6 Was es sonst noch gibt

6.1 Die Klasse Collection

Einige wichtigen Unterklassen der abstrakten Klasse **Collection** haben wir in Beispielen bereits kennen gelernt. Dazu gehören die Klassen **Array**, **ByteArray**, **String**, und **Symbol**.

Beispiel:

```
| anArray |
anArray := Array new: 5.
1 to: anArray size do: [ : index | anArray at: index put: 0 ].
^anArray                                                    ( 0 0 0 0 0 )
```

Es gibt weitere nützliche Unterklassen der Klasse. Einige sollen hier kurz übersichtsmäßig vorgestellt werden.

Object
- ***Collection***
  - **Bag**
  - ***Dictionary***
    - ***IdentityDictionery***
  - ***SequenceableCollection***
    - **AdditivSequenceableCollection**
      - ***OrderedCollection***
      - ***SortedCollection***
    - **ArrayedCollection**
      - ***Array***
      - ***ByteArray***
      - ***String***
        - ***Symbol***
      - **Interval**
  - ***Set***
    - ***EsIdentitySet***
- **...**

Collectionen sind Objekte die andere Objekte beinhalten. Der Zugriff auf die Elemente einer Collection kann über Indizes, Schlüssel oder auch das Objekt selbst erfolgen.

Indiziert durch	Klasse	Geordnet nach	Speicher	Inhalt
Integer	Array	Index	fest	beliebige Objekte
	ByteArray	Index	fest	Integer: 0-255
	Interval	Internal	fest	Bereich
	OrderedCollection	Index	variabel	beliebige Objekte
	SortedCollection	Internal	variabel	beliebige Objekte
	String	Index	fest	Zeichen
	Symbol	Index	fest	Zeichen

Schlüssel	Dictionary IdentityDictionary	Schlüssel, = Schlüssel, ==	variabel variabel	Schlüssel + bel. Objekte Schlüssel + bel. Objekte
nicht	Bag Set EsIdentitySet	nicht nicht nicht	variabel variabel variabel	bel. Objekte außer nil bel. Objekte außer nil bel. Objekte außer nil

Klassenmethoden

new	Erzeugt neues Objekt.
new: *count*	s.o. mit Festlegung der Elementezahl.
with: *element1*	s.o. mit einem Element.
with: *element1* **with:** *element2***...**	s.o. mit bis zu vier Elementen.

Instanzmethoden (Auswahl)

add: *anObject*	Anfügen von *anObject.*
addAll: *aCollection*	Anfügen von *aCollection.*
asArray	Konvertieren in ein Array-Objekt.
asBag	Konvertieren in ein Bag-Objekt.
asByteArray	Konvertieren in ein ByteArray-Objekt.
asOrderedCollection	Konvertieren in ein OrderedCollection-Objekt.
asSet	Konvertieren in ein Set-Objekt.
asSortedCollection	Konvertieren in ein SortedCollection-Objekt, s. unten.
collect: *aBlock*	s.o.
copy	Kopieren.
do: *aBlock*	s.o.
includes: *anObject*	Suchen nach *anObject.*
inject: *anObject* **into:** *aTwoArgumentBlock*	s. Beispiel unten.
reject: *aBlock*	s.o.
remove: *anObject*	Löschen von *anObject.*
remove: *anObject* **ifAbsent:** *aBlock*	s.o., falls nicht vorhanden, Ausführen von *aBlock.*
removeAll: *aCollection*	Löscht alle *aCollection* - Elemente.
select: *aBlock*	s.o.
size	Anzahl der Elemente.

Beispiel zu **inject:into:**

^(1 to: 10) inject: 1 into: [:fak :ele | fak * ele] ⇒ **3628800**

Die erste Blockvariable des *aTwoArgumentBlock* wird mit *anObject* als Startwert belegt und behält ihren Wert nach jedem Blockdurchlauf. Der zweiten Blockvariable werden der Reihe nach alle Elemente der Collection zugewiesen.

1 to: 10 Objekt der Klasse **Interval**

Beispiel zu **asSortedCollection**:

```
| set |
set := Set new.
set add: ‘ xyz ‘; add: ‘ abc ‘; add: ‘ dgh ‘.        ⇒ ‘ xyz ‘ ‘ abc ‘ ‘ dgh ‘
set asSortedCollection                               ⇒ ‘ abc ‘ ‘ dgh ‘ ‘ xyz ‘
```

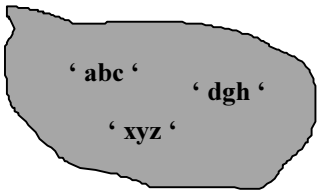
Um Elemente in eine entsprechende andere Collection zu übertragen, stehen Methoden zur Verfügung, die jede Collection versteht.

6.1.1 Set

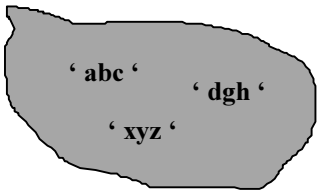
Die Klasse **Set** ist eine ungeordnete Collection, eine Menge. Alle Objekte in einem Set stehen genau einmal zur Verfügung, wobei die Reihenfolge der Eingabe keine Bedeutung hat. Die Reihenfolge der Abspeicherung wird vom System bestimmt und kann von außen nicht vorgegeben werden.

Beispiel:

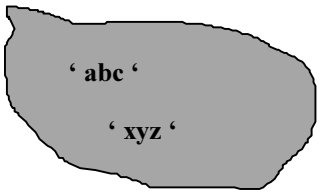
```
| set |
set := Set new.
set add: ‘ xyz ‘; add: ‘ abc ‘; add: ‘ dgh ‘.
```




```
set add: ‘ abc ‘.
```




```
set remove: ‘ dgh ‘.
```



Bemerkung zu **EsIdentitySet**:
In allen Vergleichsoperationen wird stets die Identität == verwendet. **Set** und **EsIdentitySet** unterscheiden sich darüber hinaus nicht.

6.1.2 SequenceableCollection

In dieser abstrakten Klasse werden Collectionen, deren Elemente **geordnet** sind, zusammengefasst.

Instanzmethoden (Auswahl)

, *aCollection*	Verknüpfen mit *aCollection.*
at: *anInteger*	Gibt Element mit Index *anInteger* zurück.
at: *anInteger* **put:** *anObject*	Setzt *anObject* als Element mit Index *anInteger.*
atAllPut: *anObject*	Setzt alle Elemente auf *anObject.*
first	Greift auf das erste Element zu.
last	Greift auf das letzte Element zu.

findFirst: *aBlock*	Sucht des erste Element für welches *aBlock* true ist.
findLast: *aBlock*	Sucht des letzte Element für welches *aBlock* true ist.
indexOf: *anObject*	Liefert den Index von *anObject*, sonst 0.
indexOf: *anObject* **ifAbsent:** *aBlock*	Liefert den Index von *anObject*, sonst wird *aBlock* ausgeführt.
reverse	Die Reihenfolge der Elemente wird umgekehrt.

6.1.2.1 OrderedCollection

Ordnet alle Elemente in der Reihenfolge des Einfügens in Form einer Liste an.

Beispiel:

```
| oc |
oc := OrderedCollection new.
oc add: 1; add: $A; add ‘ String ‘; add: # Symbol.
^oc
```


```
oc add: 4.0
```


```
oc remove: ‘ String ‘
```


```
oc remove: 2 ifAbsent: [ ]
```

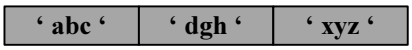
Wird das Element **2** nicht vorgefunden, so wird der Block **[ ]** (hier: leerer Block) abgearbeitet.

6.1.2.2 Sorted Collection

Prinzipiell verhält sich diese Klasse so wie die Klasse **OrderedCollection**. Während jedoch jedes neue Element in einer OrderedCollection in der Regel am Ende der Collection hinzugefügt wird, wird ein neues Element in einer SortedCollection an der „richtigen“ Stelle einsortiert. Für diese Sortierung wird normalerweise die Vergleichsnachricht <= verwendet. Die Instanzmethode **at:put:** ist hier deshalb nicht sinnvoll, eine SortedCollection reagiert mit einer Fehlermeldung.

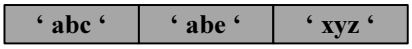
Beispiel:

```
| sc |
sc := sortedCollection new.
sc add: ‘ xyz ‘; add: ‘ abc ‘; add: ‘ dgh ‘.
```




```
sc add: ‘ abe ‘.
```


```
sc remove: ‘ dgh ‘.
```



Können Objekte nicht mit der Vergleichsnachricht <= verglichen werden oder wird eine andere Sortierordnung gewünscht, so kann zu einer SortedCollection ein geeignetes Sortierkriterium in Form eines Blocks mit zwei Argumenten festgelegt werden.

Instanzmethoden (Auswahl)

sortBlock	Gibt den Sortierblock zurück.
sortBlock: *aTwoArgumentBlock*	Setzt den Sortierblock.

Beispiel:

```
Object subclass: #Ausleihgegenstand
   instanceVariableNames: 'titel regNummer '
   classVariableNames: ''
   poolDictionaries: ''

Object subclass: #Regal
   instanceVariableNames: 'exemplare '
   classVariableNames: ''
   poolDictionaries: ''

Regal class publicMethods
new
    ^super new initialize

Regal publicMethods
initialize
    "Instanzvariablen werden initialisiert."
    exemplare := SortedCollection new.
    exemplare sortBlock: [:x :y| x regNummer <= y regNummer]

| regal ag1 ag2 |
ag1 := Ausleihgegenstand new.
ag2 := Ausleihgegenstand new.
ag1 titel: ‘ ABC ‘; regNummer: 5.
```

```
ag2 titel: ‘ CBA ‘; regNummer: 3.
regal := Regal new.
regal add: ag1; add ag2.
^regal
```

6.1.3 Dictionary

Oft ist es nicht ausreichend, Objekte einfach in einer Collection zusammenzufassen. Wird eine Abbildung benötigt, so steht in ST die Klasse Dictionary zur Verfügung. Jedes Element eines Dictionaries ist ein Paar von Objekten, bestehend aus einem *Schlüssel* (**key**) und einem *Wert* (**value**). Über den Schlüssel kann, wie über einen Index, auf seinen Wert zugegriffen werden. Die Zuordnung erfolgt durch die Nachricht **->**. Das heißt, der ST-Ausdruck **key -> value** liefert als Objekt die gewünschte *Verbindung* zwischen einem Schlüssel und seinem Wert.
Es ist allerdings unüblich, einem Dictionary mittels der **add:**-Nachricht Elemente zuzuweisen.

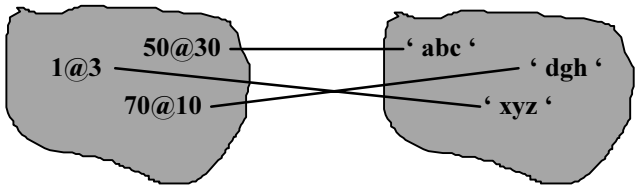
```
dict add: (1@1) -> ‘AAA ‘
```

Statt dessen verwendet man hier die **at:put:**-Nachricht der Form

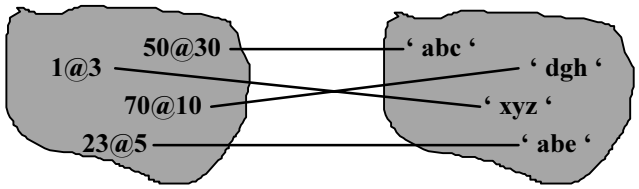
```
dict at: (1@1) put: ‘AAA ‘
```

Beispiel:

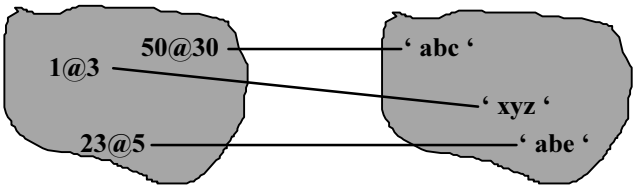
```
| dict |
dict := dictionary new.
dict at: (1@3) put: ‘ xyz ‘; at: (50@30) put: ‘ abc ‘; at: (70@10) put: ‘ dgh ‘.
```




```
dict at: (23@5) put: ‘ abe ‘.
```



dict removeKey: (70@10).



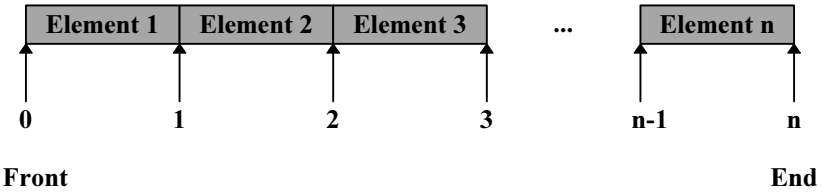
Instanzmethoden (Auswahl)

at: *aKey*	Gibt Element mit Schlüssel *aKey* zurück.
at: *aKey* **ifAbsent:** *aBlock*	Gibt Element mit Schlüssel *aKey* zurück, sonst wird *aBlock* ausgeführt.
at: *aKey* **put:** *anObject*	Setzt *anObject* als Element mit Schlüssel *aKey.*
includes: *anObject*	Suchen nach *anObject.*
includesKey: *aKey*	Suchen nach *aKey.*
keyAtValue: *anObject*	Suchen nach dem Schlüssel von *anObject.*
keyAtValue: *anObject* **ifAsent:** *aBlock*	Suchen nach dem Schlüssel von *anObject*, sonst wird *aBlock* ausgeführt.
keys	Liefert eine Collection aller Schlüssel.
keysDo: *anOneArgumentBlock*	Jeder Schlüssel wird dem *anOneArgumentBlock* übergeben.
keysAndDo: *aTwoArgumentBlock*	Jeder Schlüssel und sein Wert wird dem *aTwoArgumentBlock* übergeben.
removeKey: *aKey*	Löscht die Zuordnung mit dem Schlüssel *aKey*.
removeKey: *aKey* **ifAsent:** *aBlock*	Löscht die Zuordnung mit dem Schlüssel *aKey*, sonst wird *aBlock* ausgeführt.
values	Liefert eine Collection aller Werte.

Bemerkung zu **IdentityDictionary**:
In allen Vergleichsoperationen wird stets die Identität == verwendet. **Dictionary** und **IdentityDictionary** unterscheiden sich darüber hinaus nicht.

6.2 Die Klasse Stream

Diese abstrakte Klasse ist der Klasse **Collection** in gewissen Hinsicht sehr ähnlich. Sie wird benutzt, um einzelne Objekte zu sammeln oder Objekt für Objekt abzuarbeiten. Eine Verwaltung, wie wir sie von den Collectionklassen kennen, ist hier eher zweitrangig. Der Zugriff auf die einzelnen Stream-Objekte erfolgt über einen Zeiger.
Solche Stream-Objekte werden in ST gerne benutzt, um eine unbeschränkte Anzahl von Objekten aufzunehmen bzw. abzuarbeiten. Dabei organisieren sie lediglich den lesenden bzw. schreibenden Zugriff auf das aktuelle Element.



Object
Stream
PositionableStream
ReadStream
WriteStream
ReadWriteStream
EsRandom
...

Instanzmethoden (Auswahl)

atEnd	Fragt, ob der Zeiger am Ende des Streams steht.
do: *aBlock*	Übergibt dem Block *aBlock* alle Elemente vom Zeiger beginnend bis zum Streamende.
next	Liest das aktuelle Element und setzt den Zeiger auf das nächste Element, sonst nil.
size	Gibt die Anzahl der Elemente des Streams zurück.

Ein Stream-Objekt wird grundsätzlich aus einer **indizierbaren Collection** erzeugt. Eine Möglichkeit besteht im Senden einer der beiden folgenden Nachrichten an eine Collection, welche als Methoden in der Klasse **Collection** implementiert sind. Als Ergebnisobjekt wird ein Stream-Objekt geliefert:

Instanzmethoden (Auswahl)

readStream	Liefert ein Objekt der Klasse **ReadStream**.
writeStream	Liefert ein Objekt der Klasse **WriteStream**.

Beispiel zu **readSream**:

„Aus einem Array-Objekt wird ein neuer ReadStream erzeugt.
Anschließend wird Objekt für Objekt gelesen und auf dem Transcript ausgegeben.“

```
|stream|
stream := #( $a 'string' #symbol 0) readStream.
[stream atEnd]
   whileFalse:
      [Transcript show: stream next printString; cr]
```

⇒ **nil**

⇒ **Transcript:**
$a
'string'
#symbol
0

Beispiel zu **writeStream**:

**„Aus einem Array-Objekt wird ein neuer WriteStream erzeugt.
Anschließend werden auf ihm Objekte abgelegt.“**

```
|stream|
stream := #( $a 'string' #symbol 0) writeStream.
stream nextPutAll: #(4 $e #newSymbol 9.0 8).
stream nextPut: 'string'.
stream contents                    ⇒    (4 $e #newSymbol 9.0 8 'string')
```

Weder die Größe noch der Inhalt des Array-Objektes sind für den WriteStream von Bedeutung. Sie dienen nur als Vorlage für das Stream-Objekt, um den Inhalt des Stream-Objekts in eine „passende“ Collection einzusammeln, d.h. das Collection-Objekt bestimmt die Klasse der Objekte, die in dem Stream eingeführt werden dürfen.

Beispiel zu **writeStream**:

```
|stream|
stream := #( $a 'string' #symbol 0) writeStream.
stream contents                    ⇒    ()
```

Neben der Möglichkeit Stream-Objekte mit Hilfe von Nachrichten an indizierbare Collection zu erzeugen, gibt es die Möglichkeit Stream-Objekte direkt mit Hilfe von Klassenmethoden zu definieren.

6.2.1 PositionableStream

Diese abstrakte Klasse fasst Unterklassen zum Lesen und/oder Schreiben von Streams zusammen.

Klassenmethoden (Auswahl)

on: *aCollection*	Erzeugt einen Stream aus den Objekten der indizierbaren *aCollection*, der Zeiger ist auf dem Anfang positioniert.

Instanzmethoden (Auswahl)

contents	Gibt eine Collection aller Streamobjekte zurück.
lineDelimiter	Antwortet mit dem Zeilenendezeichen.
lineDelimiter: *aSeqCollection*	Setzt Zeilenendezeichen.
position	Gibt die aktuelle Position zurück.
position: *anInteger*	Setzt die aktuelle Position neu.
reset	Setzt die aktuelle Position auf den Streamanfang.
setToEnd	Setzt die aktuelle Position auf das Streamende.
upToEnd	Setzt die aktuelle Position auf das Streamende und gibt die dabei überlesenen Objekte als Collection zurück.

Das Pool-Dictionary **CldtConstants** umfasst die wichtigen Konstanten für nichtdruckbare Zeichen, hier eine Auswahl:

Key	Value
Bell	16r9
Bs	16r8
Cr	16rD
Del	16r7f
Esc	16r1B
Lf	16rA
LineDelimiter	Cr Lf

Key	Value
Nul	16r0
PMLineDelimiter	Cr Lf
Space	16r20
Tab	16r9
UnixLineDelimiter	Lf
WINLineDelimiter	Cr Lf

6.2.1.1 ReadStream

Instanzmethoden (Auswahl)

copyFrom: *fromInteger* **to:** *toInteger*	Liest alle Streamobjekte im angegebenen Bereich.
next: *anInteger*	Liest die nächsten *anInteger* Streamobjekte und setzt den Zeiger weiter.
nextLine	Liest alle Streamobjekte bis zum Zeilenende und setzt den Zeiger weiter.
peek	Liest das nächste Objekt, setzt den Zeiger *nicht* weiter.
skip: *anInteger*	Verschiebt den Zeiger um *anInteger* Objekte.
upTo: *anObject*	Liest die nächsten Objekte bis ausschließlich *anObject* und setzt den Zeiger hinter *anObject*.

Alle Lesemethoden beginnen auf der Zeigerposition und liefern evtl. eine Collection von Streamobjekten.

Beispiel:

```
|stream|
stream := ReadStream on: 'Hallo!', LineDelimiter, 'Wie geht es?'.
[stream atEnd]
    whileFalse:
        [Transcript show: stream nextLine printString; cr]    ⇒ nil

                                                              ⇒ Transcript: 'Hallo!'
                                                                'Wie geht es?'
```

6.2.1.2 WriteStream

Instanzmethoden (Auswahl)

cr	Setzt Zeilenendezeichen und positioniert den Zeiger dahinter.
contents	Gibt alle Objekte bis ausschließlich dem Zeiger aus.
nextPut: *anObject*	Schreibt *anObject* und setzt den Zeiger weiter.
next: *anInteger* **put:** *anObject*	Schreibt *anInteger*-Mal das Objekt *anObject* und setzt den Zeiger dahinter.
nextPutAll: *aCollection*	Schreibt alle Objekte aus *aCollection* und setzt den

	Zeiger weiter.
position: *anInteger*	Setzt den Zeiger auf die Position *anInteger*.
space	Schreibt ein Leerzeichen und setzt den Zeiger dahinter.

Alle Schreibaktionen beginnen an der aktuellen Position im Stream.

Beispiel:

```
|stream|
stream := WriteStream on: (Array new: 1).
stream lineDelimiter: #(0).
stream nextPutAll: #(1 2 3 4 5); cr.
stream nextPutAll: #(9 8 7 6); cr.
^stream contents                    ⇒  (1 2 3 4 5 0 9 8 7 6 0)
```

6.2.1.3 ReadWriteStream

Erbt alle Methoden der Klasse **WriteStream** und besitzt zusätzlich analoge Methoden wie die Klasse **ReadStream**.

Instanzmethoden (Auswahl)

contents	Gibt eine Collection aller Streamobjekte zurück.
copyFrom: *fromInteger* **to:** *toInteger*	Liest alle Streamobjekte im angegebenen Bereich.
next: *anInteger*	Liest die nächsten *anInteger* Streamobjekte und setzt den Zeiger weiter.
nextLine	Liest alle Streamobjekte bis zum Zeilenende und setzt den Zeiger weiter.
peek	Liest das nächste Objekt.
skip: *anInteger*	Verschiebt den Zeiger um *anInteger* Objekte.
upTo: *anObject*	Liest die nächsten Objekte bis ausschließlich *anObject* und setzt den Zeiger hinter *anObject*.

Alle Lesemethoden beginnen auf der Zeigerposition und liefern evtl. eine Collection von Streamobjekten. Alle Schreibaktionen beginnen an der aktuellen Position im Stream.

6.2.2 EsRandom

EsRandom ist eine spezielle Klasse, welche Methoden zum Lesen eines Streams von Zufallszahlen zur Verfügung stellt.
Die Objekte von **EsRandom** verhalten sich wie ReadStream-Objekte, welche unendlich viele gleichverteilte Zahlen zwischen 0.0 (inklusiv) und 1.0 (exklusiv) beinhalten. Zum Erzeugen einer EsRandoms reicht die Nachricht **new** aus. Auf die Nachricht **atEnd** antworten EsRandom-Objekte stets mit false. (Objekte dieser Klasse sind unendlich.)

Instanzmethoden (Auswahl)

next	Liest die nächste Zufallszahl.

Beispiel:

```
|rand wurf|
rand := EsRandom new.
wurf := Array new: 6.
1 to: wurf size do:
   [:wurfNumber|
      wurf at: wurfNumber put: (rand next *6) // 1 + 1].
^wurf                                          ⇒        (5 2 3 6 3 1)
```

6.3 Die Klasse CfsFileStream

CfsFileStream ist eine abstrakte Klasse zum Lesen und Schreiben von Bytes oder Zeichen aus bzw. auf eine Datei (**Cfs** - Common File System).

```
Object
   CfsFileStream
      CfsReadFileStream
         CfsReadWriteFileStream
      CfsWriteFileStream
```

Klassenmethoden (Auswahl)

on: *fileDescriptor*	Erzeugt einen CfsFileStream mittels einem *fileDescriptor*, einem Objekt der Klasse **CfsFileDescriptor**, welches den Öffnungsmodus beinhaltet.
open: *pathString* **oflag:** *openFlags*	Erzeugt einen CfsFileStream mit einem *pathString*, Objekt der Klasse **String** und einem Öffnungsmodus *openFlags*.

Welche der Unterklassen erzeugt wird, hängt vom Öffnungsmodus ab:

Öffnungsmodus	**CfsFileStream**
ORDONLY	CfsReadFileStream
OWRONLY	CfsWriteFileStream
ORDWR	CfsReadWriteFileStream

Beispiele:

```
"Definieren eines FileDescriptors."
|file|
file := CfsFileDescriptor
   open: 'c:\example'
   oflag: ORDONLY.
file isCfsError ifTrue: [^self error: file message ].
file close                                     ⇒    a CfsFileDescriptor

|file readStream|
file := CfsFileDescriptor
```

```
    open: ' c:\example'
    oflag: ORDONLY.
file isCfsError ifTrue: [^self error: file message ].
readStream := CfsFileStream on: file.
Transcript show: readStream nextLine.          ⇒ Transcript:  Zeile1
readStream close                               ⇒ a CfsReadFileStream
    "Es wird nur eines von beiden geschlossen."
```

Instanzmethoden (Auswahl)

atEnd	Fragt, ob der Zeiger auf dem Dateiende steht.
bufferSize	Gibt die aktuelle Puffergröße in Byte zurück.
bufferSize: *anInteger*	Setzt die aktuelle Puffergröße in Byte.
close	Schließen eines Files.
fileDescriptor	Gibt den Filedescriptor zurück, mit welchem der File geöffnet wurde.
isBytes	True, falls byteweiser Zugriff.
isBytes: *aBoolean*	Falls *aBoolean* true ist, so byteweiser Zugriff, sonst zeichenweiser Zugriff.
isCharacter	True, falls zeichenweiser Zugriff.
isCharacter: *aBoolean*	Falls *aBoolean* true ist, so zeichenweiser Zugriff, sonst byteweiser Zugriff.
lineDelimiter	Antwortet mit dem Zeilenendezeichen.
lineDelimiter: *aSeqCollection*	Setzt Zeilenendezeichen.
position	Gibt die aktuelle Position zurück.
position: *anInteger*	Setzt die aktuelle Position neu.
reset	Setzt die aktuelle Position auf den Streamanfang.
setToEnd	Setzt die aktuelle Position auf das Streamende.
size	Gibt die Anzahl der Byte des Streams zurück.
skip: *anInteger*	Verschiebt den Zeiger um *anInteger* Objekte.

6.3.1 CfsReadFileStream

Klassenmethoden (Auswahl)

open: *pathString*	Öffnet einen CfsReadFileStream mit dem *pathString*.

Beispiele

```
|readStream|
(readStream := CfsReadFileStream
    open: 'c:\example')
    isCfsError ifTrue: [^self error: readStream message ].
readStream close                                ⇒ a CfsReadFileStream

|readStream|
"Datei 'not' existiert nicht."
(readStream := CfsReadFileStream
    open: 'c:\not')
    isCfsError ifTrue: [^self error: readStream message ]. ⇒ No such file or directory.
```

readStream close

Instanzmethoden (Auswahl)

contents	Gibt eine Collection aller Streamobjekte zurück.
copyFrom: *fromInteger* **to:** *toInteger*	Liest alle Streamobjekte im angegebenen Bereich.
do: *aBlock*	Übergibt dem Block *aBlock* alle Elemente vom Zeiger beginnend bis zum Streamende.
next	Liest das aktuelle Element und setzt den Zeiger auf das nächste Element, sonst Error.
next: *anInteger*	Liest die nächsten *anInteger* Streamobjekte und setzt den Zeiger weiter.
nextLine	Liest aller Streamobjekte bis zum Zeilenende und setzt den Zeiger weiter.
peek	Liest das nächste Objekt, setzt den Zeiger nicht weiter.
upTo: *anObject*	Liest die nächsten Objekte bis ausschließlich *anObject* und setzt den Zeiger hinter *anObject*.
upToEnd	Setzt die aktuelle Position auf das Streamende und gibt die dabei überlesenen Objekte als Collection zurück.

Alle Lesemethoden beginnen an der Zeigerposition und liefern evtl. eine Collection (als ByteArray bzw. String) von Streamobjekten.

6.3.1.1 CfsReadWriteFileStream

Erbt alle Methoden der Klasse **CfsReadFileStream** und besitzt zusätzlich analoge Methoden wie die Klasse **CfsWriteFileStream**.

Klassenmethoden (Auswahl)

openEmpty: *pathString*	Öffnet einen CfsReadWriteFileStream mit dem *pathString*.

Instanzmethoden (Auswahl)

cr	Setzt Zeilenendezeichen und positioniert den Zeiger dahinter.
flush	Löscht Puffer.
nextPut: *anObject*	Schreibt *anObject* und setzt den Zeiger weiter.
next: *anInteger* **put:** *anObject*	Schreibt *anInteger*-Mal das Objekt *anObject* und setzt den Zeiger dahinter.
nextPutAll: *aCollection*	Schreibt alle Objekte aus *aCollection* und setzt den Zeiger weiter.
space	Schreibt ein Leerzeichen und setzt den Zeiger dahinter.
tab	Schreibt ein Tabulator und setzt den Zeiger dahinter.

Alle Lesemethoden beginnen an der Zeigerposition und liefern evtl. eine Collection von Streamobjekten. Alle Schreibaktionen beginnen an der aktuellen Position im Stream.

Beispiel:

```
"Schreiben und Lesen des Inhalts einer Datei."
|stream text|
(stream := CfsReadWriteFileStream
    openEmpty: 'c:\example')
    isCfsError ifTrue: [^self error: stream message ].
"Ueberschreibend !"
stream nextPutAll: 'Hallo !'; cr.
stream nextPutAll: 'Wie geht es ?'.
text := stream contents.
stream close.
^text                                    ⇒   'Hallo !
                                             Wie geht es ?'

c:\example:                              ⇒   'Hallo !
                                             Wie geht es ?'
```

6.3.2 CfsWriteFileStream

Klassenmethoden (Auswahl)

open: *pathString*	Öffnet einen CfsWriteFileStream mit dem *pathString*.
openEmpty: *pathString*	Öffnet einen CfsWriteFileStream mit dem *pathString*.

Instanzmethoden (Auswahl)

cr	Setzt Zeilenendezeichen und positioniert den Zeiger dahinter.
flush	Leert den Puffer.
nextPut: *anObject*	Schreibt *anObject* und setzt den Zeiger weiter.
next: *anInteger* **put:** *anObject*	Schreibt *anInteger*-Mal das Objekt *anObject* und setzt den Zeiger dahinter.
nextPutAll: *aCollection*	Schreibt alle Objekte aus *aCollection* und setzt den Zeiger weiter.
space	Schreibt ein Leerzeichen und setzt den Zeiger dahinter.
tab	Schreibt ein Tabulator und setzt den Zeiger dahinter.

Alle Schreibaktionen beginnen an der aktuellen Position im Stream.

Beispiele:

```
"Schreiben auf eine Datei mit 'printOn'."
|writeStream str|
"Datei 'print_on' existiert nicht, wird neu angelegt."
(writeStream := CfsWriteFileStream
    open: 'c:\print_on')
    isCfsError ifTrue: [^self error: writeStream message ].
str := 'abcd'.
str printOn: writeStream.
writeStream close                        ⇒ a CfsWriteFileStream

c:\print_on:                             ⇒   'abcd'
```

```
"Kopieren einer Datei unter Verwendung von Systemfenstern."
|old new|
(old := CfsReadFileStream
   "Oeffnet Systemfenster zum Markieren der alten Datei."
   open: (CwFileSelectionPrompter new prompt))
   isCfsError ifTrue: [^self error: old message ].
(new := CfsWriteFileStream
   "Oeffnet Systemfenster zum Eingeben des neuen Dateinamens,
   evtl. mit Pfadangabe."
   open: (System prompt: 'Output file name'))
   isCfsError ifTrue: [^self error: new message ].
new nextPutAll: old contents.
old close.
new close                                    ⇒ a CfsWriteFileStream

"Konvertiert Dos-Dateien in Unix-Dateien."
|input output str|
str := CwFileSelectionPrompter new
   title: 'Enter name of DOS format input file'; prompt.
(input := CfsReadFileStream open: str) isCfsError
   ifTrue: [^self error input message].
input lineDelimiter.

str := System
   prompt: 'Enter name of UNIX format output file'.
(output := CfsWriteFileStream open: str) isCfsError
   ifTrue: [^self error output message].
output lineDelimiter: UNIXLineDelimiter.

[input atEnd]
   whileFalse: [output nextPutAll: (input nextLine); cr].
input close.
output close                                 ⇒ a CfsWriteFileStream
```