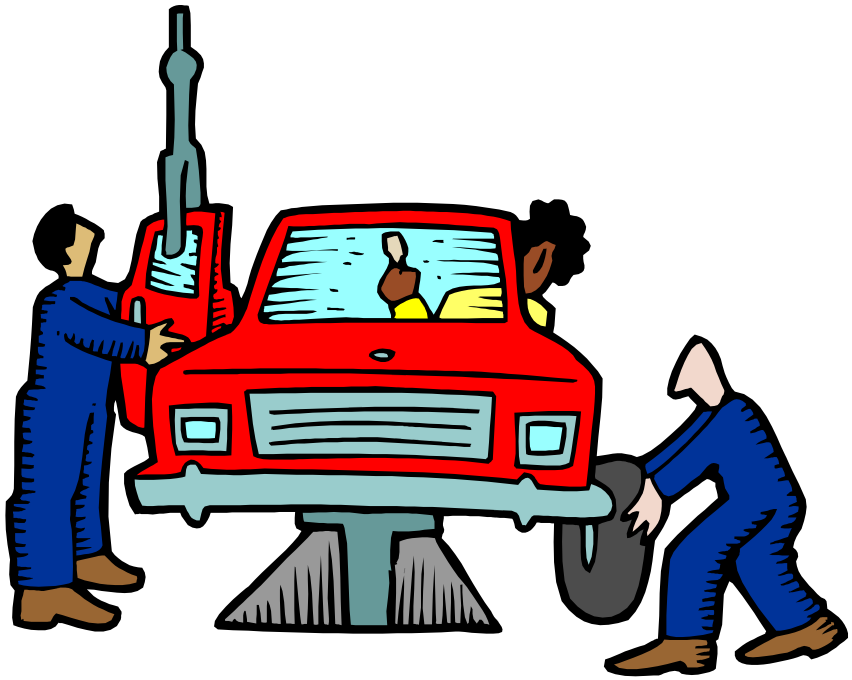
Frameworks



Plan

Frameworks

- Kollektioner
- Input/output

Nyt designmønster: Decorator

Frameworks

Et objektorienteret **framework** er en mængde af samvirkende klasser, der repræsenterer genbrugeligt design inden for et specifikt anvendelsesområde.

Et framework (ramme, skelet, stel) består typisk af en mængde af abstrakte klasser og grænseflader, der udgør delene i halvfærdige applikationer.

Til et framework er knyttet anvisninger på, hvorledes de abstrakte klasser udvides, hvorledes grænsefladerne implementeres, og hvorledes instanser bringes til at samvirke.

Eksempler på frameworks

AnimationApplet	(mini-framework til animering)
GUI frameworks	(AWT, Swing)
Collections	(kollektioner, objektbeholdere)
Input/output	(indlæsning og udskrivning)
RMI	(fjernmetodekald)
javaSimulation	(diskret simulering)
jDisco	(kombineret simulering)

Keld Helsgaun

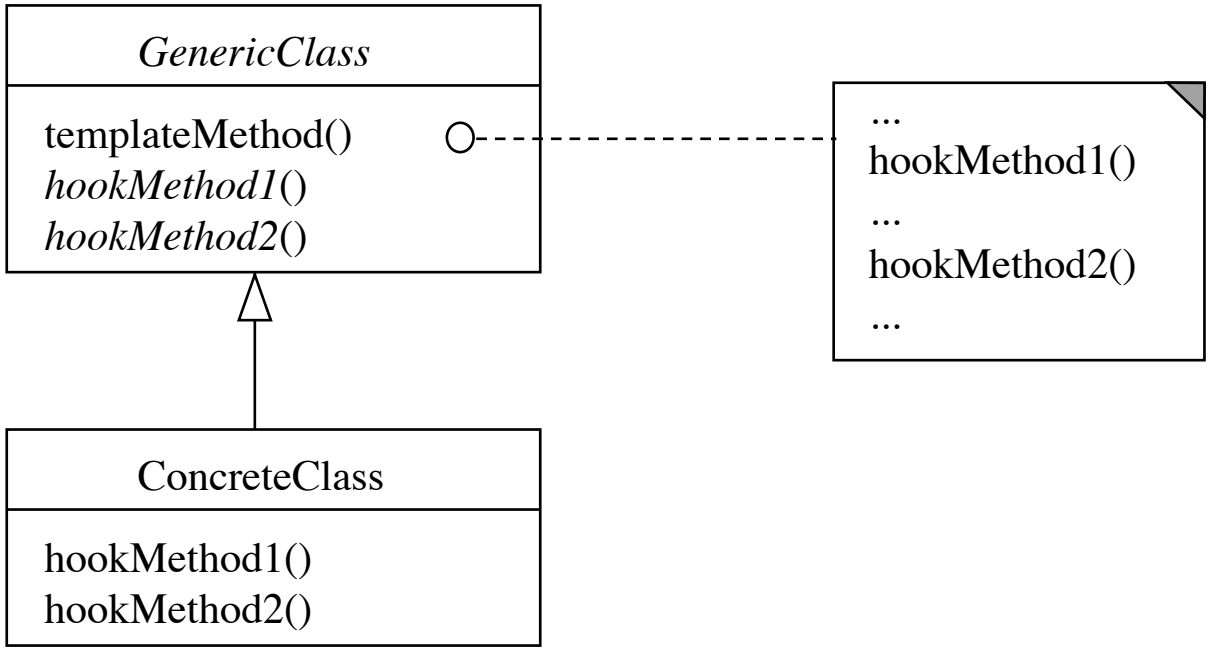
Karakteristika

- **Udvidelighed**
- **Ofte inversion af kontrol**
- **Designmønstre bruges som byggeblokke**

  Designmønstrene Template Method og Strategy bruges i næsten alle frameworks.

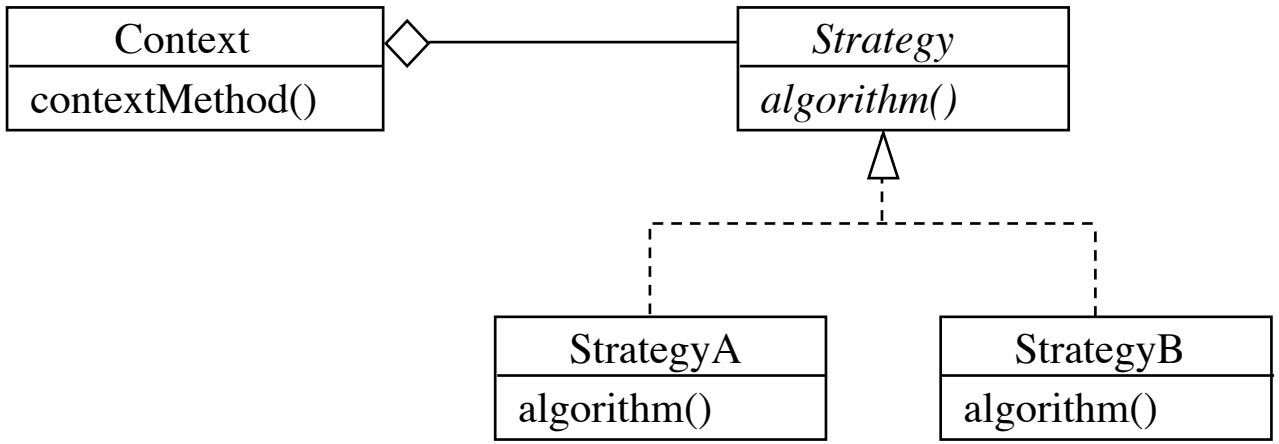
Designmønsteret Template Method

Struktur:



Designmønsteret Strategy

Struktur:



Strategy er en variant af *Template Method*: Ved *Strategy* er hook-metoderne placeret i separate klasser og kaldes ved delegering. Ved *Template Method* er hook-metoderne placeret i underklasser.

Krav til design af frameworks

(Booch, 1994)

- **Fuldstændighed** (opfylder alle relevante behov)
- **Tilpasningsevne** (let at tilpasse en given platform)
- **Effektivitet** (tid, plads, udviklingstid)
- **Sikkerhed** (håndtering af forkert brug)
- **Enkelhed** (let at overskue)
- **Udvidelighed** (nye klasser kan tilføjes)

Kollektioner

En **kollektion** er en samling af objekter.

Javas *collections* er et sæt af grænseflader og klasser, der understøtter lagring og genfinding af objekter i kollektioner.

Kollektionerne kan være baseret på forskellige datastrukturer og algoritmer (og dermed have forskellige tids- og pladskompleksitet).

En **kollektion** er et objekt, der fungerer som en *beholder* for andre objekter. Disse kaldes for kollektionens **elementer**.

Abstrakte kollektioner

En **pose** (bag, Collection) er en uordnet kollektion, som gerne må indeholde dubletter.

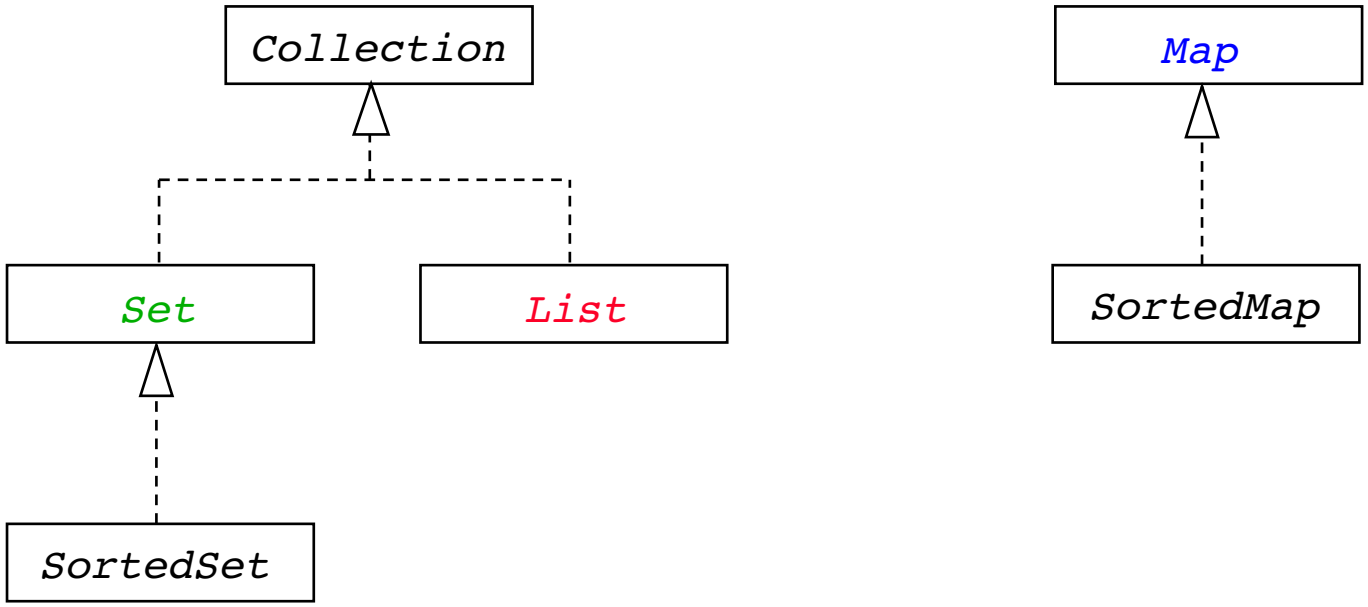
En **mængde** (Set) er en uordnet kollektion, som ikke må indeholde dubletter.

En **liste** (List) er en ordnet kollektion (sekvens), som gerne må indeholde dubletter.

En **afbildning** (Map) er en uordnet kollektion af nøgle-værdi-par. Nøglerne skal være unikke.

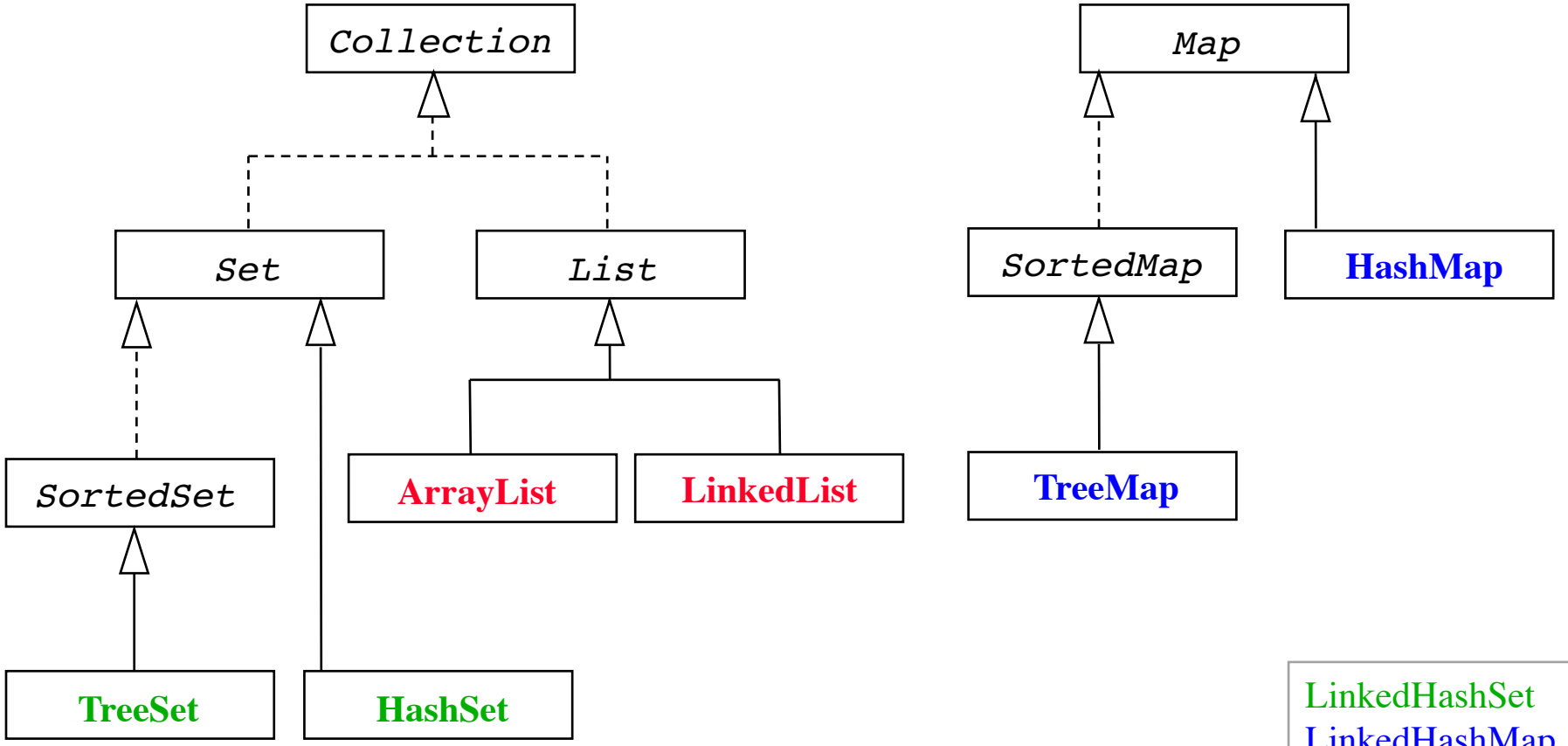
Grænseflader for kollektioner

`java.util.*`



Retningslinje for design: Maksimer ensartethed af de fælles aspekter af relaterede klasser og grænseflader

Konkrete kollektioner



LinkedHashSet
LinkedHashMap
IdentityHashMap

interface **Collection**

```
boolean add(Object o)
boolean addAll(Collection c)
void clear()
boolean contains(Object o)
boolean containsAll(Collection c)
boolean isEmpty()
Iterator iterator()
boolean remove(Object o)
boolean removeAll(Collection c)
boolean retainAll(Collection c)
int size()
Object[] toArray();
```

For primitive typer benyttes wrapper-klasser

`interface` **`Set`** `extends Collection`

Ingen nye metoder, men kontrakterne for

```
boolean add(Object o)
boolean addAll(Collection c)
```

er ændret (strammet), således at et objekt kun tilføjes, hvis der ikke i forvejen findes en dublet (dette afgøres med `equals`).

Eksempel på anvendelse af `Set`


```
Set set = new HashSet();
set.add("cat");
set.add("dog");
int n = set.size();
System.out.println("The set contains " + n + " elements");
if (set.contains("dog"))
    System.out.println("dog is in the set");
```

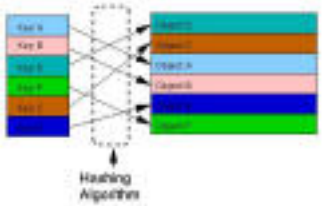
`interface List extends Collection`

Nye metoder:

```
void add(int i, Object o)
Object get(int i)
int indexOf(Object o)
int lastIndexOf(Object o)
ListIterator listIterator()
ListIterator listIterator(int i)
Object remove(int i)
Object set(int i, Object o)
List subList(int i, int j)
```

Ændrede kontrakter for `add(o)`, `addAll(c)` og `remove(o)`

interface **Map**




```
Object put(Object key, Object value)
Object get(Object key)
Object remove(Object key)

void clear()
boolean containsKey(Object key)
boolean containsValue(Object value)
boolean isEmpty()
void putAll(Map m)
int size()

Set keySet()
Collection values()
Set entrySet()
```

Eksempel på anvendelse af `Map`

(indholdsadresserbar tabel)

```
Map map = new HashMap();
map.put("cat", "kat");
map.put("dog", "hund");
Object val = map.get("dog");      // val er "hund"
map.remove("cat");
map.put("dog", "køter");
val = map.get("dog");             // val er "køter"
```

Gennemløb af kollektioner

Til gennemløb af kollektioner er defineret to grænseflader:

```
interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}

interface ListIterator extends Iterator {
    boolean hasPrevious();
    Object previous();
    void add(Object o);
    int previousIndex();
    int nextIndex();
    void set(Object o);
}
```

Konkrete iteratorer

En konkret iterator fås ved kald af `iterator()` fra grænsefladen `Collection`, eller ved kald af `listIterator()` fra grænsefladen `List`.

Gennemløb af en afbildning (3 syn):

Syn	Kald	Type
nøgler	`keySet().iterator()`	`Object`
værdier	`values().iterator()`	`Object`
indgange	`entrySet().iterator()`	`Map.Entry`

Ordning og sortering

Der er to måder at definere orden imellem objekter:

(1) Ved at implementere grænsefladen `Comparable` for klassen af objekter (**naturlig orden**).

(2) Ved at anvende et objekt fra en klasse, der implementerer grænsefladen `Comparator`.

Grænsefladen `Comparable`

```
interface Comparable {
    int compareTo(Object o);
}
```

Kontrakt for metoden `compareTo`:

Returner et heltal < 0, hvis `this` kommer før `o`.
Returner 0, hvis `this` hverken kommer før eller efter `o`.
Returner et heltal > 0, hvis `this` kommer efter `o`.

Mange af klasserne i JDK implementerer `Comparable`, bl.a. `String`.

Grænsefladen `Comparator`

```
interface Comparator {
    int compare(Object o1, Object o2);
}
```

Kontrakt for metoden `compare`:

Returner et heltal < 0, hvis `o1` kommer før `o2`.
Returner 0, hvis `o1` hverken kommer før eller efter `o2`.
Returner et heltal > 0, hvis `o1` kommer efter `o2`.

Eksempel på anvendelse af designmønstret Strategy.

`interface` **`SortedSet`** `extends Set`

Nye metoder:

```
Object first()
Object last()
SortedSet headSet(Object o)       // < o
SortedSet tailSet(Object o)       // ≥ o
SortedSet subSet(o1, o2)          // ≥ o1 && < o2
Comparator comparator()
```

En konkret implementering, f.eks. `TreeSet`, tilbyder mindst to konstruktører: en uden parametre, og en med en `Comparator` som parameter.

`interface` **`SortedMap`** `extends Map`

Nye metoder:

```
Object firstKey()
Object lastKey()
SortedMap headMap(Object o)       // < o
SortedMap tailMap(Object o)       // ≥ o
SortedMap subMap(o1, o2)          // ≥ o1 && < o2
Comparator comparator()
```

En konkret implementering, f.eks. `TreeMap`, tilbyder mindst to konstruktører: en uden parametre, og en med en `Comparator` som parameter

Klassen `Collections`

```
public class Collections {
    public static void sort(List l);
    public static void sort(List l, Comparator comp);

    public static int binarySearch(List l, Object key);
    public static int binarySearch(List l, Object key,
                                   Comparator comp);

    public static Object min(Collection c);
    public static Object min(Collection c, Comparator c);
    public static Object max(Collection c);
    public static Object max(Collection c, Comparator c);

    public static void reverse(List l);
    public static void shuffle(List l);

    public static Comparator reverseOrder();
}
```

Klassen `Arrays`

```
public class Arrays {
    public static List asList(Object[] a);

    public static void sort(type[] a);
    public static void sort(Object[] a, Comparator comp);
    public static void sort(type[] a, int from, int to);
    public static void sort(Object[] a, int from, int to,
                            Comparator comp);

    public static int binarySearch(type[] a, type key);
    public static int binarySearch(Object[] a, type key,
                                   Comparator comp);

    public static void fill(type[] a, type value);
    public static void fill(type[] a, int from, int to, type value);

    public static boolean equals(type[] a1, type[] a2);
}
```

Valg af konkret kollektion for en mængde

Hvis elementerne skal kunne gennemløbes i sorteret rækkefølge, benyttes `TreeSet`; ellers `HashSet`.

Forudsætninger:
Klassen af elementer skal implementere metoden `equals`.

Hvis `TreeSet` benyttes, skal klassen af elementer implementere grænsefladen `Comparable`; eller `TreeSet` skal instantieres med et `Comparator`-objekt.

Hvis `HashSet` benyttes, skal klassen af elementer implementere metoden `hashCode`.

Valg af konkret kollektion for en liste

Hvis opslag på indices er hyppig, og hvis længden af listen kun varierer lidt, benyttes `ArrayList`; ellers `LinkedList`.

Valg af konkret kollektion for en afbildning

Hvis nøglerne skal kunne gennemløbes i sorteret rækkefølge, benyttes `TreeMap`; ellers `HashMap`.

Forudsætninger:

Klasserne af nøgler og værdier skal implementere metoden `equals`.

Hvis `HashMap` benyttes, skal klassen af nøgler implementere metoden `hashCode`.

Hvis `TreeMap` benyttes, skal klassen af nøgler implementere grænsefladen `Comparable`; eller `TreeMap` skal instantieres med et `Comparator`-objekt.

Opgave 1:

Skriv et program, der indlæser en tekst og udskriver det totale antal ord samt antallet af forskellige ord i teksten.

Optælling af ord

```
public class CountWords {
    public static void main(String[] args) {
        Set words = new HashSet();
        int count = 0;
        try {
            BufferedReader in = new BufferedReader(
                                    new FileReader(args[0]));
            String line, delim = " \t\n.,:;?!-/()[]\"\'";
            while ((line = in.readLine()) != null) {
                StringTokenizer st = new StringTokenizer(line, delim);
                while (st.hasMoreTokens()) {
                    count++;
                    words.add(st.nextToken().toLowerCase());
                }
            }
        } catch (IOException e) {}
        System.out.println("Total number of words: " + count);
        System.out.println("Number of different words: " + words.size());
    }
}
```

Opgave 2:

Skriv et program, der indlæser en tekst og for hvert ord udskriver, hvor mange gange, det forekommer i teksten.

Optælling af ordforekomster

```
public class WordFrequency {
    static class Count {
        Count(String word, int value) {
            this.word = word;
            this.value = value;
        }

        String word;
        int value;
    }

    public static void main(String[] args) { ... }
}
```

fortsættes

```
public static void main(String[] args) {
    Map words = new HashMap();
    try {
        BufferedReader in = new BufferedReader(
                                new FileReader(args[0]));
        String line, delim = " \t\n.,:;?!-/()[]\"\'";
        while ((line = in.readLine()) != null) {
            StringTokenizer st = new StringTokenizer(line, delim);
            while (st.hasMoreTokens()) {
                String word = st.nextToken().toLowerCase();
                Count count = (Count) words.get(word);
                if (count == null)
                    words.put(word, new Count(word, 1));
                else
                    count.value++;
            }
        }
        in.close();
    } catch (IOException e) {}
    // ... print the word counts
}
```

fortsættes

```
Iterator iter = words.entrySet().iterator();
while (iter.hasNext()) {
    Map.Entry entry = (Map.Entry) iter.next();
    String word = (String) entry.getKey();
    Count count = (Count) entry.getValue();
    System.out.println(word +
                       (word.length() < 8 ? "\t\t" : "\t") +
                       count.value);
    }
}
```

Eksempel på udskrift

(A. Lincoln: The Gettisburg Address, 1863)

```
cause                   1
say                     1
portion                 1
nation                  5
note                    1
their                   1
great                   3
created                 1
power                   1
place                   1
engaged                 1
altogether              1
....
```

Opgave 3:

Skriv et program, der indlæser en tekst og for hvert ord udskriver, hvor mange gange, det forekommer i teksten.

Ordene skal udskrives i alfabetisk rækkefølge.

Udskrivning efter ordenes alfabetiske rækkefølge

Eneste ændring er, at

```
Map words = new HashMap();
```

erstattes med

```
Map words = new TreeMap();
```

Opgave 4:

Skriv et program, der indlæser en tekst og for hvert ord udskriver, hvor mange gange, det forekommer i teksten.

Ordene skal udskrives i omvendt alfabetisk rækkefølge.

Udskrivning i omvendt alfabetisk rækkefølge

Følgende klasse erklæres

```
public class StringComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        return -((String) o1).compareTo(o2);
    }
}
```

og sætningen

```
Map words = new TreeMap();
```

erstattes med

```
Map words = new TreeMap(new StringComparator());
```

Udskrivning i omvendt alfabetisk rækkefølge (fortsat)

Endnu simplere er det dog at benytte metoden `reverseOrder()` fra klassen `Collections`.

Erstat sætningen

```
Map words = new TreeMap();
```

med

```
Map words = new TreeMap(Collections.reverseOrder());
```

Opgave 5:

Skriv et program, der indlæser en tekst og for hvert ord udskriver, hvor mange gange, det forekommer i teksten.

Ordene skal udskrives i aftagende frekvensrækkefølge.

Udskrivning i aftagende frekvensrækkefølge

Følgende klasse erklæres

```
public class CountComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        return -((Count) o1).value - (Count) o2).value);
    }
}
```

og ...

fortsættes

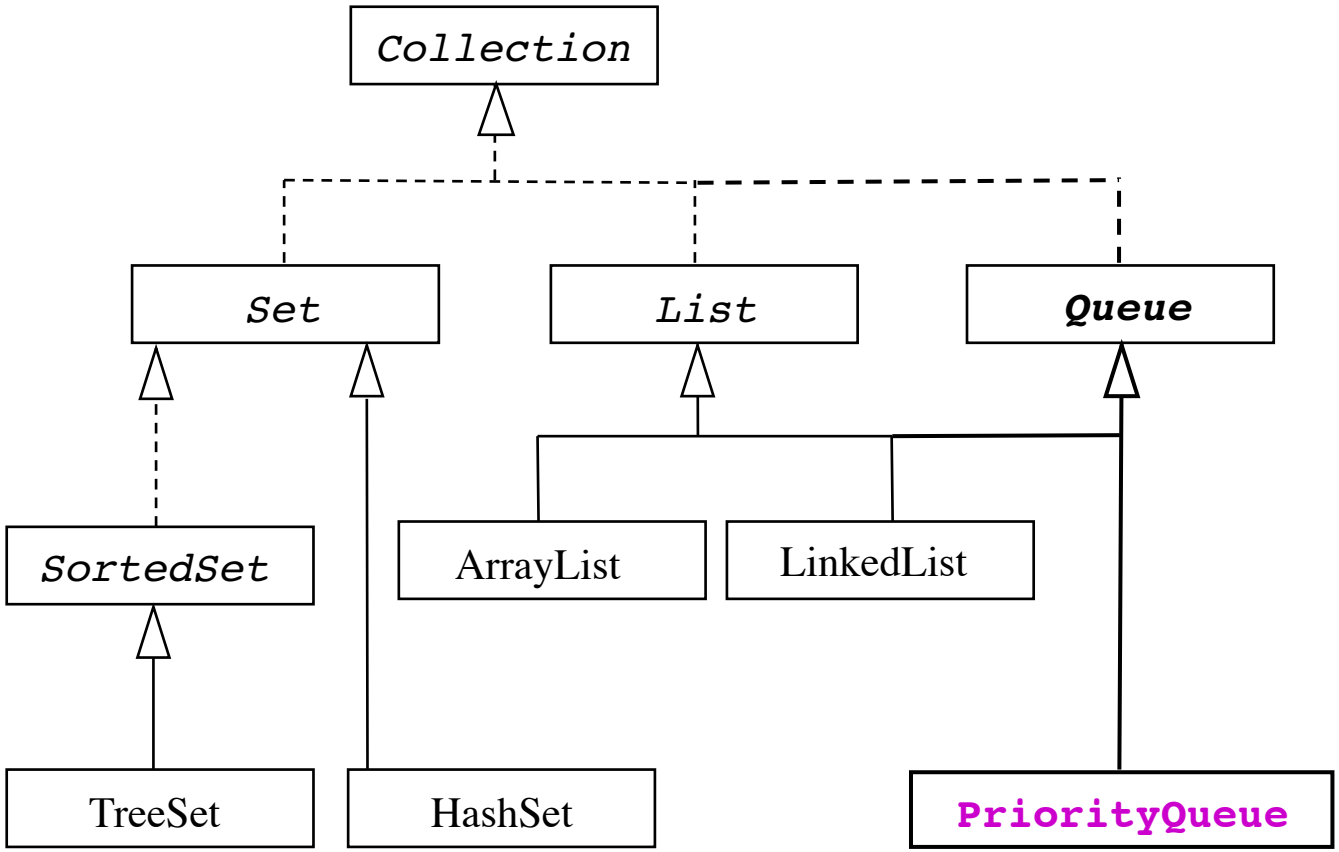
Udskrivningen foretages således

```
List list = new ArrayList(words.values());
Collections.sort(list, new CountComparator());

Iterator itr = list.iterator();
while (iter.hasNext()) {
    Count count = (Count) iter.next();
    String word = count.word;
    System.out.println(word +
                       (word.length() < 8 ? "\t\t" : "\t") +
                       count.value);
}
```

Gennemløbet kan simplificeres:

```
for (Count count : list) {
    String word = count.word;
    System.out.println(...);
}
```

Nye kollektioner i Java 5.0



`interface` **`Queue`** `extends Collection`

Nye metoder:

```
boolean offer(Object o)   // insert an element if possible at the tail of the queue
Object element()          // return, but do not remove, the head of the queue (*)
Object peek()             // return, but do not remove, the head of the queue
Object poll()             // remove and return the head of the queue
Object remove()           // remove and return the head of the queue (*)
```

*: may throw an exception

`class` **`PriorityQueue`** `implements Queue`

Ordner sine `Comparable`-objekter efter deres `compareTo`-metoder - eller efter et angivet `Comparator`-objekt.

Generiske typer

I Java 5.0 kan klasser og metoder forsynes med en eller flere typeparametre.

Fordele:

- Større typesikkerhed
  (check kan foretages på oversættelsestidspunktet)
- Øget læsbarhed
  (unødig typekonvertering undgås)

Klassen `ArrayList` med en generisk typeparameter

```
public class ArrayList<E> implements List {
    public boolean add(E o);
    public E get(int i);
    public boolean addAll(Collection<? extends E> c);
    ...
}
```

Navnet på typeparameteren, her `E`, kan vælges frit. Alternativ: `ElementType`.

Notationen `<? extends E>` angiver en vilkårlig undertype af `E`.
Tegnet `?` kaldes for et jokertegn (wildcard) og betegner en ukendt type.

Eksempel på anvendelse

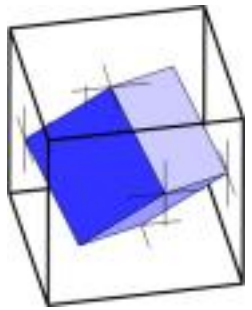
```
ArrayList<String> names = new ArrayList<String>();

names.add(”Bo Hamburger”);     // OK
names.add(new Integer(21));    // Compile time error

String name = names.get(0);    // OK without cast

for (String name : names)
   System.out.println(name.toUpperCase());
```

Klassen `HashMap` med to generiske typeparametre

```
public class HashMap<K,V> implements Map {
    public V put(K key, V value);
    public V get(Object key);
    public Set<Map.Entry<K,V>> entrySet();
    ...
}
```

Læs mere om generiske typer via kursets hjemmeside:
”Generics in the Java Programming Language” af G. Bracha

Autoboxing/unboxing i Java 5.0



Koden

```
ArrayList<Integer> arr = new ArrayList<Integer>();
arr.add(new Integer(21));
int val = arr.get(0).intValue();
```

kan erstattes med nedenstående simplere kode

```
ArrayList<Integer> arr = new ArrayList<Integer>();
arr.add(21);
int val = arr.get(0);
```

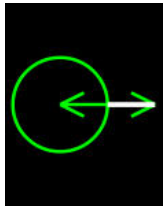
Autoboxing/unboxing

Autoboxing:
Hvis en værdi af en simpel type angives, hvor et objekt af den tilsvarende wrapper-klasse forventes, vil oversætteren indsætte en kald af en konstruktør for wrapper-klassen.

Unboxing:
Hvis et objekt af en wrapper-klasse angives, hvor en værdi af den tilsvarende simple type forventes, vil oversætteren indsætte et kald af en metode, der foretager konvertering til den simple type.

Input/output

`java.io.*`



Javas I/O-framework understøtter to typer IO:

- *Sekventiel* tilgang til filer ved hjælp af strømme.
  En **strøm** (stream) er en sekvens af bytes eller tegn.
  En strøm kan åbnes for enten læsning eller skrivning,
  men ikke begge dele samtidigt.

- *Tilfældig* tilgang (random access) til filer.
  Tillader at en fil kan åbnes for både læsning og skrivning.

To slags strømme

(1) **Byte-strømme**, der understøtter læsning og skrivning af en hvilken som helst type i binært format.

```
00001011 00010001 00001001 11010001
```

(2) Tegn-strømme, der understøtter læsning og skrivning af tekst

	H	*e*	*l*	*s*	*g*	*a*	*u*	*n*	

Byte-strømme

To abstrakte klasser `InputStream` og `OutputStream` understøtter læsning og skrivning af en enkelt byte eller et byte-array.

Metoder i `InputStream`:

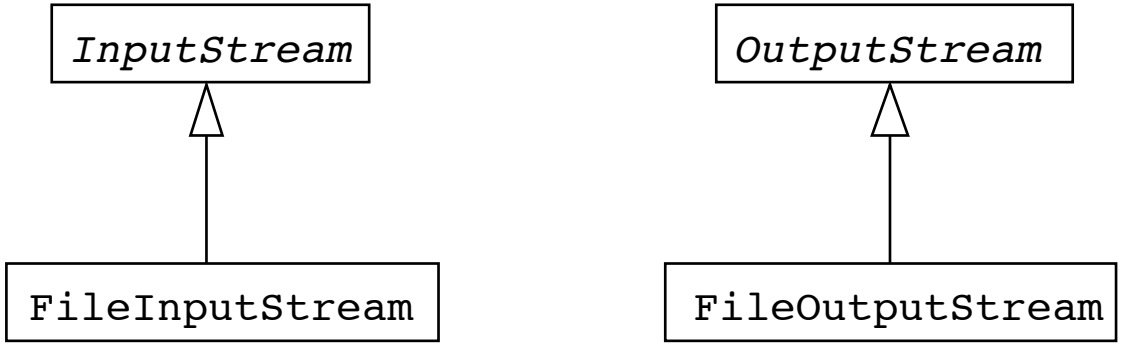
```
int read()
int read(byte[] ba)
int read(byte[] ba, int off, int len)
skip(int n)
close()
```

Metoder i `OutputStream`:

```
void write(int b)
void write(byte[] ba)
void write(byte[] ba, int off, int len)
close()
```

Konkrete byte-strømme

Klasserne `FileInputStream` og `FileOutputStream` konkretiserer de abstrakte klasser `InputStream` og `OutputStream`.



Konstruktører:

```
FileInputStream(String fileName)

FileOutputStream(String fileName)
FileOutputStream(String fileName, boolean append)
```

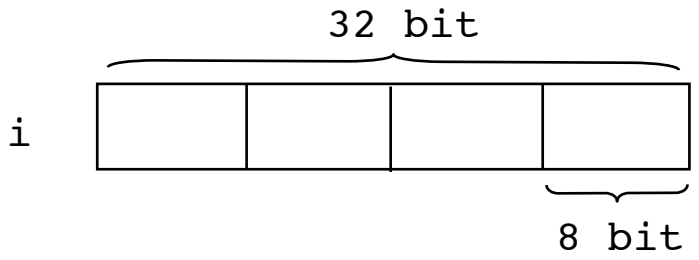
Skrivning af en 2-dimensional matrix ved hjælp af byte-orienteret I/O

```
public class WriteMatrix1 {
    static double[][] data = {{7.4, 2.1, 5.5},
                              {2.3, 3.2, 4.0}};

    public static void main(String[] args) {
        int row = data.length, col = data[0].length;
        try {
            OutputStream out = new FileOutputStream(args[0]);
            writeInt(row, out);
            writeInt(col, out);
            for (int i = 0; i < row; i++)
                for (int j = 0; j < col; j++)
                    writeDouble(data[i][j], out);
            out.close();

        } catch (IOException e) {}
    }
```

fortsættes

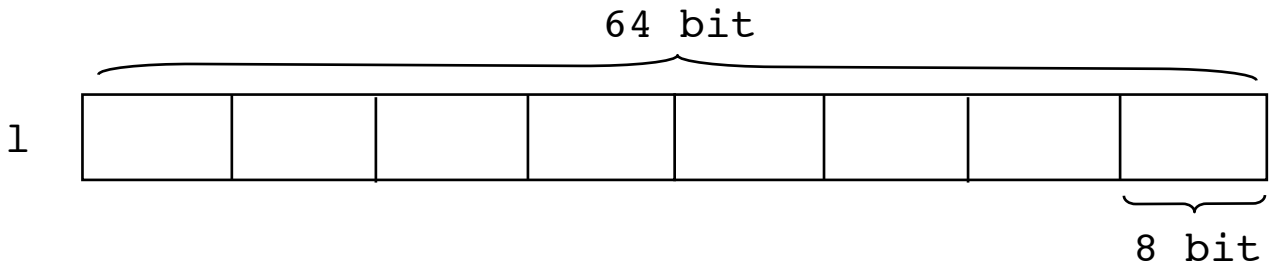
Metoden `writeInt`

```
public static void writeInt(int i, OutputStream out)
                    throws IOException {
    byte[] buf = new byte[4];
    for (int k = 3; k >= 0; k--) {
        buf[k] = (byte) (i & 0xFF);
        i >>>= 8;
    }
    out.write(buf);
}
```



Metoden `writeDouble`

```
public static void writeDouble(double d, OutputStream out)
                    throws IOException {
    byte[] buf = new byte[8];
    long l = Double.doubleToLongBits(d);
    for (int k = 7; k >= 0; k--) {
        buf[k] = (byte) (l & 0xFF);
        l >>>= 8;
    }
    out.write(buf);
}
```



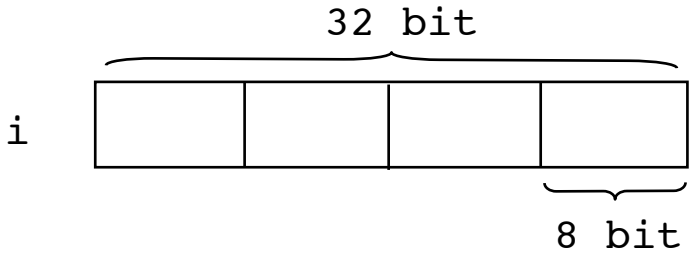
Læsning af en 2-dimensional matrix ved hjælp af byte-orienteret I/O

```
public class ReadMatrix1 {
    public static void main(String[] args) {
        try {
            InputStream in = new FileInputStream(args[0]);
            int row = readInt(in);
            int col = readInt(in);
            double[][] data = new double[row][col];
            for (int i = 0; i < row; i++) {
                for (int j = 0; j < col; j++) {
                    data[i][j] = readDouble(in);
                    System.out.println(data[i][j]);
                }
            }
            in.close();
        } catch (IOException e) {}
    }
```

fortsættes

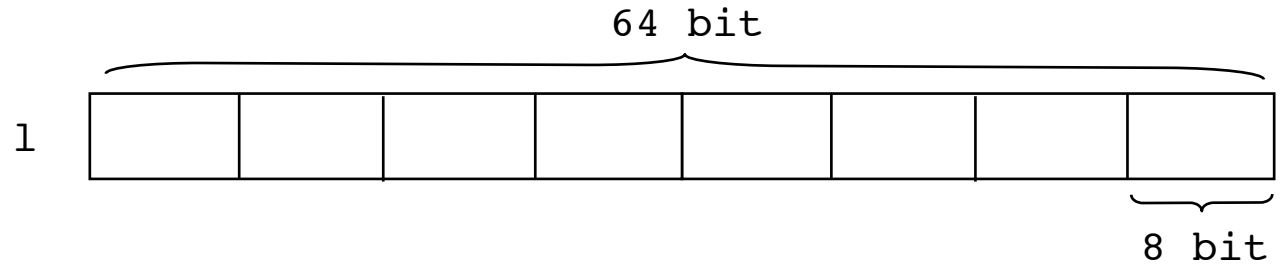
Metoden `readInt`

```
public static int readInt(InputStream in)
                    throws IOException {
    byte[] buf = new byte[4];
    in.read(buf);
    int i = 0;
    for (int k = 0; k < 4; k++) {
        i <<= 8;
        i += buf[k];
    }
    return i;
}
```



Metoden `readDouble`

```
public static double readDouble(InputStream in)
                        throws IOException {
    byte[] buf = new byte[8];
    in.read(buf);
    long l = 0;
    for (int k = 0; k < 8; k++) {
        l <<= 8;
        l += buf[k];
    }
    return Double.longBitsToDouble(l);
}
```



Data-strømme

To grænseflader `DataInput` og `DataOutput` understøtter byte-orienteret læsning og skrivning af data af Javas primitive typer.

Metoder i `DataInput`:

```
boolean readBoolean()
byte readByte()
char readChar()
double readDouble()
float readFloat()
int readInt()
long readLong()
short readShort()
String readUTF()
```

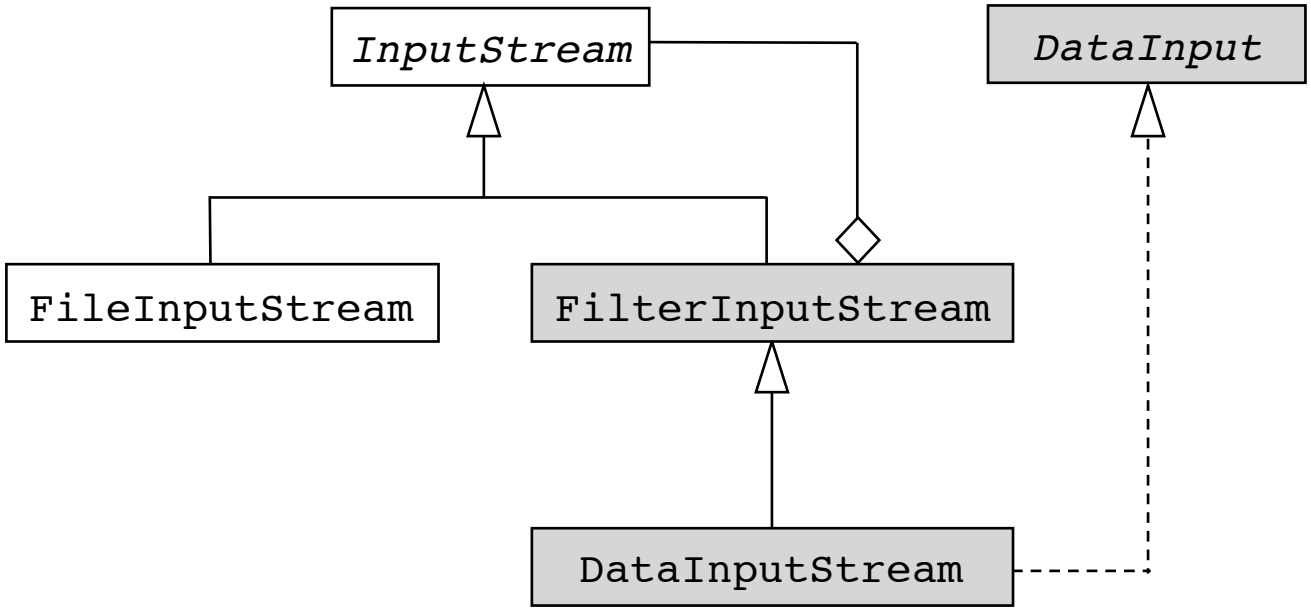
Metoder i `DataOutput`:

```
void writeBoolean(boolean b)
void writeByte(int b)
void writeChar(char c)
void writeDouble(double d)
void writeFloat(float f)
void writeInt(int i)
void writeLong(long l)
void writeShort(short s)
void writeUTF(String s)
```

UTF-8 (**U**nicode **T**ransformation **F**ormat): tabsfri 8-bitskodning af Unicode-tegn

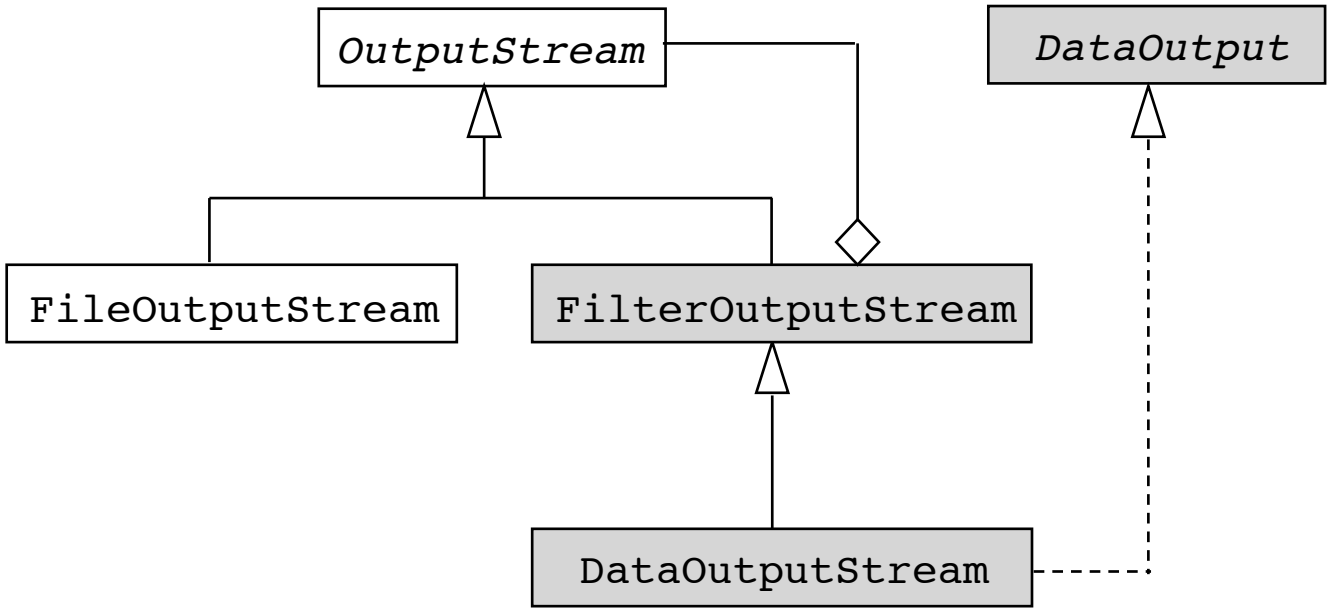
Konkrete data-strømme

Klasserne `DataInputStream` og `DataOutputStream` implementerer henholdsvis grænsefladen `DataInput` og `DataOutput`.



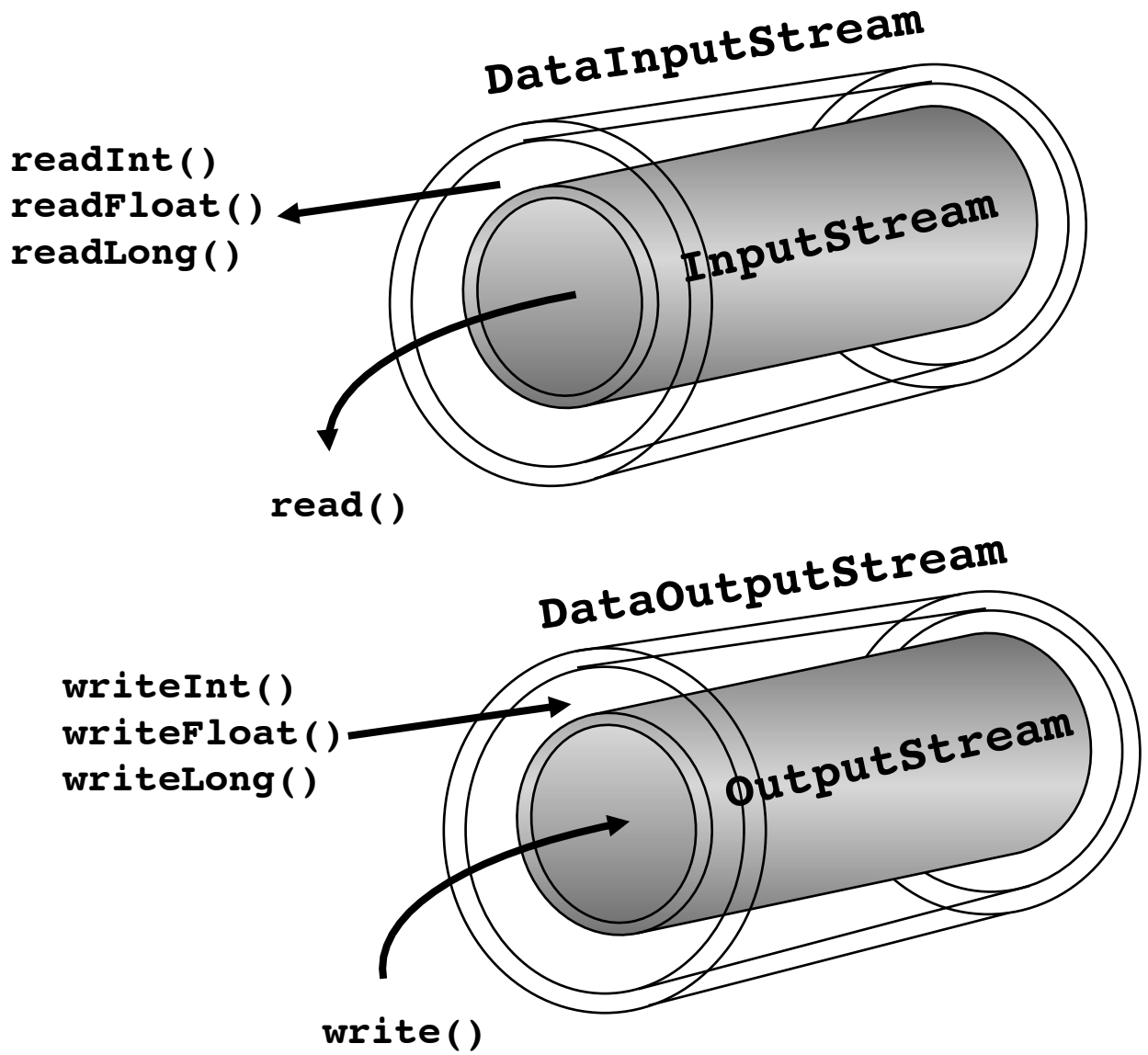
Konstruktør:

```
DataInputStream(InputStream in)
```



Konstruktør:

```
DataOutputStream(OutputStream out)
```


DataInputStream
readInt()
readFloat()
readLong()
InputStream
read()
DataOutputStream
writeInt()
writeFloat()
writeLong()
OutputStream
write()

Skrivning af en 2-dimensional matrix ved hjælp af `DataOutputStream`

Linjen

```
FileOutputStream out = new FileOutputStream(args[0]);
```

erstattes med

```
DataOutputStream out =
    new DataOutputStream(new FileOutputStream(args[0]));
```

og udskrivning foretages således

```
    out.writeInt(row);
    out.writeInt(col);
...
    out.writeDouble(data[i][j]);
```

Læsning af en 2-dimensional matrix ved hjælp af `DataInputStream`

Linjen

```
FileInputStream in = new FileInputStream(args[0]);
```

erstattes med

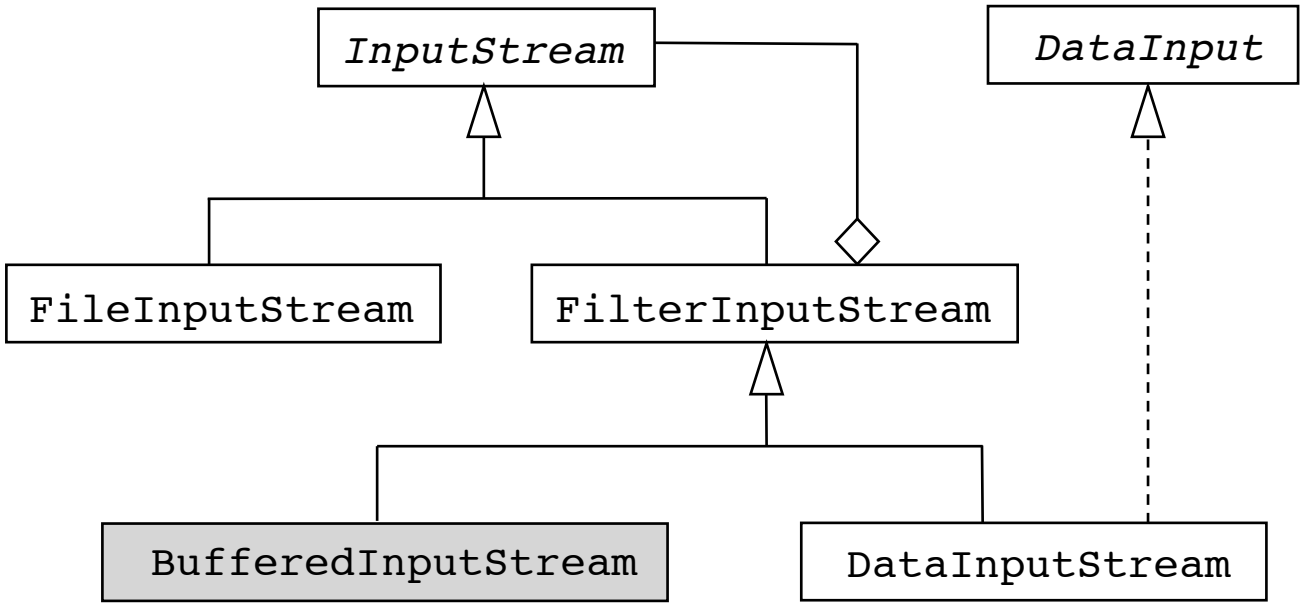
```
DataInputStream in =
    new DataInputStream(new FileInputStream(args[0]));
```

og indlæsning foretages således

```
    row = in.readInt();
    col = in.readInt();
...
    data[i][j] = in.readDouble();
```

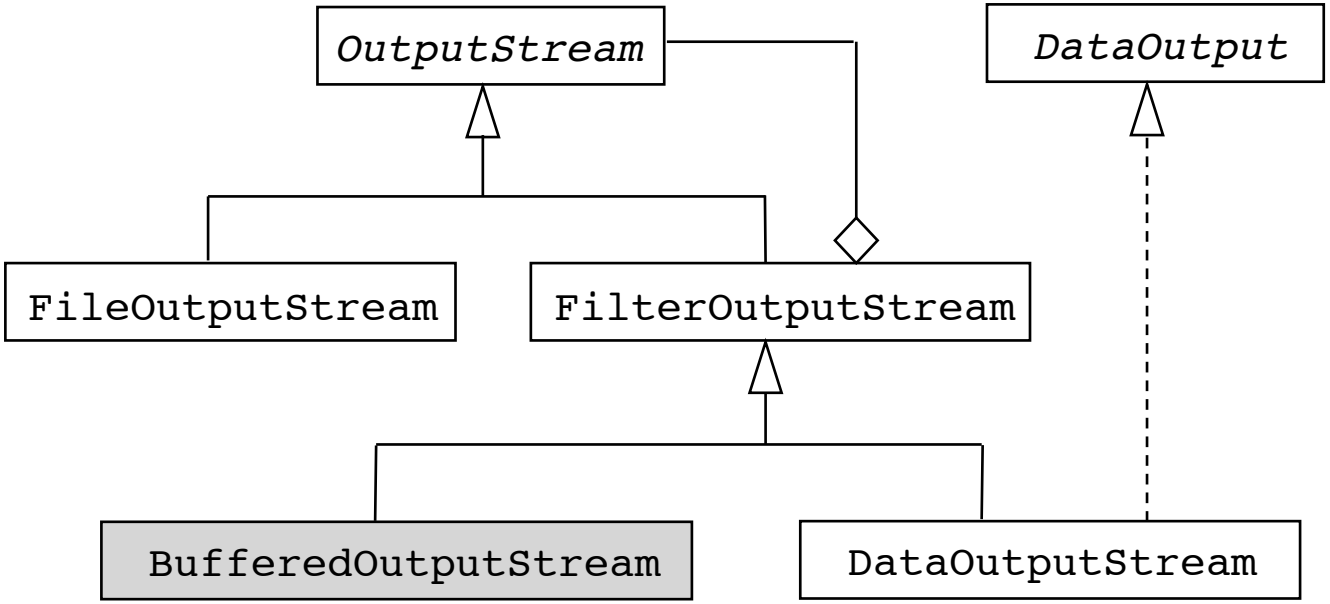
Buffered input/output-strømme

Klasserne `BufferedInputStream` og `BufferedOutputStream` er filterstrømme, der understøtter byte-orienteret, "buffered" læsning og skrivning.



Konstruktør:

```
BufferedInputStream(InputStream in)
```



Konstruktør:

```
BufferedOutputStream(OutputStream out)
```

Skrivning af en 2-dimensional matrix ved hjælp af `BufferedOutputStream`

Linjen

```
FileOutputStream out = new FileOutputStream(args[0]);
```

erstattes med

```
DataOutputStream out =
    new DataOutputStream(
        new BufferedOutputStream(
            FileOutputStream(args[0])));
```

Læsning af en 2-dimensional matrix ved hjælp af `BufferedOutputStream`

Linjen

```
FileInputStream in = new FileInputStream(args[0]);
```

erstattes med

```
DataInStream in =
    new DataInputStream(
        new BufferedInputStream(
            FileInputStream(args[0])));
```

Serialisering

Java understøtter objekt-serialisering.

Dette indbefatter to processer:

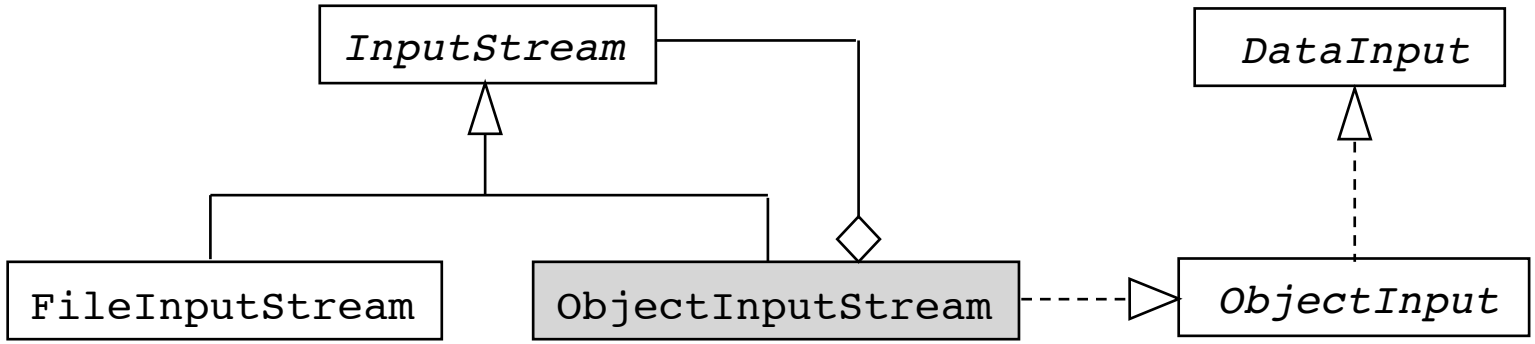
Serialisering: At skrive et objekt og alle de objekter, som refereres af objektet (direkte såvel som indirekte) til en strøm.

Deserialisering: At gendanne et serialiseret objekt og alle de objekter, som refereres af objektet (direkte såvel som indirekte).

Kun instanser af klasser, der implementerer grænsefladen `Serializable`, vil blive serialiseret.

Objekt-strømme

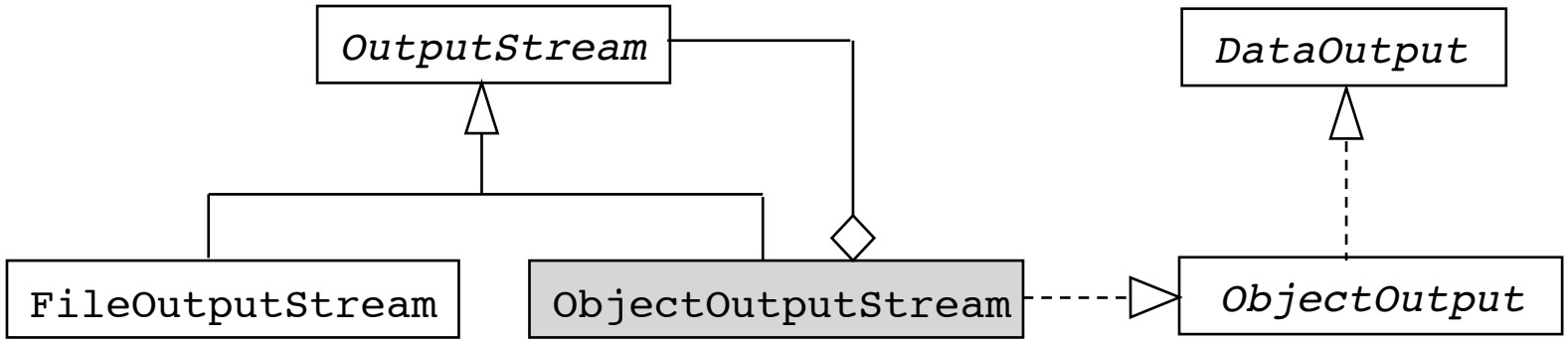
Klasserne `ObjectInputStream` og `ObjectOutputStream` er strømme, der understøtter serialisering og deserialisering.



Konstruktør:

```
ObjectInputStream(InputStream in)
```

Læsemetode: `readObject` fra grænsefladen `ObjectInput`



Konstruktør:

```
ObjectOutputStream(OutputStream out)
```

Skrivemetode: `writeObject` fra grænsefladen `ObjectOutput`

Skrivning af en 2-dimensional matrix ved hjælp af serialisering

```
public class WriteMatrix4 {
    static double[][] data = {{7.4, 2.1, 5.5},
                              {2.3, 3.2, 4.0}};

    public static void main(String[] args) {
        try {
            ObjectOutputStream out =
                new ObjectOutputStream(
                    new FileOutputStream(args[0]));
            out.writeObject(data);
            out.close();
        } catch (IOException e) {}
    }
}
```

Læsning af en 2-dimensional matrix ved hjælp af deserialisering

```
public class ReadMatrix4 {
    public static void main(String[] args) {
        try {
            ObjectInputStream in =
                new ObjectInputStream(
                    new FileInputStream(args[0]));
            double[][] data = (double[][]) in.readObject();
            in.close();
        } catch (IOException e) {}
          catch (ClassNotFoundException e) {}
    }
}
```

Designmønsteret Decorator

Kategori:

Strukturelt designmønster

Hensigt:

Dynamisk at tilføre et objekt en ekstra funktionalitet.
Decorator er et fleksibelt alternativ til nedarvning

Anvendelse:

- For at tilføre objekter ekstra funktionalitet, uden at andre objekter berøres
- For at funktionalitet kan fjernes
- Når udvidelse ved nedarvning er upraktisk, fordi der vil være behov for mange underklasser

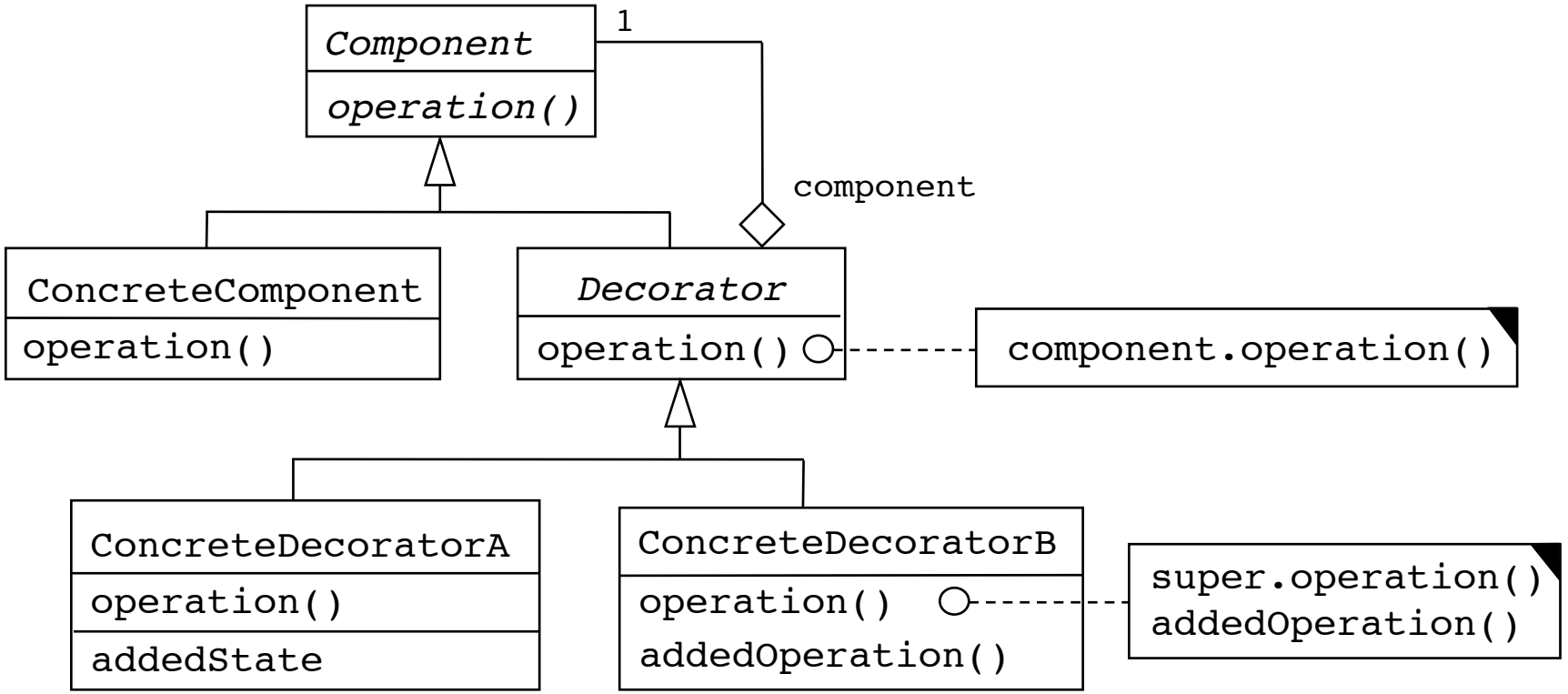
Andre navne:

Filter, Wrapper

Designmønsteret Decorator

(fortsat)

Struktur:



Designmønsteret Decorator

(fortsat)

Deltagere:

Component (f.eks. `InputStream`), der definerer grænsefladen for objekter, der kan få deres funktionalitet udvidet dynamisk

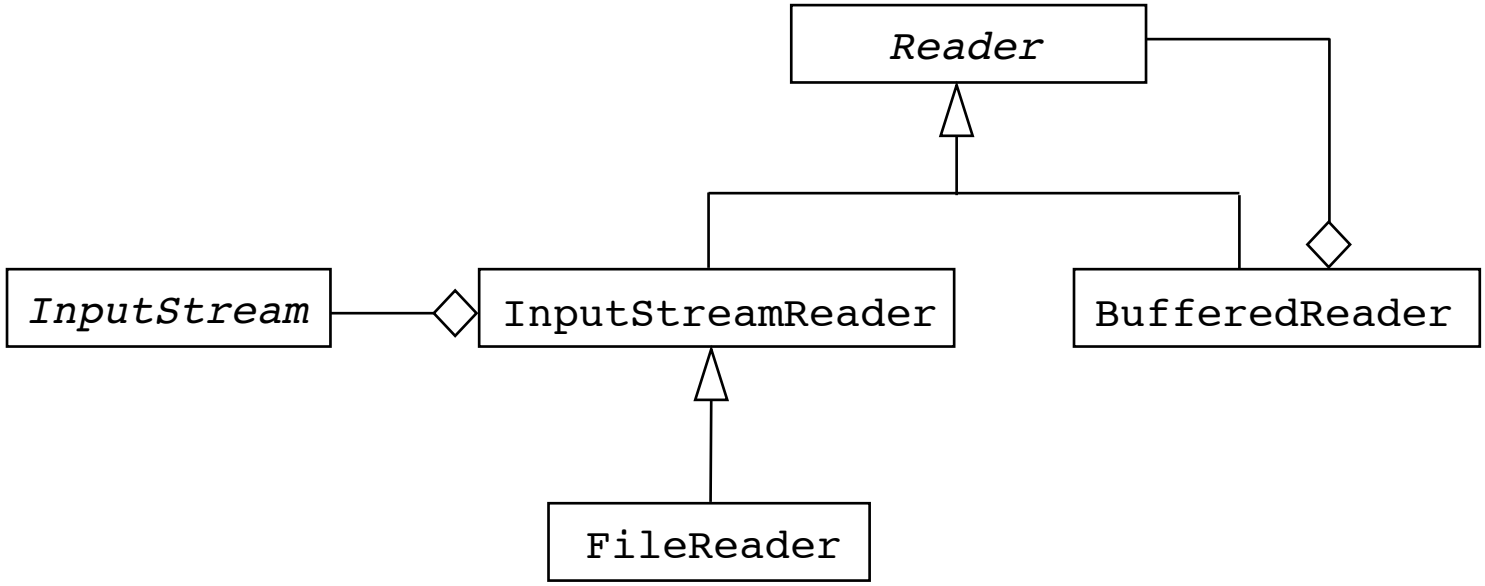
ConcreteComponent (f.eks. `FileInputStream`), der definerer en klasse af objekter, der kan få deres funktionalitet udvidet

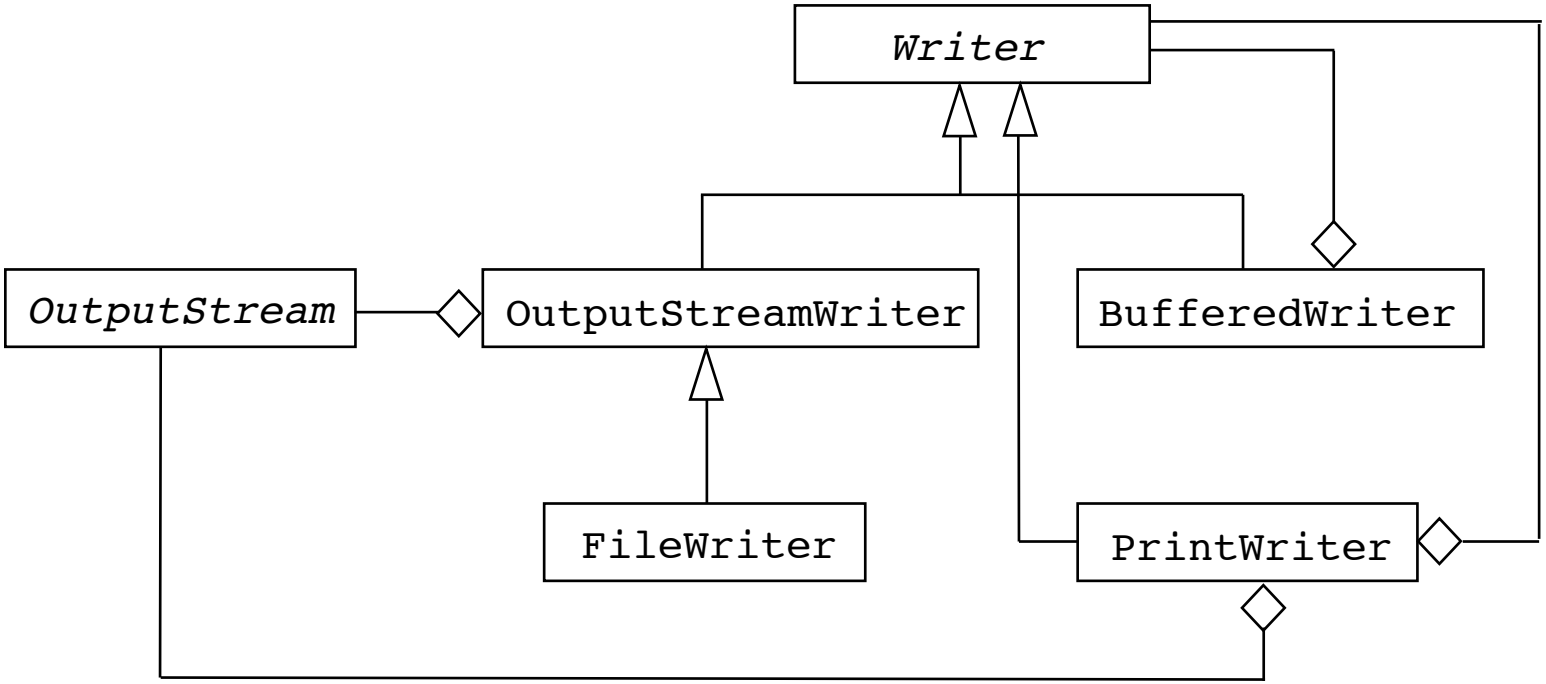
Decorator (f.eks. `FilterInputStream`), der har en reference til et *Component*-objekt og definerer en grænseflade, der overholder *Component*-grænsefladen.

ConcreteDecorator (f.eks. `DataInputStream` og `BufferedInputStream`), som tilfører en komponent ekstra funktionalitet.

Tegn-strømme

De to abstrakte klasser `Reader` og `Writer` understøtter læsning og skrivning af et enkelt tegn eller et tegn-array.




Writer
OutputStream
OutputStreamWriter
BufferedWriter
FileWriter
PrintWriter

Metoder i `Reader`:

```
int read()
int read(char[] ca)
int read(char[] ca, int off, int len)
close()
```

Metoder i `Writer`:

```
void write(int c)
void write(char[] ca)
void write(char[] ca, int off, int len)
close()
```

Konstruktører:

```
InputStreamReader(InputStream in)
InputStreamReader(InputStream in, String encoding)
OutputStreamWriter(OutputStream out)
OutputStreamWriter(OutputStream out, String encoding)

FileReader(String fileName)
FileWriter(String fileName)
FileWriter(String fileName, boolean append)

BufferedReader(Reader reader)
BufferedWriter(Writer writer)

PrintWriter(Writer writer)
```

Metode i `BufferedReader`:

```
String readln()
```

Metoder i `PrintWriter`:

```
void print(anyType v)
void println(anyType v)
```

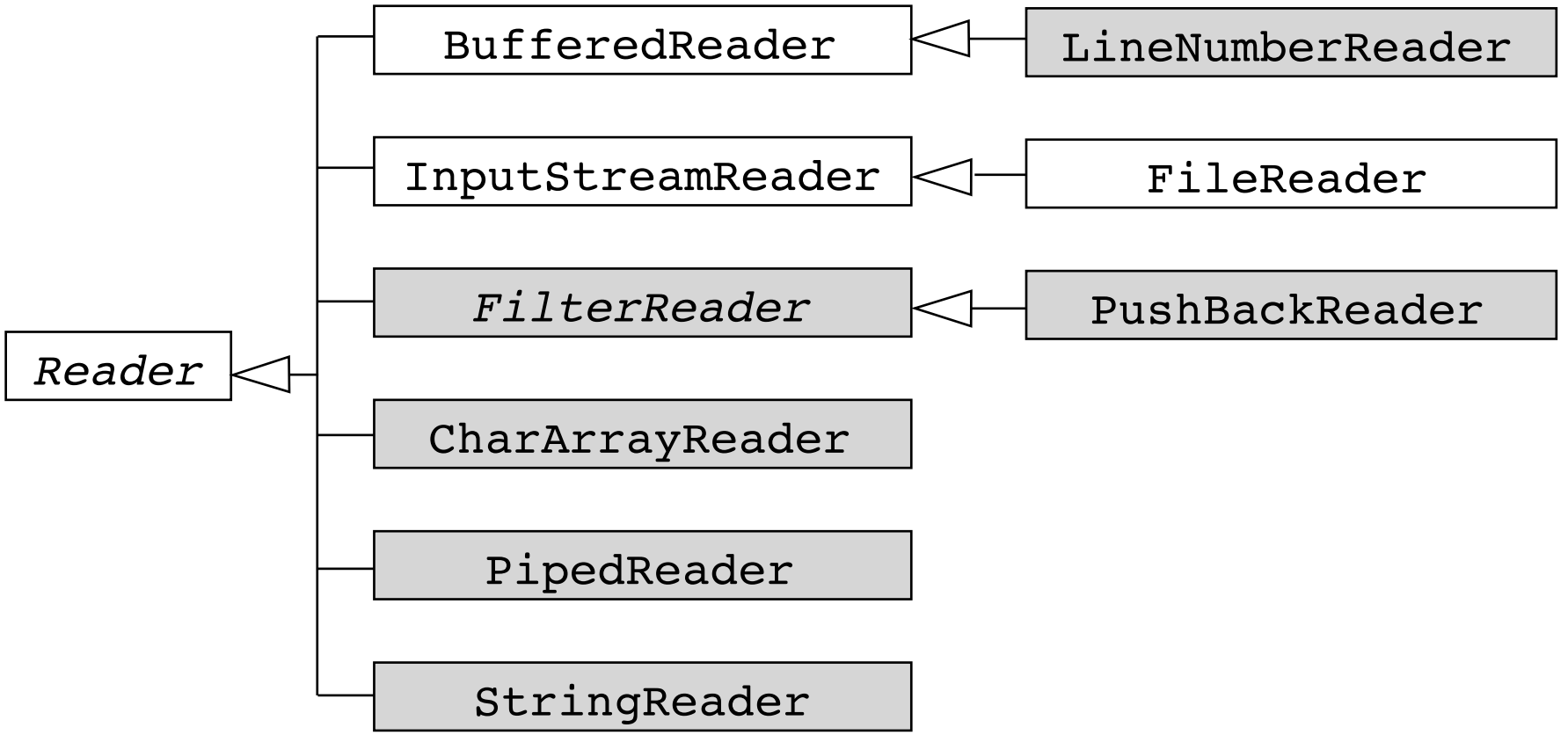
Skrivning af en 2-dimensional matrix ved tegnorienteret I/O

```
public class WriteMatrix5 {
    static double[][] data = {{ 7.4, 2.1, 5.5},
                              { 2.3, 3.2, 4.0}};

    public static void main(String[] args) {
        int row = data.length, col = data[0].length;
        try {
            PrintWriter out =
                new PrintWriter(
                    new BufferedWriter(
                        new FileWriter(args[0])));
            out.println(row);
            out.println(col);
            for (i = 0; i < row; i++)
                for (j = 0; j < col; j++)
                    out.println(data[i][j]);
            out.close();
        } catch (IOException e) {}
    }
}
```

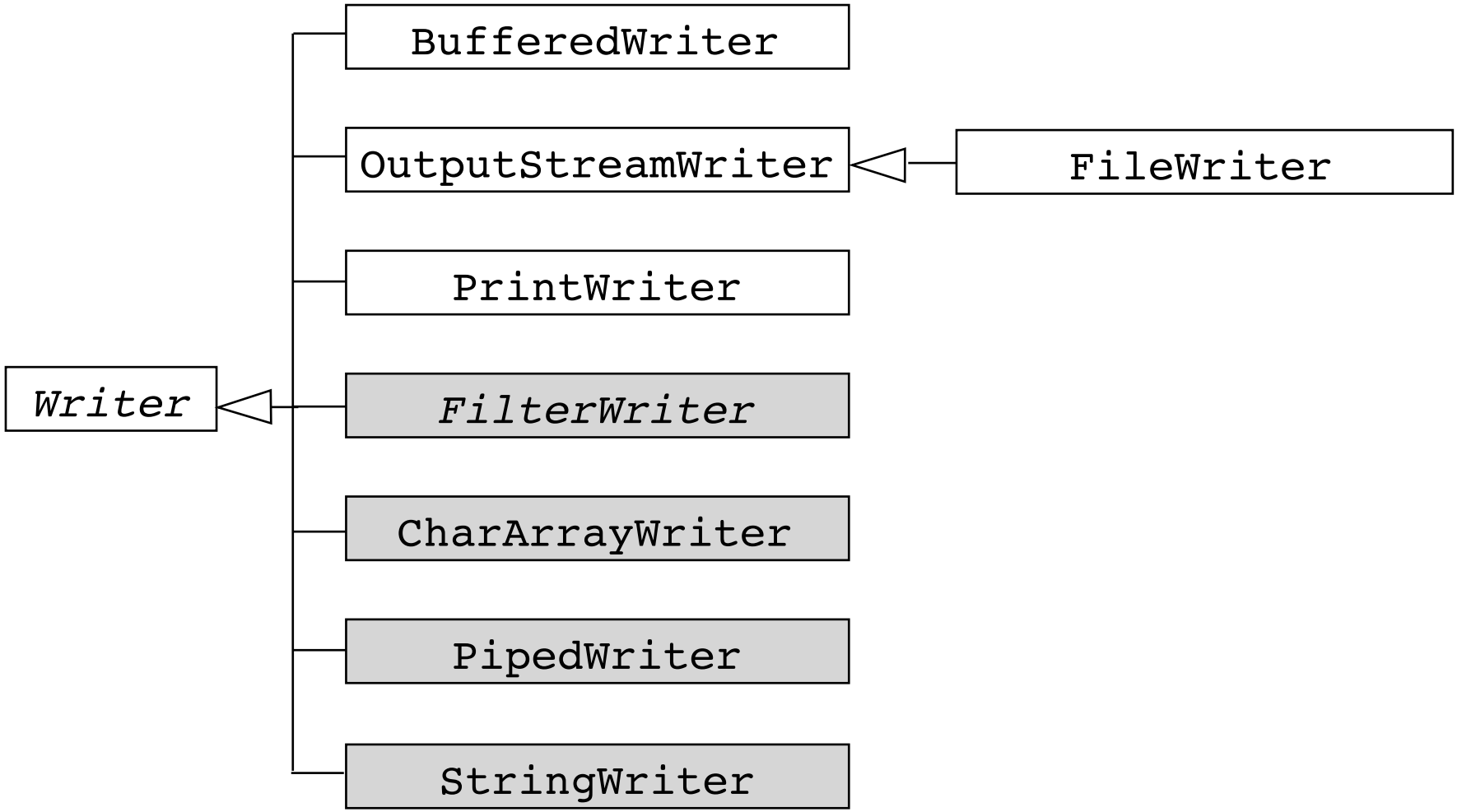
Læsning af en 2-dimensional matrix ved tegnorienteret I/O

```
public class ReadMatrix5 {
    public static void main(String[] args) {
        try {
            BufferedReader in =
                new BufferedReader(
                    new FileReader(args[0]));
            String line = in.readLine();
            int row = Integer.parseInt(line);
            line = in.readLine();
            int col = Integer.parseInt(line);
            double[][] data = new double[row][col];
            for (int i = 0; i < row; i++)    {
                for (int j = 0; j < col; j++) {
                    line = in.readLine();
                    data[i][j] = Double.parseDouble(line);
                    System.out.println(data[i][j]);
                }
            }
            in.close();
        } catch (IOException e) {}
    }
}
```

Reader-hierarkiet



Writer-hierarkiet



Indlæsning fra og udskrivning til systemets standardstrømme

```
public class System {
    public static final InputStream in;
    public static final PrintStream out;
    public static final PrintStream err;
    ...
}
```

Klassen `PrintStream` benyttes normalt ikke længere.
I stedet benyttes `PrintWriter`,

Under UNIX kan standardstrømmene omdirigeres:

`java Program <` *inputfile* `>` *outputfile*

Filer med tilfældig tilgang

Klassen `RandomAccessFile` implementerer grænsefladerne `DataInput` og `DataOutput` og understøtter læsning og skrivning på vilkårlige (byte)positioner i en fil.

Konstruktør:

```
RandomAccesFile(String fileName, String mode)
```

`mode:` `"r"` read-only

`"rw"` read-write

Nye metoder:

```
void seek(long pos)
void skipBytes(int n)
```

Lagring af objekter i en `RandomAccessFile`

```
public class ObjectRandomAccessFile extends RandomAccessFile {
    public ObjectRandomAccessFile(String name, String mode)
                                        throws IOException {
        super(name, mode);
    }

    // ... writeObject and readObject methods
}
```

fortsættes

```
public int writeObject(Object obj) throws IOException {
    if (obj instanceof Serializable) {
        ByteArrayOutputStream bOut = new ByteArrayOutputStream();
        ObjectOutputStream objOut = new ObjectOutputStream(bOut);
        objOut.writeObject(obj);
        int count = bOut.size();
        writeInt(count);
        byte[] buf = bOut.toByteArray();
        write(buf, 0, count);
        return 4 + count;
    }
    return 0;
}
```

fortsættes

```
public Object readObject()
                throws IOException, ClassNotFoundException {
    int count = readInt();
    byte[] buf = new byte[count];
    read(buf, 0, count);
    ObjectInputStream objIn =
        new ObjectInputStream(
            new ByteArrayInputStream(buf, 0, count));
    return objIn.readObject();
}
```

```
java TestWrite obj.out
```

```
public class TestWrite {
    public static void main(String[] args) {
        try {
            ObjectRandomAccessFile out =
                new ObjectRandomAccessFile(args[0], "rw");
            Object[] obj = {"Tic", "Tac", "Toe"};
            long offset = 0;
            for (int i = 0; i < obj.length; i++) {
                int count = out.writeObject(obj[i]);
                System.out.println(obj[i] +
                       " written at offset " + offset +
                       " size = " + count);
                offset += count;
            }
        } catch (IOException e) {}
    }
}
```

```
Tic written at offset 0 size 14
Tac written at offset 14 size 14
Toe written at offset 28 size 14
```

```
java TestRead obj.out 28 14 0
```

```
public class TestRead {
    public static void main(String[] args) {
        try {
            ObjectRandomAccessFile in =
                new ObjectRandomAccessFile(args[0], "r");
            for (int i = 1; i < args.length; i++) {
                long offset = Long.parseLong(args[i]);
                in.seek(offset);
                Object obj = in.readObject();
                System.out.println(obj +
                        " read at offset " + offset);
            }
            in.close();
        } catch (IOException e) {}
    }
}
```

```
Toe read at offset 28
Tac read at offset 14
Tic read at offset 0
```

Ugeseddel 5

28. september - 5. oktober

- Læs afsnit 8.3 (side 333 - 366) i lærebogen.
- Løs opgave 8.1 samt de to opgaver på de næste sider.

Ekstraopgave 2

Betragt nedenstående programskitse:

```
class PersonList {
    PersonList() {}

    PersonList(String fileName) { ... }

    void add(Person p) {
        p.nextPerson = firstPerson;
        firstPerson = p;
    }

    void toFile(String fileName) { ... }

    String toString() { ... }

    Person firstPerson;
}

class Person {
    Person(String name) {
         this.name = name;
    }

    String name;
    Person nextPerson;
}
```

fortsættes

```
public class Test {
    public static void main(String[] args) {
        PersonList list1 = new PersonList();
        list1.add(new Person("Hansen"));
        list1.add(new Person("Jensen"));
        list1.add(new Person("Nielsen"));
        list1.toFile("personer");
        PersonList list2 = new PersonList("personer");
        System.out.println(list2);
    }
}
```

Færdiggør programmet ved anvendelse af **objekt-serialisering**, således at en kørsel resulterer i følgende udskrift:

```
Nielsen
Jensen
Hansen
```

Ekstraopgave 3

Nedenfor er vist et udkast til en samling Java-klasser, der er beregnet til katalogisering af dyr.
Klassen `Animal` er en fælles overklasse for de to klasser `Carnivore` og `Herbivore`,
der repræsenterer henholdsvis mængden af kødædende og planteædende dyr.

```
class Animal {
    Animal(String n) {
        /* Kode ikke medtaget. Se spørgsmål 1 */
    }

    static void printAnimals() {
        /* Kode ikke medtaget. Se spørgsmål 3 */
    }

    static Animal firstAnimal;

    String name;
    Animal next;
    public String toString() { return name; }
}

class Carnivore extends Animal {
    Carnivore(String n, int m) {
        /* Kode ikke medtaget */
    }

    int meatNeeded;
}

class Herbivore extends Animal {
    Herbivore(String n, int g) {
        /* Kode ikke medtaget. Se spørgsmål 2 */
    }

    int grassNeeded;
}
```

fortsættes

Hvert `Animal`-objekt er forsynet med en tekststreng `name`, der angiver dyrets navn. Referencen `next` benyttes til opbevaring af `Animal`-objekterne i en envejsliste, idet den statiske reference `firstAnimal` peger på det forreste objekt i denne liste, og `next` angiver objektets efterfølger i listen.

> **Spørgsmål 1** Programmér en konstruktør for klassen `Animal`, som tildeler `name` en værdi og indsætter det genererede objekt forrest i envejslisten.

For hvert planteædende dyr (`Herbivore`-objekt) angives dets daglige fødebehov ved hjælp af attributten `grassNeeded`.

> **Spørgsmål 2** Programmér en konstruktør for klassen `Herbivore`, som tildeler værdier til `name` og `grassNeeded`, og som indsætter objektet i envejslisten ved hjælp af konstruktøren for `Animal`.
>
> **Spørgsmål 3** Programmér metoden `printAnimals`, der gennemløber envejslisten af dyr og udskriver deres navne.
>
> **Spørgsmål 4** Programmér en metode `printHerbivores`, der kun udskriver navnene på de af dyrene i envejslisten, der er planteædere (`Herbivore`-objekter).

fortsættes

Klassen `Herbivore` benyttes nu som overklasse for klassen `Giraffe`, der repræsenterer mængden af giraffer. Attributten `neckLength` angiver halslængden for en giraf.

```
class Giraffe extends Herbivore {
    Giraffe(String n, int g, double nL) {
        /* Kode ikke medtaget */
    }

    double neckLength;
}
```

Spørgsmål 5 Lad der være givet følgende erklæringer:

```
Animal a;
Herbivore h;
Carnivore c;
Giraffe g1, g2;
```

Antag at objekterne `a`, `h`, `c`, `g1` og `g2` eksisterer. Hvilke af følgende sætninger vil da give fejl under oversættelsen?

(1) `a = g1;`
(2) `a = h;`
(3) `g1 = a;`
(4) `g1 = g2;`
(5) `g1 = (Giraffe) h;`
(6) `g1 = (Carnivore) h;`
(7) `g1 = (Giraffe) c;`