Institutionen för TENTAMEN
Data- och informationsteknik 2018-06-08

**PROGRAMMERINGSTEKNIK, del B**
**TIN213**

*OBS!* Det kan finnas kurser med samma eller liknande namn på olika utbildningslinjer. Denna tentamen gäller *endast* för den eller de utbildningslinjer som anges ovan. Kontrollera därför noga att denna tentamen gäller för den utbildningslinje du själv går på.

TID: 08:30 - 11:30

---

Ansvarig: Robin Adams, tel. 031 772 63 48

Betygsgränser: Sammanlagt maximalt 30 poäng.
På tentamen ges graderade betyg:
3:a 15 poäng, 4:a 20 poäng, 5:a 25 poäng

Hjälpmedel: Skansholm, *Java direkt med Swing*, Studentlitteratur, eller
Bravaco, Simonson, *Java Programming: From the Ground up*
(Understrykningar och mindre anteckningar i boken är tillåtna.)

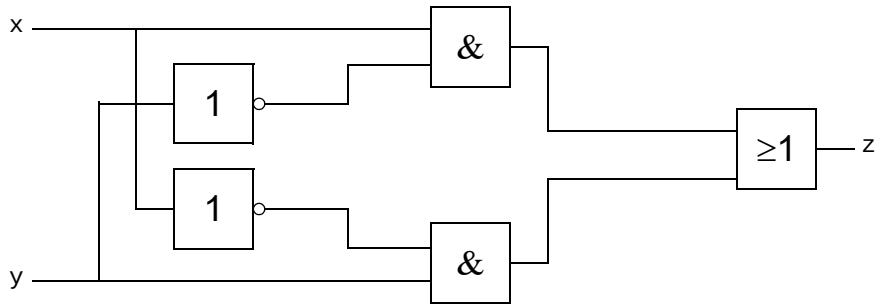
Inga kalkylatorer är tillåtna.

Tänk på:

- att skriva tydligt och disponera papperet på ett lämpligt sätt.
- att börja varje ny uppgift på nytt blad. Skriv endast på en sida av papperet
- Skriv den (anonyma) kod du fått av tentamensvakten på *alla* blad.

De råd och anvisningar som givits under kursen skall följas vid programkonstruktionerna. Det innebär bl.a. att onödigt komplicerade, långa och/eller ostrukturerade lösningar i värsta fall ej bedöms.

Uppgift 1) När man konstruerar digitala system utgår man som bekant från enkla elektroniska kretsar som brukar kallas *grindar* (*gates*). Grindarnas in- och utsignaler är elektriska spänningar på "låg nivå" eller "hög nivå" (**`false`** resp. **`true`**). De enklaste grindarna avbildar på elektronisk väg de logiska funktionerna "icke", "och" och "eller" och kallas därför NOT-grindar, AND-grindar resp. OR-grindar. Ett exempel på ett nät som använder sådana grindar visas i figuren nedan. (Grindarna markerade med symbolen 1 är NOT-grindar, de med symbolen & AND-grindar och den med beteckningen ≥1 en OR-grind.)




Konstruera en grupp klasser som beskriver de grundläggande grindarna samt konstanta insignaler. Du skall skapa en abstrakt superklass `Gate` och utgå från denna. Klassen `Gate` skall ha en abstrakt subklass `BasicGate` vilken i sin tur skall vara superklass till de tre klasserna `NotGate`, `AndGate` och `OrGate`. Dessutom skall du konstruera en klass `ConstantGate` som beskriver konstanta insignaler till nätet. Denna klass skall vara direkt subklass till klassen `Gate`.

När man ritar upp komplicerade nät som innehåller grindar måste man kunna namnge signalerna och grindarna. Därför skall klassen `Gate` ha en instansvariabel som innehåller grindens (eller insignalens) namn. Dessutom skall alla de klasser du konstruerar ha en konstruktor som har ett namn som parameter.

Superklassen `Gate` skall ha följande metoder:

- `getName`, ger namnet som resultat,
- `getOutput`, abstrakt metod som skall ge utsignalen (en **`boolean`**) från den aktuella grinden (eller insignalens värde),

Klassen `ConstantGate` används för att beskriva konstanta insignaler till de nät man vill koppla upp. (Man kan tänka sig en insignal som en "låtsasgrind" som alltid genererar en konstant utsignal.) I klassen `ConstantGate` måste man förstås omdefiniera metoden `getOutput`. Dessutom skall klassen `ConstantGate` ha en metod som heter `set`. Denna skall ha en parameter av typen **`boolean`** som anger den konstanta insignalens värde.

Varje objekt av klassen `BasicGate` skall innehålla en lista med referenser till de grindar som är kopplade till ingångarna till den aktuella grinden. Klassen `BasicGate` skall ha följande metoder av vilka vissa kan vara abstrakta:

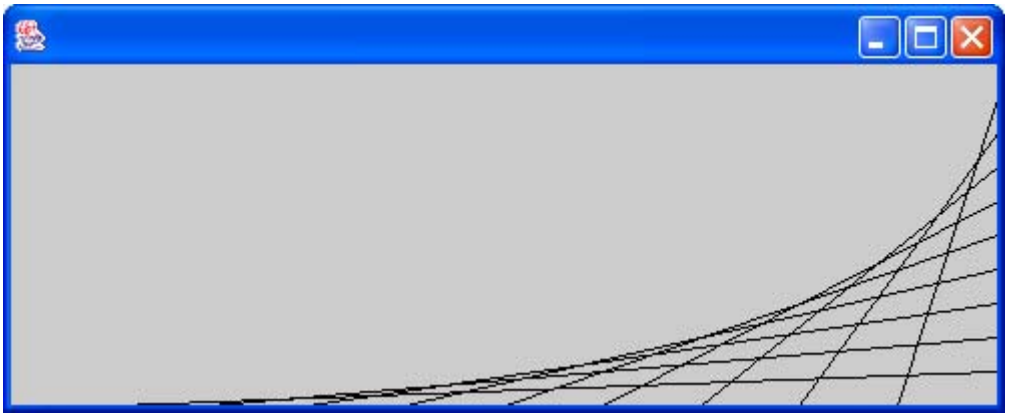
- `setInput`, får som parameter en referens `g` av typen `Gate`, kopplar utsignalen från `g` som insignal till den aktuella grinden,
- `checkInput`, kontrollerar att den aktuella grinden har korrekt antal insignaler. (AND- och OR-grindar måste ha minst två insignaler och NOT-grindar måste ha exakt en insignal.) Om insignalerna är felaktiga skall en exception av typen `IllegalStateException` genereras. Lägg in ett lämpligt felmeddelande.
- `calculateValue` förutsätter att antalet insignaler är korrekt och beräknar och returnerar utsignalens värde utgående från insignalerna,

- `getOutput`, ger utsignalen från den aktuella grinden (anropar först `checkInput` och sedan `calculateValue`).

I klasserna `AndGate`, `OrGate` och `NotGate` skall du överskugga den eller de ärvda metoder som behövs för att grindarna skall ge rätt resultat.

(10 p)

Uppgift 2) Fönstret i figuren nedan innehåller en enda grafisk komponent vilken fyller ut hela fönstret. Denna komponent är av klassen `Lines`. Klassen `Lines` är en subklass till standardklassen `JPanel`. Klassen `Lines` ritar linjer så som visas i figuren. Avstånden mellan linjernas startpunkter är lika stora. Detta gäller också för linjerna slutpunkter. Antalet linjer som skall ritas anger man som parameter till konstruktorn för klassen `Lines`. (I figuren är det 9 linjer men detta kan alltså variera.) Om storleken på en komponent av typen `Lines` ändras (fönstrets storlek kan t.ex. ändras i figuren) så ändrar sig linjerna så att de ändå fyller ut hela komponenten. Din uppgift är nu naturligtvis att skriva klassen `Lines`. (Du behöver alltså inte skriva hela programmet). Tänk på att det kan finnas en ram (`Border`) runt en `Lines`-komponent. Det måste du förstås ta hänsyn till när du beräknar start- och ändpunkterna för linjerna.



(10 p)

Uppgift 3) Antag att klassen `Flight` är definierad på följande sätt:

```
import java.text.*;
import java.io.*;
public class Flight implements Serializable, Comparable<Flight> {
  private String no, destination;
  private String time;             // avgångstid (med formen hh:mm)

  public Flight (String n, String d, String t) {
    no = n; destination = d; time = t;
  }

  public String getNumber() {
    return no;
  }

  public String getDestination() {
    return destination;
  }

  public String getTime() {
    return time;
  }

  public int compareTo(Flight f) {
    Collator c = Collator.getInstance();
    c.setStrength(Collator.PRIMARY);
    return c.compare(time+destination, f.time+f.destination);
  }
}
```

Konstruera en klass `Airport`. För varje objekt av klassen skall det finnas tre instansvariabler: flygplatsens namn, en mängd `departures` med alla flighter som avgår från den aktuella flygplatsen samt en avbildningstabell `flightsTo`. Mängden `departures` skall vara sorterad i första hand på avgångstid och i andra hand på destinationens namn. I tabellen `flightsTo` skall söknycklarna vara namnen på destinationerna och värdena mängder innehållande flighter. Man skall alltså i tabellen `flightsTo` kunna slå upp vilka flighter som avgår till en viss destination. Varje värdemängd i `flightsTo` skall vara sorterad på avgångstid.

Eftersom det finns både en mängd med alla avgångar och en tabell med avgångar till varje destination kommer en viss flight att ingå i två mängder. Men detta är inget konstigt eftersom det är referenserna till flighterna som ligger i mängderna.

Klassen `Airport` skall ha en konstruktor som får flygplatsens namn som parameter. Dessutom skall följande metoder finnas:

- `getName`, ger flyplatsens namn,
- `getDepartures()`, ger en sorterad mängd innehållande alla flighter som avgår från en aktuella flygplatsen,
- `getDepartures(dest)`, ger en sorterad mängd innehållande alla flighter som avgår till `dest` från en aktuella flygplatsen (`dest` är en `String`),
- `addFlight(f)`, lägger till flighten `f` både till mängden `departures` och till aktuell mängd i tabellen `flightsTo`,
- `removeFlight(f)`, tar bort flighten `f` både från mängden `departures` och från aktuell mängd i tabellen `flightsTo`. Om detta innebär att en viss avbildning i tabellen `flightsTo` kommer att innehålla en tom mängd så skall denna avbildning tas bort.

När du konstruerar metoderna `getDepartures` behöver du för enkelhets skull inte ta hänsyn till att det kan vara farligt att returnera referenser.

(10 p)

```
// L?sningar till tentamen, del B 2018-06-08

// Uppgift 1
import java.util.*;

public abstract class Gate {
  protected String name;

  public Gate(String name) {
    this.name = name;
  }

  public abstract boolean getOutput();

  public String getName() {
    return name;
  }
}

class ConstantGate extends Gate {

  private boolean value;

  public ConstantGate(String name) {
    super(name);
  }

  public void set(boolean v) {
    value = v;
  }

  public boolean getOutput() {
    return value;
  }

}

abstract class BasicGate extends Gate {
  protected List<Gate> input = new ArrayList<Gate>();

  protected BasicGate(String name) {
    super(name);
  }

  protected void checkInput() {
    if (input.size() < 2)
      throw new IllegalStateException("Gate " + name + ": Input signal(s)
missing");
  }

  protected abstract boolean calculateValue();

  public void setInput(Gate g) {
    input.add(g);
  }

  public boolean getOutput() {
    checkInput();
    return calculateValue();
  }
}
```

```
class AndGate extends BasicGate {

  public AndGate(String name) {
    super(name);
  }

  protected boolean calculateValue() {
    for (Gate g : input)
      if (!g.getOutput())
        return false;
    return true;
  }
}

class OrGate extends BasicGate {

  public OrGate(String name) {
    super(name);
  }

  protected boolean calculateValue() {
    for (Gate g : input)
      if (g.getOutput())
        return true;
    return false;
  }
}

class NotGate extends BasicGate {

  protected NotGate(String name) {
    super(name);
  }

  @Override
  protected void checkInput() {
    if (input.size() != 1)
      throw new IllegalStateException("Gate " + name + ": Illegal number of
input signals");
  }

  protected boolean calculateValue() {
    return !input.get(0).getOutput();
  }
}

 // Uppgift 2

     public class Lines extends JPanel {
       private int n;

       public Lines(int n) {
         this.n = n;
       }

       public void paintComponent(Graphics g) {
         super.paintComponent(g);
         Insets i = getInsets();
         int w = getWidth()  - i.left-i.right;
         int h = getHeight() - i.top - i.bottom;
```

```
          int dx = w/(n+1);
          int dy = h/(n+1);
          for (int j = 1; j <= n; j++)
            g.drawLine(i.left+j*dx, i.top+h, i.left+w, i.top+h-j*dy);
        }
      }

       // Uppgift 3

    import java.util.*;
    public class Airport {
      private String name;
      private NavigableSet<Flight> departures = new TreeSet<Flight>();
      private Map<String, NavigableSet<Flight>> flightsTo = new
HashMap<String, NavigableSet<Flight>>();

      public Airport(String n) {
        name = n;
      }

      public String getName() {
        return name;
      }

      public NavigableSet<Flight> getDepartures() {
        return departures;
      }

      public NavigableSet<Flight> getDepartures(String to) {
        return flightsTo.get(to);
      }

      public void addFlight(Flight f) {
        departures.add(f);
        if (!flightsTo.containsKey(f.getDestination()))
          flightsTo.put(f.getDestination(), new TreeSet<Flight>());
        flightsTo.get(f.getDestination()).add(f);
      }

      public void removeFlight(Flight f) {
         departures.remove(f);
         NavigableSet<Flight> e = flightsTo.get(f.getDestination());
         if (e != null) {
           e.remove(f);
           if (e.isEmpty())
             flightsTo.remove(f.getDestination());
         }
      }
    }
```