NTNU
Norges teknisk-naturvitenskapelige universitet

Fakultet for informasjonsteknologi, matematikk og elektroteknikk

Institutt for datateknikk og informasjonsvitenskap

KONTINUASJONSEKSAMEN I FAG
TDT4100 Objektorientert programmering

BOKMÅL

Onsdag 16. august 2006
Kl. 09.00 – 13.00

Faglig kontakt under eksamen:
Trond Aalberg, tlf (735)97952 / 976 31088

Tillatte hjelpemidler:

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

Sensurdato:
6. september 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 (20%): Metoder og kontrollstrukturer

Klasser og metoder som kan være nyttige for denne oppgaven finner du på siste side av eksamensoppgaven.

a) Skriv kode for en metode `String stringMulti(String str, int x)` som returnerer en ny streng bestående av strengen `str` gjentatt `x` ganger. Eksempelvis skal metodekallet `stringMulti("ah", 4)` returnere strengen *"ahahahah"*.

Oppgaven løses ved hjelp av en enkel løkke og i dette tilfellet er det mest hensiktemessig å bruke en for-løkke. En variabel deklareres før løkken og tilordnes en tom streng slik at det er mulig å konkatenere strengene. Det er mulig å bruke result.concat("str"), men det samme kan også oppnåes med bruk av operatoren +=.

```
String stringMulti(String str, int x) {
   String result = "";
   for (int i = 0; i < x ; i ++) {
      result += str;
   }
   return result;
}
```

b) Skriv kode for en metode `String swapChars(String str, int p1, int p2)` som returnerer en ny `String` hvor tegnene i indeks p1 og p2 er byttet om. Eksempelvis skal `swapChars("abcde", 2, 4)` returnere strengen "*abedc*".

Oppgaven kan løses ved å konvertere strengen til en char-array. For å kunne bytte om tegnene riktig må du mellomlagre ett av dem.

```
String swap(String s, int p1, int p2){
   char a[] = s.toCharArray();
   char temp = a[p1];
   a[p1] = a[p2];
   a[p2] = temp;
   return new String(a);
}
```

c) Et palindrom er et ord eller en setning som har samme mening enten en begynner å lese forfra eller bakfra. Eksempler på palindromer er "*agnes i senga*", "*regninger*" og "*grav ned den varg*". Et palindrom kan ha et antall bokstaver som enten er partall eller oddetall og det er uvesentlig om det er brukt store bokstaver eller små bokstaver. Du skal med andre ord i løsningen ta høyde for at både "*abcba*" og "*Abccba*" er palindromer. For palindromer er også mellomrom en del av strengen.

Skriv kode for en metode `boolean isPalindrom(String str)` som returnerer true hvis strengen er et palindrom og false hvis den ikke er et palindrom.

Oppgaven kan løses med både for- og while-løkke. For å iterere over tegnene i en streng benyttes enten string-klassens metode charAt eller strengen gjøres om til en char-array slik at det er mulig å bruke tabellindekser. Sjekking av om en streng er et palindrom gjøres ved at første og siste tegn sammenlignes, deretter andre og nest siste tegn osv. Hvis det er forskjellige tegn så er strengen ikke et palindrom. Testen kan avbrytes når man når midten, og alle tegn er sammenlignet. I forslagene under itererer vi helt til head > tail, noe som er nødvendig for å fange opp både partalls og oddetalls palindromer.

```
// Med bruk av while-løkke
public static boolean isPalindrom(String str) {
   String pal = str.toLower();
   int head = 0;
   int tail = pal.length() - 1;
   while (head < tail && pal.charAt(head) == pal.charAt(tail)){
      head ++;
      tail --;
   }
   return head >= tail;
}

// Med bruk av for-løkke
public static boolean isPalindrom2(String str) {
   int size = str.length() - 1;
   for (int head = 0, tail = size; head < tail; head++, tail--){
      if (str.charAt(head) != str.charAt(tail)){
            return false;
      }
   }
   return true;
}
```

d) Et ord/uttrykk som er et anagram for et annet ord/uttrykk består av de samme bokstavene men rekkefølgen er byttet om og meningen er forskjellig. Eksempelvis er ”*amor*” et anagram for ”*Roma*”, ”*Tom Marvolo Riddle*” er et anagram for ”*I am Lord Voldemort*”, og ”*Presbyterians*” er et anagram for ” *Britney Spears*”. Legg merke til at anagrammer kan være forskjellige fra det opprinnelige ordet med hensyn til blanke tegn og store/små bokstaver, men det er det samme sett av bokstaver som er brukt i begge.

Skriv metoden `boolean areAnagrams(String str1, String str2)` som returnerer true hvis strengene er anagrammer og false hvis de ikke er anagrammer.

I denne oppgaven er vi interessert i hvordan du vil løse et tilsynelatende komplisert problem med hjelp av de metodene som er oppgitt i appendiks til eksamensoppgaven. Problemet kan løses enkelt og greit ved å fjerne alle blanke tegn i strengene, konvertere til char-arrays, sortere disse og sammenligne om tabellene eller strenger instansiert med disse tabellene er like.

```
boolean areAnagrams(String str1, String str2){
   char s1[] = str1.replaceAll(" ", "").toLowerCase().toCharArray();
   char s2[] = str2.replaceAll(" ", "").toLowerCase().toCharArray();
   java.util.Arrays.sort(s1);
   java.util.Arrays.sort(s2);
   return (new String(s1).equalsIgnoreCase(new String(s2)));
}
```

OPPGAVE 2 (40%): Klasser

I denne oppgaven skal du lage klasser for å beskrive et hus med etasjer og rom, og du skal løse forskjellige deloppgaver i tilknytning til denne enkle objekt-modellen.
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) Definer klassene Hus, Etasje og Rom. Du kan utelate konstruktører og metoder i denne delen av oppgaven siden du skal gjøre dette i de neste deloppgavene.
   - Et rom har et areal (heltall) og har en betegnelse (f. eks. ”bad”, ”stue”, ”kjøkken”).
   - Etasjer har et navn (”kjeller”, ”første”, ”andre”, ”loft”).
   - Et hus har en adresse og en eier.
   - Det kreves at klassene dine er i stand til å holde rede på hvilke rom som finnes i hvilke etasjer og hvilke etasjer et hus inneholder.

Her skal du kun lage klasse-definisjoner med felter basert på det som er beskrevet i oppgaveteksten. Det er viktig å huske attrributter for koblingen mellom objektene (hus skal ha en liste over etasje-objekter, etasjer skal ha en liste for rom-objekter).

```
public class Rom {
   int areal;
   String betegnelse;
}

public class Etasje {
   String navn;
   ArrayList<Rom> rom = new ArrayList<Rom>();
}

public class Hus {
   String eier;
   String adresse;
   ArrayList<Etasje> etasjer = new ArrayList<Etasje>();
}
```

b) For klassen Hus skal det være mulig å kalle metoden `public int getBoligAreal()` for å få returnert boligarealet i hele huset. Lag denne metoden og evt. andre metoder du mener er nødvendige for å støtte denne funksjonaliteten. NB! Bruk metoder og tenk objekt-orientert.

Foreløpig i oppgaveteksten er det ikke sagt noe om inkapsling så her er det mulig å bruke en nøstet løkke som akkumulerer verdiene fra rommenes areal. Men siden det står at du skal tenke objektorientert og bruke metoder så bør du ha definert en metode for etasje som beregner arealet for etasjen og som Hus-klassen benytter seg av når den beregner arealet for hele huset samt ha definert en getAreal-metode for rom.

```
// Bruk av nøstet løkke :-(

// I Hus-klassen
public int getBoligAreal(){
   int areal = 0;
   for(Etasje etasje: etasjer){
      for(Rom r: etasje.rom){
            areal += r.areal;
      }
   }
   return areal;
}


// Bruk av metoder :-)

// I hus-klassen
public int getBoligAreal(){
   int areal = 0;
   for(Etasje etasje: etasjer){
      areal += etasje.getAreal();
   }
   return areal;
}

// I Etasje-klassen
public int getAareal(){
   int areal = 0;
   for(Rom r: rom){
      areal += r.getAreal();
   }
   return areal;
}

// I Rom-klassen
public int getAreal(){
   return areal;
}
```

c) Du ønsker å lage kode som gjør at det *ikke* skal være mulig å instansiere rom uten at disse knyttes til en etasje og at det *ikke* skal være mulig å instansiere etasjer uten at disse knyttes til et hus. Forklar og vis med kode hvordan du kan sikre at bare objekter med gyldig tilstand skal kunne instansieres. Hint: vi er interessert i hvordan du vil implementere klassenes konstruktør(er) for å oppnå dette.

Her spørres det etter konstruktører for Etasje og Rom som tar inn hhv. et hus-objekt og et etasje-objekt som parameter. Konstruktøren i Etasje skal sørge for at etasjen (referert til vha. this) legges til etasje-listen i hus, og konstruktøren i Rom-klassen skal sørge for at et rom legges til rom-listen til en etasje. Her bør du i tillegg tenke innkapsling: definere liste-attributtene i hhv. Hus og Etasje som private og bruke add-metoder. Når du definerer en egen konstruktør med parameter så vil ikke Java opprette en default konstruktør med tom parameterliste. For å instansiere objekter må derfor konstruktørene du har definert benyttes.

```
// I hus-klassen
private ArrayList<Etasje> etasjer = new ArrayList<Etasje>();

public void addEtasje(Etasje et){
   etasjer.add(et);
}

// I Etasje-klassen
private ArrayList<Rom> rom = new ArrayList<Rom>();

public void addRom(Rom r){
   rom.add(r);
}

// Konstruktør
public Etasje(Hus hus){
   hus.addEtasje(this);
}

// I Rom-klassen

// Konstruktør
public Rom(Etasje et){
   et.addRom(this);
}
```

d) Forklar og vis med kode hvordan du kan sikre at et rom kun er assosiert til *en* etasje og hvordan du kan sikre at et et hus ikke inneholder flere etasjer med samme navn.
Hint: i denne oppgaven kan du bygge videre på deloppgave c).

Dette er i utgangspunktet to forskjellige problemstillinger. For koblingen etasje og rom er vi interessert i å sjekke at samme objekt ikke ligger i flere etasjer. Slik klassene er implementert i løsningsforslaget over er det i praksis ingenting som hindrer deg i å kalle addRom-metoden på et etasje-objekt med hvilket som helst rom-objekt. Dette kan løses ved å la rom-objekter "huske" hvilken etasje konstruktøren ble kalt med og implementere en metode som addRom i etasje kan benytte seg av for å teste på om objektet tilhører en annen etasje. I forslaget under tester vi enkelt og greit på om et rom er assosisert med en etasje med en metode som returnerer en boolean (men dette kan også løses ved å returnere etasjen til et rom etc): etasje.addRom() må da kalles før rom-objektenes etasje-attributt settes.

```
// I Rom-klassen
Private Etasje etasje;

public Rom(Etasje etasje){
      etasje.addRom(this);
      this.etasje = etasje;
}
```

```
public boolean harEtasje(){
   if (etasje == null){
      return false;
   }
   return true;
}
```

```
// I Etasje-klassen

public void addRom(Rom r){
   if (! r.harEtasje()){
      rom.add(r);
   }
}
```

For koblingen hus og etasje er vi interessert i å kontrollere at det ikke finnes objekter med like verdier i navneattributtet koblet til samme hus. NB! Det hadde sikkert vært på sin plass med samme mekanisme som for rom-etasje, men det er ikke det vi spør om her. I dette tilfellet er det tilstrekkelig med en sjekk i Hus-klassens addEtasje-metode på om det finnes objekter med tilsvarende navn fra før. For at dette skal virke må navn settes FØR addEtasje kalles i konstruktøren!!

```
// I Hus-klassen

public void addEtasje(Etasje et){
      boolean addtest = true;
      for (Etasje eksisterende: etasjer){
            if (eksisterende.getNavn().equalsIgnoreCase(et.getNavn())){
                  addtest = false;
            }
      }
      if (addtest){
            etasjer.add(et);
      }
}
```

```
// I Etasje-klassen
private String navn;

public String getNavn(){
      return navn;
}

public Etasje(Hus h, String navn){
      this.navn = navn;
      h.addEtasje(this);
}
```

e) Du ønsker å kunne støtte automatisk nummerering av rom-objekter etter hvert som de instansieres. Det første rom-objektet som instansieres skal få nummer 1, det andre objektet 2 etc. Forklar og vis med kode hvordan du kan implementere dette i Rom-klassen (og kun i denne klassen).

Denne oppgaven løses ved bruk av en klassevariabel (variabel deklarert som static) som inkrementeres hver gang et objekt instansieres. Nummeret til et hus må lagres i en variabel som ikke er static og det er naturlig at både teller og nummer er private, mens det defineres en get-metode for nummer.

```
// I Rom-klassen

// attributter
private static  int teller = 1;
private int nummer;

// i konstruktøren
nummer = teller++;

// get-metode
public int getNummer(){
      return nummer;
}
```

f) Hva er unntakshåndtering (exceptions)? Forklar kort når du bør bruke unntakshåndtering (her er vi ute etter generelle regler, men inkluder gjerne eksempler).

Unntakshåndtering er en mekanisme som benyttes for å håndtere uønskede situasjoner. Dette benyttes i programmering fordi det er ryddigere å lage kode for normaltillfeller og håndtere unntakene som spesialtilfeller. Vi bruker med andre ord unntakshåndtering for å håndtere unntaksvise situasjoner. Eksempler på bruk av unntak kan være for å håndtere feil som oppstår under lesing/skriving til fil, håndtering av refereanser som har null-verdier, håndtering av "ulovlige" situasjoner som vil gi inkonsistens i systemet m.m.

g) Hvis et program prøver å legge samme rom-objekt til to etasjer skal det kastes et unntak. Vis ved hjelp av kode hvordan du vil deklarere en unntaksklasse for denne situasjonen og forklar med tekst eller kode hvordan du ville benyttet denne i koden fra oppgave d).

Unntaks-klassen kan defineres som vist under, men her er det mulig å variere navn, hvilke konstrutør(er) som defineres etc.

```
public class RomUnntak extends Exception {

   public RomUnntak(String melding){
      super(melding);
   }

}
```

Et unntak må kastes i etasje-klassens addRom-metode med riktig testing, bruk av throw og throws.

```
// I Etasjeklassen:
public void addRom(Rom r) throws RomUnntak{
   if (! r.harEtasje()){
      rom.add(r);
   }else{
      throw new RomUnntak(”Rommet tilhører allerde en etasje!”);
   }
}
```

Vi er ikke interessert å fange unntaket i rom-klassens konstruktør fordi dette ville forårsaket at nye objekter kunne bli instansiert uten å være assosiert til en etasje. Derfor videresender vi unntaket ved å definere at konstruktøren “throws RomUnntak”.

```
// I Rom-klassen
public Rom(Etasje etasje) throws RomUnntak{
   this.betegnelse = betegnelse;
   this.areal = areal;
   etasje.addRom(this);
   this.etasje = etasje;
   this.nummer = teller++;
}
```

Eventuelle unntak må fanges opp (og håndteres) i koden som instansierer rom-objekter. Her er dette vist en enkel main-metode og det er gjøres selvsagt ved hjelp av try-catch blokker.

```
// I en main-metode
Hus h1 = new Hus();
Etasje et1 = new Etasje(h1, "første etasje");
try{
   Rom r1 = new Rom(et1, 22, "stue");
}catch(RomUnntak e){
   System.out.println(e.getMessage();
}
```

h) I tillegg til hus ønsker du at objektmodellen din skal støtte objekter av typen leilighet. En leilighet kan på samme måte som hus inneholde flere etasjer, men en leilighet vil bestandig være del av et hus. Lag en superklasse Bolig som har Hus og Leilighet som subklasser og forklar hvilke metoder/felter du vil ha i superklassen og hvilke du vil ha i subklassen.

Her er vi stort sett ute etter superklassen og hvilke felter og metoder du mener det er ryddig å flytte opp i superklassen. Denne oppgaven kan tolkes forskjellig og vi er ute etter din forståelse av superklasser og subklasser. Subklassen Hus må i tillegg utvides med en liste som kan inneholde leiligheter. For å ivareta riktig kobling mellom leilighet og hus er det også her behov for en tilsvarende mekanisme som for rom og etasjer, men det kreves ikke at du sier noe om hvordan denne skal implementeres. Det er naturlig å deklarere denne listen som protected i superklassen eller som private og da med protected tilgangsmetoder slik at kun subklasser har tilgang til etasje-lista. Det er også mulig å definere denne som abstrakt med addEtasje som abstrakt metode.

```
public class Bolig {

   private String eier;

   protected ArrayList<Etasje> etasjer = new ArrayList<Etasje>();

   public Bolig(String eier){
      this.eier = eier;
   }

   public getEier(){
      return eier;
   }

}
```

OPPGAVE 3 (30%): Grensesnitt og abstrakte klasser

I denne oppgaven skal du implementere forskjellige figurklasser og vise og forklare hvordan grensesnitt og abstrakte klasser kan benyttes.

a) Hva er et grensesnitt (interface) og hva bruker vi grensesnitt til i programmering?
Lag et Java interface (`interface`) kalt `FigurGrensesnitt`. Objekter av denne typen skal tilby metoder som returnerer fargen, arealet og typen til en figur ved hjelp av metodene `public String getFarge()`, `public double getAreal()` og `public String getType()`. Med type mener vi her en tekststreng som beskriver hva slags figur det er (*"firkant"*, *"rektangel"* eller *"sirkel"*).

I objektorientering genererlt er et grensesnitt er et sett med samhørende metoder uavhengig av implementasjonen av en klasse og metodekroppene. I java er et grensesnitt (interface) ett sett med konstanter og abstrakte metoder (metodesingaturer uten implementasjon). Vi bruker grensesnitt når det kan være flere varianter av samme logiske funksjon/tjeneste og grensesnittet fungerer som en "type" som er uavhengig av implementasjon.

Grensesnittet i oppgaven defineres slik:

```
public interface FigurGrensesnitt {
   public double getAreal();
   public String getType();
   public String getFarge();
}
```

b) Lag klassene Rektangel og Sirkel som begge implementerer grensesnittet `FigurGrensesnitt`. Følgene metoder skal også implementeres i klassene:

Rektangel: `public int getHøyde()` og `public int getBredde()`

Sirkel: `public int getRadius()` og `public double getOmkrets()`

Alle felter skal være innkapslet og det skal *kun* være mulig å sette objektenes verdier ved instansiering.

PS! For den som ikke husker formlene for en sirkels omkrets og areal så er omkrets = $2*\pi*r$ og arealet = $\pi*r^2$. Tips: Du kan benytte `Math.PI` i Java.

Dette er en rett frem implementering av klassene hvor det kreves at "implements" er brukt og at du har laget implementasjoner av metodene definert i grensesnittet samt andre metoder som er nevnt i oppgaveteksten.
Siden det står at det kun skal være mulig å sette objektenes verdier ved instansiering må attributtene deklareres som "private". Hvis man velger å lage set-metoder må også disse være private.

```
public class Rektangel implements FigurGrensesnitt{

   private int høyde, bredde;
   private String farge;
   static private String type = "firkant";

   public Rektangel(String farge, int høyde, int bredde){
      this.farge = farge;
      this.høyde = høyde;
      this.bredde = bredde;
   }

   public int getFarge(){
      return farge;
   }

   public int getHøyde(){
      return høyde;
   }

   public int getBredde(){
      return bredde;
   }

   public double getAreal(){
      return høyde * bredde;
   }

}

public class Sirkel implements FigurGrensesnitt{

   private int radius;
   private String farge;
   static private String type = "sirkel";

   public Sirkel(String farge, int radius){
      this.farge = farge;
      this.radius = radius;
   }
```

```
public int getRadius(){
      return radius;
   }

   public double getAreal(){
      return Math.PI * radius * radius;
   }

   public double getOmkrets(){
      return Math.PI * 2 * radius;
   }

}
```

c) Lag en abstrakt klasse AbstraktFigurImpl som implementerer FigurGrensesnitt og som Rektangel og Sirkel er subklasser av. Forklar hvordan AbstraktFigurImpl, Rektangel og Sirkel bør være mht. metodene og feltene som er beskrevet i deloppgavene over. Også i denne deloppgaven skal alle felter være innkapslet og det skal kun være mulig å sette verdier ved instansiering.

Her er det unødvendig å deklarere getAreal som abstrakt metode siden denne allerede er definert i FigurGrensesnitt. Det eneste attributtene som det er hensiktsmessig å flyttes til den abstrakte klassen er farge. Det kan også defineres et typeattributt i den abstrakte klassen selv om dette evt. da gjøres for å støtte oppgave d). Det er også behov for en konstruktør for å sette farge siden vi har sagt at attributtene kun skal være lesbare, eller du kan definere farge som protected slik at også subklassene har tilgang til dem (men dette er en dårligere løsning). Hvis det benyttes kall til den abstrakte klassens konstruktør må super(…..) stå som første linje i subklassens konstruktør

```
// AbstraktFigurImplementasjon

public abstract class AbstraktFigurImpl implements FigurGrensesnitt{

   private String farge, type;

   public String getFarge(){
      return farge;
   }

   public String getType(){
      return type;
   }

   public AbstraktFigurImpl(String farge, String type){
      this.farge = farge;
      this.type = type;
   }

}
```

```
// Rektangel subklassens konstruktør

public Rektangel(String farge, int høyde, int bredde){
   super(farge, type);
   this.høyde = høyde;
   this.bredde = bredde;
}

// Sirkel subklassens konstruktør

public Sirkel(String farge, int radius){
   super(farge, type);
   this.radius = radius;
}
```

d) I denne oppgaven skal du lage `toString`-metoder som kan brukes for å få skrevet ut informasjon om figur-objekter. For alle objekter skal farge, type og areal skrives ut. I tillegg skal høyde og bredde skrives ut for rektanglar mens det for sirklar skal skrives ut radius og omkrets. Utskriften for hhv. sirkler og rektangler skal se slik ut:

"rød sirkel, areal = 78.53, radius = 5, omkrets =31.41"
"grønn rektangel, areal = 12, høyde = 3, bredde = 4"

Forklar og vis med kode hvordan du implementerer `toString`-metoder i den abstrakt klassen og i de konkrete figurklassene (Rektangel og Sirkel) slik at den abstrakte klassen har ansvar for å lage første del av teksten (eks. "*rød sirkel, areal 78.53*") mens de konkrete klassene har ansvaret for å legge til infomasjonen som er spesifikk for hver klasse (for eksempel radius og omkrets for sirkler).

Vis bruk av `toString`-metoden ved at du lager en enkel main-metode hvor du oppretter en tabell (array) av figurer og bruker ei løkke for å skrive ut informasjon om figurene.

Her er det viktig å vite at det er subklassenes toString-metode som hvis f.eks. println(sirkel) benyttes. Denne må så gjøre et eksplisitt kall til superklassens toString-metode og konkatenere strengene.

```
// I Rektangel

public String toString(){
   return (super.toString() + ", høyde = " +  høyde +
                              ", bredde = " +  bredde);
}

// I AbstraktFigurImplementasjon

public String toString(){
   return farge + ", " + type + ", areal = " + getAreal();
}
```

```
// I en main-metode:

FigurGrensesnitt[] figurer = new FigurGrensesnitt[4];

figurer[0] = new Sirkel("rød", 2);
figurer[1] = new Sirkel("blå", 3);
figurer[2] = new Rektangel("rød", 2, 4);
figurer[3] = new Rektangel("hvit", 1, 2);

for (int i = 0; i < figurer.length; i++){
      System.out.println(figurer[i]);
}
```

OPPGAVE 4 (10%): Regler for oppførsel og testing

a) Definer presise og etterprøvbare regler for oppførselen til metoden som er beskrevet i oppgave 1 d) : `boolean areAnagrams(String str1, String str2).`
I tillegg til den beskrivelsen som finnes i oppgave 1 d) skal du også definere andre regler du mener er aktuelle for en sikker og robust metode (her menes regler som kan utledes fra det som er skrevet om metoden, eller regler som du selv mener er relevant).

Fra oppgaveteksten
To ord eller setninger som er anagrammer skal gi true
To ord eller setninger som ikke er anagrammer skal gi false
Skal gi true uansett om det er forskjellig bruk av stor og små bokstaver
Skal gi true uansett om det er forskjellig bruk av mellomrom

Andre regler som kan være relevante
To helt like strenger er ikke anagrammer
En tom streng er ikke et anagram for en annen tom streng
Skal gi false hvis en eller begge strengene er null (hvis man mener dette er riktig håndtering av nullstrenger i metoden)
Skal gi false hvis det er brukt forskjellige nummer hvis man mener det er en riktig regel – eller true hvis man mener dette er korrekt håndtering av denne situasjonen.

b) Skriv en eller flere testmetoder i en tenkt TestCase-subklasse med JUnit-rammeverket, som til sammen tester `areAnagrams`-metoden. Relevante metoder fra TestCase-klassen er: `assertEquals(Object, Object)`, `assertTrue(boolean)`, `assertFalse(boolean)` og `assertNull(Object)`.

Her er vi ute etter en systematisk testing ved hjelp av assert-metoder basert på de reglene du definerte i deloppgave a).

Appendiks

Klasser og metoder som kan være nyttige i oppgave 1:

Metoder i `String`-klassen:

```
String(char[] value)
// Allocates a new String so that it represents the sequence of
// characters currently contained in the character array argument.

char charAt(int index)
// Returns the char value at the specified index.

int length()
//Returns the length of this string.

replaceAll(String regex, String replacement)
// Replaces each substring of this string that matches the given regular
// expression with the given replacement.
// Denne metoden fungerer i prakis som en søk og erstatt metode og kan
// benyttes til å erstatte en substreng (regex) med en annen streng
// (replacement) f.eks. vil replaceAll("og", "eller") erstatte alle
// forekomster av "og" med "eller". Husk at en streng også kan være ett
// enkelt tegn eller en tom streng.

char[] toCharArray()
// Converts this string to a new character array.

String toUpperCase()
// Converts all of the characters in this String to upper case.

String toLowerCase()
// Converts all of the characters in this String to lower case.

String concat(String str)
// Concatenates the specified string to the end of this string
// (på norsk: metoden legger til den spesifiserte strengen på slutten av
// strengen).
```

Metoder i Arrays-klassen

```
static boolean equals(char[] a, char[] a2)
// Returns true if the two specified arrays of chars are equal to one
another.

static void sort(char[] a)
//Sorts the specified array of chars into ascending numerical order.
```