UNIVERSITETET I BERGEN

KANDIDAT

149

PRØVE

INF101 0 Objektorientert programmering

Emnekode	INF101
Vurderingsform	Skriftlig eksamen
Starttid	04.06.2021 07:00
Sluttid	04.06.2021 12:00
Sensurfrist	--
PDF opprettet	02.10.2022 11:22

Informasjon/ Information

Oppgave	Tittel	Oppgavetype
i	Egenerklæring/ Declaration	Informasjon eller ressurser
i	Kontaktinfo under eksamen	Informasjon eller ressurser
i	Generelt info om denne eksamen	Informasjon eller ressurser

Flervalgsoppgaver

Oppgave	Tittel	Oppgavetype
1	Typeparameter	Flervalg (flere svar)
2	Prinsipper i Objektorientert Programmering	Flervalg (flere svar)
3	Inheritance	Flervalg (flere svar)
4	Typesignaturen til Collections.sort	Flervalg (flere svar)

Programmering

Oppgave	Tittel	Oppgavetype
5	Teller	Programmering
6	ShoppingTest	Programmering
7	Par	Programmering
8	IFridge	Programmering
9	Vaksineplan	Programmering

Poeng fra Semesteroppgavene

Oppgave	Tittel	Oppgavetype
10	Poeng fra Semesteroppgavene V21	Tekstfelt

1 Typeparameter

```
public class Grid<T extends ShoppingItem> {

    private List<T> cells;
    private int columns;
    private int rows;

    public Grid(int rows, int columns, T initElement) {
        if (rows <= 0 || columns <= 0)
            throw new IllegalArgumentException();
        this.columns = columns;
        this.rows = rows;
        cells = new ArrayList<>(columns * rows);
        for (int i = 0; i < columns * rows; ++i) {
            cells.add(initElement);
        }
    }
```

Hvilke datatyper kan denne Grid-klassen bruke (hvilke datatyper kan typeparameteren T være)?
Velg ett eller flere alternativer

- [ ] *Grid* er generisk og kan dermed bruke alle datatyper
- [x] Alle datatyper som er subtyper av *ShoppingItem*
- [ ] Kun imutable datatyper
- [ ] Alle datatyper som arver fra *Grid*
- [ ] Kun primitive datatyper

Maks poeng: 5

2 Prinsipper i Objektorientert Programmering

```
public class Ball {

    private double radius;
    private Color color;

    public Ball(double radius, Color color) {
        this.radius = radius;
        this.color = color;
    }

    public Color getColor() {
        return color;
    }

    public double radius() {
        return radius;
    }

}
```

Hvilke av de følgende konseptene brukes i klassen *Ball*?

Velg ett eller flere alternativer

- [ ] Generics
- [x] Abstraction
- [ ] Polymorphism
- [ ] Inheritance
- [ ] Overriding
- [x] Encapsulation

Maks poeng: 5

3 Inheritance

I denne oppgaven tar vi for oss en kodebase som består av de følgende interfacene og klassene:

- et interface A
- et interface B som utvider A
- et interface C som utvider B
- en klasse D som implementerer A
- en klasse E som implementerer B
- en klasse F som implementerer C
- en klasse G som utvider D
- en klasse H som utvider E
- en klasse I som utivder F

I et program oppretter vi en variabel av typen B:

```
B minVariabel;
```

Hvilke typer kan et objekt ha for at det skal kunne legges i variabelen?

Med andre ord, hvilke typer kan en annen variabel `a` ha slik at følgende er lov:

```
minVariabel = a;
```

Hint: Det kan være nyttig å lage et klassediagram for disse klassene

Velg ett eller flere alternativer

- [ ] A
- [ ] B
- [ ] C
- [ ] D
- [x] E
- [x] F
- [ ] G
- [x] H
- [x] I

Maks poeng: 5

4 Typesignaturen til Collections.sort

For sortering av en liste i java kan en bruke metoden *Collections.sort* som har den følgende signaturen:

```
static <T extends Comparable<? super T>> void sort(List<T> list)
```

Velg de påstandene som er riktig for metoden *sort*.

Velg ett eller flere alternativer

- [ ] "static" betyr at metoden kan kalles uten å først konstruere et Collections-objekt.
- [ ] "static" betyr at klassen List må være statisk for at en liste skal kunne sorteres.
- [x] Dersom T utvider en klasse som er sammenlignbar med seg selv, så kan List sorteres.
- [ ] Dersom T er sammenlignbar med seg selv, så kan List sorteres.
- [ ] Dersom en klasse som utvider T er sammenlignbar med seg selv, så kan List sorteres.
- [x] Klassen List må utvide klassen Comparable for at listen skal kunne sorteres.

Maks poeng: 5

5 Teller

Vi mennesker er dårlig til å telle, for eksempel antall personer som har gått inn i et lokale eller antall runder som har blitt løpt på banen. Vi prøver å oppdatere et tall i hodet mens vi teller, men dersom vi blir distrahert så glemmer vi fort hvor langt vi var kommet. I slike tilfeller er det nyttig å ha en *teller* som kan huske dette tallet for deg.

Kilde: https://www.amazon.com/GOGO-Counter-Carnival-Manual-Mechanical/dp/B001KX1VW2

En kan gjøre tre ting med en teller:

- lese av heltallet,
- øke tallet i displayet med 1
- og resette tallet til 0.

Telleren har også en datainvariant: Tallet kan aldri bli negativ.

Lag et interface *ICounter* som beskriver hva en teller kan gjøre, og lag en klasse *Counter* som implementerer *ICounter.* Sørg for at klassen overholder datainvarianten.

(Du trenger ikke å håndtere noen maksimum begrensning på telleren).

Skriv ditt svar her

```
1  package inf101;
2
3  public interface ICounter {
4
5      int getCount();
6
7      void addOne();
8
9      void resetCounter();
10 }
11 package inf101;
12
13 public class Counter implements ICounter {
14
15     private int theCount;
16
17     public Counter(int startingCounter){
18         this.theCount = startingCounter;
19     }
20
21     @Override
```

```
21      @Override
22      public int getCount() {
23          return theCount;
24      }
25
26      @Override
27      public void addOne() {
28          theCount++;
29      }
30
31      @Override
32      public void resetCounter() {
33          theCount = 0;
34      }
35  }
36
```

Maks poeng: 8

6 ShoppingTest

Pål ønsker å utfordre Ebay med en handle-nettside. Så langt har han skrevet en handlevare-klasse *ShoppingItem*. Pål har også skrevet en test for om ulike handlevarer er like, men testen feiler.

Endre på *ShoppingItem* slik at testen passerer. *ShoppingItem*-objekter ansees å være like dersom de har samme type og samme merke (produsent).

```
@Test
public void shoppingItemEqualsTest() {
    ShoppingItem chicken1 = new ShoppingItem("chicken", "Prior");
    ShoppingItem chicken2 = new ShoppingItem("chicken", "Prior");
    ShoppingItem noodles = new ShoppingItem("noodles", "MrLee");

    assertNotEquals(chicken1, null);
    assertNotEquals(null, chicken1);

    assertNotEquals(chicken1, noodles);
    assertNotEquals(noodles, chicken1);

    assertEquals(chicken1, chicken1);
    assertEquals(chicken1, chicken2);
}
```

Skriv ditt svar her

```
1  package inf101;
2
3  /**
4   * A shopping item is an item you purchase at the store.
5   * A shopping item has an item type, such as chicken, fruit,
6   * vegetables, cleaning tools, soda, snacks, etc.,
7   * and a brand, such as Fjordland, Tine, Prior, etc.
8   *
9   * @author Sondre Bolland
10  */
11 public class ShoppingItem {
12
13     /** Type of shopping item */
14     private String itemType;
15     /** Brand (producer) of shopping item */
16     private String brand;
17
18     public ShoppingItem(String itemType, String brand) {
19         this.itemType = itemType;
20         this.brand = brand;
21     }
22
23     public String getItemType() {
24         return itemType;
25     }
26
27     public String getBrand() {
28         return brand;
29     }
30
31     @Override
32     public boolean equals(Object o) {
```

```
33          if(o == null){
34              return false;
35          }
36          ShoppingItem s = (ShoppingItem) o;
37          if(this.itemType == s.getItemType()) {
38              if(this.brand == s.getBrand()) {
39                  return true;
40              }
41          }
42          return false;
43      }
44  }
```

Maks poeng: 8

7 Par

Et *par* er et objekt som inneholder to ordnede verdier, *first* og *second*, der de to verdiene kan være av ulike typer. Par kan blant annet brukes til å returnere to verdier fra en funksjon i stedet for kun en.

Denne oppgaven består av tre deler.

1. Lag et generisk interface *IPair* med to metoder *getFirst* og *getSecond* som henholdsvis henter det første og andre objektet i paret.

2. Lag et ikke-generisk interface *IIntegerPair* som representerer et par med to heltall. Interfacet skal utvide *IPair*, og skal ha en metode som returnerer summen av de to tallene.

3. Lag en klasse som implementerer *IIntegerPair* og har en konstruktør som tar inn de to tallene som argument. Man skal ikke kunne bytte ut verdiene i et eksisterende par (klassen skal med andre ord være immutable).

Skriv ditt svar her

```
package inf101;

public interface IPair<F,S> {

    F getFirst();

    S getSecond();
}

package inf101;

public interface IIntegerPair extends IPair<Integer, Integer>{
    int getSum();
}

package inf101;

public class Pair implements IIntegerPair{

    private final int firstNum;
    private final int secondNum;

    public Pair(int firstNum, int secondNum){
        this.firstNum = firstNum;
        this.secondNum = secondNum;
    }
    @Override
    public int getSum() {
        return getFirst() + getSecond();
    }

    @Override
    public Integer getFirst() {
        return firstNum;
    }

    @Override
    public Integer getSecond() {
        return secondNum;
```

```
40        }
41    }
42
```

Maks poeng: 10

8 IFridge

Pål har et sideprosjekt hvor han håper å utfordre Samsung i hvitevare-bransjen med et digitalt kjøleskap. Han har skrevet et interface for hvordan kjøleskapet skal fungere.

Implementer interfacet IFridge.

Skriv ditt svar her

```
package inf101;

import java.util.ArrayList;
import java.util.List;

public class Fridge implements IFridge {


    private ArrayList<FridgeItem> itemsInFridge;
    private int fridgeSize;

    public Fridge(int fridgeSize) {
        itemsInFridge = new ArrayList<>();
        this.fridgeSize = fridgeSize;
    }

    @Override
    public int nItemsInFridge() {
        return itemsInFridge.size();
    }

    @Override
    public int totalSize() {
        return fridgeSize;
    }

    @Override
    public boolean placeIn(FridgeItem item) {
        if (nItemsInFridge() < totalSize()) {
            itemsInFridge.add(item);
            return true;
        }
        return false;
    }

    @Override
    public void takeOut(FridgeItem item) {
        try {
            for (FridgeItem i : itemsInFridge) {
                if (i.equals(item)) {
                    itemsInFridge.remove(i);
                }
            }
        } catch (IllegalArgumentException e) {
            System.err.println("This item does not exist in the fridge");
        }
    }

    @Override
    public FridgeItem takeOut(String itemName) {
        FridgeItem takeIt = null;
        ArrayList<FridgeItem> lookingForItem = new ArrayList<>();
        for (FridgeItem i : itemsInFridge) {
            if (i.toString() == itemName) {
                if (takeIt == null) {
                    takeIt = i;
```

```
57              } else {
58                  if (i.compareTo(takeIt) > 0) {
59                      takeIt = i;
60                  }
61              }
62          }
63      }
64      if (takeIt == null) {
65          throw new IllegalArgumentException("This item does not exist in the frid
66      }
67      return takeIt;
68  }
69
70  @Override
71  public void emptyFridge() {
72      itemsInFridge.clear();
73  }
74
75  @Override
76  public List<FridgeItem> removeExpiredFood() {
77      ArrayList<FridgeItem> expiredItems = new ArrayList<>();
78      for (FridgeItem i : itemsInFridge) {
79          if (i.hasExpired()) {
80              takeOut(i);
81              expiredItems.add(i);
82          }
83      }
84      return expiredItems;
85  }
86  }
```

Maks poeng: 10

9 Vaksineplan

Du utvikler en programvare som skal fordele vaksiner til pasienter. Hver pasient som skal vurderes har en alder og en *alvorlighetsgrad for underliggende sykdom*. Graden går fra 0, som betyr at pasienten ikke har en underliggende sykdom, til 3, som betyr at pasienten har en alvorlig underliggende sykdom.

Vaksinene kommer på ulike tidspunkt, og programmet må derfor sette opp en vaksineringskø. Hvor en pasient havner i køen avhenger av to (meget forenklede) faktorer:

- Dersom en pasient A har en høyere alvorlighetsgrad enn en pasient B, skal A komme før B i køen.
- Dersom alvorlighetsgraden er lik for to pasienter, skal den eldste av de to pasientene komme først i køen.

I denne oppgaven bruker vi en enum UnderlyingConditionGrade. For å for eksempel angi alvorlighetsgrad 1 kan en bruke `UnderlyingconditionGrade.LOW`. For å hente en alvorlighetsgrad fra et enum-objekt `u` kan en bruke `u.getValue()`.

Denne oppgaven består av tre deloppgaver. Det kan være lurt å gjøre seg kjent med klassene som hører til oppgaven før man går i gang med besvarelsen.

1. For å finne ut hvem som trenger vaksinen mest må vi kunne sammenlikne to pasienter. Bruk kriteriene ovenfor til å la Patient implementere Comparable<Patient> slik at den av to pasienter som trenger vaksinen **mest** kommer **før** den andre i en sortert liste.

2. Vaccine-klassen er en spesifikasjon av en generell vaksine. Utvid klassen med to klasser Pfizer og Moderna som henholdsvis representerer vaksinene "Pfizer" og "Moderna". Begge klassene skal ha en konstruktør som tar en LocalDate som argument og setter denne til å være leveringsdatoen.

3. Du skal nå fullføre main-metoden i VaccinePlan-klassen (du kan ta vekk tegnene som kommenterer ut klassen). I main-metoden har du en liste med pasienter og en liste med vaksiner, og du skal tildele vaksinene til pasientene slik at de som trenger vaksinene mest får først. Merk at rekkefølgen i de to listene er tilfeldige. Bruk hjelpemetoden assignVaccine for å tildele en vaksine til en pasient. (Dersom du ikke har fått til de forrige deloppgavene kan du likevel gjøre denne, selv om om du får noen feilmeldinger eller svaret blir feil.)

Tips: compareTo-metoden skal generelt returnere

- -1 dersom dette objektet (this) kommer før det andre,
- 1 dersom dette objektet (this) kommer etter det andre og
- 0 dersom dette objektet verken kommer før eller etter det andre.

Skriv inn all implementerte metoder og klasser her.

```
1  package inf101;
2
3  public class Patient implements Comparable<Patient> {
4
5      private String name;
6      private UnderlyingConditionGrade underlyingConditionGrade;
7      private int age;
```

```
8
9       public Patient(String name, int age, UnderlyingConditionGrade underlyingConditi
10          this.name = name;
11          this.age = age;
12          this.underlyingConditionGrade = underlyingCondition;
13      }
14
15      public String getName() {
16          return name;
17      }
18
19      @Override
20      public int compareTo(Patient o) {
21          if (this.underlyingConditionGrade.getValue() != o.underlyingConditionGrade.
22              return Integer.compare(o.underlyingConditionGrade.getValue(), this.unde
23          } else {
24              return Integer.compare(o.age, this.age);
25          }
26      }
27  }
28
29  package inf101;
30
31  import java.time.LocalDate;
32
33  public abstract class Vaccine implements Comparable<Vaccine> {
34
35      private final LocalDate deliveryDate;
36
37      public Vaccine(LocalDate deliveryDate) {
38          this.deliveryDate = deliveryDate;
39      }
40
41      /**
42       * @return the date for when the vaccine is delivered.
43       */
44      public LocalDate getDeliveryDate() {
45          return deliveryDate;
46      }
47
48      /**
49       * @return the name of the vaccine.
50       */
51      public abstract String getName();
52
53      @Override
54      public String toString() {
55          return getName();
56      }
57
58      @Override
59      public int compareTo(Vaccine o) {
60          return this.deliveryDate.compareTo(o.deliveryDate);
61      }
62  }
63
64  package inf101;
65
66  import java.time.LocalDate;
67
68  public class Pfizer extends Vaccine{
69
70      public Pfizer(LocalDate deliveryDate) {
71          super(deliveryDate);
72      }
73
74      @Override
75      public String getName() {
            return "Pfizer";
```

```
        return "Pfizer";
    }
}

package inf101;

import java.time.LocalDate;

public class Moderna extends Vaccine{

    public Moderna(LocalDate deliveryDate) {
        super(deliveryDate);
    }

    @Override
    public String getName() {
        return "Moderna";
    }
}

package inf101;

import java.time.LocalDate;
import java.util.*;

public class VaccinePlan {

    public static void main(String[] args) {
        List<Patient> patients = getPatients();
        List<Vaccine> vaccines = getVaccines();
        Collections.sort(patients, Patient::compareTo);
        Collections.sort(vaccines);
        for(int i = 0; i<patients.size(); i++){
            printVaccineAssignment(patients.get(i),vaccines.get(i));
        }
    }

    private static void printVaccineAssignment(Patient patient, Vaccine vaccine) {
        System.out.println(patient.getName() + " får " + vaccine.getName() + "-vaks
            .");
    }


    private static List<Patient> getPatients() {
        return Arrays.asList(new Patient("Per", 43, UnderlyingConditionGrade.LOW),
                new Patient("Pål", 30, UnderlyingConditionGrade.NONE),
                new Patient("Anne", 51, UnderlyingConditionGrade.NONE),
                new Patient("Oddvar", 30, UnderlyingConditionGrade.HIGH),
                new Patient("Marie", 45, UnderlyingConditionGrade.NONE),
                new Patient("Gerd", 25, UnderlyingConditionGrade.MEDIUM));
    }

    private static List<Vaccine> getVaccines() {
        return Arrays.asList(
                new Pfizer(LocalDate.of(2021, 7, 1)),
                new Moderna(LocalDate.of(2021, 8, 5)),
                new Pfizer(LocalDate.of(2021, 8, 16)),
                new Moderna(LocalDate.of(2021, 8, 5)),
                new Pfizer(LocalDate.of(2021, 9, 3)),
                new Pfizer(LocalDate.of(2021, 8, 16)));
    }

}
```

Maks poeng: 14

10 Poeng fra Semesteroppgavene V21

Denne oppgaven skal du ikke gjøre noe på. Her får du poeng du for det du har gjort på semesteroppgavene.

Du er nå ferdig med eksamen :-)

God Sommer
hilsen Martin

Skriv en hyggelig melding til foreleser ;-)

Takk for et hyggelig semester :))

Maks poeng: 30