NTNU
Norges teknisk-naturvitenskapelige universitet

Fakultet for informasjonsteknologi, matematikk og elektroteknikk

Institutt for datateknikk og informasjonsvitenskap

BOKMÅL

EKSAMEN I FAG
TDT4100 Objekt-orientert programmering

Fredag 3. juni 2005
KL. 09.00 – 13.00

Faglig kontakt under eksamen:
Hallvard Trætteberg, tlf (735)93443 / 918 97263
Trond Aalberg, tlf (735)97952 / 976 31088

Tillatte hjelpemidler:

- Lewis & Loftus: Java Software Solutions (alle utgaver)
- Winder & Roberts: Developing Java Software
- Lervik & Havdal: Programmering i Java
- Mughal, Hamre & Rasmussen: Java som første programmeringsspråk
- Arnold, Gosling & Holms: The Java Programming Language
- Lervik & Havdal: Java the UML Way
- Liang: Introduction to Java programming
- Lewis & Loftus: Java Software Solutions
- Horton: Beginning Java 2 SDK 1.4 Edition
- Brunland,Lingjærde & Maus: Rett på Java
- Lemay/Cadenhead: SAMS Teach Yourself Java 2 platform in 21 days.

Sensurdato:
24. juni 2005. Resultater gjøres kjent på http://studweb.ntnu.no/ og sensurtelefon 81 54 80 14.

Prosentsatser viser hvor mye hver oppgave teller innen settet.

Merk: All programmering skal foregå i Java.

Lykke til!

OPPGAVE 1 (10%): Iterasjon.

a) Skriv kode for en metode `Object[] forskyv1(Object[] tabell)` som returnerer en kopi av `tabell` med alle elementene forskjøvet ett hakk oppover. I resultat-tabellen skal altså element n+1 være lik element n i `tabell`, mens det siste elementet i tabell skal ligge først i resultat-tabellen.

Løsningselementer
- Lage tabell med riktig lengde og type
- Sette første element til siste
- Iterere over de relevante elementene

```
public Object[] forskyv1(Object[] tabell) {
   Object[] kopi = new Object[tabell.length];
   kopi[0] = tabell[tabell.length - 1];
   for (int i = 0; i < tabell.length - 1; i++) {
         kopi[i + 1] = tabell[i];
   }
   return kopi;
}
```

Variasjoner i løsningen: Det er ikke så mange måter å gjøre dette på.

b) Skriv kode for en metode `List forskyv2(Iterator it)` som returnerer en `ArrayList` med alle elementene som `it` "genererer" (med `next()`). Elementene skal være forskjøvet ett hakk, slik at første element som `it.next()` returnerer skal være element nummer to i resultatlista, mens det siste elementet som `it.next()` returnerer skal ligge først i resultatlista. Metodene i `Iterator`-grensesnittet er som følger.

```
// Returns true if the iteration has more elements.
boolean hasNext()

// Returns the next element in the iteration.
Object next()
```

Nødvendige metoder i `List`-grensesnittet (som `ArrayList` implementerer):

```
// Appends the specified element to the end of this list.
boolean add(Object o)

// Inserts the specified element at the specified position.
void  add(int index, Object element)

// Returns the element at the specified position in this list.
Object get(int index)

// Replaces the element at the specified position in this list
// with the specified element.
Object set(int index, Object element)

// Returns the number of elements in this list.
int size()
```

Løsningselementer
- Deklarere resultatlistevariabel av riktig type
- Iterere riktig vha. Iterator-parameteret

- Legge inn elementene i lista med add
- Sørge for at det siste elementet til slutt havner først i resultatlista

```
public List forskyv2(Iterator it) {
   List kopi = new ArrayList();
   while (it.hasNext()) {
      Object o = it.next();
      if (it.hasNext()) {
            kopi.add(o);
      } else {
            kopi.add(0, o);
      }
   }
   return kopi;
}
```

```
public List forskyv2(Iterator it) {
   List kopi = new ArrayList();
   while (it.hasNext()) {
      kopi.add(it.next());
   }
   kopi.add(0, kopi.remove(kopi.size() - 1));
   return kopi;
}
```

Variasjoner i løsningen: Det er ikke mange måter å gjøre dette på heller. Variasjonen ligger i hvordan en håndterer det siste elementet. Over er det to forslag: Det første sjekker om elementer fra iteratoren er det siste, og det bestemmer om elementet legges først eller sist i resultatlista. De andre forslaget legger alle direkte inn og tar så det siste element ut og legger det inn først. I denne brukes imidlertid remove, en List-metode som ikke var oppgitt (burde vært).

OPPGAVE 2 (10%): Klasser og arv.

Gitt følgende klassedefinisjoner:

```
public class Baseklasse {
   public int i;
   public Baseklasse(int i) {
      this.i = i;
   }
   public String toString(int i) {
      return ”[” + (this.i + i) + ”]”;
   }
   public String toString() {
      return toString(i);
   }
}
```

```
public class Subklasse extends Baseklasse {
   public Subklasse() {
      super(4);
   }
   public String toString() {
      return toString(i + i);
   }
}
```

a) Hva skrives ut når følgende kode kjøres:

```
System.out.println(new Baseklasse(4));
System.out.println(new Subklasse());
```

Det vi ønsker å test her er evnene til å følge programflyten, både for konstruktører og toString()-metoder (en må selvsagt også vite at det er toString() som kalles når en bruker System.out.println). I tillegg må en se at this.i og i refererer til forskjellige verdier i toString(int)-metoden. I dette konkrete tilfellet er kallene (og resultatet) som følger:
#1 = new Baseklasse(4) > Baseklasse(4) > this.i = 4
System.out.println(#1) > #1.toString() > #1.toString(4) > ”[” + (4 + 4) + ”]” => “[8]”
#2 = new Subklasse() > Baseklasse(4) this.i = 4
System.out.println(#2) > #2.toString() > #1.toString(4+4) > ”[” + (4 + 8) + ”]” => “[12]”
Merk at det ikke bes om noen forklaring, men at det kan være lurt å gjøre det likevel.

b) Hva skrives ut dersom du fjerner parentesene rundt `this.i + i` i `Baseklasse` sin `String toString(int i)`-metode? Forklar hvorfor.

Poenget her er at this.i + i uten () rundt ikke lenger summerer tall men legger sammen tekst:
... > ”[” + 4 + 4 + ”]” => “[44]”
... > ”[” + 4 + 8 + ”]” => “[48]”

c) Det er ikke bra å ha attributter definert med `public`, de bør heller innkapsles med tilgangsmetoder. Skriv nye versjoner av `Baseklasse` og `Subklasse`, hvor `i`-attributtet er innkapslet.

Her skal i-attributtet gjøres private og get- og set-metoder innføres:
`public int getI() { return i;}`
`public void setI(int i) { this.i = i;}`
Alle referanser til i-attributtet i Subklasse, dvs. de to i toString()-metoden, må gjøres om til getI(). En kan alternativt deklarere i som `protected` og la være å endre Subklasse, men dette er ikke like bra. Vi trekker ikke for å bruke `geti` og `seti`, istedenfor `getI` og `setI`.

OPPGAVE 3 (45%): Klasser, grensesnitt og arv.

I denne oppgaven skal vi jobbe med grensesnitt for og implementasjon av klasser for lister, analogt med klassen java.util.List og java.util.ArrayList.

a) I Java bruker vi nøkkelordet ”interface” for å definere såkalte grensesnitt, i motsetning til vanlige klasser, hvor ”class” brukes.

- Hva kan en ikke ha med i en grensesnittdefinisjon som en kan ha med i en klassedefinisjon og hvorfor?

I en grensesnittdefinisjon kan en kun ha med vanlige metodedefinisjoner uten "kropp", mao. ikke utførbar kode og ikke attributter (kun konstanter, men det er ikke pensum) eller konstruktører. Dette fordi et grensesnitt kun spesifiserer hva en klasse skal kunne, men ikke hvordan. En kan heller ikke bruke nøkkelord som static, private og protected, siden disse er poengløse i en slik kontekst.

Navn på klasser og grensesnitt kan opptre mange steder i javakode, men det finnes noen få begrensninger på hvor hver av disse kan opptre:

- Når kan navn på grensesnitt brukes men ikke navn på vanlige klasser?
- Når kan navn på vanlige klasser brukes men ikke navn på grensesnitt?

Den viktigste forskjellen er at navn på vanlige klasser kan brukes ifm. den enkle varianten av new-operatoren (den som er pensum), noe grensesnittnavn ikke kan. Forøvrig er det kun ifm. implements og extends det er noen forskjell. Navn på grensesnitt kan brukes etter `implements` i definisjonen av vanlige klasser og etter `extends` i en grensesnittdefinisjon. Navn på vanlige klasser kan brukes etter `extends` i definisjonen av vanlige klasser.

b) Gitt følgende grensesnittdefinisjon:

```
public interface Liste {
   // size() returnerer antall elementer i lista
   public int size();

   // Returnerer element nr. indeks i lista.
   // Merk at 0 er første element.
   public Object get(int indeks);

   // Endrer element nr. indeks i lista til o.
   // Merk at 0 er første element.
   public void set(int indeks, Object o);
}
```

- Skriv kode for klassen ListeImpl som implementerer dette grensesnitt og som bruker en vanlig java-tabell (array) for å holde verdiene. Merk at lister av typen Liste i deloppgavene b) og c) ikke kan endre størrelse.
- Definér en konstruktør som tar inn en heltallsverdi som eneste parameter, og som gir java-tabellen denne størrelsen.

ListeImpl trenger et attributt av typen Object[] med størrelse tilsvarende størrelsen på lista. Konstruktøren og get og set-metodene er trivielle, når en har skjønt hvordan tabellen skal brukes. Tabellen bør primært deklareres som protected for senere oppgaver, eller sekundært som private med innkapslingmetoder deklarert med protected.

```
public class ListeImpl implements Liste {

   protected Object[] objects;

   public ListeImpl(int n) {
      objects = new Object[n];
   }
```

```
    public Object get(int i) {
        return objects[i];
    }
    public void set(int i, Object o) {
        objects[i] = o;
    }
    public int size() {
        return objects.length;
    }
}
```

c) Skriv kodelinjer for å opprette et ListeImpl-objekt og sette elementene 0, 1 og 2 til tallene 0, 1 og 2.

Her trodde jeg poenget var at en ikke kan legge tallverdier direkte inn, men må lage tall-objekter, enten med new Integer(n) eller Integer.valueOf(n). Men Java 1.5 gjør jo det automatisk, så det poenget faller bort fra oppgaven (det blir snarere tvert imot). En må passe på å ikke skrive new Liste(), selv om en godt kan deklarere liste-variabelen som en Liste.

```
Liste liste = new ListeImpl(3);
liste.set(0, new Integer(0)); eller liste.set(0, 0);
liste.set(1, new Integer(1)); eller liste.set(1, 1);
liste.set(2, new Integer(2)); eller liste.set(2, 2);
```

Forøvrig helt greit å lage en løkke istedenfor 3 enkeltsetninger.

d) Gitt følgende grensesnittdefinisjon:

```
public interface DynamiskListe ??? Liste {
    // Øker størrelsen til lista med 1 og legger objekt o inn,
    // slik at det får posisjon indeks.
    // Elementer med større eller lik indeks forskyves ett hakk opp.
    public void add(int indeks , Object o);

    // Fjerner elementet i posisjon indeks fra lista.
    // Størrelsen minker med 1, og
    // elementer med større indeks forskyves ett hakk ned.
    public void remove(int indeks);
}
```

- Hvilket nøkkelord må du erstatte "???" med for at dette skal være lovlig javakode?
- Hvilke egenskaper har objekter som er deklarert til å være av typen DynamiskListe?

??? må erstattes med extends (ikke implements), for å arve egenskapene til Liste-grensesnittet. Objekter av typen DynamiskListe kan add og remove, i tillegg til Liste-evnene size, get og set. Trekk for å ta med attributter som kun finnes i implementasjonsklassene.

- Skriv kode for en subklasse av ListeImpl ved navn DynamiskListeImpl, som implementerer DynamiskListe. Les kommentarene i grensesnittdefinisjonen nøye før du skriver koden!
- DynamiskListeImpl skal ha én konstruktør med tom parameterliste som gjør at lista blir opprettet med størrelsen 0.

Her er det viktig å bruke extends (ListeImpl) og implements (DynamiskListe) riktig. En må også kunne endre størrelsen på den underliggende tabellen, dvs. lage en kopi med plass til et element mer (eller mindre), kopiere de relevante elementene over og endre tabell-attributtet. Bygger delvis på ferdighetene i oppgave 1 a), evt. kan en bruke System.arraycopy (som ikke er noe lettere, egentlig). En må også klare å definere en konstruktør som kaller superklassens konstruktør.

```
public class DynamiskListeImpl extends ListeImpl implements DynamiskListe {

   public DynamiskListeImpl() {
      super(0);
   }

   public void add(int i, Object o) {
      Object[] newObjects = new Object[objects.length + 1];
      for (int j = 0; j < i; j++) {
            newObjects[j] = objects[j];
      }
      newObjects[i] = o;
      for (int j = i; j < objects.length; j++) {
            newObjects[j + 1] = objects[j];
      }
      objects = newObjects;
   }

   public void remove(int i) {
      Object[] newObjects = new Object[objects.length - 1];
      for (int j = 0; j < i; j++) {
            newObjects[j] = objects[j];
      }
      for (int j = i + 1; j < objects.length; j++) {
            newObjects[j - 1] = objects[j];
      }
      objects = newObjects;
   }
}
```

e) Vi ønsker å implementere en listetype som begrenser hvilke typer objekter som kan legges i lista. Først defineres følgende grensesnitt:

```
public interface BegrensetListe {
   // Angir om det er lov å legge objektet o inn i lista.
   public boolean accepts(Object o);
}
```

- Skriv kode for en subklasse av DynamiskListeImpl kalt Nummerliste. Nummerliste skal implementere BegrensetListe, slik at det kun er lov å legge tall (både heltall og desimaltall) inn i lista. Definer en passende accepts-metode og redefiner nødvendige andre metoder fra DynamiskListeImpl, slik at det ikke er mulig å legge annet enn tall inn i lista. Legg vekt på å ikke gjenta kode som allerede finnes i DynamiskListeImpl. Utnytt den heller ved å bruke super-nøkkelordet.
- Sørg for at det ”kastes” et unntak av typen IllegalArgumentException, om en prøver å legge inn andre typer objekter.

Her er poenget å redefinere superklassens set- og add-metoder, slik at en får testet objektet som skal legges inn. Dersom objektet er OK, altså accepts returnerer true, skal tilsvarende metode i superklassen kalles, ellers skal unntaket opprettes og "kastes". accepts-metoden må selvsagt være riktig definer, dvs. bruke `instanceof` og teste på Number-klassen (lite trekk for å bruke Integer eller annen tall-klasse). Merk at disse metodene faktisk kan skrives uten å kjenne implementasjonen av tilsvarende metoder i DynamiskListeImpl.

```
public class Nummerliste extends DynamiskListeImpl implements
BegrensetListe {

   public boolean accepts(Object o) {
      return (o instanceof Number);
   }

   public void set(int i, Object o) {
      if (accepts(o)) {
            super.set(i, o);
      } else {
            throw new IllegalArgumentException("Ikke lov til å legge " + o
+ " til " + this);
      }
   }
   public void add(int i, Object o) {
      if (accepts(o)) {
            super.add(i, o);
      } else {
            throw new IllegalArgumentException("Ikke lov til å legge " + o
+ " til " + this);
      }
   }
}
```

f) Det meste av koden du skrev i e) vil være nokså uavhengig av hvilke(n) type(r) objekter det er lov å legge inn i lista. Faktisk kan en skrive klassen slik at det kun er accepts-metoden som må endres, om en skal begrense lista til andre typer objekter.

- Skriv koden for en klasse AbstraktBegrensetListe, som er slik at subklasser kun trenger å implementere accepts-metoden. F.eks. skal følgende klassedefinisjon være nok for å begrense elementene til kun å være av typen String.

```
public class Stringliste extends AbstraktBegrensetListe {
   // begrenser lista til å kun akseptere String-objekter
   public boolean accepts(Object o) {
      return (o instanceof String);
   }
}
```

Her er poenget å se at Nummerliste-klassen refererer til Number-klassen kun ett sted, i accepts-metoden, og skjønne at denne kan defineres som abstract i en generell klasse (eller kan utelates, siden klassen er abstract), mens resten av Nummerliste-klassen brukes direkte. Det er også viktig at klassedeklarasjonen bruker abstract, extends og implements riktig.

```
public abstract class AbstraktBegrensetListe extends DynamiskListeImpl
implements BegrensetListe {

   public void set(int i, Object o) {
```

```
      // som i Nummerliste over
   }
   public void add(int i, Object o) {
      // som i Nummerliste over
      }
   }
}
```

OPPGAVE 4 (20%): Testing

Denne oppgaven handler om testing ift. regler for oppførsel. Merk at dette kan gjøres uten at oppgave 3 er løst, kun at en har forstått hva de spesifiserte metodene er ment å gjøre. Disse har forøvrig samme oppførsel som tilsvarende metoder i standardklassen java.util.List/java.util.ArrayList, som skal være kjent stoff.

a) Definer 2 relevante regler for oppførsel for Liste-grensesnittet definert i oppgave 3. Vis hvordan en kan skrive testkode som tester at ListeImpl følger disse reglene.

To relevante regler: size() returnerer størrelsen riktig, dvs. samme som oppgitt i konstruktøren (som riktignok ikke er definert i grensesnittet) og get returnerer det som som set legger inn. Testkode for å teste dette vil først lage en ListeImpl-instans med f.eks. ett element og sette element 0 med f.eks. liste.set(0, new Integer(0)). Så vil en sjekke om størrelsen er 1 og om liste.get(0) gir tilbake samme verdi (mao. get(0).equals(Integer(0))).

b) Du skal lage testmetoder i en tenkt TestCase-subklasse med JUnit-rammeverket. Skriv to metoder for å teste DynamiskListe-implementasjonen din, en for å teste add-metoden og en for å teste remove-metoden. Bruk metodene `assertEquals(Object, Object)` og `assertTrue(boolean)` i TestCase for å sjekke relevante verdier. Merk at du kun skal bruke metoder som er definert i DynamiskListe og ikke andre metoder som du har i din DynamiskListeImpl-klasse.

Her er det store variasjonsmuligheter, men det viktigste er at veksler riktig mellom kall til endringsmetoden (add eller remove) og assertEquals på verdier fra lesemetoder, med fornuftige tester som sikter mot det som er endret (både størrelsen på lista og verdiene). Når lista endrer størrelse er det viktig at de sjekker flere verdier enn den som er satt inn/tatt ut.

```
public void testDynamiskListeAdd() {
   DynamiskListe liste = new DynamiskListeImpl();
   assertEquals(liste.size(), 0);
   Integer i0 = new Integer(0), i1 = new Integer(1), i2 = new Integer(2);
   liste.add(0, i0);
   assertEquals(liste.size(), 1);
   assertEquals(liste.get(0), i0);
   liste.add(1, i1);
   assertEquals(liste.size(), 2);
   assertEquals(liste.get(0), i0);
   assertEquals(liste.get(1), i1);
   liste.add(1, i2);
   assertEquals(liste.size(), 3);
   assertEquals(liste.get(0), i0);
   assertEquals(liste.get(1), i2);
   assertEquals(liste.get(2), i1);
}

public void testDynamiskListeRemove() {
```

```
   DynamiskListe liste = new DynamiskListeImpl();
   Integer i0 = new Integer(0), i1 = new Integer(1), i2 = new Integer(2);
   liste.add(0, i0);
   liste.add(1, i1);
   liste.add(1, i2);
   liste.remove(0);
   assertEquals(liste.size(), 2);
   assertEquals(liste.get(0), i2);
   assertEquals(liste.get(1), i1);
   liste.remove(1);
   assertEquals(liste.size(), 1);
   assertEquals(liste.get(0), i2);
   liste.remove(0);
   assertEquals(liste.size(), 0);
}
```

c) Tilsvarende som i deloppgave b) skal du lage en testmetode for å teste Nummerliste-klassen fra oppgave 3. Testmetoden skal sjekke om Nummerliste begrenser hvilke elementer som kan legges inn på riktig måte. Du må både sjekke at accepts-metoden virker som den skal og at det kastes et unntak av typen IllegalArgumentException dersom en forsøker å legge inn objekter som ikke er tall.

(Merk at det i originalteksten sto ”som i deloppgave c)”, noe som kan ha forvirret noen). Problemet her er å skjønne hvordan man tester at en Exception ”kastes”. To muligheter er vist under. Den ene husker en evt. Exception som testes for riktig type etter try/catch-blokken. Den andre bruker assertTrue(false) for gi feil dersom det ikke ”kastes” noe unntak og sjekker unntaket som evt. kommer for riktig type. Dette er en vanskelig oppgave, hvor en godt kan være litt snill.

```
public void testNummerliste1() {
   BegrensetListe liste = new Nummerliste();
   assertTrue(liste.accepts(new Integer(0)));
   Exception e = null;
   try {
      liste.add(0, "Feil");
   }
   catch (IllegalArgumentException iae) {
      e = iae;
   }
   assertTrue(e instanceof IllegalArgumentException);
}

public void testNummerliste2() {
   BegrensetListe liste = new Nummerliste();
   assertTrue(liste.accepts(new Integer(0)));
   try {
      liste.add(0, "Feil");
      assertTrue(false);
   }
   catch (Exception e) {
      assertTrue(e instanceof IllegalArgumentException);
   }
}
```

OPPGAVE 5 (15%): Observatør-observert-teknikken

Du ønsker å lage en klasse Listesum, som observerer hvordan elementene i Nummerliste-klassen endrer seg, slik at den hele tiden har en oppdatert sum av tallene i lista. Listesum vil altså fungere som *observatør*, mens Nummerliste vil være *observert*. Som en start, definerer du følgende lyttergrensesnitt, som Listesum må implementere:

```
public interface Listeendringslytter {
   // Kall til  listeEndret-metoden angir at endretListe er endret
   // i intervallet fra og med fra til og med til
   public void listeEndret(Liste endretListe, int fra, int til);
}
```

a) Anta at en Nummerliste kun kan ha én Listeendringslytter knyttet til seg. Hvordan må du endre Nummerliste for at den skal fungere som observert og si fra om endringene.

Det å være observert krever to ting:

1) Nummerliste må kunne registrere én eller flere observatører. Her kreves bare én, så en innfører et privat attributt endringslytter av typen Listeendringslytter og en set-metode for å kunne registrere lytteren utenifra.
2) Den observerte må si fra til observatørene når relevante endringer skjer. Dette krever at alle endringsmetodene, i vårt tilfelle set, add og remove, må kalle listeEndret på Listeendringslytter-objektet med korrekte parametre. Det er ofte praktisk å gjøre dette vha. en hjelpemetode, f.eks. fireListeEndret med parametre fra og til, som igjen kaller `endringsLytter.listeEndret(this, fra, til)`. Merk at det her ikke spørres om kode, slik at en skal kunne forklare hva disse metodene må gjøre, uten at oppgave 3 a) og c) er riktig besvart. Det en må ha skjønt er at alle metodene som endrer tabellen, må si fra om det, selve essensen i observert-rollen.

b) Forklar med tekst og kode hvordan Listesum-klassen må virke, for at et privat sum-attributt av typen int hele tiden skal være oppdatert iht. Nummerliste-objektet det lytter på. Tegn sekvensdiagram som viser hva som skjer når Nummerliste-objektet endres, med set-, add- og remove-metodene.

For det første må Listesum implementere Listeendringslytter-grensesnittet og dermed også listeEndret-metoden. For det andre må den beregne en ny sum ved å gå gjennom Nummerliste-objektet som sier fra om endringen. Det er liten vits å være smart og prøve å optimalisere oppdateringen av sum-attributtet, ved kun å løpe gjennom den delen som er endret. De tre sekvensene er nesten like og vil være som vist under (på tabellform). Det er ikke så viktig med notasjonen, bare det går tydelig frem hvem som gjør hva og kallsekvensen stemmer.

#1: Nummerliste	**#2: Listesum**
>> set(i, o)	
fireListeEndret(i, i)	
	>> listeEndret(#1, i, i)
get(0) <<	
...	
get(size() – 1) <<	
>> add(i, o)	
fireListeEndret(i, size() – 1)	
	>> listeEndret(#1, i, #1.size() – 1)

get(0) <<	
...	
get(size() - 1) <<	
>> remove(i)	
fireListeEndret(i, size() – 1)	
	>> listeEndret(#1, #1. size() - 1)
get(0) <<	
...	
get(size() - 1) <<	