DAT043 – Objektorienterad programmering

Tentamen, exempeltenta 2017

De tre första uppgifterna utgår från uppgifter i exempeltenta 3 och tentan i TDA547 som gick mars 2015.
Ansvarig lärare: Fredrik Lindblad. Besöker tentamenssalarna efter cirka 1 resp. cirka 3 timmar.
Maxpoäng: 60. Betygsgränser: 24 för 3:a, 36 för 4:a, 48 för 5:a. Längd: 4 timmar. Tentamen består av 6 uppgifter och 4 sidor.
Tillåtna hjälpmedel: Den dubbelsidiga lathund som finns tillgänglig på kurssidan och upptryck vid tentamen. Inga egna anteckningar på detta blad.

Implementeringar ska skrivas i Java. Oväsentliga syntax- och namnfel ger inte poängavdrag.
Skriv tydligt och välstrukturerat. Kommentera koden där det inte är uppenbart vad som sker. Svårförståeliga lösningar kan underkännas helt eller ges poängavdrag.
Lösningar som är onödigt krångliga eller inte följer god programmeringsstil, dels allmänt och dels med tanke på idéerna med objektorienterad programmering, kan underkännas helt eller ges poängavdrag.
Om det inte uttryckligen står motsatsen i uppgiften kan du använda klasser och metoder i Java:s API.
Om det inte uttryckligen står motsatsen i uppgiften får du definiera egna hjälpmetoder.
Importeringar av klasser i Java:s API behöver inte skrivas ut.
Svara inte på flera uppgifter på samma blad. Om en uppgift har deluppgift så svara gärna på flera deluppgifter på samma blad.
Efter rättning är tentamen tillgänglig vid expeditionen på plan 4 i EDIT-huset och kan granskas där. Den som vill diskutera rättningen kan, inom tre veckor efter att resultatet har rapporterats, kontakta ansvarig lärare och boka tid för ett möte. Notera att tentamen i så fall inte ska tas med från expeditionen.

---

1. (a) Du kan anta att klassen `Person` finns, är felfri och är synlig. Vilket påstående är då korrekt om klassen `C` som är definierad nedan? Om du svarar i, ii eller iii krävs att du motiverar ditt svar!

```
import java.util.*;
public class C {
  private LinkedList<Person> list;
  private String name;
  public C(String name) { this.name = name; }
  public void add(Person p) { list.add(p); }
}
```

   i. Exekveringen riskerar att avbrytas om objekt av klassen används.
   ii. Klassen går inte att kompilera.
   iii. Det går inte att skapa objekt av klassen.
   iv. Inget av ovanstående problem.

(2p)

   (b) Klassen `Value` ser ut på följande sätt:

```
public class Value {
  private int x;
  public Value() { x = 0; }
  public void set( int x ) { this.x = x; }
  public int get() { return x; }
}
```

   Vilken utskrift ger kodavsnittet nedan?

```
List<Value> list = new LinkedList<>();
Value i = new Value();
for ( int j = 0; j < 3; j++ ) list.add( i );
int k = 3;
for ( Value x : list ) x.set( k-- );
for ( Value x : list ) System.out.println( x.get() );
```

(2p)

(c) I ett program har man lagt följande rader:

```
int x = 8;
String sa = "AB8";
String sb = sa;
String sc = "AB" + x;
if (sa==sc) {
  System.out.println("lika");
} else {
  System.out.println("olika");
}
if (sa==sb) {
  System.out.println("lika");
} else {
  System.out.println("olika");
}
```

Ange vilken utskrift programmet ger och förklara varför.

(2p)

2. (a) Implementera metoden

```
public static boolean isIncreasing(int[] a)
```

som returnerar `true` om `a` innehåller en växande talföljd, d.v.s. om varje tal i `a` är minst lika stort som föregående tal. Om `a` inte innehåller en växande talföljd så ska metoden returnera `false`. Elementen i `a` ska inte ändras av metoden.

(6p)

(b) Implementera en klass `Increasing` som går att köra och som tar ett antal tal som programargument och avgör om de utgör en växande talföljd. Man ska t.ex. kunna exekvera programmet och få resultat enligt följande:

```
> java Increasing 1 2 3 3
true
```

I denna deluppgift får du anta att metoden i deluppgift a finns implementerad, oavsett om du gjort deluppgift a.

(6p)

3. Skriv ett program `Find` som läser en befintlig textfil och söker efter de rader i filen som innehåller en viss sträng. De funna raderna skall kopieras till en ny fil. Indata till programmet skall ges som argument på kommandoraden. (Programmet skall alltså inte läsa indata från `System.in`.) När man startar programmet skall man på kommandoraden ge följande tre argument: namnet på den befintliga filen, namnet på den nya filen samt den text man söker. Man kan t.ex. ge kommandot

   ```
   java Find bilar.txt volvo.txt Volvo
   ```

   Här kommer alla de rader i filen `bilar.txt` vilka innehåller texten `Volvo` att kopieras till filen `volvo.txt`. Programmet skall kontrollera att antalet argument är korrekt och att filerna går att öppna. Om något skulle vara fel skall en felutskrift ges och programmet avslutas.

   (10p)

4. Du ska implementera en klass, `Person`, som representerar en person. Varje `Person`-instans har ett namn, som är en `String`, och ett antal barn. Varje barn är associerat med ett index. Från början har en instans inga barn. Man kan senare lägga till barn, ett och ett. Första barnet som läggs till får index 0, andra får index 1, osv.

   Klassen ska ha en konstruktor som tar en sträng som argument och sätter personens namn till denna sträng. Klassen ska vidare ha följande metoder:

   - `public void addChild(Person child)` som lägger till en person som barn till aktuell person.
   - `public String getName()` som returnerar personens namn.
   - `public int getNChildren()` som returnerar antal barn.
   - `public Person getChild(int idx)` som returnerar barnet med index `idx`. Otillåtna index behöver inte hanteras.
   - `public boolean isDescendantOf(Person n)` som avgör om aktuell person är ättling till `n`, d.v.s. är barn eller barnbarn eller barnbarnsbarn ...

   Du får själv välja vilka instansvariabler din klass ska ha. Använd gärna klasser i Java collections framework.

   Exempel på användning av klassen och vad metoderna ska returnera:

   ```
   Person p1 = new Person("Arne");
   Person p2 = new Person("Ulla");
   Person p3 = new Person("Greta");
   Person p4 = new Person("Anna");
   p1.addChild(p2);
   p2.addChild(p3);
   p2.addChild(p4);
   System.out.println(p2.getNChildren());  // 2
   System.out.println(p3.getName());  // Greta
   System.out.println(p2.isDescendantOf(p1));  // true
   System.out.println(p3.isDescendantOf(p1));  // true
   System.out.println(p1.isDescendantOf(p2));  // false
   System.out.println(p4.isDescendantOf(p4));  // false
   System.out.println(p3.isDescendantOf(p4));  // false
   ```

   (10p)

5. Implementera en metod

```
public static int[] merge(int[] a, int[] b)
```

som slår ihop två sorterade (i växande ordning) arrayer till en sorterad array. Metoden kan alltså anta att `a` och `b` är sorterade. Den skapa en ny array som fylls med alla element som finns i `a` och `b` och den nya arrayen ska vara sorterad. Metoden ska returnera den nya arrayen och inte ändra element i `a` och `b`. Om t.ex. `a` är `{1,3,7}` och `b` är `{2,3,5,9}` så ska metoden returnera en array med elementen `{1,2,3,3,5,7,9}`. Klasser och metoder i Java collections framework får inte användas.

(12p)

6. Implementera två generiska statiska metoder för att beräkna unionen och snittet av mängder. Metoden `union` ska ta två argument, `a` och `b`, av typen `Set<E>`, där `Set` är Java collection frameworks interface för mängder och `E` är en typparameter. Metoden ska skapa och returnera en ny mängd (ett objekt av typen `Set<E>`) som innehåller alla element som finns antingen i `a` eller `b`. De två ursprungsmängderna, `a` och `b`, ska vara oförändrade.

   Metoden `intersection` ska ha samma signatur men istället för unionen istället skapa en mängd som utgör snittet, d.v.s. innehåller alla element som finns både i `a` och `b`.

   Du ska definiera dessa två metoder inklusive signaturen d.v.s. det som står före metodkroppen (det inom `{ }`).

   Du kan använda implementeringen `HashSet` när du skapar den nya mängden.

   Tips: Interfacet `Set` utökar `Collection` som utökar `Iterable`.

(10p)