Animationen

Laden und Anzeigen eines Bitmap

Animation

- Die repaint-Schleife
- Verwendung von Threads
  - Abspielen einer Folge von Bitmaps
  - Animation mit Grafikprimitiven
  - Reduktion des Bildschirmflackerns
  - Doppelpufferung „double-buffering“

Laden eines Bitmap

public Image getImage(String filename)

`getImage() akzeptiert als Parameter die Bitmap-Typen` **`gif`** `und` **`jpeg`**`.`

Man kann indirekt, mit Aufruf der getImage()-Methode, Bilder laden innerhalb eines Applets oder mit Hilfe der Toolkit-Klasse innerhalb einer selbständigen Anwendung.

Ein getrenntes Thread wird gestartet, um ein Bild zu lesen.

Laden in Applikationen

```
Image img;
img = getToolkit().getImage("wurm.gif");
```

Laden in Applets

```
Image img;
img = getImage(getDocumentBase(),"wurm.gif");
```

Ausgabe auf dem Bildschirm

public boolean drawImage(Image, int, int, ImageObserver)
public Image drawImage(Image, int, int, Color, ImageObserver)
public Image drawImage(Image, int, int, int, int, int, int, int, int, Color, ImageObserver)
public Image drawImage(Image, int, int, int, int, int, int, int, int, ImageObserver)

Die drawImage Methode hat zwei Objekt-Parameter
(Image img,
int x,
int y,
int width,
int height,
ImageObserver observer)

Objekt observer dient zur Übergabe eines ImageObserver-Objektes, mit dem der Ladezustand der Bitmaps überwacht werden kann.

Ausgabe auf dem Bildschirm

```
public void paint( Graphics g )
{
    Image img;
    img = getToolkit().getImage("duke.gif");
    g.drawImage(img,40,40,this);
}
```

Die gewählte Vorgehensweise ist nicht besonders effizient, denn das Bitmap wird bei jedem Aufruf von paint neu geladen.

Innerhalb der paint()-methode sollte man die getImage()-Methode vermeiden.

```
public class GetImageTest extends Frame{
  Image image;

   public GetImageTest(){
     setTitle( "FirstImage" );
     setSize(300,200);
     image = getToolkit().getImage("wurm.gif");
     setVisible(true);
   }//end constructor

   public void paint(Graphics g){
     g.drawImage(image,0,0,this);
   }

   public static void main(String[] args)
   {  new GetImageTest(); }
 }//end Image01 class
```

MediaTracker-Klasse

Oft ist es sinnvoll, das vollständige Laden eines Bildes abzuwarten, bevor im Programm fortgefahren wird.

Beispiel:

Wenn die Größe des Bitmaps benötigt wird, um sie korrekt auf dem Bildschirm zu skalieren.

```
MediaTracker mt = new  MediaTracker( Component comp )
```

this-Zeiger des aktuellen Fensters

MediaTracker-Klasse

- Der MediaTracker bietet eine Reihe von Methoden, um den Ladezustand der Bilder zu überwachen.

z.B.

Die Methode waitForAll

wartet, bis alle Images vollständig geladen sind:

Animation

Das Grundprinzip einer Animation besteht darin, in einer Schleife die Methode repaint() wiederholt aufzurufen.

Die repaint-Schleife

```
public void paint(Graphics g)
{
      ++zähler;
      g.drawString("Counter = "+zähler,10,50);
       try {
          Thread.sleep(1000);
       } catch (InterruptedException e) {
       }
 }
```

```
public class FirstAnim extends Frame {
   int counter = 0;

   public FirstAnim()  {
      this.setSize(250,150);
      this.setVisible(true);
   }

   public void startAnimation() {
        while (true) {
             repaint();
         }
   }

   public void paint(Graphics g)  {
         ++counter;
         g.drawString("Counter = "+counter,10,50);
         try {   Thread.sleep(1000);
         } catch (InterruptedException e) { /* . .*/ }
   }
}
```

```
   ....
    public static void main(String[] args)
    {
       FirstAnim anim = new FirstAnim();
       anim.startAnimation();
    }
  ....
```

Verwendung von Threads

- Die repaint-Schleife soll in einem eigenen Thread laufen.
- Wir versuchen aus paint() die Zeitverzögerung herauszunehmen und statt dessen in die repaint()-Schleife zu verlagern.
- Der Haupt-Thread des Animationsprogramms wird genügend Zeit haben, um die Bildschirmausgabe durchzuführen und auf andere Events zu reagieren.

```
. . . . . .
    public void startAnimation()  {
          Thread th = new Thread(this);
          th.start();
    }

    public void run()   {
          while (true) {
                 repaint();
                 try {
                       Thread.sleep(1000);
                 } catch (InterruptedException e)
                       {//nichts }
          }
    }

    public void paint(Graphics g) {
       ++cnt;
       g.drawString("Counter = "+cnt,10,50);
    }
. . . . . .
```

Abspielen einer Folge von Bitmaps

```
Thread th;
Image arImg[];
int actimage;
```

```
public void startAnimation()
  {
    th = new Thread(this);
    actimage = -1;
    th.start();
  }
```

Abspielen einer Folge von Bitmaps

```
...
public void run() {
   arImg = new Image[30];
   MediaTracker mt = new MediaTracker(this);
   Toolkit tk = getToolkit();
   for (int i = 1; i <= 30; ++i) {
     arImg[i-1] = tk.getImage("images//jana"+i+".gif");
     mt.addImage(arImg[i-1], 0);
     actimage = -i;
     repaint();
     try { mt.waitForAll();
     } catch (InterruptedException e) {}
   }...
...
```

Animation mit Grafikprimitiven

- Alternativ zur Anzeige von Bilddateien kann jedes Einzelbild der Animation auch mit den Ausgabeprimitven der Klasse Graphics erzeugt werden.
- Nachteil ist, dass die Grafikoperationen zeitaufwendiger sind, und eine zügige Bildfolge bei komplexen Sequenzen schwieriger zu erzielen ist.

Reduktion des Flimmers auf dem Bildschirm

- vor jedem Aufruf von paint() wird das Fenster gelöscht
- ein vollständig leerer Hintergrund wird immer dazwischen gezeigt

update()-Methode

```
public void update( Graphics g ) {
   g.setColor(getBackground());
   g.fillRect(0, 0, width, height);
   g.setColor(getForeground());
   paint(g);
}
```

```
public void update(Graphics g) {
   paint(g);
}
```

Doppelpufferung

- Wir erzeugen ein neues Image mit der createImage()-Methode (ein Offscreen-Image) und speichern es in einer Instanzvariablen.
- Wir beschaffen für unsere neue Image einen graphischen Kontext mit Hilfe der getGraphics()-Methode.
- Alle Bildschirmausgaben (inklusive Löschen des Bildschirms) gehen zunächst auf unser Image (Offscreen-Image)
- Wenn alle Ausgabeoperationen abgeschlossen sind, wird das Offscreen-Image mit drawImage in das Ausgabefenster kopiert.

```
public void update(Graphics g)
{
  if (dbImage == null) {
    dbImage = createImage(size().width,size().height);
    dbGraphics = dbImage.getGraphics();
  }
   //Hintergrund löschen
   dbGraphics.setColor( getBackground() );
   dbGraphics.fillRect(0,0,size().width,size().height);
   //Vordergrund zeichnen
   dbGraphics.setColor(getForeground());
   paint(dbGraphics);
   //Offscreen anzeigen
   g.drawImage(dbImage,0,0,this);
}
```

Eingabe und Ausgabe in Java-Programmen

I/O in Java

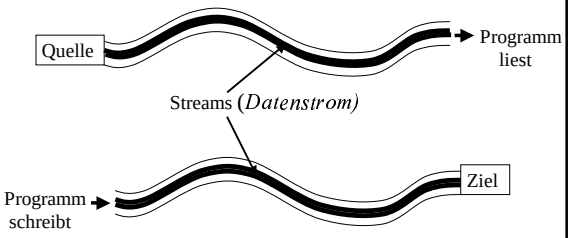
Oft müssen Programme Daten aus externen Quellen lesen, bzw. abspeichern.

Tastatur
Scanner
Speicher
Dateien auf der Festplatte
Aus einer Netzwerkverbindung
Ein anderes Programm

Die Daten sind dabei beliebig: Objekte, Zahlen, ...

I/O in Java

Streams sind abstrakte Objekte, die eine Verbindung zwischen Quelle und Ziel eines sequenziellen Flusses von Bytes herstellen.



Java Streams

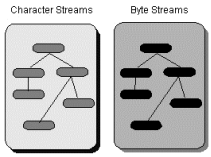
- Ein Stream ist dabei zunächst nur ein abstraktes Konstrukt, um Zeichen auf ein imaginäres Ausgabegerät zu schreiben oder von diesem zu lesen.
- Erst konkrete Unterklassen binden die Zugriffsroutinen an echte Ein- oder Ausgabegeräte, wie beispielsweise an Dateien, Netzwerk, Strings(!) .
- Beide Konzepte sind mit normalen Sprachmitteln realisiert und können selbst erweitert werden.

I/O in Java

`java.io.*`

In JAVA 1.0 hatte man **Unicode** noch nicht konsequent mit in das Stream-Konzept einbezogen.

In JAVA 1.1 wurden die Reader- und Writer-Klassen eingeführt, die eine jeweils passende Zeichencode-Umsetzung vornehmen.

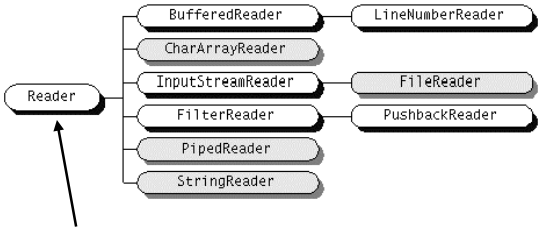


Character Streams

Für zeichenorientierte Dateien sollte man vorsichtshalber immer Reader und Writer verwenden

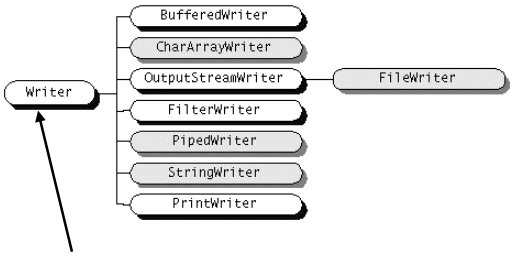
Character Streams { Reader, Writer } — **Plattform-unabhängig und schneller**

Reader



Abstrakte Klasse

Writer



Abstrakte Klasse

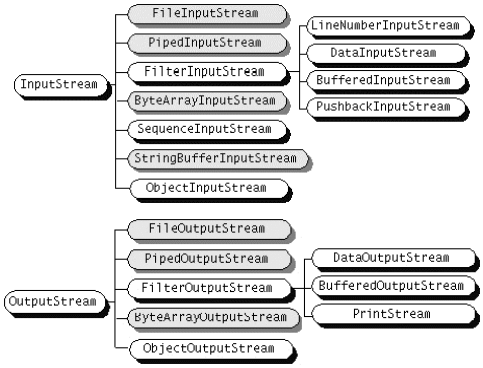
Byte Streams

Eingabeströme: Unterklassen der abstrakten Klasse `InputStream`.

Ausgabeströme: Unterklassen der abstrakten Klasse `OutputStream`.

Die Streams **`System.in`** und **`System.out`** sind die Standardeingabe (Tastatur) und die Standardausgabe (Bildschirm).

Byte Streams



Reader und InputStreams

Reader und InputStreams definieren in ihren APIs
die gleichen Methoden, aber für verschiedene Datentypen

Reader

```
int read()
int read( char cbuf[])
int read( char cbuf[], int offset, int length)
```

InputStream

```
int read()
int read( byte cbuf[])
int read( byte cbuf[], int offset, int length)
```

Stream-Operationen

- Die Operation `InputStream.read()` liefert eine ganze Zahl zwischen 0 und 255 (ein Byte) und `-1` bei Dateiende (EOF).
- Der Aufruf von `read()` blockiert die Ausführung des Programms, bis ein Datum vorliegt.
- Die Operation `OutputStream.write(int b)` schreibt eine ganze Zahl als Byte (nur die niederwertigen 8 Bit werden übertragen!)
- `write(int b)` blockiert, bis das Zeichen geschrieben ist.

Beispiel

```
import java.io.*;

public class Echo {
  public static void main(String[] argv) {
    InputStream  in  = System.in;
    OutputStream out = System.out;
    try {
       int ch = 0;
       while( ( ch=in.read() ) != -1)
         out.write(ch);
    } catch ( IOException e ) {
      e.printStackTrace();
    }
  }
}
```

Sowohl `read()` als auch `write()`werfen bei Ein/ Ausgabe-Fehlern `IOExceptions`.

Beispiel

```
import java.io.*;

public class Echo {

  public static void main( String[] args ) throws IOException {

    InputStreamReader input = new InputStreamReader( System.in );
    int i;
    while ( ( i = input.read() ) >= 0 ) {
       System.out.println( "Echo: " + (char) i  );
    }
  }

} // end of class Echo
```

Writer und OutputStreams

Reader und InputStreams definieren in ihren APIs
die gleichen Methoden, aber für verschiedene Datentypen.

Writer

```
int write( int c )
int write( char cbuf[])
int write( char cbuf[], int offset, int length)
```

OutputStream

```
int write( int c )
int write( byte cbuf[])
int write( byte cbuf[], int offset, int length)
```

Sink Type	Character Streams	Byte Streams
Memory	CharArrayReader CharArrayWriter StringReader StringWriter	ByteArrayInputStream ByteArrayOutputStream StringBufferInputStream
Pipe	PipedReader PipedWriter	PipedInputStream PipedOutputStream
File	FileReader FileWriter	FileInputStream FileOutputStream

Dateien

```
File f = new File("hello.txt");
if( f.exists() )
     ...
f = new File("bla/blubb");
f.mkdirs();
```

- Ein-/Ausgabe mit Dateien geschieht mit speziellen Unterklassen von `Input/OutputStream`, z.B. `FileInputStream`.
- Spezialfall: Zugriff auf ZIP-Dateien mit `java.util.zip.ZipFile`

Um den Inhalt einen Datei in einem anderen zu kopieren

```
...
File inputFile = new  File( "D:\\maggy\\quell.txt" );
File outputFile = new  File( "D:\\maggy\\ziel.txt" );

FileReader in = new FileReader( inputFile );
FileWriter out = new FileWriter( outputFile );
int c;

while ( ( c = in.read() ) != -1)
     out.write( c );
in.close();
out.close();
...
```

Streams

Um den Inhalt einen Datei in einem anderen zu kopieren

```
...
File inputFile = new  File( "D:\\maggy\\quell.txt" );
File outputFile = new  File( "D:\\maggy\\ziel.txt" );

FileInputStream in = new FileInputStream( inputFile );
FileOutputStream out = new FileOutputStream ( outputFile );
int b;

while ( ( b = in.read() ) != -1)
     out.write( b );
in.close();
out.close();
...
```

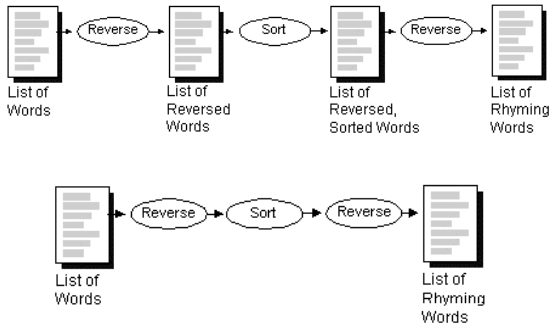
Pipes

Pipes sind I/O-Verbindungen zwischen verschiedenen Teilen eines Programms oder verschiedenen Threads.

Der schreibende Partner arbeitet mit einem PipedOutputStream, der lesende mit PipedInputStream. Auf Zeichenebene gibt es analog die Klassen PipedReader und PipedWriter.

Es gibt jeweils einen Konstruktor mit einem Objekt der jeweils anderen Klasse. Die Objekte können auch getrennt voneinander kreiert und nachträglich mit connect verbunden werden.

Pipes



Processing Streams

Process	CharacterStreams	Byte Streams
Buffering	BufferedReader	BufferedInputStream
	BufferedWriter	BufferedOutputStream
Filtering	FilterReader,	FilterInputStream
	FilterWriter	FilterOutputStream
Converting between Bytes and Characters	InputStreamReader	
	OutputStreamWriter	
Concatenation		SequenceInputStream
Object Serialization		ObjectInputStream ObjectOutputStream
Data Conversion		DataInputStream
		DataOutputStream
Counting	LineNumberReader	LineNumberInputStream
Peeking Ahead	PushbackReader	PushbackInputStream
Printing	PrintWriter	PrintStream