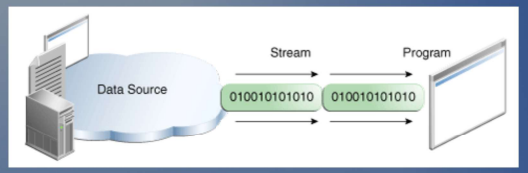
Föreläsning 13

I/O-ramverket
Nästlade klasser

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 strömmar (*streams*).

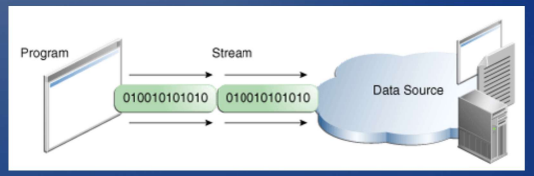


Läsning kan ske från:

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

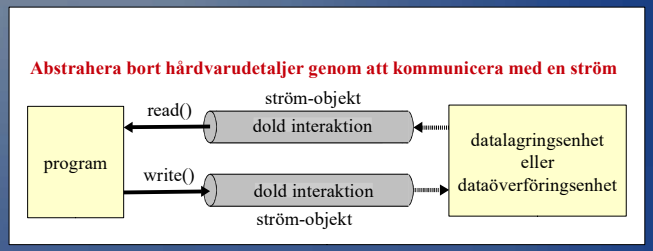
Skrivning kan ske till:

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



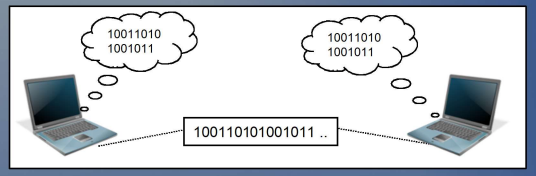
I/O-ramverket i Java

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.



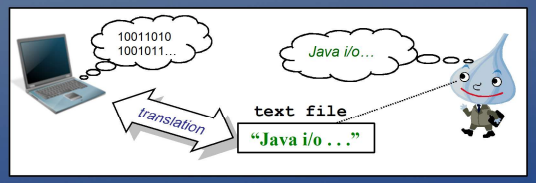
Notera: Inget strömobjekt i Java hanterar både read() och write(). Ett program som både läser och skriver data behöver således minst två strömmar.

byte-strömmar och char-strömmar



byte-strömmar:

Datorer kommunicerar effektivast binärt.



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 fås 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 inströmmar för text

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

För textströmmar skickar man en **char** (16 bitar) 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

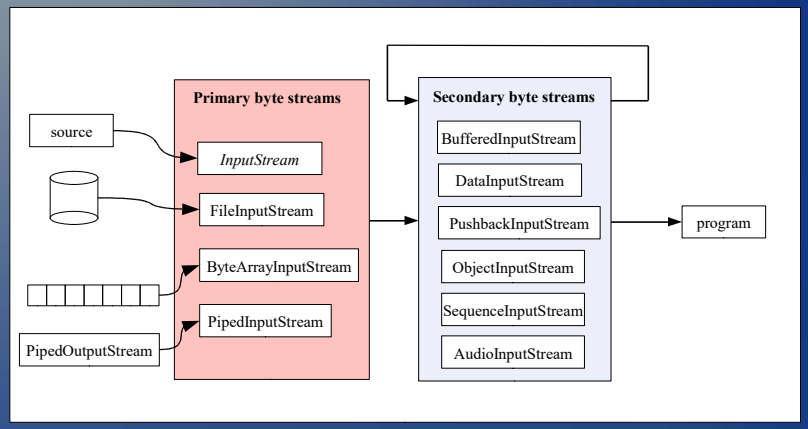
Det finns olika klasser för t.ex.

- läsning från filer
- skrivning till filer
- buffring av data
- filtrering av data
- . . .

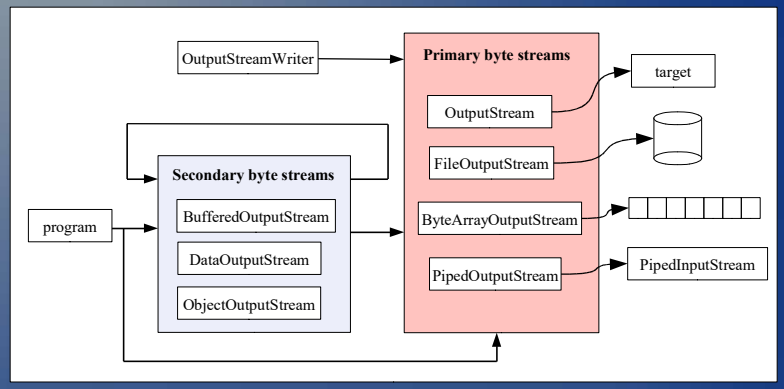
Det finns två olika kategorier av strömklasser:

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

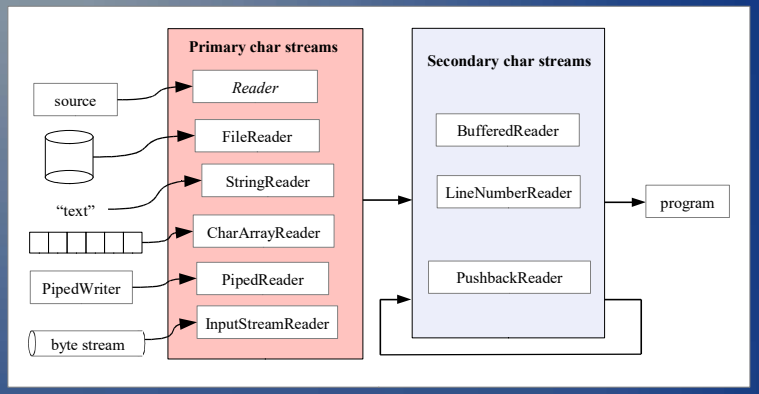
Dataflödet för byte-inströmmar



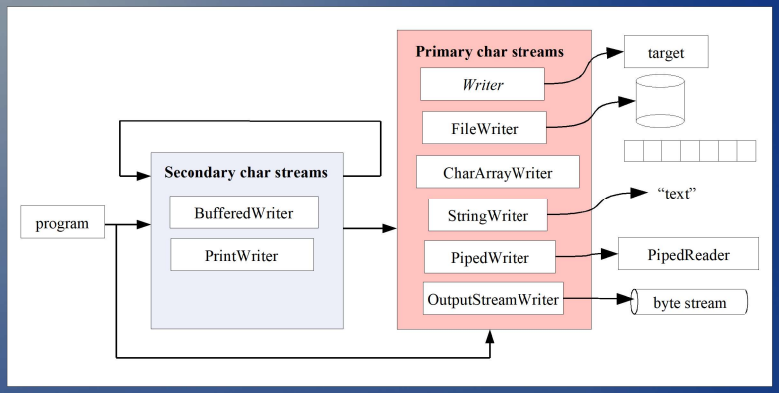
Dataflödet för byte-utströmmar



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 kopierar 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] + " doesn't exist!");
            } catch (IOException e) {
               System.out.println("Closing files failed! ");
            }
         }
      }
```


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();
         }
      }
```

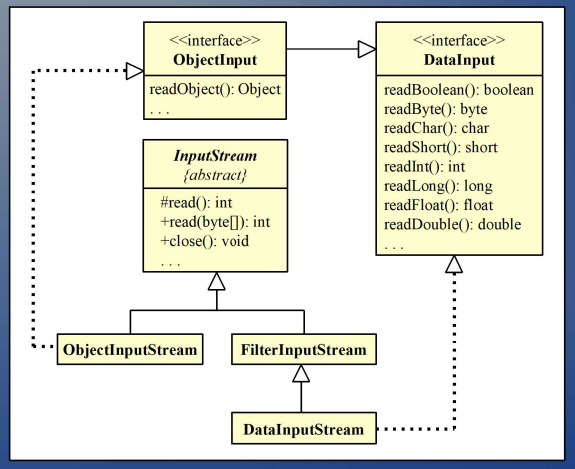
Det som skrivs läggs till sist i filen

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

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


Klasserna DataInputStream och ObjectInputStream

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

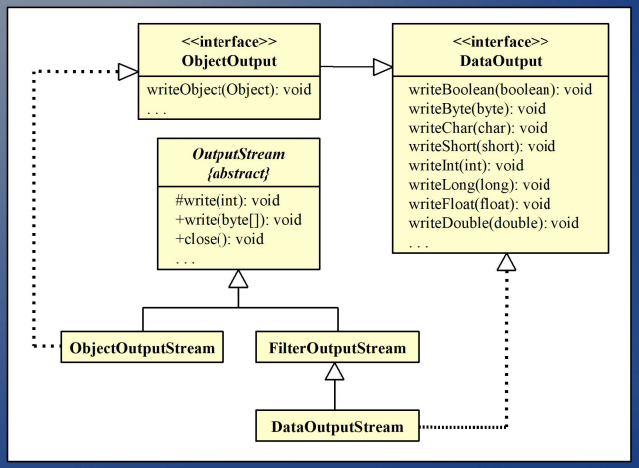


Om det inte finns något att läsa kastar metoderna EOFException.

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();
   }
}
```


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);
   }
}
```

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

Skrivning och läsning 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 seriell data och skickar iväg det på utströmmen.

I ObjectInputStream finns metoden readObject, som läser seriell 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;
   }
}
```

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!");
      }
   }
}
```


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!");
      }
   }
}
```


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();
   }
}
```


Klassen java.util.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 count = sc.nextInt();
         double[] f = new double[count];
         for (int i = 0; i < count ; i++) {
            f[i] = sc.nextDouble();
         }
         infile.close();
      } catch (FileNotFoundException e) {
         System.out.println("File could not be found!");
      }
   }
}
```

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 mappar (*directory*) i ett hierarkiskt filsystem.

Klassen File innehåller bl.a. följande metoder:

getName()	returnerar namnet på filen
getPath()	returnerar adressen som en sträng
isDirectory()	returnerar **true** om filen är en mapp
listFiles()	returnerar en lista av filerna i mappen
delete()	tar bort filen
renameTo(File dest)	ändrar namn på filen
exists()	kontrollerar om en fil finns
canRead()	kontrollerar om en fil 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!");
   }
}
```


Klassen File – Skriv ut ett helt filträd - rekursivt

```
import java.io.File;
public class FileTree {
   public static final int INDENT_STEP = 3;
   public static void main(String[] args) {
      printFileTree(0,new File(args[0]);
   }
   private static void printFileTree(int depth,File file) {
      printName(depth,file);
      if ( file.isDirectory() )
         for ( File f : file.listFiles() )
            printFileTree(depth + 1,f);
   }
   private static void printName(int depth,File file) {
      printIndentSpace(depth);
      printDirectoryMarking(file);
      System.out.println(file.getName());
   }
   ...
```

```
*lecture13          //Ex.
   *code
      *filetree
         FileTree.class
         FileTree.java
         README.TXT
   F13-OH.odp
```

Designmönstret Composite (jfr övn. 5.5)

```
private static void printIndentSpace(int depth) {
   for ( int i = 0; i < depth*INDENT_STEP; i++ )
      System.out.print(' '); // blanktecken
}
```

```
private static void printDirectoryMarking(File file) {
   if ( file.isDirectory() )
      System.out.print('*');
}
}
```


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 {
      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();
   }
}
```


Nästlade klasser

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 inkapslingen
- kan leda till kod som är lättare att läsa och underhålla.

Statiska medlemsklasser

- En statisk medlemsklass kan använda alla klassvariabler och klassmetoder 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;
   }
   public static class Helper {
      public static int oper(int val, int val2) {
         return val + val2 + minus(times*val2, val);
      }
   }
   . . .
}
```

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));
   }
}
```

Statiska medlemsklasser

Gränssnittet Map<K,V>:

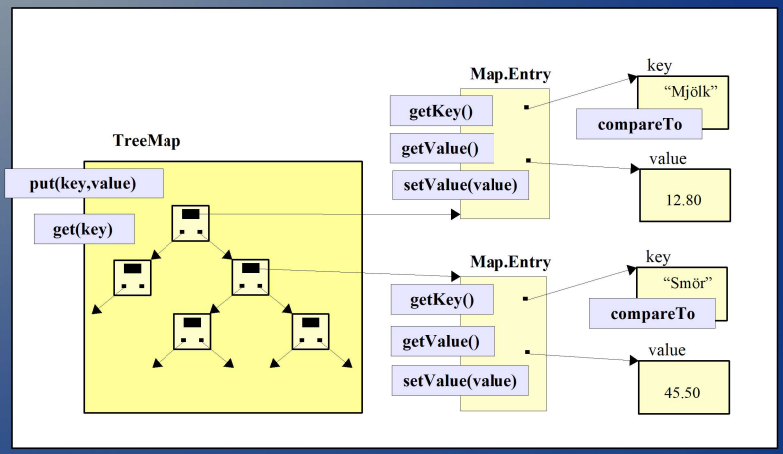
```
public interface Map<K,V> {
   // basoperationer
   V put(K key, V value);
   V get(Object key);
   V remove(Object key);
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   int size();
   boolean isEmpty();

   // samlingsvyer
   Set<K> keySet();
   Collection<V> values();
   Set<Map.Entry<K,V>> entrySet();

   // mängdoperationer
   void putAll(Map< ? extends K, ? extends V > m);
   void clear();
}
```

```
// Gränssnitt för elementen i entrySet
static interface Entry<K,V> {
   K getKey();
   V getValue();
   V setValue(V value);
}
```

Inre interface

Statiska medlemsklasser

Klassen TreeMap<K,V>:



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).
- Metoder i den omgivande klassen har åtkomst till alla medlemmar i den inre klassen (via en 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.Date;
import java.util.Iterator;
public class DummyDates implements Iterable<Date> {
      private final static int SIZE = 15;
       private Date[ ] dates = new Date[SIZE];
       . . .
       public DummyDates() {. . .}
       public Iterator<Date> iterator() {
            return new DSIterator();
       }
       . . .
       private class DSIterator implements Iterator<Date> {
            . . .
            public boolean hasNext() {. . .}
            public Date next() {. . .}
            public void remove() {. . .}
       }
}
```

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** eller är *"effectively final"* inom deklarationsblocket för den lokala klassen.

Lokala klasser

```
public class LocalClassExample {
  private int value  = 9000;

  public void aMethod(final int number) {
     class Local {
        int a = 10;
        public void printStuff() {
           System.out.println(value);
           System.out.println(number);
           System.out.println(a);
        }
     }
     Local local = new Local();
     local.printStuff();
  }

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

Ok!
number är **final** och
a är *effectively final*

Anonyma klasser

En entitet är anonym om den inte har något namn. Anonyma entiteter används ofta i ett program, t.ex. *anonyma objekt*

someList.add(***new*** *Integer(123)*);

I Java är det är även möjligt att definiera *anonyma klasser*

```
Comparator<Country> comp =
   new Comparator<Country>() { // anonymt objekt av 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();
```


Anonyma klasser – Ex. Lyssnare på GUI-komponenter

Anonyma klasser används ofta som ”throw away”-klasser när man endast behöver *ett* objekt av en klass, t.ex. lyssnarobjekt i ett GUI.

```
import java.awt.event.*;
import javax.swing.*;
public class Window implements ActionListener {
   private JButton startButton;

   public void createWindow() {
      . . .
      startButton = new JButton(“START”);
      startButton.addActionListener(this);
      . . .
   }
   . . .
   public void actionPerformed(ActionEvent e) {
      . . .
      if ( e.getSource() == startButton )
         startSomething();
      . . .
   }
}
```

Global lyssnare

Komponenten *skapas* här

… och händelsen *hanteras* här

Oflexibel fallananys

Anonyma klasser – Ex. Lyssnare på GUI-komponenter

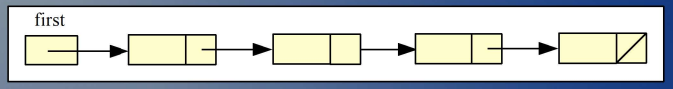
Om lyssnare definieras med anonyma klasser fås bättre *lokalitet* genom att lyssnarna kan skapas tillsammans med komponenterna de skall lyssna på:

```
import java.awt.event.*;
import javax.swing.*;
public class Window {
   private JButton startButton;
   . . .
   public void createWindow() {
      . . .
      startButton = new JButton("START");
      startButton.addActionListener(
         new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               startSomething();
            }
         }
      );
      . . .
   }
   . . .
}
```

Komponenten och lyssnaren skapas tillsammans

Ex. Inre klasser - Länkade listor

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



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

```
// skapar en tom lista
public SimpleList()
// returnerar true om listan är tom, annars false
public boolean isEmpty()
// 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 implemerntera noderna, och en privat inre klass för att implementera iteratorer för listklassen.

```
public class SimpleList<E> implements Iterable<E> {
   private Node first;

   private class Node {
      private E data;
      private Node next;
   }
   . . . methods on next slide
   private class SimpleIterator implements Iterator <E> {
      . . .
      public SimpleIterator() { . . . }
      public boolean hasNext() { . . . }
      public E next() { . . . }
      public void remove () { . . . }
   }
   . . .
}
```


Implementation – klassen SimpleList

```
import java.util.*;
public class SimpleList<E> implements Iterable<E> {
   private Node first;
   private long modificationCount;
   . . .
   public SimpleList() {
      first = null;
      modificationCount = 0;
   }
   public boolean isEmpty() {
      return first == null;
   }
   public E getFirst() {
      if ( isEmpty() )
         throw new NoSuchElementException();
      return first.data;
   }
```

```
   public void addFirst(E element) {
      Node newNode = new Node();
      newNode.data = element;
      newNode.next = first;
      first = newNode;
      modificationCount++;
   }
   public void removeFirst() {
      if ( isEmpty() )
         throw new NoSuchElementException();
      first = first.next;
      modificationCount++;
   }
   public Iterator<E> iterator() {
      return new SimpleIterator();
   }
   . . . private class SimpleIterator on next slide
```

Implementation – SimpleIterator

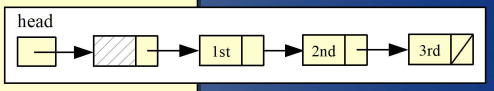
```
. . .
private class SimpleIterator implements Iterator <E> {
    private Node current;
    private final long thisModificationCount;

    public SimpleIterator() {
       current = first;
       thisModificationCount = modificationCount;
    }
    public boolean hasNext() {
       if ( modificationCount != thisModificationCount )
          throw new ConcurrentModificationException();
       return current != null;
    }
```

```
        public E next() {
           if ( !hasNext() )
              throw new NoSuchElementException();
           Node tmp = current;
           current = current.next;
           return tmp.data;
        }
        public void remove () {
           throw new UnsupportedOperationException();
        }
    }
}
```


SimpleList med metoden remove() i iteratorklassen

Implementation – SimpleIterator med remove

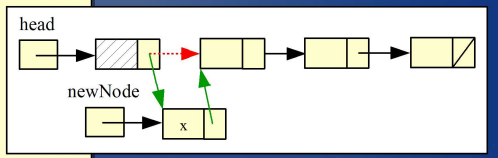
```
import java.util.*;
/**
 * SimpleList   A simple iterable linked list class.
 *         This alternative implementation uses an empty
 *         head node without data to simplify the algorithms.
 */
public class SimpleList<E> implements Iterable<E> {
   private Node head;
   private long modificationCount;

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

   public SimpleList() {
      head = new Node();
      modificationCount = 0;
   }
   . . .
```



- Genom att reservera en nod som *listhuvud* kommer alla noder i listan att ha en föregångare.
- Ingen information lagras i listhuvudets datadel (streckad).
- Listhuvudet förenklar bl.a. borttagning av noder i iterator-klassen eftersom det alltid finns en nod bakom noden som skall tas bort.

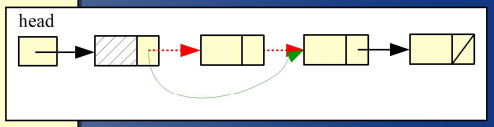
Implementation – SimpleIterator med remove

```
public boolean isEmpty() {
   return head.next == null;
}

public E getFirst() {
   if ( isEmpty() )
      throw new NoSuchElementException();
   return head.next.data;
}

public void addFirst(E element) {
   Node newNode = new Node();
   newNode.data = element;
   newNode.next = head.next;
   head.next = newNode;
   modificationCount++;
}
```



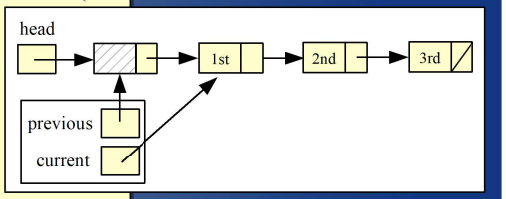
Implementation – SimpleIterator med remove

```
public void removeFirst() {
   if ( isEmpty() )
      throw new NoSuchElementException();
   head.next = head.next.next;
   modificationCount++;
}

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

. . . private class SimpleIterator on next slide
```



Implementation – SimpleIterator med remove

```
. . .
private class SimpleIterator implements Iterator <E> {
   private Node previous;
   private Node current;
   private final long thisModificationCount;
   private boolean nextIsCalled;

   public SimpleIterator() {
      previous = head;
      current = head.next;
      thisModificationCount = modificationCount;
      nextIsCalled = false;
   }

   public boolean hasNext() {
      if ( modificationCount != thisModificationCount )
         throw new ConcurrentModificationException();
      return current != null;
   }
   . . .
```

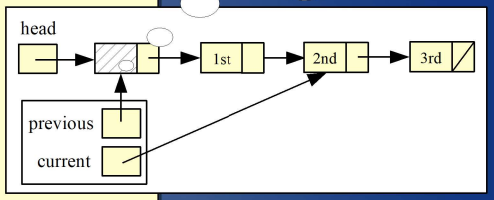


Vi måste garantera att noden som current pekar på inte tas bort.

Implementation – SimpleIterator med remove

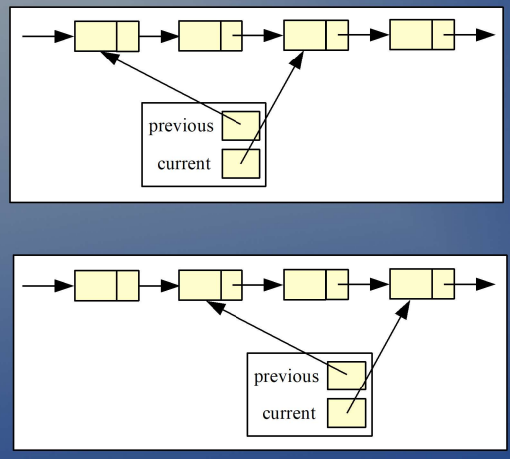
```
   public E next() {
      if ( modificationCount != thisModificationCount )
         throw new ConcurrentModificationException();
      if ( !hasNext() )
         throw new NoSuchElementException();
      advancePrevious();
      E returnValue = current.data;
      current = current.next;
      nextIsCalled = true;
      return returnValue;
   }

   private void advancePrevious() {
      if ( nextIsCalled )
         previous = previous.next;
   }
   . . .
}
```

Vid första anropet av next() ändras ej previous



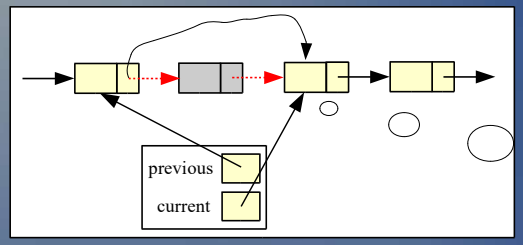

```
public void remove () {
   if ( modificationCount != thisModificationCount )
      throw new ConcurrentModificationException();
   if ( ! nextIsCalled )
      throw new IllegalStateException();
   previous.next = previous.next.next;
   nextIsCalled = false;
}
```


Implementation – SimpleIterator med remove

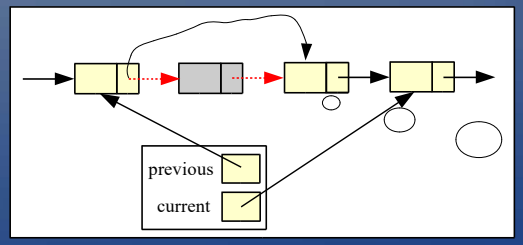


previous följer current två noder bakom

Implementation – SimpleIterator med remove



Vid anrop av remove()
behåller previous
sin position
remove får inte
anropas igen!



Efter anrop av next()
får remove
anropas igen!