move(int steps)`.*
- *`turn()` on line 32, which is a call to the non-static method declared in class `Robot` with signature `public void turn()`.*
- *`super.move(steps)` on line 41, which is a call to the non-static method declared in class `Robot` with signature `public void move(int steps)`*
- *`System.out.println(...)` on line 43, which is a call to the non-static method declared in class `PrintStream` with signature `public void println(String x)`*

c) Indicate one example of *method overloading* and one example of *method overriding* in this program by giving the line numbers. (3 points)

Answer: *Method overloading: the two methods called `move` on lines 10 and 17 are overloaded. Method overriding: the method `move` on line 39 overrides the method `move` on line 17.*

d) What text is printed when we run the main method of this program? (6 points)

Answer:
Moved from 0 to 0
Moved from 0 to -1
Moved from -1 to 1
Moved from 1 to -2
Moved from -2 to 2
Moved from 2 to -3

e) Change the `ChattyRobot` class so the program also prints "`turned around`" each time the `turn()` method is called. You should not change anything to the base class. (3 points)

Answer: *Add the following method to the body of the `ChattyRobot` class:*

```
1  public void turn() {
2    super.turn();
3    System.out.println("Turned around.");
4  }
```

Question 2 (4 points)

1. What is printed out when we run the following Java program? (2 points)

```
boolean found = true;
boolean past = false;
found = past;
past = true;
System.out.println(found);
System.out.println(past);
```

Answer:
false
true

2. What is printed out when we run the following Java program? (2 points)

```
double[] list = {0.3, 0.5, 1.3, 7.8, 9};
double[] list2 = {9, 8.7, 3.34, 3.2};
list = list2;
list2[1] = 1.56;
System.out.println(list[1]);
System.out.println(list2[0]);
```

Answer:
1.56
9

Question 3 (16 points)

A **magic square** is an n by n grid containing all numbers from 1 to n^2 such that the numbers on each row, column, and diagonal all add up to the same constant, called the **magic constant**. For example, the following squares are magic squares with magic constants 15 and 65:

2	7	6
9	5	1
4	3	8

21	3	4	12	25
15	17	6	19	8
10	24	13	2	16
18	7	20	9	11
1	14	22	23	5

Your task is to implement a method `boolean isMagicSquare(int[][] x)` that checks if the given 2-dimensional array is a magic square. You may assume that the given 2-dimensional array is square (so `x.length == x[i].length` for `i`$= 0, \ldots, n-1$).

Answer:

```
public boolean isMagicSquare(int[][] x) {
        int n = x.length;
        int magicNumber = 0;
        for (int i = 0; i < n; i++) magicNumber += x[0][i];
        for (int i = 0; i < n; i++) {
            int rowSum = 0;
            int colSum = 0;
            for (int j = 0; j < n; j++) {
                rowSum += x[i][j];
```

```
10                  colSum += x[j][i];
11              }
12              if (rowSum != magicNumber) return false;
13              if (colSum != magicNumber) return false;
14          }
15          int diag1 = 0;
16          int diag2 = 0;
17          for (int i = 0; i < n; i++) {
18              diag1 += x[i][i];
19              diag2 += x[i][n-1-i];
20          }
21          if (diag1 != magicNumber) return false;
22          if (diag2 != magicNumber) return false;
23          return true;
24      }
```

Question 4

(6 points)

Take a look at the Java code below. The code in the `main` method contains 3 errors. Give for each error its line number and indicate whether it is a compile-time error or a run-time error.

```
1   class Cls {
2       static int x;
3       int y;
4   }
5
6   public class SpotTheErrors {
7       public static void main(String[] args) {
8           Cls obj = new Cls();
9           Cls.x = Cls.y;
10          Cls.x = obj.y;
11          Cls.y = Cls.x;
12          obj.y = Cls.x;
13          obj.x = obj.y;
14          obj.y = obj.x;
15          obj   = null;
16          System.out.println(obj.y);
17      }
18  }
```

Answer:

Line 9: compile-time error
Line 11: compile-time error
Line 16: run-time error

Question 5

(8 points)

Implement a class `Student` that can be used for representing students at the university. Each student has a first and last name, a study program, a study year, a student number, and an average grade (as a percentage). The class should contain attributes, getters and setters, and at least one constructor. It should also override the 'toString' method so it prints all this data in a readable format. Finally, it should also implement the `Comparable` interface (given below). The `compareTo` method should compare two students according to their student number: it should return -1 / 0 / $+1$ if the student number of this student is less than / equal to / greater than the student number of the other student.

```java
public interface Comparable<T> {
    int compareTo(T var1);
}
```

Answer:

```java
public class Student implements Comparable<Student> {
    private String firstName, lastName, studyProgram;
    private int studyYear;
    final int studentNumber;
    private double gradeAverage;

    public Student(String firstName, String lastName, int studentNumber) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.studentNumber = studentNumber;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getStudyProgram() {
        return studyProgram;
    }

    public void setStudyProgram(String studyProgram) {
        this.studyProgram = studyProgram;
    }

    public int getStudyYear() {
        return studyYear;
    }

    public void setStudyYear(int studyYear) {
        this.studyYear = studyYear;
    }

    public int getStudentNumber() {
        return studentNumber;
    }

    public double getGradeAverage() {
        return gradeAverage;
    }

    public void setGradeAverage(double gradeAverage) {
        this.gradeAverage = gradeAverage;
```

```
55      }
56
57      public String toString() {
58          return String.format("Student %s %s (%d)\nYear %d of %s\nAverage grade: %f",
59                  firstName, lastName, studentNumber, studyYear, studyProgram, gradeAverage);
60      }
61
62      public int compareTo(Student o) {
63          if (this.studentNumber < o.studentNumber)
64              return -1;
65          else if (this.studentNumber > o.studentNumber) {
66              return 1;
67          } else {
68              return 0;
69          }
70      }
71  }
```

Question 6

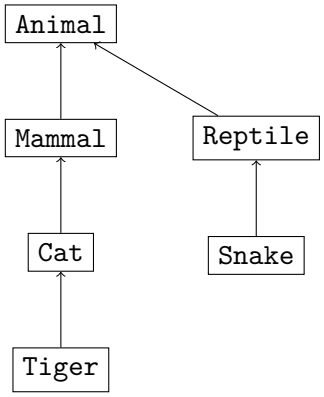
(8 points)

Take a look at the following interfaces and classes that model various kinds of animals.

```
1   interface Animal {
2     int numberOfLegs();
3   }
4
5   interface Mammal extends Animal { }
6
7   interface Reptile extends Animal { }
8
9   class Cat implements Mammal {
10      public int numberOfLegs() {
11          return 4;
12      }
13  }
14
15  class Tiger extends Cat { }
16
17  class Snake implements Reptile {
18      public int numberOfLegs() {
19          return 0;
20      }
21  }
```

1. Draw a class diagram where each type X in this hierarchy is represented by a node, and draw an arrow from X to Y if Y is a direct supertype of X. (3 points)

   Answer:



2. The `main` method below contains five type errors. Indicate the line number of each incorrect statement and explain in one sentence why it is wrong. (5 points)

```
1   public class Animals {
2       public static void main(String[] args) {
3           Animal animal1, animal2, animal3;
4           Mammal mammal1;
5           Cat cat1;
6           Snake snake1;
7           Tiger tiger1;
8           List<Animal> list1, list2;
9
10          animal1 = new Tiger();
11          snake1  = new Animal();
12          animal2 = new Animal();
13          cat1    = new Tiger();
14          mammal1 = new Snake();
15          tiger1  = mammal1;
16          animal3 = cat1;
17          list1   = new ArrayList<Animal>();
18          list2   = new ArrayList<Reptile>();
19      }
20  }
```

Answer:

1. *Line 11: Interfaces cannot have constructors, thus it is impossible to create a new object of type Animal by writing* `new Animal()`*. Even if this would work, the code would still not be correct because Animal is not a subtype of the type Snake of variable snake1.*

2. *Line 12: Interfaces cannot have constructors, thus it is impossible to create a new object of type Animal by writing* `new Animal()`*.*

3. *Line 14: The type Snake of* `new Snake()` *is not a subtype of the type Mammal of the variable mammal1.*

4. *Line 15: The type Mammal of* `mammal1` *is not a subtype of the type Tiger of variable tiger1.*

5. *Line 18: Subtyping does not propagate through parametrized classes, so even though Reptile is a subtype of Animal,* `ArrayList<Reptile>` *is not a subtype of* `ArrayList<Animal>`