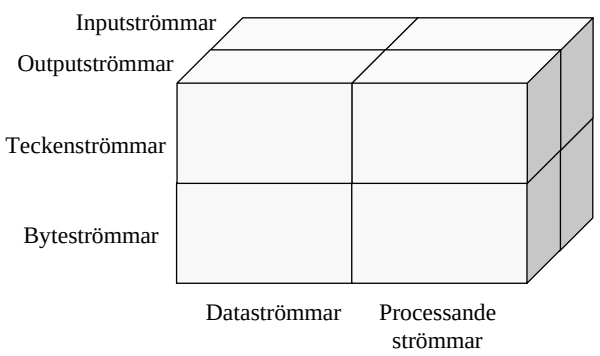
Strömmar igen...



- Strömmar är en ordnad sekvens av bytes.
- Huvudtyperna är *Inputströmmar* som man läser information ifrån och *Outputströmmar* som man skriver information till.
  - Dessa kan sedan i sin tur delas upp i mindre delar.

Strömmar igen...

- Teckenströmmar vs byteströmmar
  - Indelning utifrån vilken typ av data som behandlas
  - Teckenströmmar hanterar 16-bits Unicode tecken
    - InputStream, OutputStream och alla dess barn hanterar teckenströmmar.
  - Byteströmmar hanterar 8-bitars binärt data (0-1)
    - Reader, Writer och deras barn hanterar byteströmmar.

Strömmar igen...

- Dataströmmar vs processande strömmar
  - Indelning utifrån vilken roll strömmen spelar
  - Dataströmmar representerar en speciell källa eller målström, (tex en sträng i minnet eller en fil på hårddisken)
  - Processande strömmar gör någon form av manipulation av datat i en ström, tex konverterar mellan olika format eller buffrar
  - Bara ett annat sätt att kategorisera tecken- och bytesströmmar men alla fyra subklasserna kan delas upp i data eller processande strömmar.

Se myShapes!

Serialisering

- Objektserialisering (Object serialization) är att spara ett objekt och dess tillstånd så att det kan användas senare i tex ett annat program
- Objektserialisering sker med hjälp av klasserna `ObjectOutputStream` och `ObjectInputStream`
- Serialisering tar hänsyn till eventuella andra objekt som refereras av objektet som serialiseras och sparar dem också.
- Klassen och alla de objekt som refereras måste implementera någon av gränssnitten `Serializable` eller `Externalizable`
- `writeObject` används för att serialisera ett objekt
- `readObject` används för att “avserialisera” ett objekt

Serialisering

- Nackdel: Om man skapar en fil med serialiserade objekt och sen ändrar koden för klassen så kan man inte läsa filen senare...
  - Se till att ha utvecklat klassen klart innan man sparar ned objekt på fil.
- All information som finns lagrad i klassen skrivs ned på filen.
  - Även känslig information som tex lösenord
  - Ibland är det bara vissa delar av ett objekt som är viktigt att spara undan (dvs man vill spara minne)
- Om vi vill utesluta något så använder man `transient`-modifieraren.
  - `private transient boolean isVisible;`

Innehåll

- OOP snabbintroduktion
- Datatyper
- Uttryck
- Satser
- Arv (intro)
- Att organisera Javakod
- Klassdesign och metodik (UML, CRC)
- Arv, polymorfi och dynamisk bindning
- Fält
- Undantag
- In-/utmatning och filer
- Applets vs applikationer
- Rekursion

Felhantering

- Vissa fel kan inte förebyggas med hjälp av if-satser
  - En fil kan t ex raderas under skrivningen/läsningen, fast man har kollat i förväg att allt är OK
- Möjliga åtgärder:
  - Avbryt eller avsluta programmet
  - Ignorera felet och fortsätt
  - Skriv ut ett meddelande och fortsätt
  - Ignorera anropet som ledde till felet
  - Tolka anropet på ett meningsfullt sätt
  - Kräva en åtgärd från användaren interaktivt
  - Producera ett felaktigt resultat
  - Returnera en felkod
  - ...

Undantag

- En ***exception*** är ett objekt som signalerar ett undantag
- Exceptions kastas (are *thrown*) i ett program och sedan kan man
  - Ignorera undantaget
  - Fånga upp och hantera problemet där det uppkommer
  - Fånga upp och hantera problemet i en annan del av programmet (*propagering* av undantag).
- Vad är bäst att göra?
  - Viktig designfråga!
  - Ingen åtgärd är bäst i alla lägen utan det...
  - ... beror på anroparens mål

➔Låt anroparen bestämma

Undantag

- Ett undantag som kastas automatiskt (anropa med x = 0):

```
public int myDivision(int y, int x)
{
        return y/x;
}
```

Ger meddelandet

```
Aritmethic exception: / by zero
```

tillsammans med en beskrivning av vart felet uppkommit.

`throw`-satsen kastar ett undantag

- Kasta ett undantag:

```
public int myDivision(int y, int x)
{
  if (x==0)
  {
      throw new ArithmeticException(“/0 i metoden myDivision");
  }
  return y/x;
}
```

- Man kan också definiera egna (nya) undantag

Undantag

- … kan fångas m h a ett `try`-block
  - Kodraden som kastar ett undantag körs inuti ett try-block
- Ett try-block följs av ett eller flera catch-block som hanterar ett undantag
  - Varje catch-block är associerad till en undantagstyp och kallas för undantagshanterare - *exception handler*

```
try
{…}
catch (…)
{…}
finally
{…}
```

Satser där ett undantag kan kastas

Undantaget som fångas

Koden där ett undantag hanteras

Flera undantag kan fångas och hanteras i en `try`-sats (optional)

Koden för slutåtgärder (optional)

Exempel

```
HouseInfo hi = new HouseInfo();
String fileName = ”houses.dat”;

// Could put this in a loop that tried a new
// filename until success...
try
{
    hi.printToFile(fileName);
}
catch (FileNotFoundException e)
{
    System.err.println(”The file ” + fileName + ” was not found”);
}
catch (IOException e)
{
    System.err.println(”Error reading from file ” + fileName);
}
```

`finally`-blocket

- `finally`-blocket är frivilligt
- Alla satser i blocket körs oavsett resultatet i `try`-blocket.
- Satserna körs efter `try`- eller `catch`-blocket (beroende på vilket av dessa som är aktuellt)

Man kan definiera egna undtantag (se sidorna 459-460)

- Man deklarerar egna undantag genom att ärva från klassen `Exception`.

```
public class OutOfRangeException extends Exception
{
   OutOfRangeException (String message)
   {
      super (message);
   }
}
```


Man kan definiera egna undtantag (se sidorna 459-460)

```
import cs1.Keyboard;

public class CreatingExceptions
{
   public static void main (String[] args) throws OutOfRangeException
   {
      final int MIN = 25, MAX = 40;

      OutOfRangeException problem =
         new OutOfRangeException ("Input value is out of range.");

      System.out.print ("Enter an integer value between " + MIN +
                        " and " + MAX + ", inclusive: ");
      int value = Keyboard.readInt();

      //  Determines if the exception should be thrown
      if (value < MIN || value > MAX)
         throw problem;

      System.out.println ("End of main method.");  // may never reach
   }
}
```


Två kategorier av undantag

- ***Checked exceptions***
  - Måste fångas (dvs kod som kan kasta en checked exception måste skrivas i en `try`-sats) eller propageras vidare uppåt (vilket syns i metodhuvudet ”`throws ...`”)
  - Kompilatorn ger felmeddelande om man inte gör något av alternativen ovan.
- ***Unchecked exceptions***
  - Får ignoreras (men det är god programmeringsstil att fånga eller deklarera det)
  - De enda unchecked exceptions som finns i Java är ”RuntimeException” och de som ärver från den klassen.
- ***Fel (errors)***
  - Liknar RuntimeExceptions och dess arvingar
  - Ska inte fångas och kräver ingen ”`throw`”-sats