A.6 File, Files, Path, Paths

A.6.1 File, Path og Paths

`File` og `Path` er abstrakte representasjoner av veien (the path) fra roten (the root) i det hierarkiske filsystemtet ned til filer og mapper (directories). Klassen `File` har vært med siden Java 1.0 og har deretter blitt utvidet flere ganger. Grensesnittet `Path` som kom i Java 1.7, kan ses på som en forbedring og et alternativ til `File`.

Undertegnede bruker ofte `NetBeans` som utviklingsmiljø (IDE) og mappen `C:\NetBeans` har forskjellige NetBeans-prosjekter. Et av dem er `C:\NetBeans\AlgDat`, et prosjekt for emnet Algoritmer og datastrukturer. En kan opprette et `File`-objekt knyttet til `C:\NetBeans\AlgDat` ved f.eks. å bruke konstruktøren som har et (vei)navn (pathname) som argument. Hvis det skal representere en fil (og ikke en mappe), oppgir man navn og vanligvis type (extension). Flg. kode oppretter en instans som representerer filen `fil.txt` under denne mappen:

```
File fil = new File("C:/NetBeans/AlgDat/fil.txt");
```

I Windows brukes tegnet \ (dvs. bakoverskråstrek eller backslash) som skilletegn mellom de ulike nivåene i filhierarkiet, men det kan vi ikke bruke i en Java-tegnstreng. Der er tegnet \ et signal om at det som kommer rett etter, skal tolkes på en spesiell måte. Et eksempel er anførselstegnet. Det kan ikke stå alene inne i en tegnstreng. Men bruker vi \", så går det bra. Vi kan bruke vanlig skråstrek (som i koden over) eller dobbelt skilletegn, dvs. \\. Det første tegnet er da et signal om at det andre skal tolkes som en bakoverskråstrek. Da blir det slik:

```
File fil = new File("C:\\NetBeans\\AlgDat\\fil.txt");
```

Det er viktig å være klar over at `fil` er en referanse til et «veiobjekt» og ikke til en «fysisk» fil. Hvis en f.eks. vha. en filutforsker, undersøker hva som måtte ligge under mappen `AlgDat`, vil en ikke finne noe med navn `fil.txt`. Klassen `File` har imidlertid flere metoder som gir informasjon om «veiobjektet»:

```
File fil = new File("C:/NetBeans/AlgDat/fil.txt");
System.out.println(fil.getCanonicalPath());  // C:\NetBeans\AlgDat\fil.txt
System.out.println(fil.getParent());         // C:\NetBeans\AlgDat
System.out.println(fil.exists());            // false
```

Programkode* A.6.1 *a)

Filhierarkiet er organisert som en trestruktur med filer og mapper som noder. På toppen (som rot) har vi `C:\`. Forelder til `fil.txt` er mappen `AlgDat` eller `C:\NetBeans\AlgDat` hvis vi tar med hele veien. Metoden *`exists`*`()` returnerer `false` siden det ikke er oppettet en «fysisk» fil ennå. Klassen `File` har en egen metode for å opprette filer:

```
File fil = new File("C:/NetBeans/AlgDat/fil.txt");
System.out.println(fil.createNewFile());     // true
System.out.println(fil.exists());            // true
```

Programkode* A.6.1 *b)

Metoden *createNewFile*() returnerer true hvis det blir opprettet en «fysisk» fil og false hvis det på forhånd finnes en fil med samme navn (i den den mappen). Det betyr at hvis vi kjører kodebiten i *Programkode* A.6.1 *b)* to eller flere ganger, vil det komme true kun første gang og så false deretter. Hvis filen av en eller annen årsak ikke lar seg opprette (f.eks. at veien dit ikke eksisterer), kastes en IOException. Metoden *exists*() returnerer true hvis filen finnes og det gjør den i *Programkode* A.6.1 *b)*. Se *Oppgave* 2.

Det vil være tilfeller der vi trenger en midlertidig fil, dvs. plass som vil fjernes/frigjøres etter fullført «jobb». Den kan legges under hvilken mappe vi vil, f.eks. under C:\NetBeans\AlgDat:

```
File fil = new File("C:/NetBeans/AlgDat/fil.txt");
System.out.println(fil.createNewFile());     // true
System.out.println(fil.exists());            // true
fil.deleteOnExit();
```

Programkode* A.6.1 *c)

I koden over vil vi få true i begge linjene hver gang dette eksekveres siden filen ikke finnes når *createNewFile*() utføres. Det er en annen måte å lage midlertidige filer. Da havner den på systemets temporære område. Undertegnede bruker Windows 10 og der blir det slik:

```
File tempfil = File.createTempFile("temp", ".txt");
System.out.println(tempfil.getCanonicalPath());
// C:\Users\Uttersrud\AppData\Local\Temp\temp1811106043363475505.txt
System.out.println(tempfil.exists());  // true
tempfil.deleteOnExit();
```

Programkode* A.6.1 *d)

Legg merke til at metoden *createTempFile*() er en konstruksjonsmetode (factory method), dvs. en statisk metode som returnerer en instans av klassen File. Filnavnet (her temp) må ha minst tre tegn og type kan være null (i så fall blir den satt til .tmp). Se *Oppgave* 3.

Klassen File har mange metoder som kan brukes til å analysere et «veiobjekt», men ingen som gjør det mulig å skrive til eller lese fra en fil som eksisterer. Hvis vi skal skrive, kan vi f.eks. koble «veiobjektet» med et objekt av typen OutputStream eller Writer:

```
File fil = new File("C:/NetBeans/AlgDat/fil.txt");
OutputStream ut = new FileOutputStream(fil);

ut.write('A'); ut.write('B'); ut.write('C');  // filen inneholder nå ABC
System.out.println("Lengde: " + fil.length());

String format = "Oppdatert: %1$tA %1$te. %1$tB %1$tY kl. %1$tH:%1$tM:%1$tS%n";
System.out.printf(format, fil.lastModified());

// Utskrift:
// Lengde: 3
// Oppdatert: fredag 3. november 2017 kl. 11:59:43
```

Programkode* A.6.1 *e)

Hvis filen "fil.txt" ikke finnes fra før, vil FileOutputStream-konstruktøren opprette den. Legg merke til at det er klassen File som har metodene *length*() og *lastModified*(). Den siste returnerer en *long* og den kan formateres - se formatering av dato og klokkeslett.

Så langt har vi sett på filer. Men en instans av klassen File kan også representere en mappe (directory). En undermappe Temp til f.eks. mappen C:\NetBeans\AlgDat, opprettes slik:

```
String ny = "Temp";
File mappe = new File("C:/NetBeans/AlgDat/" + ny);
boolean ok = mappe.mkdir();
System.out.println(ny + " er " + (ok == false ? "ikke" : "") + " opprettet!");
```

Programkode* A.6.1 *f)

Hvis Temp finnes fra før eller veien dit ikke finnes, gir *mkdir*() false. Metoden *mkdirs*() virker på samme måte, men sørger for å opprette manglende mapper hvis veien innholder mapper som ikke finnes. Hvis f.eks. mappen C:\NetBeans\AlgDat\Temp skal opprettes og mappen C:\NetBeans\AlgDat ikke finnes fra før, vil også den bli opprettet.

Vi kan også opprette et «veiobjekt» (en instans av klassen File) ved å oppgi et isolert fil- eller mappenavn. Da kalles det et relativt «veiobjekt»:

```
File mappe = new File("Temp");  // ny mappe
System.out.println(mappe.getCanonicalPath());  // C:\NetBeans\AlgDat\Temp
```

Programkode* A.6.1 *g)

Her vil mappen, hvis den blir opprettet, havne under prosjektmappen C:\NetBeans\AlgDat, dvs. under det som kalles gjeldende område (eng: the current user directory). Tilsvarende blir det med et isolert filnavn. Hvis en i *Programkode* A.6.1 *g)* kaller metoden *getParent*(), får en null som resultat siden den relativt sett ikke har noen forelder. Se også *Oppgave 4*.

Filsystemet har en rotmappe (roten i trestrukturen). En datamaskin kan ha flere filsystemer. Klassen File har en metode som genererer en tabell som inneholder alle rotmappene. For undertegnedes maskin er det slik:

```
File[] filer = File.listRoots();
for (File fil : filer) System.out.print(fil + " ");
// Utskrift: C:\ D:\ E:\ F:\
```

Programkode* A.6.1 *h)

En mappe kan ha fra ingen til mange filer og (under)mapper. Disse kan vi få listet opp ved hjelp av en av flg. metoder:

```
1.  String[] list()
2.  File[] listFiles()
3.  File[] listFiles(FileFilter filter)
4.  String[] list(FilnameFilter filter)
5.  File[] listFiles(FilenameFilter filter)
```

Hvis mappe er en mappeinstans av File (og ikke en fil), får vi alt i mappen ved:

```
File mappe = new File("C:/NetBeans/AlgDat");
for (File fil : mappe.listFiles()) System.out.println(fil);

// C:\NetBeans\AlgDat\build
// C:\NetBeans\AlgDat\build.xml
// C:\NetBeans\AlgDat\fil.txt
// osv.
```

Programkode* A.6.1 *i)

I koden er det den andre av de fem metodene i oppramsingen over som brukes. Den første virker tilsvarende, men gir en tabell av tegnstrenger der hver streng representerer en vei. Vi

kan filtrere resultatet vha. funksjonsgrensesnittet `FileFilter` (et grensesnitt med nøyaktig én abstrakt metode). Metoden har `Fil` som argument og returnerer en *boolean*. Dermed kan vi filtrere ved hjelp av et lambda-uttrykk. I flg. eksempel får vi kun mappene (og ikke filene):

```
File mappe = new File("C:/NetBeans/AlgDat");
for (File fil : mappe.listFiles(f -> f.isDirectory())) System.out.println(fil);

// C:\NetBeans\AlgDat\build
// C:\NetBeans\AlgDat\nbproject
// C:\NetBeans\AlgDat\src
// osv.
```

Programkode* A.6.1 *j)

Vi kan gjøre mange typer filtreringer vha. lambda-uttrykk. Hvis vi ønsker å få ut alle tekstfilene (dvs. de som ender med `.txt`), kan vi f.eks. bruke dette lambda-uttrykket:

```
f -> f.toString().endsWith(".txt")
```

I de to siste av de fem metodene over inngår funksjonsgrensesnittet `FilenameFilter`. I det har den abstrakte metoden to argumenter. Se mer om det i *Oppgave 5*.

`Path` kan ses på som et alternativ til `File`. Flg. metoder konverterer mellom dem:

```
File fil = new File("C:/NetBeans/AlgDat/fil.txt");
Path vei = fil.toPath();    // fra File til Path
File nyfil = vei.toFile();  // fra Path til File
```

Programkode* A.6.1 *h)

`Path` er et grensesnitt og kan ikke instansieres. Men det er flere metoder som returnerer en instans av en intern systemklasse som implementerer dette grensesnittet. Det gjelder, som vist over, metoden *toPath*() i klassen `File`. Samleklassen `Paths` har disse to metodene:

```
static Path get(String first, String... more)
static Patg get(URI uri)
```

Programkode* A.6.1 *i)

I den første *get*-metoden kan vi velge om vi vil oppgi veien i én tegnstreng eller dele den opp. Her er to ytterpunkter:

```
Path vei1 = Paths.get("C:/NetBeans/AlgDat/fil.txt");
Path vei2 = Paths.get("C:", "NetBeans", "AlgDat", "fil.txt");
System.out.println(vei1);  // C:\NetBeans\AlgDat\fil.txt
System.out.println(vei2);  // C:\NetBeans\AlgDat\fil.txt

System.out.println(vei1.getClass());  // class sun.nio.fs.WindowsPath
```

Programkode* A.6.1 *j)

Klassen `FileSystems` har en konstruksjonsmetode (factory method) med argumentliste lik den i *get*-metoden over:

```
Path vei = FileSystems.getDefault().getPath("C:/NetBeans/AlgDat/fil.txt");
```

`Path` har en serie metoder, f.eks. en iterator. Her er et par eksempler (se også *Oppgave 6*) :

```
Path vei = (new File("C:/NetBeans/AlgDat/fil.txt")).toPath();
System.out.println(vei.getRoot());              // C:\
for (Path p : vei) System.out.print(p + " ");  // NetBeans AlgDat fil.txt
```

Programkode* A.6.1 *k)

Oppgaver til A.6.1

1. Tegnene " og \ kan ikke stå alene, men som \" og \\. Se utskriften fra flg. kode:
   `System.out.println("Tegnet \" kalles anførselstegn og \\ skilletegn.");`
2. Metoden *createNewFile*() kaster et unntak hvis «veien» som inngår i konstruksjonen av «veiobjektet» (en instans av klassen File), ikke finnes. Test at det skjer f.eks. ved å gjøre en endring i *Programkode* A.6.1 *b)*.
3. Sjekk hva som skjer i *Programkode* A.6.1 *d)* hvis filnavnet har færre enn tre tegn eller hvis typen er null.
4. Legg inn et kall på *getParent*() i *Programkode* A.6.1 *g)*. Hva returneres? Gjør så et kall på *mkdir*(). Hva returnerer *getParent*() nå? Gjør så om mappe til et absolutt «veiobjekt», f.eks. ved: mappe = mappe.*getAbsoluteFile*() og gjør et nytt kall på *getParent*().
5. Veien eller adressen til en fil kan deles i to: Veien til mappen som inneholder filen og selve filnavnet. Et lambda-utrykk av typen FilenameFilter må ha to argumenter, dvs. være på formen: (mappe, navn) -> .... der mappe er av typen File, navn av typen String og .... noe som returnerer en *boolean*. Bruk dette til å lage kode som til en gitt mappe skriver ut de filene og mappene i denne mappen som har stor forbokstav i navnet sitt.
6. Lag kode som skriver ut alt innhold under en oppgitt mappe. Bruk rekursjon og preorden.
7. a) Filsystemet har en trestruktur. En fil eller mappe ligger på et nivå bestemt av antall «noder» på veien. Metoden *getNameCount*() i Path finner antall navn (eller «noder») på veien, men teller ikke med roten. Gjør et kall på *getNameCount*() i *Programkode* A.6.1 *k)* og sjekk hva den returnerer.

   b) Lag en for-løkke som skriver ut alle navnene på en vei (bortsett fra roten) ved hjelp av metoden *getName*() i Path.

   c) Et relativt «veiobjekt» kan gjøres absolutt ved hjelp av metoden *getAbsoluteFile*(). Se Oppgave 4. Men klasse File har ingen metode som sjekker om objektet er relativt eller absolutt. Men Path har det. Lag en relativ instans av File, gjør den om til en Path og bruk så metoden *isAbsolute*() for å sjekke objektet.
8. En vei kan tolkes som en url. Med lokalversjonen av dette vedlegget i en web-leser vil adressefeltet vise noe som dette: file:///F:/www/appolonius/vedlegg/A/A6.html. Lag kode som gjør om en instans av klassen File til en URL. Gjør det samme med en instans av typen Path. Se også Vedlegg A.5.

A.6.2 Files

Klassen Files er en samleklasse for filmetoder - hele 65 stykker. Flere av dem gjør omtrent det samme som metoder i Path og File, men på en enklere og mer fleksibel måte. Men de fleste tilbyr ny funsjonalitet. Her får vi nøye oss med å se på noen få av dem.

Files har en serie metoder for å opprette filer/mapper (permanente og midlertidige), f.eks:

```
public static Path createFile(Path path, FileAttribute<?>... attrs)
```

Det som skiller seg fra tilsvarende metode i klassen File, er argumenttypen FileAttribute. I argumentlisten står det FileAttribute<?>... og dermed kan argumentet utelates helt. Da får vi standardversjonen av filrettigheter. I klassen File er det metoder for å sette bestemte rettigheter. Her kan vi isteden oppgi atributtverdier. Men da må vi vite hva slags filsystem som inngår. Vi har PosixFileAttributes for UNIX-varianter og DosFileAttributes for f.eks. Windows. I flg. eksempel (der Windows 10 brukes) opprettes en fil med standardrettigheter. Deretter skrives det ut informasjon om noen av de egenskapene som filen har:

```
Path vei = Paths.get("C:/NetBeans/AlgDat/fil.txt");
if (!Files.exists(vei)) Files.createFile(vei);

DosFileAttributes attrs = Files.readAttributes(vei, DosFileAttributes.class);

long tid = attrs.creationTime().toMillis();  // når ble den opprettet
boolean lesing = attrs.isReadOnly();         // kun lesing?
boolean skjult = attrs.isHidden();           // er den skjult?

String format = "Opprettet: %1$tA %1$te. %1$tB %1$tY kl. %1$tH:%1$tM:%1$tS%n";
System.out.printf(format, tid);
System.out.printf("%b %b%n", lesing, skjult);

// Opprettet: fredag 3. november 2017 kl. 11:58:35
// false false
```

Programkode A.6.2 a)

I koden over brukes metoden *createFile*() med kun *vei* som argument. Det er dessverre ikke mulig å sette bestemte filrettigheter ved hjelp av DosFileAttributes som argument i *createFile*(). Men det går med PosixFileAttributes. I flg. eksempel settes det rwx for *eier*, r-x for *grupper* og --- for *andre*. Vi antar at *vei* er en Path:

```
Set<PosixFilePermission> p = PosixFilePermissions.fromString("rwxr-x---");
FileAttribute<Set<PosixFilePermission>> attr
                        = PosixFilePermissions.asFileAttribute(p);
Files.createFile(vei, attr);
```

Programkode A.6.2 b)

Hvis et program inneholder kode som den over, kjøres under Windows, vil det komme en feilmelding om at "posix:permissions is not supported". Vårt alternativ er å starte som i *Programkode* A.6.2 *a)* og så sette attributtverdier direkte vha. metoden *setAttribute*() i Files. Flg. to setninger gjør at filen blir skrivebeskyttet (kun lesbar) og skjult (hidden):

```
Files.setAttribute(vei, "dos:readonly", true);
Files.setAttribute(vei, "dos:hidden", true);
```

Programkode A.6.2 c)

Det er totalt fire «koder» som kan brukes. I tillegg til *readonly* og *hidden* er det *archive* og *system* (se DosFileAttributeView). Det er også mulig å gå motsatt vei. Hvis filen f.eks. er skrivebeskyttet (readonly), kan beskyttelsen fjernes ved å bruke `false`. Dvs. slik:

```
Files.setAttribute(vei, "dos:readonly", false);
```

Det er også mulig å sette attributtverdier ved hjelp av metoder der metodenavnet forteller hvilket attributt som settes. Metodene ligger i DosFileAttributeView:

```
DosFileAttributeView view =
    Files.getFileAttributeView(vei, DosFileAttributeView.class);
view.setReadOnly(true);
view.setHidden(true);
```

Programkode A.6.2 d)

File har metoder som genererer et mappeinnhold. Se f.eks. *Programkode A.6.1 i)*. Ved hjelp av dem kan en (f.eks. ved rekursjon) få tak i filtreet med en oppgitt mappe som rot. I klassen Files som er nyere (Java 1.7), er det egne *walk*-metoder for dette. Her er to av dem:

```
public static Stream<Path> walk(Path start, FileVisitOptions... options)
public static Path walkFileTree(Path start, FileVisitor<? super Path> visitor)
```

Programkode A.6.2 e)

I flg. eksempel skrives innholdet (i preorden) av filtreet med `C:/NetBeans/AlgDat` som rot:

```
Stream<Path> stream = Files.walk(Paths.get("C:/NetBeansAlgDat/AlgDat"));
stream.forEach(p -> System.out.println(p));
stream.close();
```

Programkode A.6.2 f)

Metoden *walkFileTree*() er mer fleksibel, men krever forarbeid. Den har en FileVisitor som argument. Det er et generisk grensesnitt som har flg. fire abstrakte metoder:

```
FileVisitResult visitFile(T file, BasicFileAttributes attr)
FileVisitResult postVisitDirectory(T dir, IOException e)
FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attr)
FileVisitResult visitFileFailed(T file, IOException e)
```

Programkode A.6.2 g)

Filstrukturen kan ses på som et tre med filnoder eller mappenoder. I *visitFile*() bestemmes hva som skal skje når vi kommer til en filnode, *postVisitDirectory*() når vi kommer til en mappenode etter at vi er ferdig med alle dens etterkommere og *preVisitDirectory*() første gang vi kommer til mappenoden. Metoden *visitFileFailed*() bestemmer hva som skal skje hvis en filnode ikke kan «besøkes».

SimpleFileVisitor er som navnet sier, en enkel implementasjon. Metodene har «dummy» kode, dvs. de gjør egentlig ingenting. Returtypen FileVisitResult er en enum med verdiene `CONTINUE`, `SKIP_SIBLINGS`, `SKIP_SUBTREE` og `TERMINATE`.

Metoden *walk*() traverserer filtreet i *preorden*. Vi kan traversere i *postorden* ved å lage en subklasse av SimpleFileVisitor der *visitFile*() og *postVisitDirectory*() overstyres til å skrive ut veien. Klassen `PostVisitor` er her lokal i main-programmet:

```
public static void main(String... args) throws IOException
{
  class PostVisitor<Path> extends SimpleFileVisitor<Path>
  {
    @Override public FileVisitResult
    visitFile(Path file, BasicFileAttributes attr) throws IOException
    {
      System.out.println(file);
      return FileVisitResult.CONTINUE;
    }

    @Override public FileVisitResult
    postVisitDirectory(Path dir, IOException e) throws IOException
    {
      System.out.println(dir);
      return FileVisitResult.CONTINUE;
    }
  } // PostVisitor

  Path start = Paths.get("C:/NetBeansAlgDat/AlgDat");
  Files.walkFileTree(start, new PostVisitor<>());
}
```

Programkode* A.6.2 *h)

Det hadde nok vært enklere å lage en postordentraversering ved hjelp av *listFiles*() i klasen File. Se *Oppgave 2*. Men fordelen med *walkFileTree*() er at den er ferdiglaget. Hva som skal skje under traverseringen, kan kodes inn i *visit*-metodene.

Files har også metoder for å opprette InputStreams, OutputStreams, Readers og Writers ved å bruke en Path som argument. F.eks. slik:

```
String utskrift = "Dette er en utskrift!";
Path vei = Paths.get("C:/NetBeansAlgDat/AlgDat/fil.txt");

BufferedWriter ut = Files.newBufferedWriter(vei);
ut.append(utskrift);
ut.close();

BufferedReader inn = Files.newBufferedReader(vei);
System.out.println(inn.readLine());  // Dette er en utskrift!
inn.close();
```

Programkode* A.6.2 *i)

Files har ikke metoder som behandler enkeltverdier, men har såkalte bulk-metoder. Det betyr metoder som behandler en hel samling verdier i én operasjon. Det kan f.eks. være at hele innholdet i en fil i én operasjon legges inn i en datastruktur eller det motsatte, dvs. at hele innholdet av en datastruktur skrives i én operasjon til fil.

I flg. eksempel legges innholdet av en tekstfil inn i en liste (en ArrayList). Her kan det bli problemer hvis filen er laget med noe annet enn UTF-8 (f.eks. ISO-8859-1). I såfall kan en bruke den versjonen av *readAllLines*() der tegnsettet går inn som argument:

```
Path vei = Paths.get("C:/NetBeansAlgDat/AlgDat/fil.txt");
List<String> liste = Files.readAllLines(vei);
```

Programkode* A.6.2 *j)

`Files` har tre *`write`*-metoder. En av dem skriver innholdet av en *`byte`*-tabell til en fil oppgitt som en `Path`. De to andre skriver ut innholdet av en datastruktur som er itererbar (dvs. implementerer `Iterable`) der verdiene i datastrukturen er tegnstrenger eller mer generelt en subtype til `CharSequence`.

I flg. eksempel blir innholdet av en fil lest inn i en liste (linje for linje) og så blir listens innhold skrevet ut til en annen fil. Dette går bra siden en liste (her en `ArrayList`) er itererbar. Dette blir en filkopiering. Klassen har egne metoder for kopiering av filer. Se *Oppgave* 1.

```
Path vei = Paths.get("C:/NetBeansAlgDat/AlgDat/fil.txt");
List<String> liste = Files.readAllLines(vei);

Path kopivei = Paths.get("C:/NetBeansAlgDat/AlgDat/kopifil.txt");
Files.write(kopivei, liste);
```

Programkode* A.6.2 *k)

I koden over har *`write`*-metoden kun to argumenter. Metoden har formelt et tredje argument, dvs. et av typen `OpenOptions...`. Det betyr ingen, en eller mange verdier. Hvis vi ikke oppgir noe, får vi standardverdiene, dvs. `CREATE`, `TRUNCATE_EXISTING` og `WRITE`. Det betyr spesielt at hvis filen ikke finnes fra før, blir den trunkert. `OpenOptions` er et grensesnitt uten innhold, men har `StandardOpenOptions` som subklasse. Der ligger konstantene som vi kan oppgi selv:

```
Files.write(kopivei, liste, StandardOpenOption.CREATE,
      StandardOpenOption.TRUNCATE_EXISTING, StandardOpenOption.WRITE);
```

Programkode* A.6.2 *l)

`Files` har (per Java 1.9) som nevnt over, hele 65 metoder. Her har vi kun sett på en håndfull av dem. Ved å lese API-ene og eksperimentere, finner en fort ut hvordan de virker og hva de kan brukes til.

Oppgaver til A.6.2

1. Kopier en fil ved å bruke en av *copy*-metodene i `Files`.
2. Lag kode som ved hjelp av *`listFiles`*() i `File` skriver ut filtreet i postorden. Gjør så det samme, men bruk *`list`*() i `Files`.