TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 08: Streams und Files

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Klasse Optional

Bevor wir zum eigentlichen Thema kommen, müssen wir zuerst eine Hilfsklasse namens Optional aus dem Package java.lang einführen.

Ein Objekt der generischen Klasse Optional kapselt ein Objekt seines Typparameters ein – wie üblich kann es alternativ auch der symbolische Wert null sein. Und genau darum geht es: Optional bietet relativ bequeme und relativ wenig fehleranfällige Möglichkeiten damit umzugehen, dass eine Referenz auch null sein kann.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Optional<Number> opt1 = Optional.ofNullable ( new Integer ( 123 ) );
Optional<Number> opt2 = Optional.ofNullable ( null );

Number n1 = opt1.get();   //  n1 == 123
Number n2 = opt2.get();   //  NoSuchElementException
Number n3 = opt1.orElseGet ( () -> 0 );
Number n4 = opt2.orElseGet ( () -> 0 );
```

Wie üblich ein kleines illustratives Beispiel.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Optional<Number> opt1 = Optional.ofNullable ( new Integer ( 123 ) );
Optional<Number> opt2 = Optional.ofNullable ( null );

Number n1 = opt1.get();   //  n1 == 123
Number n2 = opt2.get();   //  NoSuchElementException
Number n3 = opt1.orElseGet ( () -> 0 );
Number n4 = opt2.orElseGet ( () -> 0 );
```

Wir richten zwei Variablen vom Typ Optional ein, bei denen der generische Typparameter beispielhaft mit Number instanziiert ist.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Optional<Number> opt1 = Optional.ofNullable ( new Integer ( 123 ) );
Optional<Number> opt2 = Optional.ofNullable ( null );

Number n1 = opt1.get();   //  n1 == 123
Number n2 = opt2.get();   //  NoSuchElementException
Number n3 = opt1.orElseGet ( () -> 0 );
Number n4 = opt2.orElseGet ( () -> 0 );
```

Die Klassenmethode ofNullable von Klasse Optional erzeugt ein Objekt von Klasse Optional und liefert einen Verweis darauf zurück.

***Erinnerung* an Kapitel 06, Abschnitt zu generischen Klassen: Wird eine Klassenmethode einer generischen Klasse aufgerufen, dann werden die generischen Typparameter dieser Klasse nicht hingeschrieben.**

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Optional<Number> opt1 = Optional.ofNullable ( new Integer ( 123 ) );
Optional<Number> opt2 = Optional.ofNullable ( null );

Number n1 = opt1.get();   //  n1 == 123
Number n2 = opt2.get();   //  NoSuchElementException
Number n3 = opt1.orElseGet ( () -> 0 );
Number n4 = opt2.orElseGet ( () -> 0 );
```

Das eine Objekt von Optional kapselt ein Integer-Objekt ein, das wiederum die ganze Zahl 123 einkapselt.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Optional<Number> opt1 = Optional.ofNullable ( new Integer ( 123 ) );
Optional<Number> opt2 = Optional.ofNullable ( null );

Number n1 = opt1.get();   //  n1 == 123
Number n2 = opt2.get();   //  NoSuchElementException
Number n3 = opt1.orElseGet ( () -> 0 );
Number n4 = opt2.orElseGet ( () -> 0 );
```

Das andere Objekt von Optional wird stattdessen mit null initialisiert. Ein Optional-Objekt, das null einkapselt, wird allgemein *leer* genannt.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Optional<Number> opt1 = Optional.ofNullable ( new Integer ( 123 ) );
Optional<Number> opt2 = Optional.ofNullable ( null );

Number n1 = opt1.get();   //  n1 == 123
Number n2 = opt2.get();   //  NoSuchElementException
Number n3 = opt1.orElseGet ( () -> 0 );
Number n4 = opt2.orElseGet ( () -> 0 );
```

Methode get liefert das von Optional eingekapselte Objekt zurück.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Optional<Number> opt1 = Optional.ofNullable ( new Integer ( 123 ) );
Optional<Number> opt2 = Optional.ofNullable ( null );

Number n1 = opt1.get();   //  n1 == 123
Number n2 = opt2.get();   //  NoSuchElementException
Number n3 = opt1.orElseGet ( () -> 0 );
Number n4 = opt2.orElseGet ( () -> 0 );
```

Ist kein Objekt eingekapselt, sondern null, dann wird nichts zurückgeliefert, sondern eine Exception wird geworfen. Damit ist gewährleistet, dass das Programm nicht mit einer Referenz einfach weiterläuft, die nicht auf ein Objekt verweist.

Der Vorteil ist, dass der Fehler möglichst früh auftritt, nämlich schon beim Aufruf von get, so dass er leichter zu seiner Quelle zurückverfolgt werden kann.

Allerdings ist NoSuchElementException von RuntimeException abgeleitet, muss also nicht gefangen werden. Wenn man sich absolut sicher ist, dass ein Objekt und nicht null eingekapselt ist, kann man sich also try-catch sparen, sonst sollte man es sich sicherheitshalber *nicht* sparen.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Optional<Number> opt1 = Optional.ofNullable ( new Integer ( 123 ) );
Optional<Number> opt2 = Optional.ofNullable ( null );

Number n1 = opt1.get();   //  n1 == 123
Number n2 = opt2.get();   //  NoSuchElementException
Number n3 = opt1.orElseGet ( () -> 0 );
Number n4 = opt2.orElseGet ( () -> 0 );
```

Methode orElseGet entspricht eigentlich eher dem Sinn von Optional als Methode get. Denn wenn das Optional-Objekt leer ist, dann wird mit orElseGet eben nicht null zurückgeliefert, auch keine Exception geworfen, sondern es wird statt dessen ein anderer Wert der Instanziierung des Typparameters – hier Number – zurückgeliefert.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Optional<Number> opt1 = Optional.ofNullable ( new Integer ( 123 ) );
Optional<Number> opt2 = Optional.ofNullable ( null );

Number n1 = opt1.get();   //  n1 == 123
Number n2 = opt2.get();   //  NoSuchElementException
Number n3 = opt1.orElseGet ( () -> 0 );
Number n4 = opt2.orElseGet ( () -> 0 );
```

Zum Beispiel ist es häufig sinnvoll, die Zahl 0 als Ersatz für ein nichtexistierendes Objekt von Klasse Number zu nehmen. Das wird durch diesen Parameter der Methode orElseGet realisiert.

Wie Sie sehen, ist der formale Parameter aber nicht einfach vom Typparameter von Optional, also nicht einfach der zurückzuliefernde Wert im Falle eines leeren Optional-Objektes. Nein, der formale Typ ist das Functional Interface Supplier aus Package java.util.function. Der generische Typparameter des Suppliers muss mit demselben Typ instanziiert sein wie Optional selbst.. Die funktionale Methode von Supplier hat keine Parameter und liefert einen Wert dieses Typs zurück. Das ist der Wert, den das zurückgelieferte Optional-Objekt einkapselt. In diesem Beispiel liefert der durch diesen Lambda-Ausdruck definierte Supplier einfach konstant die Zahl 0 zurück.

***Erinnerung*: Abschnitt zu Functional Interfaces und Lambda-Ausdrücken in Java im Kapitel 04c.**

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
opt1.ifPresent ( x -> { System.out.print ( x ); } );
opt2.ifPresent ( x -> { System.out.print ( x ); } );

Optional<Number> opt3 = opt1.map ( x -> x * x );
Optional<Number> opt4 = opt2.map ( x -> x * x );

Optional<Number> opt5a = opt3.filter ( x -> x % 2 == 1 );
Optional<Number> opt5b = opt3.filter ( x -> x % 2 == 0 );
Optional<Number> opt6 = opt4.filter ( x -> x % 2 == 1 );
```

Noch ein paar weitere nützliche Methoden der Klasse Optional, dann sind wir mit dem Thema Optional erst einmal durch.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
opt1.ifPresent ( x -> { System.out.print ( x ); } );
opt2.ifPresent ( x -> { System.out.print ( x ); } );
Optional<Number> opt3 = opt1.map ( x -> x * x );
Optional<Number> opt4 = opt2.map ( x -> x * x );
Optional<Number> opt5a = opt3.filter ( x -> x % 2 == 1 );
Optional<Number> opt5b = opt3.filter ( x -> x % 2 == 0 );
Optional<Number> opt6 = opt4.filter ( x -> x % 2 == 1 );
```

Methode ifPresent realisiert auf wenig fehleranfällige Art das häufige Programmiermuster, dass etwas mit einer Referenz gemacht wird, falls sie auf ein Objekt verweist, aber nichts gemacht wird, falls die Referenz den Wert null hat. Der Parameter sagt, was gemacht werden soll, falls das Optional-Objekt tatsächlich ein Objekt des generischen Typparameters und nicht null einkapselt. Der formale Typ des Parameters ist Consumer aus Package java.util.function, instanziiert mit demselben generischen Typ wie Optional. Der aktuale Parameter kann daher wieder ein Lambda-Ausdruck sein.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
opt1.ifPresent ( x -> { System.out.print ( x ); } );
opt2.ifPresent ( x -> { System.out.print ( x ); } );
Optional<Number> opt3 = opt1.map ( x -> x * x );
Optional<Number> opt4 = opt2.map ( x -> x * x );
Optional<Number> opt5a = opt3.filter ( x -> x % 2 == 1 );
Optional<Number> opt5b = opt3.filter ( x -> x % 2 == 0 );
Optional<Number> opt6 = opt4.filter ( x -> x % 2 == 1 );
```

Analog dazu die Ausführung einer Funktion nur im Falle, dass das Optional-Objekt nicht leer ist. In diesem Beispiel war ja ein Objekt von Number eingekapselt in demjenigen Optional-Objekt, auf das opt1 verweist. Daher verweist nun opt3 auf ein Optional-Objekt, das ein Number-Objekt einkapselt, welches das Quadrat der Zahl von opt1 enthält. Im Gegensatz dazu ist opt4 leer, da opt2 leer ist.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
opt1.ifPresent ( x -> { System.out.print ( x ); } );
opt2.ifPresent ( x -> { System.out.print ( x ); } );

Optional<Number> opt3 = opt1.map ( x -> x * x );
Optional<Number> opt4 = opt2.map ( x -> x * x );

Optional<Number> opt5a = opt3.filter ( x -> x % 2 == 1 );
Optional<Number> opt5b = opt3.filter ( x -> x % 2 == 0 );
Optional<Number> opt6 = opt4.filter ( x -> x % 2 == 1 );
```

Methode filter liefert ein Optional vom selben generischen Typ zurück. Formaler Parameter ist Predicate aus Package java.util.function mit demselben generischen Typ.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
opt1.ifPresent ( x -> { System.out.print ( x ); } );
opt2.ifPresent ( x -> { System.out.print ( x ); } );

Optional<Number> opt3 = opt1.map ( x -> x * x );
Optional<Number> opt4 = opt2.map ( x -> x * x );

Optional<Number> opt5a = opt3.filter ( x -> x % 2 == 1 );
Optional<Number> opt5b = opt3.filter ( x -> x % 2 == 0 );
Optional<Number> opt6 = opt4.filter ( x -> x % 2 == 1 );
```

Hier wird Methode filter mit einem Optional aufgerufen, das ein Objekt einkapselt, und dieses Objekt passiert den Filter, denn 123 ist ungerade. In diesem Fall liefert filter einen Verweis auf ein Optional-Objekt zurück, das dasselbe Number-Objekt einkapselt.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
opt1.ifPresent ( x -> { System.out.print ( x ); } );
opt2.ifPresent ( x -> { System.out.print ( x ); } );

Optional<Number> opt3 = opt1.map ( x -> x * x );
Optional<Number> opt4 = opt2.map ( x -> x * x );

Optional<Number> opt5a = opt3.filter ( x -> x % 2 == 1 );
Optional<Number> opt5b = opt3.filter ( x -> x % 2 == 0 );
Optional<Number> opt6 = opt4.filter ( x -> x % 2 == 1 );
```

Passiert das Objekt den Filter hingegen *nicht* – so wie in dieser Zeile –, dann wird ein leeres Optional-Objekt zurückgeliefert.

Klasse Optional

TECHNISCHE UNIVERSITÄT DARMSTADT

```
opt1.ifPresent ( x -> { System.out.print ( x ); } );
opt2.ifPresent ( x -> { System.out.print ( x ); } );

Optional<Number> opt3 = opt1.map ( x -> x * x );
Optional<Number> opt4 = opt2.map ( x -> x * x );

Optional<Number> opt5a = opt3.filter ( x -> x % 2 == 1 );
Optional<Number> opt5b = opt3.filter ( x -> x % 2 == 0 );
Optional<Number> opt6 = opt4.filter ( x -> x % 2 == 1 );
```

Schlussendlich wird auch bei einem leeren Optional-Objekt wieder ein leeres Optional-Objekt zurückgeliefert.

TECHNISCHE UNIVERSITÄT DARMSTADT

Streams

Nach dieser Vorarbeit können wir zum ersten Hauptthema des Kapitels kommen.

Das Interface Stream ist generisch und findet sich im Package java.util.stream. Streams bilden in gewisser Weise eine einheitliche Schnittstelle für Listen, Arrays, Dateien sowie endlichen und unendlichen Sequenzen von Werten des Typparameters.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Number> list = new LinkedList<Number>();
for ( int i = 0; i < 100; i++ )
    list.add ( new Integer ( 3 + 4 * i ) );
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
    = stream3.max ( new MyNumberComparator() );
```

Als erstes ein Beispiel für Streams generiert aus Listen.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Number> list = new LinkedList<Number>();
for ( int i = 0; i < 100; i++ )
    list.add ( new Integer ( 3 + 4 * i ) );
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
    = stream3.max ( new MyNumberComparator() );
```

Dazu richten wir beispielhaft eine Liste mit einhundert Elementen ein.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Number> list = new LinkedList<Number>();
for ( int i = 0; i < 100; i++ )
    list.add ( new Integer ( 3 + 4 * i ) );
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
    = stream3.max ( new MyNumberComparator() );
```

Das ist die besagte generische Klasse Stream aus dem Package java.util.stream, hier instanziiert mit Number.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Number> list = new LinkedList<Number>();
for ( int i = 0; i < 100; i++ )
    list.add ( new Integer ( 3 + 4 * i ) );
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
    = stream3.max ( new MyNumberComparator() );
```

Die generische Klasse List aus Package java.util hatten wir schon im Kapitel 07 kennen gelernt. Sie hat eine Methode stream ohne Parameter, die einen Stream vom selben generischen Typ, den die Liste selbst hat, zurückliefert. Der zurückgelieferte Stream besteht aus denselben Elementen wie die Liste, und auch in derselben Reihenfolge. Bis hierhin ist ein Stream also erst einmal nur eine andere Zugriffsweise auf die Elemente der Liste. Wir werden gleich sehen, dass damit ein einheitlicher Zugriff auf *verschiedene* Datenstrukturen und sogar Dateien möglich ist.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Number> list = new LinkedList<Number>();
for ( int i = 0; i < 100; i++ )
    list.add ( new Integer ( 3 + 4 * i ) );
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
    = stream3.max ( new MyNumberComparator() );
```

Das Interface Stream hat unter anderem eine Methode filter. Sie liefert einen Stream vom selben generischen Typ zurück. Der formale Parameter ist Predicate, definiert in java.util.function. Das ist das Prädikat, nach dem gefiltert wird. Es muss mit demselben Typparameter wie der Stream selbst instanziiert sein, hier also mit Number. Der zurückgelieferte Stream ist gleich dem Stream, mit dem filter aufgerufen wurde, nur dass die Elemente, die den Filter *nicht* passieren – also diejenigen, bei denen das Prädikat false liefert – nicht mehr vorhanden sind.

***Erinnerung*: Interface Predicate hatten wir schon im Kapitel 04c gesehen, Abschnitt zu Functional Interfaces und Lambda-Ausdrücken.**

***Erinnerung an Racket*: Funktion filter auf Listen in Racket ist völlig analog zu dieser Methode filter auf Java-Streams.**

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Number> list = new LinkedList<Number>();
for ( int i = 0; i < 100; i++ )
    list.add ( new Integer ( 3 + 4 * i ) );
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
    = stream3.max ( new MyNumberComparator() );
```

***Erinnerung*: In Kapitel 06 hatten wir gesehen, dass sowohl ganze Klassen und Interfaces als auch einzelne Methoden generisch parametrisiert sein können.**

Methode map ist nun selbst ebenfalls generisch, hat also neben dem Typparameter der Klasse Stream noch einen weiteren Typparameter. Rückgabe ist ein Stream des zweiten Typparameters, das heißt, potentiell sind Eingabe- und Ausgabe-Stream mit verschiedenen Typen instanziiert. In diesem Beispiel sind sie gleich instanziiert, nämlich mit Number. Der formale Parameter ist Function, definiert in java.util.function. Beide Typparameter von Function sind in diesem Beispiel also instanziiert mit Number.

***Erinnerung an Racket*: Funktion map auf Listen in Racket ist völlig analog zu dieser Methode map auf Java-Streams.**

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Number> list = new LinkedList<Number>();
for ( int i = 0; i < 100; i++ )
    list.add ( new Integer ( 3 + 4 * i ) );
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
    = stream3.max ( new MyNumberComparator() );
```

Die Methode max von Klasse Stream liefert jetzt keinen Stream mehr, sondern nur noch genau ein Element desjenigen Streams zurück, mit dem die Methode max aufgerufen wurde.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Number> list = new LinkedList<Number>();
for ( int i = 0; i < 100; i++ )
    list.add ( new Integer ( 3 + 4 * i ) );
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
    = stream3.max ( new MyNumberComparator() );
```

Erinnerung: In Kapitel 06 hatten wir einen Abschnitt zum generischen Interface Comparator und dort auch die Instanziierung von Comparator mit Number durch eine eigene Klasse MyNumberGenerator implementiert, und zwar als gewöhnlichen Größenvergleich auf Zahlen.

Der formale Parameter von Methode max ist Comparator vom generischen Typ des Streams. Der aktuale Parameter entscheidet, welches Element des Streams zurückgeliefert wird, nämlich dasjenige Element, das gemäß der implementierten Vergleichsfunktion allen anderen Elementen des Streams nachfolgend ist. Bei MyNumberGenerator wird also das größte Element eines Zahlenstreams zurückgeliefert.

Nebenbemerkung: Zumindest in der Oracle-Dokumentation ist nicht spezifiziert, welches Element zurückgeliefert wird, falls es mehrere größte Elemente gibt. Bei Tests auf Wertgleichheit macht das natürlich keinen Unterschied, bei Tests auf Objektidentität allerdings schon.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Number> list = new LinkedList<Number>();
for ( int i = 0; i < 100; i++ )
    list.add ( new Integer ( 3 + 4 * i ) );
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
     = stream3.max ( new MyNumberComparator() );
```

Zurückgeliefert wird nicht das Element selbst, sondern der Verweis auf ein Optional-Objekt, das dieses Element einkapselt. Ist der Stream leer, dann ist auch dieses Optional-Objekt leer. Der Fall, dass ein Stream auch leer sein kann, ist der Grund dafür, hier Optional zu verwenden.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Number> list = new LinkedList<Number>();
for ( int i = 0; i < 100; i++ )
    list.add ( new Integer ( 3 + 4 * i ) );
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
     = stream3.max ( new MyNumberComparator() );
```

Methoden von Klasse Stream, die wie filter und map wieder einen Stream zurückliefern, nennt man im Englischen intermediate operations, zu Deutsch also so etwas wie Zwischenoperationen.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Number> list = new LinkedList<Number>();
for ( int i = 0; i < 100; i++ )
    list.add ( new Integer ( 3 + 4 * i ) );
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
    = stream3.max ( new MyNumberComparator() );
```

Und Methode max ist ein Beispiel für terminal operations, also deutsch Terminaloperationen oder Terminalmethoden. Eine Terminaloperation ist natürlich nur wie hier am Ende einer Kette von Zwischenoperationen sinnvoll.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt = stream3.max ( new MyNumberComparator() );
```

```
Optional<Number> opt
    = list.stream().filter ( myPred )
                   .map ( myFct )
                   .max ( new MyNumberComparator() );
```

Diese einzelnen Schritte – also mehrere intermediate operations gefolgt von einer terminal operation, lassen sich auch in einer einzigen Anweisung zusammenfassen.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt = stream3.max ( new MyNumberComparator() );
```

```
Optional<Number> opt
    = list.stream().filter ( myPred )
                   .map ( myFct )
                   .max ( new MyNumberComparator() );
```

Oben etwas kleiner gedruckt sind noch einmal die vier Schritte von der ursprünglichen Liste bis zum beendenden Optional von der letzten Folie übernommen.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Stream<Number> stream1 = list.stream();
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt = stream3.max ( new MyNumberComparator() );
```

```
Optional<Number> opt
    = list.stream().filter ( myPred )
                   .map ( myFct )
                   .max ( new MyNumberComparator() );
```

Und unten sehen Sie besagte Zusammenfassung der vier Schritte in einer einzigen Anweisung. Wie auch sonst, müssen auch hier Rückgaben eben nicht unbedingt in Variablen oder Konstanten zwischengespeichert, sondern können stattdessen sofort für den jeweils nächsten Schritt weiterverwendet werden, so dass eine solche Verkettung möglich ist.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Number [ ] a = new Number [ 100 ];
for ( int i = 0; i < 100; i++ )
    add[i] = 3 + 4 * i;
Stream<Number> stream1 = Arrays.stream ( a );
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
    = stream3.max ( new MyNumberComparator );
```

Jetzt exakt dasselbe noch einmal mit dem einzigen Unterschied, dass der Stream jetzt nicht aus einer *Liste*, sondern aus einem *Array* gebildet wird.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Number [ ] a = new Number [ 100 ];
for ( int i = 0; i < 100; i++ )
    add[i] = 3 + 4 * i;
Stream<Number> stream1 = Arrays.stream ( a );
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
    = stream3.max ( new MyNumberComparator );
```

Analog zum Listenbeispiel nun eben ein *Array* mit einhundert Komponenten.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Number [ ] a = new Number [ 100 ];
for ( int i = 0; i < 100; i++ )
    add[i] = 3 + 4 * i;
Stream<Number> stream1 = Arrays.stream ( a );
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
    = stream3.max ( new MyNumberComparator );
```

Die Klasse Arrays im Package java.lang ist eine Sammlung von nützlichen Klassenmethoden rund um Arrays. Die generische Klassenmethode stream von Arrays hat einen Parameter von einem Arraytyp und liefert einen Stream zurück, der mit dem Komponententyp des aktualen Parameters instanziiert ist und dessen Elemente genau die Arraykomponenten in aufsteigender Reihenfolge der Arrayindizes sind.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Number [ ] a = new Number [ 100 ];
for ( int i = 0; i < 100; i++ )
    add[i] = 3 + 4 * i;
Stream<Number> stream1 = Arrays.stream ( a );
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
     = stream3.max ( new MyNumberComparator );
```

Die weiteren Schritte – also die intermediate operations und die terminal operation – sehen exakt genauso aus wie vorher bei Listen. Streams sind also eine Möglichkeit, den Inhalt beliebiger Sequenzen auf identische Weise weiterzuverarbeiten.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Stream<Number> stream1 = Arrays.stream ( a );
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt = stream3.max ( new MyNumberComparator() );
```

```
Optional<Number> opt = Arrays.stream(a)
                       .filter ( myPred )
                       .map ( myFct )
                       .max ( new MyNumberComparator() );
```

Und auch die Zusammenfassung der einzelnen Schritte in einer einzigen Anweisung sieht praktisch identisch aus.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Stream<Number> stream1 = Arrays.stream ( a );
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt = stream3.max ( new MyNumberComparator() );
```

```
Optional<Number> opt = Arrays.stream(a)
                       .filter ( myPred )
                       .map ( myFct )
                       .max ( new MyNumberComparator() );
```

Nur dass eben die Erzeugung des initialen Streams aus einem Array anders aussieht als aus einer Liste.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Stream<Number> stream1
    = Stream.of ( new Integer(1), new Integer(2) );

Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
     = stream3.max ( new MyNumberComparator() );
```

Den Inhalt eines Streams kann man auch ohne Umweg über eine Liste oder einen Array definieren.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Stream<Number> stream1
    = Stream.of ( new Integer(1), new Integer(2) );
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
     = stream3.max ( new MyNumberComparator() );
```

Klasse Stream hat eine Klassenmethode namens of mit einer beliebigen Anzahl von Parametern des Typparameters. Zurückgeliefert wird ein Stream mit genau diesen Parametern in genau dieser Reihenfolge als Elementen. In diesem Beispiel hat der Stream nur zwei Elemente, damit alles noch gut auf die Folie passt.

***Erinnerung* an Kapitel 06: Klassenmethoden bleiben bei Generizität außen vor, daher beim Aufruf kein generischer Typparameter.**

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Stream<Number> stream1
    = Stream.of ( new Integer(1), new Integer(2) );
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt
    = stream3.max ( new MyNumberComparator() );
```

Danach sind die einzelnen weiteren Schritte absolut identisch zu Streams aus Listen oder aus Arrays.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Stream<Number> stream1 = Stream.of ( ..... );
Stream<Number> stream2 = stream1.filter ( myPred );
Stream<Number> stream3 = stream2.map ( myFct );
Optional<Number> opt = stream3.max ( new MyNumberComparator() );
```

```
Optional<Number> opt = Stream.of ( ..... )
                       .filter ( myPred )
                       .map ( myFct )
                       .max ( new MyNumberComparator() );
```

Nur dass auch hier die initiale Erzeugung des ersten Streams anders aussieht. Für die Punkte können jeweils beliebig viele aktuale Parameter vom Typ Number oder einem Subtyp von Number eingesetzt werden. Natürlich dürfen diese Typen unter dieser Voraussetzung auch gemischt sein.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static Optional<Number> m ( Stream<Number> stream ) {
   return stream.filter ( myPred )
                .map ( myFct )
                .max ( new MyNumberComparator() );
}

X.m ( list.stream() );
X.m ( Arrays.stream(a) );
X.m ( Stream.of ( new Integer(1), new Integer(2) ) );
```

Wenn die weiteren Schritte nach Erzeugung des ersten, initialen Stream also immer dieselben sind, egal wie der Stream erzeugt wurde, dann kann man diese Schritte auch in eine Methode herausfaktorisieren.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static Optional<Number> m ( Stream<Number> stream ) {
    return stream.filter ( myPred )
                 .map ( myFct )
                 .max ( new MyNumberComparator() );
}

X.m ( list.stream() );
X.m ( Arrays.stream(a) );
X.m ( Stream.of ( new Integer(1), new Integer(2) ) );
```

Die Methode heiße der Einfachheit halber m und sei eine Klassenmethode einer Klasse X.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static Optional<Number> m ( Stream<Number> stream ) {
   return stream.filter ( myPred )
                .map ( myFct )
                .max ( new MyNumberComparator() );
}

X.m ( list.stream() );
X.m ( Arrays.stream(a) );
X.m ( Stream.of ( new Integer(1), new Integer(2) ) );
```

Die einzelnen Schritte nach Erzeugung des Streams sind dann der Inhalt der Methode.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static Optional<Number> m ( Stream<Number> stream ) {
   return stream.filter ( myPred )
                .map ( myFct )
                .max ( new MyNumberComparator() );
}

X.m ( list.stream() );
X.m ( Arrays.stream(a) );
X.m ( Stream.of ( new Integer(1), new Integer(2) ) );
```

Der Unterschied zwischen den einzelnen Fällen manifestiert sich in den aktualen Parametern bei verschiedenen Aufrufen der Methode.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Iterator iter = stream.iterator();

while ( iter.hasNext() ) {
    Number n = iter.next();
    doSomethingWith ( n );
}
```

***Erinnerung*: In Kapitel 07 haben wir das Interface Iterator im gleichnamigen Abschnitt kennen gelernt und gesehen, wie man mit einem Iterator elementweise durch eine Collection gehen kann.**

Wie wir jetzt hier sehen, bieten Streams dieselbe Möglichkeit für elementweisen Zugriff in der Reihenfolge, in der die Elemente im Stream sind. Auch hier also wieder maximale Gleichheit der Vorgehensweise, nur die Generierung des Iterators unterscheidet sich, je nachdem, worauf der Iterator laufen soll.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<String> list = stream.collect ( Collectors.toList() );

Number [ ] a = stream.toArray ( Number[ ]::new );
```

Wir haben gesehen, dass Streams aus Listen und Arrays erzeugt werden können. Als nächstes sehen wir, dass das auch umgekehrt möglich ist.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<String> list = stream.collect ( Collectors.toList() );

Number [ ] a = stream.toArray ( Number[ ]::new );
```

Klasse Collectors – mit s hinten – ist eine nützliche Sammlung von Klassenmethoden, die verschiedene gängige Möglichkeiten zur Weiterverarbeitung von Streams bereitstellt. Methode collect von Klasse Stream hat als formalen Parameter genau das, was Methode toList von Collectors zurückliefert, nämlich das generische Interface Collector – *ohne* s. Beide – Klasse Collectors und Interface Collector – finden sich im Package java.util.stream. Wir gehen auf Collectors und Collector hier nicht näher ein, sondern verweisen auf die Dokumentation beider Referenztypen.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<String> list = stream.collect ( Collectors.toList() );

Number [ ] a = stream.toArray ( Number[ ]::new );
```

Die Objektmethode toArray von Klasse Stream ist generisch. Sie hat einen Parameter vom Typ IntFunction, ein generisches Functional Interface aus java.util.function. Die funktionale Methode bekommt ein int und liefert einen Array vom generischen Typ zurück. In dem konkret hier gezeigten Beispiel ist Number der generische Typ. Operator new von Klasse Number in der Array-Variante passt also.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<String> list = stream.collect ( Collectors.toList() );

Number [ ] a = stream.toArray ( Number[ ]::new );
```

Wie das Array konkret erzeugt wird, lässt sich durch den Parameter von Methode toArray steuern.

Erinnerung: Kapitel 04c, Abschnitt zu Methodennamen als Lambda-Ausdrücke.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( list.equals ( list.stream().collect ( Collectors.toList() ) )

if ( Arrays.equals
        ( a, Arrays.stream(a).toArray ( Number[ ]::new ) ) )
```

Noch eine wichtige Erkenntnis: Die beiden Ausdrücke zur Erzeugung eines Liste beziehungsweise eines Arrays aus einem Stream sind exakt die Umkehrung zu den vorher schon eingeführten Ausdrücken zur Erzeugung eines Streams aus einer Liste beziehungsweise aus einem Array. Daher liefern die beiden booleschen Ausdrücke auf dieser Folie true zurück.

Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( list.equals ( list.stream().collect ( Collectors.toList() ) )

if ( Arrays.equals
        ( a, Arrays.stream(a).toArray ( Number[ ]::new ) ) )
```

***Erinnerung*: Wie wir in Kapitel 03b, Abschnitt zu Objektidentität vs. Wertgleichheit, gesehen haben, liefert der Vergleichsoperator == genau dann true zurück, wenn beide Referenzen auf dasselbe Objekt verweisen. Ansonsten liefert er auch bei Wertgleichheit false zurück. Für den Test auf Wertgleichheit gibt es Methode equals.**

***Nebenbemerkung*: Arraytypen selbst haben ebenfalls grundlegende Methoden wie equals aus Klasse Object. Diese sollte man aber nur verwenden, wenn man genau weiß, was man tut. Wie hier gezeigt, bietet Klasse Arrays in java.util ebenfalls eine Methode equals, aber mit der üblichen Bedeutung, und deshalb nehmen wir hier diese.**

TECHNISCHE UNIVERSITÄT DARMSTADT

System Properties

Wir sind mit Streams erst einmal soweit durch und kommen als nächstes dann zu Dateien. Vorab müssen wir uns aber Klasse java.lang.System noch einmal kurz genauer ansehen. Wir kennen Klasse System durch Bildschirmausgaben mit System.out.print.

Das Stichwort jetzt lautet system properties. Gemeint sind Attribute der Umgebung, in der das Java-Programm abläuft. Es geht um Attribute, die bei jedem Ablauf eines Programms unterschiedlich sein können beziehungsweise sich während des Ablaufs sogar teilweise ändern können, entweder durch externe Einwirkung oder durch das Programm selbst. Daher kann man diese Attribute nicht sinnvoll im Programm konstant festlegen, sondern muss sie immer abfragen, wenn man sie braucht. Klasse System bietet die Möglichkeit zum Abfragen derjenigen Attribute, die sich im Laufe der Jahrzehnte als besonders abfragenswert herauskristallisiert haben. Auch *wir* werden mit diesen auskommen.

System Properties

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDirectory
    = System.getProperty ( “user.home“ );
String workingDirectory
    = System.getProperty ( “user.dir“ );
String accountName
    = System.getProperty ( “user.name“ );
String fileSeparator
    = System.getProperty ( “file.separator“ );
String lineSeparator
    = System.getProperty ( “line.separator“ );
```

Konkret interessieren uns in diesem Kapitel nur diese fünf Attribute.

System Properties

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDirectory
    = System.getProperty ( “user.home“ );
String workingDirectory
    = System.getProperty ( “user.dir“ );
String accountName
    = System.getProperty ( “user.name“ );
String fileSeparator
    = System.getProperty ( “file.separator“ );
String lineSeparator
    = System.getProperty ( “line.separator“ );
```

Die Klassenmethode getProperty von Klasse System erhält als Parameter einen String und liefert einen String zurück. Nur bei bestimmten Strings als Parameter wird ein sinnvoller String zurückgeliefert, ansonsten wird null zurückgeliefert.

***Nebenbemerkung*: Sollte das Programm nicht das Recht haben, auf eines dieser Attribute zuzugreifen, wird beim Versuch eine SecurityException geworfen. Das betrifft uns hier in der FOP allerdings nicht. Und da SecurityException von RuntimeException abgeleitet ist, können wir das völlig ignorieren.**

***Erinnerung*: In Kapitel 05 haben wir gesehen, dass Exceptions immer gefangen oder weitergereicht werden müssen, mit einer Ausnahme: Exceptions von Klasse RuntimeException oder davon direkt oder indirekt abgeleiteten Exception-Klassen dürfen einfach ignoriert werden – führen aber zum Programmabbruch, wenn sie unerwartet *doch* geworfen werden.**

System Properties

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDirectory
    = System.getProperty ( “user.home“ );
String workingDirectory
    = System.getProperty ( “user.dir“ );
String accountName
    = System.getProperty ( “user.name“ );
String fileSeparator
    = System.getProperty ( “file.separator“ );
String lineSeparator
    = System.getProperty ( “line.separator“ );
```

Mit diesem Wert des Parameters wird der Name des Heimatverzeichnisses des Nutzers zurückgeliefert, in dessen Namen das Programm läuft. Zumindest in der FOP ist das in der Regel der Nutzer, der das Programm ausführen lässt.

System Properties

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDirectory
    = System.getProperty ( “user.home“ );
String workingDirectory
    = System.getProperty ( “user.dir“ );
String accountName
    = System.getProperty ( “user.name“ );
String fileSeparator
    = System.getProperty ( “file.separator“ );
String lineSeparator
    = System.getProperty ( “line.separator“ );
```

Das ist das momentane Arbeitsverzeichnis des Prozesses, englisch working directory.

Siehe Zusammenfassung Programme und Prozesse.

System Properties

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDirectory
    = System.getProperty ( “user.home“ );
String workingDirectory
    = System.getProperty ( “user.dir“ );
String accountName
    = System.getProperty ( “user.name“ );
String fileSeparator
    = System.getProperty ( “file.separator“ );
String lineSeparator
    = System.getProperty ( “line.separator“ );
```

Der Name des Nutzers im System.

System Properties

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDirectory
    = System.getProperty ( “user.home“ );
String workingDirectory
    = System.getProperty ( “user.dir“ );
String accountName
    = System.getProperty ( “user.name“ );
String fileSeparator
    = System.getProperty ( “file.separator“ );
String lineSeparator
    = System.getProperty ( “line.separator“ );
```

Diese Information ist systemspezifisch und bleibt auf jedem gängigen System immer konstant. Zurückgeliefert wird das Zeichen, das in Pfadnamen für Objekte im Dateisystem zur Trennung zwischen den einzelnen Bestandteilen steht. Da Rückgabe ein String ist, könnte theoretisch eine Abfolge von mehreren Zeichen dafür verwendet werden. in der Praxis läuft es fast immer auf den Slash “/“ bei UNIX-Systemen beziehungsweise den Backslash “\“ bei Microsoft-Systemen hinaus.

System Properties

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDirectory
    = System.getProperty ( “user.home“ );
String workingDirectory
    = System.getProperty ( “user.dir“ );
String accountName
    = System.getProperty ( “user.name“ );
String fileSeparator
    = System.getProperty ( “file.separator“ );
String lineSeparator
    = System.getProperty ( “line.separator“ );
```

In Textdaten werden auf den verschiedenen Systemen ebenfalls unterschiedliche Zeichen oder Zeichenfolgen verwendet, um das Ende einer Zeile zu markieren. Wenn Sie beispielsweise einen Editor für Textdateien schreiben, der auf verschiedenen Systemen korrekt laufen soll, müssen Sie bei der Anzeige der Datei im Edierfenster genau an den Stellen eine neue Zeile öffnen, an denen dass Zeichen beziehungsweise die Zeichenfolge steht, die hier durch getProperty zurückgeliefert wird.

TECHNISCHE UNIVERSITÄT DARMSTADT

Streams und Dateien

Wir sind immer noch bei Streams, kommen aber nun zur Verbindung mit Dateien.

Streams und Dateien

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDir = System.getProperty ( “user.home“ );
String fileSep = System.getProperty ( “file.separator“ );
Path path = Paths.get ( homeDir, “fop“, “streams.txt“ );
try ( Stream<String> stream = Files.lines ( path ) ) {
   String fileContentAsString = stream.reduce ( String::concat );
} catch ( IOException exc ) {
   System.out.print ( “Could not open file “ + fileSep + homeDir +
                              fileSep + “fop“ + fileSep + “streams.txt“ + “!“ );
}
```

Wie immer ein einfaches, kleines Beispiel zur Ilustration.

Streams und Dateien

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDir = System.getProperty ( “user.home“ );
String fileSep = System.getProperty ( “file.separator“ );
Path path = Paths.get ( homeDir, “fop“, “streams.txt“ );
try ( Stream<String> stream = Files.lines ( path ) ) {
   String fileContentAsString = stream.reduce ( String::concat );
} catch ( IOException exc ) {
   System.out.print ( “Could not open file “ + fileSep + homeDir +
                              fileSep + “fop“ + fileSep + “streams.txt“ + “!“ );
}
```

Wie eben gesehen, holen wir uns zwei Informationen aus Klasse System mittels Methode getProperty.

Streams und Dateien

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDir = System.getProperty ( “user.home“ );
String fileSep = System.getProperty ( “file.separator“ );
Path path = Paths.get ( homeDir, “fop“, “streams.txt“ );
try ( Stream<String> stream = Files.lines ( path ) ) {
   String fileContentAsString = stream.reduce ( String::concat );
} catch ( IOException exc ) {
   System.out.print ( “Could not open file “ + fileSep + homeDir +
                              fileSep + “fop“ + fileSep + “streams.txt“ + “!“ );
}
```

Die Klassen Path *ohne* s hinten und Paths *mit* s hinten sind in Package java.nio.file zu finden, genauso wie diverse weitere Klassen, die mit Dateien und Dateistrukturen zu tun haben.

Ein Objekt der Klasse Path verwaltet den Pfadnamen einer Datei oder eines Verzeichnisses oder eines anderen Objektes, das im jeweiligen Dateisystem über einen Pfadnamen ansprechbar ist. Zu einem Objekt der Klasse Path muss es kein Objekt im Dateisystem geben, wir kommen gleich darauf zurück, wenn wir mit einem Path eine Datei erst erzeugen.

Klasse Paths *mit* s ist eigentlich nur dazu da, so wie hier ein Objekt der Klasse Path *ohne* s zu erzeugen, und zwar auf Basis derjenigen Informationen, die der Methode get durch die aktualen Parameter übergeben werden.

Streams und Dateien

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDir = System.getProperty ( “user.home“ );
String fileSep = System.getProperty ( “file.separator“ );
Path path = Paths.get ( homeDir, “fop“, “streams.txt“ );
try ( Stream<String> stream = Files.lines ( path ) ) {
   String fileContentAsString = stream.reduce ( String::concat );
} catch ( IOException exc ) {
   System.out.print ( “Could not open file “ + fileSep + homeDir +
                              fileSep + “fop“ + fileSep + “streams.txt“ + “!“ );
}
```

Methode get von Klasse Paths ist überladen. Die hier verwendete Variante der Methode get hat eine variable Parameterzahl vom formalen Typ String, wobei die Parameterliste nicht leer sein darf. Um eine nichtleere Parameterliste zu erzwingen, hat get *zwei* Parameter: einen vom Typ String und einen vom Typ „String...“. Der hier gezeigte Aufruf von get konstruiert ein Path-Objekt, bei dem an den Pfad des Heimatverzeichnisses die weiteren Pfadbestandteile fop und streams.txt angehängt werden. Offensichtlich soll in diesem Beispiel eine Datei streams.txt in einem direkten Unterverzeichnis fop des Heimatverzeichnisses angesprochen werden.

***Erinnerung*: In Kapitel 03c hatten wir einen Abschnitt zu variabler Parameterzahl.**

Streams und Dateien

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDir = System.getProperty ( “user.home“ );
String fileSep = System.getProperty ( “file.separator“ );
Path path = Paths.get ( homeDir, “fop“, “streams.txt“ );
try ( Stream<String> stream = Files.lines ( path ) ) {
   String fileContentAsString = stream.reduce ( String::concat );
} catch ( IOException exc ) {
   System.out.print ( “Could not open file “ + fileSep + homeDir +
                              fileSep + “fop“ + fileSep + “streams.txt“ + “!“ );
}
```

Erinnerung: In Kapitel 05, speziell bei Exceptions und try-catch-Blöcken, hatten wir schon das Konstrukt try-with-resources gesehen. Der Sinn ist, dass hier Objekte von Klassen erzeugt werden, die das Interface AutoCloseable implementieren. Automatisch wird dann Methode close mit allen diesen Objekten aufgerufen, wenn der try-catch-Block verlassen wird, egal auf welchem Wege das passiert.

Streams und Dateien

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDir = System.getProperty ( “user.home“ );
String fileSep = System.getProperty ( “file.separator“ );
Path path = Paths.get ( homeDir, “fop“, “streams.txt“ );
try ( Stream<String> stream = Files.lines ( path ) ) {
   String fileContentAsString = stream.reduce ( String::concat );
} catch ( IOException exc ) {
   System.out.print ( “Could not open file “ + fileSep + homeDir +
                              fileSep + “fop“ + fileSep + “streams.txt“ + “!“ );
}
```

Klasse Files – mit s hinten – aus Package java.nio.file bietet eine nützliche Sammlung von Klassenmethoden rund um Dateien und Dateistrukturen. Methode lines öffnet die Datei, die durch den Parameter spezifiziert ist. Zurückgeliefert wird ein Stream von String, dessen Inhalt die Zeilen der geöffneten Datei sind: jeweils ein String pro Zeile. Als Kennung für die Identifikation eines Zeilenendes wird natürlich der String verwendet, den Methode getProperty von Klasse System mit aktualem Parameter “line.separator“ zurückliefert.

Diese Methode macht also eher nur Sinn bei Dateien, die tatsächlich Texte enthalten und nicht binäre Daten wie etwa Bilder.

Streams und Dateien

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDir = System.getProperty ( “user.home“ );
String fileSep = System.getProperty ( “file.separator“ );
Path path = Paths.get ( homeDir, “fop“, “streams.txt“ );
try ( Stream<String> stream = Files.lines ( path ) ) {
   String fileContentAsString = stream.reduce ( String::concat );
} catch ( IOException exc ) {
   System.out.print ( “Could not open file “ + fileSep + homeDir +
                              fileSep + “fop“ + fileSep + “streams.txt“ + “!“ );
}
```

Methode lines von Klasse Files wirft eine IOException, falls irgendetwas beim Öffnen der Datei schiefgeht, zum Beispiel wenn die Datei gar nicht wie in Path angegeben existiert oder das Programm kein Leserecht auf dieser Datei hat. IOException wird in verschiedenen Situationen, die mit Input/Output zu tun haben, geworfen. Diese Exception-Klasse ist nicht direkt oder indirekt von RuntimeException abgeleitet und muss daher gefangen werden. Sie ist im Package java.io zu finden, das heißt, wenn man mit den Klassen aus java.nio.file arbeiten will, wird man allein schon wegen IOException beinahe immer auch Package java.io importieren müssen.

Streams und Dateien

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDir = System.getProperty ( “user.home“ );
String fileSep = System.getProperty ( “file.separator“ );
Path path = Paths.get ( homeDir, “fop“, “streams.txt“ );
try ( Stream<String> stream = Files.lines ( path ) ) {
   String fileContentAsString = stream.reduce ( String::concat );
} catch ( IOException exc ) {
   System.out.print ( “Could not open file “ + fileSep + homeDir +
                              fileSep + “fop“ + fileSep + “streams.txt“ + “!“ );
}
```

Methode reduce von Klasse Stream erstellt aus allen Elementen des Streams ein einzelnes Ergebnis durch sukzessiven Aufruf der Funktion, die durch den aktualen Parameter gegeben ist. Damit ist reduce mehr oder weniger identisch zur Faltungsoperation mit lfold in Racket, die wir in Kapitel 03c kennen gelernt hatten.

***Erinnerung*: Weiter vorne im vorliegenden Kapitel haben wir in einem gleichnamigen Abschnitt Methodenaufrufe als Lambda-Ausdrücke gesehen. Wie schon im Abschnitt zu Strings in Kapitel 03b gesehen, erstellt concat ein neues String-Objekt aus dem Objekt, mit dem concat aufgerufen wird, und dem Parameter von concat, und zwar eine Aneinanderreihung beider Zeichenketten.**

Streams und Dateien

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String homeDir = System.getProperty ( “user.home“ );
String fileSep = System.getProperty ( “file.separator“ );
Path path = Paths.get ( homeDir, “fop“, “streams.txt“ );
try ( Stream<String> stream = Files.lines ( path ) ) {
   String fileContentAsString = stream.reduce ( String::concat );
} catch ( IOException exc ) {
   System.out.print ( “Could not open file “ + fileSep + homeDir +
                      fileSep + “fop“ + fileSep + “streams.txt“ + “!“ );
}
```

Wie schon bei der Einführung von System Properties eben gesagt, sollte man den Separator für die Namen von Verzeichnissen und Dateien im jeweiligen Dateysystem nicht fest kodieren, sondern durch Methode getProperty von System abfragen.

TECHNISCHE UNIVERSITÄT DARMSTADT

IntStreams, LongStreams und DoubleStreams

***Erinnerung*: In Kapitel 06, am Ende des Abschnitts zu generischen Klassen, hatten wir gesehen, dass es zu den diversen Interfaces in Package java.util.function jeweils eine generische und drei nichtgenerische Varianten gibt. Die drei nichtgenerischen Varianten erlauben es, die primitiven Datentypen int, long und double ohne Wrapper-Klassen ins Spiel zu bringen, was mit Generizität in Java ja leider nicht geht.**

Wir werden jetzt sehen, dass derselbe Workaround auch bei Streams vorgesehen ist, denn Streams von primitiven Datentypen sind trotz Boxing und Unboxing in der Regel natürlich bequemer als Streams der zugehörigen Wrapper-Klassen.

Int-/Long-/DoublesStreams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntStream stream1 = IntStream.of ( 1, 2, 3 );
IntStream stream2 = stream1.filter ( x -> x % 2 == 1 );
IntStream stream3 = stream2.map ( x -> x * x );
int n = stream3.max ( new MyNumberComparator() )
        .getAsInt();
```

Die Arbeit mit einer dieser drei speziellen, nichtgenerischen Stream-Klassen ist weitestgehend identisch zur Arbeit mit der generischen Stream-Klasse. Hier sehen wir uns nur IntStream an; LongStream und DoubleStream sind völlig analog.

Int-/Long-/DoublesStreams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntStream stream1 = IntStream.of ( 1, 2, 3 );
IntStream stream2 = stream1.filter ( x -> x % 2 == 1 );
IntStream stream3 = stream2.map ( x -> x * x );
int n = stream3.max ( new MyNumberComparator() )
        .getAsInt();
```

Auch IntStream hat die Klassenmethode of zum Einrichten eines Streams aus einer gegebenen Sequenz von Zahlen.

Int-/Long-/DoublesStreams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntStream stream1 = IntStream.of ( 1, 2, 3 );
IntStream stream2 = stream1.filter ( x -> x % 2 == 1 );
IntStream stream3 = stream2.map ( x -> x * x );
int n = stream3.max ( new MyNumberComparator() )
        .getAsInt();
```

Der Vorteil der drei spezialisierten Streams ist, dass Operationen auf den zugehörigen pritimiven Datentypen umstandslos einfach hingeschrieben werden können.

Int-/Long-/DoublesStreams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntStream stream1 = IntStream.of ( 1, 2, 3 );
IntStream stream2 = stream1.filter ( x -> x % 2 == 1 );
IntStream stream3 = stream2.map ( x -> x * x );
int n = stream3.max ( new MyNumberComparator() )
        .getAsInt();
```

Der Vorteil der drei spezialisierten Streams ist, dass Operationen auf den zugehörigen pritimiven Datentypen umstandslos einfach hingeschrieben werden können.

Int-/Long-/DoublesStreams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntStream stream1 = IntStream.of ( 1, 2, 3 );
IntStream stream2 = stream1.filter ( x -> x % 2 == 1 );
IntStream stream3 = stream2.map ( x -> x * x );
int n = stream3.max ( new MyNumberComparator() )
        .getAsInt();
```

Analog zur generischen Klasse Optional bei generischen Streams gibt es OptionalInt für IntStream, analog ling und double. Daher muss noch eine Methode von OptionalInt aufgerufen werden, um schlussendlich an das Ergebnis als int-wert heranzukommen.

Int-/Long-/DoublesStreams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntStream stream1 = IntStream.of ( 1, 2, 3 );
IntStream stream2 = stream1.filter ( x -> x % 2 == 1 );
IntStream stream3 = stream2.map ( x -> x * x );
int n = stream3.max ( new MyNumberComparator() ).getAsInt();
```

```
int n = IntStream.of(1,2,3)
                 .filter ( x -> x % 2 == 1 )
                 .map ( x -> x * x )
                 .max ( new MyNumberComparator() )
                 .getAsInt();
```

Und so wie wir bei der generischen Stream-Klasse die Operationen zu einer Anweisung zusammengefasst hatten, geht das natürlich auch bei den drei spezialisierten Stream-Klassen.

TECHNISCHE UNIVERSITÄT DARMSTADT

Random Zahlen und Streams

Eine häufige Anwendungsmöglichkeit der drei spezialisierten Stream-Klassen sind Streams aus Zufallszahlen.

Random Zahlen und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
static double [ ] randomDoubleArray ( int length ) {
    double [ ] returnArray = new double [ length ];
    Random random = new Random();
    for ( int i = 0; i < returnArray.length; i++ )
        returnArray[i] = random.nextDouble();
    return returnArray;
}
```

So sieht es zum Vergleich erst einmal aus, wenn wir ein *Array* aus Zufallszahlen erzeugen.

Random Zahlen und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
static double [ ] randomDoubleArray ( int length ) {
    double [ ] returnArray = new double [ length ];
    Random random = new Random();
    for ( int i = 0; i < returnArray.length; i++ )
        returnArray[i] = random.nextDouble();
    return returnArray;
}
```

Diese Methode soll einen Array aus so vielen Zufallszahlen erzeugen, wie der Parameter angibt.

Random Zahlen und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
static double [ ] randomDoubleArray ( int length ) {
   double [ ] returnArray = new double [ length ];
   Random random = new Random();
   for ( int i = 0; i < returnArray.length; i++ )
       returnArray[i] = random.nextDouble();
   return returnArray;
}
```

Zur Erzeugung von Zufallszahlen steht Klasse Random aus Package java.util bereit.

Random Zahlen und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
static double [ ] randomDoubleArray ( int length ) {
   double [ ] returnArray = new double [ length ];
   Random random = new Random();
   for ( int i = 0; i < returnArray.length; i++ )
      returnArray[i] = random.nextDouble();
   return returnArray;
}
```

Für verschiedene primitive Datentypen bietet Klasse Random jeweils eine Methode, die eine Zufallszahl aus diesem Datentyp zurückliefert. Die Methodennamen sind einheitlich: next vor dem Namen des primitiven Datentyps. Bei jedem Aufruf einer solchen Methode kommt eine neue Zufallszahl, daher das next im Methodennamen. Bei nextDouble und nextFloat ist das ein Wert aus dem halboffenen Intervall [0...1), bei nextInt und nextLong eine beliebige Zahl aus dem Wertebereich von int beziehungsweise long. Die Wahrscheinlichkeitsverteilung ist jeweils die Gleichverteilung (englisch uniform distribution).

Random Zahlen und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntStream stream1 = new Random().ints();

LongStream stream2 = new Random().longs();

DoubleStream stream2 = new Random().doubles();
```

Jetzt sehen wir uns dasselbe mit Streams an.

Random Zahlen und Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntStream stream1 = new Random().ints();

LongStream stream2 = new Random().longs();

DoubleStream stream2 = new Random().doubles();
```

Praktischerweise hat Random schon je eine Methode für jede der drei spezialisierten Stream-Klassen. Alle drei Methoden liefern jeweils einen Stream zurück.

Dies ist unser erstes Beispiel dafür, dass Streams in einem wichtigen Punkt über Listen, Arrays und Dateien hinausgehen: Streams können auch potentiell unendlich lang sein. Egal wie oft Sie jetzt eine Zufallszahl aus einem dieser drei Streams holen, der Stream kommt nie zu einem Ende – im Gegensatz zu dem Array von Zufallszahlen, das wir eingangs zum Vergleich aufgebaut hatten. Und auch ein Iterator über einem solchen Stream wird immer wieder true bei hasNext zurückliefern, egal wie viele Zufallszahlen Sie mit Methode next herausgeholt haben.

TECHNISCHE UNIVERSITÄT DARMSTADT

Unendliche Streams selbstgemacht

Natürlich können wir auch selbst Streams basteln, die nie enden. Daran können wir auch genauer studieren, wie das mit den unendlichen Streams eigentlich funktioniert und was man damit machen kann.

Unendliche Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StreamOfSineValues implements DoubleStream {
   private double start;
   private double samplingDist;
   public StreamOfSineValue ( double start, double samplingDist ) {
      this.start = start;
      this.samplingDist = samplingDist;
   }
   .....
}
```

Als Beispiel definieren wir eine Stream-Klasse, die eine Abtastung einer Sinuskurve simuliert.

Unendliche Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StreamOfSineValues implements DoubleStream {
   private double start;
   private double samplingDist;
   public StreamOfSineValue ( double start, double samplingDist ) {
      this.start = start;
      this.samplingDist = samplingDist;
   }
   ..........
}
```

Für die Abtastung werden ein Startpunkt und eine Distanz definiert, die reziprok zur Abtastrate steht.

Unendliche Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StreamOfSineValuesIterator
                              implements PrimitiveIterator.ofDouble {
   private int n = 0;
   public boolean hasNext() {
      return true;
   }
   public double next() {
      return Math.sin ( start + samplingDist * n++ );
   }
}
```

Selbstverständlich schauen wir uns nicht die gesamte Klasse an, es geht hier nur um das Prinzip. Deshalb wenden wir uns gleich der ebenfalls zu realisierenden Iterator-Klasse zu. Daran lässt sich das Prinzip eines unendlichen Streams sehr gut studieren.

Unendliche Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StreamOfSineValuesIterator
                          implements PrimitiveIterator.ofDouble {
   private int n = 0;
   public boolean hasNext() {
      return true;
   }
   public double next() {
      return Math.sin ( start + samplingDist * n++ );
   }
}
```

Die Iterator-Klasse zu DoubleStream heißt ofDouble und ist in einem Interface namens PrimitiveIterator in Package java.util eingebettet.

Vorgriff: Klassen können tatsächlich in andere Klassen eingebettet sein. Das schauen wir uns aber erst in Kapitel 9 genauer an, Abschnitt „Verschachtelte Klassen (nested classes)“.

Unendliche Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StreamOfSineValuesIterator
                              implements PrimitiveIterator.ofDouble {
   private int n = 0;
   public boolean hasNext() {
      return true;
   }
   public double next() {
      return Math.sin ( start + samplingDist * n++ );
   }
}
```

Gemäß Logik einer äquidistanten Abtastung wird zuerst am Startpunkt abgetastet, und danach wird der Abtastpunkt jeweils um die Abtastdistanz erhöht. Dafür hat der Iterator das int-Attribute n, ...

Unendliche Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StreamOfSineValuesIterator
                              implements PrimitiveIterator.ofDouble {
   private int n = 0;
   public boolean hasNext() {
      return true;
   }
   public double next() {
      return Math.sin ( start + samplingDist * n++ );
   }
}
```

... das natürlich bei jedem Aufruf von next um eins erhöht werden muss.

Unendliche Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StreamOfSineValuesIterator
                              implements PrimitiveIterator.ofDouble {
   private int n = 0;
   public boolean hasNext() {
      return true;
   }
   public double next() {
      return Math.sin ( start + samplingDist * n++ );
   }
}
```

Hier kommt jetzt wohl am deutlichsten der Unterschied zwischen endlichen und unendlichen Streams zum Ausdruck: Es gibt immer noch ein nächstes Element, egal, wie viele Elemente man schon gesehen hat.

TECHNISCHE UNIVERSITÄT DARMSTADT

Blick über den Tellerrand: Streams in Racket

Nachdem wir jetzt gesehen haben, wie kompliziert Streams in Java aussehen, schauen wir uns kurz die sehr viel einfacheren Streams in Racket an.

Streams in Racket

- Sind wie Listen schon in Racket eingebaut
  - Leider *nicht* in HtDP-TL
- Logik von Streams in Racket:
  - eigentlich dasselbe wie Listen
  - nur potentiell *lazily evaluated*

In Java sind Streams aus Klassen und Interfaces in der Standardbibliothek gebaut, in Racket sind sie Bestandteil der Sprache.

Streams in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sind wie Listen schon in Racket eingebaut
  - Leider *nicht* in HtDP-TL
- Logik von Streams in Racket:
  - eigentlich dasselbe wie Listen
  - nur potentiell *lazily evaluated*

Bei diesem Thema müssen wir uns allerdings Racket selbst zuwenden, da Streams in DrRacket nicht realisiert sind.

Streams in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sind wie Listen schon in Racket eingebaut
  - Leider *nicht* in HtDP-TL
- Logik von Streams in Racket:
  - eigentlich dasselbe wie Listen
  - nur potentiell *lazily evaluated*

Das ist eigentlich schade, denn Streams sind in Racket ein schönes, einfaches, schlankes Konzept.

Streams in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sind wie Listen schon in Racket eingebaut
  - Leider *nicht* in HtDP-TL
- Logik von Streams in Racket:
  - eigentlich dasselbe wie Listen
  - nur potentiell *lazily evaluated*

Wir werden gleich sehen, dass Streams in Racket im Grunde nur eine Variation von Listen sind und auch fast genauso aussehen.

Streams in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sind wie Listen schon in Racket eingebaut
  - Leider *nicht* in HtDP-TL
- Logik von Streams in Racket:
  - eigentlich dasselbe wie Listen
  - nur potentiell *lazily evaluated*

Der entscheidende Unterschied zu Listen besteht in Racket darin, dass die Elemente eines Streams gar nicht unbedingt alle physisch existieren müssen. Wie wir das bei Java inzwischen kennen gelernt haben, können Streams im Extremfall so organisiert sein, dass jedes Element erst bei seinem Abruf generiert wird – zum Beispiel beim Lesen aus einer Datei oder beim Abruf von Zufallszahlen.

Streams in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlegende eingebaute Funktionen auf Streams:

- (stream-cons x str)
- (stream-first str)
- (stream-rest str)
- (stream-empty? str)
- (stream-map fct str)
- (stream-filter pred str)
- (stream-fold init fct str)

Wenn wir einen Stream haben, dann kann man darauf einige Operationen anwenden, die völlig analog zu den entsprechenden Listenoperationen sind. Die hier gezeigten Funktionen sind in Racket vordefiniert.

Streams in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlegende eingebaute Funktionen auf Streams:

(stream-cons x str)	**(stream-map fct str)**
(stream-first str)	**(stream-filter pred str)**
(stream-rest str)	**(stream-fold init fct str)**
(stream-empty? str)	

Diese Funktion baut einen neuen Stream, indem sie x vorne an den Stream str anhängt – völlig analog zu cons bei Listen, wie auch die folgenden Funktionen.

Streams in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlegende eingebaute Funktionen auf Streams:

(stream-cons x str)	(stream-map fct str)
(stream-first str)	(stream-filter pred str)
(stream-rest str)	(stream-fold init fct str)
(stream-empty? str)	

Dieser Ausdruck liefert das erste Element des Streams zurück. Das ist der späteste Moment, an dem sich das erste Element materialisieren muss, denn nun wird es ja ausgelesen und weiterverwendet. In diesem Fall müsste Funktion stream-first die Materialisierung leisten.

Streams in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlegende eingebaute Funktionen auf Streams:

(stream-cons x str)	(stream-map fct str)
(stream-first str)	(stream-filter pred str)
(stream-rest str)	(stream-fold init fct str)
(stream-empty? str)	

Der Rest des Streams ohne das erste Element braucht sich hier noch nicht zu materialisieren.

Streams in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlegende eingebaute Funktionen auf Streams:

(stream-cons x str)	(stream-map fct str)
(stream-first str)	(stream-filter pred str)
(stream-rest str)	(stream-fold init fct str)
(stream-empty? str)	

Analog zu Listen die Abfrage, ob der Stream leer ist.

Streams in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlegende eingebaute Funktionen auf Streams:

(stream-cons x str)	(stream-map fct str)
(stream-first str)	(stream-filter pred str)
(stream-rest str)	(stream-fold init fct str)
(stream-empty? str)	

Diese drei Funktionen sind ebenfalls völlig analog zu den entsprechenden eingebauten Listenfunktionen in Racket, also map, filter und lfold. Damit beenden wir das Racket-Intermezzo.

TECHNISCHE UNIVERSITÄT DARMSTADT

Bytedaten direkt (ohne Streams)

Wir haben schon gesehen, wie man Streams aus Dateien erzeugt. Im Rest dieses Kapitels schauen wir uns Dateien *ohne* Bezug zu Klasse Stream an. Die Klassen und Interfaces finden sich in Package java.io. Ein paar Klassen werden das Wort Stream als Namensbestandteil haben, haben aber dennoch nichts mit Klasse Stream zu tun.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( InputStream in ) throws IOException {
   while ( true ) {
      int n = in.read();
      if ( n == −1 )
        return;
      processByte ( n );
   }
}
```

```
try ( FileInputStream in = new FileInputStream ( fileName ) ) {
    X.m ( in );
} catch ( FileNotFoundException | IOException exc ) .....
```

Wir fangen mit etwas ganz Grundlegendem an, mit dem byteweisen Lesen aus Dateien beziehungsweise byteweisen Schreiben in Dateien; zuerst Lesen. Wir werden gleich sehen, dass darauf dann die textbasierten Zugriffsmöglichkeiten auf Dateien aufbauen.

Byteweiser Zugriff ist sinnvoll, wenn eine Datei nicht einen Text, sondern binäre Daten enthält, zum Beispiel ein Bild oder eine Audio- oder Video-Datei. Allerdings werden wohl die wenigsten Softwareentwickler jemals Programme zu schreiben haben, die Binärdateien wirklich byteweise verarbeiten, da sämtliche gängigen Operationen auf Binärdaten durch entsprechende Methoden in der Standardbibliothek oder in anderen Bibliotheken verfügbar sind. Daher ist es nicht unwahrscheinlich, dass Sie niemals direkt mit Klasse InputStream arbeiten werden.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( InputStream in ) throws IOException {
    while ( true ) {
        int n = in.read();
        if ( n == −1 )
          return;
        processByte ( n );
    }
}
```

```
try ( FileInputStream in = new FileInputStream ( fileName ) ) {
    X.m ( in );
} catch ( FileNotFoundException | IOException exc ) .....
```

Die Methode m bekommt also einen InputStream und soll ihn irgendwie byteweise verarbeiten. Worin genau diese Verarbeitung besteht, ist uns egal, dafür steht stellvertretend die Methode processByte.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( InputStream in ) throws IOException {
   while ( true ) {
      int n = in.read();
      if ( n == −1 )
        return;
      processByte ( n );
   }
}
```

```
try ( FileInputStream in = new FileInputStream ( fileName ) ) {
    X.m ( in );
} catch ( FileNotFoundException | IOException exc ) .....
```

Hier wird das jeweils nächste Byte aus der Datei gelesen, allerdings in den primitiven Datentyp int, nicht in byte. Das hat zwei Gründe: Erstens ist der primitive Datentyp byte vorzeichenbehaftet und hat Werte zwischen -128 und +127, während ein Byte einfach nur acht binäre Bits sind, ...

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( InputStream in ) throws IOException {
   while ( true ) {
      int n = in.read();
      if ( n == −1 )
        return;
      processByte ( n );
   }
}
```

```
try ( FileInputStream in = new FileInputStream ( fileName ) ) {
    X.m ( in );
} catch ( FileNotFoundException | IOException exc ) .....
```

... zweitens muss es noch eine Möglichkeit geben anzuzeigen, dass das Dateiende erreicht ist. Noch von der Sprache C her ist es Konvention, dies dadurch anzuzeigen, dass ein unmöglicher Wert zurückgegeben wird, nämlich -1.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( InputStream in ) throws IOException {
   while ( true ) {
      int n = in.read();
      if ( n == −1 )
        return;
      processByte ( n );
    }
}
```

```
try ( FileInputStream in = new FileInputStream ( fileName ) ) {
    X.m ( in );
} catch ( FileNotFoundException | IOException exc ) .....
```

Die von Methode read potentiell geworfene IOException wird von Methode m weitergereicht.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( InputStream in ) throws IOException {
   while ( true ) {
      int n = in.read();
      if ( n == −1 )
        return;
      processByte ( n );
   }
}
```

```
try ( FileInputStream in = new FileInputStream ( fileName ) ) {
    X.m ( in );
} catch ( FileNotFoundException | IOException exc ) .....
```

Wenn Methode read nicht -1 zurückliefert, dann ist gewährleistet, dass nur das unterste Byte von Datentyp int ungleich 0 ist. Diese imaginäre Methode verarbeitet nur dieses.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( InputStream in ) throws IOException {
   while ( true ) {
      int n = in.read();
      if ( n == −1 )
        return;
      processByte ( n );
   }
}
```

```
try ( FileInputStream in = new FileInputStream ( fileName ) ) {
   X.m ( in );
} catch ( FileNotFoundException | IOException exc ) .....
```

Wir gehen wieder einmal der Einfachheit halber davon aus, dass X der Name der Klasse ist, zu der Methode m gehört.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( InputStream in ) throws IOException {
   while ( true ) {
      int n = in.read();
      if ( n == −1 )
        return;
      processByte ( n );
   }
}
```

```
try ( FileInputStream in = new FileInputStream ( fileName ) ) {
    X.m ( in );
} catch ( FileNotFoundException | IOException exc ) .....
```

Klasse InputStream ist abstrakt, es sind also nur Objekte von direkt oder indirekt abgeleiteten Klassen möglich. Klasse FileInputStream ist direkt von InputStream abgeleitet und leistet Lesezugriff auf Dateien. Sie hat einen Konstruktor, in dem man den Dateinamen als String angeben kann.

***Erinnerung*: Im Kapitel 05, Abschnitt zu Exceptions, haben wir try-with-resources gesehen. Dadurch wird gewährleistet, dass die Datei nach dem try-catch-Block in jedem Fall wieder geschlossen wird, egal wie der try-catch-Block verlassen wird.**

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( InputStream in ) throws IOException {
   while ( true ) {
      int n = in.read();
      if ( n == −1 )
        return;
      processByte ( n );
   }
}
```

```
try ( FileInputStream in = new FileInputStream ( fileName ) ) {
    X.m ( in );
} catch ( FileNotFoundException | IOException exc ) .....
```

Der Konstruktor von FileInputStream wirft eine Exception mit sicherlich sinnvoll gewähltem Namen, und zwar in dem Fall, dass die Datei gar nicht existiert oder fileName nicht der Name einer Datei, sondern eines Verzeichnisses ist oder wenn die Datei aus irgendwelchen Gründen nicht zum Lesen geöffnet werden kann.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( OutputStream out ) throws IOException {
   for ( int i = 0; i <= 2 * Byte.MAX_VALUE + 1; i++ )
      out.write ( i );
}


try ( FileOutputStream out = new FileOutputStream ( fileName )  ) {
  X.m ( out );
} catch ( FileNotFoundException | IOException exc ) .....
```

Jetzt haben wir gesehen, wie man den Inhalt einer Datei byteweise *einlesen* kann. Als nächstes schauen wir uns byteweises *Hinausschreiben* in eine Datei an. Das Ziel eines OutputStream nennt man auch *Datensenke*.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( OutputStream out ) throws IOException {
   for ( int i = 0; i <= 2 * Byte.MAX_VALUE + 1; i++ )
      out.write ( i );
}


try ( FileOutputStream out = new FileOutputStream ( fileName )  ) {
  X.m ( out );
} catch ( FileNotFoundException | IOException exc ) .....
```

Das unterste Byte der int-Variable i durchläuft hier rein zur Illustration alle 256 Bitmuster, die ein Byte bestehend aus acht Bits annehmen kann.

***Erinnerung*: Kapitel 01b, Abschnitt zu primitiven Datentypen.**

***Vorgriff*: Kapitel 11 zur internen Zahldarstellung.**

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( OutputStream out ) throws IOException {
   for ( int i = 0; i <= 2 * Byte.MAX_VALUE + 1; i++ )
      out.write ( i );
}


try ( FileOutputStream out = new FileOutputStream ( fileName )  ) {
  X.m ( out );
} catch ( FileNotFoundException | IOException exc ) .....
```

Methode write von OutputStream hat int als formalen Parametertyp. Aber es wird nur das unterste Byte geschrieben, alle anderen Bits des int-Wertes werden ignoriert.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( OutputStream out ) throws IOException {
   for ( int i = 0; i <= 2 * Byte.MAX_VALUE + 1; i++ )
      out.write ( i );
}
```

```
try ( FileOutputStream out = new FileOutputStream ( fileName )  ) {
  X.m ( out );
} catch ( FileNotFoundException | IOException exc ) .....
```

Analog zum Einlesen aus einer Datei sieht jetzt das Schreiben in eine Datei so aus. Klasse FileOutputStream ist von Klasse OutputStream abgeleitet und hat einen Konstruktor, der den Dateinamen als String erhält. Existiert die Datei schon, geht ihr bisheriger Inhalt verloren; existiert sie noch nicht, wird sie neu angelegt.

Der Konstruktor wirft eine FileNotFoundException, falls die Datei dieses Namens ein Verzeichnis ist oder nicht zum Schreiben geöffnet werden kann oder auch, wenn sie nicht existiert und auch nicht angelegt werden kann.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( OutputStream out ) throws IOException {
    for ( int i = 0; i <= 2 * Byte.MAX_VALUE + 1; i++ )
        out.write ( i );
}
```

```
try ( FileOutputStream out = new FileOutputStream ( fileName, true )  ) {
   X.m ( out );
} catch ( FileNotFoundException | IOException exc ) .....
```

Klasse FileOutputStream hat noch weitere Konstruktoren, darunter einen, der neben dem Namen der Datei noch ein boolean bekommt. Ist dieser zweite Parameter false, dann passiert exakt dasselbe wie beim ersten Konstruktor mit nur einem Parameter; ist er hingegen true wie hier gezeigt, dann geht der bisherige Inhalt der Datei *nicht* verloren, sondern die neuen Inhalte werden hinten an die bisherigen Inhalte angehängt.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileInputStream in = new FileInputStream ( fileName ) ) {
    X.m ( new BufferedInputStream ( in ) );
} catch ( FileNotFoundException | IOException exc ) .....
```

```
try ( FileOutputStream out = new FileOutputStream ( fileName ) ) {
  X.m ( new BufferedOutputStream ( out ) );
} catch ( FileNotFoundException | IOException exc ) .....
```

Ein wichtiger Punkt ist die Geschwindigkeit beim Einlesen von Daten aus einer Datei beziehungsweise beim Schreiben von Daten in eine Datei. Die Speicherhardware ist typischerweise dahingehend optimiert, dass eine gewisse Anzahl von Bytes mehr oder weniger genauso schnell eingelesen beziehungsweise hinausgeschrieben werden kann wie ein einzelnes Byte.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileInputStream in = new FileInputStream ( fileName ) ) {
    X.m ( new BufferedInputStream ( in ) );
} catch ( FileNotFoundException | IOException exc ) .....
```

```
try ( FileOutputStream out = new FileOutputStream ( fileName ) ) {
   X.m ( new BufferedOutputStream ( out ) );
} catch ( FileNotFoundException | IOException exc ) .....
```

Es gibt die Möglichkeit, gleich mehrere Bytes auf einmal einzulesen. Klasse BufferedInputReader kapselt dieses technische Detail so ein, dass wir einfach die Methode read verwenden können, die wir schon kennen und die bei jedem Aufruf nur ein einzelnes Byte zurückliefert. Aber im Hintergrund passiert Folgendes: Klasse BufferedInputStream enthält intern ein Array von Bytes, das ist der Puffer. Bei jedem Aufruf von read zum Einlesen eines einzelnen Zeichens wird jeweils ein nächstes Byte aus diesem Array zurückgeliefert. Sind alle Bytes aus dem Puffer durch entsprechend viele Aufrufe von read zurückgeliefert, wird beim nächsten read erst der Puffer „in einem Rutsch“ mit neuen Bytes aus der Datei gefüllt und das erste dieser Bytes zurückgeliefert.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileInputStream in = new FileInputStream ( fileName ) ) {
    X.m ( new BufferedInputStream ( in ) );
} catch ( FileNotFoundException | IOException exc ) .....
```

```
try ( FileOutputStream out = new FileOutputStream ( fileName ) ) {
  X.m ( new BufferedOutputStream ( out ) );
} catch ( FileNotFoundException | IOException exc ) .....
```

In gewisser Weise umgekehrt dann beim BufferedOutputStream: Wird mit write ein einzelnes Zeichen geschrieben, dann wird es nicht sofort in die Datei geschrieben, sondern erst einmal in einen internen Puffer. Wenn der Puffer voll ist, wird der gesamte Inhalt des Puffers „in einem Rutsch“ in die Datei geschrieben.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileOutputStream out1 = new FileOutputStream ( fileName );
      BufferedOutputStream out2 = new BufferedOutputStream ( out1 );
      PrintStream out3 = new PrintStream ( out2 ) ) {
   out3.print ( “pi = “ );
   out3.print ( 3.14 );
   out3.println ( ´!´ );
} catch ( FileNotFoundException | IOException exc ) .....
```

Zu OutputStream gibt es einen komfortableren Aufsatz in Package java.io: Klasse PrintStream.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileOutputStream out1 = new FileOutputStream ( fileName );
      BufferedOutputStream out2 = new BufferedOutputStream ( out1 );
      PrintStream out3 = new PrintStream ( out2 ) ) {
   out3.print ( “pi = “ );
   out3.print ( 3.14 );
   out3.println ( ´!´ );
} catch ( FileNotFoundException | IOException exc ) .....
```

***Erinnerung*: Als wir try-with-resources im Kapitel 05 eingeführt haben, hatten wir kurz erwähnt, dass darin auch mehrere Ressourcen geöffnet werden können, durch Semikolons getrennt, die letzte Ressource aber *nicht* wie eine Anweisung mit einem Semikolon angeschlossen.**

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileOutputStream out1 = new FileOutputStream ( fileName );
      BufferedOutputStream out2 = new BufferedOutputStream ( out1 );
      PrintStream out3 = new PrintStream ( out2 ) ) {
   out3.print ( “pi = “ );
   out3.print ( 3.14 );
   out3.println ( ´!´ );
} catch ( FileNotFoundException | IOException exc ) .....
```

Als erstes brauchen wir für unser Beispiel wieder einen OutputStream, zum Beispiel wieder FileOutputStream.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileOutputStream out1 = new FileOutputStream ( fileName );
      BufferedOutputStream out2 = new BufferedOutputStream ( out1 );
      PrintStream out3 = new PrintStream ( out2 ) ) {
   out3.print ( “pi = “ );
   out3.print ( 3.14 );
   out3.println ( ´!´ );
} catch ( FileNotFoundException | IOException exc ) .....
```

Es ist nicht notwendig, aber wie eben schon gesagt, aus Laufzeitgründen sehr sinnvoll, den OutputStream mit BufferedOutputStream zu puffern.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileOutputStream out1 = new FileOutputStream ( fileName );
      BufferedOutputStream out2 = new BufferedOutputStream ( out1 );
      PrintStream out3 = new PrintStream ( out2 ) ) {
   out3.print ( “pi = “ );
   out3.print ( 3.14 );
   out3.println ( ´!´ );
} catch ( FileNotFoundException | IOException exc ) .....
```

Nun der PrintStream selbst. Wie gesagt, ist ein PrintStream sozusagen ein Aufsatz auf einem OutputStream, der durch den Konstruktor übergeben wird.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileOutputStream out1 = new FileOutputStream ( fileName );
      BufferedOutputStream out2 = new BufferedOutputStream ( out1 );
      PrintStream out3 = new PrintStream ( out2 ) ) {
  out3.print ( “pi = “ );
  out3.print ( 3.14 );
  out3.println ( ´!´ );
} catch ( FileNotFoundException | IOException exc ) .....
```

Klasse PrintStream hat mehrere Methoden print für die byteweise Ausgabe von Werten verschiedener Datentypen auf den im Konstruktor übergebenen OutputStream. Klasse PrintStream dient sozusagen als Konvertierer von den verschiedenen primitiven Datentypen sowie String in eine byteweise Repräsentation, das eigentliche Schreiben übernimmt dann der OutputStream.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileOutputStream out1 = new FileOutputStream ( fileName );
      BufferedOutputStream out2 = new BufferedOutputStream ( out1 );
      PrintStream out3 = new PrintStream ( out2 ) ) {
   out3.print ( “pi = “ );
   out3.print ( 3.14 );
   out3.println ( ´!´ );
} catch ( FileNotFoundException | IOException exc ) .....
```

Beim Öffnen der Datei kann eine FileNotFoundException auftreten, beim Schreiben mit print dann eine IOException.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

System.out

System.err

System.in

Die überladene Methode print von Klasse PrintStream hat Sie sicher an etwas erinnert.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

System.out

System.err

System.in

Das Klassenattribut out von System ist von Klasse PrintStream. Neben den print-Methoden hat PrintStream also auch diverse Methoden namens println zum Schreiben nebst Zeilenumbruch.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

System.out

System.err

System.in

Klasse System hat völlig parallel noch ein weiteres Klassenattribut namens err von Klasse printStream. Die Ausgaben erscheinen zunächst einmal genau dort, wo auch die Ausgaben von System.out erscheinen. Wir werden gleich sehen, warum es dennoch sinnvoll ist, *zwei* PrintStream hier zu haben.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

System.out

System.err

System.in

Aber zuerst halten wir noch fest, dass Klasse System außerdem ein Klassenattribut in von Klasse InputStream hat. Dieser InputStream liest zunächst einmal Daten von der Tastatur, solange wir nichts daran ändern.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileInputStream in = new FileInputStream ( filename1 );
      FileOutputStream out = new FileOutputStream ( filename2 );
      FileOutputStream err = new FileOutputStream ( filename3 ) ) {
  System.setIn ( in );
  System.setOut ( new PrintStream ( out ) );
  System.setErr ( new PrintStream ( err ) );
  makeSomethingWithInOutAndErr();
}
```

Bei Einrichtung von Klasse System werden diese drei Klassenattribute auf Bildschirmausgabe beziehungsweise Tastatureingabe gesetzt.

***Erinnerung*: Im Kapitel 03c hatten wir Static Initializer im Abschnitt „Konstruktoren und Static Initializer“ kennen gelernt und gesehen, wie damit Klassenattribute zu Beginn des Programmlaufs automatisch initialisiert werden.**

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileInputStream in = new FileInputStream ( filename1 );
      FileOutputStream out = new FileOutputStream ( filename2 );
      FileOutputStream err = new FileOutputStream ( filename3 ) ) {
  System.setIn ( in );
  System.setOut ( new PrintStream ( out ) );
  System.setErr ( new PrintStream ( err ) );
  makeSomethingWithInOutAndErr();
}
```

Klasse System bietet diese drei Klassenmethoden, jeweils eine für jedes der drei Klassenattribute. Damit können die drei Klassenattribute auf andere Streams umgesetzt werden. Im Beispiel wird System.in auf eine Datei umgesetzt, liest also aus dieser Datei statt von der Tastatur. Analog werden System.out und System.err auf Dateien umgesetzt, in die sie dann schreiben statt auf den Bildschirm.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
try ( FileInputStream in = new FileInputStream ( filename1 );
      FileOutputStream out = new FileOutputStream ( filename2 );
      FileOutputStream err = new FileOutputStream ( filename3 ) ) {
  System.setIn ( in );
  System.setOut ( new PrintStream ( out ) );
  System.setErr ( new PrintStream ( err ) );
  makeSomethingWithInOutAndErr();
}
```

Jetzt lässt sich auch erklären, warum es sinnvoll ist, *zwei* PrintStream als Klassenattribute von System zu haben, out und err: Per Konvention schreibt man auf out die normalen Ausgaben eines Programms, auf err schreibt man hingegen alle Fehlerausgaben, also zum Beispiel die Fehlerausgaben, die wir öfters in catch-Blöcken definiert hatten. Die bisherigen Beispiele waren also nicht hundertprozentig sinnvoll, weil wir diese Ausgaben zur Vereinfachung und gegen die Konvention auf out und nicht auf err geschrieben hatten.

Auf dieser Folie sehen wir, dass die Ausgaben auf out und err getrennt werden können, zum Beispiel wie hier in verschiedene Dateien umgeleitet. In der Praxis ist es in der Regel häufig so, dass out *nicht* umgeleitet wird, also weiterhin auf den Bildschirm schreibt, und err wird in eine Log-Datei umgeleitet.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

java.util.zip.ZipInputStream

java.util.jar.JarInputStream

javax.sound.sampled.AudioInputStream

java.io.PipedInputStream

java.io.PipedOutputStream

Am Ende des Abschnitts zum Lesen und Schreiben von Bytedaten noch ein paar weitere ausgewählte Klassen, die von InputStream beziehungsweise OutputStream direkt oder indirekt abgeleitet sind und die Vielseitigkeit des Konzepts demonstrieren.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

java.util.zip.ZipInputStream

java.util.jar.JarInputStream

javax.sound.sampled.AudioInputStream

java.io.PipedInputStream

java.io.PipedOutputStream

So gibt es eine InputStream-Klasse zum Einlesen von komprimierten Dateien im zip-Format und eine InputStream-Klasse für jar-Dateien. Eine jar-Datei enthält kompilierte Java-Dateien, mit zip komprimiert. Beide Klassen finden sich nicht in Package java.io, sondern in Packages, in denen nützliche Funktionalität für zip-Dateien beziehungsweise jar-Dateien jeweils zusammengefasst ist.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

java.util.zip.ZipInputStream

java.util.jar.JarInputStream

javax.sound.sampled.AudioInputStream

java.io.PipedInputStream

java.io.PipedOutputStream

Diese InputStream-Klasse bietet spezielle Zugriffsmethoden für Audio-Dateien an. Auch diese Klasse findet sich nicht in java.io, sondern in einem thematisch passenden Package.

Bytedaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

java.util.zip.ZipInputStream

java.util.jar.JarInputStream

javax.sound.sampled.AudioInputStream

java.io.PipedInputStream

java.io.PipedOutputStream

Vorgriff: Im Kapitel 09 werden wir diese beiden Klassen in Aktion sehen.

Grob gesprochen, kann man ein PipedInputStream-Objekt und ein PipedOutputStream-Objekt so miteinander koppeln, dass alles, was in das PipedOutputStream-Objekt geschrieben wurde, aus dem PipedInputStream-Objekt wieder herausgelesen werden kann. Laut Oracle-Dokumentation ist es nicht zu empfehlen, eine solche Kopplung einfach so herzustellen, besser nur im Zusammenhang mit Threads, so wie wir es in Kapitel 09 sehen werden.

TECHNISCHE UNIVERSITÄT DARMSTADT

Textdaten direkt (ohne Streams)

Wir haben jetzt Klassen zum byteweisen Einlesen von Daten in ein Programm beziehungsweise Hinausschreiben von Daten aus dem Programm kennen gelernt. Für Bytedaten wie etwa Bilddateien oder Audio- oder Video-Dateien sind diese Möglichkeiten recht bequem. Aber für Textdaten sind bequemere Zugriffsmöglichkeiten wünschenswert. Dafür gibt es in Package java.io die Klassen Reader und Writer und die von ihnen abgeleiteten Klassen.

Eine Textdatei besteht aus einzelnen Zeichen, also Werten vom Typ char. Jedes char ist zwei Byte groß, also sechzehn Bit.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int m ( Reader reader, char [ ] buffer ) throws IOException {
   return reader.read ( buffer );
}
```

```
      try ( FileReader reader1 = new FileReader ( fileName );
            BufferedReader reader2 = new BufferedReader ( reader1 ) ) {
         char [ ] a = new char [ 256 ];
         int numberOfReadChars = X.m ( reader2, a );
      }
```

Zuerst zum Einlesen von Textdaten, wie üblich ein kleines, illustratives Beispiel. Sie sehen per Vergleich schon auf den ersten Blick, dass alles eigentlich ziemlich genau so wie bei byteweisem Einlesen aussieht.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int m ( Reader reader, char [ ] buffer ) throws IOException {
   return reader.read ( buffer );
}
```

```
      try ( FileReader reader1 = new FileReader ( fileName );
            BufferedReader reader2 = new BufferedReader ( reader1 ) ) {
         char [ ] a = new char [ 256 ];
         int numberOfReadChars = X.m ( reader2, a );
      }
```

Analog zur Basisklasse InputStream für das Einlesen von Bytedaten gibt es die Basisklasse Reader für das Einlesen von Textdaten.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int m ( Reader reader, char [ ] buffer ) throws IOException {
   return reader.read ( buffer );
}
```

```
      try ( FileReader reader1 = new FileReader ( fileName );
            BufferedReader reader2 = new BufferedReader ( reader1 ) ) {
         char [ ] a = new char [ 256 ];
         int numberOfReadChars = X.m ( reader2, a );
      }
```

Klasse Reader bietet mehrere Methoden namens read, darunter diese, die ein char-Array als Parameter hat. Diese Methode read liest so viele Zeichen ein, bis entweder das Array voll oder das Ende der Datenquelle erreicht ist. Rückgabe ist die Anzahl der tatsächlich eingelesenen Zeichen.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int m ( Reader reader, char [ ] buffer ) throws IOException {
   return reader.read ( buffer );
}
```

```
      try ( FileReader reader1 = new FileReader ( fileName );
            BufferedReader reader2 = new BufferedReader ( reader1 ) ) {
         char [ ] a = new char [ 256 ];
         int numberOfReadChars = X.m ( reader2, a );
      }
```

Methode read wirft potentiell eine IOException, die in diesem Beispiel von Methode m nicht gefangen, sondern weitergereicht wird.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int m ( Reader reader, char [ ] buffer ) throws IOException {
   return reader.read ( buffer );
}
```

```
try ( FileReader reader1 = new FileReader ( fileName );
      BufferedReader reader2 = new BufferedReader ( reader1 ) ) {
   char [ ] a = new char [ 256 ];
   int numberOfReadChars = X.m ( reader2, a );
}
```

Wir nehmen wieder der Einfachheit halber an, dass Methode m zu einer Klasse x gehört.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int m ( Reader reader, char [ ] buffer ) throws IOException {
   return reader.read ( buffer );
}
```

```
try ( FileReader reader1 = new FileReader ( fileName );
      BufferedReader reader2 = new BufferedReader ( reader1 ) ) {
   char [ ] a = new char [ 256 ];
   int numberOfReadChars = X.m ( reader2, a );
}
```

Wie gesagt, Rückgabe ist die Anzahl der tatsächlich eingelesenen Zeichen.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int m ( Reader reader, char [ ] buffer ) throws IOException {
   return reader.read ( buffer );
}
```

```
try ( FileReader reader1 = new FileReader ( fileName );
      BufferedReader reader2 = new BufferedReader ( reader1 ) ) {
   char [ ] a = new char [ 256 ];
   int numberOfReadChars = X.m ( reader2, a );
}
```

Völlig analog zu InputStream und FileInputStream gibt es auch eine von Reader abgeleitete Klasse FileReader, die die Datei, deren Name im Parameter übergeben wird, zum Lesen öffnet.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int m ( Reader reader, char [ ] buffer ) throws IOException {
   return reader.read ( buffer );
}
```

```
try ( FileReader reader1 = new FileReader ( fileName );
      BufferedReader reader2 = new BufferedReader ( reader1 ) ) {
   char [ ] a = new char [ 256 ];
   int numberOfReadChars = X.m ( reader2, a );
}
```

Und ebenso analog kann man noch eine Pufferung darauf setzen.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static String m ( BufferedReader reader ) throws IOException {
   return reader.readLine ();
}
```

```
try ( FileReader reader1 = new FileReader ( fileName );
      BufferedReader reader2 = new BufferedReader ( reader1 ) ) {
   X.m ( reader2 );
}
```

Eine kleine Vereinfachung: Klasse BufferedReader hat eine Methode namens readLine, die ein Stringobjekt einrichtet und eine ganze Zeile einliest. Genauer gesagt, wird alles eingelesen nach dem letzten eingelesenen Zeichen bis zum nächsten Zeilenende.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
InputStream in = .....;

Reader reader = new InputStreamReader ( in );
```

Byteweises Einlesen von Daten lässt sich auch in zeichenweises Einlesen umwandeln, was natürlich in der Regel nur dann Sinn macht, wenn ein InputStream auf einer Datenquelle geöffnet ist, die eigentlich Textdaten enthält, nicht Bytedaten. Die Brücke zwischen byteweise und zeichenweise ist Klasse InputStreamReader, die von Reader abgeleitet ist und den InputStream als Parameter im Konstruktor bekommt.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( Writer writer ) throws IOException {
   writer.write ( ´H´ );
   writer.write ( “ello World“ );
}
```

```
      try ( FileWriter writer1 = new FileWriter ( fileName );
            BufferedWriter writer2 = new BufferedWriter ( writer1 ) ) {
            X.m ( writer2 );
      }
```

Nun zum Gegenstück für das Hinausschreiben, Klasse Writer und davon abgeleitete Klassen. Wie zu erwarten, sieht auch hier beim Schreiben alles auf den ersten Blick wieder so aus wie beim Lesen.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( Writer writer ) throws IOException {
   writer.write ( ´H´ );
   writer.write ( “ello World“ );
}
```

```
try ( FileWriter writer1 = new FileWriter ( fileName );
      BufferedWriter writer2 = new BufferedWriter ( writer1 ) ) {
      X.m ( writer2 );
}
```

Die Analogie endet bei den Signaturen der Methoden. Klasse Writer hat mehrere Methoden namens write, darunter eine, die ein einzelnes Zeichen schreibt, sowie eine, die einen ganzen String schreibt.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
OutputStream out = .....;

Writer writer = new OutputStreamWriter ( out );
```

Und völlig analog zur Brücke InputStreamReader zwischen InputStream und Reader gibt es die Brücke OutputStreamWriter zwischen OutputStream und Writer.

TECHNISCHE UNIVERSITÄT DARMSTADT

Klasse Files

Ganz zu Beginn dieses Kapitels, am Anfang des Abschnitts zu Streams und Dateien, sind wir Klasse Files aus Package java.nio.file schon einmal kurz begegnet zusammen mit einer Klassenmethode mit Namen lines.

Klasse Files mit s hinten ist im Prinzip eine Sammlung von Klassenmethoden, das Grundprinzip haben wir ja schon mehrfach in anderen Kontexten gesehen, zum Beispiel die Klassen Paths und Arrays. Zum Ende dieses Kapitels schauen wir uns ein paar ausgewählte Methoden von Files an.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path = Paths.get ( ..... );
if ( Files.exists ( path ) )
if ( Files.isReadable ( path ) )
if ( Files.isWritable ( path ) )
if ( Files.isRegularFile ( path ) )
if ( Files.isDirectory ( path ) )
long size = Files.size ( path );
```

Zuerst ein paar ausgewählte von denjenigen Methoden, mit denen Attribute von Dateien abgefragt werden können.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path = Paths.get ( ..... );
if ( Files.exists ( path ) )
if ( Files.isReadable ( path ) )
if ( Files.isWritable ( path ) )
if ( Files.isRegularFile ( path ) )
if ( Files.isDirectory ( path ) )
long size = Files.size ( path );
```

Weiter vorne in diesem Kapitel, zu Beginn des Abschnitts zu Streams und Dateien, hatten wir schon gesehen, dass ein Objekt von Klasse Path einen Pfadnamen im Dateisystem repräsentiert.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path = Paths.get ( ..... );
if ( Files.exists ( path ) )
if ( Files.isReadable ( path ) )
if ( Files.isWritable ( path ) )
if ( Files.isRegularFile ( path ) )
if ( Files.isDirectory ( path ) )
long size = Files.size ( path );
```

Die Methode get ist im Klasse Paths überladen, beide Varianten machen im Grunde dasselbe, nur mit verschiedenen Parametern. Hier spielen die konkreten Parameter keine Rolle, daher nur durch Punkte angedeutet.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path = Paths.get ( ..... );
if ( Files.exists ( path ) )
if ( Files.isReadable ( path ) )
if ( Files.isWritable ( path ) )
if ( Files.isRegularFile ( path ) )
if ( Files.isDirectory ( path ) )
long size = Files.size ( path );
```

Die boolesche Methode exists liefert genau dann true zurück, wenn es im Dateisystem irgendetwas gibt, das durch den Pfadnamen in path identifiziert wird. Dieses „irgendetwas“ kann eine Datei sein oder ein Verzeichnis oder auch eine andere Art von Objekt, die im konkreten Dateisystem verwaltet wird.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path = Paths.get ( ..... );
if ( Files.exists ( path ) )
if ( Files.isReadable ( path ) )
if ( Files.isWritable ( path ) )
if ( Files.isRegularFile ( path ) )
if ( Files.isDirectory ( path ) )
long size = Files.size ( path );
```

Diese beiden booleschen Methoden fragen die Zugriffsrechte ab. In typischen Dateisystemen haben Dateien, Verzeichnisse und so weiter veränderliche Attribute, die festlegen, wer lesend beziehungsweise schreibend zugreifen darf. Die Rückgabe von isReadable beziehungsweise isWritable richtet sich nach dem Nutzer, dem der Prozess gehört, der hier gerade auf Basis des Java-Programms ausgeführt wird.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path = Paths.get ( ..... );
if ( Files.exists ( path ) )
if ( Files.isReadable ( path ) )
if ( Files.isWritable ( path ) )
if ( Files.isRegularFile ( path ) )
if ( Files.isDirectory ( path ) )
long size = Files.size ( path );
```

Wir hatten eben gesagt, dass es Dateien – auch *reguläre* Dateien genannt –, Verzeichnisse und potentiell noch weitere Arten von Objekten in einem Dateisystem gibt. Selbstverständlich gibt es Methoden zum Abfragen, von welchem Typ das Objekt ist, dessen Pfadname durch path spezifiziert ist.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path = Paths.get ( ..... );
if ( Files.exists ( path ) )
if ( Files.isReadable ( path ) )
if ( Files.isWritable ( path ) )
if ( Files.isRegularFile ( path ) )
if ( Files.isDirectory ( path ) )
long size = Files.size ( path );
```

Auch die Größe in Bytes lässt sich abfragen. Heutzutage sind viele Dateien so lang, dass die Anzahl Bytes nicht in den Datentyp int passt, daher long.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path1 = Paths.get ( ..... );
Path path2 = Paths.get ( ..... );
Path path3 = Paths.get ( ..... );
Path path4 = Paths.get ( ..... );

Files.createFile ( path1 );
Files.copy ( path1, path2 );
Files.move ( path3, path4 );  // move or rename
Files.delete ( path1 );
Files.deleteIfExists ( path2 );
```

Zum Abschluss noch ein paar ausgewählte Methoden zur Manipulation von Objekten des Dateisystems.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path1 = Paths.get ( ..... );
Path path2 = Paths.get ( ..... );
Path path3 = Paths.get ( ..... );
Path path4 = Paths.get ( ..... );
Files.createFile ( path1 );
Files.copy ( path1, path2 );
Files.move ( path3, path4 );  // move or rename
Files.delete ( path1 );
Files.deleteIfExists ( path2 );
```

Als Spielmaterial richten wir vier Pfadnamen ein.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path1 = Paths.get ( ..... );
Path path2 = Paths.get ( ..... );
Path path3 = Paths.get ( ..... );
Path path4 = Paths.get ( ..... );
Files.createFile ( path1 );
Files.copy ( path1, path2 );
Files.move ( path3, path4 );  // move or rename
Files.delete ( path1 );
Files.deleteIfExists ( path2 );
```

Das ist jetzt ein ganz wichtiger Punkt, den wir bisher nur en passant angesprochen hatten: Ein Pfadname in einem Path-Objekt bedeutet *nicht*, dass im Dateisystem tatsächlich ein Objekt mit diesem Pfad existiert. Ein Pfadname ist also nur ein Pfadname und sonst gar nichts. Bisher haben wir Pfadnamen zu existierenden Objekten kreiert, mit Methode createFile richten wir zu einem Pfadnamen ein Objekt erst *ein*, das dann diesen Pfadnamen hat.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path1 = Paths.get ( ..... );
Path path2 = Paths.get ( ..... );
Path path3 = Paths.get ( ..... );
Path path4 = Paths.get ( ..... );

Files.createFile ( path1 );
Files.copy ( path1, path2 );
Files.move ( path3, path4 );  // move or rename
Files.delete ( path1 );
Files.deleteIfExists ( path2 );
```

Kopieren des Inhalts eines Objektes ist eine der Grundoperationen in jedem Dateisystem, ...

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path1 = Paths.get ( ..... );
Path path2 = Paths.get ( ..... );
Path path3 = Paths.get ( ..... );
Path path4 = Paths.get ( ..... );

Files.createFile ( path1 );
Files.copy ( path1, path2 );
Files.move ( path3, path4 );  // move or rename
Files.delete ( path1 );
Files.deleteIfExists ( path2 );
```

... genauso das Umbenennen eines Objektes, meist spricht man vom Bewegen des Objekts im Dateisystem.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path1 = Paths.get ( ..... );
Path path2 = Paths.get ( ..... );
Path path3 = Paths.get ( ..... );
Path path4 = Paths.get ( ..... );

Files.createFile ( path1 );
Files.copy ( path1, path2 );
Files.move ( path3, path4 );  // move or rename
Files.delete ( path1 );
Files.deleteIfExists ( path2 );
```

Und schließlich noch das Entfernen eines Objektes. Sollte das Objekt gar nicht existieren, wird eine NoSuchFileException geworfen.

Textdaten direkt

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Path path1 = Paths.get ( ..... );
Path path2 = Paths.get ( ..... );
Path path3 = Paths.get ( ..... );
Path path4 = Paths.get ( ..... );

Files.createFile ( path1 );
Files.copy ( path1, path2 );
Files.move ( path3, path4 );  // move or rename
Files.delete ( path1 );
Files.deleteIfExists ( path2 );
```

Für den Fall, dass man nicht ausschließen kann, dass das zu entfernende Objekt gar nicht existiert, ist die Methode deleteIfExists oft bequemer: Falls das Objekt nicht existiert, passiert einfach gar nichts.

Damit ist dieses Kapitel beendet.