CHALMERS

Institutionen för data- och informationsteknik

TENTAMEN

KURSNAMN	**Objektorienterad programutveckling, fk 7.5p**
PROGRAM:	**TKIEK-2, TKTFY-3, TKTEM-3** **2018/2019, lp 2**
KURSBETECKNING	**TDA550**
EXAMINATOR	**Uno Holmer**
TID FÖR TENTAMEN	**Onsdagen den 21/8 2019, 08.30 – 12.30**
HJÄLPMEDEL	**Java API** (delas ut av skrivningsvakten)
ANSV LÄRARE	**Uno Holmer** **tel. 772 5730** **besöker tentamen ca kl. 9.30 samt ca 11.30**
DATUM FÖR ANSLAG	**Senast den 9/9 2019** **Datum för granskning meddelas på kursens hemsida**
ÖVRIG INFORM.	**Betygsgränser: 3 - 24p, 4 - 36p, 5 - 48p. (max 60p)**

TENTAMEN: Objektorienterad programutveckling, fk

Läs detta!

- *Uppgifterna är inte ordnade efter svårighetsgrad.*
- Börja varje hel uppgift på ett nytt blad. Skriv inte i tesen.
- Ordna bladen i uppgiftsordning.
- Skriv din tentamenskod på varje blad (så att vi inte slarvar bort dem).
- **Skriv rent dina svar. Oläsliga svar r ä t t a s e j!**
- Skriv inte med rödpenna.
- Programkod som finns i tentamenstesen behöver ej upprepas.
- Programkod skall skrivas i Java 5, eller senare version, och vara indenterad och renskriven.
- Onödigt komplicerade lösningar ger poängavdrag.
- Givna deklarationer, parameterlistor etc. får inte ändras. Fråga i oklara fall!
- Läs igenom tentamenstesen och förbered ev. frågor.

I en uppgift som består av flera delar får du använda dig av funktioner klasser etc. från tidigare deluppgifter, även om du inte löst dessa.

Lycka till!

Uppgift 1

Arve Arvidsson gillar arv men inte att fält och listor indexeras från 0 och uppåt. Arve anser att indexering bör starta på 1 och skapade därför nedanstående subklass till `ArrayList`. Var det en bra idé? Vilken välkänd designprincip bryter konstruktionen mot? Beskriv med stöd av kodexempel två scenarion där klassen ger upphov till fel!

```
public class ArveList<E> extends ArrayList<E> {
    public ArveList() {
        super();
    }
    // Code for the other constructors omitted

    @Override
    public E get(int index){
        if ( index < 1 || index > size() )
            throw new IndexOutOfBoundsException();
        return super.get(index-1);
    }

    @Override
    public E set(int index,E element){
        if ( index < 1 || index > size() )
            throw new IndexOutOfBoundsException();
        E oldElement = super.get(index-1);
        super.set(index-1,element);
        return oldElement;
    }
}
```

(5 p)

Uppgift 2

Betrakta nedanstående klasser:

```
public class Model {
    private Gui gui;
    private int state;
    public Model(Gui gui) {
        this.gui = gui;
    }
    public void changeState() {
        gui.showState(state);
    }
}
```

```
public class Gui extends JFrame {
    public void showState(int state) {
        // display state
    }
}
```

```
public class Main {
    public static void main(String [] arg) {
        Model m = new Model(new Gui());
        m.changeState();
    }
}
```

Ovanstående design är bristfällig eftersom modellen (klassen `Model`) känner till vyn (klassen `Gui`). Modifiera designen med användning av Observer-mönstret så att `Model` kan kompileras utan tillgång till `Gui`.

(6 p)

Uppgift 3

a)

I vilken grad är metoderna i klassen A är synliga i klasserna B, C, D respektive E? Motivera!

```
package p1;
public class A {
    private void f() {}
    public void g() {}
    protected void h() {}
    void i() {}
}

package p1;
public class B extends A { ... }

package p1;
public class C { ... }

package p2;
public class D extends p1.A { ... }

package p2;
public class E { ... }
```

(4 p)

b)

Vissa rader i `main` ger kompileringsfel, vilka och varför? Vad skrivs ut av de anrop som är korrekta? Motivera!

```
public interface Int {
    void f();
}
```

```
public abstract class A implements Int {
    public void g() { System.out.println("A.g"); }
    public void h() { System.out.println("A.h"); }
}
```

```
public class B extends A {
    public void f() { System.out.println("B.f"); }
    public void g() { System.out.println("B.g"); }
}
```

```
public class Main {
    public static void main(String[] arg) {
        Int x = new B();
        x.f();
        x.g();
        x.h();
        A y = new B();
        y.f();
        y.g();
        y.h();
    }
}
```

(6 p)

Uppgift c på nästa sida

c)

Förklara varför koden ger ett kompileringsfel:

```
public class Game {
    private int score;
    private void startGame() {
        score = 0;
        // Details omitted
    }
    public static void main(String[] args){
        startGame();
    }
}
```

Refaktorera koden så att kompileringsfelet försvinner utan att ändra något i metoden `startGame`. Du får bara ändra i `main`.

(2 p)

Uppgift 4

Nedan finns ett fragment från ett studieadministrationsprogram

```
public class CourseAdmin {
    private boolean isCTH;
    public CourseAdmin() {
        isCTH = true;
    }
    public String getGrade(int marks) {
        if ( isCTH ) {
            if ( marks < 24 )
                return "U";
            if ( marks < 36 )
                return "3";
            if ( marks < 48 )
                return "4";
            return "5";
        } else {
            if ( marks < 30 )
                return "U";
            if ( marks < 48 )
                return "G";
            return "VG";
        }
    }
    public void setGU() { isCTH = false; }
    public void setCTH() { isCTH = true; }
}
```

Koden är inte särskilt skalbar. Vid utvidgning till andra betygssystem kommer villkorssatsen att växa och den booleska variabeln måste bytas mot en av mer ändamålsenlig typ. Refaktorera koden med designmönstret *State*.

(8 p)

Uppgift 5

Konstruera en klass `SearchFiles` som låter användaren söka efter textfiler som innehåller en viss sträng i en angiven filkatalog samt alla dess underkataloger. Varje gång den sökta strängen påträffas i en fil skall filens absoluta filnamn, radnumret för sökträffen, samt den matchande raden skall skrivas ut. Klassen skall ha en `main`-metod och kunna köras som exemplet nedan visar.

Exempel: Givet är följande filträd:

```
/foo/bar/A.java
/foo/bar/temp/B.java
/foo/C.java
```

Filerna `A.java`, `B.java` och `C.java` innehåller bara nonsenskod som i sig inte är relevant för uppgiften:

```
public class A {
    public static float f (float w) {
        return w*Math.PI;
    }
}
```

```
public class B {
    public float cost(float weight) {
        return weight*42;
    }
}
```

```
public class C {
    public static float size(String what) {
        return what.length();
    }
    public static int zero() { return 0; }
}
```

Om vi i rotkatalogen / kör kommandot

```
> java SearchFiles foo static
```

skall utskriften bli:

```
foo\bar\A.java 2:     public static float f (float w) {
foo\C.java 2:     public static float size(String what) {
foo\C.java 5:     public static int zero() { return 0; }
```

Tips: Utnyttja klassen `File` i lösningen. Problemet är en instans av designmönstret *Composite.*

(8 p)

Uppgift 6

Gränssnittet `MiniComparator` motsvarar en miniversion av Javas standardklass `Comparator`:

```
public interface MiniComparator<T> {
    int compare(T a,T b);
    MiniComparator<T> reversed();
}
```

Konstruera klassen `LexOrder` så att den implementerar `MiniComparator`. Klassen skall kunna användas för att jämföra strängar. Metoden `compare` skall ge vanlig bokstavsordning. Metoden `reversed` skall returnera ett komparatorobjekt som ger den omvända ordningen. I följande exempel får vi anta att sorteringsmetoden

```
public static void sort(String[] a,MiniComparator comp)
```

och printmetoden redan är skrivna. Om man exekverar raderna

```
String[] a = {"ccc","bb","d","bbb","aaaaa","cc","a"};
sort(a,new LexOrder());
print(a);
sort(a,(new LexOrder()).reversed());
print(a);
sort(a,(new LexOrder()).reversed().reversed());
print(a);
```

skall utskriften bli

```
a aaaaa bb bbb cc ccc d
d ccc cc bbb bb aaaaa a
a aaaaa bb bbb cc ccc d
```

Tips: Utnyttja en nästlad klass för att implementera metoden `reversed`. Det yttre objektet kommer man åt med syntaxen <*yttre klass*>`.this`, ex. `LexOrder.this`.

(7 p)

Uppgift 7

I ett IT-system för en web-shop hanteras information om olika produkter i varulagret. Produkter beskrivs av gränssnittet

```
public interface Product {
   String getId();
   String getDescription();
   float getPrice();
   void setPrice(float price);
}
```

Metoden `getId` returnerar produktens artikelnummer, `getDescription` dess beskrivning samt `getPrice` priset. Med `setPrice` kan man ändra priset.

a)
Konstruera klassen `MyProduct` så att den implementerar gränssnittet `Product`. Inför lämpliga instansvariabler. Klassen skall ha en konstruktor med parametrar så att nya produktobjekt kan initieras med produktid och beskrivning. Dessutom skall klassen ha metoderna `equals` och `hashCode`. Två produktobjekt skall vara lika om deras artikelnummer (id) är lika, annars olika.

(6 p)

I systemet skall också finnas ett objekt som håller reda på lagersaldot för alla produkter. När man köper in en produkt ökas saldot och när man säljer minskas det. Följande gränssnitt beskriver denna information:

```
public interface Store {
   void buy(Product p,int n)
   void sell(Product p,int n) throws IllegalArgumentException;
   int getBalance(Product p) throws IllegalArgumentException;
   float getValue(Product p) throws IllegalArgumentException;
   float getTotalSold();
}
```

Metoden `buy` registrerar att `n` st `p` har köpts in till lagret, metoden `sell` att `n` st `p` har sålts, metoden `getBalance` returnerar antalet `p` i lager, metoden `getValue` ger det totala lagervärdet för produkten `p` och slutligen returnerar `getTotalSold` försäljningssumman för alla sålda produkter. `IllegalArgumentException` kastas av `sell`, `getBalance` och `getValue` om den angivna produkten är okänd, samt av `sell` om angivet antal är negativt.

b)
Konstruera klassen `MyStore` så att den implementerar gränssnittet `Store`. Inför lämpliga instansvariabler och en konstruktor. *Tips:* Lagra informationen i en map av lämpligt slag.

(8 p)

Exempel: På nästa sida visas ett exempel på hur klasserna ovan kan användas:

```
public static void main(String[] arg) {
   Product apple = new MyProduct("123456","Red apple");
   apple.setPrice(25.0);
   Product banana = new MyProduct("987654","Banana");
   banana.setPrice(39.0);
   Store store = new MyStore();
   store.buy(apple,100);
   store.buy(banana,200);
   store.sell(apple,50);
   store.sell(banana,25);
   System.out.println("Apple(balance): " + store.getBalance(apple));
   System.out.println("Apple(value): " + store.getValue(apple));
   System.out.println("Banana(balance): " + store.getBalance(banana));
   System.out.println("Banana (value): " + store.getValue(banana));
   System.out.println("Total sold: " + store.getTotalSold());
}
```

Utskriften då `main` exekveras blir:

```
Apple (balance): 50
Apple (value): 1250.0
Banana (balance): 175
Banana (value): 6825.0
Total sold: 2225.0
```

Lösningsförslag till tentamen

Kurs	**Objektorienterad programutveckling, fk**
Tentamensdatum	**2019-08-21**
Program	**TKIEK-2,TKTFY-3,TKTEM-3**
Läsår	**2018/2019, lp 2**
Examinator	**Uno Holmer**

Uppgift 1 (5 p)

Arvet bryter mot Liskov's substitutionsprincip. Man skall kunna byta objekt av subtyp till `List` mot varandra utan att drabbas av oväntade resultat men så är inte fallet här. Oavsett vilket objekt referensen pekar på har man enligt listkontraktet rätt att förvänta sig att objektet har ett 0:e element. Här är två situationer som visar på problemet. Om användaren tror att `l` pekar på en `ArveList` förväntas `l.get(1)` returnera `"foo"` men undantaget `IndexOutOfBoundsException` kastas istället:

```
List<String> l = new ArrayList<>();
l.add("foo");
```

Om användaren tror att `l` pekar på en `ArrayList` förväntas anropet `l.get(0)` returnera `"bar"` men ett undantag kastas istället:

```
List<String> l = new ArveList<>();
l.add("bar");
```

Uppgift 2 (6 p)

```
public class Model extends Observable {
    private int state;
    public Model() {}
    public void changeState() {
        // commands changing the state
        setChanged();
        notifyObservers(state);
    }
    public int getState() {
        return state;
    }
}
public class Gui extends JFrame implements Observer {
    private Model model;
    public Gui(Model model) {
        this.model = model;
    }
    public void showState(int state) {
        // display state
    }
    public void update(Observable o,Object obj) {
        if (o instanceof Model && obj instanceof Integer) {
            showState((Integer)obj);
        }
        // Alternatively, we can fetch the state using the access method:
       showState(model.getState());
    }
}
```

```
public class Main {
    public static void main(String [] arg) {
        Model model = new Model();
        Gui gui = new Gui(model);
        model.addObserver(gui);
        model.changeState();
    }
}
```

Uppgift 3 (4+6+2 p)

a)
B: g, h, i
C: g, h, i
D: g, h
E: g

b)
Variabeln `x` får här statisk typ `Int` och dynamisk typ `B`. Endast metoder som är definierade i `Int`, eller ev. basgränssnitt kan anropas via `x`, i det här fallet `f`:

```
Int x = new B();
x.f();           // Utskrift: B.f
x.g();           // Kompileringsfel: g ej känd i Int
x.h();           // Kompileringsfel: h ej känd i Int
```

Variabeln `y` får här statisk typ `A` och dynamisk typ `B`. Bara metoder som är kända i `A`, eller dess basklasser kan anropas via `y`, i det här fallet `f`, `g` och `h`. Metoden `f` implementeras i `B`, `g` överskuggas i `B` och `h` ärvs ifrån `A`:

```
A y = new B();
y.f();           // Utskrift: B.f
y.g();           // Utskrift: B.g
y.h();           // Utskrift: A.h
```

c)

En klassmetod får inte anropa en instansmetod med ett *internt* anrop. I ett sådant anrop anges inget objekt. Då JVM exekverar programmet anropas `main` utan att något `Game`-objekt skapas och därför existerar inte instansvariabeln `score`. Inget hindrar dock att vi skapar ett `Game`-objekt i `main` och anropar `startGame` med ett externt metodanrop på objektet.

```
public class Game {
    private int score;
    private void startGame() {
        score = 0;
        // Details omitted
    }
    public static void main(String[] args){
        Game g = new Game();
        g.startGame();
    }
}
```

Uppgift 4 (8 p)

```
public interface Grader {
    String getGrade(int marks);
}
public class CTHGrader implements Grader {
    public String getGrade(int marks) {
        if ( marks < 24 )
            return "U";
        if ( marks < 36 )
            return "3";
        if ( marks < 48 )
            return "4";
        return "5";
    }
}

public class GUGrader implements Grader {
    public String getGrade(int marks) {
        if ( marks < 30 )
            return "U";
        if ( marks < 48 )
            return "G";
        return "VG";
    }
}

public class CourseAdmin {
    private Grader CTHGrader,GUGrader;
    private Grader currentGrader;
    public CourseAdmin() {
        CTHGrader = new CTHGrader();
        GUGrader = new GUGrader();
        currentGrader = CTHGrader;
    }
    public void setGU() {
        currentGrader = GUGrader;
    }
    public void setCTH() {
        currentGrader = CTHGrader;
    }
    public String getGrade(int marks) {
        return currentGrader.getGrade(marks);
    }
}
```

Uppgift 5 (8 p)

```
public class Search {
    public static void main(String[] arg) {
        if ( arg.length != 2 ) {
            System.out.println(
                 "Usage: java Search <directory> <string>");
            System.exit(0);
        }
        try {
            searchFileTree(new File(arg[0]),arg[1]);
        }
        catch ( IOException e) {
            e.printStackTrace();
        }
    }

    private static void searchFileTree(File file,String pattern)
    throws IOException
    {
        if ( file.isFile() )
            searchInFile(file,pattern);
        else if ( file.isDirectory() )
            for ( File f : file.listFiles() )
                searchFileTree(f,pattern);
    }

    private static void searchInFile(File file,String pattern)
    throws IOException
    {
        Scanner sc = new Scanner(file);
        int lineNo = 1;
        while ( sc.hasNextLine() ) {
            String line = sc.nextLine();
            if ( line.contains(pattern) )
                System.out.println(file.getPath() + " " + lineNo +
                                   ": " + line);
            lineNo++;
        }
        sc.close();
    }
}
```

Uppgift 6 (7 p)

Metoden `reversed` är en fabriksmetod.

```
public class LexOrder implements MiniComparator<String> {
    private final MiniComparator<String> reverseComp;

    public LexOrder() {
        reverseComp = new LexOrderReversedComparator();
    }

    public int compare(String s1,String s2) {
        return s1.compareTo(s2);
    }

    public MiniComparator<String> reversed() {
        return reverseComp;
    }

    private class LexOrderReversedComparator
    implements MiniComparator<String>
    {
        public int compare(String s1,String s2) {
            return s2.compareTo(s1);
        }
        public MiniComparator<String> reversed() {
            return LexOrder.this;
        }
    }
}
```

Uppgift 7 (6+8 p)

a)

```
public class MyProduct implements Product {
    private final String id;
    private final String description;
    private float price;

    public MyProduct(String id,String description) {
        this.id = id;
        this.description = description;
        price = 0.0F;
    }

    @Override
    public String getId() {
        return id;
    }

    @Override
    public String getDescription() {
        return description;
    }

    @Override
    public float getPrice() {
```

```
        return price;
    }

    @Override
    public void setPrice(float price) {
        this.price = price;
    }

    @Override
    public final boolean equals(Object o) {
        if ( this == o )
            return true;
        if ( o instanceof MyProduct ) {
            MyProduct other = (MyProduct)o;
            return id.equals(other.id);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }
}
```

b)

```
public class MyStore implements Store {
    private Map<Product,Integer> store;
    private float totalSold;

    public MyStore() {
        store = new HashMap<>();
        totalSold = 0;
    }

    @Override
    public int getBalance(Product p) throws IllegalArgumentException{
        if ( ! store.containsKey(p) )
            throw new IllegalArgumentException("Unknown product");
        return store.get(p);
    }

    @Override
    public void buy(Product p,int n) {
        if ( ! store.containsKey(p) )
            store.put(p,n);
        else
            store.put(p,getBalance(p) + n);
    }
```

```
    @Override
    public void sell(Product p,int n)
    throws IllegalArgumentException
    {
        int balance = getBalance(p);
        if ( balance < n )
            throw
               new IllegalArgumentException(
                        "amount exceeds store balance");
        store.put(p,balance-n);
        totalSold += n*p.getPrice();
    }

    @Override
    public float getValue(Product p)
    throws IllegalArgumentException
    {
        return p.getPrice()*getBalance(p);
    }

    @Override
    public float getTotalSold() {
        return totalSold;
    }
}
```