

Løsning på opgaven stillet til øvelserne 19. september 2003.

Løsning på spørgsmål 1:

Princippet for løsning af opgaven er beskrevet i opgaveformuleringen. Denne måde at repræsentere af ord på er en almindeligt trick som benyttes ofte i compilere og andre systemer som arbejder med tekster. En compiler møder mange forekomster af variabelnavne (metodenavne etc.), og hver gang skal den slå op i en symboltabel for at undersøge »var det en vi kendte i forvejen, og i givet fald hvad referer den til«. Ofte bygger compileren syntakstræer som den så analyserer flere gange og måske optimerer på. Fordelen ved at oversætter navne (eller ord) til tal, er at antal gange tekststreng skal sammenlignes minimeres, og så selvfølgelig at pladsforbruget begrænses betydeligt. Hver gang et navn (ord) mødes i en tekst, skal det sammenlignes med andre navne én gang, nemlig under indlæsning. Al efterfølgende behandling sker ved sammenligning af tal.

Her følger en implementation i Java af klassen Word med konstruktor; andre metoder er beskrevet nedenfor.

```
public class Word {  
  
    public Word(String s)  
        { for( wordNo=0 ; (wordNo < nextWordNo &&  
                    !s.equals(table[wordNo])) ; wordNo++ )  
          {};  
  
          if( wordNo == nextWordNo) {table[nextWordNo++]=s;}; }  
  
    private int wordNo;  
  
    private static final int Max = 10000;  
    private static String [] table = new String[Max];  
  
    public static void main(String [] args) { .. testprogram ...}  
  
}
```

Bemærk at vi for nemheds skyld har droppet check for overløb af tabellen.

Vi skulle også tilføje toString-metode. Den er lige ud ad landevejen:

```
public String toString() {return table[wordNo];}
```

Som vi diskuterede i forelæsningen, er det altid en god ting at skrive en toString-metode for en klasse, også selvom det endelige program ikke forventes at skrive objekter af denne klasse ud. Den er nyttig til testformål, og den tvinger programmøren til at gøre klart hvordan den interne repræsentation skal forstås. I eksemplet her kan man direkte læse at arrayet indeholder den oprindelige string ud for et ord, og at for et givet Word-objekt findes dets streng i celle nr. wordNo. Konstruktoren, som er et

mere kompliceret stykke kode, skal så holdes op imod om den bevarer den egenskab som er udtrykt i toString.

Equals-metoden er, udover at den er nyttig (og nødvendig i de fleste interessante programmer), også med til at tvinge programmøren til at præcisere yderligere egenskaber ved den interne repræsentation.

```
public boolean equals(Object rhs){
    if(rhs==null || getClass() != rhs.getClass())
        return false;
    return wordNo == ((Word)rhs).wordNo;
}
```

Her står desværre en masse uinteressant, men også den essentielle oplysning, at to Words er ens hvis og kun hvis deres wordNo er ens. Altså: Konstruktoren skal sørge for at opretholde en entydning sammenhæng mellem streng og wordNo, dvs. hvis konstruktoren kunne finde på at lægge samme streng, to gange ind i arrayet, så er der tale om en fejl.

Som vi diskuterede i forelæsningen er den måde Javas design tvinger os til at skrive sammenligningsmetoden på en klodset og meget ineffektiv måde. Vi må skrive en metode som arbejder på objekter af Object-klassen, med dynamisk typecheck og typecasting og det hele. Gevinsten er så at diverse generiske klasser som er specificeret i termer af Object også kan bruges for Word-objekter. Uheldigvis bliver behandlingen langsom fordi hvert kald til equals indeholder alle de ekstra ting, vi har nævnt. Til den programmering, vi selv foretager om Word-objekter, er det en fordel at skrive en mere effektiv metode, som forudsætter, at begge dens argumenter (den implicite »selv«-objekt og det som er nævnt eksplicit) er Word-objekter:

```
public boolean wordEquals(Word rhs)
{ return wordNo==rhs.wordNo; }
```

Skulle man lave en realistisk implementation af en Word-klasse ville man ikke benytte sekventiel søgning, som vi har anvendt, men en eller anden form for hash-tabel (kommer vi til senere på kurset). Opslag i en hashtabellen er stort set konstant i tid uafhængigt af, hvor mange objekter der er i den, hvorimod ved sekventiel søgning er den proportional med antallet af objekter.

Hvis vi ser på tekster, hvor antallet af ord øges gradvist gennem teksten (proportionalt med tekstens længde), kan man indse at den samlede tid for blot at læse sig igennem en tekst og oprette Word-objekter hen ad vejen, tager kvadratisk til målt i forhold til tekstens længde.

Endelig en slutbemærkning: Hvis man ønsker et virkeligt effektivt system, som benytter tricket med at repræsentere ord ved numre, vil man nok undlade at indføre en klasse, men blot repræsentere hvert ord som et heltal. Problemet med klassen, og dermed at hvert ord er repræsenteret som et objekt, er at man hver gang skal forfølge en reference og så lokalisere nummeret. Ved at benytte »rå heltal« undgår man denne ekstra reference, man undgår den dyre »new« operation, og »equals« kan undgås helt idet man blot sammeligner de to dataemner direkte ved »==«.

Løsning på spørgsmål 2:

Her følger en løsning, som er forholdsvis let at skrive, men som på mange måder ikke er så elegant. De generelle bemærkninger ovenfor gælder stadig, men det store problem i den valgte løsning er, at det er meget svært at ændre koden hvis den maksimale sekvenslængde øges fra 4 til 5!!!

```
public class Sequence {
    // Sequences of Words - of length 1 to 4
    // NB: The number number 4 is hardwired into the code
    private Word[] words = new Word[4];
    private int noOfWords;

    public Sequence(Word W0, Word W1, Word W2, Word W3 )
    {noOfWords=4;words[0]=W0;words[1]=W1;words[2]=W2;words[3]=W3;}
    public Sequence(Word W0, Word W1, Word W2)
    {noOfWords=3; words[0]=W0; words[1]=W1; words[2]=W2;}
    public Sequence(Word W0, Word W1)
    {noOfWords=2; words[0]=W0; words[1]=W1;}
    public Sequence(Word W0)
    {noOfWords=1; words[0]=W0;}

    public String toString()
    {return noOfWords==1 ? words[0].toString()
        : noOfWords==2 ? words[0].toString() + " "
            + words[1].toString()
        : noOfWords==3 ? words[0].toString() + " "
            + words[1].toString() + " " +
            words[2].toString()
        //: noOfWords==4 ?
        : words[0].toString() + " " + words[1].toString()
            + " " + words[2].toString()
            + " " + words[3].toString()
    ;}

    // To provide an equals method as required for subclasses of
    // Object we *need* to write it in the following hopeless way:
    public boolean equals(Object rhs){
        if(rhs==null || getClass() != rhs.getClass())
            return false;
        if( noOfWords != ((Sequence) rhs).noOfWords) return false;
        return noOfWords==1 ? words[0].equals(((Sequence) rhs).words[0])
            : noOfWords==2 ? words[0].equals(((Sequence) rhs).words[0])
                && words[1].equals(((Sequence) rhs).words[1])
            : noOfWords==3 ? words[0].equals(((Sequence) rhs).words[0])
                && words[1].equals(((Sequence) rhs).words[1])
                && words[2].equals(((Sequence) rhs).words[2])
            //: noOfWords==4 ?
            : words[0].equals(((Sequence) rhs).words[0])
                && words[1].equals(((Sequence) rhs).words[1])
                && words[2].equals(((Sequence) rhs).words[2])
                && words[3].equals(((Sequence) rhs).words[3])
    ;}
}
```

```

// For our own purposes, we do with:

public boolean sequenceEquals(Sequence rhs)
{if( noOfWords != rhs.noOfWords) return false;
  return noOfWords==1 ? words[0].wordEquals(rhs.words[0])
    : noOfWords==2 ? words[0].wordEquals(rhs.words[0])
      && words[1].wordEquals(rhs.words[1])
    : noOfWords==3 ? words[0].wordEquals(rhs.words[0])
      && words[1].wordEquals(rhs.words[1])
      && words[2].wordEquals (rhs.words[2])
    //: noOfWords==4 ?
    : words[0].wordEquals(rhs.words[0])
      && words[1].wordEquals(rhs.words[1])
      && words[2].wordEquals(rhs.words[2])
      && words[3].wordEquals (rhs.words[3])
  ;}

public static void main(String [] args) {
  // small testprogram
  Sequence fido = new Sequence( new Word("Fido"),
    new Word("er"), new Word("en"), new Word("Hund"));
  System.out.println(fido);

  }}

-----

public class SequenceCount {

  public static void main(String [] args)
  { EasyFile file = new EasyFile("duckling");
    file.openEasyFileForRead();
    Word w1;
    Word w2 = new Word(file.readStringEasy());
    Word w3 = new Word(file.readStringEasy());
    Word w4 = new Word(file.readStringEasy());

    while( ! file.endOfEasyFile() )
      { w1 = w2; w2 = w3; w3 = w4;
        w4 = new Word(file.readStringEasy());

        countSequence(new Sequence(w1));
        countSequence(new Sequence(w1,w2));
        countSequence(new Sequence(w1,w2,w3));
        countSequence(new Sequence(w1,w2,w3,w4));};

    //the end: loop stopped because nothing to shift into w4
    // special treatment of final Sequences not
    // counted above
    countSequence(new Sequence(w2));
    countSequence(new Sequence(w2,w3));
    countSequence(new Sequence(w2,w3,w4));
    countSequence(new Sequence(w3));
    countSequence(new Sequence(w3,w4));
    countSequence(new Sequence(w4));
  }
}

```

```

        // print results:
        int treshold = 3;
        for(int i=0; i<nextUnusedEntry; i++)
            { if (countArray[i]>=treshold)
                System.out.println( sequenceArray[i] + ": " +
                    countArray[i]); } };

// We use two arrays to define mapping from sequences
// to no. of occurrences,

static Sequence [] sequenceArray = new Sequence [100000];

static int [] countArray = new int [100000];

static int nextUnusedEntry=0;

private static void countSequence(Sequence s)
{ int entry;

    for( entry=0 ; (entry < nextUnusedEntry &&
        !s.sequenceEquals(sequenceArray[entry]))
        ; entry++ )
        {};

    if( entry == nextUnusedEntry)
        {sequenceArray[nextUnusedEntry]= s;
        countArray[nextUnusedEntry++]=1;}
    else
        countArray[entry]++;
}

}

```

Svar på spørgsmål 3 handlede om, hvor nemt(eller besværligt) det ville være at 1) indføre et begreb at stopord og 2) at ændre optællingen, så man under optællingen så bort fra sekvenser som starter eller slutter med stopord.

Der foreligger ikke en programtekst med udvidelserne, men her er nogle idéer:

Følgende ide *jeg ikke har undersøgt om virker:*

Man kan indføre en underklasse af Word for stopord, StopWord. Man kunne så lave indlæsningen så snedig at man først indlæste alle stopord fra en stopordfil og oprettede dem med »new StopWord(...)«. Dernæst kunne maskineriet køre videre som sædvanligt, blot sådan at »new Word« returnerer et StopWord-objekt i de relevante tilfælde. *Ekstraopgave for de interesserede: Undersøg om det sidste overhovedet er muligt i Java!*

Følgende er afprøvet: Vi indfører den konvention, at stopord ligger først i tabellen, og vi har en ekstra tæller som holder styr på hvor langt stopordene rækker i tabellen. De efterfølgende ord er så ikke-stopord. Denne repræsentation skal så skhules ved at man indfører metoder som ved at sammeholde et ords wordNo med den nævnte tæller kan sige om det er et stopord eller et interessant ord.

Disse metoder benyttes så i sekvens-klassen til at implementere en metode som afgør om et segment er interessant eller ej. Denne metode kan så benyttes i tællerværket for at afgøre om et segment skal tælles med eller ej.

