Föreläsning 12

Inre klasser
Anonyma klasser
Kloning
I/O-ramverket

Nästlade klasser

En nästlad klass är en klass som är definierad i en annan klass. Det finns fyra olika slag av nästlade klasser:

- statiska medlemsklasser
- icke-statiska medlemsklasser
- lokala klasser
- anonyma klasser

Nästlade klasser, förutom statiska medlemsklasser kallas också för *inre klasser*.

Motiv för användning av nästlade klasser:

- ett sätt att logiskt gruppera klasser som endast används på ett ställe
- förstärker inkapsligen
- kan leda till kod som är lättare att läsa och underhålla.

Statiska medlemsklasser

- En statisk medlemsklass kan använda alla statiska variabler och statiska metoder i den omgivande klassen (även privata)
- Medlemmar i den omgivande klassen har åtkomst till alla medlemmar i den statiska medlemsklassen.

```
public class SillyClass {
   private static int times = 10;
   . . .
   public static int minus(int val, int val2){
      return val - val2;
   }//minus
   public static class Helper {
      public static int oper(int val, int val2) {
         return val + val2 + minus(times*val2, val);
      }//oper
   }//Helper
   . . .
}//SillyClass
```

class-filen för medlemsklassen får namnet SillyClass$Helper.class

```
public class Main {
   public static void main(String[] args) {
      System.out.println(SillyClass.Helper.oper(1, 2));
   }//main
}//Main
```

Inre klasser

- En icke-statisk medlemsklass kallas för *inre klass*.
- En inre klass kan använda alla attribut och metoder i den omgivande klassen (även privata).
- Medlemmar i den omgivande klassen har åtkomst till alla medlemmar i den inre klassen (via instans av den inre klassen).
- Om den inre klassen deklareras som **private** så innebär det att det inte går att skaffa sig instanser av klassen utanför den omgivande klassen.
- En metod i den yttre klassen kan returnera en instans av den inre. Detta är användbart då den inre klassen implementerar ett gränssnitt (t.ex. i samband med händelsehantering och trådar).

Inre klasser

```
import java.util.*;
public class SimpleList<E> implements Iterable<E> {
   private Node first;
    public Iterator<E> iterator() {
       return new OurIterator();
   } //iterator
   . . .
```

```
private class OurIterator implements Iterator <E> {
      private Node position;
      private Node previous;
      public OurIterator() {
         position = null;
         previous = null;
      }//constructor
      public boolean hasNext() {
         . . .
      }//hasNext
      public E next() {
         . . .
      }//next
      public void remove () {
         throw new UnsupportedOperationException();
      }//remove
   } //OurIterator
}//SimpleList
```

Lokala klasser

Inre klass som deklareras i ett block (d.v.s. inom { och }).

- Endast synlig i blocket – analogt med en lokal variabel.
- Kan använda attribut och metoder i omgivande klass.
- Deklareras vanligtvis i metoder.
- Kan använda alla variabler och metodparametrar som är deklarerade **final** inom deklarationsblocket för den lokala klassen.

Lokala klasser

```
public class LocalClassExample {
  private int value  = 9000;
   public void aMethod(final int number) {
      class Local {
         public void printStuff() {
            System.out.println(value);
            System.out.println(number);
         }
      }//Local
      Local local = new Local();
      local.printStuff();
   }//amethod

   public static void main(String[] args) {
      new LocalClassExample().aMethod(20);
   }
}//LocalClassExample
```

Okey, eftersom number är **final**

Anonyma klasser

En entitet är anonym om den inte har något namn. Anonyma entiteter används ofta i ett program

```
add.data(new Integer(123));
```

I Java är det är möjligt att definiera anonyma klasser:

```
Comparator<Country> comp = new Comparator<Country>() {
   //anonym klass
   public int compare(Country1 c1, Country c2) {
      return c1.getName().compareTo(c2.getName());
   }
};
```

Ovanstående är likvärdigt med att skriva:

```
public class MyComparator implements Comparator<Country> {
   public int compare(Country c1, Country c2) {
      return c1.getName().compareTo(c2.getName());
   }
}
. . .
Comparator<Country> comp = new MyComparator();
```

Mer om anonyma klasser

Anonyma klasser används ofta som ”throw away”-klasser när man endast behöver *ett* objekt av en klass.

Men genom att lägga den anonyma klassen i en metod kan man enkelt skapa multipla objekt av en anonym klass.

Antag att vi i klassen Country vill ha en klassmetod som returnerar ett Comparator-objekt för att jämföra namnen på länder. Detta kan åstadkommas enligt följande:

```
public class Country {
   . . .
   public static Comparator<Country> comparatorByName() {
      return new Comparator<Country>()
      {
         public int compare(Country c1, Country c2) {
            return c1.getName().compareTo(c2.getName());
         }//compare
      };
   }//comparatorByName
   . . .
}//Country
```

Nästlade interface och inre interface

Ett nästlat interface är ett interface som deklareras inuti ett annat interface.

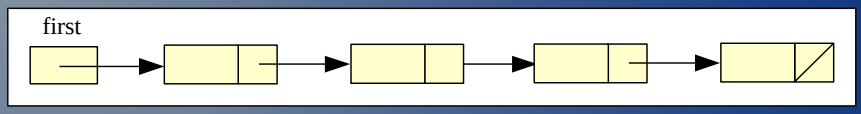
```
public interface OuterInterface {
   public void aMethod();
   public void anOtherMethod();

   public static interface NestedInterface {
      public void aThirdMethod();
   }//NestedInterface
}//OuterInterface
```

En klass kan ha inre interface, d.v.s. interface som deklareras i klassen.

Länkade listor

En länkad lista består av ett antal *noder*. Varje nod innehåller dels en referens till nästa nod, dels en referens till ett element i listan.



Vi skall göra en implementation av en enkel generisk länkad lista. Specifikationen är enligt följande:

```
//skapar en tom lista
public SimpleList()

//returnerar första elementet i listan
//kastar NoSuchElementException om inget sådant element finns
public E getFirst()

//tar bort första elementet i listan
//kastar NoSuchElementException om inget sådant element finns
public void removeFirst()

//lägger in ett element först i listan
public void addFirst(E element)

//returnerar en iterator för listan
public Iterator<E> iterator()
```

Länkad lista – implementation

Vi använder en privat inre klass för att handha noderna, och en privat inre klass för att implementera iteratorn för vår lista.

```
public class SimpleList<E> implements Iterable<E> {
   private Node first;
   . . .
   private class Node {
      private E data;
      private Node next;
   }//Node
   private class OurIterator implements Iterator <E>{
      . . .
      public OurIterator() { . . . }
      public boolean hasNext() { . . . }
      public E next() { . . . }
      public void remove () { . . . }
   }//OurIterator
     . . .
}//SimpleList
```

Implementation – klassen SimpleList

```
import java.util.*;
public class SimpleList<E> implements Iterable<E> {
   private Node first;
   public SimpleList() {
      first = null;
   } //constructor

   public E getFirst() {
      if (first == null)
         throw new NoSuchElementException();
      return first.data;
   } //getFirst

   public void removeFirst() {
      if (first == null)
         throw new NoSuchElementException();
      first = first.next;
   } //removeFirst
```

```
   public void addFirst(E element) {
      Node newNode = new Node();
      newNode.data = element;
      newNode.next = first;
      first = newNode;
   } //addFirst

   public Iterator<E> iterator() {
      return new OurIterator();
   } //iterator

   private class Node {
      private E data;
      private Node next;
   } //Node
```

Implementation – OurIterator

```
private class OurIterator implements Iterator <E> {
   private Node position;

   public OurIterator() {
      position = null;
   }//constructor

   public boolean hasNext() {
      if (position == null)
         return first != null;
      else
         return position.next != null;
   }//hasNext
```

```
   public E next() {
      if (!hasNext())
         throw new NoSuchElementException();
      if (position == null)
         position = first;
      else
         position = position.next;
      return position.data;
   }//next

   public void remove () {
      throw new UnsupportedOperationException();
   }//remove
  } //OurIterator
}//SimpleList
```

Metoden clone

Metoden clone() används för att skapa en kopia av ett existerande objekt.

Klassen Object definierar en standardimplementering av clone():

```
protected Object clone() throws CloneNotSupportedException {
   if (this instanceof Cloneable) {
      //Copy the instance fields
      . . .
   }
   else
      throw new CloneNotSupportedException();
}//clone
```

Vi ser att metoden clone() är **protected** och att ett undantag av typen CloneNotSupportedException kastas om metoden anropas för ett objekt av en klass som inte implementerar gränssnittet Cloneable.

Metoden clone

En klass vars objekt skall gå att klona måste implementera gränssnittet Cloneable och överskugga metoden clone().

Det rekommenderas att överskuggade metoder av clone skall uppfylla följande villkor:

x.clone() != x

x.clone().equals(x)

x.clone().getClass() == x.getClass()

Grund kopiering (*shallow copy*)

Antag att vi vill kunna skapa kopior av objekt av klassen SomeClass nedan:

```
public class SomeClass {
   private int value;
   private boolean  status;
   private String str;
   public SomeClass(int value, boolean status, String str) {
      this.value = value;
      this.status = status;
      this.str = str;
   }//constructor
    . . .
}//SomeClass
```

Vi måste således låta klassen implementera interfacet Cloneable och överskugga metoden clone().

Vill man att clone() skall vara allmänt tillgänglig skall clone() deklareras som **public**.

Grund kopiering (*shallow copy*)

Metoden clone i klassen Object, skapar och returnerar en kopia av det aktuella objektet. Varje instansvariabel i kopian har *samma värde* som motsvarande instansvariabeln i originalet.

I klassen SomeClass är instansvariablerna value och status primitiva variabler, och instansvariabeln str är ett icke-muterbart objekt. Därför gör clone() i klassen Object allt som är nödvändigt för att skapa en kopia.

```
public class SomeClass implements Cloneable {
   private int value;
   private boolean  status;
   private String str;
   public SomeClass(int value, boolean status, String str) {
      this.value = value;
      this.status = status;
      this.str = str;
   }//constructor

   @Override
   public SomeClass clone() {
      try {
         return (SomeClass) super.clone();
      }
      catch (CloneNotSupportedException e) {
         return null; //never invoked
      }
   }//clone
}//SomeClass
```

kovariant returtyp

Grund kopiering (*shallow copy*)

```
public class SomeClass {
   private int value;
   private boolean  status;
   private String str;
   public SomeClass(int value, boolean status, String str) {
      this.value = value;
      this.status = status;
      this.str = str;
   }//constructor
   @Override
   public SomeClass clone() {
      try {
         return (SomeClass) super.clone();
      }
      catch (CloneNotSupportedException e) {
         return null; //never invoked
      }
   }//clone
} //SomeClass
```

Vi har glömt att implementera Cloneable!
Vad händer?

```
. . .
SomeClass master = new SomeClass(5, true, "Hello ");
SomeClass copy = master.clone();
```

Vilket värde har copy?

Djup kopiering (*deep copy*)

Grund kopiering är trivial, eftersom det bara är att nyttja metoden clone() i klassen Object.

Grund kopiering fungerar då attributen för objektet som klonas utgörs av *primitiva typer och icke-muterbara typer*. För primitiva typer skapas kopior och för icke-muterbara typer spelar det ingen roll att de delas, eftersom de inte kan förändras.

Om ett objekt har muterbara referenser måste man använda djup kopiering (*deep copy*) för att klona objektet. Djup kopiering innebär att man skapar *kopior av de refererade objekten*. Vilket kan vara mycket komplicerat.

Djup kopiering (*deep copy*)

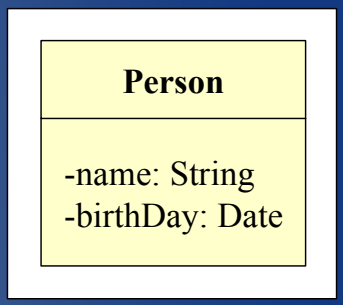
Principiellt gör man djup kopiering enligt följande:

```
public class SomeClass implements Cloneable {
   . . .
   @Override
   public SomeClass clone() {
      try {
         // Fix all primitives and immutables
         SomeClass result = (SomeClass) super.clone();
         // Handle mutables
         // code here (deep copy)
         . . .
         return result;
      }
      catch (CloneNotSupportedException) {
         return null; //never invoked
      }
   }//clone
}//SomeClass
```

Djup kopiering – enkelt exempel

Klassen Person har en muterbar referensvariabel birthDay, av typen java.util.Date.

```
import  java.util.Date;
public class Person implements Cloneable {
   private String name;
   private Date birthDay;
   . . .
   @Override
   public Person clone() {
      try {
         Person result = (Person) super.clone();
          result.birthDay = (Date) birthDay.clone();
          return result;
      }
      catch (CloneNotSupportedException e) {
         return null; //never invoked
      }
   }//clone
}//Person
```



Kloning och arv

Vid arv anropas clone() metoden i superklassen, varefter de muterbara referensvariablerna som deklareras i subklassen måste djup kopieras.

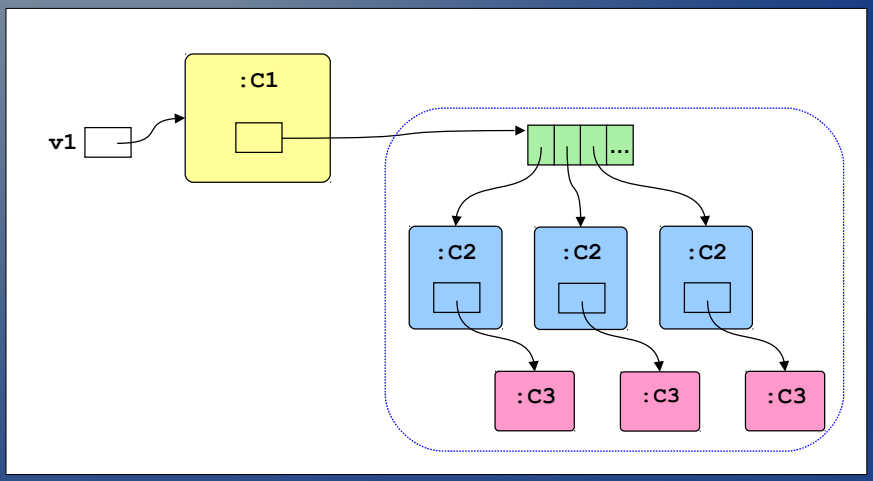
```
import java.util.Date;
public class Member extends Person {
   private Date dayOfMembership;
   . . .
   @Override
   public Member clone() {
      Member result = (Member) super.clone();
      //add copies of sub class specified fields to result
      result.dayOfMembership = (Date) dayOfMembership.clone();
      return result;
   }//clone
}//Member
```

Varför behöver inte Member implementera Cloneable?

Varför behövs inte något **try-catch**-block?

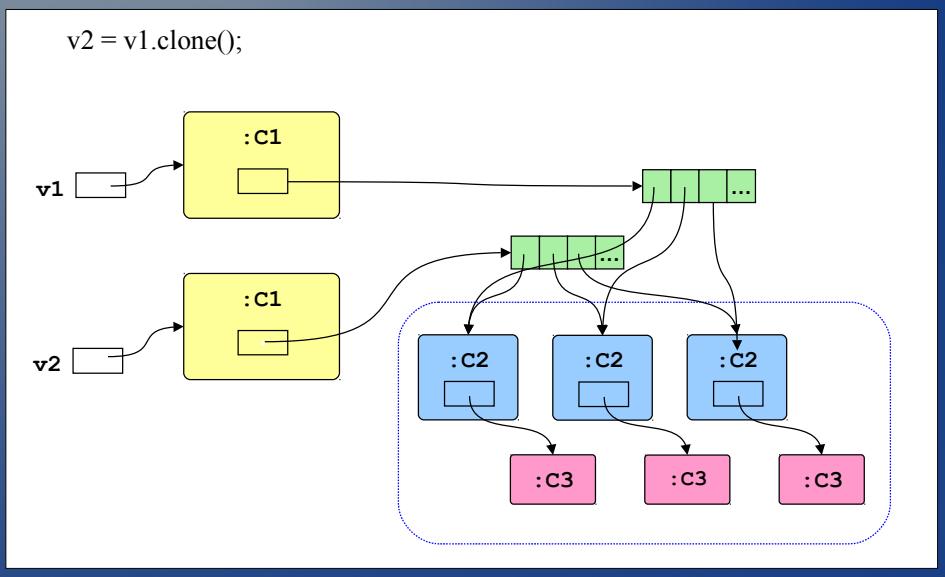
Djup kopiering – krångligare exempel

I nedanstående scenario vill vi göra en göra en djup kloning av objektet som refereras av variabeln v1.



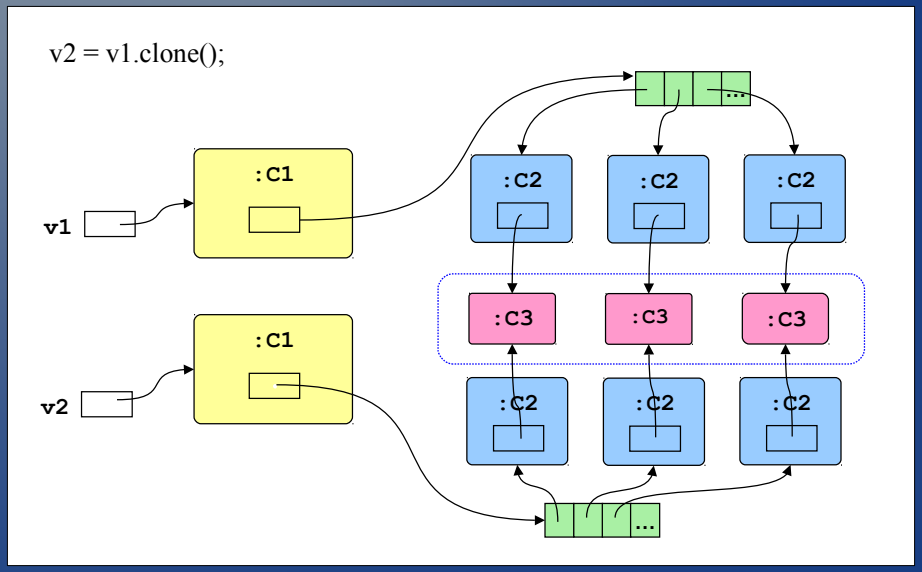
Djup kopiering – felaktig

I nedanstående scenario skapar metoden clone() i klassen C1 en kopia av sin instansvariabel, vilken är en lista. Men clone() skapar inte kopior av elementen i listan.



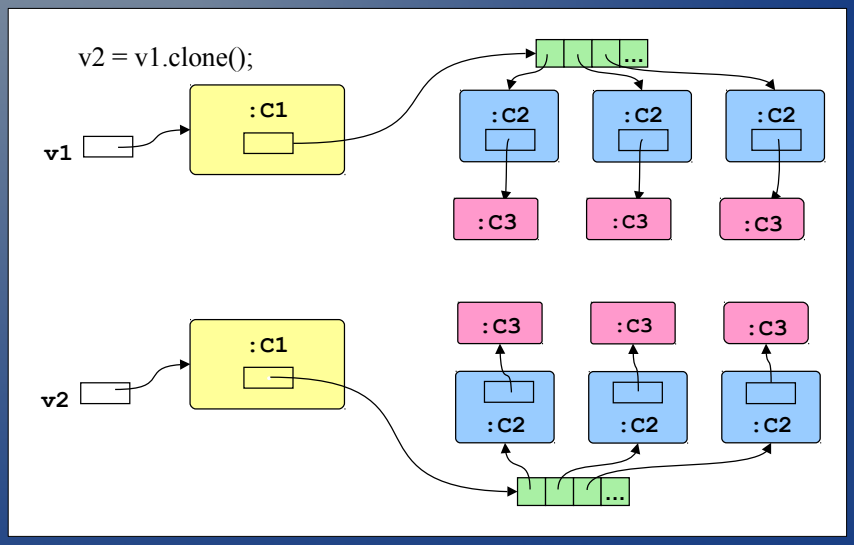
Djup kopiering – felaktig

I nedanstående scenario skapar metoden clone() i klassen C1 en djup kopia av sin instansvariabel (listan och elementen i listan) på ett korrekt sätt – men clone() i klassen C2 gör en grund kopia.



Djup kopiering – korrekt

I nedanstående scenario har vi en korrekt djup kopiering. Metoden clone() i klassen C1 skapar en kopia av listan och elementen i listan – och clone() i klassen C2 gör en djup kopiering.



Djup kopiering – samlingar

Metoden clone() i samlingar och arrayer använder grund kloning. Detta betyder att alla instansvariabler som är samlingar eller arrayer måste klonas ”element för element” om elementen är muterbara.

```
public class SomeClass implements Cloneable {
   private List<Person>  list = new ArrayList<Person>();
   . . .
   @Override
   public SomeClass clone() {
      try {
         SomeClass result = (SomeClass) super.clone();
         List<Person> copy = new ArrayList<Person>(list.size());
         for (Person item: list)
               copy.add(item.clone());
         result.list = copy;
         return result;
      }
      catch (CloneNotSupportedException e) {
         return null; //never invoked
      }
   }//clone
}//SomeClass
```

Kopieringskonstruktor och kloning

En *kopieringskonstruktor* tar som parameter ett objekt av samma klass och gör en djupkopiering av parameterobjektet.

```
import  java.util.Date;
public class Person implements Cloneable {
   private String name;
   private Date birthDay;
   public Person(String name, Date birthDay) {
      this.name = name;
      this.birthDay = (Date) birthDay.clone();
   }
   public Person(Person other) {
      this.name = other.name;
      this.birthDay = (Date) other.birthDay.clone();
   }
   . . .
   @Override
   public Person clone() {
      return new Person(this);
   }//clone
}//Person
```

kopieringskonstruktor

Kopieringskonstruktor och arv

Om en basklass har en kopieringskonstruktor, kan denna anropas av en kopieringskonstruktor i en subklass:

```
import java.util.Date;
public class Member extends Person implements Cloneable {
   private Date dayOfMembership;
   public Member(String name, Date birthDay, Date dayOfMembership) {
      super(name, birthDay);
      this.dayOfMembership = (Date) dayOfMembership.clone();
   }
   public Member(Member other) {
      super(other);
      this.dayOfMembership = (Date) dayOfMembership.clone();
   }
    . . .
@Override
   public Member clone() {
      return Member(this);
   }//clone
}//Member
```

kopieringskonstruktor

Förhindra kloning

Det finns situationer då man, av olika anledningar, vill förhindra kloning. Nedanstående scheman är då användbara:

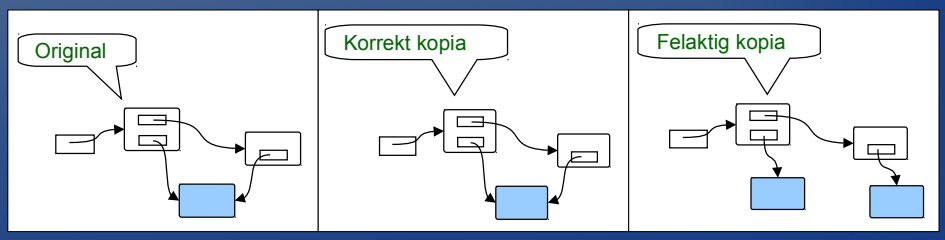
```
public class NoCopy {
   private int value;
   private boolean  status;
   private String str;
   . . .
   public NoCopy clone() throws CloneNotSupportedException {
      throw new CloneNotSupportedException();
   }//clone
}//NoCopy
```

```
public class NoCopySubClass extends SomeCopyClass {
   private int value;
   private boolean  status;
   private String str;
   . . .
   public NoCopySubclass clone() throws CloneNotSupportedException {
      throw new CloneNotSupportedException();
   }//clone
}//NoCopySubClass
```

Problem vid kloning

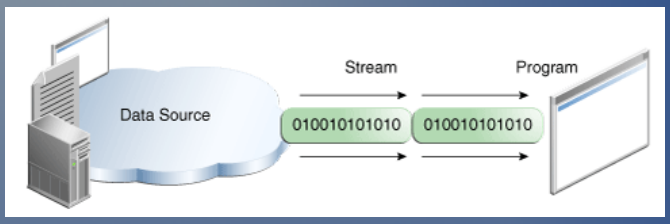
Det finns många problem med att implementer clone():

- superklassen måste implementerar clone()
- superklassens implementation måste vara korrekt
- många klasser saknar implementation av clone()
- många klasser har felaktig implementation av clone()
- alla instansvariabler som är samlingar eller arrayer måste klonas ”element för element”
- i cykliska strukturer och strukturer där objekt är delade, måste också motsvarande objekt vara delade i kopian
- . . .



I/O-ramverket i Java

Utan att kunna läsa och skriva data skulle de flesta program vara ganska meningslösa. För läsning och skrivning använder Java *streams* (strömmar).

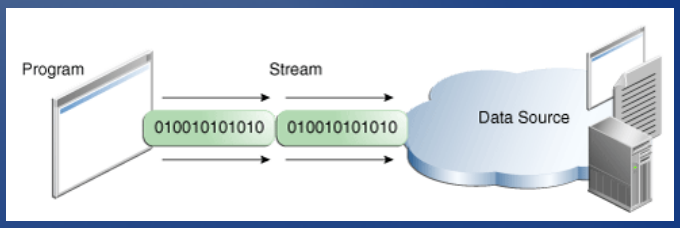


Läsning kan ske från:

- tangentbordet
- musen
- filer
- spelkonsoler
- nätverk
- temperaturgivare
- . . .

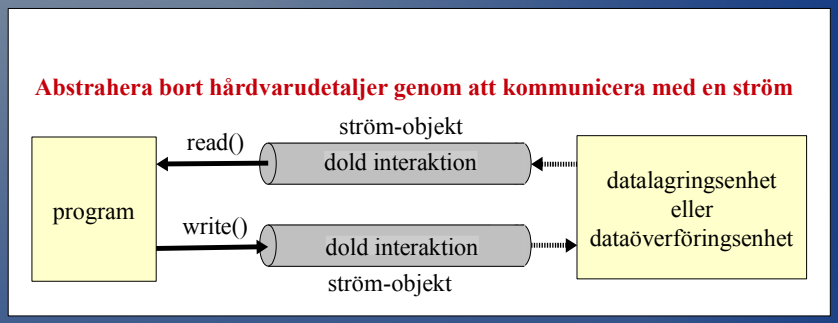
Skrivning kan ske till:

- bildskärmen
- filer
- nätverk
- spelkonsoler
- högtalare
- . . .



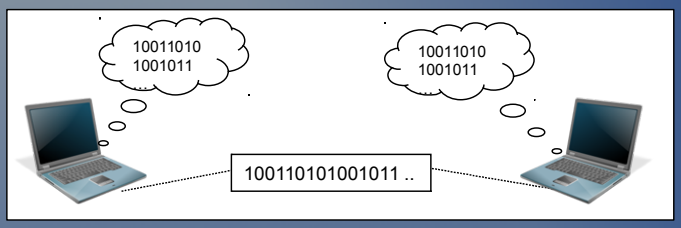
I/O-ramverket i Java

En ström abstraherar bort hårdvarudetaljerna i den fysiska enheten till vilken läsning/skrivning sker. Programmet vet egentligen inte vad som finns i andra ändan av strömmen. Oavsett enhet, sker läsning och skrivning på ett likartat sätt.



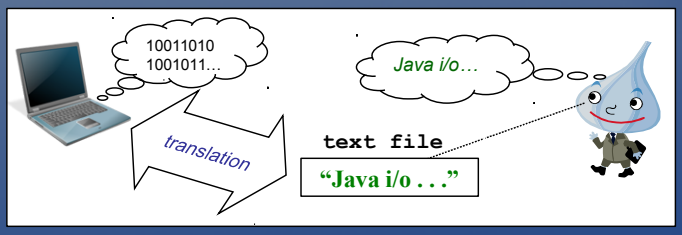
Notera: Inget strömobjekt i Java hanterar både read() och write(). Ett program som både läser och skriver data behöver således minst två strömmar.

byte-strömmar och char-strömmar



byte-strömmar:

Datorer kommunicerar effektivast genom att använda binärer.



char-strömmar:

Människor föredrar att kommunicera med text.

Grunderna för I/O-klassernas utformning

Javas I/O-ramverk är uppbyggt med användning av designmönstret *Decorator*.

Varje ström-klass har ett *mycket begränsat ansvarsområde* och önskat beteende får genom att koppla i olika strömmar på lämpligt sätt.

Basen utgörs av de fyra *abstrakta* klasserna

OutputStream	hantering av utströmmar för godtycklig data
InputStream	hantering av inströmmar för godtycklig data
Writer	hantering av utströmmar för text
Reader	hantering av utströmmar för text

Strömmar för godtycklig data kallas ***byte**-strömmar*, eftersom man skickar en byte (8 bits) i taget. Kan användas för alla sorters data (objekt, komprimerade filer, bilder, ljud, osv).

För textströmmar skickar man en **char** (16 bits) i taget, varför dessa även kallas ***char**-strömmar*. De är speciellt anpassade för text (e-post, överföring av HTML-kod, .java-filer, osv).

Grunderna för I/O-klassernas utformning

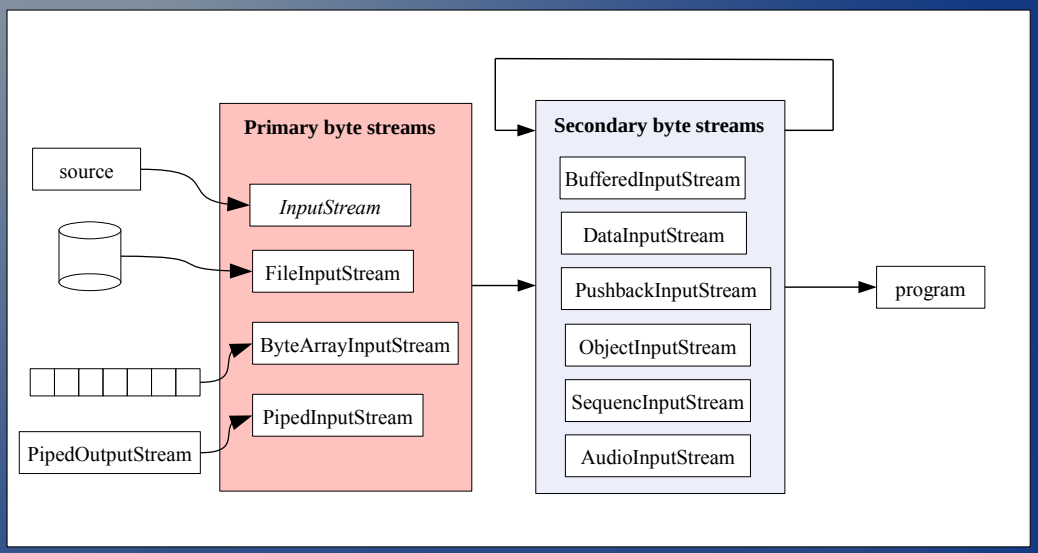
Det finns olika klasser för att t.ex.:

- läsa från filer
- skriva till filer
- buffring av data
- filtrering av data
- . . .

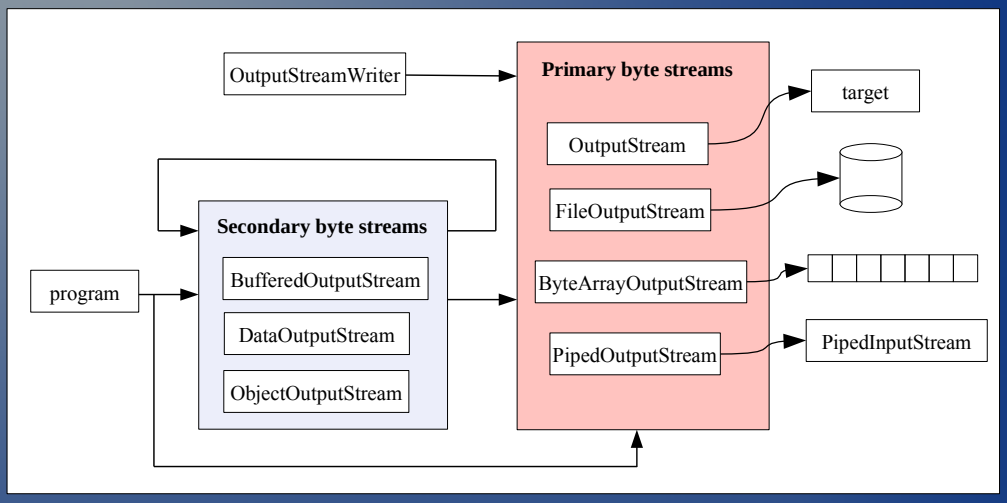
Det finns två olika kategorier av strömklasser:

- *konstruerande strömklasser*, som används för att skapa nya strömmar.
- *dekorerande strömklasser*, som används för att ge existerande strömmar nya egenskaper.

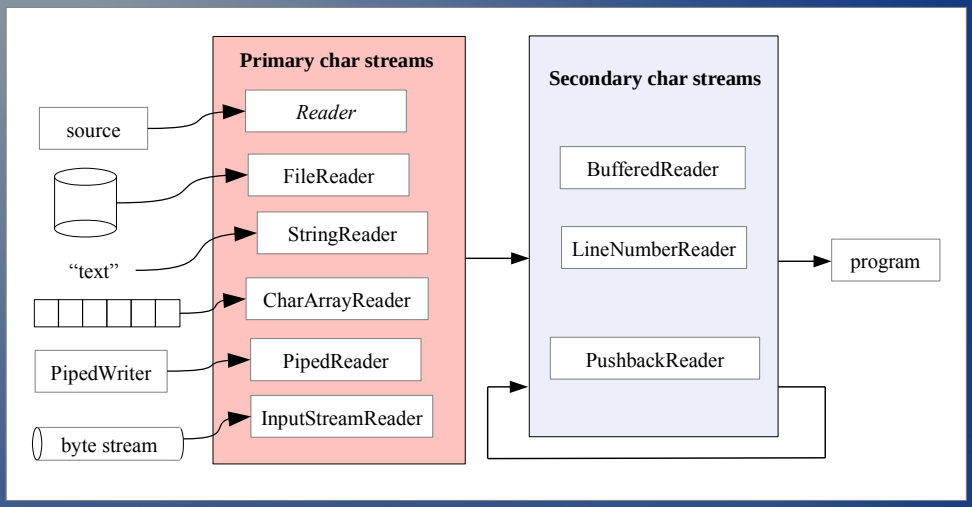
Dataflödet för byte-inströmmar



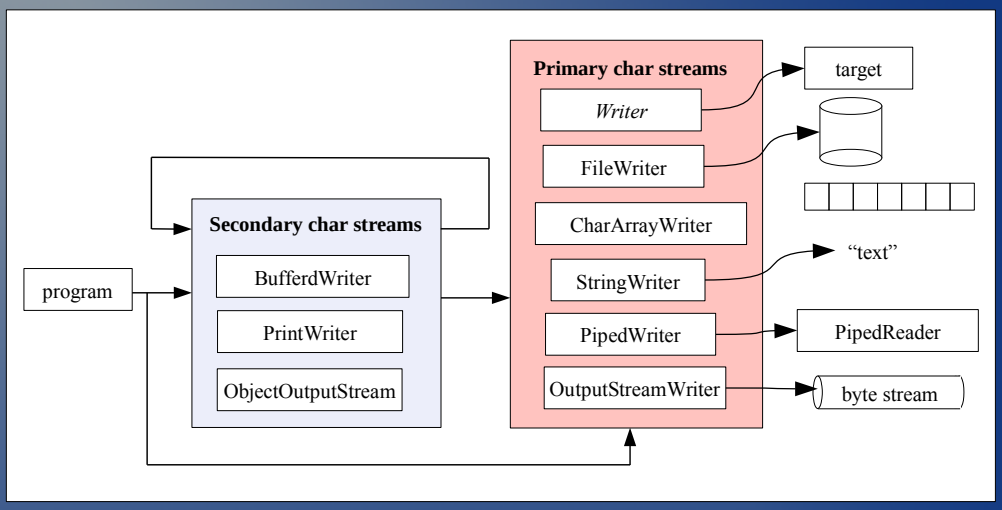
Dataflödet för byte-utströmmar



Dataflödet för tecken-inströmmar



Dataflödet för tecken-utströmmar



read() och write()

InputStream och Reader har bl.a. metoden

public int read() **throws**IOException

OutputStream och Writer har bl.a. metoden

public void write(**int** i) throws IOException

Vare sig det gäller läsning eller skrivning av bytes eller tecken, så behövs det ett extra värde för att *markera slutet av en ström.*

I Java har man valt att markera slutet med heltalet -1.

”Normala” bytes representeras som heltal från 0 till 255.

”Normala” tecken representeras som heltal från 0 till 65535.

read() och write()

Kontroll av slut i en byteström:

```
InputStream in = . . . ;
int i = in.read();
if (i != -1)
   byte b = (byte) i;
```

- testa om 'slut på strömmen'
- om inte 'slut på strömmen': typomvandla

Kontroll av slut i en teckenström:

```
Reader in = . . . ;
int i = in.read();
if (i != -1)
   char c = (char) i;
```

- testa om 'slut på strömmen'
- om inte 'slut på strömmen': typomvandla

Metoder i InputStream

De viktigaste metoderna är:

public abstract int read() **throws**IOException
Läser en **byte**, returnerar den som en **int** (0-255). Returnerar -1 om strömmen är slut. Väntar till indata är tillgängliga. Den **byte** som returneras tas bort från strömmen.

public int read(**byte**[] buf) **throws**IOException
Läser in till ett **byte**-fält. Slutar läsa när strömmen är slut. Returnerar så många bytes som lästes.

public void close() **throws**IOException
Stänger strömmen.

IOException kan t.ex. fås om man försöker läsa från en stängd ström.

Man skall alltid stänga en ström när man läst eller skrivit färdigt, annars riskerar man att data kan gå förlorat.

Det finns ingen open()-metod: strömmen öppnas då den skapas.

Kopiera en binärfil till en annan binärfil

```
import java.io.*;
public class CopyBinaryFile{
   public static void main(String[] args) {
      try {
         InputStream source = new FileInputStream(args[0]);
         InputStream in = new BufferedInputStream(source);
         OutputStream target = new FileOutputStream(args[1]);
         OutputStream out = new BufferedOutputStream(target);
         int input;
         try {
            while ((input = in.read()) != -1) {
               out.write(input);
            }
         } catch (IOException e) {
            System.out.println("Reading failed! ");
         } finally {
            in.close();
            out.close();
         }
      }
```

Om filen finns skrivs den över!

Programmet kopiera en binärfil till en annan binärfil. Filnamnen ges via argumentlistan. Buffring används av effektivitetsskäl. Felhantering görs.

```
         catch (FileNotFoundException e) {
            System.out.println("The file " + args[0] + " don't exist!");
         } catch (IOException e) {
               System.out.println("Closing files failed! ");
         }
      }//main
   }//CopyBinaryFile
```

Lägg till innehåll i en binärfil

```
import java.io.*;
public class CopyBinaryFile{
   public static void main(String[] args) {
      try {
         InputStream source = new FileInputStream(args[0]);
         InputStream in = new BufferedInputStream(source);
         OutputStream target = new FileOutputStream(args[1], true);
         OutputStream out = new BufferedOutputStream(target);
         int input;
         try {
            while ((input = in.read()) != -1) {
               out.write(input);
            }
         } catch (IOException e) {
            System.out.println("Reading failed! ");
         } finally {
            in.close();
            out.close();
         }
      }
```

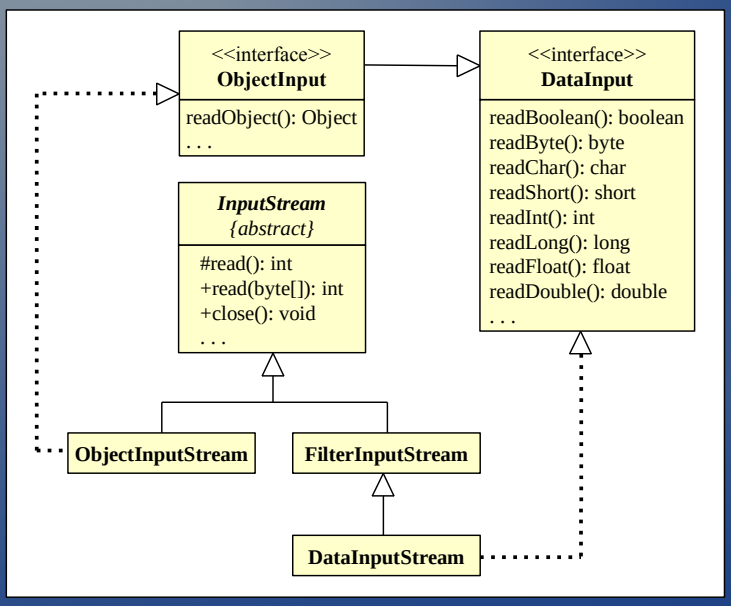
Det som skrivs läggs till sist i filen

Programmet lägger till innehållet i en binärfil till en annan binärfil.

```
         catch (FileNotFoundException e) {
            System.out.println("The file " + args[0] + " don't exist!");
         } catch (IOException e) {
               System.out.println("Closing files failed! ");
         }
      }//main
   }//CopyBinaryFile
```

Klasserna DataInputStream och ObjectInputStream

Klassen DataInputStream används för att läsa Javas primitiva datatyper och klassen ObjectInputStream används för att läsa objekt.

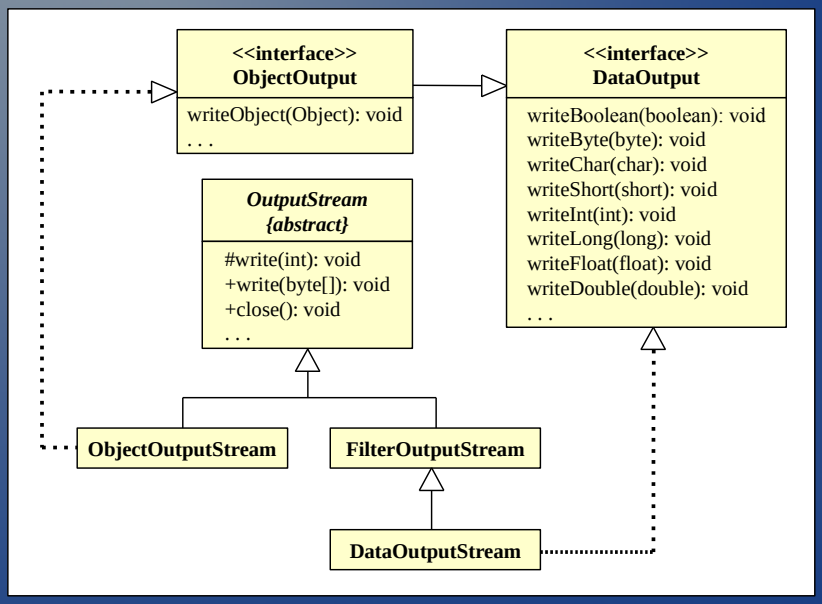


Om det inte finns något att läsa kastar metoderna EOFException.

readObject() kastar ClassNotFoundException om felaktigt objekt läses.

Klasserna DataOutputStream och ObjektOutputStream

Klassen DataOutputStream används för att skriva Javas primitiva datatyper och klassen ObjectOutputStream används för att skriva objekt.



Exempel: Skriva reella tal till en binärfil

Programmet skriver ut reella tal på en binärfil.

```
import java.io.*;
public class WriteDoublesToBinaryFile {
   public static void main( String[] args ) throws IOException {
      double[] f = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};
      FileOutputStream  outfile =   new FileOutputStream("data.bin");
      DataOutputStream destination = new DataOutputStream(outfile);
      for (int i = 0; i < f.length; i++)
         destination.writeDouble(f[i]);
      destination.close();
   }//main
}//WriteDoublesToBinaryFile
```

Exempel: Läsa tal från en binärfil

```
import java.io.*;
public class ReadDoublesFromBinaryFile {
   public static void main( String[] args ) throws IOException {
      FileInputStream infile =  new FileInputStream("data.bin");
      DataInputStream source = new DataInputStream(infile);
      double sum = 0.0;
      while (source.available() > 0) {
         double value = source.readDouble();
         sum = sum + value;
      }
      source.close();
      System.out.println("The sum of the numbers are " + sum);
   }//main
}//ReadDoublesFromBinaryFile
```

Programmet läser ett okänt antal reella tal från en binärfil och beräknar summan av dessa tal.

In- och utmatning av objekt

För att skriva respektive läsa objekt använder man sig av strömklasserna ObjectOutputStream respektive ObjectInputStream.

I ObjectOutputStream finns metoden writeObject, som omvandlar ett godtyckligt objekt till ren data och skickar iväg det på utströmmen.

I ObjectInputStream finns metoden readObject, som läser data och omvandlar den tillbaks till ett objekt igen.

Det som krävs är att de objekt som skickas måste implementera gränssnittet Serializable.

Innehåller objektet referenser till andra objekt måste också dessa objekt implementera gränssnittet Serializable, vilket är enkelt då detta gränssnittet saknar metoder.

Många av standardklasserna implementerar Serializable.

Serializable - exempel

```
import java.io.*;
public class Product implements Serializable {
   private String name;
   private int number;
   private double price;
   public Product(String name, int number, double price) {
      this.name = name;
      this.number = number;
      this.price = price;
   }
   . . .
   public String toString() {
     return "Product name: " + name
           + "\nProductId: " + number
           + "\nPrice: " + price;
   }
}//Product
```

Klassen Product implementerar Serializable.

ObjectOutputStream - exempel

Programmet skriver en lista med objekt av klassen Product på filen product.data.

```
import java.io.*;
import java.util.*;
public class WriteProductList {
   public static void main(String[] args )  {
      List <Product> productList = new ArrayList<Product>();
      productList.add(new Product("Screwdriver", 121835, 123.5));
      productList.add(new Product("Spanner", 534893, 358.5));
      productList.add(new Product("Nippers", 989567, 292.0));

      try {
         FileOutputStream outfile = new FileOutputStream("product.data");
         ObjectOutputStream destination = new ObjectOutputStream(outfile);
         try {
            destination.writeObject(productList);
         } catch (IOException e) {
            System.out.println("Writing failed!");
          } finally {
            try {
               destination.close();
            } catch (IOException e) {
               System.out.println("Closing file failed!");
            }
         }
      } catch (IOException e) {
         System.out.println("Open file failed!");
      }
   }//main
}//WriteProductList
```

ObjectInputStream - exempel

Programmet läser in och skriver ut en lista med objekt av klassen Product från filen product.data.

```
import java.io.*;
import java.util.List;
public class ReadProductList {
   public static void main(String[] args ) {
      try {
         FileInputStream infile = new FileInputStream("product.data");
         ObjectInputStream source = new ObjectInputStream(infile);
         try {
            List<Product>  productList = (List<Product>) source.readObject();
            for (Product p : productList)
               System.out.println(p);
         }
         catch (EOFException e) {}
         catch (ClassNotFoundException e) {
            System.out.println("Unknown object read!");
         } catch (IOException e) {
            System.out.println("Reading failed!");
         } finally {
            try {
               source.close();
            } catch (IOException e) {
               System.out.println("Closing file failed!");
            }
         }
      } catch (IOException e) {
         System.out.println("Opening file failed!");
      }
   }//main
}//ReadProductList
```

Bekvämlighetsklassen PrintWriter

Klass PrintWriter används för att skriva ut objekt och primitiva typer till en textström.

PrintWriter innehåller bl.a. de överlagrade metoderna print och println för samtliga primitiva typer, samt för String och Object:

```
PrintWriter pw = newPrintWriter("values.txt");
pw.println(12.34);
pw.print(456);
pw.println("Some words");
pw.print(new Rectangle(5, 10, 5, 15));
```

Anropar objektets toString()-metod

Exempel: Skriv ett fält av reella tal till en textfil

Programmet skriver ut storleken samt elementen i ett fält med reella tal på en textfil. Om filen finns görs utskrifterna sist i filen.

```
import java.io.*;
public class WriteDoubleArray {
   public static void main( String[] args ) throws IOException {
      double[] f = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};
      FileWriter outfile =  new FileWriter("data.txt", true);
      PrintWriter destination =  new PrintWriter(outfile);
      destination.println(f.length);
      for (int i = 0; i < f.length; i++)
         destination.print(f[i] + " ");
      destination.close();
   }//main
}//WriteDoubleArray
```

Klassen Scanner

För läsning finns ingen ström motsvarande PrintWriter som tillhandahåller metoder för att översätta strängar till numeriska värden. Istället används klassen Scanner. Klassen Scanner innehåller bl.a. följande konstruktorer och metoder:

Scanner(Readable source)	skapar en ny scanner som kopplas till source
int nextInt()	returnerar nästa token som en **int**
double nextDouble()	returnerar nästa token som en **double**
boolean nextBoolean()	returnerar nästa token som en **boolean**
String next()	returnerar nästa token som en `String`
boolean hasNextInt()	returnerar värdet **true** om nästa token är en **int**, annars returneras **false**
boolean hasNextDouble()	returnerar värdet **true** om nästa token är en **double**, annars returneras **false**
boolean hasNextBoolean()	returnerar värdet **true** om nästa token är en **boolean**, annars returneras **false**
boolean hasNext()	returnerar värdet **true** om det finns fler tokens, annars returneras **false**

Exempel: Läsa ett fält av reella tal från en textfil

```
import java.io.*;
import java.util.Scanner;
public class ReadDoubleArray {
   public static void main( String[] args ) throws IOException {
      try {
         FileReader infile =  new FileReader("data.txt");
         Scanner sc = new Scanner(infile);
          int antal = sc.nextInt();
          double[] f = new double[antal];
          for (int i = 0; i < antal ; i++) {
             f[i] = sc.nextDouble();
          }
          infile.close();
      } catch (FileNotFoundException e) {
         System.out.println("File could not be found!");
      }
   }//main
}//ReadDoubleArray
```

Programmet läser antalet element i ett reellt fält från en textfil, skapar fältet samt läser in elementen från textfilen.

Klassen File

Klassen File ger en plattformsoberoende och abstrakt representation av filer och mappars fullständiga adresser (*pathnames*).

Klassen File innehåller bl.a. följande metoder:

getName()	returnerar namnet på filen
getPath()	returnerar adressen som en sträng
isDirectory()	returnerar **true** om filen är ett bibliotek
listFiles()	returnerar en lista av filerna i biblioteket
delete()	tar bort filen
renameTo(File dest)	ändrar namn på filen
exists()	kontrollerar om en fil eller ett bibliotek finns
canRead()	kontrollerar om en fil eller ett bibliotek får läsas

Klassen File

```
import java.io.File;
public class DirectoryTest {
   public static void main(String[] args) {
      File theFile = new File(args[0]);
      if (theFile.isDirectory()) {
         System.out.println("Ett bibliotek!");
         File[] content = theFile.listFiles();
         for (int i = 0; i < content.length; i++)
            System.out.println(content[i].getName());
      }
      else
         System.out.println("Inget bibliotek!");
   }//main
}//DirectoryTest
```

Networking – strömmar över nätverk

I paketet java.net finns klasser som erbjuder kommunikation över nätverk.

```
URL url = new URL("http://www.valutor.se/");
InputStreamReader insr = new InputStreamReader(url.openStream());
BufferedReader buf = new BufferedReader(insr);
```

```
int portNumber = 1234;
Socket sock = new Socket(InetAddress.getLocalHost(), portNumber);
InputStreamReader insr = new InputStreamReader(sock.getInputStream());
BufferedReader in = new BufferedReader(insr);
PrintWriter out = new PrintWriter(sock.getOutputStream());
```

Networking – web scraping*

Consumption for Sweden

Last 8 | 48 | 47 | 46

Graph | Table

Last 8 days (Volume in MWh/h)

	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Date	12.10.14	13.10.14	14.10.14	15.10.14	16.10.14	17.10.14	18.10.14	**19.10.14**
00-01	11872	12146	12861	12561	13493	13386	12732	**12249**
01-02	11530	11922	12520	12258	13159	12998	12453	**12223**
02-03	11449	11781	12289	12241	12950	12870	12211	**12045**
03-04	11342	11945	12137	12305	12911	12821	12176	**12026**
04-05	11442	12095	12253	12364	12928	13045	12108	**12002**
05-06	11615	12665	12968	13048	13676	13850	12221	**11913**
06-07	11745	14754	14721	15164	15552	15780	12617	**12238**
07-08	12044	16617	16470	16825	17109	17552	13167	**12399**
08-09	12342	17000	16761	17105	17245	17806	13892	**12754**
09-10	13170	17057	16782	17205	17843	17726	14547	**13380**
10-11	13753	17063	17027	17515	18016	17741	14629	**13896**
11-12	14028	17352	16808	17545	18058	17429	14771	**14139**
12-13	13953	16967	16536	17231	17790	16950	14674	**14255**
13-14	13979	16938	16297	17102	17627	16620	14566	-
14-15	13733	16763	16169	16828	17515	16346	14466	-
15-16	13800	16582	16367	17068	17591	16089	14424	-
16-17	13961	16352	16742	17124	17341	15963	14593	-
17-18	14390	16831	16833	17516	17763	16506	15139	-
18-19	15149	17274	17170	17869	18119	16871	15305	-
19-20	15221	17275	17007	17568	17648	16471	14844	-
20-21	14779	16417	16204	16739	17009	15653	14117	-
21-22	13832	15360	14894	15874	16143	14874	13544	-
22-23	13058	14164	13899	14745	14951	13902	13004	-
23-24	12413	13220	12929	13821	13737	13206	12387	-
Min	11342	11781	12137	12241	12911	12821	12108	**11913**
Max	15221	17352	17170	17869	18119	17806	15305	**14255**
Sum	314600	366540	364644	375621	386174	372455	328587	**165519**

Source: SVENSKA KRAFTNÄT SWEDISH NATIONAL GRID

På hemsidan finns en tabell överden totala elförbrukningen i Sverige i MWh för varje timme under de senaste 8 dygnen.

**Web scraping* – hämta data från nätet och sedan förädla denna data.*

Networking – web scraping

Programmet läser och skriver ut elförbrukningsdatan från hemsidan.

```
import java.io.*;
import java.net.URL;
public class URLTest {
   public static void main(String[] args) throws IOException {
      webScarping();
   }
   public static void webScarping() throws IOException {
      URL url = new URL("http://www.cse.chalmers.se/edu/year/2014/course/TDA550/data.html");
      Scanner in = new Scanner( new InputStreamReader(url.openStream()));
      for(int i = 0; i < 24; i++) {
         while(in.findInLine("[0-2][0-9]-[0-2][0-9]") == null)
            in.nextLine();
         in.nextLine();
         for(int j = 0; j < 8; j++) {
            String num = in.findInLine("[0-9]+");
            if (num == null) {
                  System.out.print(0 + " ");
            } else {
                  System.out.print(Integer.parseInt(num) + " ");
            }
            in.nextLine();
         }
          System.out.println();
      }
      in.close();
   }
}//URLTest
```