

Exceptions



Oliver Wiese
SoSe 2018

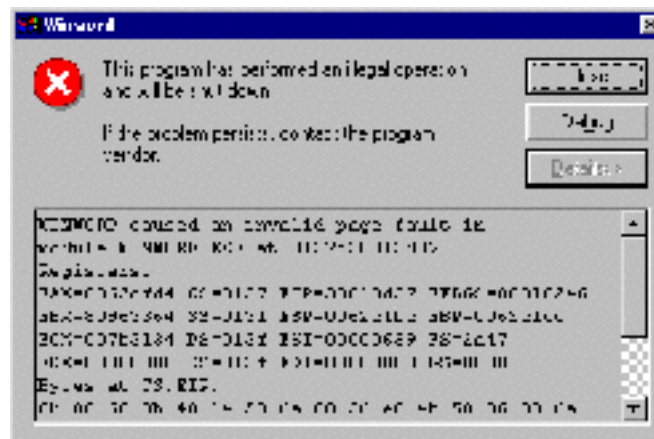
Ausnahmen



Eine Ausnahme (Exception) ist ein Fehler oder ein nicht geplantes Ereignis, das während der Ausführung eines Programms vorkommt und dessen normalen Ablauf stört.

Ausnahmen

Wenn Programme nicht auf Ausnahmesituationen reagieren können, führt das zu den von den Anwendern gefürchteten Abstürzen.



In Java wurde von Anfang an die Behandlung von Ausnahmen in die Sprache integriert.

Java ermöglicht eine "weiche" Landung nach Fehlern.

Ausnahmen und Fehler

Ausnahmen
und
Fehler

Fehler (Error) in Java ist ein nicht reparierbarer Laufzeitfehler oder ein Hardware-Problem, das zum Absturz des Programms führt.

Ausnahmen (*Exceptions*) sind meistens keine eigentlichen Fehler, sondern unerwartete Fälle, auf die das Programm reagieren muss.

Warum Ausnahmebehandlung?

Nehmen wir an, wir bieten eine Klasse *Menge* mit entsprechenden Einfüge-, Lösch- und Such-Operationen.

Was passiert, wenn bei einer Suchoperation ein Element nicht gefunden wird?

1. Lösung:

Wir können unsere Suchoperation so definieren, dass die ***null*** Konstante zurückgegeben wird, wenn ein Element nicht gefunden wird.

Probleme:

Der Benutzer muss dieses Verhalten genau kennen, und wenn er das vergisst, führt das oft zum unbeliebten **NullPointerException**-Fehler mit entsprechendem Programmabsturz.

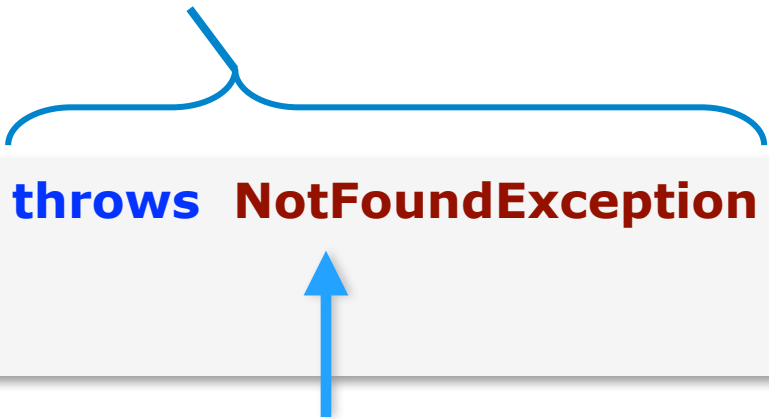
2. Bessere Lösung:

Wir zwingen den Benutzer, den Fall zu behandeln, sonst wird sein Programm nicht übersetzt.

Ausnahmebehandlung in Java

Die Ausnahme wird Bestandteil der Methoden-Signatur

Beispiel:



```
public Object suche ( Object x ) throws NotFoundException {  
    . . .  
}
```

Das hier bedeutet, dass innerhalb des Methodenrumpfs eine Ausnahme mit diesem Namen stattfinden kann.

Ein Ausnahme-Objekt der Klasse **NotFoundException** wird innerhalb der Methode erzeugt, falls das gesuchte Objekt nicht gefunden wird.

Ausnahmebehandlung in Java

Ausnahmen (***Exceptions***) sind unerwartet auftretende **Laufzeitfehler**.

Eine **Ausnahme** in Java ist ein **Objekt**, das eine Instanz der Klasse ***Throwable*** (oder einer ihrer Unterklassen) ist.

- Beispiele:
- Division durch **0**
(***ArithmeticException***)
 - Lesen über Arraygrenzen hinweg
(***IndexOutOfBoundsException***)
 - Lesen über das Ende einer Datei hinaus
(***EOFException***)

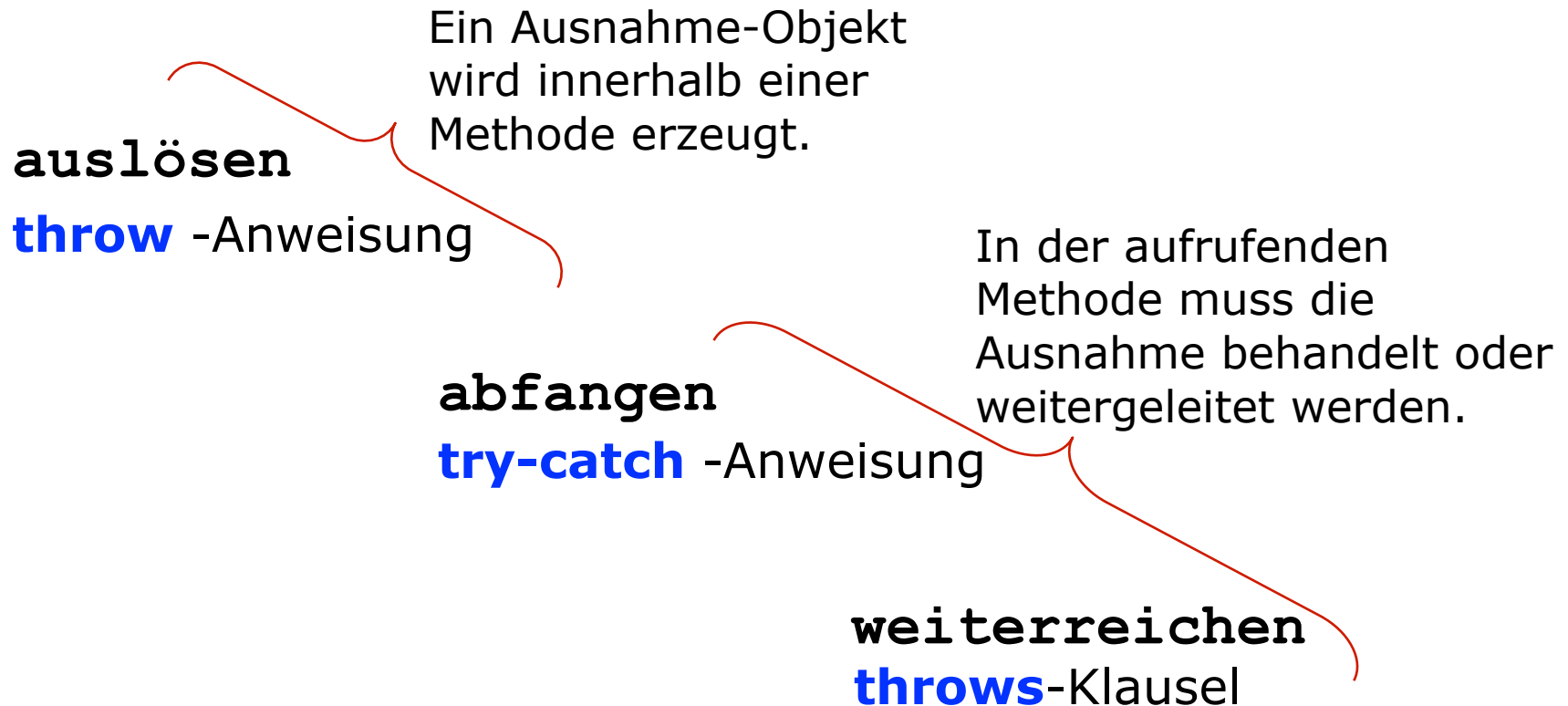
throw-Anweisung

throw **new** Exception ("Schlechte Nachrichten");

```
public class Time{
    private int seconds, minutes, hours;
    ...
    public void setSeconds( int secs) throws Exception
    {
        if( secs<0 || secs>59 )
            throw new Exception( "Falscher ..." );
        else
            this.seconds = secs;
    }
    ...
}
```


Ausnahmebehandlungsschema

Ausnahmen müssen grundsätzlich abgefangen oder weitergereicht werden.



Programmieren von Ausnahme-Klassen

```
public class NegativeUeberweisungException extends Exception {  
  
    double betrag;  
    String dieb;  
    String opfer;  
    // Konstruktor  
    public NegativeUeberweisungException(String opfer,String dieb,double betrag)  
    {  
        super ( "Negative Überweisungen sind nicht erlaubt!" );  
        this.betrag = betrag;  
        this.opfer = opfer;  
        this.dieb = dieb;  
    }  
} // end of class NegativeUeberweisungException
```

Ausnahme-Erzeugung

Ein Ausnahme-Objekt wird mit der **new**-Anweisung erzeugt und mit der **throw**-Anweisung geworfen.

```

. . .
public void ueberweisung ( Bankkonto empfaenger, double betrag )
    throws NegativeUeberweisungException
{
    if ( betrag > 0 ) {
        abheben( betrag );
        empfaenger.zahlen( betrag );
    } else {
        throw new NegativeUeberweisungException
            ( empfaenger.name, this.name, betrag );
    }
}
. . .

```

Das Grundkonzept

Das Grundkonzept der Behandlung von Ausnahmen in Java folgt dem Schema:

throw	Auslösen
try	Ausprobieren
catch	Auffangen

Beim Auftreten einer Ausnahme wird die Ausführung des Programms unterbrochen und ein **Ausnahmeobjekt** "geworfen"

(mit **throw**).

Es wird nach einer passenden Ausnahmebehandlung gesucht, die das Ausnahmeobjekt "auffängt" (**catch**).

Wird keine Ausnahmebehandlung gefunden, wird das Programm abgebrochen.

Behandlung von Ausnahmen

```
...
```

```
try
```

```
{
```

```
    riskyBusiness ( );
```

```
}
```

```
catch ( SomeExceptionType e )
```

```
{
```

```
    // Handle bad stuff
```

```
}
```

```
...
```

Methode, die
Ausnahmefehler
auslösen kann

Referenz des
Ausnahmefehlers, der
"gefangen" werden muss.

Reaktion auf den
Ausnahmefehler

Ausnahme-Behandlung

Wenn eine Methode, die Ausnahmen erzeugen kann, benutzt wird, muss diese in einer **try-catch**-Anweisung umschlossen werden, um mögliche Ausnahmefehler zu behandeln, anderenfalls muss die Behandlung delegiert werden.

Ausnahme-Behandlung

```

. . .
Bankkonto bk1 = new Bankkonto( "Benjamin", 5000 );
Bankkonto bk2 = new Bankkonto( "Tobias", 2000 );
. . .
try {
    bk1.ueberweisung (bk2,-1000);
} catch ( NegativeUeberweisungException nue ) {
    System.out.println( nue.getMessage() );
    System.out.println( nue.betrag );
    System.out.println( "Opfer = " + nue.opfer );
    System.out.println( "Dieb = " + nue.dieb );
}
. . .

```

Negative Überw ... !
 -1000.0
 Opfer = Tobias
 Dieb = Benjamin

Exceptions

Der aufrufende "unsichere" Code wird mit einem **try**- Block umschlossen.

Die darin auftretenden **Exceptions** können mit 1 oder mehr **catch** (**Exception e**) - Blöcken abgefangen werden.

Jeder **Exception-Typ e** wird spezifisch behandelt.

In dem Ausnahme-Objekt kann **zusätzliche Information zwecks Konfliktbereinigung** übergeben werden.

Um den genauen Ort des Fehlers festzustellen, kann man **e.printStackTrace()** aufrufen.

Funktionsprinzip

Mit **try** wird eine Momentaufnahme des Zustands der Umgebung des Programms gemacht.

Stack + Register

Dann wird der nach **try** in {...} angegebene Block betreten.

Tritt eine Programmausnahme auf, so wird der Stack nach **catch**-Routinen durchsucht.

try/catch kann nicht mehr als den Stack zurückrollen und Inhalte von Registern wiederherstellen.

try-catch-Anweisung

```
try
{
    riskyMethod ( );
}
catch ( NullPointerException ne ) {
    // Anweisungen für NullPointerException..
}
catch ( ArithmeticException ae ) {
    // Anweisungen für ArithmeticEx..
}
catch ( IndexOutOfBoundsException ie ) {
    // Anweisungen für IndexOutOfBou..
}
catch ( Exception e ) {
    // behandelt alle anderen Fehler
}
```

riskyMethod() {
wirft einen
Ausnahmefehler
}

Exception

Ausnahmebehandlung delegieren

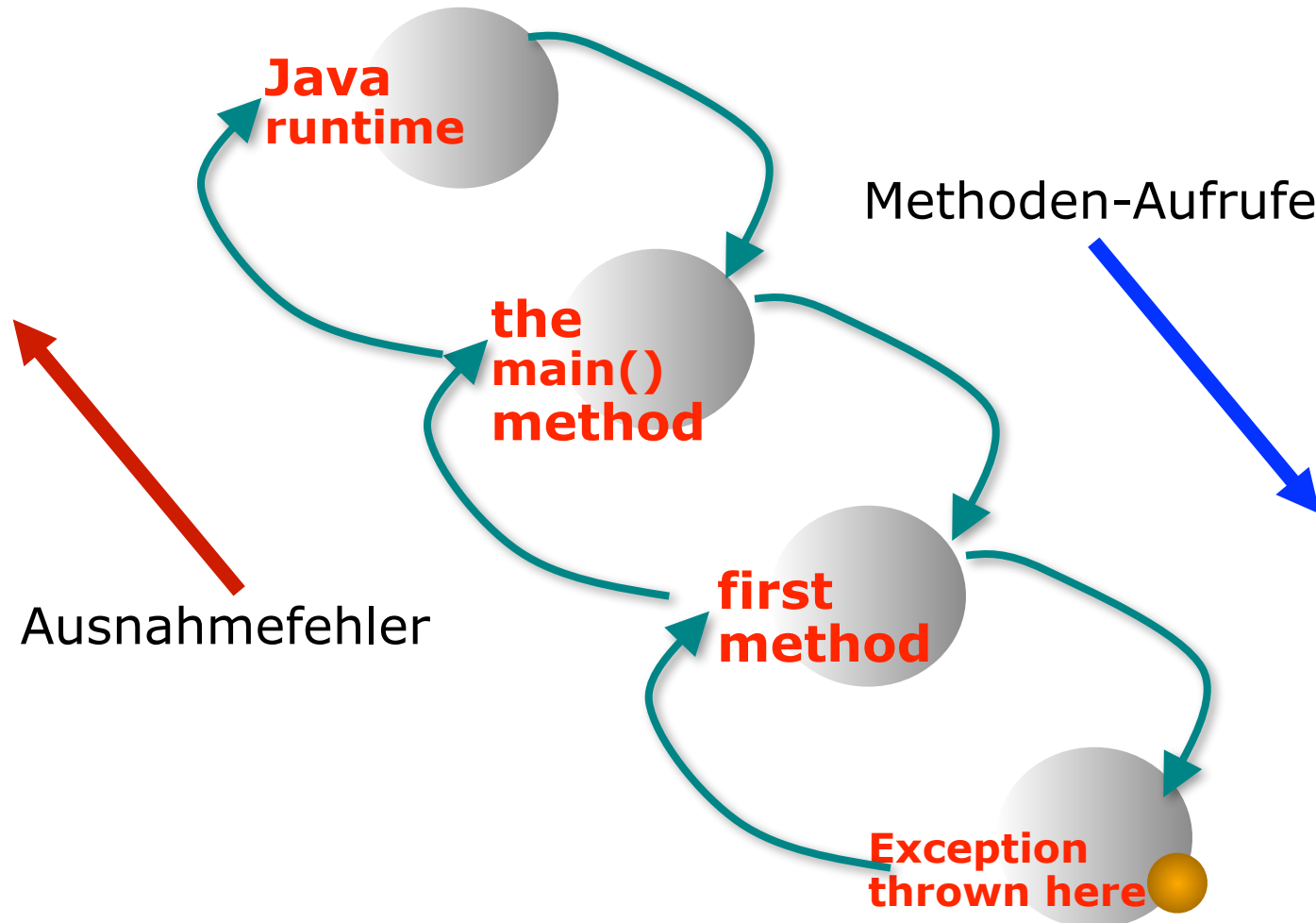
Wenn innerhalb einer Methode die Ausnahmebehandlung nicht gemacht wird, muss wenigstens die Behandlung an die aufrufenden Methoden, mit Hilfe der **throws**-Anweisung, delegiert werden.

Beispiel:

```
. . .  
void zahlt (Bankkonto zahler, Bankkonto empfaenger, double geld)  
    throws NegativeUeberweisungException  
{  
    zahler.ueberweisung ( empfaenger, geld );  
}  
. . .
```

Wenn alle Methoden die Ausnahmebehandlung delegieren, kann ein Laufzeitfehler verursacht werden.

Laufzeitfehler



```
main {  
  method_1( );  
}
```

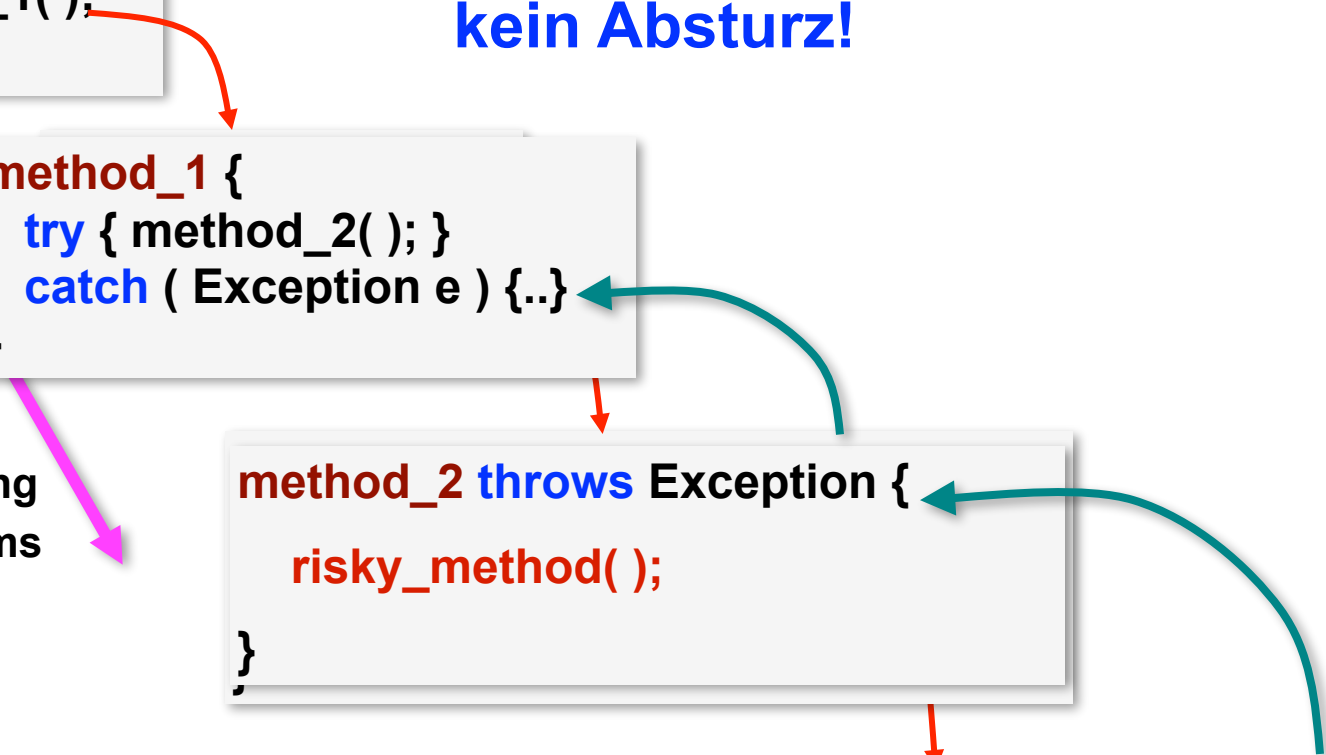
kein Absturz!

```
method_1 {  
  try { method_2( ); }  
  catch ( Exception e ) {..}  
}
```

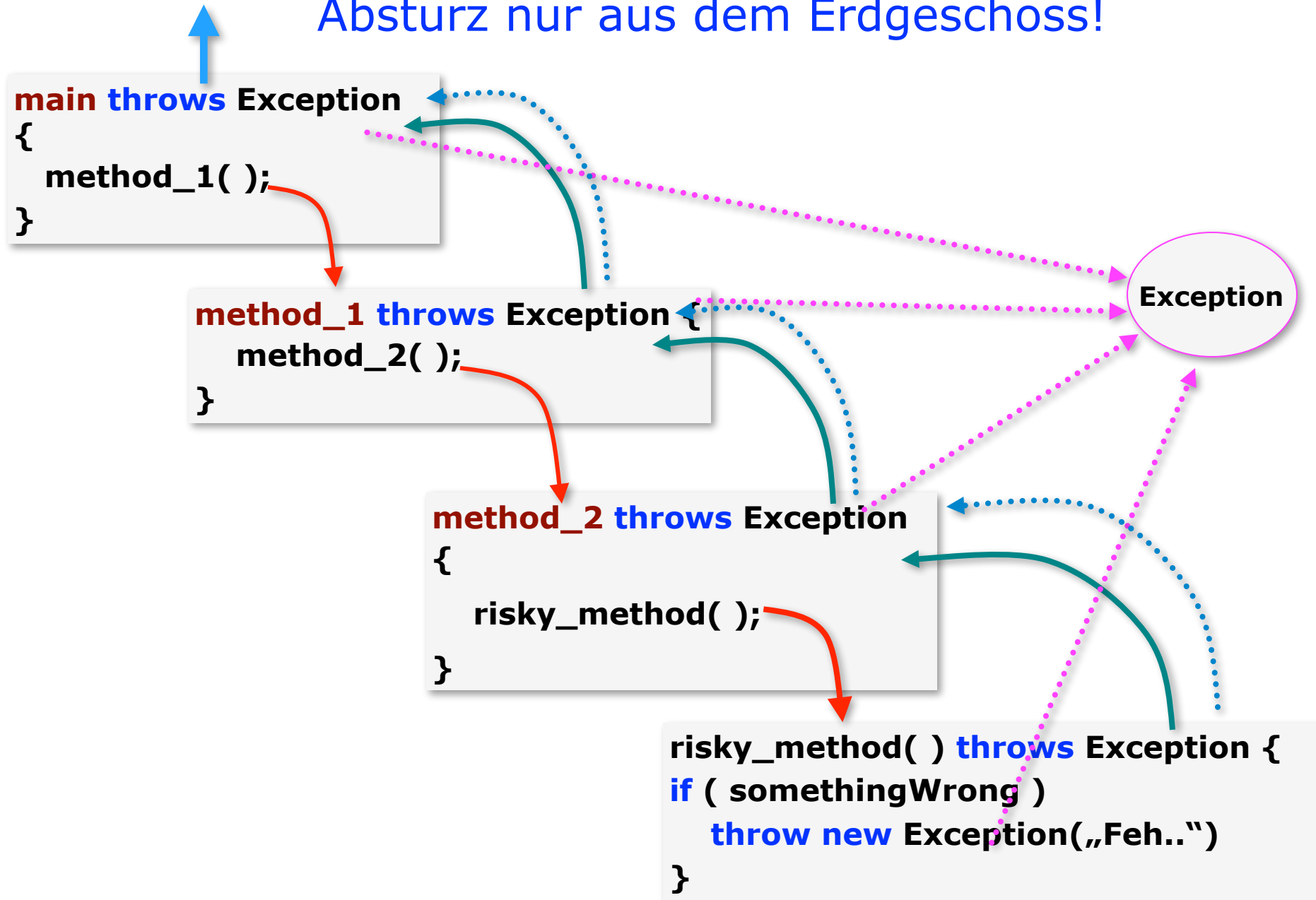
Die Ausführung
des Programms
geht weiter.

```
method_2 throws Exception {  
  risky_method( );  
}
```

```
risky_method( ) throws Exception {  
  if ( somethingWrong )  
    throw new Exception( "Feh..." )  
}
```



Absturz nur aus dem Erdgeschoss!



throws-Klausel

```
public int readInt()  
    throws IOException, NumericFormatException  
{  
    int ch;  
    int integer = 0;  
    String s = "";  
    while ( ( ch = System.in.read() ) != '\n' )  
        { s = s + (char) ch; }  
    integer = Integer.parseInt(s);  
    return integer;  
}
```

kann eine
NumericFormatException !!
auslösen

kann eine
IOException !!
auslösen

try-catch-finally-Anweisung

Es besteht zusätzlich die Möglichkeit, zu jedem **try**-Block Anweisungen zu definieren, die auch im Ausnahmefall sicher ausgeführt werden.

- Beispiele:
- Objekte in einen sicheren Zustand bringen
 - Dateien schließen

Dazu wird ein **finally**-Block vereinbart, der diese Anweisungen enthält:

```
try { ... }  
  catch (  $E_1$   $e_1$  ) { ... }  
  :  
  catch (  $E_n$   $e_n$  ) { ... }  
  finally { ... }
```

wird immer ausgeführt!

try-catch-finally-Anweisung

Es gab **keine Ausnahme!**

Es gab **eine Ausnahme** und diese wurde **in** einer **catch-Anweisung behandelt !**

Es gab **eine Ausnahme**, aber **keine catch-Anweisung** war für diesen Fehler zuständig !

In allen 3 Fällen wird die Finally-Anweisung ausgeführt!

Exception-Klasse

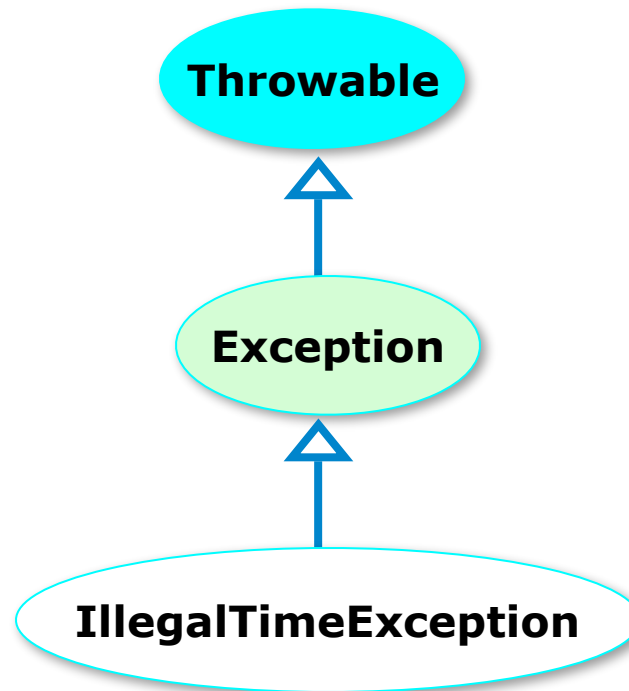
Konstruktoren:

```
Exception( )  
Exception( String s )
```

Einige Methoden, die von der Klasse `java.lang.Throwable` vererbt werden, sind:

```
getMessage()  
printStackTrace()  
toString()  
fillInStackTrace()  
getLocalizedMessage()
```

Definition neuer Ausnahmen



Neue Ausnahmeklassen werden als Unterklasse der Klasse **Exception** definiert.

Definition neuer Ausnahme-Klassen

```
public class IllegalTimeException extends Exception {  
    Time wrongTime;  
    public IllegalTimeException ( String reason ) {  
        super ( reason );  
    }  
    public IllegalTimeException (String reason, Time wrongTime) {  
        super ( reason );  
        this.wrongTime = wrongTime;  
    }  
    ...  
}
```

RuntimeException

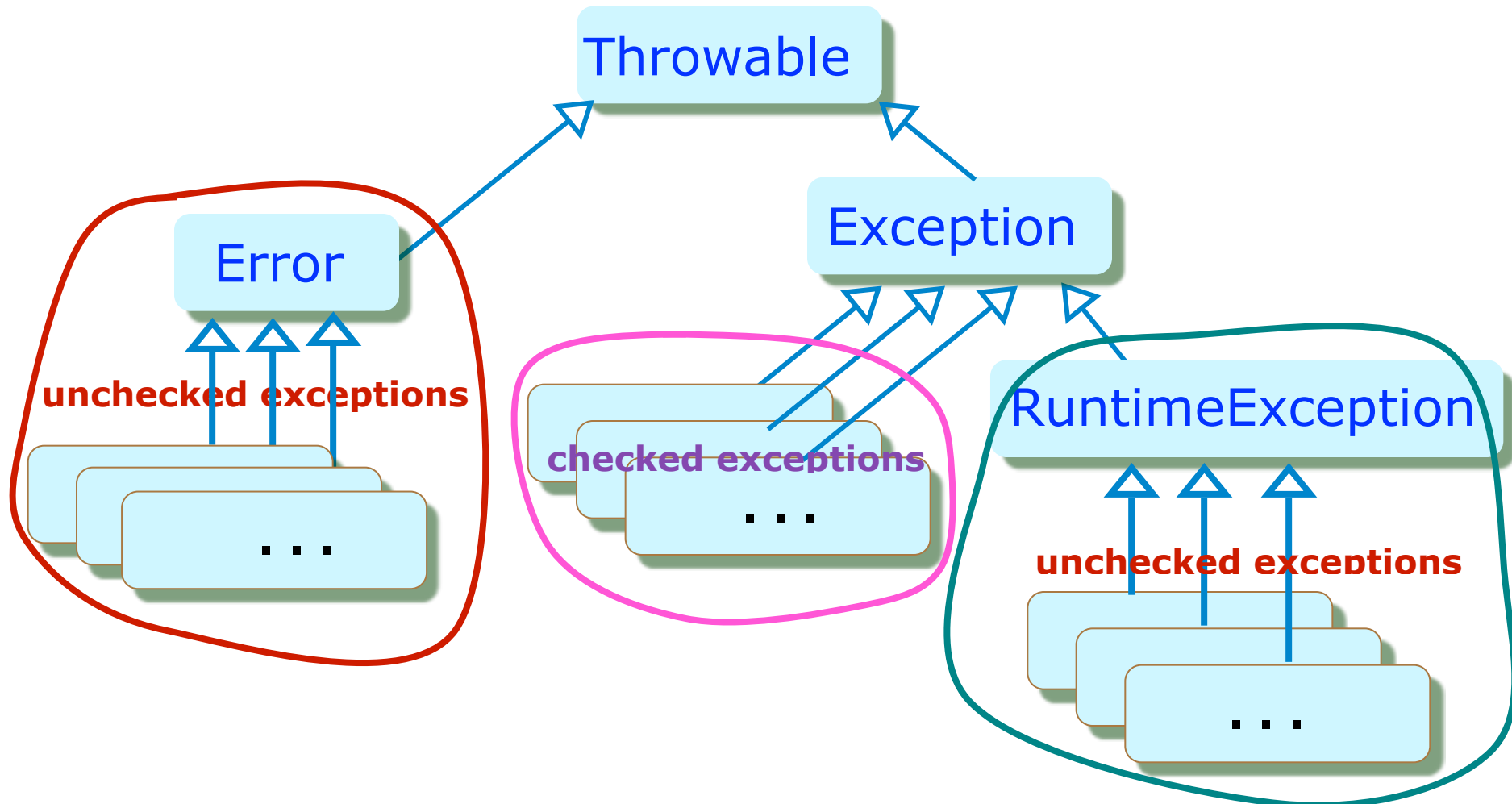
Exceptions, die Unterklassen von **RuntimeException** sind, sind sogenannte "**unchecked exceptions**" – sie müssen nicht in einer **throws**-Klausel deklariert werden, und der Übersetzer verlangt keine expliziten Ausnahmebehandlungen.

RuntimeExceptions sind Ausnahmen, die durch gutes Programmieren vermeidbar sind!!

"checked exceptions"

Ausnahmen, die im Allgemeinen nicht vermeidbar sind, aber vom Programmierer behandelt werden können bzw. sollten.
Sie müssen in einer **throws**-Klausel deklariert werden.

Verschiedene Arten von Ausnahmen



Gemeinsame Oberklassen

Ist die Behandlung aller möglichen Ausnahmen jeweils gleich, kann man sie ggf. auch zusammenfassen. Z.B. haben Ausnahmeobjekte bei der Ein-/Ausgabe eine gemeinsame Oberklasse **IOException**.

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

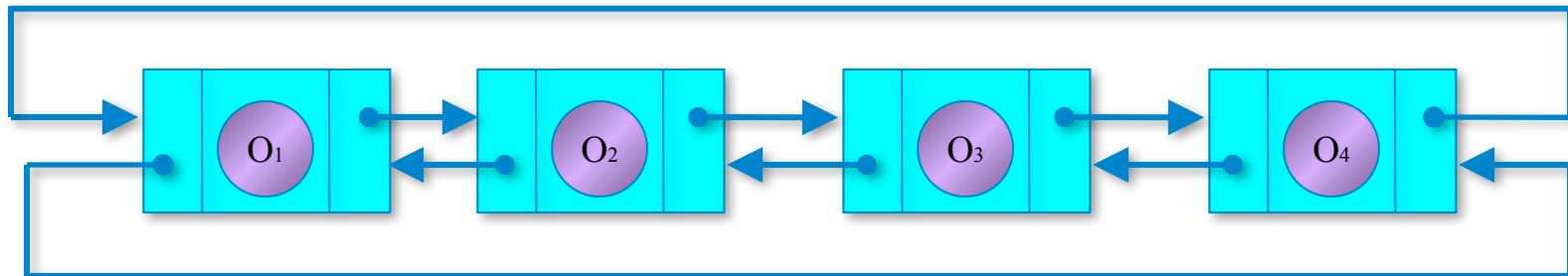
Regeln zum Umgang mit Ausnahmen

- Ausnahmebehandlung nicht zur Behandlung normaler Programmsituationen einsetzen !
- Ausnahmebehandlung nicht in zu kleinen Einheiten durchführen !
- Auf keinen Fall Ausnahmen "abwürgen" oder "ignorieren" z.B. durch triviale Fehlermeldungen !
- Ausnahmen zu propagieren ist keine Schande!

OOP

Dynamische Datenmengen

Datenabstraktion



Datenabstraktion

Verschiedene Verständnisse von Datenabstraktion:

- * **Allgemeine Abstraktion**

Details werden hinter einer allgemeinen einfachen Idee versteckt.

- * **Baukastenprinzip**

Software-Architektur mit unabhängigen Modulen, die getrennt entwickelt und getestet werden können.

- * **Datenkapselung**

Kontrollierter Zugriff auf Daten und Methoden, um die interne Integrität von Objekten zu bewahren.

- * **Schnittstellen**, die von Implementierung-Details getrennt sind und dadurch diese Information vom Rest des Systems verstecken. Die Implementierungen sind dadurch unabhängig und veränderbar.

- * **Zuständigkeitstrennung**

Einzelne Module übernehmen die Verantwortung für Features

Abstrakte Datentypen ADT

Erste Ideen:

Ole-John Dahl

Erfinder der Programmiersprache Simula



Ausgereifte Idee

Barbara Liskov vom MIT

Erhielt den Turing Award für die Arbeit im Bereich ADT



Invarianten von ADT

Invarianten sind Eigenschaften oder Merkmale, die immer gelten.

Gut definierte abstrakte Datentypen haben auch Invarianten und sorgen selber dafür, dass diese erhalten bleiben.

Eine wichtige Invariante vom String ist, dass die unveränderter sind (immutable)

Eine gut definierte ADT für Heap-Datenstrukturen sorgt dafür, dass die Heap-Eigenschaft nach jeder Operation wieder hergestellt wird.

Dynamische Datenmengen als ADT

Dynamische Datenmengen können durch verschiedene **Datenstrukturen** im Rechner dargestellt werden.



Stapel und Schlangen

Stapel und Schlangen sind die einfachsten dynamischen Datenstrukturen.

Mögliche Implementierungen:

- Arrays
- "Dynamische Arrays"

Wenn ein Feld voll ist, wird zur Laufzeit ein neues erzeugt, das doppelt so groß ist, und alle Daten des alten Feldes werden auf das neue Feld kopiert. Das Ganze wird wiederholt, wenn das Feld wieder ausgefüllt ist.

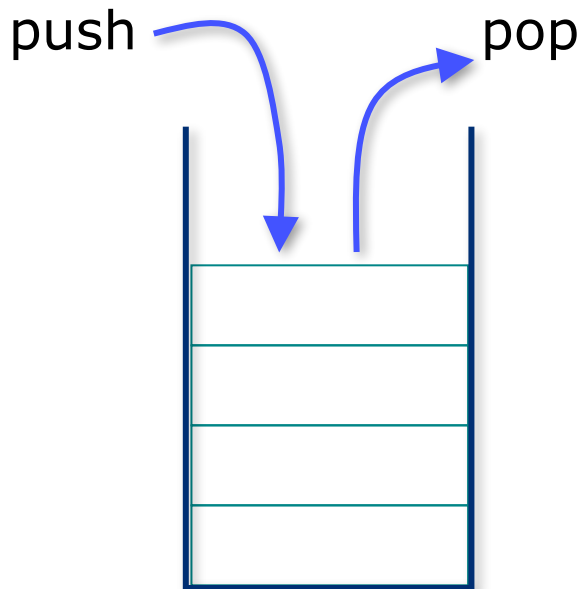
- Verkettete Listen

Stapel

In einem Stapel ("**stack**") darf nur das Element entfernt werden, das als letztes eingeführt worden ist.

LIFO - Datenstruktur

"**L**ast **I**n - **F**irst **O**ut"



Mögliche Operationen:

push ist der Standard-Name der Einfüge-Operation in einem Stapel (**stack**).

pop ist der Standard-Name der Lösch-Operation.

peek liest das nächst verfügbare Element des Stapels, ohne dieses Element zu entfernen.

empty überprüft, ob der Stapel leer ist.

full überprüft, ob der Stapel voll ist.

Stapel-Schnittstelle

```
public interface Stack <E> {  
  
    public boolean empty();  
  
    public void push( E elem );  
  
    public E pop() throws EmptyStackException;  
  
    public E peek() throws EmptyStackException;  
  
}
```

In dieser Implementierung werden wir eine `EmptyStackException` erzeugen bei dem Versuch, ein Element zu entfernen oder zu lesen (`pop`- und `peek`-Operationen), wenn der Stapel leer ist.

Stapel-Schnittstelle

```
void push( E elem );
```

Das **elem**-Objekt wird als oberstes Element des Stapels eingefügt

```
E pop() throws EmptyStackException;
```

Wenn der Stapel nicht leer ist, wird das oberste Element des Stapels entfernt und als Ergebnis zurückgegeben, andernfalls wird ein **EmptyStackException**-Objekt erzeugt.

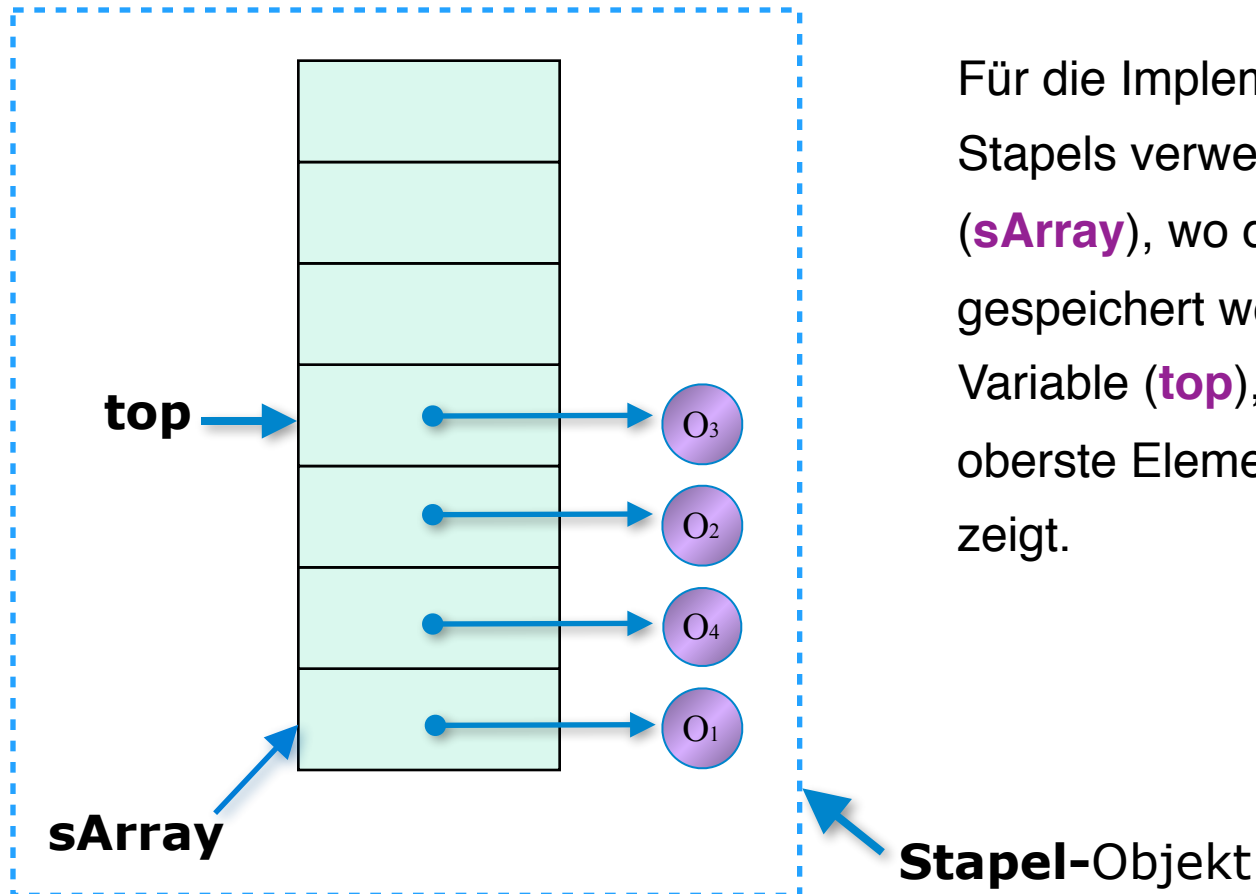
```
E peek() throws EmptyStackException;
```

Wenn der Stapel nicht leer ist, wird das oberste Element des Stapels gelesen und als Ergebnis zurückgegeben, andernfalls wird ein **EmptyStackException**-Objekt erzeugt.

```
boolean empty();
```

Überprüft, ob der Stapel leer ist.

Implementierung der Stapel-Schnittstelle



Für die Implementierung unseres Stapels verwenden wir ein Array (**sArray**), wo die Stapелеlemente gespeichert werden und eine **int**-Variable (**top**), die immer auf das oberste Element des Stapels zeigt.

Implementierung der Stack-Schnittstelle

Im **sArray** werden die Stapелеlemente gespeichert.

top zeigt immer auf das oberste Element des Stapels.

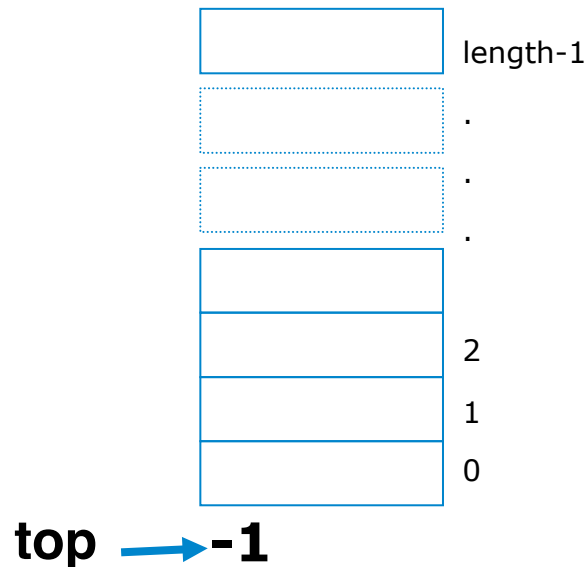
Zwei Konstruktoren werden definiert, die das Array mit einer Anfangsgröße initialisieren, und den **top**-Zeiger mit **-1** (für leere Stapel) initialisieren.

```
public class ArrayStapel<E> implements Stack<E> {
    private E[] sArray;
    private int top;

    public ArrayStapel(E[] sArray){
        top = -1;
        this.sArray = sArray;
    }

    public ArrayStapel(){
        this( (E[]) new Object[100]);
    }
    ...
}
```

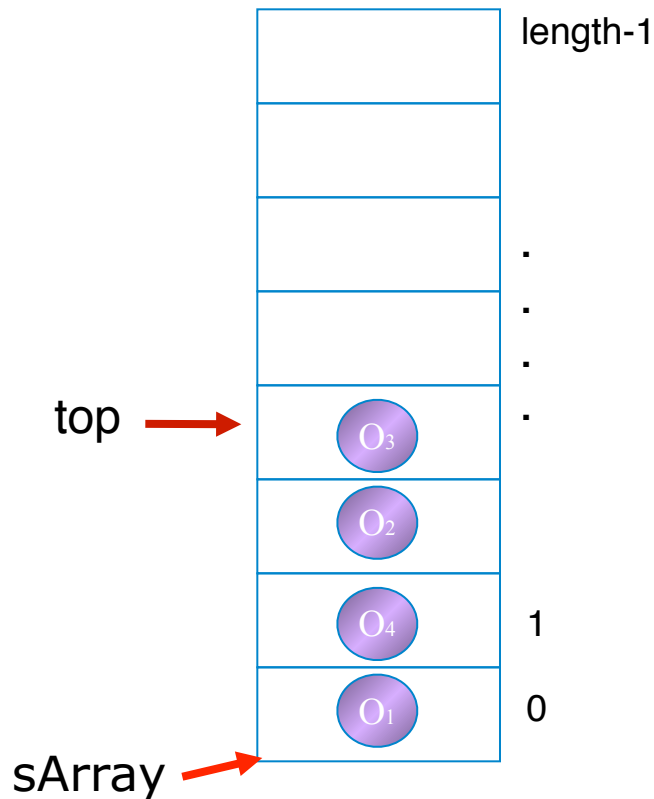
Die **empty**-Operation des Stapels



Der Stapel ist leer, wenn **top** auf keinen gültigen Stapelplatz zeigt.

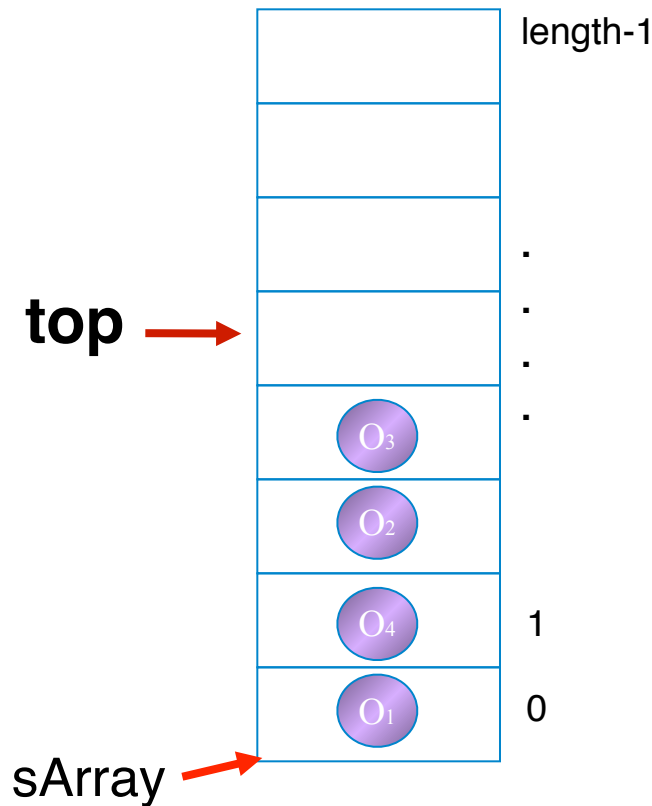
```
public boolean empty() {  
    return top == -1;  
}
```

Die **push**-Operation



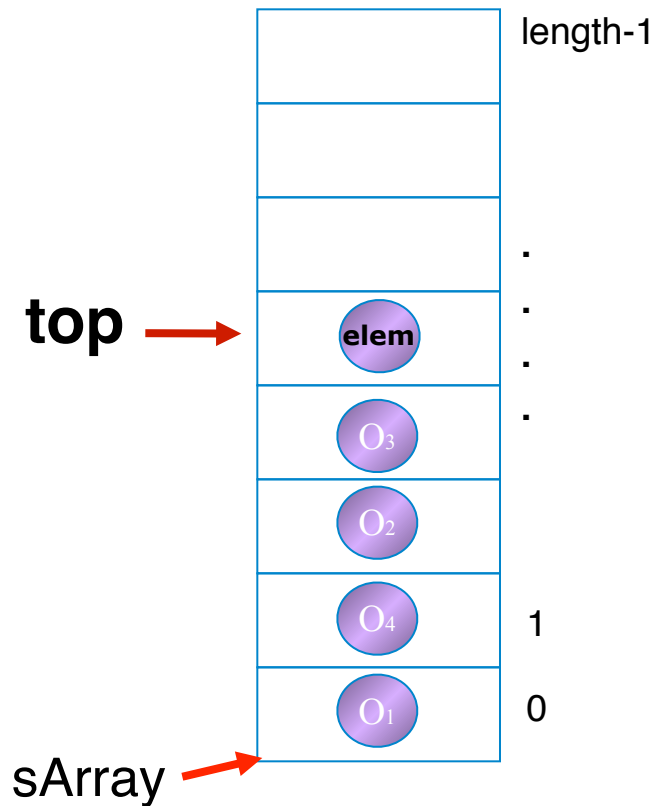
```
public void push(E elem) {
    if ( !full() ) {
        top++;
        sArray[top] = elem;
    } else {
        resizeSArray();
        top++;
        sArray[top] = elem;
    }
}
```

Die **push**-Operation



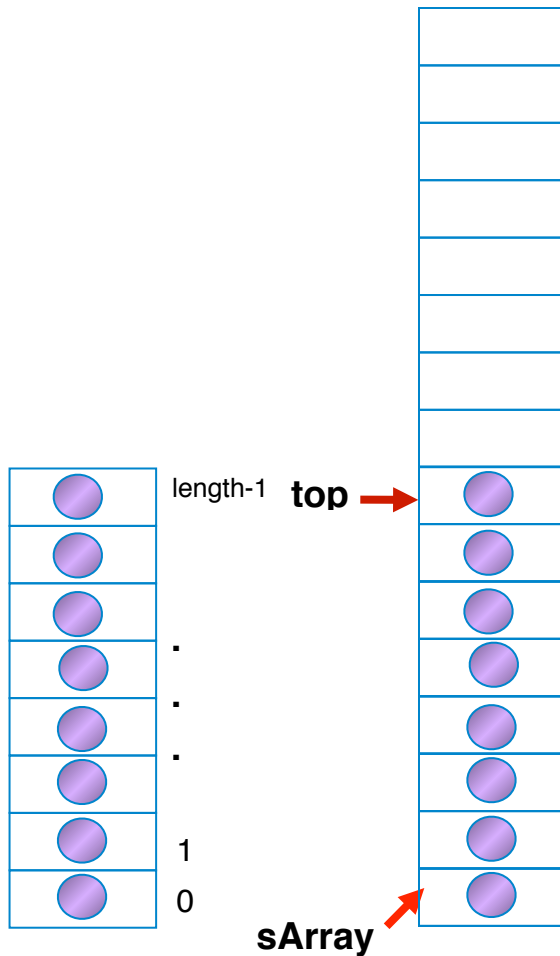
```
public void push(E elem) {
    if ( !full() ) {
        top++;
        sArray[top] = elem;
    } else {
        resizeSArray();
        top++;
        sArray[top] = elem;
    }
}
```

Die **push**-Operation



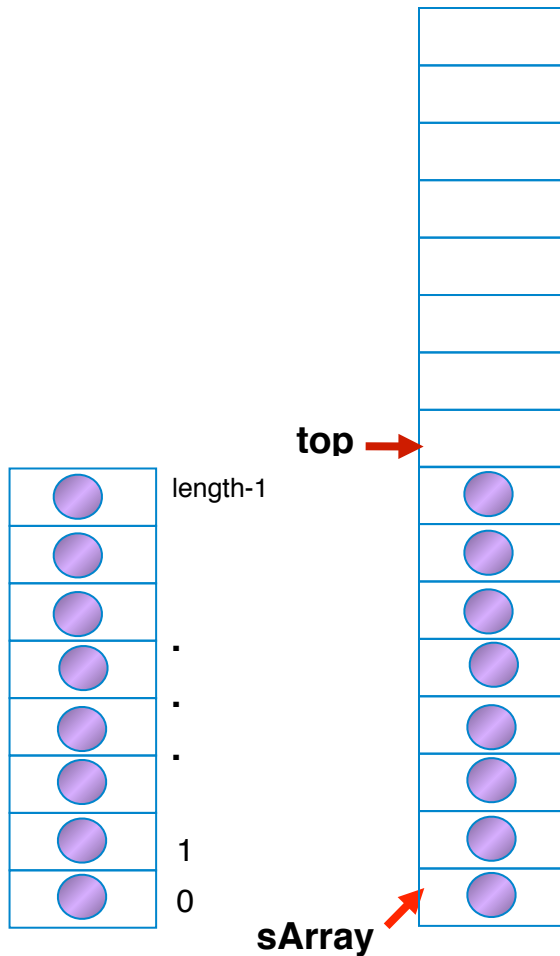
```
public void push(E elem) {
    if ( !full() ) {
        top++;
        sArray[top] = elem;
    } else {
        resizeSArray();
        top++;
        sArray[top] = elem;
    }
}
```

Die **push**-Operation



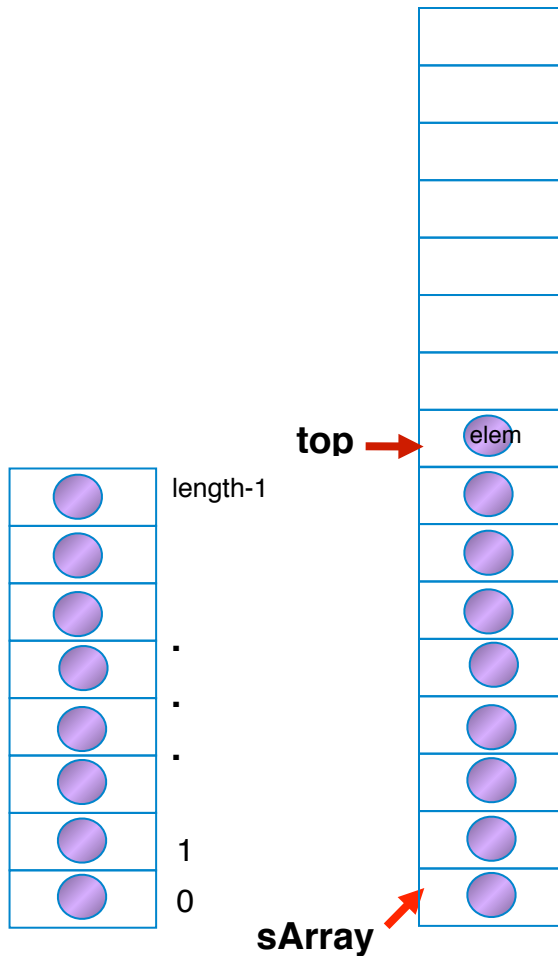
```
public void push(E elem) {
    if ( !full() ) {
        top++;
        sArray[top] = elem;
    } else {
        resizeSArray();
        top++;
        sArray[top] = elem;
    }
}
```


Die **push**-Operation



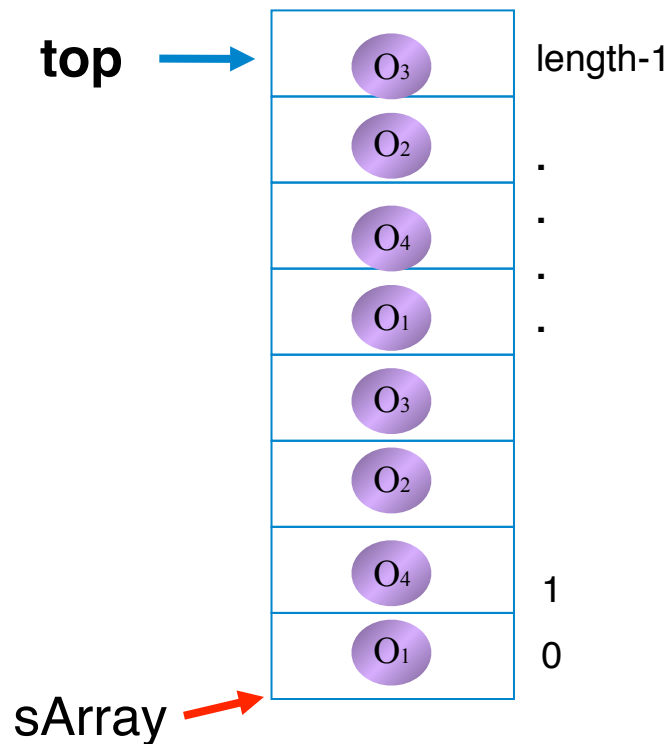
```
public void push(E elem) {
    if ( !full() ) {
        top++;
        sArray[top] = elem;
    } else {
        resizeSArray();
        top++;
        sArray[top] = elem;
    }
}
```

Die **push**-Operation



```
public void push(E elem) {
    if ( !full() ) {
        top++;
        sArray[top] = elem;
    } else {
        resizeSArray();
        top++;
        sArray[top] = elem;
    }
}
```

Die **full**-Hilfsmethode



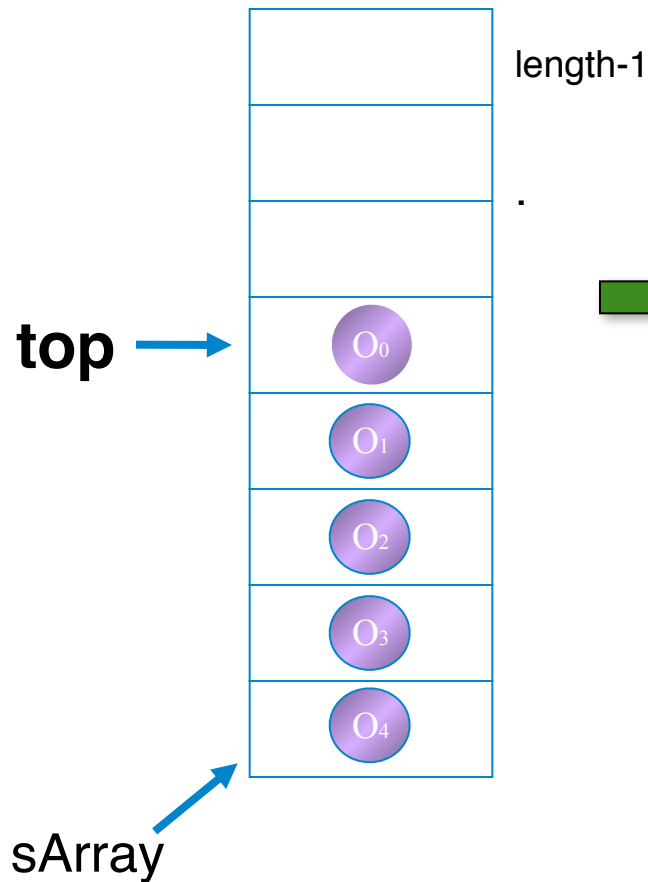
Der Stapel ist voll, wenn **top** gleich **stack.length-1** wird.

```
private boolean full() {  
    return !( top < sArray.length-1 );  
}
```

Die **resizeArray**-Hilfsmethode

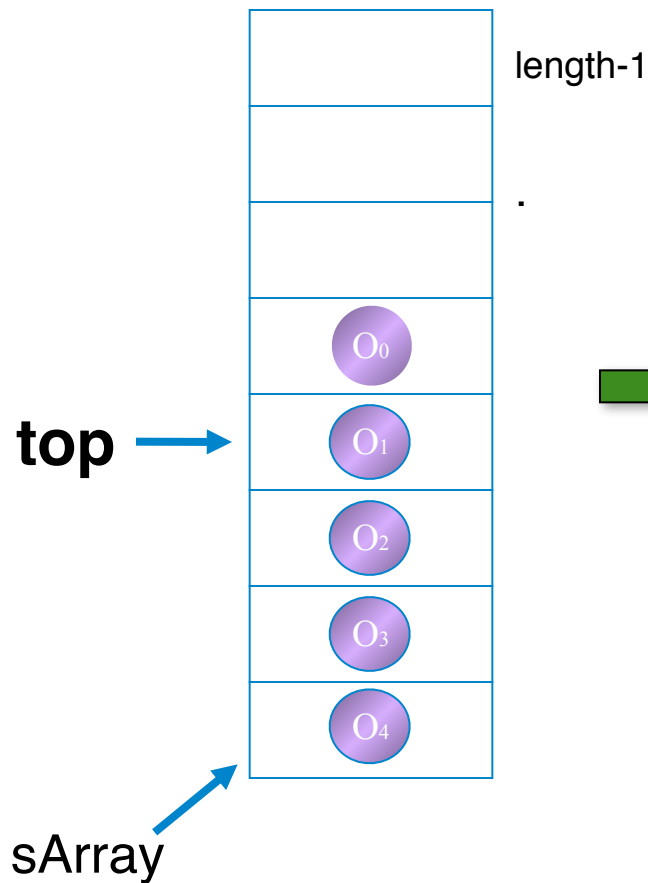
```
private void resizeArray(){  
    E[] temp = (E[]) new Object[sArray.length*2];  
    for (int i=0; i<sArray.length; i++){  
        temp[i] = sArray[i];  
    }  
    sArray = temp;  
}
```

Die **pop**-Operation



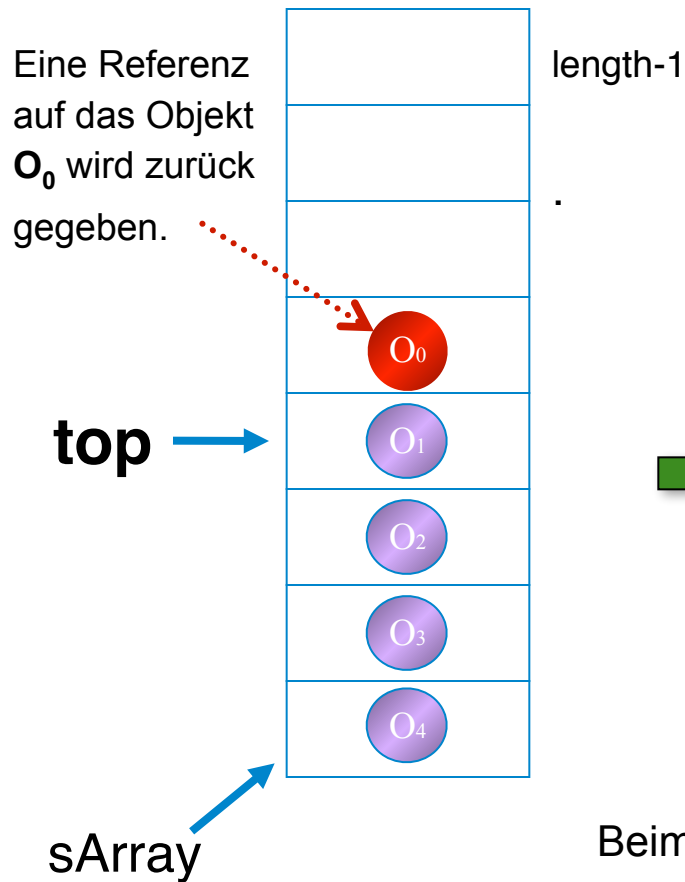
```
public E pop ()  
    throws EmptyStackException {  
    if ( !empty() ) {  
        top--;  
        return sArray [ top+1 ];  
    } else  
        throw new EmptyStackException();  
}
```

Die **pop**-Operation



```
public E pop ()  
    throws EmptyStackException {  
    if ( !empty() ) {  
        top--;  
        return sArray [ top+1 ];  
    } else  
        throw new EmptyStackException();  
    }
```

Die **pop**-Operation

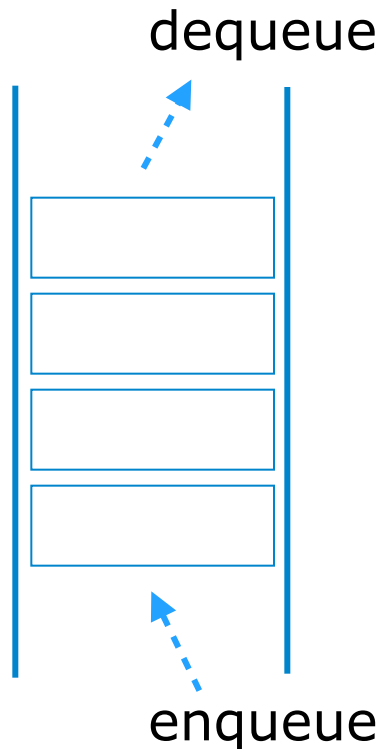


```
public E pop ()
    throws EmptyStackException {
    if ( !empty() ) {
        top--;
        return sArray [ top+1 ];
    } else
        throw new EmptyStackException();
}
```

Beim Einfügen eines neuen Elements im **sArray** wird die alte Referenz einfach überschrieben.

Implementierung einer Warteschlange als Array

Warteschlangen implementieren eine **FIFO**-Strategie. D.h. das erste Element, das eingeführt worden ist, ist das erste, das später entfernt wird.



FIFO - Datenstruktur

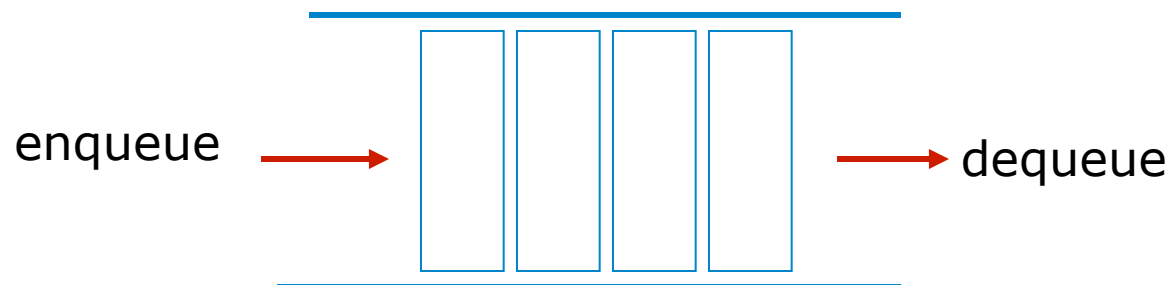
"**F**irst **I**n - **F**irst **O**ut"

enqueue ist der Standard-Name der Einfüge-Operation.

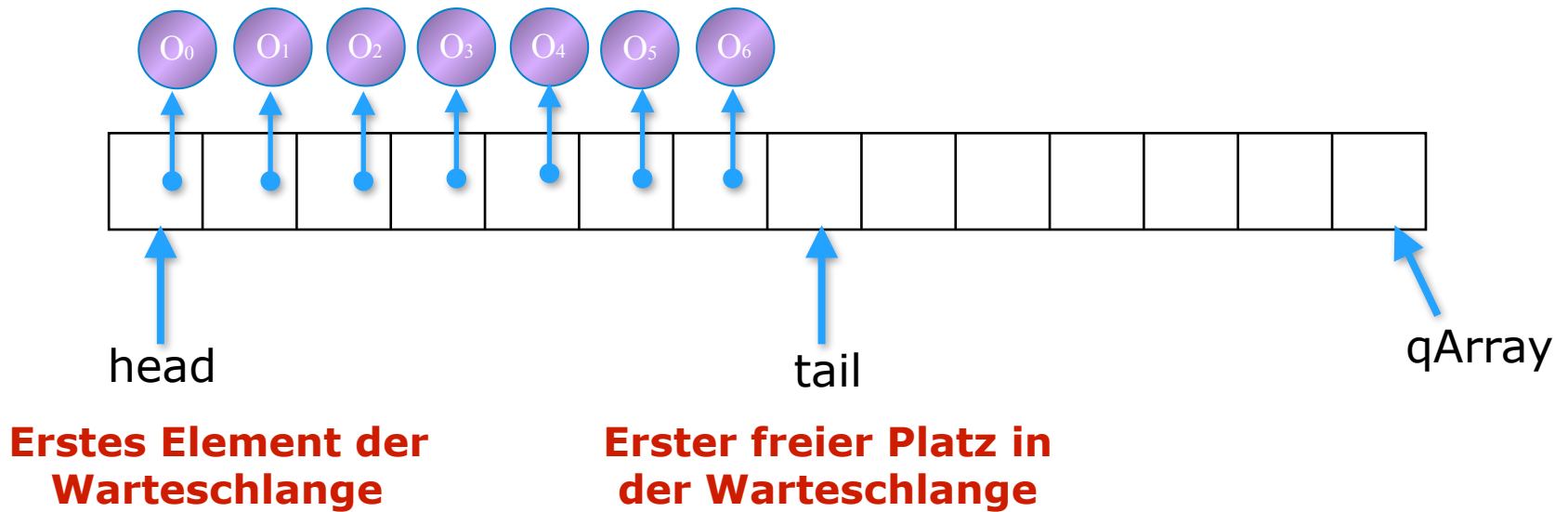
dequeue ist der Standard-Name der Lösch-Operation.

Warteschlangen-Operationen

Operationen {
enqueue
dequeue
head
empty
full



Implementierung der Warteschlange



Implementierung einer Warteschlange



Implementierung einer Warteschlange



Implementierung einer Warteschlange



alle Personen bewegen sich!

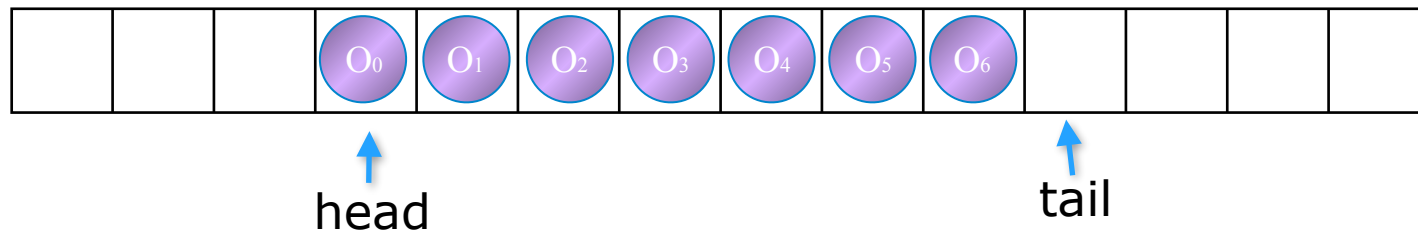
keine gute Idee für die Implementierung!

Wraparound-Strategie

Um die **Einfüge**- und **Lösch**-Operation in unserer Warteschlange in einer konstanten Zeit **$O(1)$** zu realisieren, wird das Feld in unserer Warteschlange als eine zirkulare Struktur betrachtet.

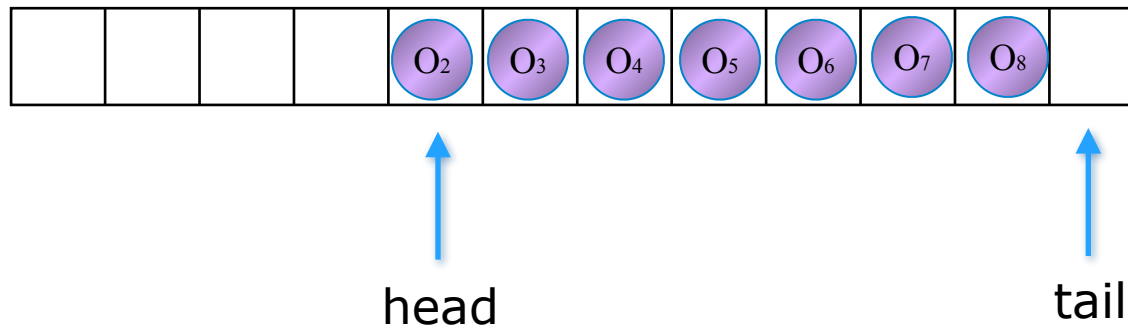
Neue Elemente in der Warteschlange werden in der Position eingefügt, die von **tail** angezeigt wird, und **tail** wird um eine Position nach rechts verschoben.

Wenn ein Element aus der Warteschlange entfernt wird, wird der **head**-Zeiger einfach um eine Position nach rechts bewegt.



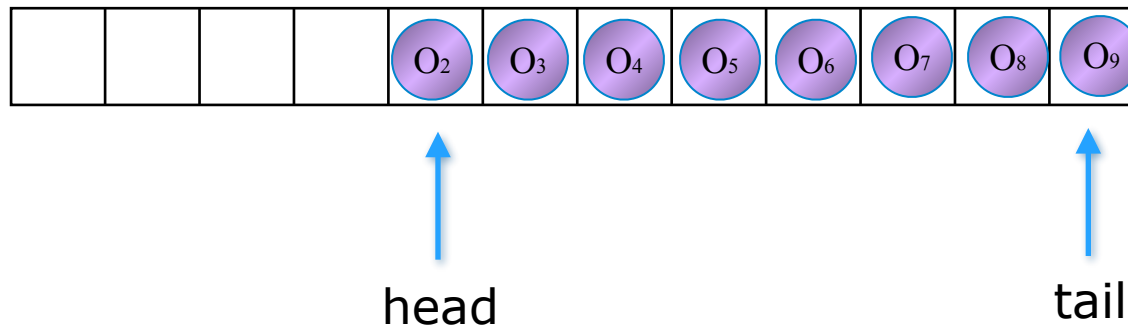
Wraparound-Strategie

enqueue



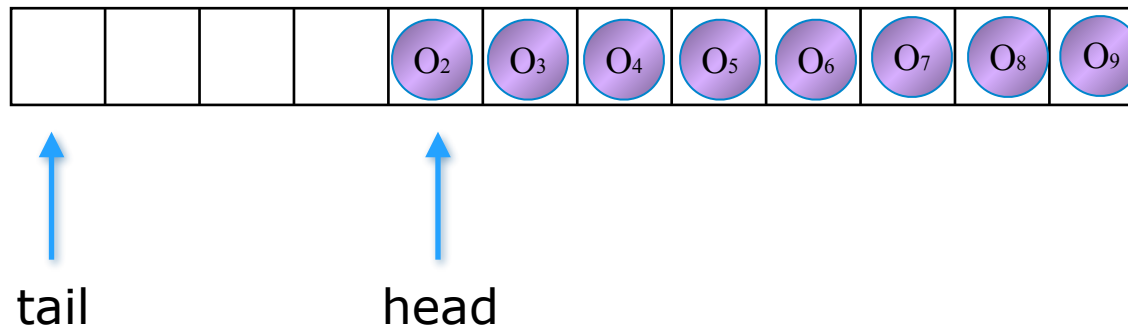
Wraparound-Strategie

enqueue



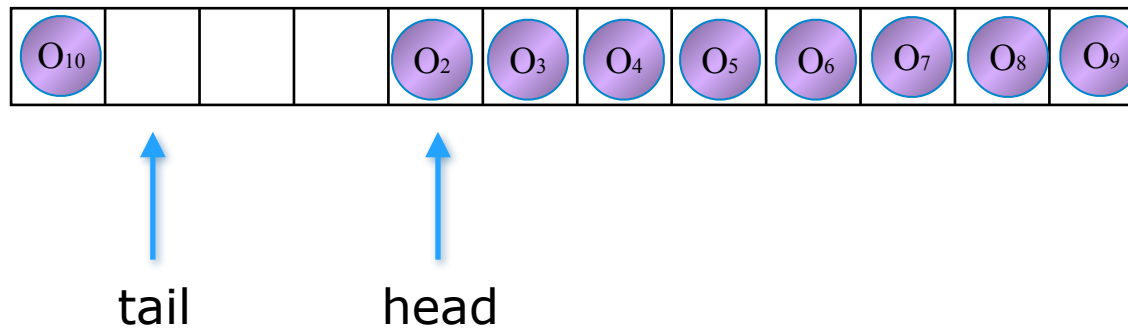
Wraparound-Strategie

enqueue



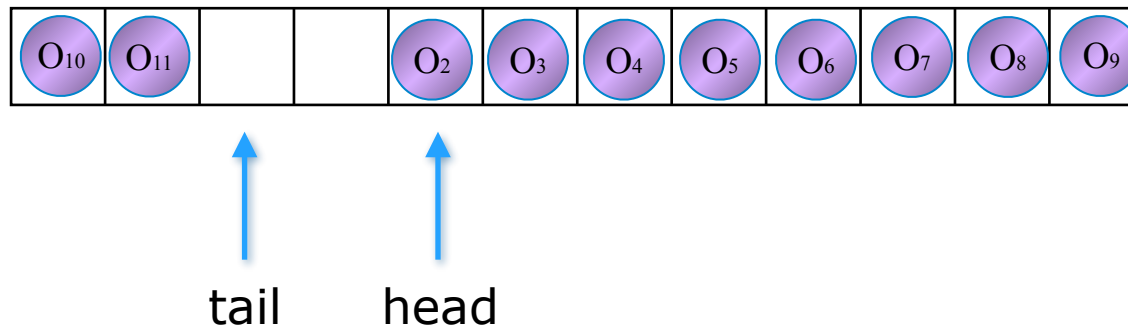
Wraparound-Strategie

enqueue



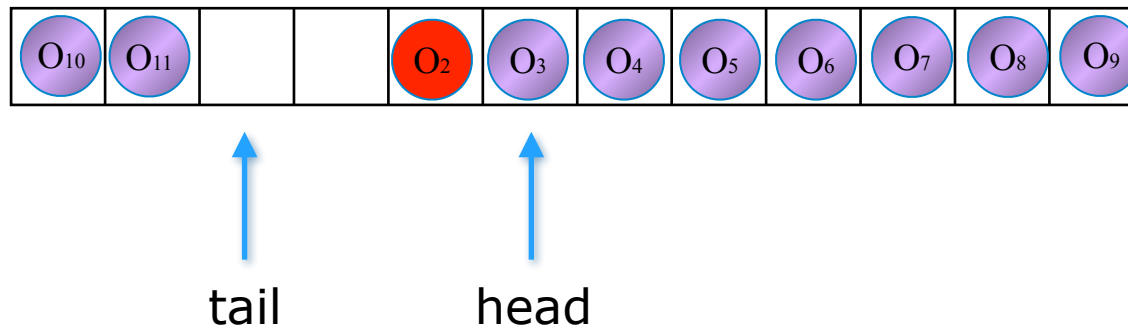
Wraparound-Strategie

enqueue



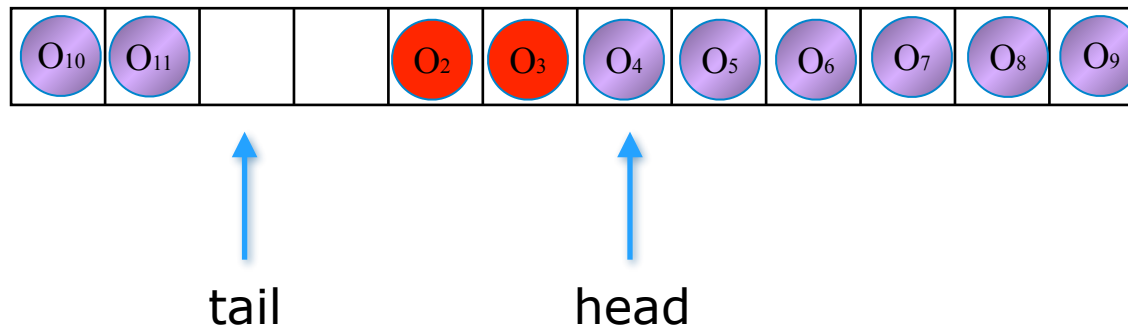
Wraparound-Strategie

dequeue



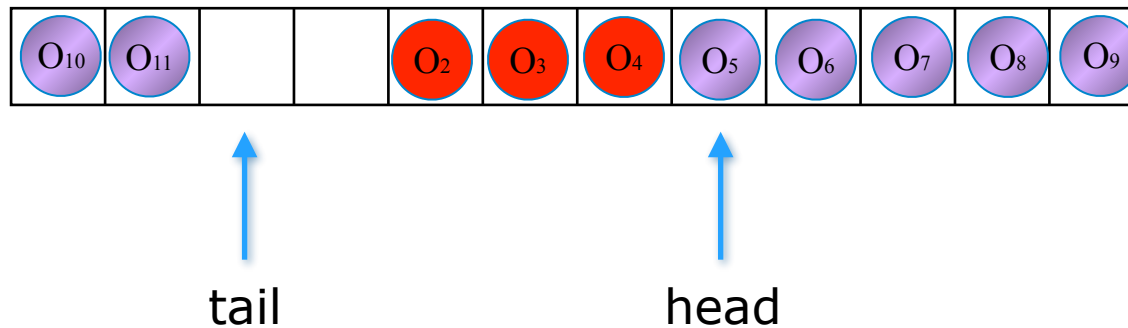
Wraparound-Strategie

dequeue



Wraparound-Strategie

dequeue



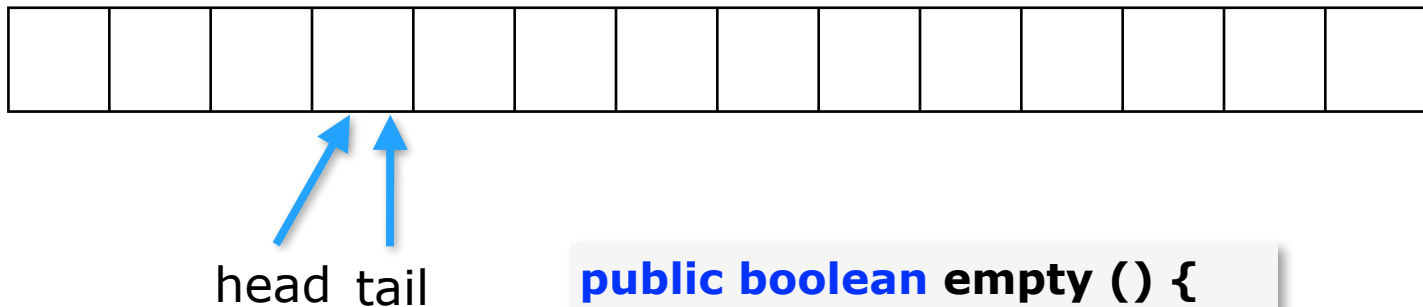
Die Warteschlangen-Schnittstelle

```
public interface Queue <E> {  
    public void enqueue( E elem ) throws FullQueueException;  
    public E dequeue() throws EmptyQueueException;  
    public E head() throws EmptyQueueException;  
    public boolean empty();  
    public boolean full();  
    public void toString();  
}
```

Warteschlange mit fester Größe!

Die **empty**-Operation

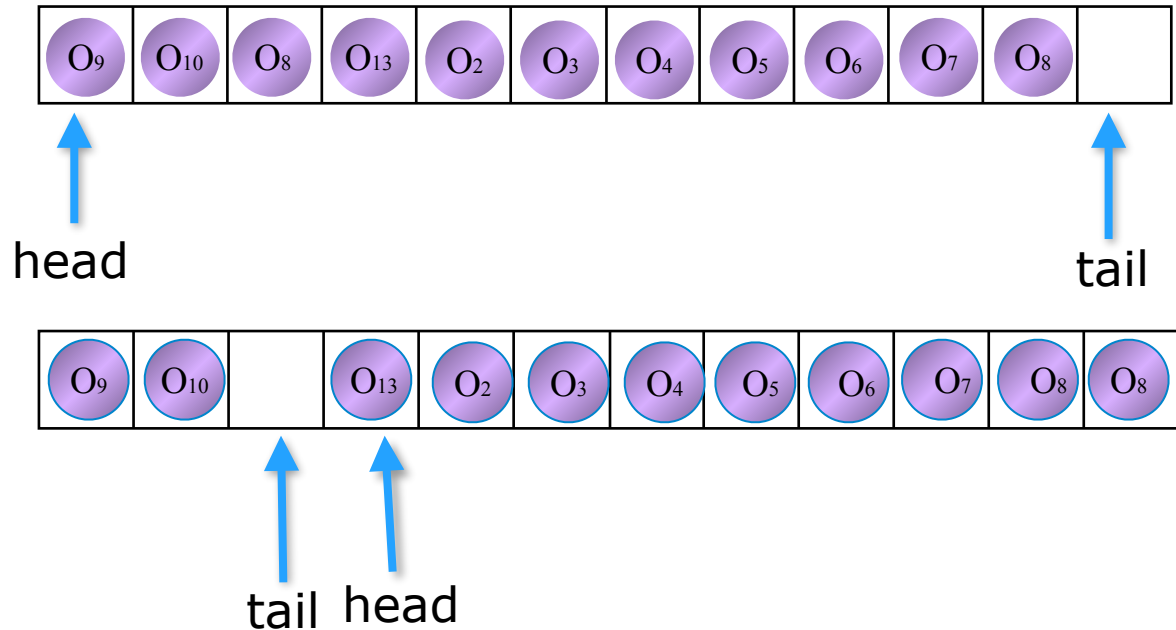
Unsere Warteschlange ist leer, wenn **head** und **tail** auf die gleiche Position des Feldes zeigen.



```
public boolean empty () {  
    return head == tail;  
}
```

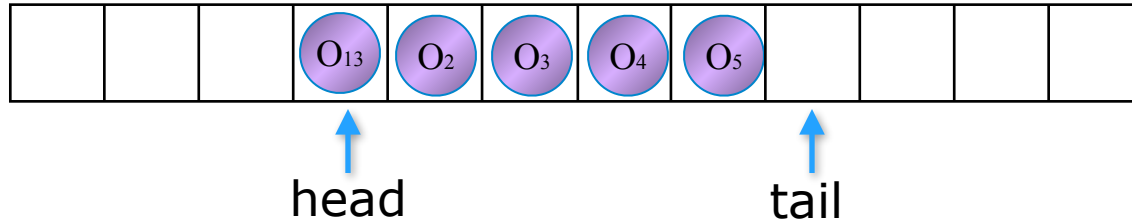

Die **full**-Operation

Wenn in diese Felder das letzte Element eingefügt wird, werden **tail** und **head** auch gleich sein und die Warteschlange wäre voll. D.h., wir würden nicht eine leere von einer vollen Warteschlange unterscheiden können. Deshalb werden wir nie unsere Warteschlange ausfüllen und den Zustand **full** definieren, wenn **tail** eine Position von **head** entfernt liegt.



```
public boolean full () {  
    return (( tail == queue.length-1 ) && ( head == 0 ))  
           || ( head == ( tail+1 ) ) ;  
}
```

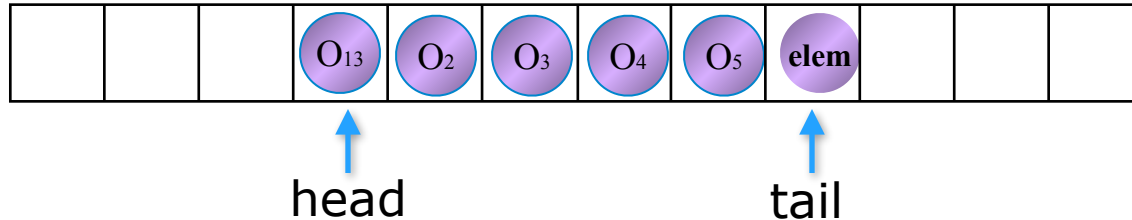
Die **enqueue**-Operation



Wenn die
Warte-
schlange
nicht voll
ist

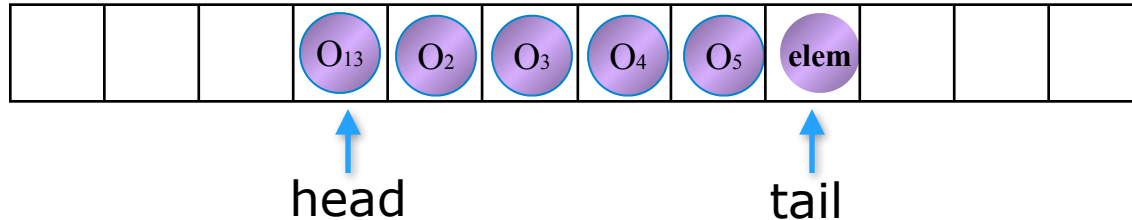
```
public void enqueue ( E elem ) throws FullQueueException {  
    if ( !full() ) {  
        queue[tail] = elem;  
        if ( tail == (queue.length-1) )  
            tail = 0;  
        else tail++;  
    } else  
        throw new FullQueueException();  
}
```

Die **enqueue**-Operation

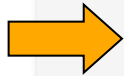


```
public void enqueue ( E elem ) throws FullQueueException {  
    if ( !full() ) {  
        ➡ queue[tail] = elem;  
        if ( tail == (queue.length-1) )  
            tail = 0;  
        else tail++;  
    } else  
        throw new FullQueueException();  
}
```

Die **enqueue**-Operation

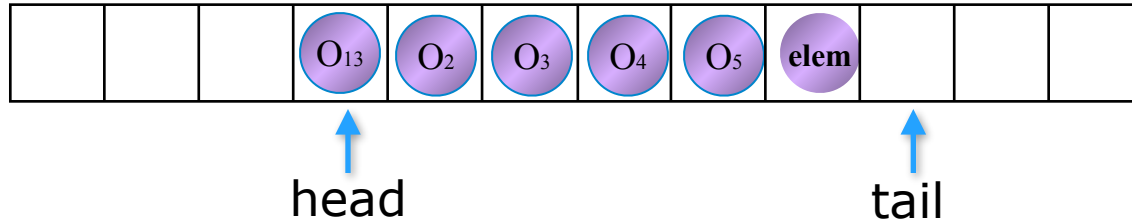


Hier wird
geprüft,
ob **tail** am
Ende des
Feldes ist



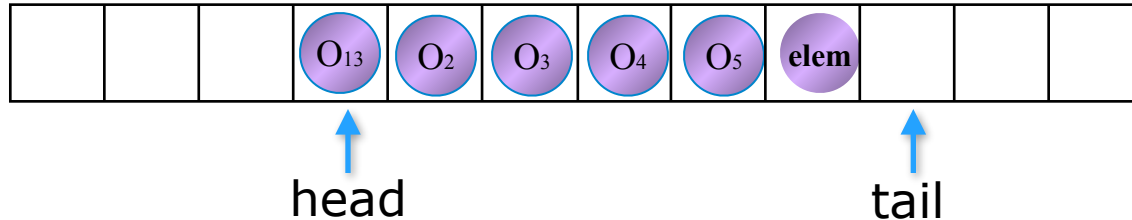
```
public void enqueue ( E elem ) throws FullQueueException {  
    if ( !full() ) {  
        queue[tail] = elem;  
        if ( tail == (queue.length-1) )  
            tail = 0;  
        else tail++;  
    } else  
        throw new FullQueueException();  
}
```

Die **enqueue**-Operation

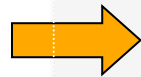


```
public void enqueue ( E elem ) throws FullQueueException {  
    if ( !full() ) {  
        queue[tail] = elem;  
        if ( tail == (queue.length-1) )  
            tail = 0;  
        else tail++;  
    } else  
        throw new FullQueueException();  
}
```

Die **dequeue**-Operation

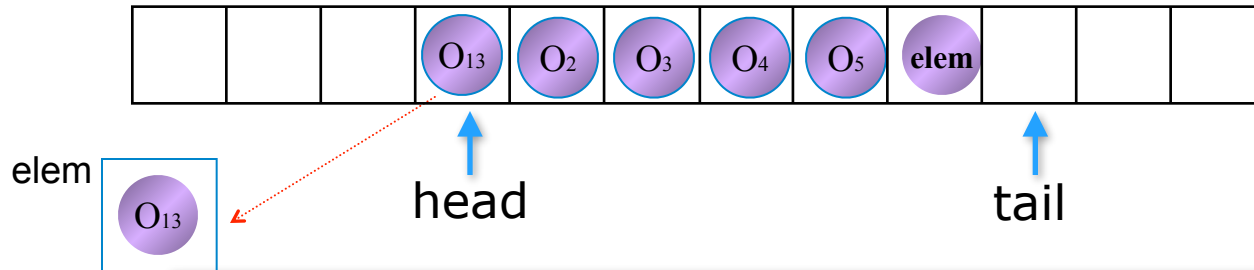


Wenn die
Warteschlange
nicht leer ist.



```
public E dequeue() throws EmptyQueueException {  
    if ( !empty() ) {  
        E elem = queue[head];  
        if ( head == (queue.length-1) )  
            head = 0;  
        else head++;  
        return elem;  
    } else  
        throw new EmptyQueueException();  
}
```

Die **dequeue**-Operation



```
public E dequeue() throws EmptyQueueException {
```

```
    if ( !empty() ) {
```



```
        E elem = queue[head];
```

```
        if ( head == (queue.length-1) )
```

```
            head = 0;
```

```
        else head++;
```

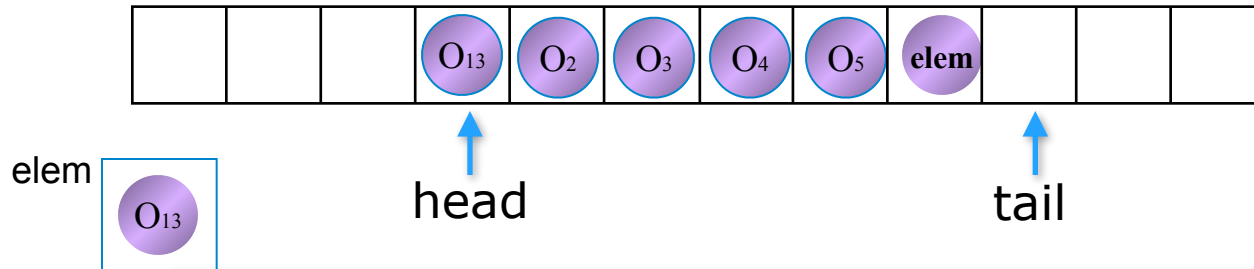
```
        return elem;
```

```
    } else
```

```
        throw new EmptyQueueException();
```

```
}
```

Die **dequeue**-Operation

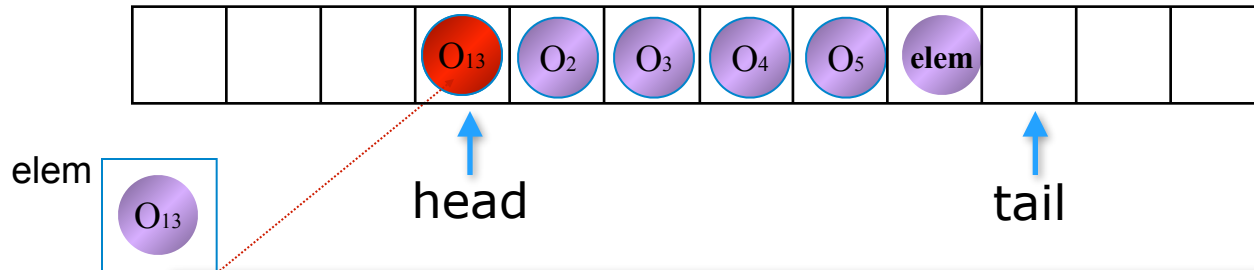


```
public E dequeue() throws EmptyQueueException {  
    if ( !empty() ) {  
        E elem = queue[head];  
        if ( head == (queue.length-1) )  
            head = 0;  
        else head++;  
        return elem;  
    } else  
        throw new EmptyQueueException();  
}
```

Wenn **head** am
Ende des
Feldes ist.



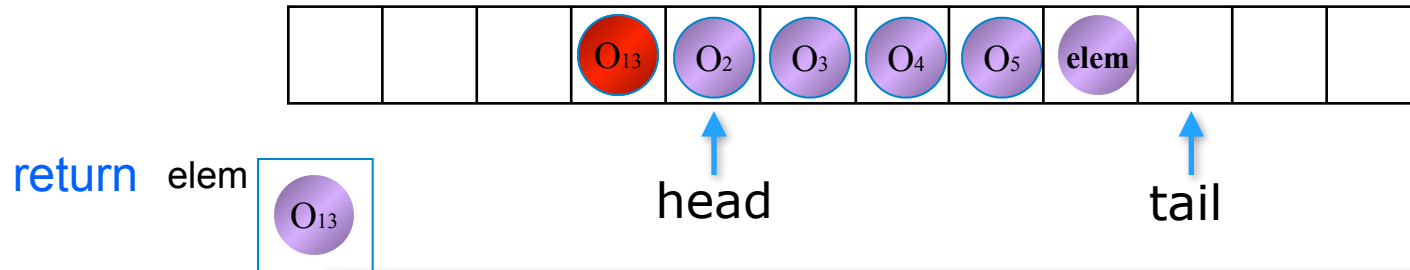
Die **dequeue**-Operation



Die verbliebene
Objekt-Referenz
wird später
überschrieben.

```
public E dequeue() throws EmptyQueueException {  
    if ( !empty() ) {  
        E elem = queue[head];  
        if ( head == (queue.length-1) )  
            head = 0;  
        else head++;  
        return elem;  
    } else  
        throw new EmptyQueueException();  
}
```

Die **dequeue**-Operation



```
public E dequeue() throws EmptyQueueException {  
    if ( !empty() ) {  
        E elem = queue[head];  
        if ( head == (queue.length-1) )  
            head = 0;  
        else head++;  
        return elem;  
    } else  
        throw new EmptyQueueException();  
}
```

Java-Klassen für Stapel und Warteschlangen

Vector



Stack

```
public class Stack<E> extends Vector <E> {  
    ...  
}
```

```
public interface Queue <E> extends Collection <E>
```

Einige Implementierungen sind:

AbstractQueue

PriorityBlockingQueue

ConcurrentLinkedQueue

DelayQueue

LinkedBlockingQueue

ArrayBlockingQueue

PriorityQueue

SynchronousQueue

LinkedList

Zusammenfassung

Stapel und Schlangen sind dynamische Datenstrukturen, doch die Implementierung mit Hilfe von Feldern hat die **Einschränkung**, dass die **maximal erreichbare Größe** vorher bekannt sein muss.

Eine mögliche **Lösung** dieses Problems sind „**Dynamische Arrays**“. Wenn ein Feld voll ist, wird zur Laufzeit ein neues erzeugt, das doppelt so groß ist, und alle Daten des alten Feldes werden auf das neue Feld kopiert. Das Ganze wird wiederholt, wenn das Feld wieder ausgefüllt ist.

Eine **zweite Lösung** ist, von Anfang an **echte dynamische Datenstrukturen** zu verwenden (wie **Listen**, **Bäume**, usw.).