Läsanvisningar

Börja med att läsa kapitlen i kursboken, därefter resurserna på Internet och till sist själva lektionsmaterialet.

Kursboken (5-8:e upplagan): Kapitel 16.1, 16.2.5, 16.3.4, 16.6, 14.7

Internet: The Java Tutorial, Trail: Essential Classes: - Basic I/O
(http://docs.oracle.com/javase/tutorial/essential/io/streams.html)

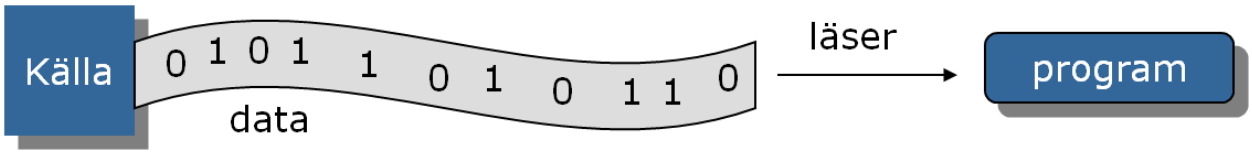
Ej avsnitten om Byte Streams, Data Streams och Object Streams. Inget under kapitlet "File I/O (Featuring NIO.2)" ingår.

Internet: The Java Tutorial, Trail: Creating a GUI With JFC/Swing: - How to Use File Choosers
(http://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html)

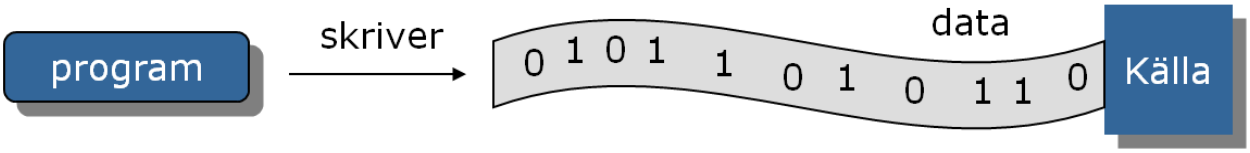
Filhantering

I Java sker all kommunikation (till/från tangentbord, filer, nätverk) via så kallade strömmar. En ström i Java hanterar en sekvens av antingen byte eller tecken. Teckenströmmar används för att läsa/skriva tecken (char) medan byteströmmar är tänkt att användas för att läsa/skriva binärdata. En ström kopplas mellan en källa och en destination.

För att läsa information till en applikation, öppnas en ström från en informationskälla (till exempel en fil på hårddisken, tangentbordet eller en resurs på internet) och läser denna information sekventiellt tecken för tecken (eller byte för byte beroende på vilken typ av ström som används).

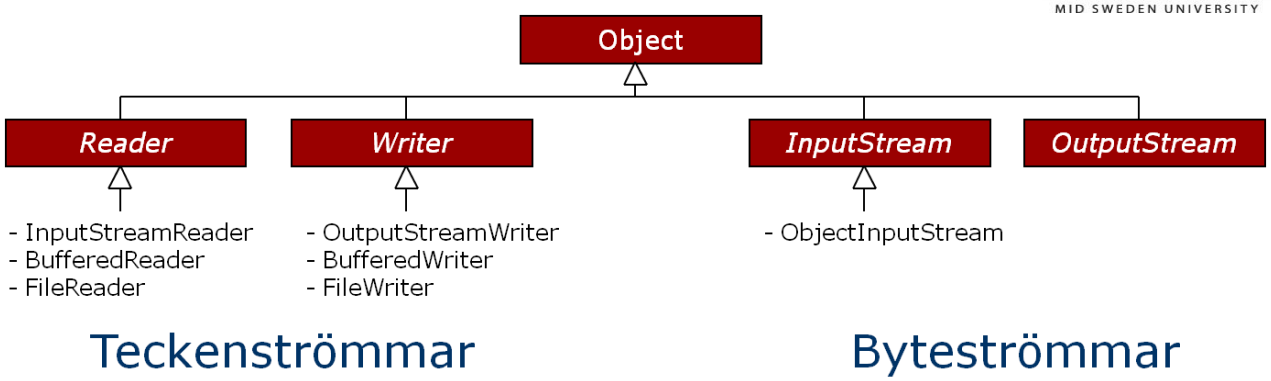


På liknande sätt kan ett program skicka information till en extern destination genom att öppna en ström till destinationen och skriva informationen sekventiellt.



Paket för filhantering

I paketet java.io finns de allra flesta klasserna för att läsa från och skriva till strömmar. Paketet java.io är det mest omfattande paketet som följer med i JDK. Klasserna i paketet delas in i tre huvudgrupper. Inströmmar som hanterar strömmar för att läsa från en källa, utströmmar som hanterar strömmar för att skriva till en destination, och diverse filklasser (bl.a. klassen File för att representera en fil i aktuell plattform). In- och utdataströmmarna delas även in i två olika typer som nämnts tidigare, nämligen teckenströmmar och byteströmmar.



En teckenström känner man igen genom att ordet Reader eller Writer återfinns i klassnamnet. En byteström känner man igen genom att ordet Stream återfinns i klassnamnet.

Vi kommer i denna lektion endast ta en kort titt på ett par vanliga klasser i paketet java.io och någon närmare förklaring på hur exakt dessa ska användas kommer inte att ges. Mer om strömmar kommer i kursen Java III.

Utöver paketet java.io finns det fler paket som har att göra med filhantering. I JDK 7 har det dessutom tillkommit ytterligare paket. Bland annat paketet java.nio.file för att hantera filer och filsystem. Det är dock inget vi titta på i denna lektion.

Standardströmmar i Java

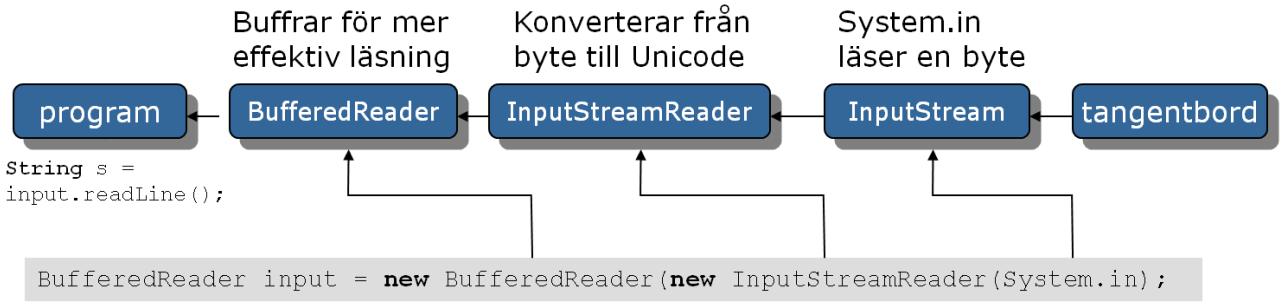
För att kunna använda en ström måste denna skapas genom att skapa ett objekt av någon av klasserna i java.io. Dock finns det redan tre strömmar skapade när ett Javaprogram körs. Dessa är System.in, System.out och System.err (in, out, err är publika och statiska objekt i klassen System). System.in är av typen InputStream och är kopplad till ”källan” tangentbordet. System.out och System.err är av typen PrintStream är kopplade till ”destinationen” kommandofönstret.


```
int b = System.in.read();
```

Klassen PrintStream innehåller bl.a. metoderna print och println för att skriva ut olika typer av data (primitiva typer och objekt). Du känner förhoppningsvis igen dessa metoder från System.out.println? Med metoden read i InputStream läser man en byte i taget från källan. Detta är väldigt opraktiskt och inte heller speciellt effektivt att använda om man i en applikation vill läsa strängar från tangentbordet.

Filtrera strömmar

Det är sällan att man enbart använder en klass i java.io när man läser/skriver via strömmar. Normalt använder man flera olika klasser som ”kopplas” till varandra. Detta för att låta en ström filtrera data från en annan ström. Du som har använt klassen BufferedReader för att läsa data från tangentbordet har gjort detta. Du har då kopplat strömmen System.in till en ström av klassen InputStreamReader (som konverterar mellan de olika typerna av strömmar; byte och tecken). Denna ström kopplas sen till en ström av klassen BufferedReader, som innehåller metoden readLine, för att läsa en hel rad från tangentbordet.



Ett annat exempel på när olika strömmar behöver kopplas ihop är om vi vill läsa ett tal från en fil i en zip-fil. Om vi vill läsa tal från en zip-fil kan vi använda klassen FileInputStream för att läsa binärdata från zip-filen. Vi kopplar denna ström till en ström av klassen ZipInputStream som kan läsa filer i zip-formatet (packade och opackade). Vi kopplar denna ström i sin tur till en ström av klassen DataInputStream som innehåller metoder för att läsa primitiva typer. Exempel på hur vi hanterar ZIP-filer i Java får du i kursen Java III.

Undantagshantering vid filhantering

Alla strömmar som läser eller skriver data kan kasta ett undantag (exception) om något oförutsett händer. Det kan t.ex. vara att filen vi vill läsa från inte finns eller att nätverkskopplingen till den fjärrdator vi kommunicerar med bryts. Som nämnts i tidigare lektioner måste vi ta hand om dessa fel på något sätt. Enklast är att enbart kasta dem vidare från den metod felet uppstod i. Som du kommer ihåg gör vi detta genom att skriva throws typ_av_exception i metoddeklarationen.

Normalt vill vi dock kontroller vad det är för fel som har uppstått för att till exempel se om vi kan åtgärda det på något sätt (som att ange ett annat filnamn om en fil som ska läsas inte finns). Detta gör vi genom att fånga eventuella fel som kan uppstå genom en vanlig try-catch i källkoden. Den kod som kan generera ett exception omsluter vi i ett try-block. Efter blocket anger vi de eventuella fel som koden i try-blocket kan generera (catch), tillsammans med den kod som ska utföras om felet uppstår.

```
public String getInput() {
  try {
    return input.readLine();
  }
  catch (IOException e) {
    System.err.println("Något gick fel vid inmatning:" + e.toString());
  }
}
```

Läsa och skriva textfiler

Att i en applikation läsa från och skriva till textfiler är så pass vanligt att det i Java finns speciella strömmar för detta. Dessa är FileReader och FileWriter. Minns du förresten vilken typ av ström dessa klasser hanterar (byte- eller teckenströmmar)? För att skapa en ström till en fil anger man namnet på filen i konstruktorn. Finns inte den fil vi anger som argument till konstruktorn, skapas den automatiskt. Om filen finns sedan tidigare skrivs innehållet i filen över.

```
FileReader in = new FileReader("filnamnet");
FileWriter out = new FileWriter("filnamnet");
// Filnamnet kan t.ex. vara "minFil.txt","resultat.dat","page.html"
```

En FileReader är en subklass till InputStreamReader och används för att på ett enkelt sätt kunna läsa

tecken från en fil, medan en FileWriter är en subklass till OutputStreamWriter och då givetvis används för att på ett enkelt sätt kunna spara tecken till en fil.

För att mer effektivt skriva data till en fil kopplar vi en ström av typen FileWriter (som i sin tur är kopplad till en fil) till en ström av klassen BufferedWriter. Denna klass innehåller metoden write() som skriver en sträng till filen.

```
FileWriter fw = new FileWriter("minfil.txt");
BufferedWriter filut = new BufferedWriter(fw);
```

För att åstadkomma en radbrytning måste metoden newLine() användas. Anledningen till att vi inte kan använda \n är att olika plattformar hanterar radbrytning på lite olika sätt.

```
filut.write("Programmering i"); // Skriver en sträng
filut.newLine();                // Skriver radbrytning
filut.write("Java\n");          // Radbrytning på fel sätt
filut.close();                  // tömmer och stänger
```

Det vi alltid ska komma ihåg när vi skriver data till en källa är att stänga strömmen genom att anropa metoden close. Gör vi inte detta är det inte säkert att de data vi skrivit till strömmen sparas.

För att göra skrivning av data till filer ännu mer bekvämt kan vi använda en ström av klassen PrintWriter. Denna klass innehåller metoderna print och println (som vi är vana att använda från System.out). Med dessa metoder kan vi nu enkelt skriva både strängar och primitiva typer till en fil. Radbrytningar hanteras även korrekt oavsett vilken plattform som används.

```
FileWriter fw = new FileWriter("minfil.txt");
BufferedWriter bw = new BufferedWriter(fw);
PrintWriter filut = new PrintWriter(bw);

filut.println("Programmering i");
filut.println("Java");
filut.print(100);

filut.close();  // Stäng filen när vi inte ska skriva mer till den
```

Vi har redan tidigare använt BufferedReader för att läsa hela rader från tangentbordet. Då kopplade vi till denna en ström av InputStreamReader som i sin tur var kopplad till System.in. För att använda metoden readLine för att läsa en rad från en fil kan vi koppla en ström av klassen BufferedReader till en FileReader.

Som med FileWriter anger vi i konstruktorn till FileReader den fil som vi vill koppla strömmen till. Finns inte filen genereras ett FileNotFoundException som vi antingen kastar vidare med throws eller fångar upp med try-catch. Metoden readLine returnerar en sträng för varje rad i filen. När det inte finns fler rader att läsa i filen returneras null. Detta kan vi utnyttja i en while-loop som fortsätter att loopa så länge som det returnerade värdet inte är null.

```
FileReader in = new FileReader("minfil.txt");
BufferedReader filin = new BufferedReader(in);

String rad = filin.readLine(); // Läs första raden i filen

while (rad != null) {           // Returnerar null när filen är slut
  System.out.println(rad);     // Skriver ut innehållet till kommandofönstret
  rad = filin.readLine();      // Läser nästa rad i filen
```

```
}

filin.close();  // Stäng filen när vi inte ska läsa mer
```

Om du i stället för att läsa rad för rad från filen vill läsa ord för ord kan du använda klassen Scanner och dess metod hasNext. Använd fortfarande klasserna FileReader och BufferReader, men koppla din Scanner till den senare enligt nedanstående exempel.

```
FileReader in = new FileReader("minfil.txt");
BufferedReader filin = new BufferedReader(in);
Scanner scanner = new Scanner(filin);

while (scanner.hasNext()) {    // false när inga fler ord finns att läsa
  String ord = scanner.next(); // Läs nästa ord i filen
  System.out.println(ord);     // Skriver ut ordet till kommandofönstret
}

scanner.close();  // Stäng när vi inte ska läsa mer
```

Klassen File

En fil eller en katalog representeras i Java med ett objekt av typen File (paketet java.io). Denna skapas genom att ange filens eller katalogens namn som en sträng. Vi kan även ange en sökväg till filen. I det senare fallet måste du vara uppmärksam på vilket tecken du använder för att separera delarna i sökvägen eftersom olika plattformar använder olika tecken. I exemplet nedan använder jag det tecken som gäller i en Windows-miljö.

```
File minFil = new File("data.txt");
File minKatalog = new File("Mina dokument");
File minFilMedSökväg = new File("c:\\katalog1\\katalog2\\data.txt");
// Bör använda File.separator eller File.separatorChar i stället för \
```

Det skapade objektet refererar till en fil eller katalog i användarens filsystem. Det finns inga garantier på att filen eller katalogen verkligen existerar. Inga undantag kommer till exempel att kastas om vi till konstruktorn anger ett filnamn som inte existerar.

När vi har skapat ett File-objekt finns det metoder vi kan anropa för att kontrollera om den fil eller katalog vi refererar till verkligen existerar och för att kontrollera om objektet i fråga refererar till en fil eller en katalog.

```
if (minFil.exists()) {
  // Gör någonting om minFil existerar
}

if (minFile.isFile()) {
  // Gör någonting om minFil är en fil
}

if (minKatalog.isDirectory()) {
  // Gör någonting om minKatalog är en katalog
}
```

För att skriva till eller läsa från den fil som File-objektet refererar anger vi den som argument till konstruktorn i klasserna FileWriter och FileReader. Därefter gör vi på precis samma sätt som tidigare exempel visat.

```
File minFil = new File("minfil.txt");
FileWriter fw = new FileWriter(minFil);
BufferedWriter filut = new BufferedWriter(fw);
// Använd objektet filut för att skriva till filen
```

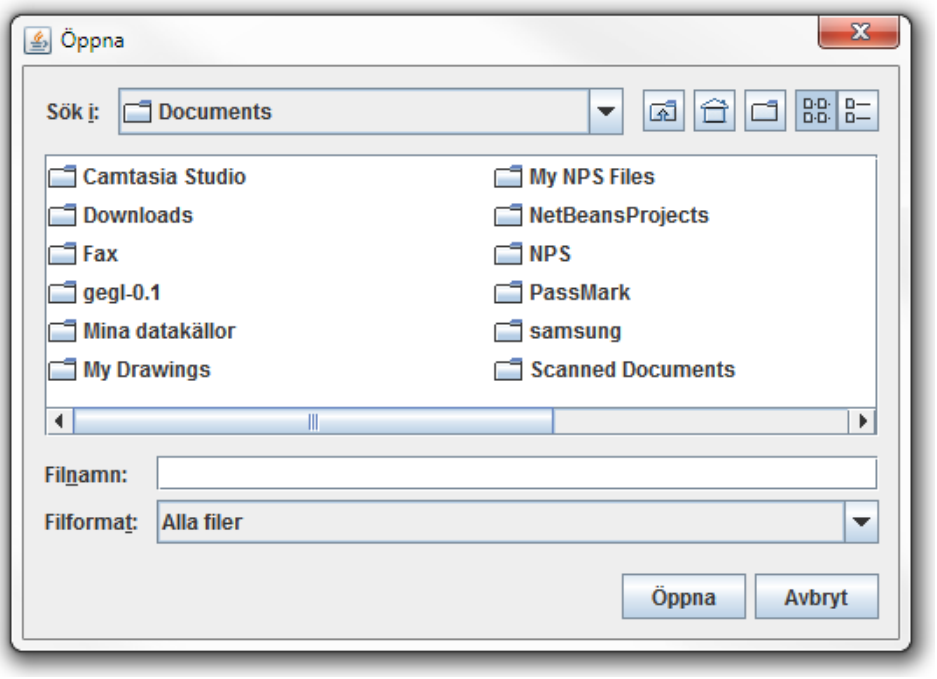
Klassen JFileChooser

I Swing finns det en färdig dialogruta som kan användas när man behöver spara och öppna filer i en applikation med ett grafiskt användargränssnitt. Med en JFileChooser kan användaren enkelt navigera runt i filsystemet och välja en eller flera filer (eller kataloger) som ska öppnas, eller välja en katalog i vilken en fil ska sparas. Notera att JFileChooser inte är en äkta dialogruta som ärver av JDialog utan i stället ärver den ifrån JComponent. JFileChooser är en väldigt flexibel klass och det är enkelt att filtrera vilka typer av filer som ska visas, vilken katalog som ska visas när filväljaren visas, om bara kataloger eller filer ska visas med mera.

För att skapa och visa en JFileChooser där användaren ska öppna en fil anropar vi metoden showOpenDialog(parent). Om vi i stället vill visa en dialogruta där användaren ska välja att spara en fil anropar vi metoden showSaveDialog(parent). Argumentet parent anger vilken förälderkomponent som dialogrutan beror av. Detta har framför allt att göra med var på skärmen dialogrutan ska visas. Vanligtvis använder vi this som argument.

För att skapa och visa en JFileChooser räcker det att skriva följande två rader med kod.

```
JFileChooser chooser = new JFileChooser();
int option = chooser.showOpenDialog(this);
```



Som standard visar dialogrutan användarens hemkatalog. För att ändra vilken katalog som ska visas kan en annan konstruktor i JFileChooser användas, alternativt att vi efter att JFileChooser är skapad, anropar metoden setCurrentDirectory.

Både metoden showOpenDialog och showSaveDialog returnerar ett heltal som säger oss om användaren tryckte på knappen Öppna/Spara eller Avbryt (eller stängde dialogrutan på annat sätt). Vi undersöker detta heltal för att avgöra vilka vidare återgärder vi ska ta. I klassen JFileChooser

finns ett par konstanter deklarerade vilka vi använder vid jämförelsen. Vanligtvis räcker det med att använda JFileChooser.APPROVE_OPTION.

```
if (option == JFileChooser.APPROVE_OPTION) {
  File file = chooser.getSelectedFile();
  // Gör något med filen, t.ex. läs innehållet i den.
}
else {
  // Användaren avbröt/stängde dialogrutan
}
```

Metoden getSelectedFile använder vi för att få en referens till den fil/katalog som valdes. Samma metod används oavsett om det är en dialogruta för att spara eller öppna som används.

Klassen FileNameExtensionFilter

En JFileChooser visar normalt alla filer och kataloger (utom dolda) i användarens filsystem. Många gånger kan det vara önskvärt att filtrera vilka filer som ska användas. Till exempel vill man kanske i en textredigerare endast visa filer med filändelsen .txt och i ett bildredigeringsprogram vill man enbart visa gilltiga bildfiler (png, jpg, bmp m.fl.). För detta använder vi klassen FileNameExtensionFilter (i paketet javax.swing.filechooser).

```
FileNameExtensionFilter txtFilter = new
  FileNameExtensionFilter("Textfiler", "txt");

FileNameExtensionFilter imgFilter = new
  FileNameExtensionFilter("Bildfiler", "jpg", "png", "bmp", "gif");
```

Konstruktorn i klassen ser ut så här:

```
FileNameExtensionFilter(String description, String... extensions)
```

Som första argumentet till konstruktorn anger vi en beskrivning över vilken/vilka typer av filer som ska visas. Därefter listar vi de filändelser som är gilltiga och som då kommer att visas i dialogrutan. Tack vare att konstruktorn som andra argument använder varargs kan ett godtyckligt antal filändelser anges.

För att lägga till filtret i en JFileChooser anropar vi metoden addChoosableFileFilter(filter).

```
JFileChooser chooser = new JFileChooser();
chooser.addChoosableFileFilter(txtFilter);
int option = chooser.showOpenDialog(this);
```

Det nya filtret läggs då till i listan över valbara filter, men väljs inte. Detta måste användaren göra själv när JFileChooser visas. Det finns en alternativ metod som både lägger till det nya filtret och väljer detta när JFileChooser visas, nämligen setFileFilter(filter).

```
JFileChooser chooser = new JFileChooser();
chooser.setFileFilter(txtFilter);
int option = chooser.showOpenDialog(this);
```

En JFileChooser har som default alltid ett filfilter som accepterar alla filtyper. När vi lägger till nya filter med någon av ovanstående metoder kan användaren ändå välja att visa alla filtyper genom att välja detta filfilter. För att förhindra detta kan vi anropa metoden setAcceptAllFileFilterUsed och

skicka false som argument. Då tas filfiltret för att visa alla filtyper bort från listan över valbara filter.

```
JFileChooser chooser = new JFileChooser();
chooser.setAcceptAllFileFilterUsed(false);
chooser.setFileFilter(txtFilter);
int option = chooser.showOpenDialog(this);
```