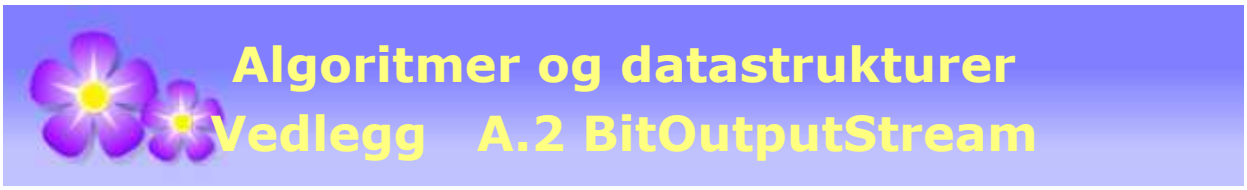

A.2 BitOutputStream

A.2.1 Instansiering og skriving

BitOutputStream har fire konstruktører og to konstruksjonsmetoder (eng: factory methods):

```
public BitOutputStream(OutputStream out)                  // konstruktør
public BitOutputStream(OutputStream out, int size)        // konstruktør
public BitOutputStream(String fileName)                   // konstruktør
public BitOutputStream(String fileName, int size)         // konstruktør
public static BitOutputStream toFile(String fileName)
public static BitOutputStream toFile(String fileName, boolean append)
```

Programkode A.2.1 a)

BitOutputStream bruker en *int*-variabel og en *byte*-tabell som interne buffere for å kunne gjøre bithåndteringen og fil-skrivingen mest mulig effektiv. Når *byte*-tabellen blir full skrives innholdet automatisk ut ved hjelp av én skriveoperasjon. Standardstørrelsen på *byte*-tabellen er 4kb, men det er mulig å velge en annen størrelse (parameteren *size*). En kan enten knytte en BitOutputStream til en OutputStream eller til et filnavn.

Hvis vi skal opprette en instans av BitOutputStream knyttet til en filen *"utfil.txt"* (og med standard bufferstørrelse), kan vi gjøre det på en av flg. to måter. Hvis det er en fil med dette navnet fra før, blir den overskrevet (egentlig fjernet og erstattet med en med samme navn):

```
String utfil = "utfil.txt";
BitOutputStream ut1 = new BitOutputStream(utfil);     // konstruktør
BitOutputStream ut2 = BitOutputStream.toFile(utfil);  // konstruksjonsmetode
```

Programkode A.2.1 b)

Hvis *"utfil.txt"* finnes fra før og vi ønsker å skrive videre på den, settes *append* til *true*:

```
BitOutputStream ut1 = BitOutputStream.toFile("utfil.txt", true);
BitOutputStream ut2 = new BitOutputStream(new FileOutputStream(utfil,true));
```

En instans av BitOutputStream «lukkes» ved **void** *close*().

BitOutputStream har åtte skrivemetoder:

```
public void write0Bit()                                  // 0-bit
public void write1Bit()                                  // 1-bit
public void writeBit(int bit)                            // den siste i bit
public void write(int value)                             // 8 biter
public void writeBits(int value, int numberOfBits)       // numberOfBits biter
public void writeBits(int value)                         // signifikante biter
public void writeLeftBits(int value, int numberOfBits)   // numberOfBits biter
public void writeLeftBits(int value)                     // omvendt signifikante
```

Programkode A.2.1 c)

De tre første *write*-metodene er vel selvforklarende. Metoden *write*(**int** value) skriver ut de åtte siste og *writeBits*(**int** value, **int** numberOfBits) de *numberOfBits* siste bitene i *value*. Metoden *writeBits*(**int** value) skriver ut de signifikante bitene, dvs. alle bitene fra og med første (fra venstre) 1-bit. Det betyr spesielt at hvis *value* er negativ, vil alle bitene bli skrevet ut. Tallet 0 har egentlig ingen signifikante biter, men hvis *value* er 0, vil det likevel bli skrevet ut en 0-bit. Se flg. eksempel. Hva vil filen inneholde i tekstformat?

```
BitOutputStream ut = BitOutputStream.toFile("utfil.txt");

ut.writeBit(0);          // skriver ut en 0-bit
ut.writeBits(32);        // skriver ut de signifikante bitene, dvs. 100000
ut.write1Bit();          // skriver ut en 1-bit
ut.writeBits('B', 16);   // skriver ut 0000000001000010

ut.close();              // VIKTIG: Avslutt alltid med close()
```

Programkode* A.2.1 *d)

Vi får bitsekvensen 01000001000000000001000010 = 01000001 00000000 01000010 = A B

De to siste *write*-metodene er mer spesielle. Det normale er at biter blir hentet fra høyre ende av parameterverdien. Men i noen situasjoner kan det være aktuelt å hente bitene fra venstre ende. Metoden *writeLeftBits*(**int** value, **int** numberOfBits) skriver derfor ut de *numberOfBits* første bitene i *value*. Metoden *writeLeftBits*(**int** value) skriver ut de «omvendt signifikante» bitene i *value*, dvs. alle bitene (fra venstre) til og med siste 1-bit. Hvis f.eks. siste bit i *value* er en 1-bit, vil alle bli skrevet ut. Hvis *value* er 0, blir det ikke skrevet ut noe:

```
BitOutputStream ut = BitOutputStream.toFile("utfil.txt");

int a = 'A' << 24;       // a = 01000001000000000000000000000000

ut.writeLeftBits(a);     // skriver ut 01000001
ut.writeLeftBits(a, 8);  // skriver ut 01000001

ut.close();              // VIKTIG: Avslutt alltid med close()
```

Programkode* A.2.1 *e)

Oppgaver til A.2.1

1. BitOutputStream er en subklasse av klassen OutputStream. Metoden write(**int** value) (se *Programkode* A.2.1 *c*) arves derfra. Den skriver ut en *byte* (dvs. de siste åtte bitene i *value*). OutputStream har to utskriftsmetoder til (står ikke i *Programkode* A.2.1 *c*). De skriver ut av en *byte*-tabell. Lag en BitOutputStream slik som i *Programkode* A.2.1 *d)*, lag en *byte*-tabell med bokstavene A, B, C og D og skriv den til "*utfil.txt*". Sjekk etterpå at filen fikk det innholdet.
2 Lag kode slik at "*utfil.txt*" får bokstavene A, B, C og D med kun to *write*-setninger. Det er kun de fra *Programkode* A.2.1 *c* som skal brukes.
3 Som i Oppgave 2, med med kun én *write*-setning fra *Programkode* A.2.1 *c*.

A.2.2 Missing bits, close og flush

Avslutningsmetoder:

```
public int missingBits()                // det som mangler på en hel byte
public void flush()                     // tømmer (og skriver ut) bufrene
public void close()                     // flusher og lukker
```

Programkode A.2.2 a)

Hvis en skal skrive ut et variabelt antall biter, vil som oftest ikke det sammenlagte antallet biter bli delelig med 8, dvs. at det tilsammen ikke blir et helt antall byter. Se flg. eksempel:

```
BitOutputStream ut = BitOutputStream.toFile("utfil.txt");

ut.writeBits(-1,9);     // skriver ut 9 biter, dvs:  111111111
ut.writeBits(45,7);     // skriver ut 7 biter, dvs:  0101101
ut.writeBits(22,5);     // skriver ut 5 biter, dvs:  10110

ut.close();             // VIKTIG: Avslutt alltid med close()
```

Programkode A.2.2 b)

I koden over blir det skrevet ut 9 + 7 + 5 = 21 biter. Det gir et problem. Normalt er én byte den minste enheten for utskrift. `BitOutputStream` løser dette ved at biter fortløpende settes sammen til byter som så skrives ut. Dette går bra inntil vi skal avslutte. Da kan det være at vi ikke har nok biter til å fylle den siste byten. Systemet løser det selv ved at et tilstrekkelig antall 0-biter legges på til slutt. I *Programkode* A.2.2 *b)* der det skrives ut 21 biter, vil det internt bli lagt til 3 ekstra 0-biter bakerst. Tilsammen 24 biter som utgjør 3 byter. Utskriftsfilen *"utfil.txt"* vil derfor komme til å inneholde flg. 24 biter:

```
111111111010110110110000
```

Når en fil som er laget på denne måten, senere skal leses, vil problemet dukke opp igjen. Hvis det ligger en eller flere 0-biter bakerst på filen, er det ikke mulig å vite om det er biter vi har skrevet ut eller om det er biter som systemet selv har lagt til. Hvis det er viktig å vite dette, må vi «notere» det antallet 0-biter som systemet har lagt inn. Eventuelt kan vi i steden «notere» det totale antallet biter som *write*-metodene har skrevet ut.

Metoden `public int` *missingBits*`()` forteller til enhver tid hvor mange biter som mangler for å kunne fylle den siste byten, dvs. et heltall i intervallet [0,7]. Hvis f.eks. den kalles rett før vi lukker, så får vi vite hvor mange 0-biter systemet kommer til å legge til bakerst.

Det er særdeles viktig at vi lukker ved hjelp av metoden *close*() når vi er ferdige med skrivingen. Hvis ikke kan vi risikere at ikke alle bitene blir skrevet ut. Det kommer av at `BitOutputStream` bruker et internt buffer-system. Den interne *byte*-tabellen blir automatisk «tømt» (dvs. innholdet skrives ut) når det er fullt. Men det vil normalt ligge noe igjen i begge de interne bufferne (en *int* og en *byte*-tabell) når vi skal avslutte. Dette blir garantert skrevet ut når *close*() kalles.

Det vil kunne være situasjoner der vi må «tømme» bufferne selv om vi ennå ikke er ferdige med skrivingen. Metoden *flush*() er laget for dette formålet. En må da være klar over at det vil bli lagt til tilstrekkelige mange 0-biter bakerst slik at det blir et helt antall byter. Metoden *missingBits*() vil som nevnt over, fortelle hvor mange. OBS: Det er ikke nødvendig eksplisitt å kalle *flush*() før *close*(). Det skjer allerede implisitt i *close*().

Oppgaver til A.2.2

1. Kjør *Programkode* A.2.2 *b)* og les filen *"utfil.txt"* som en tekstfil. Er den lesbar?
2. Legg inn en fjerde utskrift med lengde 3 i *Programkode* A.2.2 *b)*. Bytt ut tallene -1, 5 og 22 med tall slik at slik at *"utfil.txt"* vil inneholde ABC.

A.2.3 Testing av metoder

I kodeeksemplene i avsnittene over skrev vi til en navngitt fil. Ved så å lese filen som en tekstfil, kunne vi sjekke at den inneholdt det som vi forventet. Men hvis vi skriver med et variabelt antall biter, vil filen kunne bli uleselig som en tekstfil. En mulig løsning er da å kopiere filinnholdet over i en *byte*-tabell og så analysere tabellen. Men dette kan vi få til uten å gå veien om den første filen. Vi kan isteden skrive til en ByteArrayOutputStream og så hente resultatet fra dens interne *byte*-tabell. Vi gjør litt om på *Programkode* A.2.2 *b)*:

```
ByteArrayOutputStream utskrift = new ByteArrayOutputStream();
BitOutputStream ut = new BitOutputStream(utskrift);

ut.writeBits(-1,9);     // skriver ut 9 biter, dvs:  111111111
ut.writeBits(45,7);     // skriver ut 7 biter, dvs:  0101101
ut.writeBits(22,5);     // skriver ut 5 biter, dvs:  10110

ut.close();             // legger til tre ekstra 0-biter, dvs. 000

for (byte b : utskrift.toByteArray()) System.out.print(b + " ");
// Utskrift: -1 -83 -80
```

Programkode A.2.3 *a)*

Metoden *toByteArray*() gir oss den interne *byte*-tabellen. Vi skriver ut 21 biter. Systemet legger til 3 0-biter. Dermed: 111111111010110110110000. Deler vi dette i åtte og åtte biter får vi 11111111 10101101 10110000. Den første (med kun 1-ere) representerer tallet -1, den andre -83 og det tredje -80. Alle er negative siden de starter med en 1-bit.

Det kunne også være interessant å se selve bitinnholdet i en *byte*-tabell. BitOutputStream har flere muligheter for dette:

```
public static String toBitString(byte... b)
public static String toBitStringWithSpace(byte... b)
public static String toBitString(byte[] b, int[] part)
public static String toBitString(byte[] b, int numberOfBits)
```

Programkode A.2.3 *b)*

Metoden *toBitString*(byte... b) returnerer en *String* med bitene (som '0' og '1'). Metoden *toBitStringWithSpace*(byte... b) gjør det samme, men med et mellomrom (space) for hver åttende bit (en byte) (Obs: byte... betyr en generalisert tabell):

```
byte[] b = {-1, -83, -80};  // bitsekvensen 111111111010110110110000

String biter1 = BitOutputStream.toBitString(b);
String biter2 = BitOutputStream.toBitStringWithSpace(b);

System.out.println(biter1 + "  " + biter2);
// Utskrift: 111111111010110110110000  11111111 10101101 10110000
```

Programkode A.2.3 *c)*

Det er også mulig å dele opp *byte*-tabellen i noe annet enn 8 og 8 biter, f.eks. i enheter på 5 og 5. Men hvis det totale antallet biter ikke er delelig med 5, vil den siste enheten få færre enn 5. I en *byte*-tabell med tre elementer er det 24 biter og dermed fire enheter med 5 biter og til slutt én enhet med 4 biter:

```
byte[] b = {-1, -83, -80};  // bitsekvensen 111111111010110110110000
System.out.println(BitOutputStream.toBitString(b, 5));
// Utskrift: 11111 11110 10110 11011 0000
```

Programkode A.2.3 d)

Det må ikke deles opp i like store enheter. I *Programkode A.2.3 a)* tabellen til ved utskrift av forskjellige lengder. Vi kan dele den opp på samme måte:

```
byte[] b = {-1, -83, -80};  // bitsekvensen 111111111010110110110000
int[] oppdeling = {9, 7, 5};
System.out.println(BitOutputStream.toBitString(b, oppdeling));
// Utskrift: 111111111 0101101 10110
```

Programkode A.2.3 e)

Summen av tallene i oppdelingstabellen må ikke, slik som også eksemplet over viser, være er lik antall biter i *byte*-tabellen. Hvis oppdelingen f.eks. er på 1, 2, 3 og 4 (sum = 10), kommer kun de 10 første bitene ut. Hvis den f.eks. er på 1, 2, 3, 4, 5, 6 og 7 (sum = 28), vil det kun komme ut 3 biter siste gang (og ikke 7) siden det kun er 3 biter igjen.

```
byte[] b = {-1, -83, -80};  // bitsekvensen 111111111010110110110000
int[] oppdeling = {1, 2, 3, 4, 5, 6, 7};
System.out.println(BitOutputStream.toBitString(b, oppdeling));
// Utskrift: 1 11 111 1110 10110 110110 000
```

Programkode A.2.3 f)

Oppgaver til A.2.3

1. Skriv ut én og én bit (et mellomrom mellom hver) fra tabellen {-1, -83, -80}.

A.2.4 Metodeoversikt

```
public BitOutputStream(OutputStream out)              // konstruktør
public BitOutputStream(OutputStream out, int size)    // konstruktør
public BitOutputStream(String fileName)               // konstruktør
public BitOutputStream(String filename, int size)     // konstruktør
public static BitOutputStream toFile(String fileName) // konstr.metoder
public static BitOutputStream toFile(String fileName,  boolean append)

public void writeBit(int bit)      // den siste biten i bit
public void write0Bit()            // en 0-bit
public void write1Bit()            // en 1-bit
public void write(int b)           // de siste 8 bitene i b
public void writeBits(int value, int numberOfBits)  // de bakerste
public void writeBits(int value)     // de signifikante bitene i value
public void writeLeftBits(int value, int numberOfBits) // de første
public void writeLeftBits(int value)  // de omvendt signifikante bitene

public void flush()                // tømmer bufrene og skriver ut
public int missingBits()           // det som magler på en full byte
public void close();               // lukker filen

public static String toBitStringWithSpace(byte... b)  // åtte og åtte
public static String toBitString(byte... b)  // sammenhengende
public static String toBitString(byte[] b, int numberOfBits) // gruppevis
public static String toBitString(byte[] b, int[] part)       // gruppevis
```