Linköpings Tekniska Högskola,
ITN / Campus Norrköping,
Examinator: Dr Mark Eric Dieckmann

Tentamen

TND002 Object-oriented programming

Date: 2022-06-10   Time: 8-13 (8-15)

Tools: The lecture slides will be available on your computer during the exam. No other tools are permitted. You do not have access to the internet.

Point requirements: The exam consists of 3 exercises, each of which gives a maximum of 10 points. Full points are awarded if the code runs correctly and if its structure (methods and variables) matches the description in the exercise sheet. A program that is partially and correctly implemented gives points too. Comments in the code can be useful but they are not necessary.

A total of 30 points is awarded. You need 15 points or more to pass the exam. You need 20 points or more to get a 4. You need 25 points or more to get a 5.

LiU instructions regarding computer exam: When instructed to do so by the examination supervisor, log in using your LiU-ID. If you are writing your examination anonymously, or are unregistered or have had help registering for the examination, you must take your examination anonymously. In these cases, a box appears where you enter your AID (anonymous ID) number, which is handed out by the examination guard. After log-in, a folder opens, containing the files you will need for the examination. The folder can also be reached with the shortcut "Exam workspace", which is located on the desktop. Follow the examination guards instructions, to complete your examination. When you are finished, move the files you want to submit to "Exam hand in", which is located on the desktop. Remember to log out of the computer.

Specific to TND002: Submit the source files (ending with .java). Ideally, you put all source files belonging to the same exercise into one .zip file and submit 3 .zip files (Ex1.zip, Ex2.zip and Ex3.zip).

Examiner: Mark Dieckmann, tel 011-363257, I will visit regularly the exam rooms.

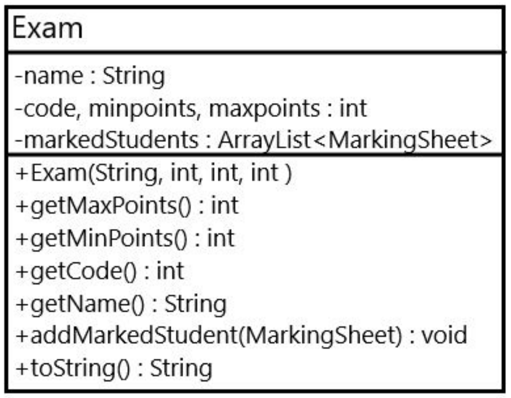
Results: I will send you my marking sheet on request.

Question 1: 10 points

Summary: You develop a framework that allows you to register students for exams. Students' names are stored in the external file "Students.txt". You specify via console input the maximum number of points an exam gives and the minimum number of points a student needs to pass it. Students who complete an exam get either a "G" (godkänt) or an "U" (underkänt). Instances of **Student** and **Exam** correspond to students and exams, while instances of **MarkingSheet** correspond to one marked exam of one student. The students and exams are managed by an instance of the class **Database**. Once you implemented everything, running the main method in the fully implemented class **Question1** should give you the console output appended at the end of Question 1.

Task 1: Implement Exam (1.5 p)

This class contains variables and methods that are relevant for an exam.


Exam
-name : String
-code, minpoints, maxpoints : int
-markedStudents : ArrayList<MarkingSheet>
+Exam(String, int, int, int)
+getMaxPoints() : int
+getMinPoints() : int
+getCode() : int
+getName() : String
+addMarkedStudent(MarkingSheet) : void
+toString() : String

name is the name of the course. It is initialized with the first constructor argument.

The other constructor arguments initialize the course code *code*, the maximum number of points *maxpoints* the exam can give and the lowest number of points *minpoints* that give a *godkänt* "G". If the student gets fewer points, the grade is *underkänt* "U".

markedStudents holds all marking sheets of this exam. It is initialized in the constructor.

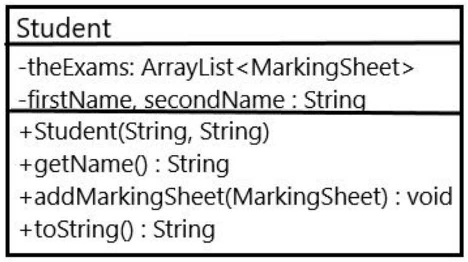
getMaxPoints() and *getMinPoints()* return the values of *maxpoints* and *minpoints*. *getCode()* returns the course code. *getName()* returns the name of the course.

addMarkedStudent(arg) adds a marking sheet to the array *markedStudents*.

toString() returns a header followed by the return values of the *toString()* methods of all marking sheets stored in *markedStudents*. The header is a formatted string with 12 spaces for the name (String) of the exam followed by ", Code: " and 4 spaces for the course code (int), followed by ", Maximum points: " and 2 spaces for the corresponding value (int), followed by ", Minimum points: " and 2 spaces for the corresponding value (int). See also the first line of the last paragraph on the console output.

Task 2: Implement Student (1.5 p)

This class contains variables and methods that are relevant for a student.


Student
-theExams: ArrayList<MarkingSheet>
-firstName, secondName : String
+Student(String, String)
+getName() : String
+addMarkingSheet(MarkingSheet) : void
+toString() : String

Each student has one first name *firstName* and one surname *secondName*. Both names are initialized in the constructor with the values of the argument list.

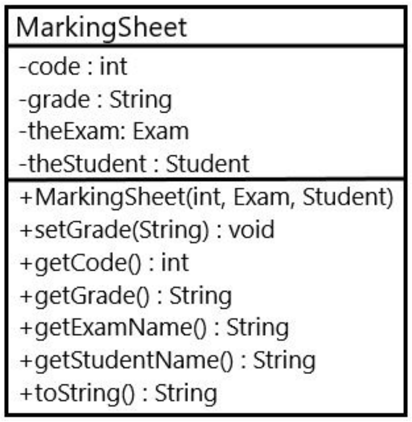
theExams holds the marking sheets of the exams that were taken by the student. This array is initialized in the constructor.

getName() returns the surname of the student and *addMarkingSheet(arg)* adds *arg* to the array *theExams*.

toString() returns the unformatted header "Student: ", followed by the first name, followed by one space, followed by the surname. This header is followed by one line of information for each exam the student took. You start with the name of the course, followed by ", Grade: ", followed by the grade the student got. See also the second-last paragraph on the console output.

Task 3: Implement MarkingSheet (2 p)

A marking sheet contains the result of one exam taken by one student.



code contains the course code of the exam. It is initialized with the first constructor argument.

grade is the grade "G" or "U" the student got. It is set with the *setGrade(arg)* method.

theExam and *theStudent* are references to the student and to the exam. Both are initialized in the constructor with its arguments.

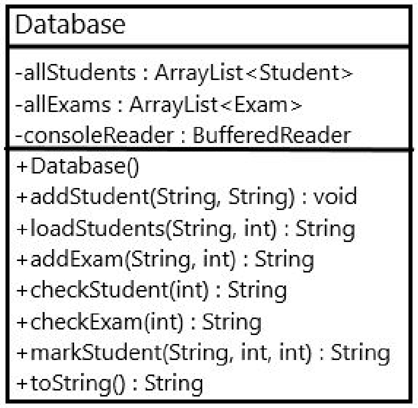
getCode() and *getGrade()* return the course code and the student's grade.

getExamName() returns the name of the exam.

getStudentName() returns the surname of the student.

toString() returns the unformatted string "Student : ", followed by the surname of the student, followed by ", Grade: ", followed by the grade (see the bottom 4 lines of the console output).

Task 4: Implement Database (5 p)


Database
-allStudents : ArrayList<Student>
-allExams : ArrayList<Exam>
-consoleReader : BufferedReader
+Database()
+addStudent(String, String) : void
+loadStudents(String, int) : String
+addExam(String, int) : String
+checkStudent(int) : String
+checkExam(int) : String
+markStudent(String, int, int) : String
+toString() : String

The database provides the methods to create and manage students and exams.

allStudents and *allExams* contain lists of all students and exams. Both are initialized in the constructor.

consoleReader will be used to read in text from the console. It is initialized in the constructor.

addStudent(arg1, arg2) creates an instance of **Student** and initializes his/her first name and surname with the values in the argument list. The method attaches the new student to *allStudents*.

loadStudents(arg1, arg2) loads in *arg2* students from the file named *arg1*. Make sure that an IO exception does not crash the code. You create a reader in the method (local variable) and close it at the end of the method. The method returns the formatted string

"Loaded x students" where x is an integer with two digits (see first line in the console output). The method should return "Something went wrong" if there was an exception.

addExam(arg1, arg2) initializes the name and the course code of an exam with *arg1* and *arg2*. It asks the question "Maximum points possible: " and reads in *maxpoints* using the *consoleReader*. It asks "Minimum points possible: " and reads in *minpoints*. Make sure that the code is not crashed by an IO exception or by a number format exception. If there is a number format exception, the method should repeat these steps until they worked. Once an exam has been created, the method should return an unformatted string "Course: " followed by the course name followed by " added" (see also paragraphs 2-4 in the console output).

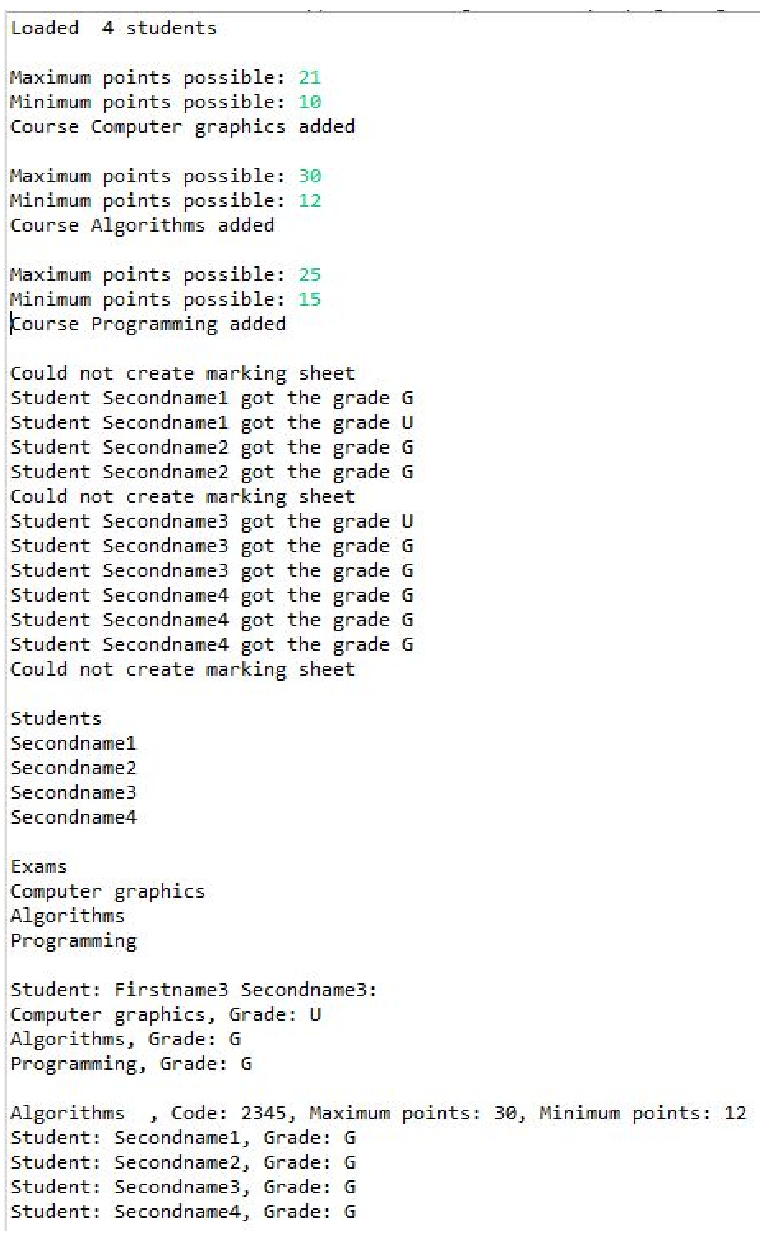
checkStudent(arg) returns the return value of the *toString()* method of the student in the slot *arg* of *allStudents* or "Student does not exist" if *arg* exceeds the maximum slot number (See second-last paragraph in the console output).

checkExam(arg) returns the return value of the *toString()* method of the exam in the slot *arg* of *allExam* (See last paragraph in the console output) or "Exam does not exist" if there is no slot *arg*.

markStudent(arg1, arg2, arg3) fetches the student with the surname *arg1* and the exam with the course code *arg2*. If that student or that course code does not exist or if *arg3* exceeds the maximum points given for the exam, then the method should return "Could not create marking sheet".

If both exist and *arg3* is valid, the method should create a marking sheet using the course code, the exam, and the student. It should set the grade to "G" if the student got more than the minimum number of points needed or "U" otherwise. It should return a formatted string that starts with "Student " and is followed by a string with 10 spaces for the surname of the student followed by "got the grade " followed by the grade (one character). These return values are shown in the 5th paragraph of the console output.

toString() should return the String "Students", followed by a list of the student's surnames followed by an empty line, followed by "Exams", followed by the names of the courses (see paragraphs 6 and 7 on the console output).

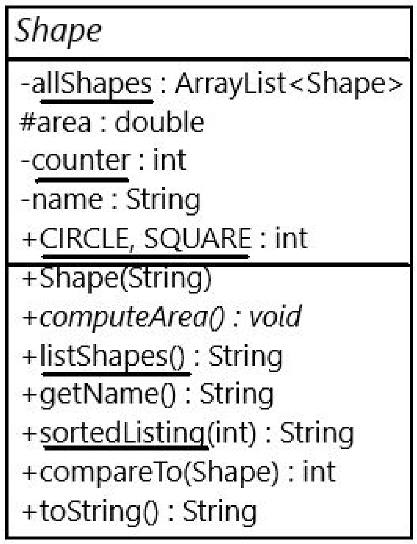


```
Loaded  4 students

Maximum points possible: 21
Minimum points possible: 10
Course Computer graphics added

Maximum points possible: 30
Minimum points possible: 12
Course Algorithms added

Maximum points possible: 25
Minimum points possible: 15
Course Programming added

Could not create marking sheet
Student Secondname1 got the grade G
Student Secondname1 got the grade U
Student Secondname2 got the grade G
Student Secondname2 got the grade G
Could not create marking sheet
Student Secondname3 got the grade U
Student Secondname3 got the grade G
Student Secondname3 got the grade G
Student Secondname4 got the grade G
Student Secondname4 got the grade G
Student Secondname4 got the grade G
Could not create marking sheet

Students
Secondname1
Secondname2
Secondname3
Secondname4

Exams
Computer graphics
Algorithms
Programming

Student: Firstname3 Secondname3:
Computer graphics, Grade: U
Algorithms, Grade: G
Programming, Grade: G

Algorithms  , Code: 2345, Maximum points: 30, Minimum points: 12
Student: Secondname1, Grade: G
Student: Secondname2, Grade: G
Student: Secondname3, Grade: G
Student: Secondname4, Grade: G
```

Question 2: 10 points

Summary: You develop a software framework with one superclass **Shape** and two subclasses **Square** and **Circle**, which correspond to geometrical shapes. The superclass contains methods and variables that are used by both subclasses but it should not be possible to initialize **Shape** only by itself. All instances of **Square** should be connected by a linked list and the same should be the case for all instances of **Circle**. I describe what a linked list is in Task 2. **Question2** is partially implemented. You need to expand it so that it can access the linked list and write its content to the console.

Task 1: Implement Shape (5 p)



allShapes is an array that contains all shapes that were created in *main()*.

area is the area of an object that is an instance of a subclass of **Shape**.

counter is initialized to 1 when it is declared.

name is the name given to the instance of shape.

CIRCLE and *SQUARE* should be set to 0 and 1, respectively.

Initialize *name* in the constructor *Shape(arg)* by concatenating *arg* and the value of *counter.* Increase *counter* by one. Add the new shape to *allShapes.*

computeArea() is only declared in **Shape**.

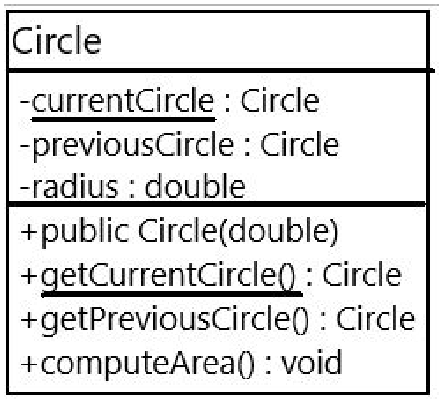
listShapes() should return a string with the header "List of shapes:" followed by the return values of the *toString()* methods of all shapes in *allShapes* (See also first paragraph / left column of the console output of **Question2**).

getName() returns the value of *name.*

sortedListing(arg) returns a sorted list of all circles in *allShapes* if *arg* equals *CIRCLE* or a sorted list of all squares in *allShapes* if *arg* equals *SQUARE.* You sort by the value of *area* using the inbuilt sorting method that is based on *compareTo(arg)* (See also **Question2** and the console output / right column).

toString() returns the unformatted string "Name: " followed by the value of *name*, followed by a formatted string "Area: x" where x is a float with 2 digits before the radix mark and two after it (for example: 10,25 or _2,30 if the number is less than 10).

Task 2: Implement Circle, a subclass of Shape (2 p)



```
Circle
-currentCircle : Circle
-previousCircle : Circle
-radius : double
+public Circle(double)
+getCurrentCircle() : Circle
+getPreviousCircle() : Circle
+computeArea() : void
```

currentCircle and *previousCircle* are the reference variables you use to build the linked list.

A linked list works as follows. You always keep the address of the latest circle in *currentCircle*. *previousCircle* contains the address of the circle you instantiated before that one. When you want to go through the linked list, you start at the circle referenced by *currentCircle*. You access the next one in the list through *previousCircle*. You repeat this until you get to the last one in the linked list (the first one you created).

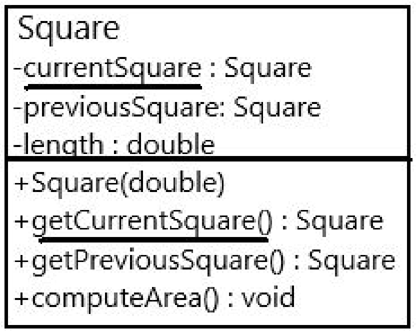
radius stores the value of the circle's radius.

The constructor *Circle(arg)* initializes *radius* with *arg* and calls the constructor of the superclass with "Circle". It initializes *area* by calling the method *computeArea()*. It also builds the linked list as discussed above.

getCurrentCircle() and *getPreviousCircle()* return the values stored in *currentCircle* and *previousCircle*.

computeArea() computes the area of the circle based on the value of *radius*.

Task 3: Implement Square, a subclass of Shape (2 p)



```
Square
-currentSquare : Square
-previousSquare: Square
-length : double
+Square(double)
+getCurrentSquare() : Square
+getPreviousSquare() : Square
+computeArea() : void
```

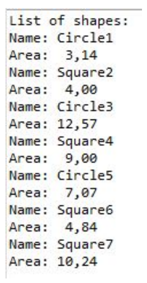
You follow the same implementation strategy as the one for **Circle**. The only differences are the way you calculate the area of a square and the variable types and the return types and names of some methods.

You build the linked list for the squares in the same way as the one for the circles.

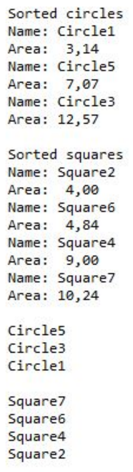
Task 4: Complete main() (1 p)

You have to add some code to the *main()* method, which goes through the linked lists of circles and squares and writes the return values of their *getName()* methods to the console. Your console output should equal the one shown on the next page. The linked lists can be seen on the bottom right. There should be 4 squares and 3 circles.

The console output is split into two parts and shown to the right.



```
List of shapes:
Name: Circle1
Area:  3,14
Name: Square2
Area:  4,00
Name: Circle3
Area: 12,57
Name: Square4
Area:  9,00
Name: Circle5
Area:  7,07
Name: Square6
Area:  4,84
Name: Square7
Area: 10,24
```

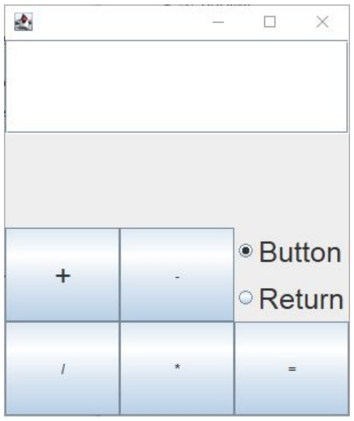


```
Sorted circles
Name: Circle1
Area:  3,14
Name: Circle5
Area:  7,07
Name: Circle3
Area: 12,57

Sorted squares
Name: Square2
Area:  4,00
Name: Square6
Area:  4,84
Name: Square4
Area:  9,00
Name: Square7
Area: 10,24

Circle5
Circle3
Circle1

Square7
Square6
Square4
Square2
```

Question 3: 10 points

Summary: You develop a pocket calculator that can add, subtract, divide and multiply a sequence of numbers. You should be able to switch between two operational modes. In the first mode, you type in the full sequence of numbers separated by the arithmetic symbols + - / *. There has to be an "=" at the end. Otherwise the GUI should not do anything. Once you press return, your calculator should perform the operation and write the result at the end of the equation. In the second mode, you type in the numbers directly into the text field and use the buttons to type in the arithmetic symbols.

Task 1: Implement this GUI (5 p)



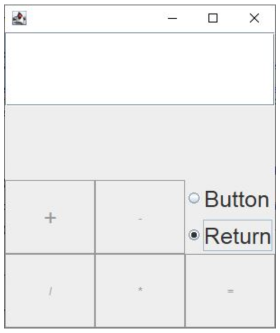
A text field and a text label take up the upper half of the GUI. There are 5 buttons and two radio buttons in the lower half.

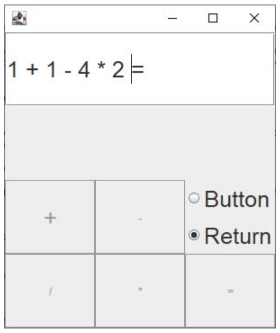
When you launch the GUI, the radio button labeled "Button" should be selected and all buttons should be enabled.

All fonts should be plain Sans Serif with the size 25.

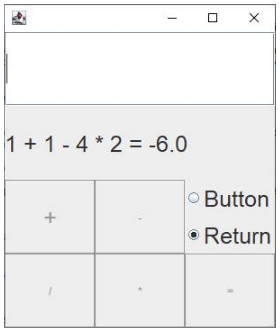
The size of the GUI should be such that it is optimal for the size of the elements and the code should stop running when you close the window.

Task 2: Implement the option "Return" (2 p)





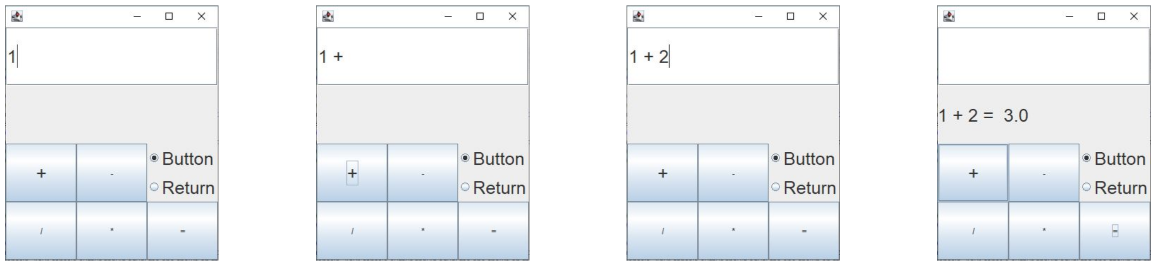
1 + 1 - 4 * 2 =



1 + 1 - 4 * 2 = -6.0

If the button "Return" is selected, all buttons should be disabled and the GUI should look like to the left. You can type in an equation like the one shown in the middle image leaving one space between numbers and arithmetic symbols. When you press return, your calculator should perform the arithmetic operation, move the equation from the text field to the label, and write the result to the right of the "=" sign (right image).

Task 3: Implement the option "Button" (2 p)



All buttons should be enabled. You type in numbers directly into the text field (left image). Pressing a button should let the arithmetic symbol appear in the text field (second from the left). You type in the next number directly into the text field. When you press "=", the equation should be moved from the text field to the label and the result should appear on the right of "=" (right image).

Task 4: Extra point (1 p)

If you can change smoothly between both states and if only one radio button is clicked at any time, you get 0.5 points extra. If your algorithm takes into account that "*" and "/" operations are performed before the "+" and "-", you get another 0.5 points.