DAT043 – Objektorienterad programmering för D, DIT011 – Objektorienterad programvaruutveckling för GU

lösningsförslag till tentamen 2017-03-11

1. (a) Vad skrivs ut om man exekverar metoden `g` i programkoden nedan?

```
public static void f(int[] x, int y, Integer z) {
  int sum = 0;
  for (int i = 0; i < x.length; i++) {
    x[i] += y;
    sum += x[i];
  }
  y = sum + z;
  z = x.length;
}
public static void g() {
  int[] a = {2};
  int b = 7;
  Integer c = 3;
  f(a, b, c);
  System.out.println(a[0] + " " + b + " " + c);
}
```

svar:

```
9 7 3
```

(3p)

(b) Givet följande gränssnitt och klasser:

```
interface A {
}
public class B implements A {
}
public class C extends B {
}
```

Vilka rader i följande kodutsnitt innehåller fel?

```
A x1 = new A();
B x2 = new B();
C x3 = new B();
B x4 = new C();
A x5 = new C();
A x6 = x2;
B x7 = x5;
C x8 = (C)x2;
C x9 = (C)x4;
```

Ange vilken eller vilka rader som innehåller fel och för varje felaktig rad om felet uppstår vid kompilering eller exekvering av programmet. Använd numret på variabeln som deklareras när du anger rader, d.v.s. ett tal mellan 1 och 9.

svar:

Kompileringsfel på rad 1,3 och 7. Exekveringsfel på rad 8.

(3p)

2. (a) Implementera metoden

`public static int maxDiff(int[] a)`

som ska returnera största absolutbeloppet av skillnaden mellan två efterföljande tal i `a`. Om t.ex. `a` är `{1,3,-2,2,5,7,1,4,2}` så är absolutbeloppet av skillnaderna $2, 5, 4, 3, 2, 6, 3, 2$ och metoden ska returnera `6`. Om arrayen har mindre än två element ska metoden returnera `-1`. Innehållet i `a` ska inte ändras av metoden.

svar:

```
public static int maxDiff(int[] a) {
  int maxdiff = -1;
  for (int i = 1; i < a.length; i++) {
    int diff = Math.abs(a[i] - a[i-1]);
    if (diff > maxdiff) maxdiff = diff;
  }
  return maxdiff;
}
```

(6p)

(b) Implementera metoden

`public static void transpose(double[][] a)`

som transponerar en kvadratisk matris representerad av `a`. Du kan anta att `a.length > 0` och `a.length == a[i].length` för alla tillåtna `i`. Metoden returnerar inget, utan `a` ska ändras till att vara transponeringen av sig själv. Ingen ny, tillfällig array ska skapas av metoden. Bara primitiva variabler får deklareras. Transponering av en matris innebär spegling kring diagonalen. Som exempel, matrisen

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \text{transponerad blir} \quad \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

svar:

```
public static void transpose(double[][] a) {
  int m = a.length;
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < i; j++) {
      double t = a[i][j];
      a[i][j] = a[j][i];
      a[j][i] = t;
    }
  }
}
```

(6p)

3. Skriv en klass som implementerar ett program `Diff` som jämför innehållet i två textfiler rad för rad. När man startar programmet skall man på kommandoraden ge två argument; filnamnen för de två filer som ska jämföras. Man kan t.ex. ge kommandot

   ```
   java Diff fil1.txt fil2.txt
   ```

   Programmet ska jämföra rad 1 i ena filen med rad 1 andra filen, rad 2 i ena filen med rad 2 i andra, o.s.v.. För varje rad `n` där de inte är precis lika så ska programmet skriva ut "`Line n is different`". Den ska alltså ange radnumret. Första raden räknas som rad 1. Om den ena filen har fler rader än den andra ska programmet på slutet skriva ut "`Different number of lines`".

   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.

   svar:

```
public class Diff {
  public static void main(String[] args) {
    if (args.length != 2) {
      System.out.println("Wrong number of arguments.");
      return;
    }
    Scanner s1 = null, s2 = null;
    try {
      s1 = new Scanner(new File(args[0]));
      s2 = new Scanner(new File(args[1]));
    } catch (IOException e) {
      if (s1 != null) s1.close();
      System.out.println("Could not open one of the files.");
      return;
    }

    int line = 1;
    while (s1.hasNextLine() && s2.hasNextLine()) {
      String str1 = s1.nextLine();
      String str2 = s2.nextLine();
      if (!str1.equals(str2)) {
        System.out.println("Line " + line + " is different.");
      }
      line++;
    }
    if (s1.hasNextLine() || s2.hasNextLine()) {
      System.out.println("Different number of lines.");
    }
    s2.close();
    s1.close();
  }
}
```

(10p)

4. Givet är följande gränssnitt som implementeras av heltalsuttryck vars värde kan beräknas:

```
interface Exp {
  int compute();
}
```

Du ska skriva tre klasser, `Add`, `Neg` och `Lit` som alla implementerar `Exp`. `Add` representerar ett additionsoperation, $a + b$, `Neg` en negation, $-a$, och `Lit` en literal, d.v.s. en konkret heltal, t.ex. 384. Klasserna ska ha följande konstruktorer (en per klass):

```
Add(Exp oper1, Exp oper2)
Neg(Exp oper)
Lit(int val)
```

Konstruktorn för `Add` ska alltså ta två uttryck som argument. Dessa motsvarar de två operanderna i operationen. Konstruktorn för `Neg` ska ta ett uttryck; den enda operanden. Konstruktorn för `Lit` ska ta det heltal som objektet ska representera. För att skapa objekt som motsvarar uttrycket $(-(4 + 7)) + 14$ kan man skriva

```
Exp e = new Add(new Neg(new Add(new Lit(4), new Lit(7))), new Lit(14));
```

Resultaten av att anropa `e.compute()` ska vara `3`.

svar:

```
class Add implements Exp {
  private Exp oper1, oper2;
  Add(Exp oper1, Exp oper2) {
    this.oper1 = oper1;
    this.oper2 = oper2;
  }
  public int compute() {
    return oper1.compute() + oper2.compute();
  }
}
class Neg implements Exp {
  private Exp oper;
  Neg(Exp oper) {
    this.oper = oper;
  }
  public int compute() {
    return -oper.compute();
  }
}
class Lit implements Exp {
  private int val;
  Lit(int val) {
    this.val = val;
  }
  public int compute() {
    return val;
  }
}
```

(10p)

5. Du har följande klass för att representera heltalsintervall:

```
public class Interval {
  public final int start, end;
  public Interval(int start, int end) {
    if (start >= end) throw new RuntimeException();
    this.start = start; this.end = end;
  }
}
```

Implementera en klass `Room` som representerar ett rum som kan bokas vid olika tillfällen. Varje instans ska innehålla en uppsättning bokningar som gjorts för rummet.

För enkelhets skull ska heltal användas för att representera tidpunkter. För en viss tidpunkt motsvaras den av det heltal som är antalet sekunder sedan kl 00:00 1 januari 1970. (Vad starten för tideräkningen är spelar ingen roll för uppgiften.)

Klassen `Room` ska ha en konstruktor utan argument. När ett `Room`-objekt skapas så har den inga bokningar. Varje bokning sträcker sig från en starttid och en sluttid, båda mätta i sekunder enligt ovan. Ett `Interval`-objekt används för att representera en bokning.

Klassen ska ha en metod

```
public boolean book(Interval b)
```

där argumentet är ett intervall som representerar tidsintervallet för den önskade bokningen. Om bokningen är möjlig ska den läggas till rummets bokningar och metoden ska returnera `true`. Om den inte är möjlig ska metoden returnerna `false` och inte påverka uppsättningen av bokningar. En bokning är möjlig när tidsintervallet inte överlappar tidsintervallet för någon av de redan existerande bokningarna.

Klassen ska också ha en metod

```
public Interval book(int fromTime, int duration)
```

där `fromTime` är första tänkbara tid i sekunder för den önskade bokningens starttid och `duration` är dess längd i sekunder. Metoden ska alltså hitta det tidigaste intervall som är möjligt att boka, har längden `duration` och börjar vid tiden `fromTime` eller senare. Detta intervall ska läggas in som en ny bokning och returneras av metoden.

Du kan använda metoder och klasser i Java:s standard-API, inklusive Java collections framework.

svar:

```
class Room {

  private List<Interval> bookings = new ArrayList<>();

  public Room() {
  }

  public boolean book(Interval b) {
    int i;
    for (i = 0; i < bookings.size(); i++) {
      if (bookings.get(i).start < b.end && bookings.get(i).end > b.start) {
        return false;
      }
      if (bookings.get(i).start >= b.end) {
        break;
      }
    }
    bookings.add(i, b);
    return true;
  }

  public Interval book(int fromTime, int duration) {
    int i;
    for (i = 0; i < bookings.size(); i++) {
      if (bookings.get(i).end >= fromTime) {
        if (bookings.get(i).start <= fromTime) {
          fromTime = bookings.get(i).end;
          i++;
        }
        break;
      }
    }
    for (; i < bookings.size(); i++) {
      if (bookings.get(i).start >= fromTime + duration) {
        break;
      }
      fromTime = bookings.get(i).end;
    }
    Interval b = new Interval(fromTime, fromTime + duration);
    bookings.add(i, b);
    return b;
  }
}
```

(12p)

6. Implementera den generiska metoden

```
public static <E> E mostFrequent(E[] a)
```

Metoden ska returnera ett objekt vars värde förekommer flest gånger i `a`. För att avgöra om två element i `a` har samma värde ska instansmetoden `equals` användas. Om t.ex. `a` är `{"X", "Y", "BA", "Y", "Z", "D", "BA"}` så ska metoden returnera `"Y"` eller `"BA"` för de förekommer båda två gånger och alla andra strängar förekommer färre gånger (en gång). Notera att metoden ska vara generisk och alltså hantera annat än strängar som element i `a`. Den ska ha precis den signatur som anges ovan. Om `a` är tom så ska metoden returnera `null`.

Du kan anta att metoderna `equals` och `hashCode` är implementerade på ett tillfredsställande sätt för varje konkret typ som metoden används med. Du kan använda metoder och klasser i Java:s standard-API, inklusive Java collections framework.

svar:

```
public static <E> E mostFrequent(E[] a) {
  Map<E, Integer> freq = new HashMap<>();

  for (E x : a) {
    if (freq.containsKey(x)) {
      freq.put(x, freq.get(x) + 1);
    } else {
      freq.put(x, 1);
    }
  }

  int maxfreq = 0;
  E maxelt = null;
  for (E x : freq.keySet()) {
    if (freq.get(x) > maxfreq) {
      maxfreq = freq.get(x);
      maxelt = x;
    }
  }

  return maxelt;
}
```

(10p)