Tentamen Programmeringsteknik I 2012-06-04

Skrivtid: 1400-1700
Hjälpmedel: Java-bok

Tänk på följande

- Det finns en referensbok (Java) hos tentavakten som du får gå fram och läsa men inte ta tillbaka till bänken.
- Skriv läsligt! Använd *inte* rödpenna!
- Skriv bara på framsidan av varje papper.
- Lägg uppgifterna i ordning. Skriv uppgiftsnummer och pin-kod (eller namn om du saknar sådan) på alla papper. Skriv inte längst upp i vänstra hörnet - det går inte att läsa där efter sammanhäftning.
- Fyll i försättssidan fullständigt.
- Det är principer och idéer som är viktiga. Skriv så att du övertygar examinator om att du har förstått dessa även om detaljer kan vara felaktiga.
- Programkod skall vara läslig dvs den skall vara vettigt strukturerad och indenterad. Namn på variabler, metoder, klasser etc skall vara beskrivande men kan ändå hållas ganska korta.
- Det är totalt 30 poäng på skrivningen. Betygsgränser: 15 ger säkert 3, 21 ger säkert 4, 26 ger säkert 5.

Lycka till!

Tom

Uppgifter

1. För vart och ett av punkterna A till H skall du ange det alternativ 1 - 14 som passar. Ange bara *ett* alternativ! Om du tycker att flera alternativ passar så välj det som passar bäst!

A. lokal variabel

B. instansvariabel

C. formell parameter

D. aktuell parameter

E. `while`-sats

F. omslagsklass

G. primitiv datatyp

H. villkorsuttryck (boolskt uttryck)

1. Måste ha en konstruktor
2. `double`
3. `true`
4. `String`
5. Deklareras i en metodkropp
6. `ArrayList`
7. Deklareras i ett metodhuvud
8. En iterationskonstruktion
9. array
10. En lyssnare
11. Kan ha olika värden för olika objekt
12. Anges vid metodanrop
13. `x = 4`
14. Ingen beskrivning passar bra

(4p)

2. Nedan finns en klass `Clock` som representerar en klocktid i timmar, minuter och sekunder. Konstruktorn sätter initialvärden på tiden. Metoden `tick()` ökar klockslaget med en sekund. Metoden skall också hantera minut- och timgränser så att minut- och sekundvärdena alltid ligger mellan 0 och 59. Se körexemplet! `main`-metoden testar klassen genom att stega fram ett antal gånger och hämtar och skriver klockslagen vid lite olika tidpunkter.

   a) Skriv klar konstruktorn (2p)

   b) Skriv klar metoden `tick` (4p)

   c) Vilka värden har instansvariablerna efter det sista, felaktiga, konstruktoranropet? (2p)

Koden:

```
public class Clock {
  private int sec;
  private int min;
  private int hour;

  public Clock(int hour, int min, int sec) {
    if (min<0 || min>=60 || sec<0 || sec>=60 || hour<0) {
     System.out.println("Illegal initial values for Clock");
    } else {
       // Uppgift a
    }
  }

  public int getSec() { return sec; }
  public int getMin() { return min; }
  public int getHour() { return hour; }
  public void tick() { /* Uppgift b */}

  public static void main(String[] args) {
    Clock c = new Clock(0, 59, 57);
    for (int i = 1; i<=100; i++) {
      c.tick();
      if ( i<6|| i%13==0)
        System.out.format("%d:%02d:%02d\n", c.getHour(), c.getMin(), c.getSec());
    }
    System.out.format("%d:%02d:%02d\n", c.getHour(), c.getMin(), c.getSec());
    c = new Clock(0, -10, 123);
    // Uppgift c: Vilka värden har instansvariablerna nu?
  }
}

Output:

0:59:58
0:59:59
1:00:00
1:00:01
1:00:02
1:00:10
1:00:23
1:00:36
1:00:49
1:01:02
1:01:15
1:01:28
1:01:37
Illegal initial values for Clock
```

Anm: Det är inte nödvändigt att förstå hur `format`-metoden fungerar. Den hämtar tim-, minut- och sekundvärden med hjälp av `get`-metoderna och skriver sedan ut dessa värden (med inledande nolla vid behov)

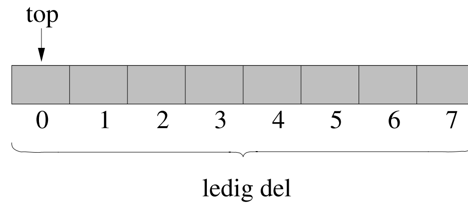
3. En *stack* är en mekanism för att lagra och hämta poster på så sätt att den senast lagrade, ännu ej uttagna, posten är den som står på tur att tas ut. Det är alltså en slags kö där den som kommit sist får lämna först. Man kan lika det vid en trave där man alltid både lägger och hämtar överst. (Ett annat namn är LIFO som står för "Last In, First Out". En vanlig kö kallas med andra ord en "FIFO" — First In, First Out.)

   Nedan finns (delar av) en klass som implementerar en stack med hjälp av en *array*. För enkelhetens skull lagrar denna stack heltal – i verkligheten har man vanligen andra datatyper. I koden finns också en `main`-metod och angiven output.

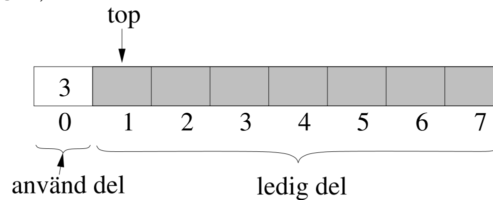
```
import java.util.Scanner;

public class Stack {

  private int [] stack; // För att lagra värdena
  private int top;      // Antal lagrade värden tillika index för första lediga plats

  /** Skapa en stack med plats för n värden */
  public Stack(int n) {  /* Uppgift a */ }

  /** Lagra ett nytt värde överst på stacken */
  public void push(int v) { /* Uppgift b */ }

  /** Hämta översta värdet från stacken och ta bort det */
  public int pop() { /* Uppgift c */  }

  /** Lagra ett värde överst på stacken. Utöka arrayen vid behov */
  public void safePush(int v) { /* Uppgift d */ }

  public String toString() {
    String ret = "";
    for (int i= 0; i<top; i++)
      ret = ret + stack[i] + " ";
    return "[ " + ret + "]";
  }

  /** Returnera antalet lagrade element tillika index för första lediga plats */
  public int size() {
    return top;
  }

  public static void main(String[] arg) {
    Scanner sc = new Scanner(System.in);
    Stack s = new Stack(8);                               // Se fig 1
    s.push(3);
    System.out.println("Efter en push           : " + s);  // Se fig 2
    s.push(7); s.push(5); s.push(8);
    System.out.println("Efter ytterligare 3 push: " + s);  // Se fig 3
    System.out.println("Första pop ger värdet   : " + s.pop());
    System.out.println("och stackutseendet      : " + s);  // Se fig 4
    System.out.print("Poppa resten: ");
    while (s.size()>0)
      System.out.print(s.pop() + " ");
    System.out.println();
  }
}

/** Output:
Efter en push           : [ 3 ]
Efter ytterligare 3 push: [ 3 7 5 8 ]
Första pop ger värdet   : 8
och stackutseendet      : [ 3 7 5 ]
Poppa resten: 5 7 3
*/
```

   Förklarande figurer finns på nästa sida.

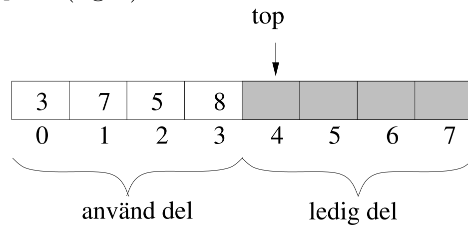
Arrayens utseende efter konstruktorn (fig 1):



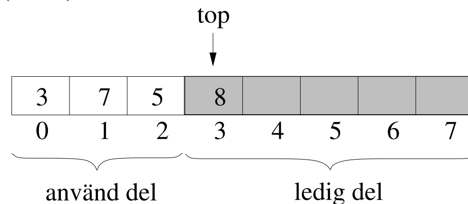
efter första `push` (fig 2):



efter ytterligare tre `push` (fig 3):



och efter första `pop` (fig 4):



(Värdet 8 finns kvar i arrayen men inte i den använda delen .)

a) Implementera konstruktorn `Stack(int n)` så att den skapar en stack med plats för `n` tal. (2p)

b) Skriv klar metoden `push(int v)` som lagrar `v` överst på stacken. Om stacken är full skall en felutskrift ges och inget lagras. (3p)

c) Skriv klar metoden `pop()` som tar bort det översta värdet från stacken och returnerar det ("tar bort" betyder att värdet inte skall returneras igen - det går ju inte att fysiskt ta bort array-emenetet). Ge felutskrift om operationen inte går att utföra! (3p)

d) Skriv klar metoden `safePush(int v)` som gör som `push` men, vid behov, utökar stacken med en faktor 2. I exemplet innebär det att stacken vid det nionde anropet av `safePush` utökas till 16 platser. Det som redan finns lagrat skall givetvis finnas med i den utökade stacken.

(4p)

4. En köteoretiker som är lite osäker på teorin vill experimentera med hjälp av tidsstegad datorsimulering. Vid varje tidsteg är sannolikheten a ($0 < a < 1$) att det anländer ett objekt till kön. Oberoende av detta försöker man vid varje tidssteg med sannolikheten b ($0 < b < 1$) ta ut en ett objekt ur kön. (Om kön är tom går det naturligtvis inte ta ut något.) Teoretikern vill veta hur många objekt som kommit ut ur kön och hur lång tid de i genomsnittligt tillbringat i kön.

   Nedan finns klassen `Queue` som representerar en kö med objekt av typen `Qelem`. Dokumentationen saknas dock varför man bör läsa koden för att förstå vad den gör.

   Skriv en klass `Simulation` med en `main`-metod. Metoden skall läsa in två sannolikheter a och b, skapa ett `Queue`-objekt och genomföra simuleringen i 500 tidssteg. Vid vart hundrade tidssteg skall programmet skriva hur många objekt som tagits ut samt hur lång tid dessa i genomsnitt har tillbringat i kön. Se exempelkörningen!

   Klasserna `Queue` och `Qelem`:

```
import java.util.ArrayList;

public class Queue {
  private ArrayList<Qelem> q;

  public Queue() {
    q = new ArrayList<Qelem>();
  }

  public void put(Qelem e) {
    q.add(e);
  }

  public Qelem get() {
    Qelem e = q.get(0);
    q.remove(0);
    return e;
  }

  public int size() {
    return q.size();
  }
}
```

```
public class Qelem {
  private int bornTime;

  public Qelem(int born) {
    bornTime = born;
  }

  public int getBorn() {
    return bornTime;
  }
}
```

   Exempel på körresultat:

```
Ankomstsannolikhet:  0.5
Uttagssannolikhet :  0.5
Tidssteg: 100
   Antal element ut: 49
   Genomsnittstid  : 3.3061225
Tidssteg: 200
   Antal element ut: 88
   Genomsnittstid  : 11.261364
Tidssteg: 300
   Antal element ut: 142
   Genomsnittstid  : 14.697183
Tidssteg: 400
   Antal element ut: 188
   Genomsnittstid  : 20.122341
Tidssteg: 500
   Antal element ut: 234
   Genomsnittstid  : 25.41453
```

(6p)