CHALMERS
EXAMINATION / TENTAMEN

Course code/kurskod	Course name/kursnamn			
DIT093	Object-Oriented Programming			
Anonymous code Anonym kod		Examination date Tentamensdatum	Number of pages Antal blad	Grade Betyg
474		22-10-24	74	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 eximinationen.

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	X	10	
2	X	37	
3	X	37	
4	X	5	
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
Bonus poäng		8	
Total examination points Summa poäng på tentamen		89+8=97	Great Work! Nice Answers!

1

Question 1:

x= 9 y= 13 total = 23

x= 10 y= 6 total = 23

x= 7 y= 8 total = 23

p1 = 17 p2 = 27

x= 9 y= 13 total = 39

x= 7 y= 8 total = 39

x= 9 y= 13 total = 39

x= 20 y=7 total = 66

x= 20 y=17 total = 66

x= 8 y= 7 total = 66

10/10

Excellent job :)

(y)

1

Question 2 :

Q2.1 :

The bug in this code is that the supposed attributes of the Food class are created as static. This makes them to not belong to the object and be constants (as they are also final) instead of states of Food instances. we learned that in OOP, we do not use the static keyword as it is used for procedural and not OOP. ~~Se else~~

The consequence of this problem is once a food instance is created (line 58), every other instance of food created afterwards will have the same values for its attributes as the first one created. In other words, all food instances will be equal.

The only change needed to fix this problem is to remove the word "static" from the attributes (lines 2, 3, 4, 5) and replace them with the access modifier "private" as encapsulation is always a good practice.

7p

Q 2.2 :

```
public double getTotalDietKcals() {
    double totalKcals = 0.0;
    for (Food food : this.dietMenu) {
        totalKcals = totalKcals + food.getKcal();
    }
    return totalKcals;
}

public boolean addFood(Food food) {
    double totalKcals = getTotalDietKcals() + food.getKcal();
    if (totalKcals > this.maxKcal) {
        return false;
    } else {
        this.dietMenu.add(food);
        return true;
    }
}
```

GOOD JOB !!

8/8

1

Q 2.3 :

```
public void printTopProteinDiff() throws Exception {
    double proteinDiff = 0.0;
    String firstFoodName;
    String secondFoodName;
    if (this.dietMenu.size() < 2) {
        throw new Exception("Not enough food items. At least two
                food items must be registered");
    } else {
        for (Food food1 : this.dietMenu) {
            for (Food food2 : this.dietMenu) {
                double localProteinDiff = Math.abs(food1.getProtein
                        (), food2.getProtein());
                if (localProteinDiff > proteinDiff) {
                    proteinDiff = localProteinDiff;
                    firstFoodName = food1.getName();
                    secondFoodName = food2.getName();
                }
            }
        }
        System.out.println("Top protein difference: " + firstFoodName
                + " and " + secondFoodName);
    }
}
```

store the obj instead !!

-3p.

For-loop instead and use i and j=i+1

Here you get the wrong output for a list ex (1,1,7)

Init: 0
Rea: 2
Exep: 1
Alg: 4

7p

Q2.4 :

The common aspect between encapsulation and immutable objects is that they both strive to protect the state of the object, encapsulation does that by giving the control wheel to the object, so the object can ~~decide and~~ control its attributes and decide ~~no~~ whether they can be accessed (by adding getter methods) or changed (by adding setter methods), immutable objects protect their state by making them final which means that after creation, the object's attributes cannot be altered. +4

The distinguishing aspect (the difference) between encapsulation and immutable objects is that encapsulation gives the object the control of its attributes, so if the object wants to allow its attributes to be changed, it creates setter methods. However, immutable objects are robust and even the object itself cannot change its attributes, as it cannot add setters. So, in encapsulation the object has control ~~whether~~ over whether or not its attributes can change while immutable objects do not have control over changing their attributes. +4

8p

1

Q2.5 :

+3 Ash's suggestion with using the ObjectOutputStream requires the classes Diet and Food to implement the Serializable interface, because in order for ObjectOutputStream to be able to write the objects into Files, the abstraction of objects need to be serializable. However, we know that we cannot modify the Food class and since the ~~die~~ Diet class has composition ~~relation~~ (actually aggregation) relationship with the Food class, writing Diet instances will need the Food abstraction to be Serializable as well. Therefore, Ash's suggestion is not good since we cannot modify the food abstraction (class).

+3 Misty's suggestion is possible to implement. However, it requires modifications to the Diet class. Since, she suggests to use a text stream, which only writes strings, we can no longer save the Food instances as objects in a List in the Diet class. ~~Instead~~ Instead we will need to save the toString of each Food instance in a List (as an attribute) in the Diet class. Then we can write the Diet instances in a file. However, it should be noted that, we lose the objects in this process and in any case that we want to read the file, we can read strings and ~~it is~~ not objects. Therefore, this suggestion is to some extent appropriate but not ideal.

Continues on the next sheet

1

Continuation of Q2.5 :

+1 Brock's suggestions is not appropriate whatsoever. Scanner is used to read files or input, but not to write into files. So, this is wrong ~~and~~ and we cannot use the scanner to write into files.

7p

1

Q3.1 :

Paying attention to the Encoder class, I realised that every time that we add a new type of encoder, we will have to make modifications to the method encodedMessage() (line 76) by adding more if-then-else statements to the implementation of the method, so that we can satisfy the new conditions for how the newly added encoder type works. This violates the Open-Closed principle which states that classes should be open to extention but closed to modifications.

Using this principle, I will create hierarchy with the superclass being the abstract class Encoder that has the attributes "content" ~~and "splitPoint". So I would take away the attribute "type" as the class with each class would represent its type in its name. I would make the Encoder abstract because it is not of sense to instatiate an encoder with no type. All the methods in the Encoder The The second constructor would eliminate~~ make the method encodeMessage abstract ~~as each subclass would have its own specific implementation. The constructors would not receive "type" anymore and the other methods, the getters and setters would be intact. The second constructor (line 11) where the splitPoint is not always 0 will throw an Exception in case the splitPoint is negative or greater than content length.~~ ↳ still correct. :)

Continue on Next Sheet

1

Q3.7 continuation:

I will take away the attributes "splitPoin" and "type" as each subclass name will represent the type and only Splitator and Quantum subclasses will use the attribute splitPoint. The ~~metho~~ there will only be one constructor for the abstract class Encoder which only takes content. The method encodeMessage will become abstract and the setters and getters for splitPoint are deleted.

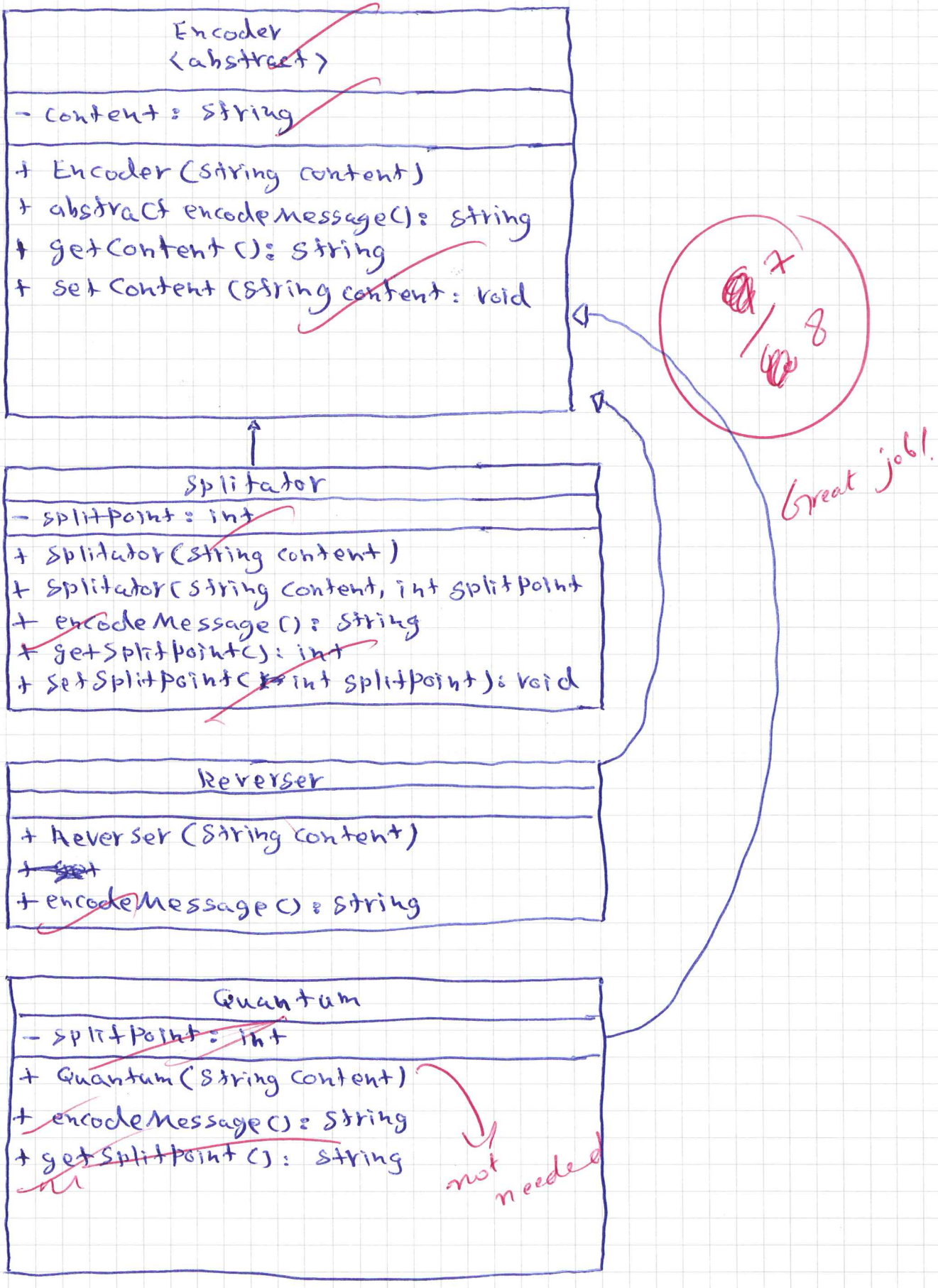
quantum doesn't need the split point

10p

Nice answer, but organize better

1

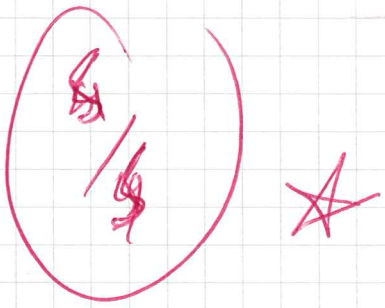
Q3.2 :


Encoder
<abstract>
- content : String
+ Encoder(String content)
+ abstract encodeMessage(): String
+ getContent(): String
+ setContent(String content): void
Splitator
- splitPoint : int
+ Splitator(String content)
+ Splitator(String content, int splitPoint
+ encodeMessage(): String
+ getSplitPoint(): int
+ setSplitPoint(int splitPoint): void
Reverser
+ Reverser(String content)
+ encodeMessage(): String
Quantum
- splitPoint : int
+ Quantum(String content)
+ encodeMessage(): String
+ getSplitPoint(): String
not needed
7/8
Great job!

1

Q3-3 :

to prevent invalid creation of encoders with negative ~~or~~ splitPoints or splitPoint greater ~~or equal~~ than the content length, the constructor "public Splitator(String content, int splitPoint) needs ~~the~~ "throws Exception" in the end of its signature and an if-then-else statement to throw the Exception if splitPoint is negative or greater than the content's length. The setSplitPoint method also needs "throws Exception" and the ~~cond~~ the if-then-else mentioned. No other classes need methods that throw Exceptions.



1

Q 3.4 :

```
public class Quantum extends Encoder {
    private int splitPoint;
    public Quantum(String content) {
        super(content);
        this.splitPoint = 0;
    }

    public int getSplitPoint() { return this.splitPoint }

    public String encodeMessage() {
        String encodedMessage;
        int divisiblity = super.getContent().length % 2;
        if ( divisiblity == 0 ) {
            encodedMessage = super.getContent.subString(0,
                    super.getContent().length / 2 );
        } else {
            encodedMessage = super.getContent + super.getContent;
        }
        return encodedMessage;
    }
```

7 / 7

1

Q3.5 :

Inheritance weakens encapsulation as the subclasses can have access to the superclass attributes via getters and setters. Composition does not weaken encapsulation as the composite cannot have access to the components attributes.

Inheritance is the reuse of both behaviour and attributes while composition is only the reuse of behaviour. via forwarding

Inheritance has higher resuability than composition as methods are inherited by the subclasses while although there is resusability in composition, but forwarding needs to be done.

Inheritance makes the code more complex due to overriding and late binding while composition is not ax complex as Inheritance.

composition has runtime dependency

8/10

1

Q4 : the letters a, b, c, d, e, f are not printed. they are there for the examiner to correspond to the code.

a) 26 ✓

b) -2 ✓

c) 42 ✓

d) -52 x 865

e) -2 x 5

f) 85 ✓

5/10