**NTNU**
**Norges teknisk-naturvitenskapelige universitet**

**Fakultet for informasjonsteknologi, matematikk og elektroteknikk**

**Institutt for datateknikk og informasjonsvitenskap**

**BOKMÅL**

## EKSAMEN I FAG
## TDT4100 Objektorientert programmering

## Fredag 2. juni 2006
## 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:**

- Én og kun én trykt bok, f.eks. Lewis & Loftus: Java Software Solutions

**Sensurdato:**
23. juni 2006. 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 (15%): Iterasjon.

a) Skriv kode for en metode `Object[] flett1(Object[] tabell1, Object[] tabell2)` som returnerer en tabell med alle elementene fra `tabell1` og `tabell2` *flettet*. Anta at `tabell1` er `[x1, x2, ..., xN]` og `tabell2` er `[y1, y2, ..., yN]`. Da skal resultat-tabellen være `[x1, y1, x2, y2, ..., xN, yN]`. Du kan anta at `tabell1` og `tabell2` er like lange.

Oppgaven skal teste opprettelse av tabell, iterasjon over tabell og enkel logikk. Løsningsforslag:

```
Object[] flett1(Object[] tabell1, Object[] tabell2) {
   Object[] result = new Object[tabell1.length + tabell2.length];
   for (int i = 0; i < result.length; i += 2) {
      result[i] = tabell1[i / 2];
      result[i + 1] = tabell2[i / 2];
   }
   return result;
}
```

Mange alternativer er mulige, f.eks. å bruke i++ og veksle mellom å hente verdier fra den ene eller andre tabellen, enten basert på om i-variablen er delelig på 2 eller egne indeksvariabler for hver tabell. *Det gis 6 poeng: 2 poeng for opprettelse av tabell med rett lengde, 2 poeng for riktig iterasjon (altså det rundt innmaten i løkka) og 2 poeng for riktig innmat i løkka.*

b) Skriv kode for en metode `List flett2(Iterator it1, Iterator it2)` som returnerer en ny `List` med elementene fra `it1` og `it2` flettet. Annenhvert element i resultat-lista skal komme fra `it1` og annenhvert fra `it2`, *inntil en av dem er tømt*. Anta at `it1` genererer `[x1, x2, ..., xN]` og `it2` genererer `[y1, y2, ..., yN+2]`. Da skal resultat-lista inneholde `[x1, y1, x2, y2, ..., xN, yN]`

Oppgaven skal teste opprettelse av instans av List-implementasjon, iterasjon med Iterator og enkel logikk. Løsningsforslag:

```
List flett2(Iterator it1, Iterator it2) {
   List result = new ArrayList();
   while (it1.hasNext() && it2.hasNext()) {
      result.add(it1.next());
      result.add(it2.next());
   }
   return result;
}
```

Lista opprettes vha. new og en List-implementasjon, ikke List selv. Siden en List har dyamisk størrelse, er den tom fra starten og en bruker add for å legge til (på slutten). Løkka må veksle mellom hasNext() og next(), kan ha mer enn ett hasNext() kall pr. løkke, men kun ett kall til next() pr. løkka. Siden løkka stopper når en av iteratorene er tomme, må en bruke &&-logikk. *Det gis 6 poeng: 1 poeng for opprettelse av ArrayList, 2 poeng for riktig iterasjon (altså det rundt innmaten i løkka) og 3 poeng for riktig innmat i løkka.*

c) Skriv kode for en metode `List flett3(Iterator it1, Iterator it2)` som returnerer en ny `List` med elementene fra `it1` og `it2` flettet, som i b), men hvor alle elementene fra `it1` og `it2` brukes. Anta at `it1` genererer `[x1, x2, ..., xN]` og `it2` genererer `[y1, y2, ..., yN+2]`. Da skal resultat-lista inneholde `[x1, y1, x2, y2,`

`..., xN, yN, yN+1, yN+2]` Du trenger ikke gjenta uendret kode fra `flett2`, så lenge det går klart frem hva som er uendret og hva som endres/erstattes.

Forskjellen mellom denne og forrige er løkketesten, som skal stoppe først når begge er tomme, dvs. en eller begge har flere, og at en må ha en if for hver iterator inni løkka. Det er greit om dette forklares og en bare gjentar løkketesten. Løsningsforslag:

```
   // samme som over
   while (it1.hasNext() || it2.hasNext()) {
      if (it1.hasNext()) {
            result.add(it1.next());
      }
      if (it2.hasNext()) {
            result.add(it2.next());
      }
   }
   // samme som over
}
```

Et alternativ er å ha samme løkke som over, og så ha nye løkker under, for iteratoren som ikke er tømt. *Det gis 3 poeng for riktig svar.*

Relevante metoder fra `List`-grensesnittet:

```
// 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()

// Returns the index in this list of the first occurrence
// of the specified element, or -1 if it doesn't exist in this list.
int indexOf(Object element)
```

Relevante metoder fra Iterator-grensesnittet:

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

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

## OPPGAVE 2 (40%): Enkle klasser

Du skal lage klasser for å beskrive en CD og musikken som er på den. Klassene skal brukes i en applikasjon for CD-brenning, hvor en skal kunne fylle en eller flere CD'er med musikk, før de(n) brennes. Merk at ikke alle deloppgaver krever at de foregående er løst, så ikke hopp

over de resterende deloppgavene om én blir for vanskelig. Bruk gjerne grensesnitt og klasser fra Java sitt API/klassebibliotek, f.eks. Collection-rammeverket.

a) Hver CD har et navn og inneholder et sett spor, og hvert spor har et navn, en ventepause angitt i hele sekunder og en spillelengde angitt i tidels sekunder. Definer klassene CD og Spor og feltene som trengs for å holde disse dataene.

Her skal en teste grunnleggende syntaks i en klassedeklarasjon. Feltene må ha riktig datatype og det er fint om en bruker generics for å spesialisere lista. Det er ikke så farlig med synlighetsmodifikatorer for feltene, så lenge dette håndteres i punkt c).

```
public class CD {
   String name;
   List<Spor> sporliste = new ArrayList<Spor>();
}

public class Spor {
   String name;
   int pauseTime;
   double playTime;
}
```

Ikke så mye å variere på her. Det er greit å bruke en tabell istedenfor List/ArrayList til sporene, så lenge den brukes riktig. *Det gis 5 poeng: 1 poeng for klassesyntaks, og 4 poeng for riktige datatyper (String, (Array)List<Spor>, int, double).*

b) Forklar begrepet *innkapsling*.

Kontrollert tilgang til og endring av tilstanden til et objekt, ved at all tilgang gjøres vha. av metoder. Tilgangsmetodene må følge reglene for oppførsel, slik at tilstanden aldri blir inkonsistent eller på annen måte ulovlig. *Det gis 5 poeng.*

c) Skriv kode for nødvendige metoder i CD og Spor for å kapsle inn feltene definert i a). Forklar hvordan du velger navn, retur- og parametertyper for disse metodene.

Her skal de vise at de skjønner hva innkapsling betyr i praksis. De må både bruke synlighetsmodifikatorene riktig, ha riktige sett med metoder og riktige metodedeklarasjoner (navn og datatyper) og riktig innmat.

I CD:

```
   // innkapsling av name: get- og set-metoder
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }

   // innkapsling av spor: metode for antall, hente ut, legge til og fjerne
   public int getSporCount() {
      return sporliste.size();
   }
```

```
   public Spor getSpor(int i) {
      return sporliste.get(i);
   }

   public void addSpor(Spor spor) throws TooLongPlayTimeException {
      if (! sporliste.contains(spor)) {
            sporliste.add(spor);
      }
   }

   public void removeSpor(Spor spor) {
      // testen er strengt tatt ikke nødvendig her, men trengs senere
      if (sporliste.contains(spor)) {
            sporliste.remove(spor);
      }
   }
```

I Spor:

```
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getPauseTime() {
      return pauseTime;
   }
   public void setPauseTime(int pause) {
      this.pauseTime = pause;
   }
   public double getPlayTime() {
      return playTime;
   }
   public void setPlayTime(double playTime) {
      this.playTime = playTime;
   }
```

Ikke så mye å variere på her. *Det gis 5 poeng: 1 poeng for riktig bruk av synlighetsmodifikatorer (public), 2 poeng for innkapsling av de enkle feltene og 2 poeng for innkapsling av spor-lista.*

d) Lag en metode computePlayTime i CD-klassen som tar inn en logisk verdi og som returnerer tiden i antall sekunder (med desimaler) det tar å spille av CD'en. Den logiske verdien skal avgjøre om ventepausene skal regnes med (logisk verdi er sann) eller ikke (logisk verdi er usann).

Her tester vi enkel iterasjon over liste, bruk av logisk (boolean) variabel, bruk av metodekall på relatert objekt og if-setning.

```
public double computePlayTime(boolean pause) {
   double time = 0.0;
   for (int i = 0; i < sporliste.size(); i++) {
      Spor spor = sporliste.get(i);
      time += spor.getPlayTime();
      if (pause) {
            time += spor.getPauseTime();
      }
```

```
   }
   return time;
}
```

Ikke så mye å variere på her. *Det gis 5 poeng: 2 poeng for riktig iterasjon over liste (indeks-basert eller med iterator), 1 poeng for riktig håndtering av summen (dekarasjon, initialisering og akkumulasjon) og 2 for riktig bruk av metodekall, logisk (boolean) variabel og test.*

e) Du ønsker å lage en hjelpemetode `sorter` i Spor-klassen som sorterer en tabell med Spor-objekter, basert på musikklengden. Skriv metodedeklarasjonen og forklar hvordan du vil implementere `sorter`-metoden vha. klasser/grensesnitt og metoder definert i Collection-rammeverket. Merk at du trenger ikke implementere noen sorteringsalgoritme selv.

Her tester vi om de kjenner til sortering vha. sammenligningsgrensesnitt, enten Comparable eller Comparator. Det enkleste er å la Spor implementere Comparable og så la denne sammenligne playtime. Så brukes Arrays.sort til sortering (det viktigste er at de vet at det finnes en standardmetode for å sortere elementer og som implisitt bruker Comparable-grensesnittet). En kan alternativt bruke Comparator, som er litt mer komplisert men helt analog.

I Spor:

```
public static void sorter(Spor[] spor) {
   Arrays.sort(spor);
}
// from Comparable
public int compareTo(Object o) {
   double otherPlayTime = ((Spor)o).getPlayTime();
   if (playTime < otherPlayTime) {
         return -1;
   } else if (otherPlayTime < playTime) {
         return 1;
   } else {
         return 0;
   }
}
```

Vi krever ikke så mye kode, men deklarasjonen av sorter må være rett og bruken av compareTo i Comparable må være forklart. *Det gis 5 poeng: 1 poeng for deklarasjonen av sorter, 1 poeng for bruk av standard-metode, 2 poeng for bruk av Comparable (implementeres av Spor) og 1 poeng for forståelse av compareTo.*

En CD kan ha plass til maksimum 72 minutter med musikk (altså unntatt ventepausene mellom sangene) og det er viktig å unngå at brukeren prøver å brenne CD'er med for liten plass til innholdet.

f) Definer en type unntak for å si fra om at en CD er i ferd med å bli overfylt. Utvid relevante metoder fra b) slik at det genereres/kastes et unntak av denne typen dersom en prøver å legge Spor til en CD slik at musikklengden overstiger 72 minutter.

Her testes bruk av Exception-mekanismen, både deklarasjon av RuntimeException-subklasse og bruk av throw.

```
public class TooLongPlayTimeException extends RuntimeException {
   public TooLongPlayTimeException(String message) {
      super(message);
   }
}

   // metode må endres slik at ny spillelengde testes
   public void addSpor(Spor spor) throws TooLongPlayTimeException {
      if (! sporliste.contains(spor)) {
            double newPlayTime = computePlayTime(false) +
spor.getPlayTime();
            if (newPlayTime > getMaxPlayTime()) {
                  throw new TooLongPlayTimeException("Too long play time: "
+ newPlayTime);
            }
            sporliste.add(spor);
      }
   }
```

Exception-klassen følger vanlig klassesyntaks, med RuntimeException (til nød Exception) som superklasse. addSpor-metoden må utvides ved at den nye spilletiden sjekkes *før* det nye Spor-objektet legges til i lista og throw evt. brukes. *Det gis 5 poeng: 2 poeng for deklarasjon av RuntimeException-subklassen, 1 poeng for riktig bruk av throw og 2 poeng for riktig utvidelse av (og plassering av throw-setning) av addSpor.*

g) Hva skjer med din implementasjon frem til og med f), dersom musikklengden til et Spor-objekt økes *etter* at den er lagt til CD-objektet? Forklar (kode er ikke nødvendig) hvordan du vil endre CD og/eller Spor for å sikre at det også da kastes unntak om den nye musikklengden er for stor.

Her er poenget at endring av Spor-objektet direkte ikke vil merkes av CD-klassen, der logikken for testing av musikklengden ligger (og nødvendigvis må ligge). For at et Spor-objekt skal kunne sjekke den nye spilletiden og evt. ”kaste” et unntak, må den ha en referanse til ”eier”-CD’en. Altså må en innføre et CD-felt og innkapslingsmetoder for feltet og sørge for at feltet blir satt/resatt i add/removeSpor-metodene. I metoden i Spor hvor musikklengden blir satt, må en legge inn kode tilsvarende den i addSpor for å beregne den nye lengden og bruke evt. throw-setningen om lengden er for stor.

I Spor:

```
   private CD cd;

   public void setCD(CD cd) {
      this.cd = cd;
   }

   public void setPlayTime(double playTime) {
      if (cd != null) {
            double newPlayTime = cd.computePlayTime(false) - this.playTime
+ playTime;
            if (newPlayTime > cd.getMaxPlayTime()) {
                  throw new TooLongPlayTimeException("Too long play time: "
+ newPlayTime);
            }
      }
```

```
      this.playTime = playTime;
   }
```

I CD:

```
   public void addSpor(Spor spor) throws TooLongPlayTimeException {
      if (! sporliste.contains(spor)) {
            ...
            sporliste.add(spor);
            spor.setCD(this);
      }
   }

   public void removeSpor(Spor spor) {
      if (sporliste.contains(spor)) {
            sporliste.remove(spor);
            spor.setCD(null);
      }
   }
```

Selv om vi ikke ber om kode, må forklaringen inneholde de vesentlige elementene: en referanse fra Spor- til CD-objektet, innkapsling og bruke av denne og inkludere sjekken på den nye musikklengden. Det er også mulig å bruke en mer generell lytterteknikk/observatør-observert-samspill. Dette blir litt mer omstendelig og krever at en introduserer et eget lyttergrensesnitt for å si fra om endringer i musikklengden. Poengene gis på samme måte, siden lytterløsningen er helt analog (selv om teknikken er mer generell). *Det gis 5 poeng: 2 poeng for referansen fra Spor til CD, 2 poeng for håndtering av referansen i add/removeSpor i CD og 1 poeng for bruk av referansen i metoden for å endre musikklengden.*

h) Anta at du skal støtte både vanlige CD'er og mini-CD'er, hvor forskjellen er at de rommer ulik mengde musikk (målt i spilletid og ikke antall spor). Du innfører derfor en ny klasse MiniCD, og ønsker å ha *mest mulig kode* samlet i en felles abstrakt klasse med navn AbstractCD, som CD- og MiniCD-klassene arver fra. Forklar hvordan MiniCD- og AbstractCD-klassene blir og hvordan du vil strukturere om og evt. endre koden du allerede har skrevet.

Her tester vi forståelsen av abstrakte klasser: At det som er felles samles i en abstrakt klasse, at det som er forskjellen *deklareres* i den abstrakte klassen og *implementeres* i subklassene. I dette tilfellet vil abstractCD inneholde en abstrakt metode for å beregne maksimal spilletid og forøvrig inneholde samme kode som opprinnelig var i CD. Den abstrakte metoden implementeres så (på hver sin måte) i subklassene, gjerne ved å returnere en konstant. Dette kan også håndteres ved at det defineres et felt for maksimal spilletid, som settes i konstruktøren til hver subklasse, men det er ingen grunn til å lagre denne verdien, da den vil være konstant og lik for alle objekter av en bestemt klasse.

```
public abstract class AbstraktCD {
   ...
   public abstract double getMaxPlayTime();
   ...
}

public class CD extends AbstraktCD {
   public double getMaxPlayTime() {
      return 72 * 60.0;
   }
```

```
}
// tilsvarende for MiniCD
public class MiniCD extends AbstraktCD {
   public double getMaxPlayTime() {
      return ...;
   }
}
```

Selv om vi ikke ber om kode, må forklaringen inneholde de vesentlige elementene: felles metoder i abstractCD, deklarasjon og bruk av abstrakt metode i abstractCD i addSpor (og i metoden i Spor for å sette musikklengden) og implementasjon i subklassene. *Det gis 5 poeng: 2 poeng for deklarasjon av abstrakt metode i abstractCD, 1 poeng for bruk av denne i abstractCD og 2 poeng for implementasjon av den abstrakte metoden i CD og MiniCD-klassene.*

## OPPGAVE 3 (30%): Collection-rammeverket.

I denne oppgaven skal du jobbe med en implementasjon av en forenklet java.util.Map (heretter bare kalt Map). Det er *ikke* lov til å bruke klasser fra Collection-rammeverket i denne oppgaven. Det kan være lurt å lese gjennom hele oppgaven før du bestemmer deg for hvordan du løser hver enkelt del.

Map-grensesnittet er definert som følger:

```
public interface Map {
   // size returnerer antall nøkkel/verdi-par.
   public int size();

   // put knytter value til nøkkelen key,
   // slik at get(key) returnerer value.
   // En evt. eksisterende verdi for key blir erstattet.
   // Dersom verdien er null, skal nøkkel/verdi-paret fjernes.
   public void put(Object key, Object value);

   // get returnerer verdien knyttet til key,
   // eller null dersom nøkkelen ikke finnes.
   public Object get(Object key);

   // keys returnerer en tabell med alle nøklene.
   public Object[] keys();
}
```

Anta at følgende hjelpeklasse allerede er implementert:

```
public class ArrayUtil {
   // Returns a new array containing all the objects from os and o
   public static Object[] add(Object[] os, Object o)

   // Returns a new array containing
   // all the objects from os except the one at position indeks
   public static Object[] remove(Object[] os, int indeks)
}
```

a) Skriv kode for klassen ArrayMap som implementerer Map-grensesnittet og som bruker én eller flere tabeller for å holde nøklene og verdiene.

Dette er både en test av om de forstår hva en Map er og kan bruke en underliggende datastruktur til å implementeres metoder deklarert i et grensesnitt. Den enkleste løsningen er å bruke to tabeller, en for nøkler og en for verdier, med nøkkel/verdi-par på samme indeks. Siden både get- og put- må finne indeksen til en eksisterende nøkkel, er det greit å skille indexOf ut som en egen metode. I tillegg har vi i løsningen vår skilt ut nøkkelsammenligning som egen metode, som et svar på deloppgave c). get-metoden må ta høyde for om nøkkelen finnes fra før. put-metoden må ta høyde for fire tilfeller:
1) nøkkel finnes fra før og verdien er ulik null: verdi settes i riktig posisjon
2) nøkkel finnes fra før og verdien er null: nøkkel og verdi fjernes
3) nøkkel finnes ikke fra før og verdi er ulik null: ny nøkkel og verdi legges til
4) nøkkel finnes ikke fra før og verdi er lik null: ingenting

```
public class ArrayMap implements Map {

   private Object[] keys = new Object[0];
   private Object[] values = new Object[0];

   protected boolean isKey(Object key, Object other) {
      return key == other || (key != null && key.equals(other));
   }

   private int indexOf(Object key) {
      for (int i = 0; i < keys.length; i++) {
            if (isKey(key, keys[i])) {
                  return i;
            }
      }
      return -1;
   }

   public Object get(Object key) {
      int pos = indexOf(key);
      return (pos < 0 ? null : values[pos]);
   }

   public Object[] keys() {
      return keys;
   }

   public void put(Object key, Object value) {
      int pos = indexOf(key);
      if (pos >= 0) {
            if (value == null) {
                  keys = ArrayUtil.remove(keys, pos);
                  values = ArrayUtil.remove(values, pos);
            } else {
                  values[pos] = value;
            }
      } else if (value != null) {
            keys = ArrayUtil.add(keys, key);
            values = ArrayUtil.add(values, value);
      }
   }

   public int size() {
      return keys.length;
   }
```

```
}
```

Det er mulig å bruke én tabell som inneholder vekselvis nøkler og verdier, med de samme tilfellene å ta hensyn til. Den største praktiske forskjellen blir keys-metoden, da den må lage en ny tabell med kun nøkler i. En løsningsvariant er å ikke fjerne nøkkel/verdi-par når verdien er null, men sørge for at size() og keys() ikke regner/tar dem med. *Det gis 14 poeng: 2 poeng for size, 2 poeng for get, 4 poeng for put og 2 poeng for keys og 4 ekstra poeng for ryddig kode f.eks. å skille ut indexOf som felles metode. Det trekkes ikke for alternative løsninger, så lenge de virker.*

b) Skriv kode for en konstruktør ArrayMap(Map annenMap) som initialiserer det nye ArrayMap-objektet med nøkkel/verdi-parene fra annenMap.

Her tester vi om en vet hva en konstruktør er og at en skjønner forskjellen på å programmere mot et grensesnitt ift. en konkret implementasjon. Her må en altså oppdage at parameteret er kun garantert å være en Map og ikke deres egen ArrayMap-implementasjon. Det betyr at en må bruke keys- og get-metoden for å få tak i alle nøkkel- og verdi-parene. Vi viser to løsninger, én som bruker put for å legge dem inn i ArrayMap-objektet som initialiseres og én som kopierer rett inn i de to tabellene for en mer effektiv bruk av minnet.

```
public ArrayMap(Map other) {
   Object[] keys = other.keys();
   for (int i = 0; i < keys.length; i++) {
         put(keys[i], other.get(keys[i]));
   }
}

public ArrayMap(Map other) {
   Object[] keys = other.keys();
   this.keys = new Object[keys.length];
   this.values = new Object[keys.length];
   for (int i = 0; i < keys.length; i++) {
         this.keys[i] = keys[i];
         values[i] = other.get(keys[i]);
   }
}
```

*Det gis 6 poeng: 2 for å skjønne at en kun kan bruke Map-metoder og 4 for løsningen.*

c) Når en sammenligner nøkler i ArrayMap sine metoder, har en valget mellom (minst) to måter å sammenligne objekter (for likhet) på. Hvilke to er det, hva er forskjellen og evt. fordelen/ulempen med hver av disse? Hvordan kan du skrive (om) ArrayMap slik at det er lett å bytte (til en annen) sammenligningsmetode?

Her er poenget å forstå forskjellen mellom == og equals. Den første er raskere og mer forutsigbar, den andre er tregere, men en kan selv bestemme hva det vil si at noe er likt. Dette er allerede brukt i Java, slik at to heltall av typen Integer er equals hvis dere intValue er like, selv det er forskjellige objekter. Tilsvarende er to String-objekter equals, dersom de inneholder den samme bokstavsekvensen. I vår løsning over har vi vist hvordan man kan lage en egen metode for sammenligningen og som lett kan redefineres i en subklasse. *Det gis 10 poeng: 3 for forståelsen av forskjellen mellom == og equals, 3 for god forklaring på fordelen/ulempen, 2 for eksempler og 2 for å skille ut sammenligningen i en egen metode som kan redefineres.*

## OPPGAVE 4 (15%): Testing

a) Definer *regler for oppførselen* til metodene i Map-grensesnittet definert i oppgave 3. Forklar hvordan en på grunnlag av slike regler kan skrive testkode som tester at en spesifikk Map-implementasjon (f.eks. ArrayMap) følger disse reglene. Merk at det er den generelle teknikken vi ønsker forklaring på, ikke spesifikt JUnit-rammeverket.

Den mest fundamentale regel for oppførsel er gitt i oppgaveteksten: etter put(key, value) skal get(key) returnere value. Dersom key ikke finnes skal size returnere en mer og tabellen som keys returnerer skal inneholde den nye nøkkelen. Dersom key finnes og value imidlertid er null, skal nøkkelen fjernes, slik at size sine returverdier påvirkes tilsvarende. Testteknikken baserer seg på å opprette objekter av typen en ønsker å test, og så vekselvis kalle metoder som endrer tilstanden til objektet og lesemetoder og sjekke verdiene de returnerer. F.eks. map.put(”key”, ”value”) og sjekke at map.get(”key”) faktisk returnerer ”value”. Ved å formulere regler for før- og etter-tilstanden ved bruk av endringsmetoder, kan en lett omsette regler i slik testkode. *Det gis 9 poeng: 4 for regler tilsvarende spesifikasjonen (1 poeng for hver put-tilfelle) i oppgaven og 5 for en god forklaring av testteknikken. Merk at reglene bør formuleres slik at det er lett å skrive testkoden, f.eks. etter kall til put(key, value) og value != null skal get(key) returnere value*.

b) Skriv en eller flere testmetoder i en tenkt TestCase-subklasse med JUnit-rammeverket, som til sammen tester get-, put- og size-metodene i ArrayMap-implementasjonen av Map-grensesnittet. Bruk metodene assertEquals(Object, Object) og assertTrue(boolean) i TestCase for å sjekke relevante verdier.

Det er ikke så farlig om en skriver flere testmetoder eller samler hele testen i én metode. Det viktigste er å prøve alle de fire tilfellene av put: med eksisterende og ny nøkkel og med verdi som er lik og ulik null. For hver av disse bør en sjekke resultatet av get og size.

```
public void testArrayMap(Map map) {
   String key1 = "key1", key2 = "key2", key3 = "key3";
   String value1 = "value1", value2 = "value2";

   assertEquals(0, map.size());
   assertNull(map.get(key1));
   map.put(key1, value1);
   assertEquals(value1, map.get(key1));
   assertEquals(1, map.size());

   assertNull(map.get(key2));
   map.put(key2, value2);
   assertEquals(value2, map.get(key2));
   assertEquals(2, map.size());

   map.put(key1, value2);
   assertEquals(value2, map.get(key1));
   assertEquals(2, map.size());

   map.put(key2, null);
   assertNull(map.get(key2));
   assertEquals(1, map.size());

   map.put(key1, null);
   assertNull(map.get(key2));
   assertEquals(0, map.size());
```

```
    map.put(key3, null);
    assertNull(map.get(key3));
    assertEquals(0, map.size());
  }
```

*Det gis 6 poeng: 1 poeng for å sjekke starttilstanden, 4 poeng for kode tilsvarende de fire put-tilfellene og 1 poeng for riktig bruk av assertEquals, assertTrue og assertNull.*