Algoritmer og datastrukturer
Vedlegg B – Formatert utskrift

B Formatert utskrift

B.1 Hva er formatering?

Vi formaterer for å bedre lesbarheten. Første valg er vanligvis å bestemme hvor stor plass som skal brukes til å skrive ut en verdi, dvs. bredden til *utskriftsfeltet*. Hvis det er større enn nødvendig, avgjør vi så om utskriften skal ligge lengst til høyre i feltet (*høyrejustert*) eller lengst til venstre (*venstrejustert*). For å få til dette og andre typer effekter, bruker vi et *formateringsdirektiv*:

```
(B.1.1)  %[argumentnummer$][flagg][feltbredde][.presisjon]konverteringssymbol
```

Her er noen eksempler på mulige formateringsdirektiver:

```
(B.1.2)  %5d    %-7.2f    %2$07X    %-5.10s    %n
```

I B.1.1 markerer hakeparenteser (dvs. [og]) det som kan (men ikke må) være med (eng: optional). Et direktiv må alltid starte med % (prosent) og ende med et *konverteringssymbol*. Det siste eksemplet i B.1.2 viser et direktiv med kun de to delene. Direktivet %n står forøvrig for linjeskifte. De andre eksemplene i B.1.2 har et konverteringssymbol (d, f, X og s) til slutt, men har også en eller flere av de andre elementene fra B.1.1.

Klassene `PrintStream` og `PrintWriter` har begge en utskriftsmetode med følgende signatur:

```
(B.1.3)  public PrintStream printf(String format, Object... args)
```

Formateringssetningen *format* (en `String`) kan inneholde flere direktiver og argumentlisten *args* skal normalt ha like mange argumenter som direktiver. Vi skal bruke metoden *printf*() til å forklare kort hva hensikten er med de ulike delene i det generelle formateringsdirektivet i B.1.1. For enkelhets skyld velger vi utskrift til *konsollet*, dvs. ved bruk av `System.out`. Den er en instans av `PrintStream` og har bl.a. metodene *print*(), *println*() og *printf*().

Ta som eksempel at vi skal skrive ut informasjon om en gruppe studenter. Informasjonen består av navn, poeng (antall studiepoeng) og alder. Her er den gitt i form av tre tabeller. Vi tar som gitt at de er like lange (her lengde fire):

```
String[] navn = {"Petter", "Kari", "Aleksander", "Elin"};
int[] poeng = {0, 140, 90, 70};
int[] alder = {19, 21, 20, 19};
```

Tabell **B.1.1**

Et første forsøk er å skrive ut på enklest mulig måte, dvs. ved hjelp av *println*(). Da skjøter vi sammen informasjonen (navn, poeng og alder) til en tegnstreng der det ligger en blank (et mellomrom) mellom verdiene:

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

Programkode **B.1** ***a)***

Denne kodebiten vil gi oss flg. utskrift:

```
Petter 0 19
Kari 140 21
Aleksander 90 20
Elin 70 19
```

Dette er grei nok utskrift for såpass få linjer og såpass få verdier per linje. Men hvis det var mange linjer (mange studenter) og flere verdier enn dette som har variabelt antall tegn, så vil utskriften kunne bli uoversiktlig. Da er det bedre med formatert utskrift, dvs. at verdiene kommer i faste kolonner. F.eks. slik:

```
Petter        0  19
Kari        140  21
Aleksander   90  20
Elin         70  19
```

Her kan vi bruke felt med lengder på hhv. 12, 3 og 4 for navn, poeng og alder. Symbolene for konvertering er s for tegnstrenger og d for heltall på desimalform. Navn leses best når de er venstrejustert (- som flagg) og tall høyrejustert (det er standard). Dermed får vi det vi ønsker ved hjelp av flg. kode:

```
for (int i = 0; i < navn.length; i++)
{
  System.out.printf("%-12s%3d%4d\n", navn[i], poeng[i], alder[i]);
}
```

Programkode* B.1 *b)

I *printf*-setningen inngår de tre formateringsdirektivene %-12s, %3d og %4d. De tre er satt opp fortløpende, dvs. uten mellomrom. Det er tillatt å ha mellomrom, men det vil påvirke utskriften siden de også kommer med der. Hvis vi ønsker nøyaktig samme utskrift som over, men med mellomrom i formateringssetningen, kan vi f.eks. lage den slik:

```
(B.1.4)  "%-10s  %3d %3d\n"
```

Her er bredden for navn satt til 10 og den for alder til 3. Dvs. at først reserveres 10 plasser til tegnstrengen (venstrejustert). Så kommer to blanke (mellomrom) i utskriften, så et felt med bredde 3 til poeng, så en blank og til slutt et felt med bredde 3 til alder. Men det er også mulig å ha tekst mellom direktivene. F.eks. slik som dette:

```
for (int i = 0; i < navn.length; i++)
{
  System.out.printf
    ("Navn: %-12sPoeng: %3d Alder: %2d\n", navn[i], poeng[i], alder[i]);
}
```

Programkode* B.1 *c)

Dette gir flg. utskrift:

```
Navn: Petter      Poeng:   0 Alder: 19
Navn: Kari        Poeng: 140 Alder: 21
Navn: Aleksander  Poeng:  90 Alder: 20
Navn: Elin        Poeng:  70 Alder: 19
```

Obs: \n er ikke et direktiv, men tegnet for linjeskifte. Det inngår i utskriften og fører til at vi får «ny linje». Vi kan bytte ut \n med %n. Dermed fire direktiver, men kun tre argumenter.

Normalt skal det være like mange av hver type. Men %n holdes utenfor regnskapet. Hvis det er færre direktiver enn argumenter, blir de overflødige (bakerste) argumentene oversett.

Hvis det derimot er flere direktiver enn argumenter, kastes det normalt et unntak. Men ikke hvis det signaliseres (ved hjelp av et argumentnummer - se B.1.1) at flere direktiver skal kobles til samme argument (som dermed bli skrevet ut flere ganger). Normalt hører første direktiv til første argument, osv. Det er mulig å stokke om på dette. Det generelle direktivet (se B.1.1) tillater at en spesifiserer hvilket direktiv som hører til hvilket argument. Men dette er en mer uvanlig teknikk. Se også *Oppgave* 6 og 7.

Det å angi feltbredde for et argument kan i noen tilfeller være problematisk. I eksemplet med navn, poeng og alder er poeng og alder uproblematiske. En poengsum kan ikke ha flere enn tre siffer og en alder kan ha maks to siffer. Men et navn kan variere mye i lengde. Et fornavn kan være fra to bokstaver og oppover. I eksemplet over er det Aleksander med ti bokstaver som er lengst. Men det er fornavn som er lengre enn det og hvis vi tillater doble eller triple fornavn, kan det bli svært langt. Hvis vi også skal ha med etternavn, blir det enda verre. Velger man så stor feltbredde at alle tenklige navn får plass, vil det ofte ikke bli plass til resten av utskriften hvis den f.eks. skal på papir. Velger en for liten feltbredde, kan det være navn som ikke får plass. I praksis velger en ofte en rimelig stor feltbredde og trunkerer (fjerner det som ikke får plass). Med andre ord vil ekstra lange navn ikke bli skrevet fullt ut.

I en *printf*-setning er det annereledes. Hvis en verdi trenger mer plass enn den reserverte feltbredden, vil bredden automatisk bli utvidet. Men da vil de verdiene som kommer deretter på samme linje bli forskjøvet tilsvarende. Vi ser fortsatt på studenter med navn, poeng og alder. Se *Tabell* B.1.1. La nå Elin isteden hete Elin Aleksandra med 170 poeng. Det gir 13 tegn for navn og tre siffer for poeng. Da vil *Programkode* B.1 *b)* gi flg. utskrift:

```
Petter        0  19
Kari        140  21
Aleksander   90  20
Elin Aleksandra170  19
```

Det er flere effekter man kan få til enn de som er diskutert til nå. En del av dem skal vi se på i de neste avsnittene.

Oppgaver til Avsnitt B.1

1. Sjekk at formateringssetningen B.1.4 i *Programkode* B.1 *b)*, gir samme utskrift som før.
2. Bytt ut \n med %n i *Programkode* B.1 *b)* og sjekk at det gir samme utskrift.
3. Sjekk at *Programkode* B.1 *c)* gir den utskriften som er satt opp.
4. I *Programkode* B.1 *b)* er %-12s formateringsdirektiv for tegnstrengen. Symbolet - (dvs. minus) er et flagg (se B.1.1) og fører til venstrejustert utskrift. Ta vekk minustegnet og se hva slags utskrift du får. Hva blir det hvis du bruker minustegnet i alle direktivene.
5. Gjør om *Programkode* B.1 *b)* slik at alle navn med flere enn 10 tegn blir trunkert.
6. Ta utgangspunkt i *Programkode* B.1 *b)*. Normalt skrives argumentene ut i den rekkefølgen de står. Det kan imidlertid overstyres ved hjelp av argumentnummer på direktivene. Se B.1.1. Bevar rekkefølgen av direktivene, men bruk argumentnummer slik at rekkefølgen på utskriften blir navn, alder og poeng.
7. Både d, o og x/X er konverteringssymboler for heltall (desimal, oktal eller heksadesimal form). Lag en *printf*-setning som skriver ut et og samme heltall på alle tre formene.

B.2 Klassen Formatter

I Avsnitt B.1 ble metoden *printf*() brukt til å demonstrere hvordan man kan lage utskrifter ved hjelp av formateringsdirektiver. Det var naturlig å bruke metoden *printf*() siden de fleste allerede er familiære med *print*() og *println*(). Dessuten er det mange som allerede kjenner *printf*() fra programmeringsspråket C. I Java er dette generalisert ytterligere. F.eks. kan vi konstrurere en formatert tegnstreng ved hjelp av metoden *format*() i klassen String. Dermed vil flg. kode gi samme utskrift som *Programkode* B.1 *b)*:

```
for (int i = 0; i < navn.length; i++)
{
  System.out.println(String.format("%-12s%3d%4d", navn[i], poeng[i], alder[i]));
}
```

Programkode B.2 *a)*

Dette er generalisert i klassen Formatter. Et formateringsobjekt, dvs. en instans av klassen, kan knyttes til en enhet som kan «ta imot» utskrift. Klassen har hele fjorten konstruktører:

```
public Formatter()
public Formatter(File file)
public Formatter(File file, String csn)
public Formatter(File file, String csn, Locale l)
public Formatter(OutputStream os)
public Formatter(OutputStream os, String csn)
public Formatter(OutputStream os, String csn, Locale l)
public Formatter(PrintStream ps)
public Formatter(Appendable a)
public Formatter(Appendable a, Locale l)
public Formatter(String fileName)
public Formatter(String fileName, String csn)
public Formatter(String fileName, String csn, Locale l)
public Formatter(Locale l)
```

Programkode B.2 *b)*

Formatering er avhengig både av hvilket tegnsett en bruker og av lokale forhold (i et land eller i en språkkultur). Et eksempel er en datoangivelse. Der sier Norsk standard at den skal være på formen: *dag måned år*. Men i andre land kan det være annerledes. F.eks. er det *måned dag år* i USA. Et annet eksempel er desimaltall. Der heter det *desimalkomma* på norsk og tallet π med to desimalers nøyaktighet oppgis derfor som 3,14. Det samme gjelder på tysk og fransk. Men på engelsk er det 3.15 og der heter det *desimalpunktum*. Kjør koden nedenfor på ditt system. Hvilken utskrift får du?

```
System.out.printf("%5.2f%n", Math.PI);  // Får du 3,14 eller 3.14 her?
System.out.println(Math.PI);            // Hva får du her?
```

Programkode B.2 *c)*

Åtte av konstruktørene i Formatter (se *Programkode* B.2 *b*) spør etter tegnsett (String *csn*) eller etter lokale regler (Locale *l*). På norsk har det blitt mest vanlig med UTF-8. Men også iso-8859-1 er i bruk, f.eks. i hele kompendiet som dette avsnittet hører til. Klassen Locale har flere konstanter, f.eks. Locale.UK (britisk) og Locale.US (amerikansk). Men ingen for norsk eller andre skandinaviske språk. Isteden kan vi lage en norsk instans av klassen Locale på flg. måte:

```
(B.2.1)  Locale l = Locale.forLanguageTag("NO");
```

Dette kan vi bruke til garantert å få skrevet π som 3,14:

```
Formatter f = new Formatter(System.out, "UTF-8", Locale.forLanguageTag("no"));
f.format("%5.2f%n", Math.PI);  // nå burde det komme 3,14
f.flush();  // må ha flush() eller close() for å få utskrift
```

Programkode* B.2 *d)

Men hva vil skje hvis vi lager en instans av `Formatter` ved hjelp av en konstruktør som hverken ber om tegnsett eller lokale regler? F.eks. slik:

```
(B.2.2)  Formatter f = new Formatter(System.out);  // utskrift til konsollet
```

Denne koden fører til at det du har som standard av tegnsett og lokale regler, vil bli brukt. Hvis du bruker et utviklingsmiljø, vil et prosjekt bli satt opp med utviklingsmiljøets standard tegnsett. Det vil nok som oftest være UTF-8. Men du kan endre tegnsettet for et prosjekt hvis du ønsker det, f.eks. til ISO-8859-1. Flg. kode skriver ut hva du har som tegnsett:

```
(B.2.3)  System.out.println(Charset.defaultCharset().name());
```

Når det gjelder `Locale` brukes det som systemet ditt er satt opp med. Hvis du bruker norsk oppsett på maskinen din, så blir det ditt «locale». Dette kan du få sjekket ved:

```
(B.2.4)  System.out.println(Locale.getDefault().getLanguage());
```

Alle *konstruktørene* i `Formatter`, bortsett fra to, krever en «mottaker» (destination). Det er enheter der vi kan «legge til» (append) verdier, dvs. instanser av klasser som implementerer grensesnittet `Appendable`. Det er mange klasser i Java som er gjør det. Det gjelder f.eks. `PrintStream`, `BufferedWriter`, `PrintWriter` og `StringBuilder`.

Men det er kun i to av dem der det eksplisitt står `Appendable` som parametertype. Men hva med de andre? Ta f.eks. de tre som har et filnavn som parameter. Se på flg. kode:

```
Formatter f = new Formatter("ut.txt");  // utskrift til fil
System.out.println(f.out());            // BufferedWriter
```

Programkode* B.2 *e)

`Formatter` har metoden *`out`*`()` som returnerer en `Appendable`, dvs. «utskriftsmottakeren». I dette tilfellet er det en `BufferedWriter` knyttet til filen `"ut.txt"`. Men hva blir det for den parameterløse konstruktøren? Vi får svaret hvis vi sjekker kildekoden. Den er kodet slik:

```
public Formatter()
{
  this(Locale.getDefault(Locale.Category.FORMAT), new StringBuilder());
}
```

Programkode* B.2 *f)

Her står det eksplisitt at en `StringBuilder` er «mottaker». Men hva får vi som utskrift hvis vi lager samme type kode som i *Programkode* B.2 *e)*?

```
Formatter f = new Formatter();  // utskrift til en StringBuilder
System.out.println(f.out());    // her kommer det ikke noe
```

Programkode* B.2 *g)

I denne koden kommer det ingen utskrift fordi metoden *`toString`*`()` for en `StringBuilder` returnerer innholdet, dvs. det som lagt inn. Men forløpig er den tom og dermed ingen utskrift.

I *Programkode* B.2 *e)* kalles også metoden *toString*(), men der er det basisklassens (klassen Object) versjon som brukes og den sender ut navnet på klassen og informasjon om dens minneadresse som f.eks. dette: java.io.BufferedWriter@4aa298b7.

Metoden *format*() i Formatter gir formatert utskrift slik som f.eks. i *Programkode* B.1 *b)* og *Programkode* B.2 *a)*. Her er det informasjonen i tabellene i Tabell B.1.1 som brukes:

```
Formatter f = new Formatter(System.out);  // utskrift til konsollet

for (int i = 0; i < navn.length; i++)
{
  f.format("%-12s%3d%4d%n", navn[i], poeng[i], alder[i]);
}

f.close();
```

Programkode* B.2 *h)

Oppgaver til Avsnitt B.2

1. Hvilken utskrift får du i *Programkode* B.2 *c)*? Får du 3,14 i *Programkode* B.2 *d)*?
2. Du finner tegnsett og språk på ditt system ved hjelp av B.2.3 og B.2.4. Prøv det!
3. Opprett en Formatter som i *Programkode* B.2 *g)*. Legg inn den formatterte informasjonen om studenter (Tabell B.1.1) og skriv til slutt innholdet til konsollet.
4. I B.2.4 brukes metoden *getLanguage*() til å finne hvilket språk som hører til en instans av klassen Locale. Klassen har flere metoder av denne typen. Prøv også *getCountry*(), *getISO3Country*() og *getISO3Language*(). Den statiske metoden *getISOLanguages*() gir en tabell av typen String[] med alle språk som er definert i ISO 639. Prøv den!
5. Den statiske metoden *getISOLanguages*() gir som nevnt i Oppgave 4, en String-tabell med alle språk definert i ISO 639. Finn ut hvike av dem som formaterer desimaltall med desimalkomma (som på norsk) og ikke desimalpunktum. Bruk f.eks. 3,14 som eksempel. Hint: Konstruer en Locale for hver språktag som *getISOLanguages*() gir og bruk den til å formatere Math.PI til 2 desimaler. Bruker du direktivet %4.2f blir resultatet enten 3,14 eller 3.14. Finn så hvilke språk disse totegnskodene representerer. Se f.eks. denne siden.

B.3 Formatering av heltall

Det generelle *formateringsdirektivet* har som tidligere nevnt, flg. utseende:

```
(B.3.1)  %[argumentnummer$][flagg][feltbredde][.presisjon]konverteringssymbol
```

For hele tall (typene `byte`, `Byte`, `short`, `Short`, `int`, `Integer`, `long`, `Long` og `BigInteger`) gjelder konverteringssymbolene d, o, x og X der d står for desimal utskrift, o for oktal, x og X for heksadesimal utskrift (x gir små bokstaver og X store).

I flg. eksempel brukes konverteringssymbolene d, o og x sammen med et *argumentnummer* til å skrive ett og samme heltall på desimal, oktal og heksadesimal form:

```
int k = 1234;
System.out.printf("%1$5d%1$5o%1$4x%n", k);  // 1234 2322 4d2
```

Programkode* B.3 *a)

I koden over har de tre direktivene 1 som argumentnummer. En utskrift får like mange verdier som direktiver. Men alle tre refererer til det første (og her det eneste) argumentet og dermed skrives det ut tre ganger.

Hvis tallet *k* er negativt, blir det annereledes enn det mange forventer. La f.eks. k = -1. Den er her satt opp som en *int*-verdi og er dermed representert binært med 32 1-ere. Vi får den oktale formen ved å gruppere tre og tre biter fra høyre mot venstre. Hver slik representerer tallet 7. Siden det er 32 biter blir det kun to 1-ere i den siste, dvs. tallet 3. Dermed blir -1 representert oktalt som `37777777777`. Det blir tilsvarende heksadesmimalt, men da med grupper på fire og fire 1-ere. Dermed blir det `ffffffff` eller `FFFFFFFF`. Men på desimal form blir det som vanlig -1.

Hvis vi skal bruke *feltbredde* i et utskriftsdirektiv, bør vi vite hvor stor plass som trengs. Hvis vi skal skrive ut både posistive og negative tall oktalt eller heksadesimalt, må vi ta hensyn til at de negative kan få mange flere siffer siden vi normalt ikke tar med ledende 0-er i posistive tall. Vi kan droppe feltbredden. Da brukes nøyaktig så mye plass som trengs:

```
int k = -1;
System.out.printf("%1$d %1$o %1$x%n", k);  // -1 37777777777 ffffffff
```

Programkode* B.3 *b)

Konverteringssymbolene d, o, x og X gjelder for alle heltallstypene, både de grunnleggende og referansetypene. Hva skjer hvis du lar *k* være en av de andre typene? Se også Oppgave 1.

Et *flagg* brukes til å oppnå ulike effekter. Her er noen av mulighetene:

- – (minustegnet) gir som vanlig venstrejustert utskrift i det reserverte feltet.
- + (plusstegnet) gjør at en utskrift med d som konverteringssymbol får fortegnet + for et ikke-negativt tall. Dette gjelder alle heltallstypene. Hvis en bruker o, x eller X som konverteringssymbol, gjelder dette kun for typen `BigInteger`.
- 0 (siffer 0) gjør at ledige plasser i utskriftsfeltet blir fylt med 0-er (zero padding).
- ((venstre parentes) gjør at negative tall får en parentes rundt seg istedenfor fortegn. Gjelder både d, o, x eller X for `BigInteger`, men kun for d for de andre heltallstypene.
- , (komma) gjør at store heltall blir delt opp i siffergrupper på tre og tre separert med et symbol (det er språkspesifikt, normalt komma, punktum eller mellomrom). Gjelder kun for konverteringssymbolet d.
- # (nummertegnet) gir 0 først i oktal og 0x (eller 0X) først i heksadesimal utskrift.

Vi kan bruke ingen, ett eller flere flagg. I flg. eksempel bruker vi både komma, venstre parentes og plusstegn. Komma gjør at store tall deles opp i siffergrupper. Hva som brukes som skilletegn er språkspesifikt. Venstre parentes gjør at hvis et tall er negativt, kommer det ikke fortegn, men parentes rundt tallet. Plusstegnet setter en + foran ikke-negative tall:

```
int a = 10000000, b = -10000000;  // et positivt og et negativt tall

Locale us = Locale.forLanguageTag("us");  // amerikansk
Locale de = Locale.forLanguageTag("de");  // tysk
Locale no = Locale.forLanguageTag("no");  // norsk

System.out.printf(us, "%,(+15d%,(+15d%n", a, b);  // +10,000,000   (10,000,000)
System.out.printf(de, "%,(+15d%,(+15d%n", a, b);  // +10.000.000   (10.000.000)
System.out.printf(no, "%,(+15d%,(+15d%n", a, b);  // +10 000 000   (10 000 000)
```

Programkode B.3 *c)*

Rekkefølgen av flaggene er likegyldig. I koden over kunne det vært +,(eller en annen rekkefølge. Det er også mulig å bruke s (eller S) som konverteringssymbol for heltall. Egentlig er s for tegnstrenger, men hvis k er et heltall, gir `Integer.toString(k)` utskriften.

Oppgaver til Avsnitt B.3

1. Bruk først `byte`, så `short` og til slutt `long` som datatype for *k* = -1 i *Programkode* B.3 *b)*. Sjekk utskriften! Hva blir det hvis du lar *k* være av typen `BigInteger`, dvs. `BigInteger k = new BigInteger("-1")`?
2. Gitt tabellen `int[][] a = {{5,15,100,10},{200,5,90,0},{50,150,50,100}};` Lag kode slik at den blir skrevet ut på flg. form:

   ```
     5   15  100   10
   200    5   90    0
    50  150   50  100
   ```

   Gjør så om koden slik at det isteden blir:

   ```
   005  015  100  010
   200  005  090  000
   050  150  050  100
   ```

3. Gjør om *Programkode* B.3 *b)* slik at den heksadesimale utskriften starter med 0x og den oktale med 0. Hint: bruk # som flagg. Lag et tillegg i koden slik at tallet også skrives ut på binær form med 0b først. Det finnes ikke noe konverteringssymbol eller flagg for dette. Men du kan f.eks. bruke metoden `Integer.toBinaryString()`. Sjekk hva utskriften blir for forskjellige verdier på *k* - både positive og negative.
4. Lag en tabell som viser tallene fra 1 til 31 skrevet ut på både desimal, oktal, heksadesimal og binær form. En linje for hvert tall og med i rette kolonner. Dvs. slik:

   ```
    1   1   1      1
    2   2   2     10
    3   3   3     11
    . . .
    . . .
   31  37  1f  11111
   ```

5. Bruk %s som utskriftsdirektiv for de ulike heltallstypene. Sjekk hva som blir utskrift.

B.4 Formatering av desimaltall

Et desimaltall som f.eks. π = 3,14 har en heltallsdel og en desimaldel. Heltallsdelen er 3, mens desimaldelen egentlig er uendelig lang. Med f.eks. 15 desimalers nøyaktighet blir det (trenger du flere kan du f.eks. slå opp på Wikipedia):

```
π = 3,141592653589793
```

Klassen `Math` har en konstant med navnet `PI` og den kan skrives ut på vanlig måte:

```
(B.4.1)  System.out.println(Math.PI);  // Utskrift: 3.141592653589793
```

`Math.PI` har *double* som datatype. Beregninger utført med datatypen *double* får normalt stor nøyaktighet, men selvfølgelig ikke eksakt siden den kun har 64 biter. Ofte er det nok å oppgi svaret med noen få desimaler. Slik formatering kan vi gjøre ved hjelp av et direktiv. Her bruker vi 5 desimalers nøyaktighet:

```
(B.4.2)  System.out.printf("%7.5f", Math.PI);  // Utskrift: 3,14159
```

Tar vi utgangspunkt i det generelle formateringsdirektivet, dvs. i

```
(B.4.3)  %[argumentnummer$][flagg][feltbredde][.presisjon]konverteringssymbol
```

ser vi at `B.4.2` bruker 7 som feltbredde, 5 som presisjon og f som konverteringssymbol, mens argumentnummer og flagg ikke inngår. Feltbredde er som vanlig størrelsen på det reserverte utskriftsfeltet og presisjon er antall desimaler som skal være med. Hvis tallet har flere desimaler enn det, vil det bli avrundet. Hvis vi i `B.4.2` bruker 4 istedenfor 5 som presisjon, vil utskriften bli 3,1416 og ikke 3,1415.

De aktuelle konverteringssymbolene for desimaltall (eng: floating point number) er a, A, e, E, f, g, G. De dekker datatypene `float`, `Float`, `double`, `Double` og `BigDecimal`. Symbolet f (floating point) som er brukt i `B.4.2`, er det som vanligvis brukes.

Symbolene e og E (e for eksponent) brukes til å skrive ut et desimaltall på vitenskaplig form (scientific notation). Ta tallene 123,456 og 0,00123 som eksempler. De kan også skrives som $1{,}23456 \cdot 10^2$ og $1{,}23 \cdot 10^{-3}$, dvs. på en entydig måte på formen $a \cdot 10^b$ der $1 \leq a < 10$. Dette gjelder alle positive tall. Hvis a er negativ, kan tallverdien til a skrives på denne formen. Tallet 0 er eneste unntak siden $0 = 0 \cdot 10^0$. Se flg. kode:

```
double a = -123.456, b = 0.00123, c = 0.0;
System.out.printf("%.4e %.4e %.4e%n", a, b, c);
// Utskrift: -1,2346e+02 1,2300e-03 0,0000e+00
```

Programkode* B.4 *a)

I direktivene i koden over inngår ikke *feltbredde*. Dermed brukes den plassen som trengs. Men *presisjon* inngår. Siden den er 4 blir det avrunding hvis det er flere enn 4 signifikante desimaler. Hvis det er færre, blir det lagt på 0-er.

Konverteringssymbolet g (og G) gir en mellomting mellom e og f. Det betyr at små tall (og store tall) skrives på vitenskaplig form, men ellers på vanlig desimal form. F.eks. vil 0,0001 bli skrevet ut som 0,0001, mens 0,00001 blir skrevet ut som 1,0e-05.

Symbolet a (og A) gir vitenskaplig form med 16 som grunntall. F.eks. vil 0,00001 med %a som direktiv, komme ut som 0x1.4f8b588e368f1p-17. Legg merke til at her inngår bokstaven e som et heksadesimalt siffer. Derfor brukes p som symbol for grunntallet 16. Med norsk som «locale» kommer det desimalkomma ved e og g, men ikke med a. Antageligvis en Java-feil.

Hvis en skriver ut uformatert ved hjelp av *print*() eller *println*(), blir det utskrift på vanlig desimal form for tall innenfor et bestemt størrelsesområde. Men utenfor det området (svært store eller små tall) kommer det på vitenskaplig form:

```
System.out.println(0.001 + "  " + 0.0001);          // 0.001  1.0E-4
System.out.println(1000000.0 + "  " + 10000000.0);  // 1000000.0  1.0E7
```

Programkode* B.4 *b)

De fleste av flaggene som gjelder for heltallsutskrift har tilsvarende funksjon for desimaltall (se heltallsflagg). Her er noen eksempler:

```
System.out.printf("%-+(6.2f%-+(6.2f%-+(6.2f%n", Math.PI, Math.E, -0.5);
// Utskrift: +3,14 +2,72 (0,50)
```

Programkode* B.4 *c)

Det er også mulig å bruke konverteringssymbolet s for desimaltall selv om s egentlig er for tegnstrenger. Det lages da en tegnstreng ved hjelp av en *toString*-metode. Hvis vi i tillegg bruker presisjon, får vi flg. effekter:

```
System.out.printf("%1$.4f  %1$.6s%n", Math.PI);  // 3,1416  3.1415
System.out.printf("%1$.4f  %1$.6s%n", 0.1);      // 0,1000  0.1
```

Programkode* B.4 *d)

Først skrives π ut som desimaltall (f som symbol) med 4 desimalers nøyaktighet. Da blir det 3,1416 på grunn av avrundig og ikke 3,1415 siden neste desimal er 9. Med 0,1 blir utskriften 0,1000 siden det legges på 0-er til det blir 4 desimaler. Men det blir annereledes med s som symbol. Siden 6 er presisjonen, blir tegnstrengen trunkert hvis den har en lengde større enn 6. I tilfellet π er den tilhørende tegnstrengen lik "3.141592653589793" og den blir trunkert til å ha 6 tegn, dvs. til "3.1415".

Oppgaver til Avsnitt B.4

1. Det er mulig å oppgi et desimaltall på heksadesimal form i Java-kode. Da må det starte med 0x og ha p som grunntallssymbol. Hva blir utskriften fra flg. kodebit:

   ```
   double x = 0x1.921fb54442d18p1;
   System.out.println(x);
   ```

   Lag så en *printf*-setning som skriver ut x på nøyaktig samme form som x er oppgitt.

2. Som nevnt over, kan desimaltall skrives ut med s som konverteringssymbol. Lag en slik *printf*-setning der både 0.001 og 0.0001 skrives ut. Gjør så det samme med 1000000.0 og 10000000.0. Sammenlign utskriftene med det som *Programkode* B.4 *b)* gir.

B.5 Formatering av tegn og tegnstrenger

For tegn (*char*/Character) gir konverteringssymbolet c(C) et tegn på Unicode-format. For tegnstrenger brukes s(S). Det gjelder egentlig generelt, dvs. for alle mulige datatyper. Hvis et direktiv som inneholder dette, ikke er knyttet til et tegnstrengargument, vil det bli dets *toString*-metode som vil definere utskriften. Hvis argumentet ikke er en referansetype, så konverteres det først til en instans av den tilhørende omslagsklassen (*int* til Integer, osv).

Det er kun ett flagg som gjelder for tegnstrenger og det er minustegnet. Det er tillatt kun når direktivet inneholder en feltbredde og fører til venstrejustering av utskriften. I de fleste tilfeller der tegnstrenger inngår som en del av en utskrift på tabellform, brukes venstrejustering. Tabell B.1.1 inneholder informasjon om en samling studenter (navn, studiepoeng og alder). Det kan f.eks. skrives ut slik:

```
String[] navn = {"Petter", "Kari", "Aleksander", "Elin"};
int[] poeng = {0, 140, 90, 70};
int[] alder = {19, 21, 20, 19};

for (int i = 0; i < navn.length; i++)
{
  System.out.printf("%-12s%3d%4d%n", navn[i], poeng[i], alder[i]);
}
```

Programkode* B.5 *a)

```
Petter        0  19
Kari        140  21
Aleksander   90  20
Elin         70  19
```

Det er ofte et problem å velge feltbredde for tegnstrenger som varierer i lengde. I eksemplet over brukes fornavn, men selv fornavn kan variere mye. Hvis det inngår doble og kanskje triple fornavn, blir variasjonen enda større. Hvis en velger for stor feltberdde, vil utskriften kunne inneholde mye «luft» og kan dermed bli lite lesbar. Hvis en har for liten feltbredde, kan en risikere å få tegnstrenger som bryter utskriftsmønsteret.

En løsning som ofte velges, er å bruke en rimelig stor feltbredde og så trunkere (kappe av) de som er for lange. Dette er det enkelt å få til ved å bruke *presisjon* i utskriftsdirektivet. Vi bruker eksemplet over på nytt og lar 10 være makslengde for navn og dermed %-12.10s som direktiv. Så lar vi Elin isteden hete Elin Aleksandra. Det gir:

```
String[] navn = {"Petter", "Kari", "Aleksander", "Elin Aleksandra"};
int[] poeng = {0, 140, 90, 70};
int[] alder = {19, 21, 20, 19};

for (int i = 0; i < navn.length; i++)
{
  System.out.printf("%-12.10s%3d%4d%n", navn[i], poeng[i], alder[i]);
}
```

Programkode* B.5 *b)

```
Petter        0  19
Kari        140  21
Aleksander   90  20
Elin Aleks   70  19
```

Syntaksen for *printf* er: *printf*(String format, Object... args). Men det er også mulig å bruke *printf*() med kun en tegnstreng som parameter. Da er det den som skrives ut, dvs. *printf*() oppfører seg som *print*():

```
String s = "abcdef\n";
System.out.printf(s);    // abcdef
System.out.print(s);     // abcdef
```

Programkode* B.5 *c)

Oppgaver til Avsnitt B.5

1. I utskriften i *Programkode* B.5 *b)* skal det være kolonner for navn, poeng og alder med to mellomrom mellom kolonnene. Det kan også ordnes ved å legge mellomrommene inn i i formateringssetningen. Gjør det!

B.6 Formatering av dato og klokkeslett

Java hadde allerede fra starten av (Java 1.0) klassen Date, men det ble allerede ganske tidlig klart at den ikke dekket behovet for et globalt sytem for å behandle datoer og klokkeslett. I et slikt system må en kunne ta hensyn til f.eks. tidssoner og sommer/vintertid. Det aller meste av hva Date inneholder er derfor «utgått» (deprecated). Isteden ble klassen Calendar innført.

Opp gjennom historien har det være flere kalendersystemer. De fleste har nok hørt om den julianske kalenderen innført av av Julius Cæsar i år 46 f.Kr. Den hadde 365 dager, pluss en skuddårsdag hvert fjerde år. Men dette var i gjennomsnitt litt mer enn et tropisk år (den tiden jorden bruker rundt solen). I løpet av 128 år ble feilen et helt døgn. Dette ble rettet opp av pave Gregor XIII i 1582 ved at at noen av skuddårsdagene ble fjernet. Dvs. år som er delelige med 100 (men ikke de som er delelige med 400) er ikke skuddår. Dette kalles i dag den gregorianske kalenderen.

Den gegorianske kalenderen er nå nærmest enerådende verden over. Men det finnes andre som er i bruk. F.eks. er det i Java tilrettelagt for to andre:

```
System.out.println(Calendar.getAvailableCalendarTypes());
// Utskrift: [gregory, buddhist, japanese]
```

Klassen Calendar er abstract, men har GregorianCalendar som subklasse. Dagens dato avhenger av tidssone (TimeZone) og navn på ukedager og måneder av språk (Locale). Hvis ikke noe annet er sagt, vil standardverdiene bli brukt. Det er oppsettet på maskinen din som avgjør hva som er standard. Hos undertegnede er tidssone lik *Central European Time* og språk lik *norsk*. Lager du kode som skriver ut flg. tegnstrenger, vil du se hva det er hos deg:

```
String tidssone = TimeZone.getDefault().getDisplayName();
String språk = Locale.getDefault().getDisplayLanguage();
```

Begrepet «nå», dvs. dette øyeblikk, har i seg både en dato og et klokkeslett. På norsk har vi vel ikke noe godt navn på et slikt begrep. Ordet *tidspunkt* kunne kanskje brukes siden et tidspunkt ofte har en dato (i tillegg til klokkeslett). Vi skal derfor si at en instans av klassen Calendar representerer et tidspunkt. Tidspunktet «nå» kan vi på flg. måter. Begge gir en instans av klassen GregorianCalendar:

```
Calendar nå1 = Calendar.getInstance();   // konstruksjonsmetode
Calendar nå2 = new GregorianCalendar();  // standardkonstruktør
```

Et tidspunkt er definert ved år, måned, dag og klokkeslett (timer, minutter, sekunder). I Java er dette internt representert ved hjelp av en *Long*-verdi, dvs. antall millisekunder siden 1. januar 1970. Tidspunktet «nå» får vi også ved hjelp av `System.currentTimeMillis()`:

```
System.out.println(Calendar.getInstance().getTimeInMillis());
System.out.println(System.currentTimeMillis());
```

I koden over vil det bli samme utskrift begge ganger. Det er selvfølgelig mulig å opprette en instans av `GregorianCalendar` knyttet til et hvilket som helst tidspunkt. Det er flere konstruktører for dette:

```
public GregorianCalendar(int år, int måned, int dag)
public GregorianCalendar(int år, int måned, int dag, int time,  int min)
public GregorianCalendar(int år, int måned, int dag, int time,  int min, int sek)
```

I flg. eksempel lages en instans av `GregorianCalendar` for julaften 2017. Her må en være obs på at månedene er nummerert fra 0 til 11. Med andre ord er desember lik 11. Alternativt kan en bruke `Calendar.DECEMBER`:

```
Calendar julaften1 = new GregorianCalendar(2017, 11, 24);
Calendar julaften2 = new GregorianCalendar(2017, Calendar.DECEMBER, 24);
```

Formatering handler om å få skrevet ut et tidspunkt (dato og/eller klokkeslett) på den formen vi måtte ønske. Flg. formateringsdirektiv gjelder for typene `Calendar`, `Date`, *`long`* og `Long`:

```
(B.6.1)  %[argumentnummer$][flagg][feltbredde]konverteringssymbol
```

Her inngår det ikke presisjon (se også B.1.1). Et *konverteringssymbol* består her av ***to*** tegn. Det første er enten t eller T, dvs. t* eller T* der * indikerer det andre tegnet. For datoer (dvs. datodelen av et tidspunkt) kan * være et av flg. tegn:

- y/Y for årstall der Y gir gir fire sifre (f.eks. 2017), mens y gir de to siste (f.eks. 17)
- C for de to første sifrene i årstallet (f.eks. 20 for 2017)
- b/B for måned der b er kortformen (f.eks. des) og B hele navnet (f.eks. desember). Ved T istedenfor t blir det store bokstaver (DES/DESEMBER)
- h virker som b
- m for måned på tallform, dvs. fra 01 til 12
- a/A for ukedagen med a på kortform (f.eks. søn) og A hele navnet (f.eks. søndag). Ved T istedenfor t blir det store bokstaver (SØN/SØNDAG)
- e/d for dagnummer i måned der e gir fra 1 til 31, mens d gir fra 01 til 31
- j for dagnummer på årsbasis, dvs. et tall fra 001 til 365 (eller 366 i skuddår)

Eksempler:

```
Calendar julaften = new GregorianCalendar(2017, Calendar.DECEMBER, 24);
Calendar mai17 = new GregorianCalendar(2017, Calendar.MAY, 17);
Calendar nå = Calendar.getInstance();

String format = "%1$tA %1$te. %1$tB %1$tY%n";
System.out.printf(format, julaften).printf(format, mai17).printf(format, nå);

// søndag 24. desember 2017
// onsdag 17. mai 2017
// lørdag 21. oktober 2017 (eller egentlig den dagen koden kjøres)
```

Programkode* B.6 *a)

Legg merke til at formateringssetningen over inneholder (bortsett fra %n) fire direktiver som alle har argumentnummer 1 (dvs. 1$). Det betyr at alle refererer til første (og eneste) argument i *printf*(). Det hadde vært mulig å få til det samme uten argumentnummer, men da måtte vi sette opp det samme argumentet fire ganger. Tar vi instansen med navn *nå*, må det gjøre slik:

```
Calendar nå = Calendar.getInstance();
String format = "%tA %te. %tB %tY%n";
System.out.printf(format, nå, nå, nå, nå);
// lørdag 21. oktober 2017 (eller egentlig den dagen koden kjøres)
```

Programkode* B.6 *b)

Formateringsdirektivene gjelder som nevnt over, også for typene *long* og Long fordi et tidspunkt er definert som et bestemt antall millisekunder siden den 1. januar 1970. I flg. eksempel inngår longtallene 0 og System.*currentTimeMillis*():

```
String format = "%1$tA %1$te. %1$tB %1$tY%n";
System.out.printf(format, 0L).printf(format, System.currentTimeMillis());

// torsdag 1. januar 1970
// lørdag 21. oktober 2017 (eller egentlig den dagen koden kjøres)
```

Programkode* B.6 *c)

Som nevnt over består et *konverteringssymbol* her av ***to*** tegn, dvs. t* eller T* der * utgjør det andre tegnet. For klokkeslett (dvs. klokkeslettdelen av et tidspunkt) kan * være et av disse (og i tillegg noen som normalt ikke er aktuelle for norske forhold):

- H for timetall fra 00 til 23 (to siffer for tall mindre enn 10)
- l som H, men fra 0 til 23 (kun ett siffer for tall mindre enn 10)
- I for timetall fra 00 til 12. F.eks. blir kl. 13 da isteden kl. 01
- M for minutter fra 00 til 59
- S for sekunder fra 00 til 59 (og i noen tilfeller til 60)
- L for millisekunder fra 000 til 999
- Z for aktuell tidssone

Eksempler (i alle direktiver angående timer, minutter osv. virker t og T likt):

```
String format = "%1$tA %1$te. %1$tB %1$tY kl. %1$tH:%1$tM:%1$tS%n";
System.out.printf(format, 0L).printf(format, System.currentTimeMillis());

// torsdag 1. januar 1970 kl. 01:00:00
// lørdag 21. oktober 2017 kl. 19:13:00 (eller egentlig når koden kjøres)
```

Programkode* B.6 *d)

Direktivkombinasjoner som ofte brukes, har fått egne konverteringssymboler:

- %tR er det samme som %tH:%tM
- %tT er det samme som %tH:%tM:%tS
- %tD er det samme som %tm/%td/%ty
- %tF er det samme som %tY-%tm-%td
- %tc er det samme som %ta %tb %td %tT %tZ %tY

I flg. eksempler brukes tidspunktet ***nå*** og utskriften er den som kom da koden ble kjørt:

```
long nå = System.currentTimeMillis();

System.out.printf("%1$tH:%1$tM%n", nå);  // 13:46
System.out.printf("%tR%n", nå);          // 13:46

System.out.printf("%1$tH:%1$tM:%1$tS%n", nå);  // 13:46:50
System.out.printf("%tT%n", nå);                // 13:46:50

System.out.printf("%1$ta %1$tb %1$td %1$tT %1$tZ %1$tY%n", nå);
// ma okt 23 13:46:50 CEST 2017
System.out.printf("%tc%n", nå);  // ma okt 23 13:46:50 CEST 2017
```

Programkode* B.6 *e)

I utskriften over står CEST for Central European Standard Time. Hvis en bruker `%Tc` istedenfor `%tc` i den siste setningen, kommer det store bokstaver i kortnavnet for ukedag og måned.

Dette avsnittet handler om formatering av et «tidspunkt» (instanser av klassene `Calendar`, `Date`, *`long`* og `Long`). Vi har ikke diskutert hva f.eks. `Calendar`/`GregorianCalendar` inneholder og kan brukes til. Det krever nesten et eget kapittel. Her tar vi kun et eksempel på hvordan en instans av `GregorianCalendar` kan brukes til å lage en «klokke». En klokke «tikker». Her kan en velge om «tikkingen» er per sekund, per dag eller kanskje per år. «Tikkingen» utføres ved hjelp av metoden *`roll()`*. I flg. kode blir ukedagen for julaften i årene fra 2017 til 2026 (en ti-års periode) skrevet ut:

```
Calendar c = new GregorianCalendar(2017, Calendar.DECEMBER, 24);
for (int i = 0; i < 10; i++)
{
  System.out.printf("Julaften %1$tY faller på en %1$tA%n", c);
  c.roll(Calendar.YEAR, true);
}
// Julaften 2017 faller på en søndag
// Julaften 2018 faller på en mandag
// . . . .
// Julaften 2026 faller på en torsdag
```

Programkode* B.6 *f)

Oppgaver til Avsnitt B.6

1. Gjør om *Programkode* `B.6` *`f)`* slik at det starter med inneværende år. Bruk kode som gir det året.
2. Lag en metode som returnerer en tegnstreng (String) med navn på den ukedagen julaften faller på i et bestemt år. Årstallet (fire siffer) skal være argument til metoden.
3. Lag en metode som returnerer årstallet for (fra og med inneværende år) første år der julaften faller på en oppgitt ukedag. Ukedagen (søndag = 1, mandag = 2, osv.) skal være argument i metoden. Bruk metoden til å finne første år fra og med i år der julaften faller på en onsdag.
4. Datatypen `Calendar` er sammenlignbar (implementerer `Comparabel<Calendar>`). Opprett en tabell med tidspunkter og sorter den.

B.7 Andre formateringer

Konverteringssymbolet ***b*** (***B***) står for boolean. Hvis det tilhørende argumentet er null, gir det *false* (eller *FALSE*). Hvis det er av en logisk type (*boolean* eller Boolean), gir det argumentets verdi. I alle andre tilfeller gir det true (eller TRUE).

```
String s = null;
boolean b = true;
int k = 100;
System.out.printf("%b %b %b%n", s, b, k);
// Utskrift: false true true
```

Programkode* B.7 *a)

Konverteringssymbolet ***h*** (***H***) står for heksadesimal og for hash. Alle referansetypene i Java har, siden de er subtyper av Object, metoden *hashCode*(). Konverteringssymbolet h (eller H) gir hashverdien (det som *hashCode*-metoden returnerer) på heksadesimal form. Dette vil også gjelde for de grunnleggende typene siden de gjøres om til en referansetype ved hjelp av en omslagsklasse (*int* til Integer, osv). Se flg. eksempel:

```
System.out.printf("%h %h %h%n", "ABCDE", true, 100);  // 3b2fc43 4cf 64
```