Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 6 – I/O, Streams and serialization

Java för C++-programmerare

Syfte:	Lektionen presenterar Javas uppbyggnad av I/O-systemet och ingående klasser för strömmar.
Läs	Skansholm avsnitt 9.1 och kapitel 16

Kataloger och filer i Java.

Kataloger och filer i Java

- En fil och katalog representeras med ett objekt av typen `java.io.File`
- Skapas genom att ange filens eller katalogens namn som en sträng

```
File minFil = new File("data.txt");
File minKatalog = new File("Mina dokument");
```

- Objektet relaterar till en fil eller katalog på datorn
- Det finns inga garantier att filen eller katalog verkligen existerar

Kataloger och filer i Java

- Finns metoder för att testa vilken typ det är och om den existerar

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

if (minFile.isFile())
  // Gör någontingom minFil är en fil

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

- Kan även ange en hel sökväg när ett `File`-objekt skapas

```
File minFil = new File("c:\\katalog1\\katalog2\\data.txt");
// Var uppmärksam på tecknet för att separera delarna i sökvägen
// Bör använda File.separator eller File.separatorChar
```

Ändra kataloger och filer

- Finns metoder som utför förändringar i filsystemet
- Metoden `renameTo` byter namn

```
File original = new File("data.txt");
File nyttNamn = new File("minaData.txt");
original.renameTo(nyttNamn);
```

- Metoden `mkdirs` skapar en katalog

```
File nyKatalog = new File("MinKatalog");
nyKatalog.mkdirs();
```

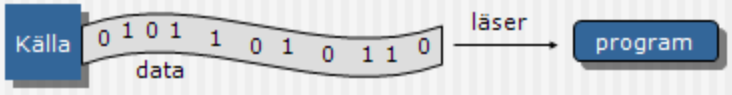
- Metoden `delete` tar bort fil/katalog

```
original.delete();
nyKatalog.delete(); // Katalogen måste var tom för att tas bort
```

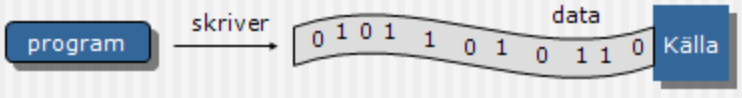
Strömmar och IO

Javaströmmar

- All in och utmatning av data i Java sker med hjälp av ”strömmar”
- Är en koppling mellan en källa och destination...
- ... tangentbordet, skärmen, filer, nätverket
- Vid inmatning öppnas en ström från en källa



- Vid utmatning öppnas en ström till en källa



All kommunikation (till/från tangentbord, filer, nätverk) sker i Java via s.k. strömmar. En ström i Java hanterar en sekvens av antingen byte eller tecken.

- teckenströmmar används för att läsa/skriva tecken (char)
- byteströmmar är tänkt att användas för att läsa/skriva binärdata.

För att läsa in information till en applikation, öppnas en ström från en informationskälla (en fil, minnet, socket etc) och läser denna information sekventiellt (tecken för tecken).
På liknande sätt kan ett program skicka information till en extern destination genom att öppna en ström till denna destination och skriva informationen sekventiellt.

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 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
- Diverse filklasser (bl.a. Klassen `File` för att representera en fil i aktuell plattform).

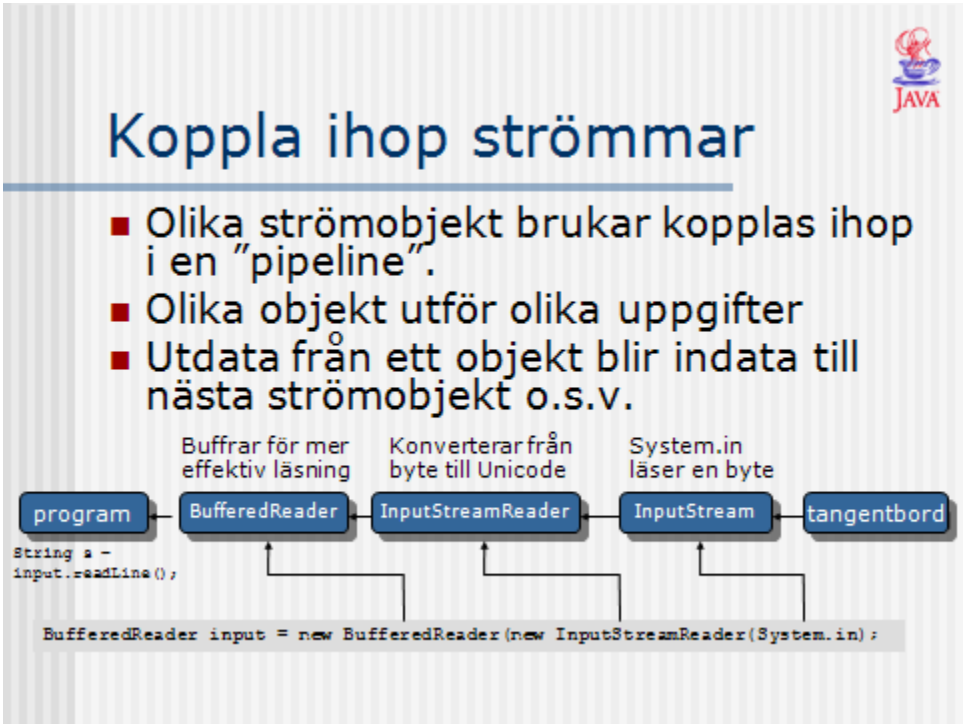
In- och utdataströmmarna delas även in i två olika typer som nämnts tidigare: **teckenströmmar** och **byteströmmar**.

- 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.

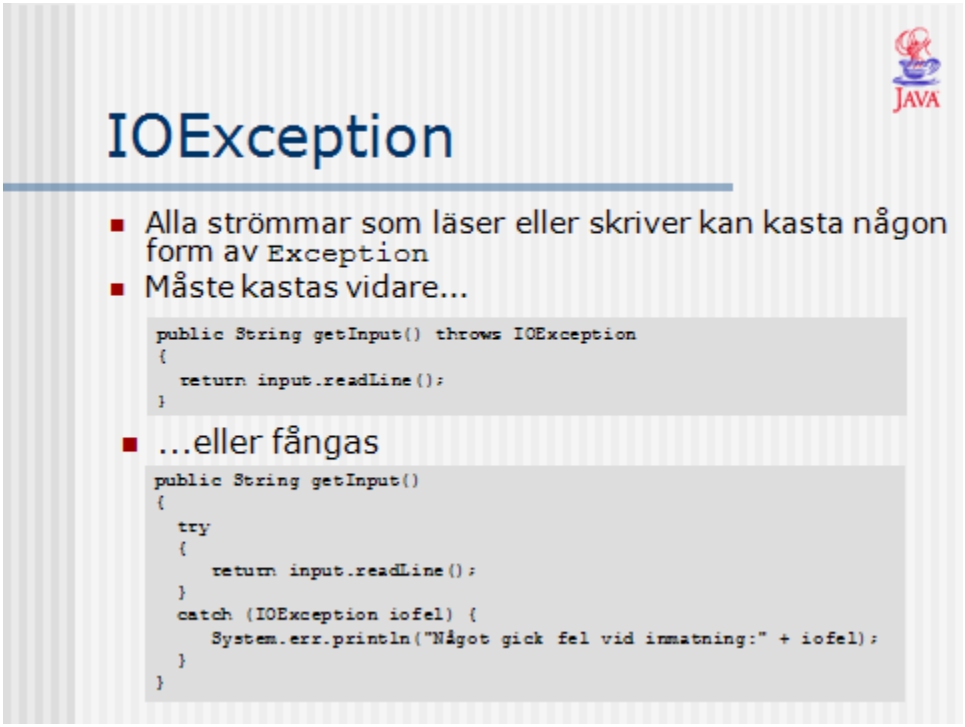
För att kunna använda en ström måste denna skapas genom att skapa ett objekt av någon av klasserna i `java.io`. Dock finns det redan tre skapade strömmar när ett Javaprogram körs. Dessa är `System.in`, `System.out` och `System.err` (in, out, err är publika statiska objekt i klassen `System`). Dessa strömmar motsvarar cin, cout och cerr i C++.

`System.in` är av typen `InputStream` och är kopplad till ”källan” tangentbordet. `System.out` och `System.err` (obuffrad) är av typen `PrintStream` är kopplade till ”destinationen” kommandofönstret.

Klassen `PrintStream` innehåller bl.a. metoderna `print` och `println` för att skriva ut olika typer av data (primitiva typer och objekt). Med metoden `read` i `InputStream` läser man en byte i taget från källan. Det är väldigt opraktiskt och inte speciellt effektivt om man i en applikation vill läsa strängar från tangentbordet (exempel **SystemIn.java**).



Det är sällan att man enbart använder en klass i `java.io` när man läser/skriver via strömmar. Normalt använder man flera strömmar som kopplas till varandra som en ”pipeline”. Detta för att låta en ström behandla data från en annan ström. Detta har vi t.ex. gjort när vi använt klassen `BufferedReader` för att läsa data från tangentbordet. Vi har då kopplat strömmen `System.in` till en ström av klassen `InputStreamReader` (som konverterar från byteström till teckenström). Denna ström kopplar vi sen till en ström av klassen `BufferedReader` som innehåller metoden `readLine` för att läsa en hel rad från tangentbordet.



Alla strömmar som läser eller skriver data kan generera ett exception om något oförutsett händer. Det kan t.ex. vara att filen vi vill läsa från inte finns eller att nätverkskopplingen till den dator vi kommunicerar med bryts. Vi måste hantera ett exception på något sätt. Enklast är att enbart kasta dem vidare från den metod felet uppstod i. Detta gör vi med ett `throws` i metoddeklarationen. Annars måste vi fånga det med `try/catch`.

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.
Vi ska dock inte direkt läsa och skriva med strömmar från dessa klasser utan koppla dem via andra strömmar.

Skriva Till En Fil

- För mer effektiv skrivning koppla `FileWriter` till en `BufferedWriter`

```
FileWriter fw = new FileWriter("filnamn");
BufferedWriter filut = new BufferedWriter(fw);
```

- Skapar alltid en ny fil

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

- Skriver enbart strängar
- Stäng alltid filen med `close()`

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. För att åstadkomma en radbrytning måste metoden `newLine` användas. Anledningen till att vi inte kan använda `\n` är att olika plattformar hanterar radbrytning på olika sätt. Om data ska skrivas fysiskt till disken direkt ska metoden `flush` anropas, annars mellanlagras data in en buffert tills filen stängs med `close`. Om du glömmer `close` kan du alltså förlora data.

Skriva Till En Fil (2)

- Koppla till en `PrintWriter` för att använda metoderna println och print

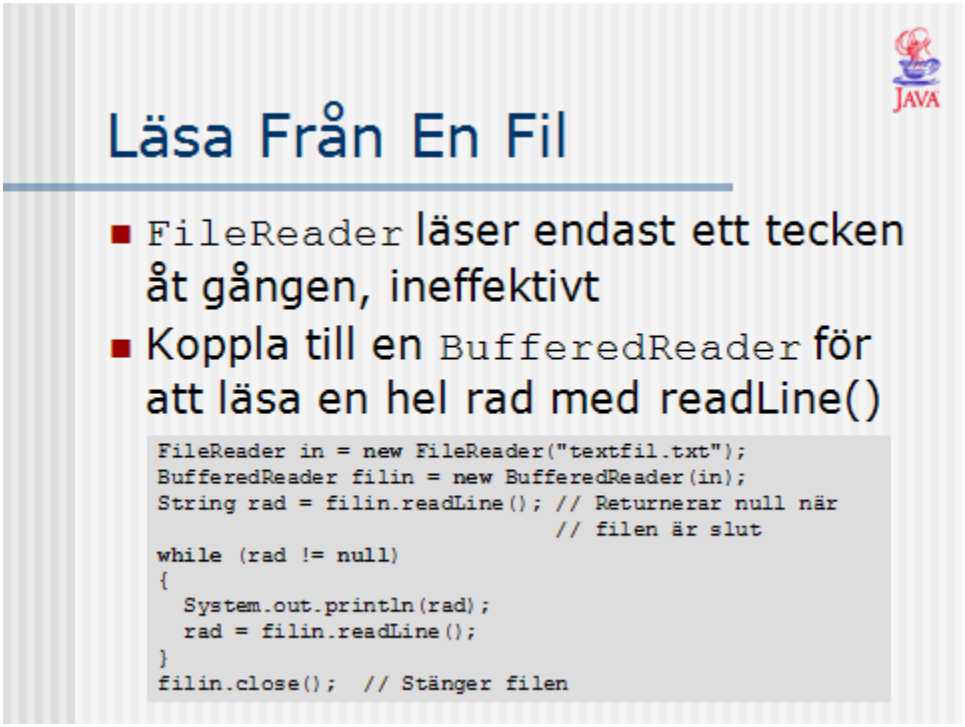
```
FileWriter fw = new FileWriter("filnamn");
BufferedWriter bw = new BufferedWriter(fw);
PrintWriter filut = new PrintWriter(bw);

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

filut.close();  // Stänger filen
```

- Kan nu även skriva primitivatyper
- Glöm inte att kasta eller fånga eventuella `Exception`

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()`. Med dessa metoder kan vi nu enkelt skriva både strängar och primitiva typer till en fil. Radbryt hanteras även korrekt oavsett vilken plattform som används.
I exemplet **SkrivaFil.java** används klasserna FileWriter, BufferedWriter och PrintWriter för att spara rader som användaren skriver in till en fil på hårddisken.



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.

Vi anger 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 returneras `null`. Detta kan vi utnyttja i en while-loop som fortsätter att snurra så länge som det returnerade värdet inte är null.

I exemplet **LäsaFil.java** skrivs innehållet i en fil ut på skärmen.

Läsa filer med `Scanner`

- `Scanner` kan användas till att läsa innehållet i textfiler
- Skapa ett `File`-objekt och koppla till Scannern

```
File minFil = new File("data.txt");
Scanner infil = new Scanner(minFil);
// FileNotFoundException om filen inte finns
```

- `nextInt`, `nextDouble`, `nextLine` etc för att läsa olika typer av innehåll

Läsa filer med `Scanner`(2)

- Innehållet i en textfil kan vara organiserad t.ex. så här:

```
1 Tyskland 9 8 5
2 Österrike 8 6 5
3 USA 7 7 4
4 Ryssland 7 3 6
```

```
1;Tyskland;9;8;5
2;Österrike;8;6;5
3;USA;7;7;4
4;Ryssland;7;3;6
```

- Använd `useDelimiter` för att ange hur de olika delarna separeras

```
File minFil = new File("medaljligan.txt");
Scanner infil = new Scanner(minFil);
infil.useDelimiter(";");
```

Läsa filer med `Scanner(3)`

- Därefter kan de olika delarna enkelt läsas in:

```
// Varje rad är uppdelad i int;String;int;int;int
int placering = rad.nextInt();
String land = rad.next();
int guld = rad.nextInt();
int silver = rad.nextInt();
int brons = rad.nextInt();
```

Se exemplet **Medaljligan.java**.

Binära filer

- Fil som innehåller binära representationer av olika uppgifter
- Kan inte läsas i vanliga texteditorer
- Kan hanteras på två olika sätt
  - Som en sekventiell fil där data läses/skrivs från början till slutet
  - Som en fil med direkt åtkomst där läsning/skrivning kan ske på valfri plats

Läsa och skriva binära filer

- Att skriva binär data till fil görs med `FileOutputStream`

```
FileOutputStream(File file)
FileOutputStream(File file, boolean append)
FileOutputStream(String name)
FileOutputStream(String name, boolean append)
```

- Att läsa binär data från fil görs med `FileInputStream`

```
FileInputStream(File file)
FileInputStream(String name)
```

- Endast läsa/skriva `byte`

Skriva primitiva typer

- För utmatning av primitiva typer till binärfil används `DataOutputStream`

```
Person p = new Person("Kalle", "Svensson", 22);
DataOutputStream ut = new DataOutputStream(new
  FileOutputStream("personer.dat"));

ut.writeUTF(p.hämtaFörnamn());
ut.writeUTF(p.hämtaEfternamn());
ut.writeInt(p.hämtaÅlder());
int antalBytes = ut.size();  // Antal skrivna byte
ut.close();
```

- Bör inte använda UTF för strängar
- Kan använda `BufferedOutputStream`

Läsa primitiva typer

- Inmatning av primitiva typer från binärfil görs med `DataInputStream`

```
DataInputStream in = new DataInputStream(new
  FileInputStream("personer.dat"));

String f = in.readUTF();
String e = in.readUTF();
int å = in.readInt();
in.close();

Person p = new Person(f, n, å);
```

- Viktigt att läsa data i rätt ordning

Se exempel **Person.java**, **SparaPersoner.java** och **LaddaPersoner.java**.

Läsa/Skriva Objekt Till Fil

- Serialisering är en teknik för att enkelt och smidigt spara och hämta hela objekt till en ström
- Gör det möjligt att lagra objektets alla data direkt till en fil
- Eller att skicka objektet via en kommunikationsförbindelse

Serialisering

- Objekt som ska stödja serialisering måste implementera gränssnittet `Serializable`
- Detta gränssnitt är tomt så inget behöver implementeras i klassen
- Klasserna som används är:
  - `ObjectOutputStream`
  - `ObjectInputStream`

Serialisering (2)

- Dessa strömmar kopplas till `FileOutputStream` respektive `FileInputStream`
- Spara och läsa objekt görs sen genom att anropa metoderna `writeObject` och `readObject`

```
FileOutputStream fos = new FileOutputStream("filnamn");
ObjectOutputStream ut = new ObjectOutputStream(fos);

ut.writeObject(new Person("Kalle", "Svensson", 22));
ut.close();
```

Serialisering (3)

- `readObject` returnerar en referens till klassen `Object`
- Det är därför nödvändigt med en explicit typomvandling

```
FileInputStream fis = new FileInputStream("filnamn");
ObjectInputStream in = new ObjectInputStream(fis);

Person p = (Person)in.readObject();
```

- För att kunna läsa in objektet måste objektets klass vara tillgänglig om inte genereras `ClassNotFoundException`

Se exempel **Person.java**, **SparaPersonObjekt.java** och **LaddaPersonObjekt.java**.

SequenceInputStream

- Används för att koppla ihop flera strömmar (`InputStream`) till en

```
FileInputStream f1 = new FileInputStream("data1.dat");
FileInputStream f2 = new FileInputStream("data2.dat");
SequenceInputStream sis = new SequenceInputStream(f1, f2);
```

- Läser först all data från `f1` och därefter från `f2` innan `sis` indikerar att slutet har nåtts (end of file)
- Fungerar för alla subklasser till `InputStream`

SequenceInputStream (2)

- Kan slå ihop fler än två inströmmar om en befintlig `SequenceInputStream` anges till konstruktorn

```
FileInputStream f1 = new FileInputStream("data1.dat");
FileInputStream f2 = new FileInputStream("data2.dat");
FileInputStream f3 = new FileInputStream("data3.dat");
SequenceInputStream sis1 = new SequenceInputStream(f1, f2);
SequenceInputStream sis2 = new SequenceInputStream(sis1, f3);
```

- Kan även använda en `Enumeration`

```
Vector<InputStream> v = new Vector<InputStream>();
v.add(f1);
v.add(f2);
v.add(f3);
Enumeration<InputStream> e = v.elements();
SequenceInputStream sis = new SequenceInputStream(e);
```

SequenceInputStream (3)

- Användbara metoder:

```
SequenceInputStream sis = new SequenceInputStream(e);

// Antal byte tillgänglig att läsas från strömmen
int i = sis.available();

// Läser nästa byte från strömmen och returnerar en byte
// som int mellan 0 - 255 eller -1 om slut på strömmen
int b = sis.read();

// Läser i antal byte från strömmen till byte-arrayen b
// med början på position 0. Returnerar antal lästa byte.
byte[] b = new byte[i];
int a = sis.read(b, 0, i);

// Stänger strömmen
sis.close();
```

Exempel

- Klass som sparar innehållet i en eller flera textfiler till en enda fil

OBS!
Felhanteringen är borttagen

```
public class FileConcatenator {
  private File[] files;

  public FileConcatenator(File[] files) {
    this.files = files;
  }

  public void concatenate(File file) {
    if (files == null || files.length == 0)
      return;

    Vector<FileInputStream> v =
      new Vector<FileInputStream>();

    for (File f : files) {
      v.add(new FileInputStream(f));
    }
    Enumeration<FileInputStream> e = v.elements();
    SequenceInputStream in = new SequenceInputStream(e);
    FileOutputStream out = new FileOutputStream(file);
    int c;
    while ((c = in.read()) != -1) {
      out.write(c);
    }
    in.close();
    out.close();
  }
}
```

Som ett exempel på användning av SequenceInputStream skriver vi en klass som sparar innehållet i en eller flera filer till en ny fil. Vi kallar klassen för FileConcatenator. Som enda instansvariabel i klassen har vi en array av klassen File. Denna array innehåller de filer vi ska slå ihop till en. För att skapa ett objekt av klassen måste man som argument till konstruktorn skicka en array med de filer som ska slås ihop.
För att spara innehållet i filerna till en enda ny fil används metoden `concatenate`. För att enkelt kunna skapa en SequenceInputStream med flera strömmar använder vi en Vector och metoden `elements` i denna. Vi skapar därför en Vector för att rymma objekt av klassen FileInputStream. Därefter loopar vi igenom alla element i arrayen och skapar ett ny FileInputStream av filen som vi sen lägger till i vektorn. När detta är gjort anropar vi metoden `elements` för att erhålla ett Enumeration-objekt. Detta objekt använder vi när vi sen skapar vår SequenceInputStream. Vi skapar även en FileOutputStream som används för att spara filernas innehåll till.
Till sist läser vi tecken för tecken från vår SequenceInputStream med metoden `read()`. Dessa skriver vi sen till den nya filen med `write`. När vi är klara stänger vi båda strömmarna med metoden `close`.
Bland exemplen hittar du **FileConcatenator.java**, **TextFileFilter.java** och **Concatenate.java**. Dessa klasser demonstrerar användandet av FileConcatenator där användaren med hjälp av en JFileChooser får välja vilka filer som ska slås ihop och var den nya filen ska sparas.

Filtrerande Strömmar

- Dessa klasser i Java kan användas för att ”filtrera” data som läses från eller skrives till strömmen:
  - `FilterReader`
  - `FilterInputStream`
  - `FilterWriter`
  - `FilterOutputStream`
- Konkreta klasser – kan användas direkt.

Filtrerande Strömmar (2)

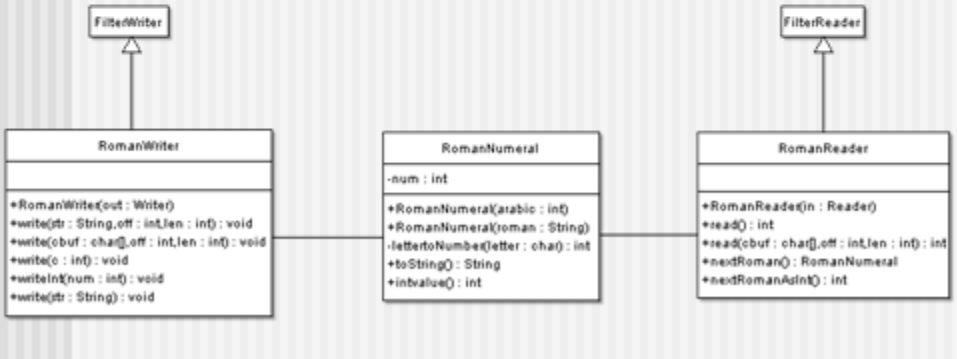
- En filtrerande ström kopplas på efter eller före en annan befintlig ström
- Filtrerande inströmmar läser data från underliggande strömmen, filtrerar och skickar vidare data
- Filtrerande utströmmar filtrerar data och skriver data till underliggande ström

Filtrerande Utströmmar

- `FilterOutputStream` och `FilterWriter` har en konstruktor
- Tar en utström (`OutputStream` resp. `Writer`) som parameter
- I subklasser bör de olika varianterna av metoden `write` omdefinieras så att data ”filtreras” innan skrivning

Exempel RomanWriter

- Klass som filtrerar data från ett heltal till dess romerska värde innan det romerska värdet sen skrivs till utströmmen



För att visa ett exempel på hur filtrerade in- och utströmmar kan användas kan vi studera ett program som läser heltal, konverterar talet till dess romerska värde och skriver det romerska värdet till en utström.
Konverteringen eller filtreringen sköts av två klasser RomanReader och RomanWriter som ärver FilterReader respektive FilterWriter. Dessa klasser använder i sin tur klassen RomanNumeral som utför själva konverteringen av heltal (1 – 3999) till romerskt tal (och tvärtom).
Se detaljer i **RomanNumeral.java**, **RomanWriter.java**, **RomanReader.java** och **TestAvRoman.java**

URL är en klass som ligger i paketet java.net. Med denna klass kan vi relativt enkelt skapa en ström från en fil på nätet. Ett objekt av klassen URL pekar ut en viss fil på nätet. För att skapa ett URL-objekt anger vi adressen till filen som ett argument till konstruktorn. Adressen består av flera delar där man först måste ange vilket protokoll som ska användas (http, ftp, file etc). Därefter följer en sökväg till filen. Ett eventuellt MalformedURLException måste antingen kastas vidare eller fångas upp.

Via URL-objektet kan man sen väldigt enkelt öppna en ström för att läsa filens innehåll. Detta kan göras på två sätt där ena sättet där ett sätt är att anropa metoden `openStream`. Denna metod returnerar en ström av klassen InputStream. Detta är samma typ av ström som System.in men nu kopplad till en fil på nätet istället för till tangentbordet.


Läsa och Skriva ZIP-filer
I paketet java.util.zip finns klasser för att bl.a:
Läsa innehållet i ZIP- och GZIP-filer
Packa filer till ZIP eller GZIP
Packa upp filer från ZIP eller GZIP
Med ZipInputStream kopplad till en FileInputStream läser vi från ZIP
Med ZipOutputStream kopplad till en FileOutputStream skriver vi


Klassen ZipFile
Används för att läsa innehållet i en ZIP-fil och kan skapas enligt:
ZipFile zip1 = new ZipFile("filer.zip");
ZipFile zip2 = new ZipFile(new File("filer.zip"));
Kaster ett ZipException om en icke giltig ZIP-fil anges
Användbara metoder:
String path = zip2.getName(); // Sökväg till ZIP-filen
int a = zip2.size(); // Antal filer i ZIP-fil
Enumeration e = zip2.entries(); // Alla filer i ZIP-filen
InputStream is = zip2.getInputStream(// Inström från fil
(ZipEntry)e.nextElement());

Klassen ZipEntry

- Representerar en fil i ZIP-filen
- Användbara metoder:

`String getComment()`	Returnerar eventuell kommentar för filen, eller null om ingen finns.
`long getCompressedSize()`	Returnerar filens komprimerade storlek eller -1 om inte känd.
`long getCrc()`	Returnerar en CRC-32 kontrollsumma för den okomprimerade filen eller -1 om inte känd.
`byte[] getExtra()`	Returns the extra field data for the entry, or null if none.
`int getMethod()`	Returnerar vilken komprimeringsmetod som använts eller -1 om ingen komprimeringsmetod specificerats.
`String getName()`	Returnerar filens namn.
`long getSize()`	Returnerar filens okomprimerade storlek eller .1 om inte känd.
`long getTime()`	Returnerar tidpunkt när senaste förändringen gjordes eller -1 om inte känd.
`boolean isDirectory()`	Returnerar true om det är en katalog.

Lista innehållet i en ZIP-fil

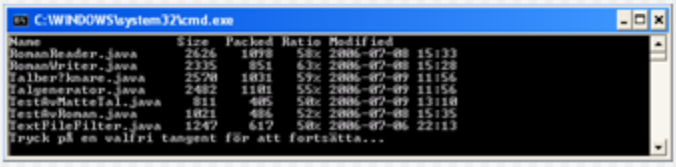
```
ZipFile zip = new ZipFile("källkod.zip");
Enumeration e = zip.entries();

while(e.hasMoreElements()) {
  ZipEntry entry = (ZipEntry)e.nextElement();

  String filename = entry.getName();
  long size = entry.getSize();
  long compressedSize = entry.getCompressedSize();

  System.out.format("%-20s %5s %6s\n", filename, size, compressedSize);
}

zip.close();
```



Att lista innehållet i en ZIP-fil är väldigt enkelt.

- Skapa ett objekt av klassen ZipFile som kopplas till en giltig zip-fil. Observera att ett ZipException kan kastas om angiven zip-fil inte finns eller är ogiltig.
- Anropa vi metoden entries() för att erhålla en Enumeration över alla filer i zip-filen.
- Loopa igenom alla filer vilket vi gör så länge som metoden hasMoreElements() returnerar true. I loopen plockar vi ut nästa ZipEntry och via denna kan vi hämta de data om filen som är av intresse.

I bilden ovan hämtar vi namn på filen, dess ursprungliga storlek och dess komprimerade storlek som vi sen skriver ut på skärmen.
När alla filer har hanterats stänger vi zip-filen med metoden `close()`.
Se exemplet **ReadZIP.java** .

Packa filer (2)

- Skapa en `ZipEntry` för filen som ska packas och lägg till den i ZIP-filen

```
ZipEntry entry = new ZipEntry(filename);
out.putNextEntry(entry);
```

- Överför data från filen till ZIP-filen och stäng strömmarna

```
int c;
while ((c = in.read()) > 0) {
  out.write(c);
}
out.closeEntry(); // Stäng nuvarande ZipEntry
out.close();      // Stäng ZIP-filen
in.close();       // Stäng filens inström
```

Packa upp filer

- Erhåll en `InputStream` för aktuell `ZipEntry` genom att anropa `getInputStream`

```
BufferedInputStream in = new
  BufferedInputStream(zip.getInputStream(entry));
```

- Ta reda på namnet och skapa en utström för att spara

```
String filename = entry.getName();
BufferedOutputStream out = new BufferedOutputStream(
  new FileOutputStream(filename));
```

Packa upp filer (2)

- Överför data från ZIP-filen till filen och stäng strömmarna

```
int c;
while ((c = in.read()) != -1)
{
  out.write(c);
}

in.close();
out.close();
```

- Tänk på att hantera filens destinationskatalog korrekt

Se exemplen **ZIP.java** och **TestAvZIP.java.** Klassen ZIP innehåller statiska klassmetoder för att packa en eller flera filer till en angiven zip-fil och/eller packa upp en zip-fil till en angiven katalog på hårddisken.