Læringsmål for forelesningen

- Objektorientering
  - lesing og skriving av data
- Java-programmering
  - IO – klassene
- VS Code
  - ....

NTNU

Dagens forelesning

- Motivasjon: hvorfor IO?
  - bruke data utenfor applikasjonen
  - lagre data på tvers av kjøringer/sesjoner
- Om IO
  - rør-metafor
  - IO av bytes vs. tekst
  - adressering av ressurser
- Eksempel: IO for person-objekter

Mål

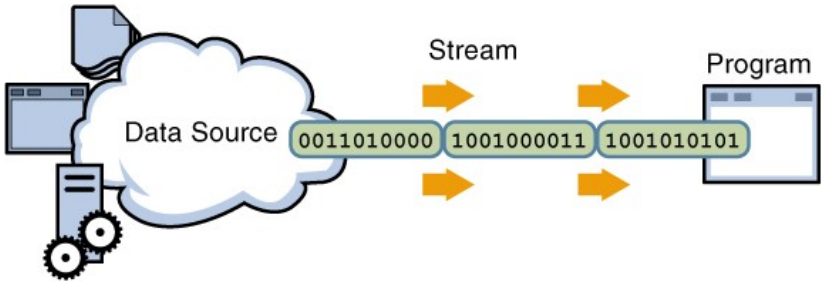
- Dere skal ha gått igjennom en del ulike måter å manipulere filer
- Dere skal forstå lagene
- Dere skal vite om absolutt og relativ sti
- Dere skal kunne velge en passende måte å gjøre det på for prosjektet!

Java-pakken: java.io

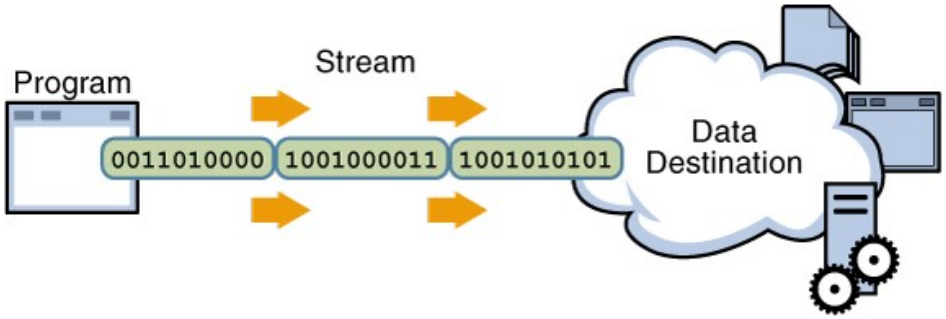
- Java har klasser som vi kan bruke for å:
  - representere/håndtere filer og kataloger
  - lese/skrive fra/til datastrømmer, f.eks. filer og nettressurser
- OBS! Mange klasser å holde styr på – vanskelig å huske selv for oss som har brukt Java lenge
- Basisklassene gjør enkel IO, så bygger vi ut med klasser som gjør mer
- Et godt eksempel på hvor nyttig det er å kunne ty til Java-dokumentasjonen og eksempelkode

Streams *(io, ikke lambda)*

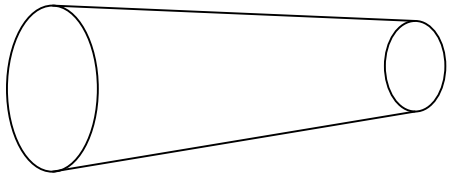
- En ”stream” (strøm) er en kommunikasjonslinje for data
- Input stream: data flyter inn i programmet



- Output stream: data flyter ut av programmet



Mental modell: et rør

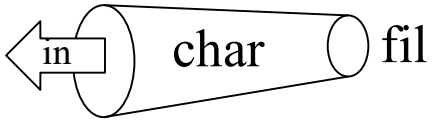


- Tenk på en stream som et rør
  - input: programmet suger data ut av røret
  - output: programmet dytter data inn i røret
- Et rør har noen viktige egenskaper
  - *hva slags* type **grunndata** røret transporterer, f.eks. byte eller char
  - *hvordan* data behandles og ”klumpes” sammen, f.eks. linjer eller objekter
  - *hvor* den andre enden av røret er (begynner eller slutter), f.eks. fil eller nettressurs
- Det er mange valg å ta, som styrer hvilke egenskaper røret bør ha
- Dersom en allerede har et rør, så kan et ”adapter” (rørstuss) brukes for å endre egenskapene

Rør-egenskaper

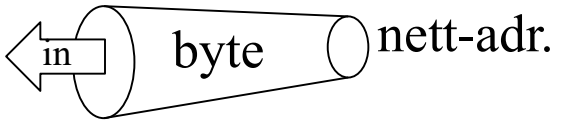
- *hva slags* type grunndata røret transporterer
  - byte-orientert: InputStream/OutputStream-klassene
  - char-orientert: Reader/Writer-klassene
  - Object-orientert: ObjectStream-klassene
- *hvordan* data behandles og ”klumpes” sammen
  - et stk. data om gangen, f.eks. én byte eller én char
  - buffer-håndtering: del av tabell med offset og lengde
  - satt sammen til String, basert på leksikalske enheter og skilletegn
- *hvor* den andre enden av røret er (begynner eller slutter)
  - fil angitt med navn, evt. basert på File-klassen
  - nettressurs (lokal eller fjern) angitt med URL
  - lokalt buffer, f.eks. tabell, StringBuffer e.l.

Mange ulike kombinasjoner



- char-orientert input fra fil:
  - Reader reader = new FileReader(new File(”C:/temp/test.txt”));
    char[] buffer = new char[1000];
    int charCount = reader.read(buffer);

BrukFileReader



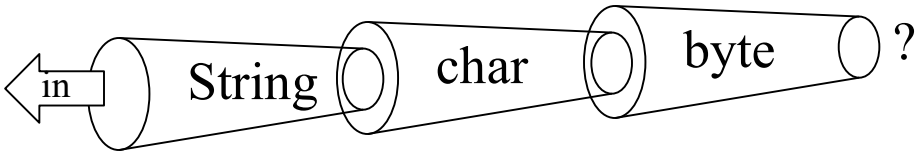
- byte-orientert input fra nettressurs:
  - InputStream input = new URL(urlString).openStream();
    int byteVerdi = input.read();

MyIO.readInputStream

- String-orientert output til fil:
  - PrintWriter writer = new PrintWriter (”C:/temp/test.txt”);
    writer.println(”Hei hopp hvor det går, hilsen ” + person.getName());

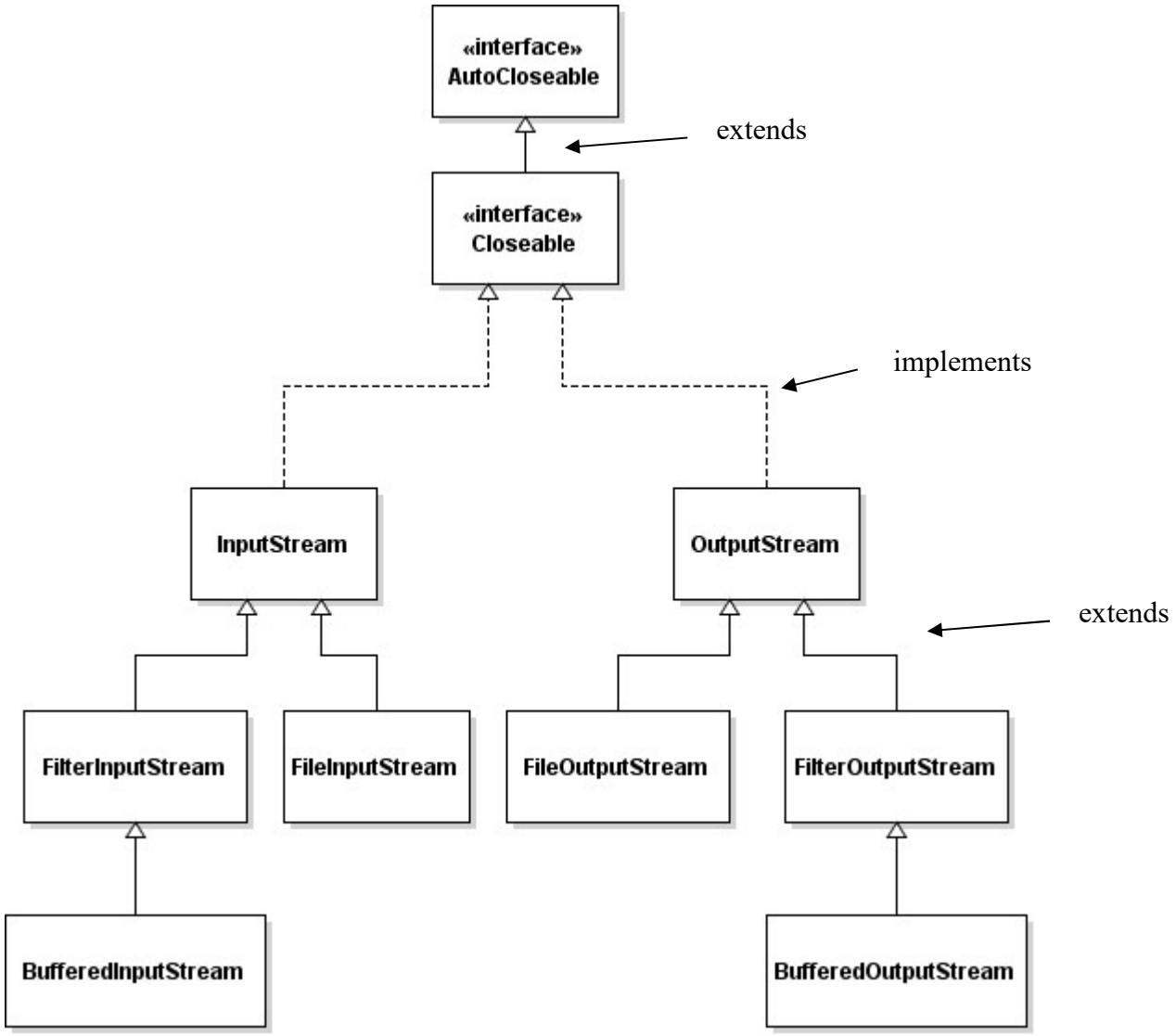
MyIO.printWriter

Mange ulike kombinasjoner



- String-orientert input fra eksisterende InputStream: MyIO.BufferedISR
  - BufferedReader reader = new BufferedReader(
    new InputStreamReader(inputStream));
    String line = reader.readLine();

- Input/Output fra/til String: MyIO.main
  - Reader reader = new StringReader(s); // innholdet leses fra en String
  - PrintWriter writer = new StringWriter();
    writer.print(...); MyIO.main
    String s = writer.toString(); // alt som er print'et samlet i en String



https://www.codejava.net/java-se/file-io/how-to-read-and-write-binary-files-in-java

java.io.{InputStream og OutputStream}

- Klasser som definerer grunnleggende sett med metoder for lesing/skriving av data
  - input/output stream av bytes
  - dekker mange typer byte-streams, også over nettet

MyIO.ReadInputStream

- Mange typer streams ”pakker inn” en enklere stream-type og tilbyr ekstrametoder
- FileInputStream / FileOutputStream
  - lesing/skriving av binære data fra/til fil
- PrintStream
  - enklere skriving av objekter, basert på toString-metoden

BrukPrintStream

- ByteArrayInput/OutputStream
  - lesing/skriving av binære data fra byte array i minnet
  - velegnet for intern testing

byte og char i Java

- datatypen **`byte`** er en 8-bits verdi
  - Benyttes for mange typer "binære" data f.eks. representere punkter i et bilde, digitale lydsignaler osv.
  - Filsystemet skiller ikke mellom binære filer og tekstfiler – alt er binære data og på laveste nivå skriver og leser vi bytes.
- **`char`** er en 16-bits verdi for teksttegn
  - Java benytter UNICODE-tegnsettet for å representere tekst-data i programmer
  - Internt representeres char og String vha. "unsigned" 16-bit integer
  - **Charset**-klassen håndterer oversetting fra byte-sekvenser til char-sekvsenser
- Tekstfiler må håndteres av riktig **Charset**
  - UTF-8: koding av UNICODE-tegn med byte (8 bit)-enheter
  - UTF-16: koding av UNICODE-tegn med short (16-bit)-enheter
  - ISO 8859-1 brukes av Windows for maskiner som er konfigurert for norsk
  - tegnkodingen (Charset) kan bare til en viss grad auto-detekteres...

java.io.{Reader og Writer}

- Klasser for hhv.
  - Lesing fra eller skriving til streams av *tekst-tegn* (character streams)
  - dekoder altså bytes til riktige char-verdier
  - kan oppgi ønsket Charset (UTF-8 er default)

- FileReader / FileWriter
  - lesing/skriving av tekstfiler

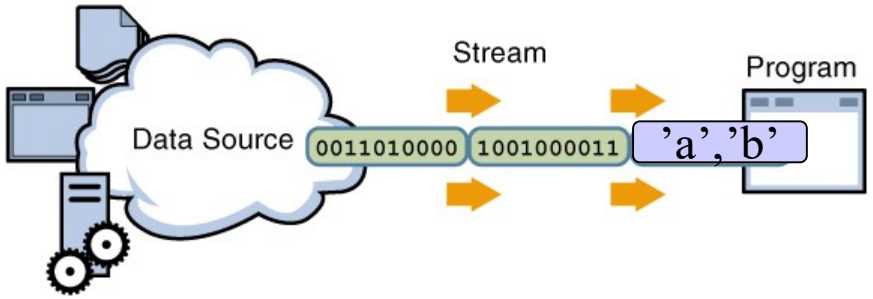
MyIO.file_Reader

java.io.InputStreamReader

- **`InputStreamReader`**
  er en "bro" mellom byte-streams og tegn-streams

Eksempel: ISR

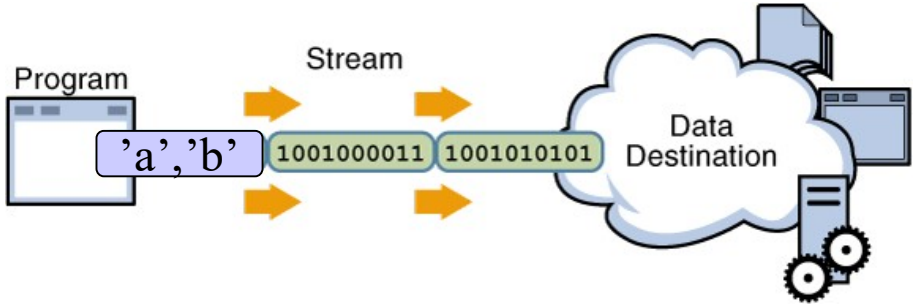
- Leser bytes fra en underliggende byte-stream, f.eks. fra fil, og konverterer dem til tegn



- Kan oppgi ønsket Charset (UTF-8 er default)

java.io.OutputStreamWriter

- **`OutputStreamWriter`**
  er en "bro" mellom tegn-stream og byte-stream
- Skriver tegn til en underliggende byte-stream, f.eks. til fil, og konverterer dem til bytes.



- Kan oppgi ønsket Charset (UTF-8 er default)

BufferedReader / BufferedWriter

- Problem:
  - Upraktisk med lesing av enkelttegn: For lesing av tekst ønsker vi ofte å lese en og en linje av gangen (som kan ha varierende lengde)
  - Ineffektivt: Lesing/skriving av en og en byte fra ekstern ressurs kan være svært lite effektivt
- BufferedReader frigjør oss fra ”lavnivå” lesing av fil ved at den benytter et mellomlager (buffer)
  - leser inn større blokker av gangen og lagrer midlertidig
  - kan lese inn data på forskudd
  - her kan en bruke readLine()

Unntak ved IO

- Mange av metodene som kalles på IO-objekter utløser ***unntak***
  - subtyper av **`java.io.IOException`**
  - f.eks **`java.io.FileNotFoundException`**
- Nødvendig å bruke **try/catch** og **finally**
- Når vi er ferdige med å lese eller skrive må vi si fra at vi er ferdige
  - for å frigjøre ressursene brukes close()
  - dersom filer ikke lukkes, kan det hindre andre filoperasjoner
- Finnes en egen, sikker try/catch-variant

IO-unntak og lukking

- Har egen **try/catch**-variant for **Closeable**-objekter, f.eks. InputStream
- try med **Closeable**-argument sikrer lukking:
  - // input stream blir automatisk lukket
    try (InputStream input = …) {
      …
    }

java.nio.file.{Path og Files}

- abstrakt representasjon av fil- eller katalog*navn*
  - Stier har ulik syntaks på UNIX og Windows, men vi har samme funksjonelle krav til håndtering av filer

```
Path minfil = Paths.get("mappe", "fil.txt");
```

  - Har omtrent samme funksjon som java.io.File, se http://docs.oracle.com/javase/tutorial/essential/io/legacy.html
- mange metoder for å manipulere Path-objekter, uten å måtte analysere String-er
- statiske Files-metoder bruker Path-objekter
  - lage InputStream/OutputStream- og Reader/Writer-objekter
  - utføre filsystem-operasjoner

Eksempel: Flerkamp

java.net.URL

- Abstrakt representasjon av nettressurs
  - standardisert syntaks: <protokoll><adresse><sti>
  - http://www.idi.ntnu.no - IDI sin hjemmeside
  - file:///C:/temp/test.txt - testfil på lokal harddisk
  - ftp ...
- Fra en URL kan en få en InputStream
  - **url.openStream();** evt.
  - **URLConnection con = url.openConnection();**
    con.set...(...);
    InputStream input = con.getInputStream();
- Fra en URL kan en også få en OutputStream
  - **URLConnection con = url.openConnection();**
    con.set...(...);
    OutputStream output = con.getOutputStream();
  - krever litt detaljkunnskap om protokollen, f.eks. HTTP

Fillagring av objekter

- Vi har noe som likner på Pythons ‘pickle’ i Java, også…

Eksempel: Dyrehage

Bruk av Class.getResource

- Problemet med adressering med Path/File
  - adressering med absolutte referanser gjør koden spesifikk for oppsettet på maskina en kjører på
  - det er uklart hva en relativ referanse er relativ til
    - current directory, f.eks. der programmet ligger
    - hjemmekatalog
    - pakken klassen ligger i
- *Husk at programmer bør kunne kjøres fra hvor som helst: desktop, web, mobil*
- Class.getResource og Class.getResourceAsStream brukes for å referere absolutt eller relativt ift. en klasse og prosjektets pakkestruktur
  - **object.getClass().getResource(<name>)** returnerer en **URL** til **<name>**, som tolkes relativt til koden til klassen til **object**, eller absolutt ift. kildekodemappene (f.eks. **src**, **tests** og **resources**)
  - **object.getClass().getResourceAsStream(<name>)** returnerer en **InputStream**, hvor **<name>** tolkes på samme måte som for **getResource**.

Scanner-klassen

- Brukes til å stykke opp tegn fra en **InputStream** i deler kalt *tokens*
- Du styrer selv hva som utgjør et token, f.eks. basert på et skilletegn
  - **scanner.useDelimiter(",");**
- Gjør linje-basert innlesing enda enklere
  - **scanner.useDelimiter("\n");**
- Skjuler unntakshåndteringen i den underliggende strømmen
  - **IOException iox = scanner.ioException();**
- Implementerer **Iterator<String>**, slik at en kan bruke kjente metoder som **hasNext()** og **next()**.

Regulære uttrykk

- *Minispråk* som brukes for å beskrive og analysere strukturen til tekst, ala filmønstre som *.pdf
- Brukes mye ifm. oppstykking av tekst i *tokens*, såkalt leksikalsk analyse
- **Pattern**-klassen brukes for å representere regulære uttrykk
- **Scanner**-klassen kan sjekke om neste input-del (token) matcher et gitt **Pattern**
  - scanner.hasNext(pattern) gir true dersom neste token (hel linje dersom delimiter er satt til \n) matcher <pattern>
  - scanner.next(pattern) returnerer og hopper over <pattern>

Standard-formater

- XML
  - format basert på hierarkiske “elementer”, med attributter:

```
<element1 attr1=“value1”>
  <element2 attr2=“value2”>litt tekst</element2>
  <element3 attr3=“value3”/>
</element1>
```

- JSON
  - format basert nøstede blokker med attributt-verdi-par

```
{
  “attr1”: “value1”,
  “element2”: { “attr2”: “value2”, “text”: “litt tekst” },
  “element3”: { “attr3”: “value3” }
}
```

- egne API-er for å lese/skrive

Organisering av IO-kode

- La alle metoder som skriver/leser ta inn en **InputStream/OutputStream** som argument
  - gjør koden mindre avhengig av spesifikk datakilde
  - gjør det lettere å teste med en test-strøm basert på file eller String
- IO av datastrukturer, to varianter
  - read- og write-metoder, med argumenter
    - objektet som skal fylles inn (read) eller skrives ut (write)
    - strømmen som skal lese fra eller skrives til
  - lag en egen klasse for innholdet, som har read- og write-metoder
    - konstruktøren tar inn originalobjektene
    - read/write-metodene tar inn strømmen som argument
    - fyller inn innholdsobjektet (read) eller skriver det ut (write)

Læringsmål for forelesningen

- Objektorientering
  - lesing og skriving av data
- Java-programmering
  - IO – klassene
- …og jobbing i VS Code