TENTAMEN: Objektorienterad programmering

Läs detta!

- *Uppgifterna är inte ordnade efter svårighetsgrad.*
- Börja varje uppgift på ett nytt blad.
- Skriv ditt idnummer 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 version) och vara indenterad och kommenterad.
- 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 två poäng. *Besvara direkt i tesen!*

1. Ett av villkoren är sant, vilket?

```
Integer i1 = new Integer(123), i2 = new Integer(123), i3 = i1;
```

   a. `i2.equals(i3)`
   b. `i1 == i2`
   c. `i2 == i3`

2. Vilket påstående är korrekt om `Register` på basis av vad som visas nedan?

```
public class Register
{
    private LinkedList<Person> list;
    private String name;
     ...

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

    public void add(Person p) { list.add(p); }
     ...
}
```

   a. klassen går ej att kompilera
   b. det går ej att skapa objekt av klassen
   c. exekveringen riskerar att avbrytas om objekt av klassen används
   d. klassen är felfri

3. Vilket påstående är korrekt?

   a. En klass med hög kohesion har låg koppling till andra klasser.
   b. En klass med hög koppling till andra klasser har låg kohesion.
   c. En klass med låg koppling har få kontakter med andra klasser.
   d. En klass med hög kohesion löser många olika problem.

4. Vilken aktivitet bör alltid utföras innan ett program byggs ut med ny funktionalitet?

   a. avlusning
   b. regressionstestning
   c. refaktorering

5. Vilken av satserna a eller b ger ett kompileringsfel?

```
public class A
{
    private int x;
    public int y;
}

public class B extends A
{
    public void f() {
        x = 10;                          // a
    }
}

...
B obj = new B();
obj.y = 11;                              // b
```

(10 p)

Uppgift 2

Rita ett UML-diagram som motsvarar följande java-kod:

```
interface C {
   void f();
}

public abstract class D implements C {
    public void f() { }
    public void g() {}
    public abstract void h();
}

public class E extends D {
    public void h() { }
}

public class F extends D {
    public void g() { }
    public void h() { }
}

public class G extends D {
    public void h() { }
}

public class B {
    private ArrayList<D> table;

    public B() {
        table = new ArrayList<D>();
        add(new E());
        add(new F());
        add(new G());
    }
    public void add(D x) { table.add(x); }
}

public class A {
    private B b;

    public A(B b) { this.b = b; }
}
```

Antag att följande objekt skapas

```
B b = new B();
A a1 = new A(b);
A a2 = new A(b);
 ...
A an = new A(b);
```
många!

För full poäng krävs att du använder rätt sorts pilar på rätt ställen. Multiplicitet skall anges om den inte är ett. Klassikonerna skall innehålla rätt metodnamn, men du behöver inte ta med konstruktorerna.

(5 p)

Uppgift 3

I ett lagerhanteringssystem hos företaget X sker med ojämna mellanrum uppdatering av lagerlistor med nya priser. I lagerlistan finns för varje artikel uppgift om artikelnummer, pris, samt antal artiklar i lager. Prislistan med aktuella priser innehåller för varje artikel uppgift om artikelnummer och pris. Exempel: Om lagerlistan till vänster uppdateras med de nya priserna till höger

Artikelnr	Pris	Lagersaldo
12345	100	23
78901	200	8
76563	300	1034
27276	400	199
99213	500	0

Artikelnr	Pris
99213	596
76563	1270
12345	120

så skall lagerlistan ha följande innehåll efter uppdateringen:

Artikelnr	Pris	Lagersaldo
12345	120	23
78901	200	8
76563	1270	1034
27276	400	199
99213	596	0

I uppgiften finns tre klasser som skall kompletteras: `Item` (prisuppgift i prislistan), `StoreItem` (information om en vara i lagerlistan), samt `Store` (lagerlistan). Se kodskelett på de följande sidorna. Fyll i dina lösningar direkt i tesen, eller på lösblad, om det inte får plats.

a) Implementera metoden `equals` i `Item` så att den returnerar sant om två objekt innehåller likadana artikelnummer, och falskt annars. Metoden skall ej vara överskuggningsbar vid arv.
(4 p)

b) Implementera metoden `hashCode` i `Item`.
(1 p)

c) Implementera metoden `toString` i `Item`. (se även uppgift g)
(1 p)

d) Definiera konstruktorn i `StoreItem`. Konstruktorn skall ha inparametrar motsvarande samtliga instansvariabler.
(1 p)

e) Implementera metoden `toString` i `StoreItem`. (se även uppgift g)
(1 p)

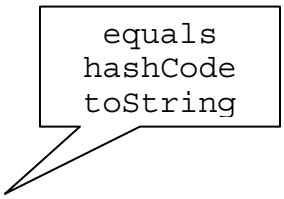
f) Implementera metoden `add` i `Store` som adderar ett nytt artikelobjekt till lagerlistan. Metoden skall bara lägga till objektet om det inte redan finns i listan. Returvärdet skall avspegla om objektet adderades eller ej.
(1 p)

g) Implementera metoden `update` i `Store`. Metoden skall uppdatera priset på alla artiklar i lagerlistan som finns i listan som ges som inparameter.
(4 p)

h) Implementera metoden `print` i `Store`. Varje rad i utskriften skall ha formen
artikelnr`:` *pris* `(`*antal*`)`
(1 p)

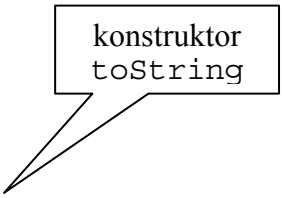
```
public class Item {
    private String productId;
    private int price;

    public Item(String productId,int price) {
        this.productId = productId;
        this.price = price;
    }
```




```
    public int getPrice() { return price; }
    public void setPrice(int price) { this.price = price; }
}
```

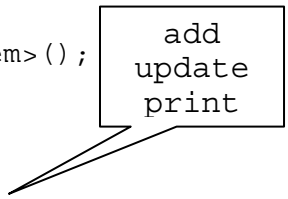
```
public class StoreItem extends Item {
    private int storeLevel = 0; // Number of items in the store
```




```
    public int getStoreLevel() { return storeLevel; }

    public void addToStore(int noOfItems) {
        storeLevel += noOfItems;
    }
}
```

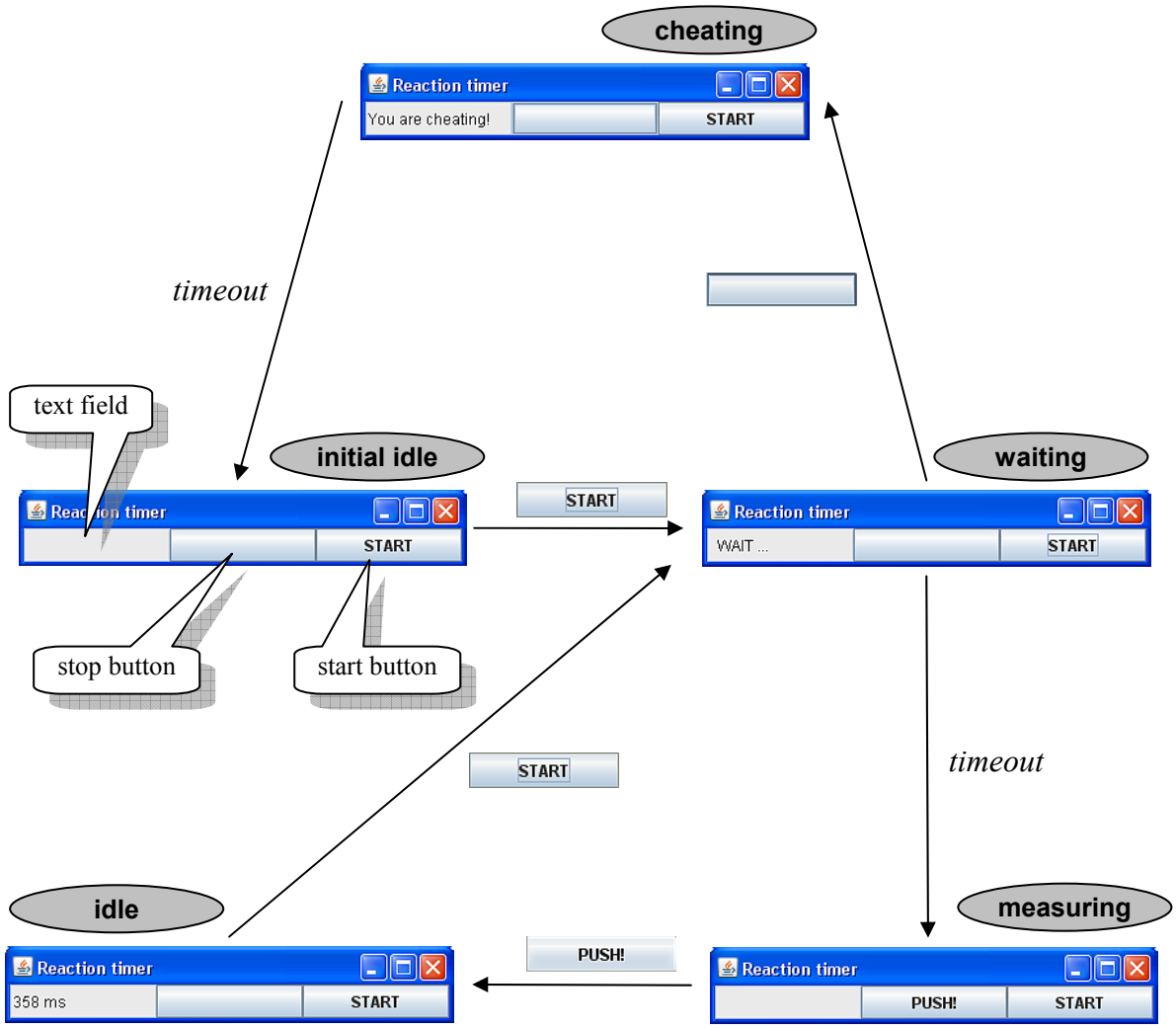
```
public class Store {
    private List<StoreItem> items = new ArrayList<StoreItem>();
```




```
}
```

Uppgift 4

Denna uppgift går ut på att programmera en reaktionstidmätare. Mätarens fönster, som visas nedan, skall ha ett textfält där den uppmätta reaktionstiden och meddelanden kan visas, samt två knappar. När man trycker på den högra knappen väntar programmet under ett slumpmässigt tidsintervall. Därefter tänds den vänstra knappen (**PUSH!** visas i knappen) samtidigt som tidmätning startar. Det gäller nu att trycka ner den vänstra knappen så snabbt som möjligt, varvid tidmätningen upphör. Tiden visas i textfältet. Om man fuskar och trycker ner den vänstra knappen under vänteperioden innan den vänstra knappen tänds visas texten **You are cheating!** i textfältet tills vänteperioden är slut. Inga knapptryckningar påverkar tillståndet under tiden. Figuren nedan visar de olika tillstånden som gränssnittet kan befinna sig i.



Uppgiften blir nu att programmera delar av programmet. För mätning av reaktionstid kan du använda den färdiga timerklassen

```
public class Timer {
    void start();          startar tidmätning
    void stop();           stoppar tidmätning
    void reset();          nollställer timern
    long getValue();       returnerar ackumulerad tid i millisekunder
}
```

En timer kan vara vilande eller gående. En timer kan ackumulera tid. Man kan alltså addera gångtid för flera mätperioder genom att starta och stoppa den omväxlande. Detta fortgår tills timern nollställs i viloläge. Timern har följande egenskaper:

- En nyskapad timer är nollställd och vilande.
- Anrop av `getValue` på en nollställd vilande timer returnerar 0.
- Efter anropet av `start` på en vilande timer blir timern gående.
- Anrop av metoderna `start` och `reset` har ingen effekt på en gående timer.
- Anrop av `stop` har ingen effekt på en vilande timer.
- Efter anrop av `reset` på en vilande timer nollställs timern.
- Tidsackumulering: Vid ett anrop av `stop` på en gående timer blir timern vilande. Om den senaste gångperioden varat tiden t och `getValue` skulle returnerat t' om den anropats före det senaste anropet av `start`, så skall `getValue` nu returnera $t + t'$.
- Om `getValue` anropas på en gående timer skall den returnera samma värde som om den anropats före det senaste anropet av `start`. Tiden för den senaste gångperioden adderas alltså inte förrän timern stoppas.

För att hantera den slumpmässiga vänteperioden finns ett färdigt javagränssnitt och en klass:

```
public interface Timeout {
    void timeout();
}

public class Alarm {
    public static void setTimeout(Timeout tm,int min,int max){ ... }
}
```

Exempel:

```
class MinKlass implements Timeout {
   ...
   public void timeout() { ... }
   ...
}
...
MinKlass obj = new MinKlass();
Alarm.setTimeout(obj,1000,5000);    // ms
```

Efter anropet ovan av `setTimeout` kommer `obj.timeout()` att anropas automatiskt efter en slumpmässig fördröjning på mellan 1 och 5 sekunder.

Konstruera det grafiska gränssnittet. Det finns kodskelett med plats för lösningen på de kommande sidorna.
Använd ett textfält och två knappar. Händelsehanteringen får göras valfritt som centraliserad eller med anonyma inre klasser. Tre olika händelser skall hanteras: 1. ett tryck på startknappen, 2. ett tryck på stoppknappen, 3. timeout. De fyra statiska konstanterna kan användas för att hålla reda på i vilket tillstånd mätaren befinner sig. Inför en lämplig instansvariabel för detta. Tips: Definiera tre metoder i klassen som tar hand om respektive händelse ovan. Anropa två av dessa i samband med händelsehanteringen för knapptryckningarna. Den tredje skall hantera timeout.

(12 p)

```
public class ReactionTimer

                          // ange ev. Arv ovan

{
    private static final int IDLE = 0;
    private static final int WAITING = 1;
    private static final int MEASURING = 2;
    private static final int CHEATING = 3;


    // instansvariabler









    // konstruktor









    // övrigt, se följande sidor ---->

}
```

```
    public void makeFrame()  {

    }
```

... mer ---->

Version: 2012-01-12 Objektorienterad programmering D2, DAT042, 11/12, lp 1

```
    // övriga metoder
```

Uppgift 5

Vilka av arven 1-7 är tillåtna i Java, vilka ger kompileringsfel? Motivera svaret!

```
a)
public interface I1 { ... }

1. public interface I2 extends I1 { ... }

2. public interface I3 extends I1 { ... }

3. public interface I4 extends I2, I3 { ... }

4. public class C1 implements I3 { ... }

5. public class C2 extends C1 implements I4 { ... }

6. public class C3 extends C1 { ... }

7. public class C4 extends C2, C3 { ... }
```

(7 p)

b)
Vissa av klasserna nedan måste deklareras som abstrakta, vilka? Vilka klasser kan instansieras? Motivera svaret!

```
public interface I1 {
    void f();
}

public interface I2 {
    void g();
}

public class C1 implements I1, I2  {
    public void f() { ... }
    public void h() { ... }
}

public class C2 implements I2  {
    public void g() { ... }
    abstract public void i();
}

public class C3 extends C1 {
    public void g() { ... }
}

public class C4 extends C2 {
    public void i() { ... }
}
```

(4p)

Uppgift 6

Ibland skulle det kunna vara praktiskt om man med en iterator kunde titta på nästa element utan att avancera iteratorn, som blir fallet om metoden `next` används. En (halvbra) analogi är metoden top som returnerar det översta elementet i en stack, utan att ta bort elementet. Man kan i princip utvidga en iteratorklass med en sådan metod, som vi kan kalla `peek`. Klassen skulle antingen kunna byggas ut med en metod till, eller den nya metoden placeras i en subklass. Ett problem uppstår emellertid med javas standardklasser eftersom de itererbara klasserna i `Java.util` inte avslöjar namnen på sina iteratorklasser. De hanteras ju alltid via gränssnittet `Iterator` och aldrig via sina namn. Vi kommer alltså inte ens åt dem för att ärva från dem. Dessutom skulle arv vara opraktiskt eftersom samma tillägg skulle behöva göras i samtliga iteratorklasser om den nya operationen skall finnas för alla iteratorer, vilket den bör göra av uniformitetsskäl. Lösningen på problemet blir att utnyttja designmönstret Decorator. Vi börjar med att definiera ett gränssnitt som specificerar vår ”tittoperation”:

```
public interface Lookaheadable<E> {
    E peek() throws NoSuchElementException;
}
```

Därefter kan vi definiera en dekoratorklass

```
public class LookAheadIterator<E>
      implements Iterator<E>,Lookaheadable<E>
```

Metoderna `hasNext` och `next` skall fungera som i `Iterator` (se tentamensbilagan) men `remove` skall ej implementeras i vår klass utan kasta undantaget `UnsupportedOperationException` om den anropas.

Exemplet nedan skriver ut `1 1 2 2 3 3`

```
ArrayList<Integer> l = new ArrayList<Integer>();
for ( int i = 1; i <= 3; i++ )
    l.add(i);

LookAheadIterator<Integer> it =
    new LookAheadIterator<Integer>(l.iterator());

while ( it.hasNext() )
    System.out.print(it.peek() + " " + it.next() + " ");
```

Skriv färdigt klassen `LookAheadIterator` genom att delegera till en ”vanlig” iterator i enlighet med Decorator-mönstret. Tips: Inför en instansvariabel av typ `E` för att buffra ett element i taget. Kasta lämpligt undantag om metoden i fråga ej kan utföra sin uppgift, t.ex. om `next` eller `peek` anropas när det ej finns fler element att hämta.

(8 p)