TENTAMEN: Objektorienterad programmering

Läs detta!

- *Uppgifterna är inte ordnade efter svårighetsgrad.*
- Börja varje uppgift på ett nytt blad (utom uppg. 1).
- Numrera och 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!**
- Programkod som finns i tentamenstesen behöver ej upprepas.
- Programkod skall skrivas i Java 5 (eller senare) och vara indenterad och snyggt uppställd.
- Du behöver ej skriva importdeklarationer.
- Onödigt komplicerade lösningar ger poängavdrag.
- Omotiverad användning av klassvariabler och klassmetoder ger poängavdrag.
- Givna deklarationer, parameterlistor etc. får ej ändras.
- 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

Välj **ett** alternativ för varje fråga! Garderingar ger noll poäng. Inga motiveringar krävs. Varje korrekt svar ger 2.5 poäng.

1. Om man vill lagra heltal i en fil kan man göra på olika sätt. Ett är att lagra talen som de är i en binär fil. Ett annat sätt är att lagra talens sifferrepresentationer i en textfil. Para ihop lämpliga komponenter till höger för läsning av tal från resp. filtyp.

```
1) Läsning av tal av typen int från en binärfil.
2) Läsning av tal från en textfil
```

```
A. DataInputStream
B. InputStreamReader
C. FileReader
D. FileInputStream
E. Integer.parseInt()
F. BufferedReader
```

   a. 1: A+B, 2: C+E
   b. 1: B+D, 2: B+D
   c. 1: A+D, 2: C+E+F
   d. 1: A+D, 2: D+E+F
   e. 1: A+C, 2: C+E+F

2. Undantagsklasserna `E`, `F` och `G` samt klasserna `Base` och `Sub` definieras

```
public class E extends Exception {}
public class F extends E {}
public class G extends F {}

public class Base {
    public void f() throws F {
        ...
    }
}
public class Sub extends Base {
    @Override
    public void f() throws ...
}
```

   En av nedanstående kan inte vara en överskuggning av `f` i `Sub`, vilken?

   a. `public void f() throws F,G {}`
   b. `public void f() throws F {}`
   c. `public void f() throws G {}`
   d. `public void f() throws E {}`

3. Vilket designmönster har en framträdande roll i organiseringen av strömklasserna i Javas API?
   a. Factory method
   b. Singleton
   c. Composite
   d. Decorator
   e. Observer

4. Vi vill använda metoden `swap` för att låta listelementen på plats `i` och `j` byta innehåll med varandra. Vilken/vilka implementation(er) av `swap` är korrekt(a)?

```
public static void swap1(String s1,String s2) {
    String temp = s1;
    s1 = s2;
    s2 = temp;
}

anrop:
ArrayList<String> list = new ArrayList<String>();
...
swap1(list.get(i),list.get(j));
```

```
public static void swap2(List<String> l,int i,int j) {
    String temp = l.get(i);
    l.set(i,l.get(j));
    l.set(j,temp);
}

anrop:
ArrayList<String> list = new ArrayList<String>();
...
swap2(list,i,j);
```

a. endast `swap1` är korrekt, den kan dessutom användas för annat än listelement
b. endast `swap2` är korrekt
c. ingen är korrekt
d. båda är korrekta

(10 p)

Uppgift 2

Konstruera en klass med namnet `DieButton` som subklass till `JButton`. När man trycker på en `DieButton` i fönstret skall tärningen visa ett nytt slumpmässigt värde. Varje objekt av klassen skall fungera som en självständig tärning. Ex. Ett fönster med tre tärningar:

För att sätta lämpliga bilder på tärningen beroende på vilket tärningsvärde som skall visas finns till din hjälp den färdiga klassen

```
public class DieIcons {
    public static DieIcons getInstance() { ... }
    public int size() { ... }
    public ImageIcon getIcon(int i)
        throws IndexOutOfBoundsException { ... }
}
```

Observera att klassen konstruerats enligt Singleton-mönstret. Den hämtar automatiskt ikonerna från bildfiler. Vi går inte in närmare på hur klassen är implementerad.

- `getInstance` ger tillgång till ett objekt av klassen.
- `size` returnerar antalet olika ikoner som objektet har.
- `getIcon(i)` returnerar en ikon med `i` ögon.
  Tärningen skall ha lika många sidor som det finns ikoner. (8 p)

Uppgift 3

En klassisk fransk kortlek har fyra färger (suit) och 13 valörer (rank) av varje kort. Följande gränssnitt beskriver ett minimum av metoder:

```
public interface FrenchCard {
    enum Suit {HEARTS,SPADES,CLUBS,DIAMONDS}
    int getRank();
    Suit getSuit();
}
```

Vidare finns följande klass, som vi inte beskriver närmare:

```
public class Card implements FrenchCard,Cloneable,Comparable<Card>
```

Uppgiften är nu att konstruera två metoder (se nedan) i klassen `CardHand` för att representera en *hand* av kort. En hand är en samling kort ur en eller flera kortlekar. Klassen har bl.a. följande metoder:

`public void add(Card c)`	adderar kortet `c` till handen
`public int size()`	returnerar antalet kort i handen
`public Card look(int i) throws IndexOutOfBoundsException`	returnerar det i:te kortet i handen
`public boolean discard(int i) throws IndexOutOfBoundsException`	tar bort det i:te korten ur handen

Dessutom skall klassen ha metoderna `clone` som returnerar en djup kopia av handen, samt `equals` som avgör om två händer är lika. Två händer är lika om de innehåller lika många kort, samt lika många av varje kort. Den interna ordningen bland korten spelar ingen roll för om två händer är lika eller ej.
Ex.

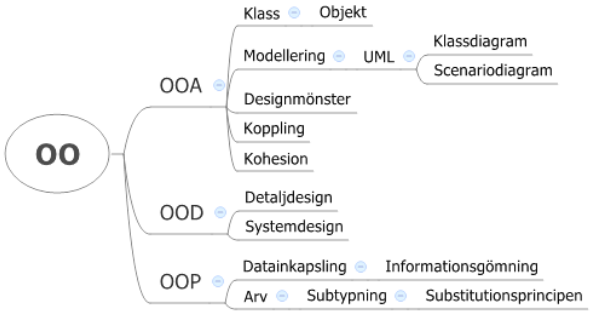
```
Card c1 = new Card(4,FrenchCard.Suit.DIAMONDS);
Card c2 = new Card(12,FrenchCard.Suit.SPADES);
Card c3 = new Card(9,FrenchCard.Suit.HEARTS);
CardHand h1 = new CardHand();
h1.add(c1); h1.add(c2); h1.add(c3);
CardHand h2 = new CardHand();
h2.add(c3); h2.add(c1); h2.add(c2);
System.out.println(h1.equals(h2));   // true
System.out.println(h2.equals(h1));   // true
```

Du kan anta att handen är representerad som en lista av kort. Implementera metoderna `clone` och `equals`. Ingen av metoderna får ändra kortens inbördes ordning. *Tips*: Utnyttja `clone` och `Collections.sort` i `equals`!

(10 p)

Uppgift 4

För att visualisera samband mellan olika begrepp använder man ibland en s.k. *mind map*, vilken består av noder med text i, sammanbunda med streck och pilar. Ofta utgår man från ett centralt begrepp (topic) och lägger till noder i olika riktningar för begrepp som på något sätt kan associeras till begreppet. Ex. En trädformad mind map:



Vi skall konstruera en klass för att bygga sådana träd. Klassen skall implementera gränssnittet

```
public interface MindMap extends Iterable<MindMap> {
    void setDescription(String description);
    String getDescription();
    MindMap addSubtopic(String description);
    boolean hasSubtopics();
    MindMap getSubtopic(int index) throws IndexOutOfBoundsException;
    MindMap getParent();
    void print();
}
```

Klassens skall heta `ListMindMap`. Inför lämpliga instansvariabler. Använd en lista av lämplig typ för att lagra underbegrepp till en nod. Man skall även kunna nå föräldern till en nod (utom för startnoden). För att spara tid behöver du inte skriva `set`- och `get`-metoderna eller `hasSubtopics`. Konstruktorn skall ta en sträng med begreppet som inparameter. Metoden `print` skall ge en utskrift så att varje begrepp indenteras till en nivå som motsvarar begreppets djup i trädet. Begreppen skall också numreras med hierarkiska index enligt exemplet till höger nedan. *Tips*: låt `print` anropa en privat rekursiv hjälpmetod som tar indenteringsnivån och en sträng med indexet som parameter, ex. `"1.2.3"`.

```
MindMap m1 = new ListMindMap("OO");
MindMap m11 = m1.addSubtopic("OOA");
MindMap m111 = m11.addSubtopic("Klass");
m111.addSubtopic("Objekt");
MindMap m112 = m11.addSubtopic("Modellering");
MindMap m1121 = m112.addSubtopic("UML");
m1121.addSubtopic("Klassdiagram");
m1121.addSubtopic("Scenariodiagram");
m11.addSubtopic("Designmönster");
m11.addSubtopic("Koppling");
m11.addSubtopic("Kohesion");
MindMap m12 = m1.addSubtopic("OOD");
m12.addSubtopic("Detaljdesign");
m12.addSubtopic("Systemdesign");
MindMap m13 = m1.addSubtopic("OOP");
MindMap m131 = m13.addSubtopic("Datainkapsling");
m131.addSubtopic("Informationsgömning");
MindMap m132 = m13.addSubtopic("Arv");
MindMap m1321 = m132.addSubtopic("Subtypning");
m1321.addSubtopic("Substitutionsprincipen");
m1.print();
if ( m11.hasSubtopics() )
    for ( MindMap m : m11 )
        System.out.println(m.getDescription());
```

```
1 OO
   1.1 OOA
      1.1.1 Klass
         1.1.1.1 Objekt
      1.1.2 Modellering
         1.1.2.1 UML
            1.1.2.1.1 Klassdiagram
            1.1.2.1.2 Scenariodiagram
      1.1.3 Designmönster
      1.1.4 Koppling
      1.1.5 Kohesion
   1.2 OOD
      1.2.1 Detaljdesign
      1.2.2 Systemdesign
   1.3 OOP
      1.3.1 Datainkapsling
         1.3.1.1 Informationsgömning
      1.3.2 Arv
         1.3.2.1 Subtypning
            1.3.2.1.1 Substitutionsprincipen
Klass
Modellering
Designmönster
Koppling
Kohesion
```

(10 p)

Uppgift 5

Studera följande klasser:

```
public interface I {
    void f1();
}
```

```
public abstract class C1 {
    public abstract void f2();
    public void f3(){ print("C1.f3");}
    protected void print(String s) {
        System.out.println(s);
    }
}
```

```
public class C2 extends C1 implements I {
    public void f1() { print("C2.f1"); }
    public void f2() { print("C2.f2"); }
}
```

```
public class C3 extends C2 {
    public void f2() { print("C3.f2"); }
    public void f4() { print("C3.f4"); }
    public void print(String s) {
        System.out.println("++"+s+"++");
    }
}
```

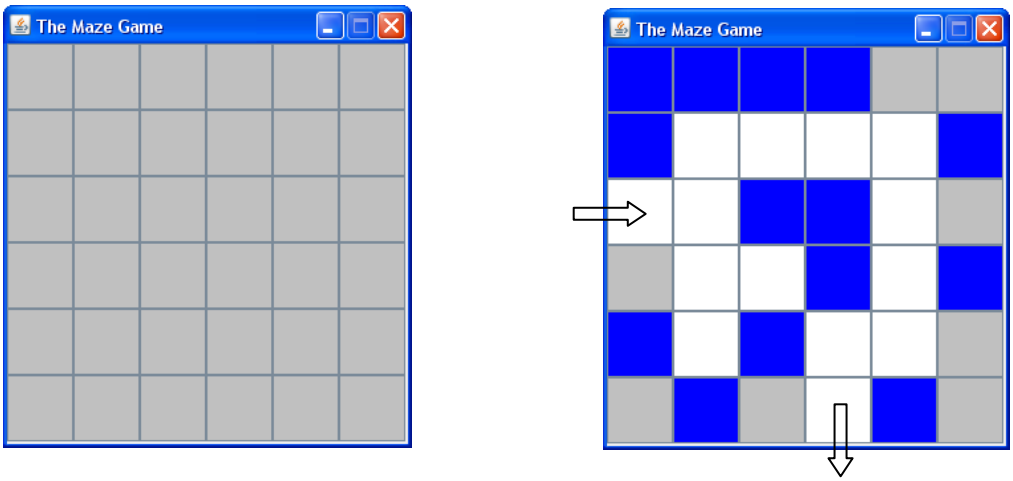
```
public class Main {
    public static void func(C2 obj) {
        obj.f1();
        obj.f2();
        obj.f3();
        obj.f4();
    }

    public static void func2(C3 obj) {
        obj.f4();
    }

    public static void main(String[] arg) {
        func(new C1());
        func(new C2());
        func(new C3());
        func2(new C2());
        func2(new C3());
    }
}
```

Tre metodanrop i klassen `Main` ger kompileringsfel. Vilka är de, och vad är det för fel på dem! Antag nu att vi avlägsnar de felaktiga satserna och kompilerar de fem klasserna. Vad skrivs då ut om `main` exekveras?

(10 p)

Uppgift 6

Ett labyrintspel går till på följande sätt: Spelaren skall i ett kvadratiskt fönster med kvadratiska celler försöka hitta kortast möjliga väg igenom med så få försök som möjligt. Alla cellerna är från början grå. En cell kan antingen vara ogenomtränglig vägg (svart), eller golv (vit) som man kan gå på. När spelaren klickar på en grå cell ”avtäcks” den och det blir synligt om den är golv eller vägg. En väg igenom labyrinten är en följd av vågräta eller lodräta vita celler från en kant till en annan. Labyrinten får antas innehålla minst en sådan väg.
Ex. Här har spelaren lyckats hitta en väg utan att klicka på alla celler:



Som uppgift skall du konstruera det grafiska gränssnittet (GUI) till ett labyrintprogram. Designen bygger på Model View Controller-arkitekturen. Modellklasserna består av `Cell` och `GameEngine`. GUI:t består av ett fönster och varje cell i modellen observeras av ett `CellView`-objekt. Allt enligt samma princip som i laboration 3. `GameEngine` är färdig och ser i sammandrag ut på följande sätt:

```
public class GameEngine {
    public GameEngine(int size) ...
    public void makeVisible(int cellId) ...
    public void addCellObserver(int cellId,Observer obs) ...
}
```

Parametern `size` anger labyrintens sidlängd (6 ovan). Klassen skapar `size*size` antal cellobjekt som numreras radvis från 0 och uppåt. Du får anta att konstruktorn konstruerar cellobjekt som motsvarar en korrekt labyrint, t.ex. en sådan som visas ovan.

En cell kan vara av två typer: golv eller vägg, samt synlig eller osynlig. En cell som gjorts synlig med `makeVisible` förblir synlig. `makeVisible` uppdaterar observatörerna (men den skickar ingen parameter). En uppdatering innebär alltid att cellen skall göras synlig i vyn.

```
public class Cell extends Observable {
    public enum Type {FLOOR,WALL}
    public enum State {VISIBLE,INVISIBLE}

    public Cell(int id,Type type) ...
    public Type getType() ...
    public void makeVisible() ...
}
```

v.g.v.

Uppgifter

a) Implementera klassen `CellView`. Klassen skall ärva från `JButton` och implementera gränssnittet `Observer`. Initialt visas komponenten med ljusgrå färg. Vid uppdatering visas vit eller svart färg, beroende på den observerade cellens typ.
Ex. `setBackground(Color.WHITE);`

(4 p)

b) Implementera klassen `Gui`. Klassens konstruktor skall ha signaturen

```
public Gui(GameEngine gameEngine,int size)
```

där `size` anger labyrintens sidlängd räknat i celler. Klassens skall skapa ett fönster och placera ut `size*size CellView`-komponenter i ett kvadratiskt mönster. Varje `CellView` skall ha en lyssnare som rapporterar positionen till spelmotorn.

(7 p)

c) Skriv en `Main`-klass med en enkel main-metod som skapar och kopplar ihop objekt av `GameEngine` och `Gui` på lämpligt sätt.

(1 p)