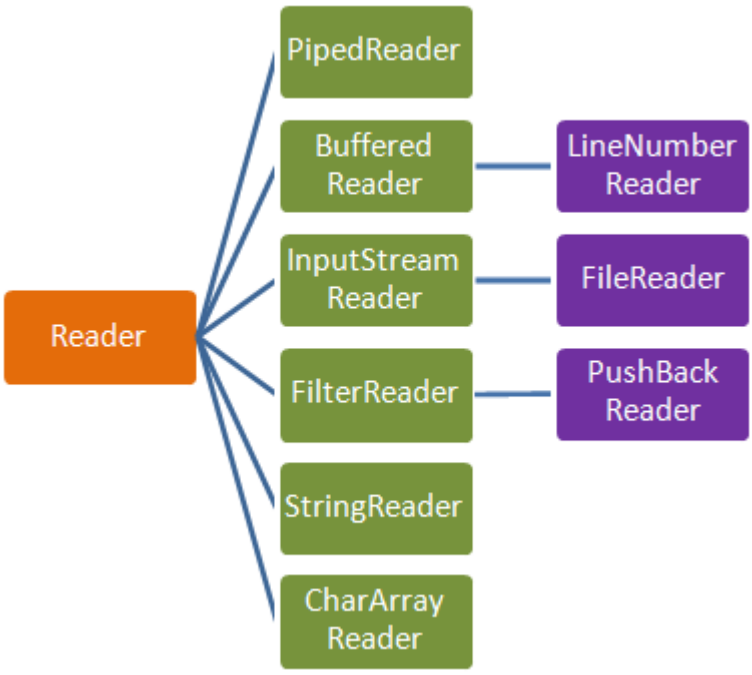
Algoritmer og datastrukturer
Vedlegg A.4 Filbehandling på *char*-nivå

A.4 Filbehandling på *char*-nivå

A.4.1 Reader-klassene

`Reader` er en abstrakt basisklasse for de klassene som leser data på *char*-nivå. Den definerer det minimum av metoder som slike klasser skal ha. På tegningen under er en del av klassehierarkiet til `Reader` satt opp:



Figur A.4.1 a) En del av klassehierarkiet til Reader

Den abstrakte klassen `Reader` inneholder flg. offentlige metoder:

```
public abstract class Reader
{
  public int read() throws IOException
  public int read(char[] c) throws IOException
  public int read(CharBuffer target) throws IOException
  public abstract int read(char[] c, int off, int len) throws IOException
  public long skip(long n) throws IOException
  public boolean ready() throws IOException
  public abstract void close() throws IOException
  public void mark(int readAhedLimit) throws IOException
  public void reset() throws IOException
  public markSupported()
}
```

Programkode A.4.1 a)

`InputStream`-klassene arbeider på *byte*-nivå og kan derfor lese alle filer. `Reader`-klassene arbeider på *char*-nivå og brukes derfor normalt kun til tekstfiler. De må leses vha. samme tegnsett som de ble skrevet med. Hvis ikke kan det bli problemer, f.eks. med Æ, Ø og Å.

Her skal vi kun se på noen av Reader-klassene. De mest brukte av dem er nok FileReader sammen med en BufferedReader.

1. **FileReader** har flg. tre konstruktører (og ellers ingen metoder utover de som arves):

```
public FileReader(String fileName) throws FileNotFoundException
public FileReader(File file) throws FileNotFoundException
public FileReader(FileDescriptor fd)
```

Programkode A.4.1 b)

I flg. eksempel opprettes en instans av FileReader med fil.txt som filnavn:

```
public static void main(String[] args) throws IOException
{
  FileReader f = new FileReader("fil.txt");  // oppretter en instans
  System.out.println(f.getEncoding());       // tegnsett
  f.close();                                 // lukker
}
```

Programkode A.4.1 c)

Ingen av konstruktørene til FileReader krever at vi eksplisitt oppgir et tegnsett. Utskriften er derfor avhengig av hva slags tegnsett ens Java-prosjekt er satt opp med. Et utviklingsmiljø (som f.eks. NetBeans, Eclipse eller IntelliJ) bruker nok UTF-8 som standard (default), men det kan vi forandre hvis det er ønskelig. Hva som er standard, kan du også finne ut ved flg. kode:

```
System.out.println(Charset.defaultCharset().name());
```

Vi får normalt et problem hvis den filen som skal leses vha. en instans av FileReader, er laget med et annet tegnsett, f.eks. med ISO-8859-1 som ble mye brukt tidligere. Men det finnes massevis av andre tegnsett. Java 1.8 har støtte for 170 forskjellige. Se *Oppgave* 1.

2. **InputStreamReader** Hvis filen som skal leses, bruker et annet tegnsett enn det som er standard hos oss, kan vi f.eks. bruke en InputStreamReader. Klassen har fire konstruktører:

```
public InputStreamReader(InputStream in)
public InputStreamReader(InputStream in, Charset cs)
public InputStreamReader(InputStream in, CharsetDecoder dec)
public InputStreamReader(InputStream in, String charsetName)
```

Programkode A.4.1 d)

Hvis filen fil.txt f.eks. er laget med ISO-8859-1 (Latinsk alfabet nr. 1), kan det gjøres slik:

```
InputStream is = new FileInputStream("fil.txt");
String latin1 = "ISO-8859-1";
InputStreamReader inn = new InputStreamReader(is, latin1);
// kode som leser filen
inn.close();  // lukker
```

Programkode A.4.1 e)

InputStreamReader har kun én metode i tillegg til de den arver fra *Programkode* A.4.1 *a)*. Det er metoden *getEncoding*() som vi brukte i *Programkode* A.4.1 *c)*. Det fungerte der siden FileReader er en subklasse til InputStreamReader. Se *Figur* A.4.1 *a)*.

3. BufferedReader En FileReader (eller InputStreamReader) brukt som i *Programkode* A.4.1 *c)* (eller i *Programkode* A.4.1 *e*), er normalt ineffektiv hvis vi leser enkeltverdier, dvs. bruker metoden *read*(). Klassen BufferedReader «reparerer» det problemet. En instans av den inneholder et buffer i form av en *char*-tabell. Det vil kunne fylles opp med kun én filaksess (eller med noen få). Dette er imidlertid avhengig av størrelsen på bufferet og på hva slags medium filen ligger på. Bufferet har 8192 som standard størrelse (Java 1.8).

BufferedReader har kun to konstruktører og begge med en Reader som argument. Den første ber om bufferstørrelse, mens den andre bruker standardstørrelsen:

```
public BufferedReader(Reader in, int size)
public BufferedReader(Reader in)
```

Programkode* A.4.1 *f)

BufferedReader har to metoder utover de som arves fra Reader. Det er

```
public Stream<String> lines()    // samtlige linjer
public String readLine()         // en og en linje
```

Programkode* A.4.1 *g)

Hvis filen fil.txt ikke inneholder flere linjer enn at en utskrift til konsollet ikke tar for stor plass, vil flg. kode demonstrere bruken av metoden *lines*():

```
BufferedReader inn = new BufferedReader(new FileReader("fil.txt"));
inn.lines().forEach(System.out::println);
inn.close();
```

Programkode* A.4.1 *h)

Metoden *forEach*() i Stream har en Consumer som argument. Se *Avsnitt* 1.9.2.

Vi kan få til det samme ved hjelp av metoden *readLine*(). Hvis det ikke er flere linjer igjen, returnerer den null:

```
BufferedReader inn = new BufferedReader(new FileReader("fil.txt"));
String linje;
while ((linje = inn.readLine()) != null) System.out.println(linje);
inn.close();
```

Programkode* A.4.1 *i)

Obs: En tekstfil består av linjer og hver linje avsluttes med kode som markerer linjeslutt. Men koden er plattformavhengig. Under Windows (en PC) er det tegnene CR (carriage return) og LF (line feed) med tallverdier 13 og 10. På et Unix-system er det kun LF og på en Mac kun CR. Men dette trenger vi ikke tenke på. Dette ordnes internt.

Ingen av de to konstruktørene til BufferedReader ber om tegnsett. Dermed skjer lesingen med det tegnsettet som er standard. Men hva hvis filen bruker et annet tegnsett? Da kan vi bruke samme teknikk som i *Programkode* A.4.1 *e)*:

```
InputStream is = new FileInputStream("fil.txt");
String latin1 = "ISO-8859-1";
InputStreamReader isr = new InputStreamReader(is, latin1);
BufferedReader inn = new BufferedReader(isr);
// kode som leser filen
inn.close();
```

Programkode* A.4.1 *j)

`Reader`-klassene har vært med omtrent siden starten (fra Java 1.1). I Java 1.7 kom det en del nye teknikker for filbehandling. Samleklassen `Files` (se også *Vedlegg* A.6.2) inneholder en serie praktiske metoder som gjør det enklere for oss. Følgende metode lager en instans av `BufferedReader` der en vei (`Path`) og et tegnsett (`Charset`) inngår som argumenter:

```
Path vei = Paths.get("fil.txt");
Charset latin1 = Charset.forName("ISO-8859-1");
BufferedReader inn = Files.newBufferedReader(vei, latin1);
// kode som leser filen
inn.close();
```

Programkode* A.4.1 *k)

4. **`CharArrayReader`** og **`StringReader`** er de to siste vi skal se på fra `Reader-hierarkiet`. De brukes til å lese innholdet av hhv. en *char*-tabell og en tegnstreng som om det var filer. Den første har to konstruktører og den andre kun én:

```
public CharArrayReader(char[] buf)
public CharArrayReader(char[] buf, int offset, int length)
public StringReader(String s)
```

Flg. eksempel viser hvordan en `CharArrayReader` kan brukes. Se *Oppgave x* når det gjelder `StringReader`. Husk at metoden *read*() returnerer en *int*:

```
char[] c = "Dette er en test!\nDette er linje nr. 2.\n".toCharArray();
Reader inn = new CharArrayReader(c);
for (int k = inn.read(); k != -1; k = inn.read()) System.out.print((char)k);
inn.close();  // er egentlig unødvendig her

/* Utskrift:
Dette er en test!
Dette er linje nr. 2.
*/
```

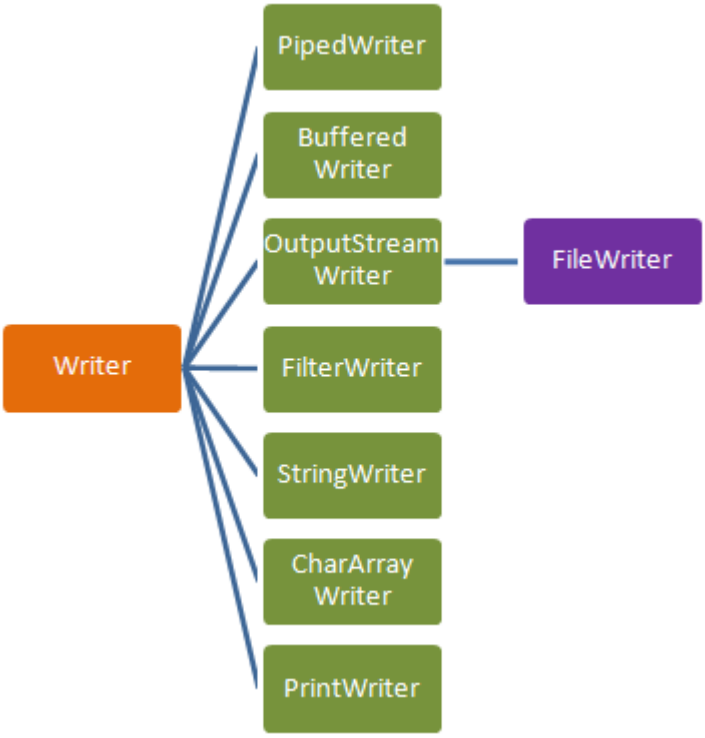
Programkode* A.4.1 *l)

Oppgaver til A.4.1

1. Den statiske metoden *availableCharsets*() i `Charset` har `Map<String,Charset>` som returtype. Bruk den til å få skrevet ut alle tilgjengelige tegnsett.
2. Gjør flg. eksperiment: Lag en fil f.eks. med navn `fil.txt` med UTF-8 som tegnsett og les den så med UTF-8 som tegnsett. Pass på at filen inneholder Æ, Ø og Å - både store og små. Bruk deretter ISO-8859-1 begge steder. La så filen ha UTF-8 som tegnsett og les den med ISO-8859-1. Gjør til slutt det motsatte. Hvis du bruker en teksteditor, vil den normalt gi deg muligheten av å velge tegnsett når det du har skrevet, skal lagres. Det er også mulig å velge metoden *newBufferedWriter*() fra klassen `Files`. Den har et tegnsett som argument.
3. Sett deg inn i hvordan tegnsettene er definert. Du finner en masse om dette på internett. Se spesielt på ISO-8859-1, UTF-8 og UTF-16.
4. Gjør de endringene som er nødvendig i *Programkode* A.4.1 *l)* for at det isteden handler om en `StringReader`. Behold tegnstrengen, dvs. ta vekk *toCharArray*().

A.4.2 Writer-klassene

`Writer` er en abstrakt basisklasse for de klassene som leser data på *char*-nivå. Den definerer det minimum av metoder som slike klasser skal ha. På tegningen under er en del av klassehierarkiet til `Writer` satt opp:



Figur A.4.2 a) En del av klassehierarkiet til Writer

Den abstrakte klassen `Writer` inneholder flg. offentlige metoder:

```
public abstract class Writer
{
  public void write(int c) throws IOException
  public void write(char cbuf[]) throws IOException
  public abstract void write(char cbuf[], int off, int len) throws IOException
  public void write(String str) throws IOException
  public void write(String str, int off, int len) throws IOException
  public Writer append(CharSequence csq) throws IOException
  public Writer append(CharSequence csq, int start, int end) throws IOException
  public Writer append(char c) throws IOException
  public abstract void flush() throws IOException
  public abstract void close() throws IOException
}
```

Programkode A.4.2 a)

`OutputStream`-klassene arbeider på *byte*-nivå og kan skrive hva som helst. `Writer`-klassene arbeider på *char*-nivå. Derfor er det av betydning hvilket tegnsett vi bruker. På norsk er det spesielt Æ, Ø og Å som kan skape problemer her.

Her skal kun se på noen av `Writer`-klassene. De mest brukte er `FileWriter`, `BufferedWriter` og `PrintWriter`.

1. `FileWriter` har flg. fem konstruktører (og ellers ingen metoder utover de som arves):

```
public FileWriter(File file)
public FileWriter(File file, boolean append)
public FileWriter(FileDescriptor fd)
public FileWriter(String fileName)
public FileWriter(String fileName, boolean append)
```

Programkode A.4.2 b)

Ingen av konstruktørene har et tegnsett som argument. Det betyr at en utskrift med en `FileWriter` vil skje med det tegnsettet som er standard på ditt system. Klassen har en metode som forteller hva det er:

```
FileWriter ut = new FileWriter("fil.txt");
System.out.println(ut.getEncoding());
ut.close();
```

Programkode A.4.2 c)

Men det er også andre måter å finne ut dette på. F.eks. slik:

```
System.out.println(Charset.defaultCharset().name());
```

Hvis vi skal opprette en instans av `FileWriter` knyttet til en fil med navn f.eks. lik `fil.txt`, så oppstår flere tilfeller. I de to konstruktørene som har *append* som et argument, kan vi la argumentet enten være *true* eller *false*. Velger vi *false*, er det det samme som å bruke en av de andre konstruktørene. Uansett vil det da, hvis det ikke finnes en fil med det navnet fra før, bli opprettet en slik fil. Hvis derimot navnet finnes fra før, vil filen bli «trunkert», dvs. redusert til lengde 0. Velger vi *true* som argument for *append*, blir det annerledes. Hvis det finnes en fil med det navnet fra før, vil utskrift fortsette videre på slutten av filen (appended). Hvis ikke, vil filen bli opprettet og utskrift skjer på vanlig måte. Se flg. eksempel:

```
1   FileWriter ut = new FileWriter("fil.txt");
2   ut.write("Dette er starten på utskriften. ");
3   ut.close(); // lukker
4
5   ut = new FileWriter("fil.txt", true);
6   ut.write("Dette er resten."); // skriver på slutten av filen
7   ut.close(); // lukker
8
9   BufferedReader inn = new BufferedReader(new FileReader("fil.txt"));
10  System.out.println(inn.readLine());
11  inn.close(); // lukker
12   // Utskrift: Dette er starten på utskriften. Dette er resten.
```

Programkode A.4.2 d)

I linje 1 blir filen opprettet hvis den ikke finnes fra før. Ellers blir den trunkert. I linje 2 brukes den versjonen av *write* (se *Programkode* A.4.2 *a*) som har en tegnstreng som argument. I linje 5 brukes den konstruktøren som har filnavn og *append* som argumenter. Siden verdien er *true*, blir ikke filen trunkert hvis den finnes fra før. Det betyr at utskriften i linje 6 fortsetter på slutten av filen. I linje 9 - 12 leses filen på vanlig måte med utskrift til konsollet.

To av konstruktørene har `File` som argumenttype. Les om `File` i *Vedlegg* A.6.1.

I tillegg til *write*-metodene har har Writer, og dermed FileWriter, tre *append*-metoder. De er definert i Appendable, men tilbyr ikke mer funksjonalitet. Hvis ut er en FileWriter og c en *char*, vil ut.append(c) oppføre seg som ut.write(c). Se også *Oppgave 1*.

2. OutputStreamWriter Hvis vi skal skrive til en fil med et annet tegnsett enn det som er standard hos oss, kan vi f.eks. bruke en OutputStreamWriter. Den har fire konstruktører

```
public OutputStreamWriter(OutputStream out)
public OutputStreamWriter(OutputStream out, Charset cs)
public OutputStreamWriter(OutputStream out, CharsetEncoder enc)
public OutputStreamWriter(OutputStream out, String charsetName)
```

Programkode A.4.2 e)

Hvis fil.txt f.eks. skal lages med ISO-8859-1 (Latinsk alfabet nr. 1), kan det gjøres slik:

```
OutputStream os = new FileOutputStream("fil.txt");
String latin1 = "ISO-8859-1";
OutputStreamWriter ut = new OutputStreamWriter(os, latin1);
System.out.println(ut.getEncoding());  // Utskrift: ISO8859_1

// kode som skriver til ut

ut.close();
```

Programkode A.4.2 f)

OutputStreamWriter har kun én metode i tillegg til de den arver fra *Programkode* A.4.2 *a)*. Det er metoden *getEncoding*() som vi har brukt i koden over. Der brukte vi først ISO-8859-1 som navn på tegnsettet. Men utskriften gir oss ISO8859_1, dvs. uten den første bindestreken og med understrekingstegn istedenfor den siste bindestreken. Det er faktisk mange lovlige navn på dette tegnsettet. Se *Oppgave 2*.

3. BufferedWriter har et internt buffer i form av en *char*-tabell. Utskriftsmetodene (*write* og *append*) legger verdiene fortløpende i tabellen og når den er full, skrives den ut til den underliggende utstrømmen. Det fører normalt til vesentlig mer effektiv kode. Den eneste metoden klassen har i tillegg til de som arves, er *flush*(). Den skriver ut det som måtte ligge i den interne tabellen. Det er viktig alltid å avslutte med *close*(). Da trengs ikke *flush*() på forhånd siden et kall på den inngår implisitt i *close*().

Klassen har kun to konstruktører, begge med en Writer som argument. Den første ber om bufferstørrelse, mens den andre bruker standardstørrelsen (som i Java 1.8 er på 8192):

```
public BufferedWriter(Writer out, int size)
public BufferedWriter(Writer out)
```

Hvis en skal skrive f.eks. til filen fil.txt, blir det slik:

```
BufferedWriter ut = new BufferedWriter(new FileWriter("fil.txt"));

// write eller append

ut.close();
```

Programkode A.4.2 g)

Java har hatt BufferedWriter nesten siden starten (fra Java 1.1). I Java 1.7 kom klassen Files (se også *Vedlegg* A.6.2) som inneholder en serie konstruksjonsmetoder. Bla. disse:

```
BufferedWriter newBufferedWriter(Path path, Charset cs, OpenOption... options)
BufferedWriter newBufferedWriter(Path path, OpenOption... options)
```

Her inngår både Path, Charset og OpenOption (se *Vedlegg* A.6). Hvis vi f.eks. skulle ønske å skrive til fil.txt med ISO-8859-1 som tegnsett (hvis vi har noe annet som standard), kan det gjøres slik:

```
Path vei = Paths.get("fil.txt");
Charset latin1 = Charset.forName("ISO-8859-1");
BufferedWriter ut = Files.newBufferedWriter(vei, latin1);
// kode som skriver ut
ut.close();
```

Programkode A.4.2 h)

Legg merke til at i de to metodene inngår argumentet OpenOption... *options*, mens vi ikke har oppgitt noen verdi. Det er ok, men da får vi «standardverdiene», dvs. CREATE, TRUNCATE_EXISTING og WRITE. Det betyr spesielt at hvis filen finnes fra før, blir den trunkert. Hvis filen finnes fra før og vi ønsker å skrive videre på den, kan vi gjøre det slik:

```
Path vei = Paths.get("fil.txt");
BufferedWriter ut = Files.newBufferedWriter(vei, StandardOpenOption.APPEND);
// utskrift vil skje på slutten av filen
ut.close();
```

Programkode A.4.2 i)

Files har *write*-metoder som bruker en BufferedWriter implisitt. Se *Oppgave* 3.

4. En **CharArrayWriter** skriver til en intern *char*-tabell. Den kan vi dimensjonere (den første konstruktøren) eller vi kan bruke standardstørrelsen på 32:

```
public CharArrayWriter(int initialSize)
public CharArrayWriter()
```

Klassen har fire metoder i tillegg til de som arves fra Writer. Metoden *toString*() er overstyrt til å returnere innholdet i den interne tabellen som en String:

```
void reset()                // nullstiller intern tabell
int size()                  // antall i intern tabell
char[] toCharArray()        // kopi av intern tabell
void writeTo(Writer out)    // tabellinnholdet skrives til out
String toString()           // intern tabell som String
```

Programkode A.4.2 j)

Flg. eksempel viser hvordan dette kan brukes:

```
CharArrayWriter ut = new CharArrayWriter(10);  // utvider seg om nødvendig
ut.write("ABCDEFGHIJKLMNO");  // skriver til intern tabell
// ut.close() er unødvendig her - metoden gjør ingenting

System.out.println("Antall tegn: " + ut.size());  // Antall tegn: 15
ut.reset();  // nullstiller - nå er size lik 0
String s = Arrays.toString(ut.toCharArray());
System.out.println(s);   // [A, B, C, D, E, F, G, H, I, J, K, L, M, N, O]
System.out.println(ut);  // ABCDEFGHIJKLMNO
```

Programkode A.4.2 k)

5. I **StringWriter** brukes en **StringBuffer** som internt buffer. En metode returnerer en referanse til det interne bufferet. Det betyr at vi kan bygge en StringBuffer både ved hjelp av metodene fra Writer og fra StringBuffer. F.eks. slik:

```
StringWriter ut = new StringWriter();     // bruker standardstørrelse på 16
StringBuffer sb = ut.getBuffer();         // refrense til internt buffer

ut.write("AB");                           // metode fra Writer
sb.append('D').append('E');               // metode fra StringBuffer
ut.write('F');                            // metode fra Writer
sb.setCharAt(0, 'X');                     // metode fra StringBuffer

System.out.println(ut + "  " + sb);       // Utskrift: XBDEF  XBDEF
```

Programkode A.4.2 l)

6. **PrintWriter** kan brukes til å lage formatert utskrift. Den har nesten de samme metodene som PrintStrem. Se *Avsnitt* A.3.13 og *Vedlegg* B. Klassen har åtte konstruktører:

```
public PrintWriter(File file)
public PrintWriter(File file, String csn)
public PrintWriter(OutputStream out)
public PrintWriter(OutputStream out, boolean autoFlush)
public PrintWriter(String fileName)
public PrintWriter(String fileName, String csn)
public PrintWriter(Writer out)
public PrintWriter(Writer out, boolean autoFlush)
```

De seks første lager en instans av PrintWriter som internt bruker en BufferedWriter. Hvis vi bruker en av de to der det boolske argumentet *autoFlush* inngår, får vi en instans der det implisitt gjøres et kall på *flush*() for hver ny linje. Det normale ellers er at *flush*() kalles først når det interne bufferet er fullt.

Hvis vi kun benytter metodene *append*(), *format*(), *printf*(), *print*() og *println*(), er det hipp som happ om vi bruker PrintWriter eller PrintStream. Det er kun *write*-metodene som er forskjellige. PrintStream skriver ut verdier på *byte*-nivå (enkeltverdier og tabeller), mens PrintWriter skriver ut på *char*-nivå (ekeltverdier, tabeller og tegnstrenger).

Oppgaver til A.4.2

1. Et kall på hvert av de tre *append*-metodene i Writer, kan erstattes med et kall på en write-metode. Vis hvordan det kan gjøres!
2. Finn alle lovlige navn (i Java) på tegnsettet ISO-8859-1. Du kan teste ved hjelp av flg. metodekall: `System.out.println(Charset.isSupported("et navn"));`
3. Lag en liste (f.eks. en ArrayList) som inneholder en samling fornavn (Per, Kari, Ole, osv). Skriv så innholdet av listen til filen `fil.txt` ved hjelp av en *write*-metode i Files. Sjekk så hva filen inneholder.