Data och Informationsteknik / Computer Science and Engineering
Chalmers University of Technology

Göteborg 13 March 2020

COMPUTER PROGRAMMING part B TIN213

Date: 13 March 2020 Time: 08.30-11.30 Place: SB

Course responsible:	Anton Ekblad, tel. 070 7579 070 Will visit hall at 09.00 and 10.30
Examiner:	Krasimir Angelov
Allowed aids:	Skansholm, *Java Direkt med Swing* **or** Bravaco, Simonson, *Java Programming: From the Ground Up* (Underlinings and light annotations are permitted.) No calculators are permitted.
Grading scale:	Maximum total 30 points For this exam the following grades will be given: 3: 15 points, 4: 20 points, 5: 25 points
Exam review:	date **Fri. April 17th, 11.45 - 13.15** room **EDIT Analysen**

- Read through the whole exam before answering any questions.
- Start each new question on a new page.
- Write your anonymous code and the question number on each page.
- You may write your answers in English or Swedish.
- Write clearly. Unreadable = wrong!
- Try to answer all the questions. Partial answers can still get partial credit. There are 4 questions.
- Points may be deducted for solutions which are unnecessarily complex or uses poor programming practices.
- A quick reference guide to Java is included, starting on page 7.

Good luck!

1 Reading Code

Consider the following code:

```
1  public class Question1 {
2      public static void go(String task) {
3          if(task.equals("task a")) { new Bepa().doTheThing(100); }
4          if(task.equals("task b")) { new Bepa().doTheThing("the thing"); }
5          if(task.equals("task c")) { new Cykel().printValuePlus(5); }
6      }
7  }
8
9  abstract class Apa {
10     protected int value = 0;
11     public Apa(int value) { this.value = value; }
12     public abstract int getValue();
13     public void doTheThing(String arg) {
14         System.out.println("arg = " + arg);
15     }
16 }
17
18 class Bepa extends Apa {
19     public Bepa() { super(42); }
20     public int getValue() {
21         return super.value;
22     }
23     public void doTheThing(String arg) {
24         System.out.println("ignoring arg");
25     }
26     public void doTheThing(Object arg) {
27         super.doTheThing(arg.toString());
28     }
29 }
30
31 class Cykel extends Bepa {
32     private int value = 1000;
33     public void printValuePlus(int extra) {
34         System.out.println(this.getValue() + extra);
35     }
36 }
```

Answer the following questions about the above code. Motivate your answers with no more than *three* sentences per question.

(a) What does the program print if we call `Question1.go("task a")`? (2 points)

(b) What does the program print if we call `Question1.go("task b")`? (2 points)

(c) What does the program print if we call `Question1.go("task c")`? (2 points)

(d) If we change line 9 to `interface Apa` and line 18 to `class Bepa implements Apa`, would the program still compile? (2 points)

8 points total

2 Pokemon Battle

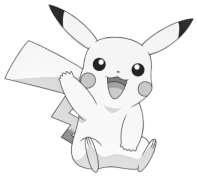
Pokemon are small monsters who fight each other in rock-paper-scissors-like battles. Every pokemon has exactly one *type*: fire, water, or plant.

When two pokemon fight:

- If the two pokemon have the same type, then either can win.
- If a plant type pokemon fights a water type pokemon, the *plant* pokemon wins.
- If a fire type pokemon fights a plant type pokemon, the *fire* pokemon wins.
- If a water type pokemon fights a fire type pokemon, the *water* pokemon wins.

We use the `Pokemon` interface to represent a pokemon, and the `PokemonType` enum[1] to represent a pokemon's type:

```
enum PokemonType {Fire, Water, Plant};
interface Pokemon {
    PokemonType getType();
    String getName();
}
```



(a) **Your task:** write an abstract class `FighterPokemon` which implements the `Pokemon` interface. The `FighterPokemon` class must have a single method `public boolean canDefeat(Pokemon opponent)` which returns `true` if the pokemon represented by `this` *could possibly* defeat the pokemon represented by `opponent`, and otherwise returns `false`.

That is, the method should return `true` if and only if:

- the two pokemon have the same type (in which case either *could* win), or
- the `this` pokemon has a type which beats the type of the `opponent` pokemon.

(3 points)

(b) In the previous question, the abstract `FighterPokemon` class you were told to write implements the `Pokemon` interface, but does not contain any of the interface's methods (`getType` and `getName`).

Your task: explain why this does not produce a compilation error, using at most *two* sentences.

(2 points)

(c) Some pokemon do not only have a type, but also a *level*. A pokemon's level represents its combat strength: if two pokemon *of the same type* fight, the one with the higher level wins. If two pokemon of *different* types fight, the rock-paper-scissors rule still applies (i.e. water beats fire, etc.).

Your task: write an abstract class `LeveledPokemon`, which extends `FighterPokemon` and has the following members:

- A constructor which takes the initial level of the pokemon as its only argument.
- A method `public int getLevel()` which returns the level of the pokemon.
- A method `public boolean canDefeat(LeveledPokemon opponent)`, which returns true if and only if the `this` pokemon could possibly defeat the `opponent` pokemon, *taking the both the types and levels of the two pokemon into account.*

(3 points)

(**Question 2 continues on next page**)

[1] See the Java reference starting on page 7 if you are unsure about how to use enums.

(d) **Your task:** implement a non-abstract class `MyPokemon` which extends `LeveledPokemon` and implements the full `Pokemon` interface. The class must have a constructor `public MyPokemon(int level)` which allows the caller to set the initial level of the pokemon. You may choose the type and name of the pokemon (as returned by `getType` and `getName` respectively) yourself.
(2 points)

(e) While it is optional to put `@Override` annotations on overridden methods, putting an `@Override` annotation on a method which is *not* overridden is a compilation error.

Your task: which methods in the `FighterPokemon`, `LeveledPokemon` and `MyPokemon` classes *may* have `@Override` annotations without causing compilation errors? You do *not* need to motivate your answer.
(2 points)

For full marks, your solutions to tasks *(a)*, *(c)* and *(d)* must observe the DRY (Don't Repeat Yourself) principle. You should also make all instance variables *private*.

12 points total

3 Generics

Please answer the following questions, using no more than *two* sentences for each question.

(a) When should you use a `List<String>` instead of an array of `String`s (i.e. `String[]`)? (1 points)

(b) When should you use an array of `String`s instead of a `List<String>`? (1 points)

(c) What will the following code print? (1 points)

```
Map<String, Integer> levels = new HashMap<String, Integer>();
levels.put("Pikachu", 5);
levels.put("Gardevoir", 12);
levels.put("Pikachu", 7);
System.out.println(levels.size());
```

(d) In task *(c)*, why does `levels` have the type `Map<String, Integer>` and not `Map<String, int>`? (1 points)

4 points total

4 Recursion

(a) The *Ackermann-Péter function* is defined as follows:

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,1) & m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & m > 0 \text{ and } n > 0 \end{cases}$$

There is nothing particularly interesting about this function, except that its value grows *very* quickly.

Your task: write a recursive method `public static int ack(int m, int n)` which implements the Ackermann-Péter function. (2 points)

(b) SAS has contracted you to write a small program to compute a flight plan for travel from an airport A to an airport B, with 0 or more transfers. For the purpose of this question, a flight plan is simply a string listing, in order from start to finish, all airports visited when travelling from A to B. Figure 4 shows a graph of which direct connections between airports exist.

You have been given a method `public static String[] directFlights(String airport)`, which returns an array containing the name of every airport to which there are direct flights from `airport`. For example, `directFlights("GOT")` will return {`"ARN"`, `"HEL"`, `"FRA"`}, as Landvetter (GOT) has direct flights to Arlanda (ARN), Helsinki (HEL) and Frankfurt (FRA).

Your task: write a recursive method `String flightPlan(String from, String to)` which returns a flight plan for traveling from airport `from` to airport `to`. If there is no possible flight plan, the method should return `null`.

- `flightPlan("GOT", "GOT")` should return `"GOT"`, because our journey both starts and ends at Landvetter.
- `flightPlan("GOT", "FRA")` should return `"GIT FRA"`, because we start at Landvetter and just need to take a single direct flight to Frankfurt.
- `flightPlan("GOT", "KIX")` should return `"GOT FRA KIX"`, because while there are no direct flights from Landvetter to Osaka, there is a direct flight from Landvetter to Frankfurt, and then a direct flight from Frankfurt to Osaka.
- `flightPlan("FRA", "GOT")` should return `null`, because there are no flights from Frankfurt to Landvetter, even with transfers.

Note that your solution needs to work *even if more airports are added to the graph*, but you may assume that there will be *no cycles* in the graph. I.e. if it's possible to travel from GOT to KIX, it will *not* be possible to travel from KIX to GOT.

Also note that your solution does *not* need to return the best flight plan available: `flightPlan("GOT", "FRA")` may for instance return either `"GOT FRA"` or `"GOT ARN FRA"`, because both are possible ways of getting from Landvetter to Frankfurt. (4 points)

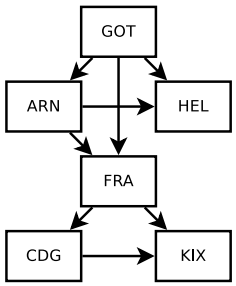


Figure 1: Example of direct flight connections

6 points total

Java Quick Reference Guide

User Input and Output Java applications and applets can get input and output through the console (command window) or through dialogue boxes as follows:

```
System.out.println("This is displayed on the console");

Scanner scanner = new Scanner(System.in);
String input = scanner.nextLine();
int n = scanner.nextInt();

import javax.swing.*;
JOptionPane.showMessageDialog(null,
  "This is displayed in a dialogue box");

String input = JOptionPane.showInputDialog("Enter a string");
```

Data Types

`boolean`	Boolean type, can be `true` or `false`
`byte`	1-byte signed integer
`char`	Unicode character
`short`	2-byte signed integer
`int`	4-byte signed integer
`long`	8-byte signed integer
`float`	Single-precision fraction, 6 significant figures
`double`	Double-precision fraction, 15 significant figures

Operators

`+ - * / %`	Arithmetic operators (`%` means *remainder*)
`++ --`	Increment of decrement by 1 `result = ++i;` means increment by 1 first `result = i++;` means do the assignment first
`+= -= *= /= %=` etc.	E.g. `i+=2` is equivalent to `i = i + 2`
`&&`	Logical AND, e.g. `if (i > 50 && i < 70)`
`||`	Logical OR, e.g. `if (i < 0 || i > 100)`
`!`	Logical NOT, e.g. `if (!endOfFile)`
`== != > >= < <=`	Relational operators

Control Flow - `if ... else` `if` statements are formed as follows (the `else` clause is optional).

```
String dayname;
...
if (dayname.equals("Sat") || dayname.equals("Sun")) {
  System.out.println("Hooray for the weekend");
}
else if (dayname.equals("Mon")) {
  System.out.println("I don't like Mondays");
}
else {
  System.out.println("Not long for the weekend!");
}
```

Control Flow - Loops Java contains three loop mechanisms:

```
int i = 0;
while (i < 100) {
  System.out.println("Next square is: " +  i*i);
  i++;
}

for (int i = 0; i < 100; i++) {
  System.out.println("Next square is: " +  i*i);
}

int positiveValue;
do {
  positiveValue = getNumFromUser();
}
while (positiveValue < 0);
```

Defining Classes When you define a class, you define the data attributes (usually `private`) and the methods (usually `public`) for a new data type. The class definition is placed in a `.java` file as follows:

```
// This file is Student.java. The class is declared
// public, so that it can be used anywhere in the program
public class Student {
  private String name;
  private int    numCourses;

  // Constructor to initialize all the data members
  public Student(String name, int numCourses) {
    this.name = name;
    this.numCourses = numCourses;
  }

  // No-arg constructor, to initialize with defaults
  public Student() {
    this("Anon", 0);       // Call other constructor
  }

  // Other methods
  public void attendCourse() {
    this.numCourses++;
  }
}
```

To create an object and send messages to the object:

```
public class MyTestClass {
  public static void main(String[] args) {
    // Step 1 - Declare object references
    // These refer to null initially in this example
    Student me, you;

    // Step 2 - Create new Student objects
    me  = new Student("Andy", 0);
    you = new Student();

    // Step 3 - Use the Student objects
```

```
    me.attendCourse();
    you.attendCourse()
  }
}
```

Enum An `enum` is a type which may have one of only a specific set of values. To declare an `enum` by the name `Day`, which can represent any of the weekdays:

```
enum Day {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

Values of type `Day` must be one of `Day.Mon`, `Day.Tue`, etc., and can be safely compared using the `==` operator. As an example:

```
public static boolean isWeekend(Day day) {
  if(day == Day.Sat || day == Day.Sun) {
    return false;
  } else {
    return true;
  }
}
```

Arrays An array behaves like an object. Arrays are created and manipulated as follows:

```
// Step 1 - Declare a reference to an array
int[] squares;          // Could write int squares[];

// Step 2 - Create the array "object" itself
squares = new int[5];

// Creates array with 5 slots
// Step 3 - Initialize slots in the array
for (int i=0; i < squares.length; i++) {
  squares[i] = i * i;
  System.out.println(squares[i]);
}
```

Note that array elements start at [0], and that arrays have a `length` property that gives you the size of the array. If you inadvertently exceed an array's bounds, an exception is thrown at run time and the program aborts.

Note: Arrays can also be set up using the following abbreviated syntax:

```
int[] primes = {2, 3, 5, 7, 11};
```

Static Variables A `static` variable is like a global variable for a class. In other words, you only get one instance of the variable for the whole class, regardless of how many objects exist. `static` variables are declared in the class as follows:

```
public class Account {
  private String accnum; // Instance var
  private double balance = 0.0; // Instance var
  private static double intRate = 5.0;  // Class var
  ...
}
```

Static Methods A `static` method in a class is one that can only access `static` items; it cannot access any non-`static` data or methods. `static` methods are defined in the class as follows:

```
public class Account {
  public static void setIntRate(double newRate) {
    intRate = newRate;
  }

  public static double getIntRate() {
    return intRate;
  }
  ...
}
```

To invoke a `static` method, use the name of the class as follows:

```
public class MyTestClass {
  public static void main(String[] args) {
    System.out.println("Interest rate is" +
      Account.getIntRate());
  }
}
```