CHALMERS
EXAMINATION / TENTAMEN

Course code/kurskod	Course name/kursnamn		*
DIT042/DIT043	Object-Oriented Programming		X
Anonymous code Anonym kod	Examination date Tentamensdatum	Number of pages Antal blad	Grade Betyg
~~547~~ DIT043 547	28/10/2021	9	5

* I confirm that I've no mobile or other similar electronic equipment available during the examination.
Jag intygar att jag inte har mobiltelefon eller annan liknande elektronisk utrustning tillgänglig under examinationen.

Solved task Behandlade uppgifter No/nr	Points per task Poäng på uppgiften	Observe: Areas with bold contour are to completed by the teacher. Anmärkning: Rutor inom bred kontur ifylles av lärare.
1	15	
2	25	
3	40	
4	~~13~~	
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
Bonus credits/ poäng	10	
Total examination points Summa poäng	93+10=103	

4

Q1)

15/15

Total in the constructor is -5
Total in the constructor is -2
Total in the constructor is 2
Total after update is 0
mA is num1: 5, num2: 3
mB is num1: -7, num2: -5
mC is num1: 4, num2: 2
method: num1 = 11 and num2 = 8
main: num1 = 8 and num2 = 3
mA is num1: 2, num2: 4
mB is num1: 10, num2: 5
mC is num1: 10, num2: 5
mA: num1: 2, num2: 4
mB: num1: 8, num2: 3
mC: num1: 8, num2: 3

Excellent :)

4

Question 2

Q2.1) 10p

In this particular situation, I would say that using an Interface is a better design solution than using an Abstract Class.

Firstly, there are no attributes that can be shared between the ~~Xxx~~ Rectangle, Triangle and Circle classes. This ~~already~~ An advantage of using an Abstract Class would be that it can ~~store~~ declare "common attributes (as in those attributes are used by the subclasses). However, that is not ~~too~~ that helpful in this scenario ~~Rectangle~~ the Rectangle class and Triangle class both have a height attribute (which ~~Circle do~~ Circle does not have) but this isn't enough to justify the switch to using an abstract from the current design (as seen in the class diagram).

Secondly, the methods such as getArea and getPerimeter will have different implementations in the Rectangle, Triangle and Circle classes. ~~Creating~~ Specifying the signature of abstract methods in the Interface (and leaving the bodies blank) makes much more sense than specifying an implementation in the abstract class unneccasarily and overriding in the ~~subt~~ subclasses. Of course, one could argue to just create abstract methods in the abstract class but if all methods in an abstract class are abstract and no attributes are needed (which is the case here), one might as well just use an Interface as only an Interface's functionality is needed. Furthermore, the constructor an abstract class could provide isn't greatly beneficial either.

(Continued on Page 3)

4

Q 2.1 Continued

Finally, using an Interface will lead to more flexibility (as classes outside this hierarchy could implement it if needed). An interface also helps the code look clean and readable, which helps with maintainability. In conclusion, due to all the above reasons, I disagree with Jane.

Great answer. Very complete and well written!

Q 2.2

Choosing to inherit from an Abstract ~~Class~~ Class instead of normal (non-abstract) class leads to a number of advantages and disadvantages.

One advantage of using an Abstract Class is that it allows abstract methods to be specified. This means that the subclasses that inherit from this abstract class can implement the body of the methods themselves. The Abstract Class simply specifies (and enforces) what functionality needs to be implemented elsewhere. This not only increases readability but also due to the nature of the design, allows for better maintainability and easier refactoring.

Another advantage with this design is that abstract methods could be used to avoid or at least reduce overriding. Overriding is generally considered a bit risky as we don't want the correct implementation to be overridden and changed accidentally. Especially in larger hierarchies, it can be difficult to keep track of different overridden methods in different classes and abstract classes can help solve or at least improve the situation.

On the other hand, a disadvantage of using an abstract class for inheritence is that often times, it isn't fully functional due to abstract methods and is ~~dependant~~ dependant on sub classes. ~~This dependency is something we want to generally. These kinds of dependencies between two c~~ This kind of situation where one class is heavily dependent on another is something we generally want to avoid (Continued on Pg 4)

Q 2.2 Continued

15p.

~~to~~ ~~oo~~

Another significant disadvantage of using an Abstract Class is that although it ~~it often~~ has a constructor, it cannot be instantiated as an object. There are many situations where we make the top ~~to~~ most class ~~ob~~ as kind of a default class with basic behaviour. We might make a student class and have subclasses for specific types of students. In these kinds of situations, we still often want to be able to instantiate a default ~~stock~~ student object but that is not possible if this top-most (in the heirarchy) student class is an Abstract Class.

In conclusion, both inheriting from ~~X~~ abstract classes and regular classes have their pros and cons and should be used when appropriate.

Again. Great answer.
Each point very well explained.

4

(Q 3.1)

```
public Class CourseResult {
    private final String COURSE_NAME;
    private final double CREDIT;
    private final int COURSE_RESULT;
    private final static int LOWEST_RESULT = 0;
    private final static int HIGHEST_RESULT = 100;
    CourseResult ( String courseName, double credit, int courseResult) throws Exception {
        this.COURSE_NAME = courseName;
        this.CREDIT = credit;
        if (courseResult < LOWEST_RESULT || courseResult > HIGHEST_RESULT){
            throw new Exception("Invalid course result data.");
        } else {
            this.COURSE_RESULT = courseResult;
        }
    }
    public String getCourseName() {
        return COURSE_NAME; this.COURSE_NAME;
    }
    public double getCredit() {
        return this.CREDIT;
    }
    public int getCourseResult() {
        return this.COURSE_RESULT;
    }
    // Code is continued on Pg 6
```

4

```
public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (obj != null && obj.instanceof(this)) {
        CourseResult resultObj = (CourseResult) obj;
        if (resultObj.getCourseName == this.COURSE_NAME && resultObj.getCredit == this.CREDIT) {
            return true;
        }
    }
    return false;
}
}
```

9/10

Excellent :)

missing method getCourseWeight()

Q3.2

~~Re~~ Reuse?

(i) Composition will be used to create the relationship between the student class and the course result class. The student class will be the composite and the CourseResult class will be the component in this relationship. The reason this relationship should be chosen is that this is the simplest and most straight-forward object-oriented way of correctly implementing this functionality. Composition is often used when a "has-a" relationship is needed and that is the case here.

13/15

how does courseResult depend on student?

(ii) A new attribute will need to be declared in the student class. Something like, "private List<CourseResult> courseResults;" should be added. The constructor should initialize this ~~abst~~ attribute as an empty arraylist (as per Request 1) so no additional constructor parameters are needed. Since it is a private attribute, a getter is needed. An additional public void method that can add CourseResult objects to the list is also needed.

4

Q3.3

```
public double getGPA() {
    double totalWeightedGrades = 0.0;
    double totalCredits = 0.0;
    for (CourseResult result : courseResults) {
        totalWeightedGrades = totalWeightedGrades + (result.getCredit * result.getCourseResult);
        totalCredits = totalCredits + result.getCredit;
    }
    return totalWeightedGrades / totalCredits;
}
```

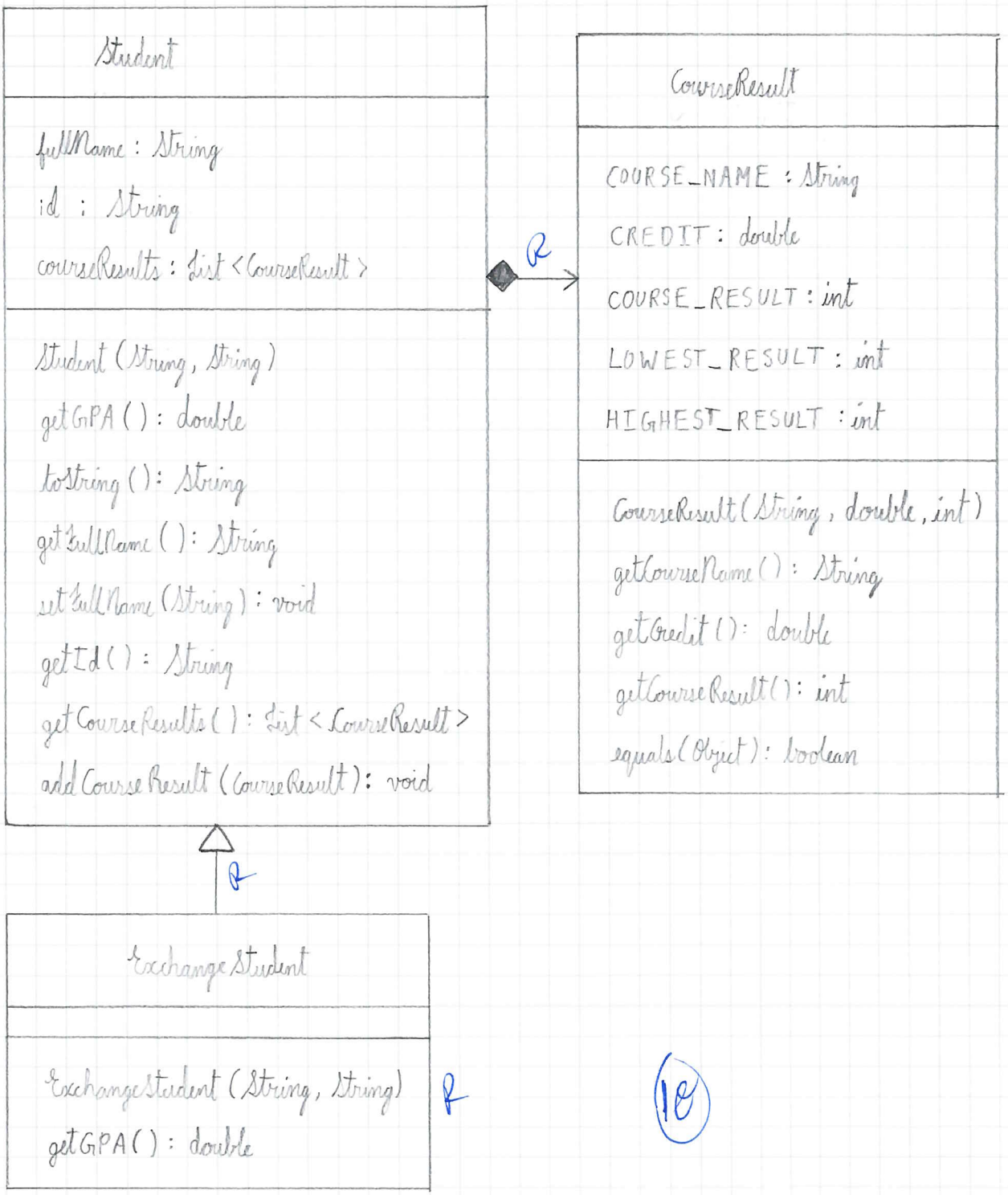
8/10

division by zero?

Q3.4

4

Q3.4)


Student
fullName : String
id : String
courseResults : List<CourseResult>
Student(String, String)
getGPA() : double
toString() : String
getFullName() : String
setFullName(String) : void
getId() : String
getCourseResults() : List<CourseResult>
addCourseResult(CourseResult) : void
R
CourseResult
COURSE_NAME : String
CREDIT : double
COURSE_RESULT : int
LOWEST_RESULT : int
HIGHEST_RESULT : int
CourseResult(String, double, int)
getCourseName() : String
getCredit() : double
getCourseResult() : int
equals(Object) : boolean
R
ExchangeStudent
ExchangeStudent(String, String)
getGPA() : double
R

10

4

Q4.1)

(i) The bug is on line 9 which currently states "userName = userName;"

~~(ii) The attributes of membership~~

(ii) One of the attributes of ~~num~~ the Membership class is called userName and the parameter in the constructor is called userName. This bug will result in the attribute "userName" not being assigned the value that is inputted while creating Membership objects (this is ~~done~~ attempted in the ~~mm~~ main method).

(iii) This can be fixed by changing line 9 to "this.userName = userName;" Alternatively, ~~it~~ either the attribute's or parameter's name could be changed to differentiate between them.

Q4.2)

~~Alan Turing: Points = 100. Credits: 6020~~
~~Ada Lovelace: Points =~~

~~Alan Turing: Points = 100. Credits: 6020.0~~
~~Ada Lovelace: Points = 120. Credits: 1116.0~~

Alan Turing: Points = 100. Credits: 6020.0 1
Ada Lovelace: Points = 130. Credits: 1116.0 3
Mary Keller: Points = 310. Credits: 6020.0 2
~~Grace Hopper:~~ ← Please ignore this crossed out line
Not enough points to convert. 2