Barry Linnert
Institut für Informatik

Freie Universität Berlin

Programmieren

Barry Linnert

Sommersemester 2020

Gliederung der heutigen Vorlesung

- Kurze Wiederholung
- Information Hiding
  - Abstrakte Datentypen vs. Datenstrukturen
- Java ADT Schnittstelle
- Stacks
  - Eigene Implementierung in Java
- Queues
  - Eigene Implementierung in Java
- Zusammenfassung

Freie Universität Berlin

Elementare Datenstrukturen

MOTIVATION

Prinzip Information Hiding

- Programme wurden größer und unübersichtlicher, daher erschien es sinnvoll, Daten und Operationen, die diese Daten manipulieren, als Einheit zu sehen und in Module auszulagern

- Das Prinzip des *Information Hiding*
  - Bedeutet, dass ein Teilsystem (z.B. Modul, Objekt) nichts von den Implementierungsentscheidungen eines anderen Teilsystems weiß
  - Es wird vermieden, dass ein Teilsystem von der Implementierung eines anderen Teilsystems abhängt

Datenstruktur = Menge + Operationen

- Datenstruktur (oder Datentyp) ist eine Menge (von Daten) mit einer Reihe von Zugriffsoperationen

- Axiome sind Methoden, solche Datentypen zu beschreiben, ohne auf die Natur der Elemente eingehen zu müssen

- Ein durch Axiome beschriebener Datentyp heißt Abstrakter Datentyp.

- Der Programmierer muss nur die Axiome zur korrekten Anwendung des Datentyps kennen. Der Implementierer muss genau diese Axiome garantieren (kann aber die konkrete Implementierung ändern).

Terminologie

- Ein **abstrakter Datentyp (ADT)** besteht aus einem Wertebereich (d.h. einer Menge von Objekten) und darauf definierten Operationen.

- Die Menge der Operationen bezeichnet man auch als Schnitt-stelle des Datentyps.

- Eine **Datenstruktur** ist eine Realisierung bzw. Implementierung eines ADT.

Verbergen der Implementierung eines abstrakten Datentyps

Meyer, B., "Object-Oriented Software Construction", 2nd ed., Prentice Hall, 1997

Elementare Datenstrukturen

JAVA ADT SCHNITTSTELLE

Collections

- **Collections** sind **Zusammenstellungen von Daten**, d.h. von **Objekten**.
- Zur Verwaltung dieser Objekte im Arbeitsspeicher werden bei der Programmierung Datenstrukturen (Strukturen von Objekten) eingesetzt.
- Die einfachste dieser Datenstrukturen ist das bekannte Array.
- Darüber hinaus kennt man in der Informatik jedoch noch eine ganze Reihe weiterer Datenstrukturen mit unterschiedlichen Eigenschaften.

Java Collection-API

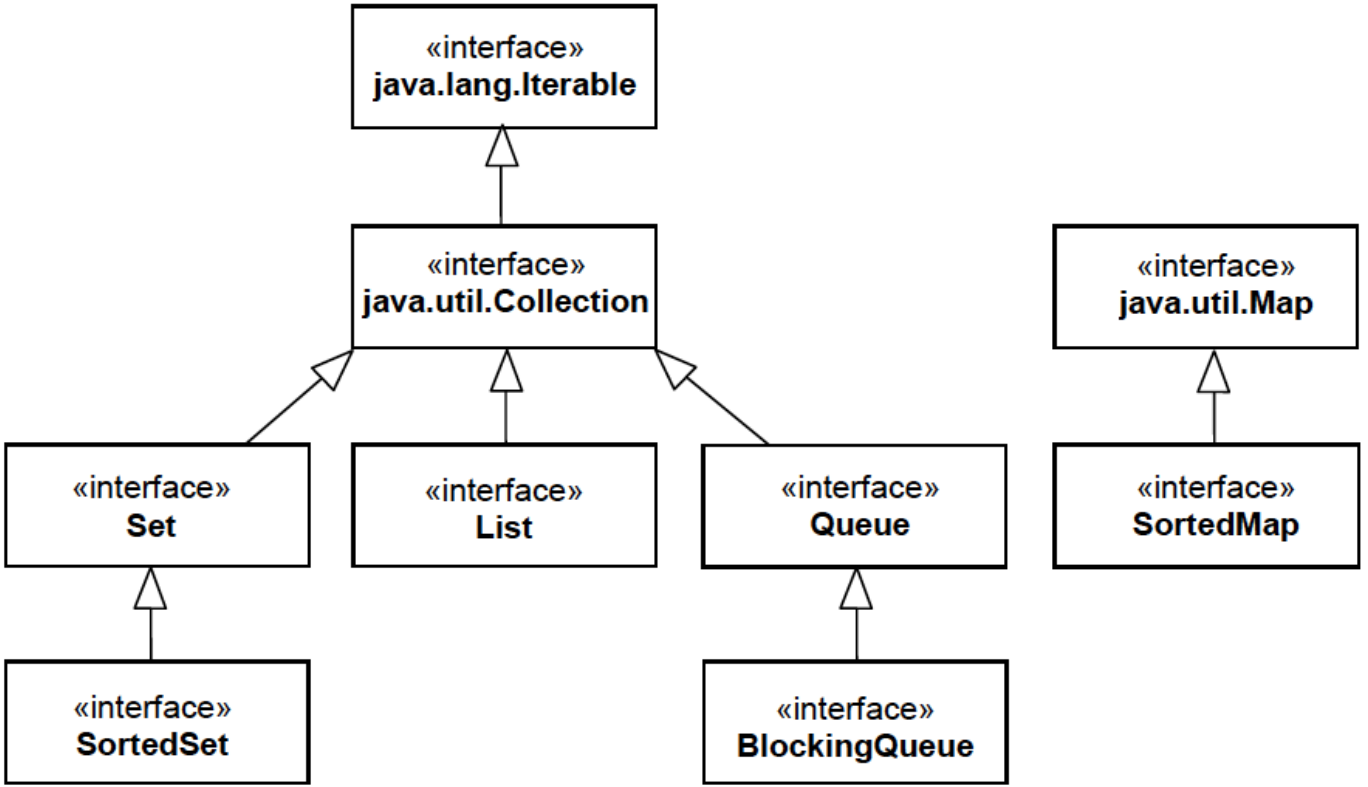
- Das Paket java.util enthält vier verschiedene Familien von Collections:
  - Listen (Lists)
  - Warteschlangen (Queues)
  - Mengen (Sets) und
  - Verzeichnisse (Maps)

- Zu jeder der Collection Familien (Listen, Warteschlangen, Mengen und Verzeichnisse) existiert eine Schnittstelle sowie eine abstrakte Basisklasse, welche diese Schnittstellen implementiert und als Basis für konkrete Collection-Klassen dient.

Collections und ihre funktionalen Eigenschaften

Interface	Implementierung	sequenziell	geordnet	wahlfrei	sortiert	Duplikate	Kommentar
Lists	ArrayList	X	X	X		X	
	LinkedList	X	X	X		X	
	Stack	X	X			X	
	Queue	X	X			X	
	PriorityQueue	X	X		X	X	
Sets	HashSet	X		(X)			wahlfreier Test
	TreeSet	X		(X)	X		wahlfreier Test
Hashes	HashMap	X		X		(X)	nur doppelte Werte
	TreeMap	X		X	X	(X)	nur doppelte Werte

Auswahl der Collection-Schnittstellen in java.util

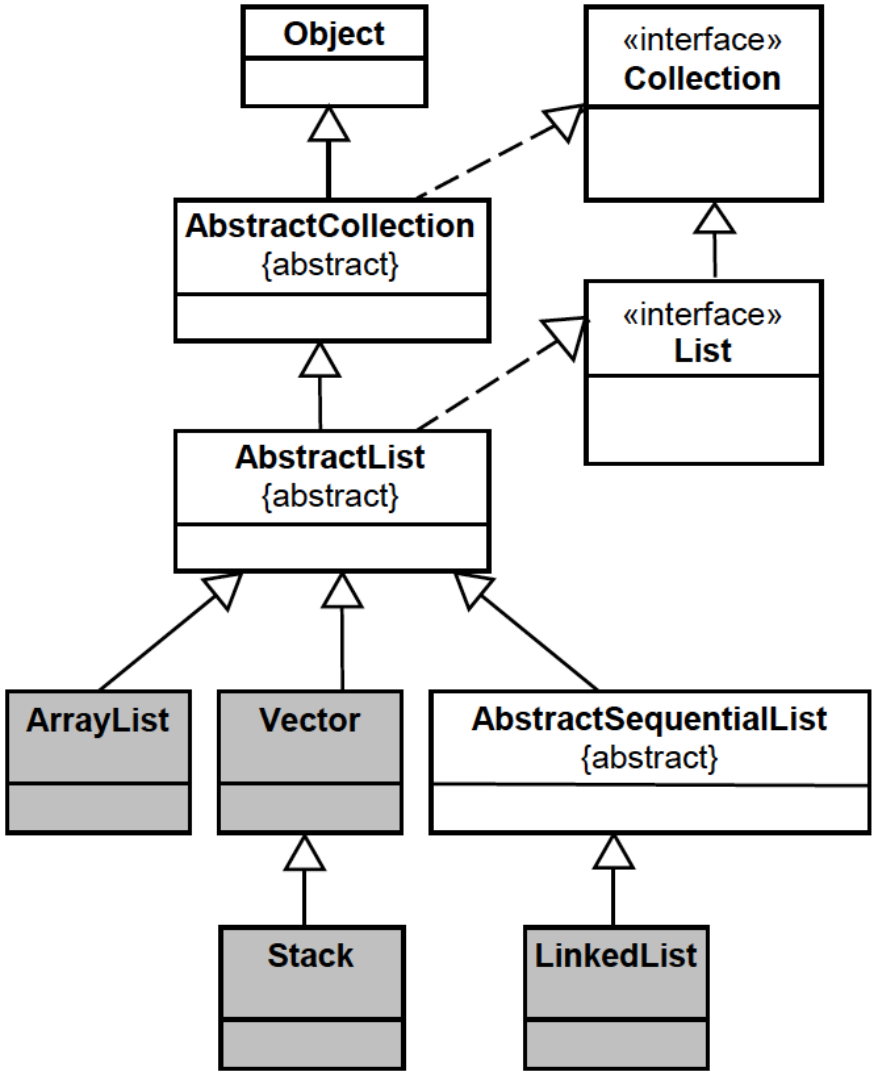


Konkrete Collection-Klassen im Paket java.util

		Organi-sation	Freiheit beim Zugriff	Zugriff	Duplikate zugelassen	Besondere Eigenschaften
Listen	ArrayList	geordnet	wahlfrei	wahlfrei über Index	ja	schnelles Lesen
	LinkedList	geordnet	wahlfrei	wahlfrei über Index	ja	schnelles Einfügen
	Vector	geordnet	wahlfrei	wahlfrei über Index	ja	synchronisiert
	Stack	geordnet	sequenziell	sequenziell (letztes Element)	ja	LIFO
Queues	LinkedList	geordnet	sequenziell	sequenziell, nächstes Element	ja	schnelles Ein-fügen (am Rand)
	PriorityQueue	sortiert	sequenziell	sequenziell, nächstes Element	ja	-
Sets	HashSet	ungeordnet	wahlfrei	ungeordnet, Test auf Existenz	nein	schnelles Lesen
	TreeSet	sortiert	wahlfrei	sortiert, Test auf Existenz	nein	-
Maps	HashMap	ungeordnet	wahlfrei	wahlfrei über Schlüssel	Schlüssel nein Werte ja	schnelles Lesen
	TreeMap	sortiert	wahlfrei	wahlfrei über Schlüssel	Schlüssel nein Werte ja	-

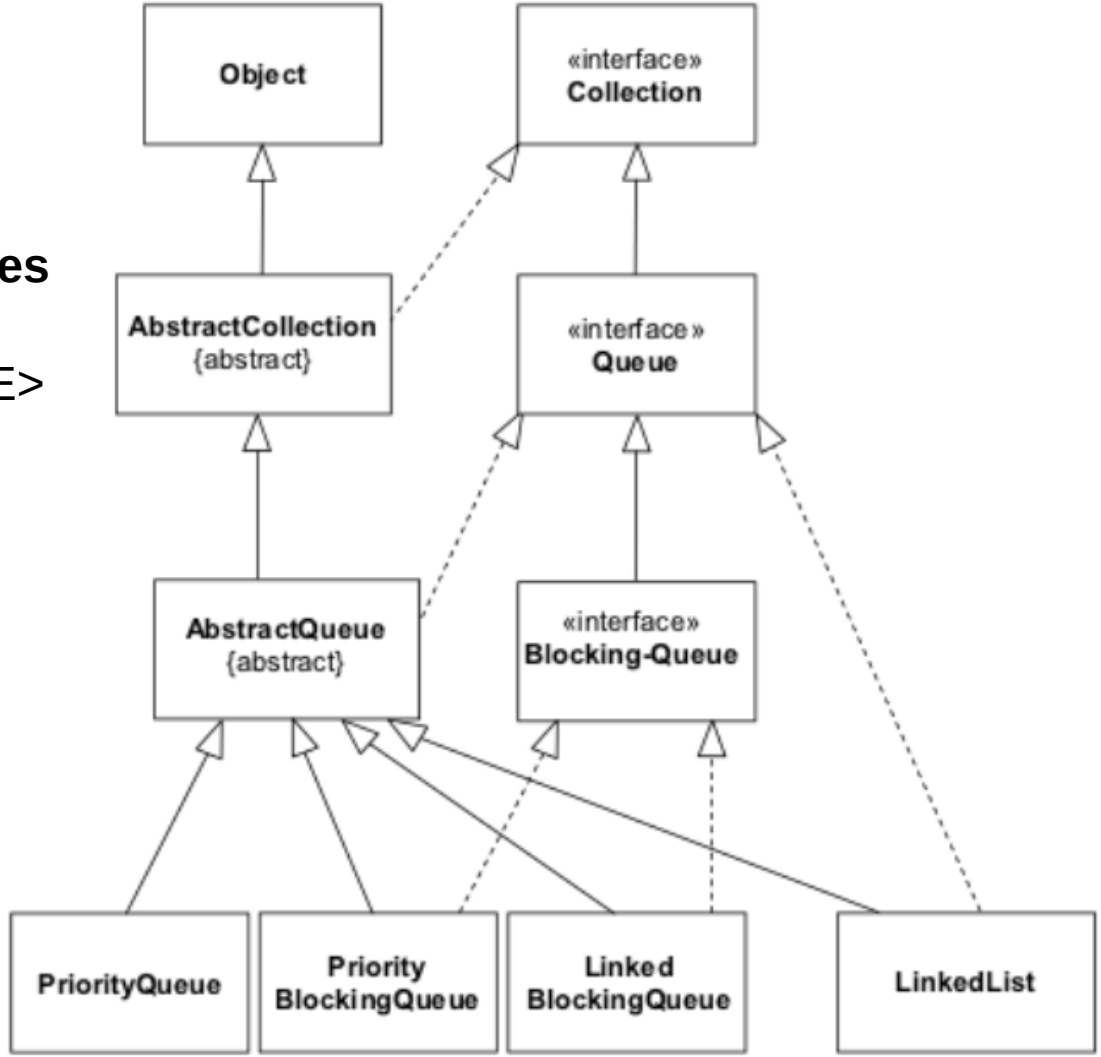
Listen

- Als Listen werden die Klassen bezeichnet, welche die Schnittstelle List<E> implementieren und das sind die Klassen
  - ArrayList<E>
  - Vector<E>
  - LinkedList<E> und
  - Stack<E>.



Warteschlangen

- Als **Warteschlangen** oder **Queues** werden die Klassen bezeichnet, welche die Schnittstelle Queue<E> implementieren und das sind die Klassen
  - PriorityQueue<E>
  - PriorityBlockingQueue<E>
  - LinkedBlockingQueue<E> und
  - LinkedList<E>.



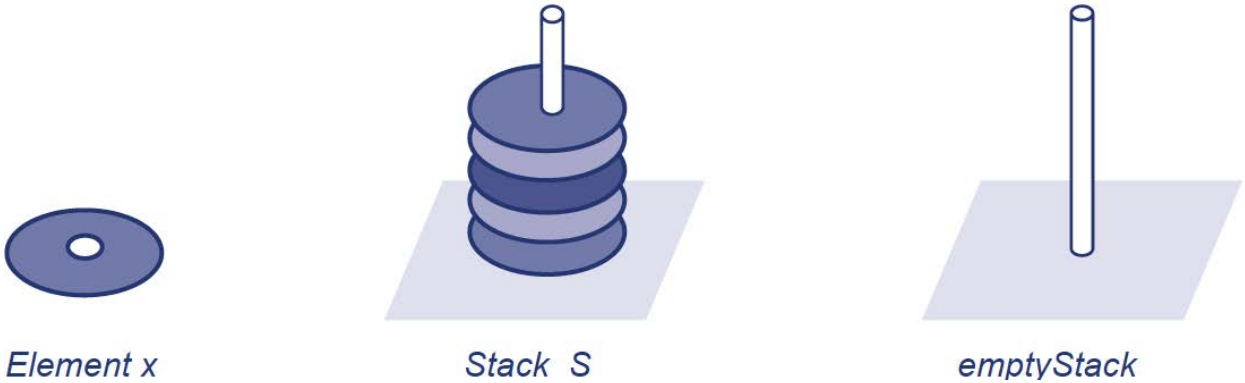
Freie Universität Berlin

Abstrakte Datenstrukturen

STACKS (STAPEL)

Einführung in den Stack

- Ein **Stack ist ein abstrakter Datentyp**, bei dem Elemente eingefügt und wieder entfernt werden können.
- Ein Stack dient als Behälter für Elemente und wird auch als Container oder Collection bezeichnet.
- Stacks arbeiten nach dem LIFO Prinzip, d.h. “Last-In-First-Out”.



Praktische Anwendungen eines Stacks

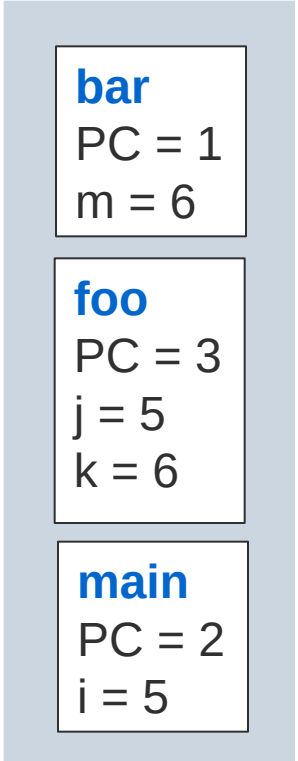
- Direkte Anwendungen
  - Historie der besuchten Seiten in einem Webbrowser
  - Aktivitäten in einem Texteditor rückgängig machen
  - Organisation der Reihenfolge der Methodenaufrufe in der Java Virtual Maschine (JVM)

- Indirekte Anwendungen
  - Hilfsdatenstruktur für Algorithmen
  - Bestandteil anderer Datenstrukturen

Methoden Stack in der JVM

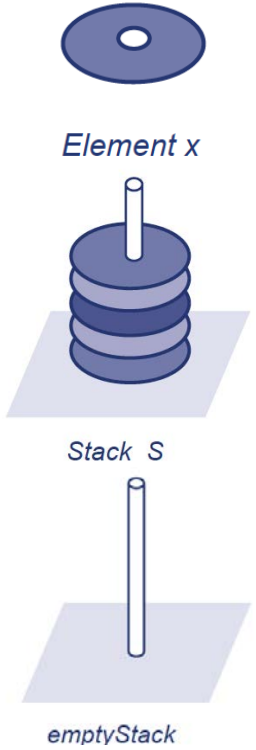
- Die Java Virtual Machine (JVM) überwacht die Abfolge der Methodenaufrufe mit einem Stapel
- Wenn eine Methode aufgerufen wird, packt die JVM die folgenden Informationen (Frame) auf den Stapel:
  - Lokale Variablen und Rückgabewert
  - Programmzähler, welcher verfolgt welche Anweisung gerade ausgeführt wird
- Wenn eine Methode abgearbeitet wurde, werden die Informationen (Frame) vom Stapel entfernt und die Kontrolle wird an die nächste obenauf liegende Methode übergeben

```
main() {
int i = 5;
foo(i);
}

foo(int j) {
int k;
k = j+1;
bar(k);
}

bar(int m) {
…
}
```

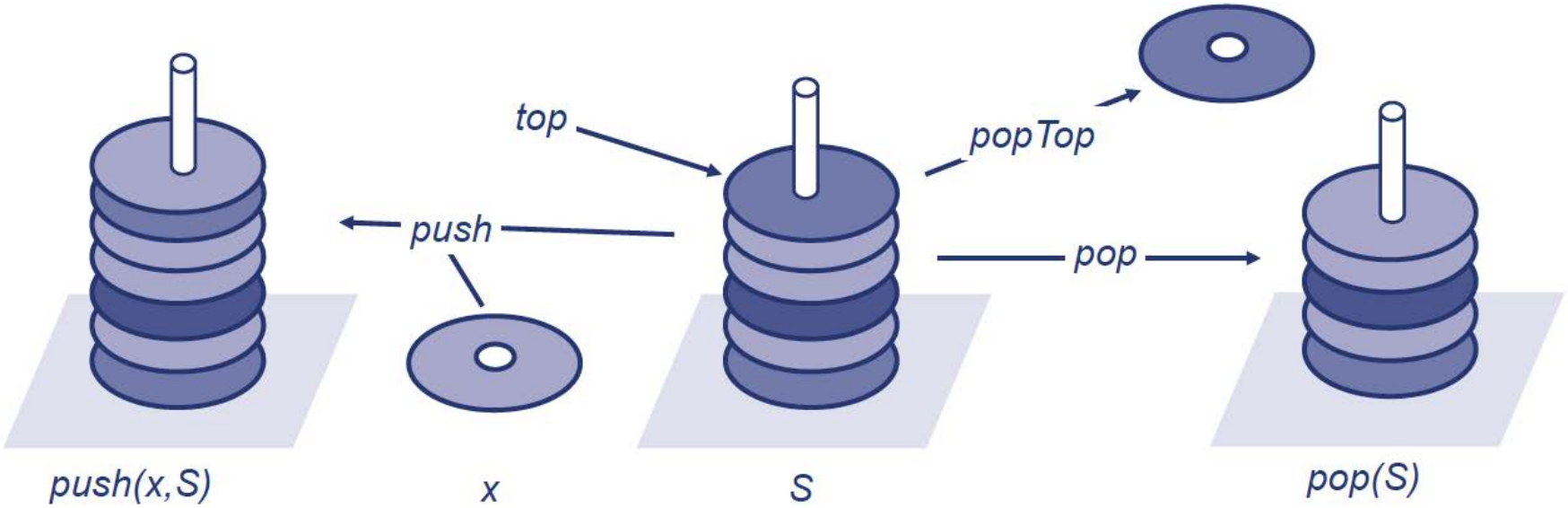


Stackoperationen

- Der einfachste Stapel ist der *leere Stapel*, d.h. *emptyStack*
- Die Prüfung des Stacks kann mittels eines Prädikats *isEmpty* erfolgen
- Grundlegende Stackoperationen sind:
  - push(x, s) legt ein Element x auf den Stack s,
  - top(s) liefert das zuletzt auf den Stack s gelegte Element,
  - pop(s) entfernt das zuletzt auf den Stack s gelegte Element.



Push und Pop auf dem Stack



Umsetzung von Stacks in Programmiersprachen

- Logische und funktionale Sprachen, wie z.B. Prolog, Haskell, Erlang oder LISP, besitzen eine eingebaute Datenstruktur *Liste*, die man als erweiterte Version der Datenstruktur Stack ansehen kann.

- Imperative Programmiersprachen haben üblicherweise keine eingebauten Stack- (oder Listen-) Datentypen, aber sie sind oft in Zusatzmodulen oder Paketen vorhanden.

- Java besitzt in dem Package *java.util* eine Klasse *Stack<E>*, wobei die Operation *top peek* genannt wird und die Operation *pop poptop* entspricht.

Stacks

EIGENE IMPLEMENTIERUNG IN JAVA

Implementierungsansatz

- “Normale” Arrays
  - Ein Array ist ein Objekt, das aus Komponenten (Elementen) zusammengesetzt ist, wobei jedes Element eines Arrays vom selben Datentyp sein muss.
  - Zunächst wird der Speicher zur Benutzung angefordert und festgelegt (auch Ende), dann kann beliebig auf die Elemente zugegriffen werden.

- "Dynamische Arrays"
  - Wenn ein Feld voll ist, wird zur Laufzeit ein neues erzeugt, das doppelt so groß ist.
  - Alle Daten des alten Feldes werden auf das neue Feld kopiert.
  - Das Ganze wird wiederholt, bis das Feld wieder ausgefüllt 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;
}
```

Wir erzeugen eine eine EmptyStackException bei dem Versuch, ein Element zu entfernen oder zu lesen (pop- und peek-Operationen), wenn der Stapel leer ist.

Methodenköpfe der Klasse Stack<E>

```
public boolean empty();
```

- Überprüft, ob der Stapel leer ist.

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

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

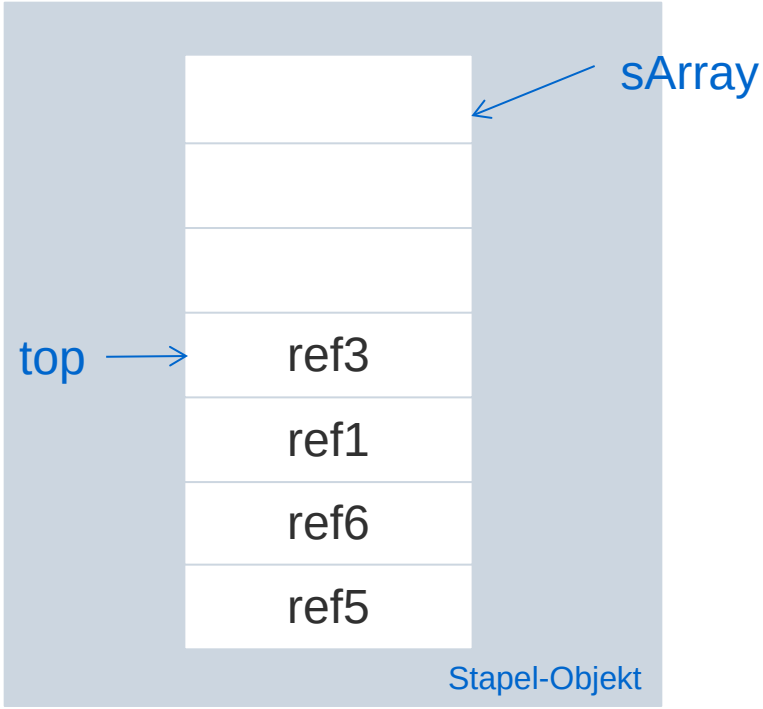
```
public E pop() throws EmptyStackException;
```

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

```
public 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.

Implementierung der Stapel-Schnittstelle



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

Implementierung der Stapel-Schnittstelle

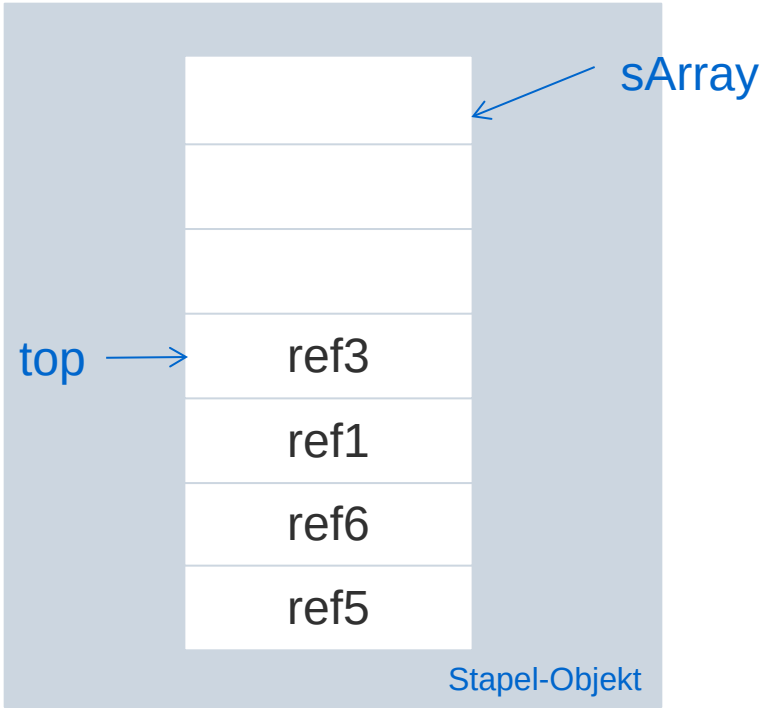
Im sArray werden die Stapelelemente gespeichert.

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

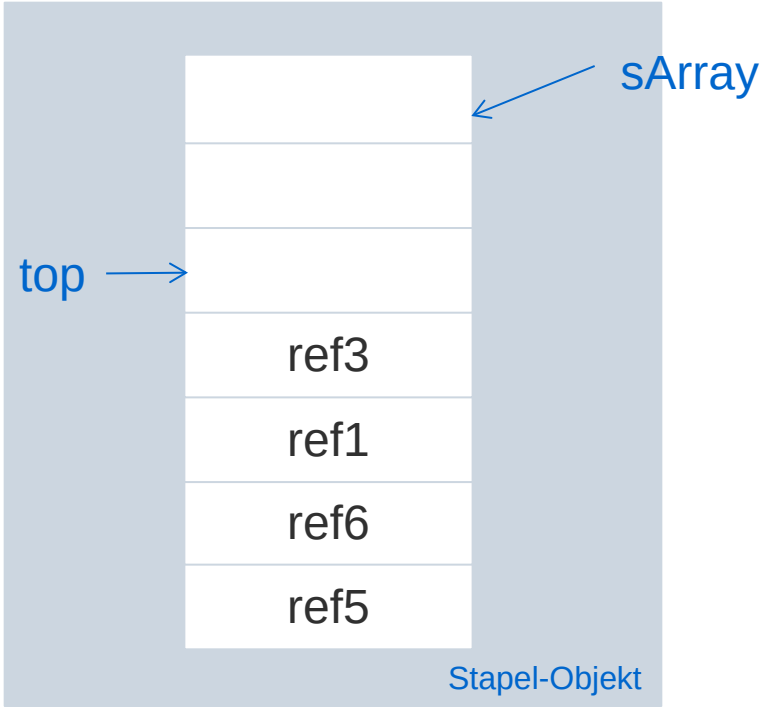
```
public class ArrayStapel<E> implements Stack<E>{
    private E[] sArray;
    private int top;

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

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

Die push-Operation



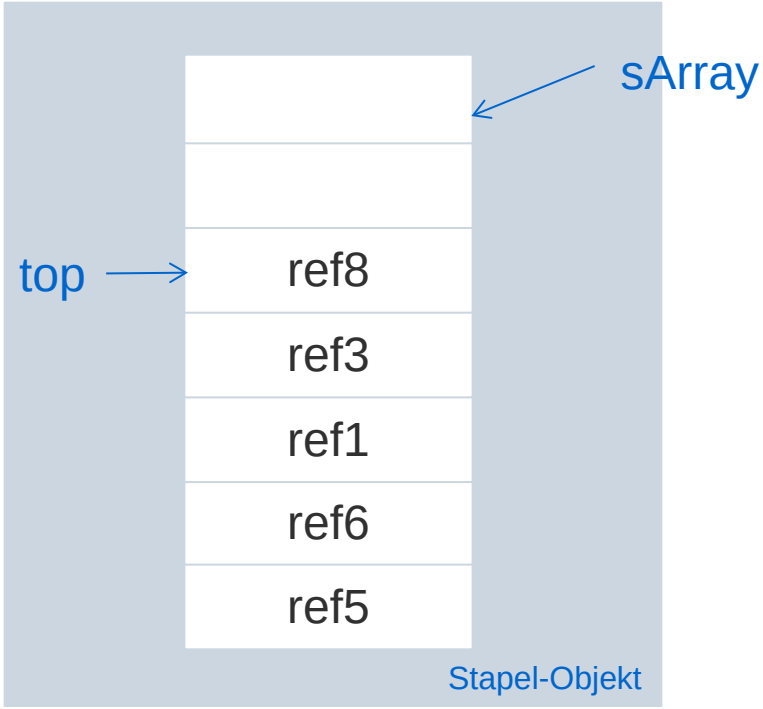

```
public void push(E elem) {

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

Die push-Operation



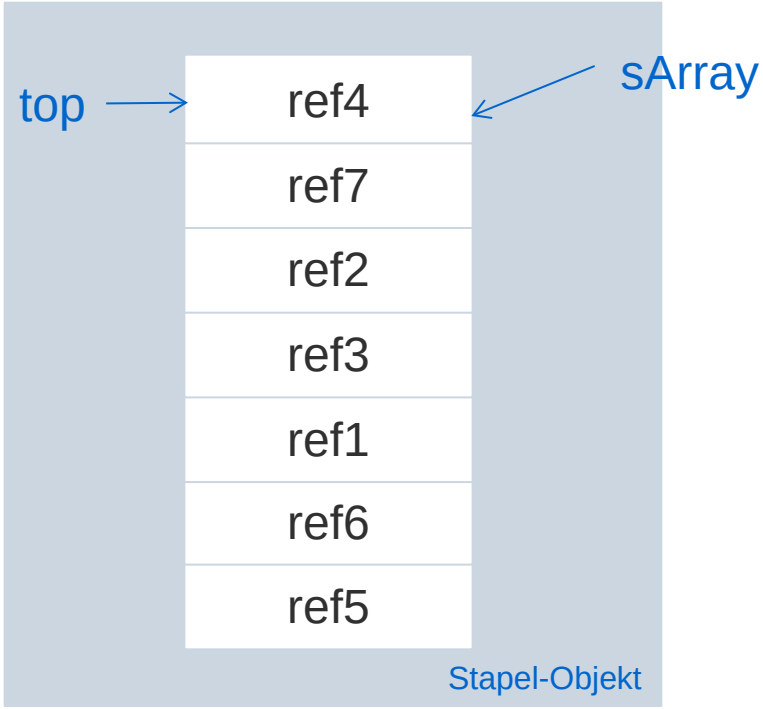

```
public void push(E elem) {

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

Die push-Operation



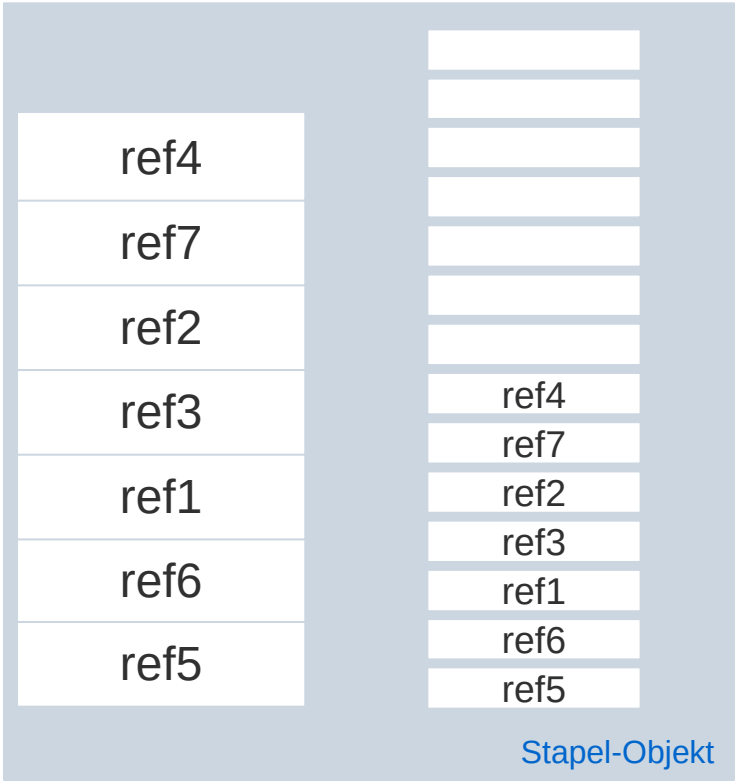

```
public void push(E elem) {

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

Die push-Operation



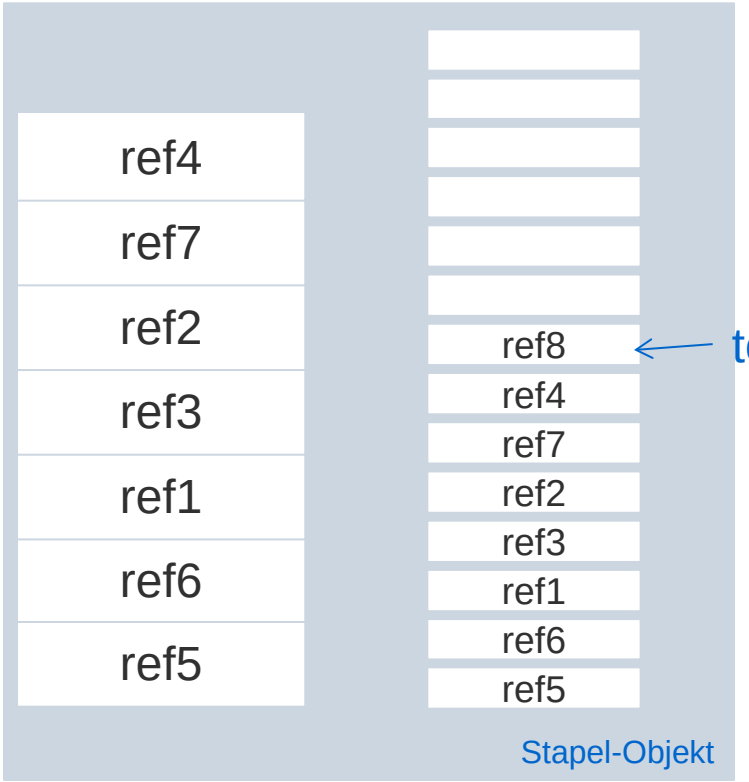

```
public void push(E elem) {

  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;
  }
}
```

Hilfmethoden

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

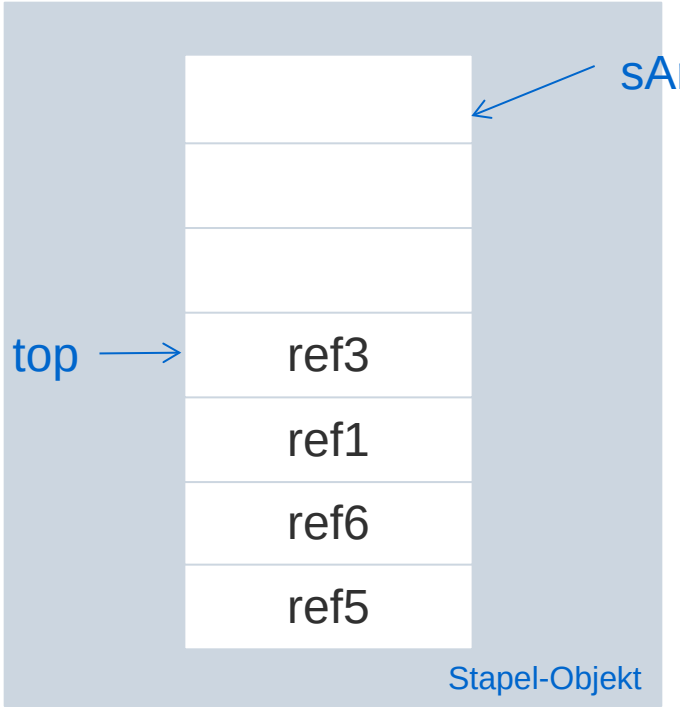
Die full-Hilfsmethode

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

Die resizeSArray-Hilfsmethode

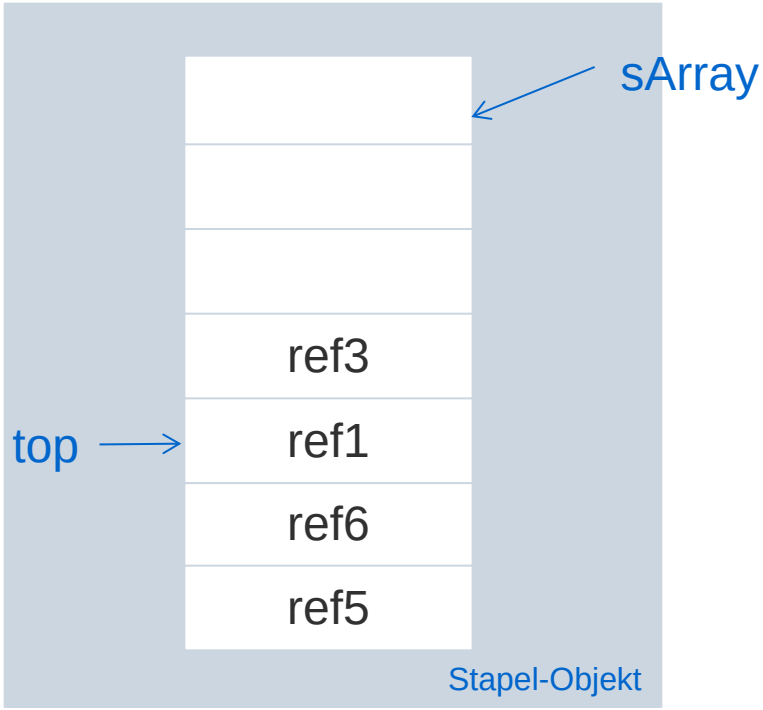
```
private void resizeSArray(){
   E[] temp = (E[]) new Object[sArray.length*2];
   for (int i=0; i<sArray.length; i++){
      temp[i] = sArray[i];}
   sArray = temp;}
```

Die pop-Operation



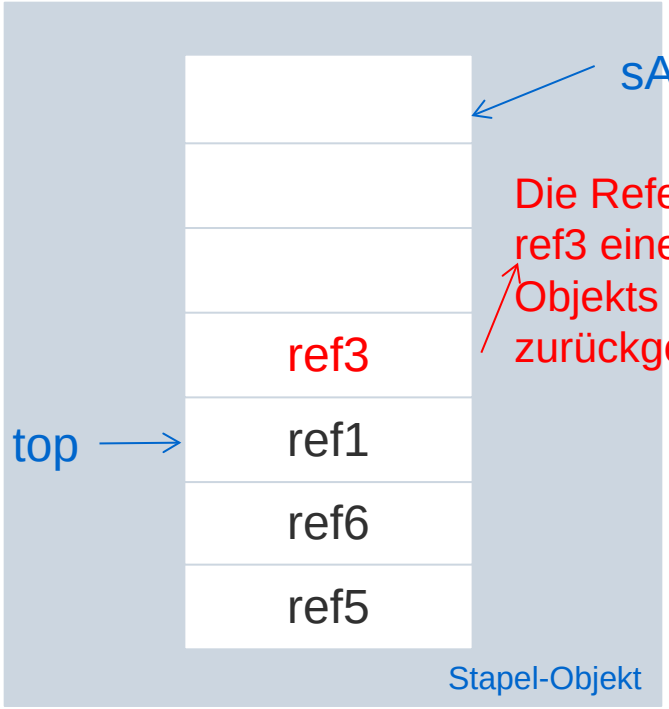

```
public E pop ()
      throws EmptyStackException {

   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();
   }
```

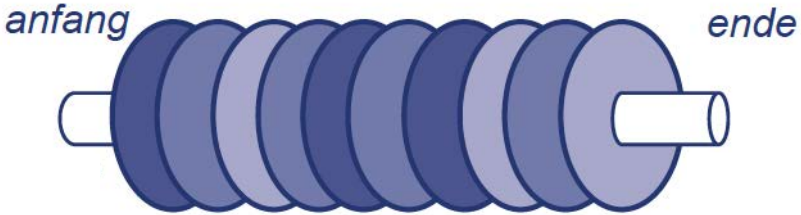
Elementare Datenstrukturen

QUEUES (WARTESCHLANGEN)

Einführung in die Warteschlange

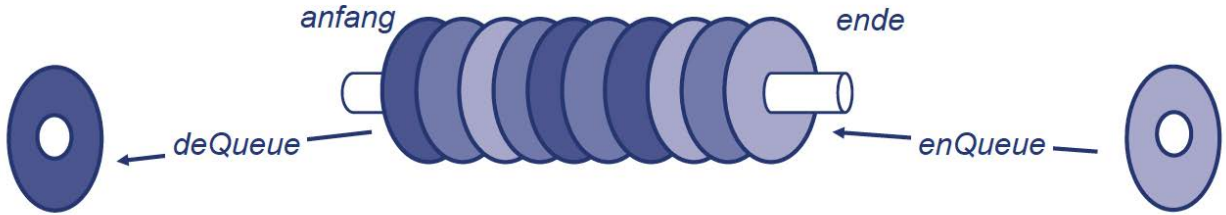
- Eine **Queue ist ein abstrakter Datentyp**, in den Elemente eingefügt und nur in einer bestimmten Reihenfolge wieder entnommen werden können.
- Eine Queue dient als Behälter für Elemente und wird auch als Container oder Collection bezeichnet.

- Stacks arbeiten nach dem FIFO Prinzip, d.h. “First-In-First-Out”.



Queue-Operationen

- Die einfachste Queue ist die *leere* Queue, d.h. empty*Queue*
- Die Prüfung der Queue kann mittels eines Prädikats *isEmpty oder isFull* erfolgen.
- Grundlegende Queue-Operationen sind:
  - enqueue ist die Einfüge-Operation.
  - dequeue ist die Lösch-Operation.



Praktische Anwendungen einer Warteschlange

- Direkte Anwendungen
  - “Reale” Wartelisten
  - Webserver bei der Auslieferung von Webseiten (IP-Packete) oder lokal im Webbrowser (Cache)
  - Zugriff auf geteilte Ressourcen (z. B. Drucker)
  - Multiprogrammierung

- Indirekte Anwendungen
  - Hilfsdatenstruktur für Algorithmen
  - Bestandteil anderer Datenstrukturen

Queues (Warteschlangen)

EIGENE IMPLEMENTIERUNG IN JAVA

Implementierung der Warteschlange

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.

Queue-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!

Hilfmethoden

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

Die empty-Hilfsmethode

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

Die full-Hilfsmethode

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

Hilfmethoden

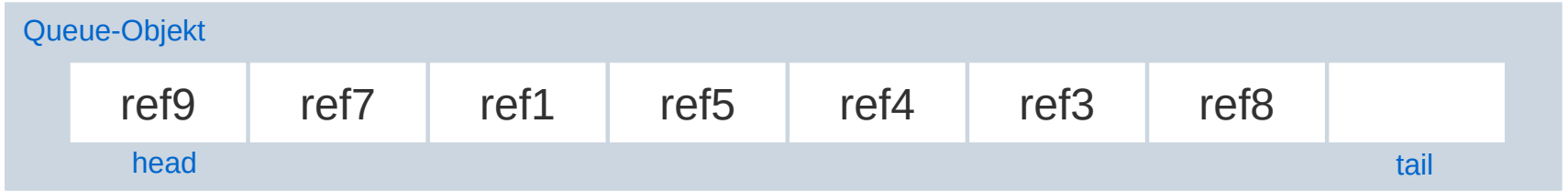
Der Stapel ist voll, wenn **top** gleich **stack.length-1** wird.

Die empty-Hilfsmethode

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

Die full-Hilfsmethode

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



Hilfmethoden

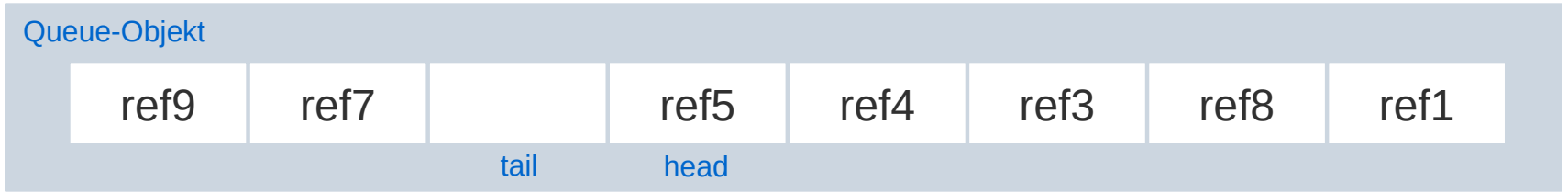
Der Stapel ist voll, wenn **top** gleich **stack.length-1** wird.

Die empty-Hilfsmethode

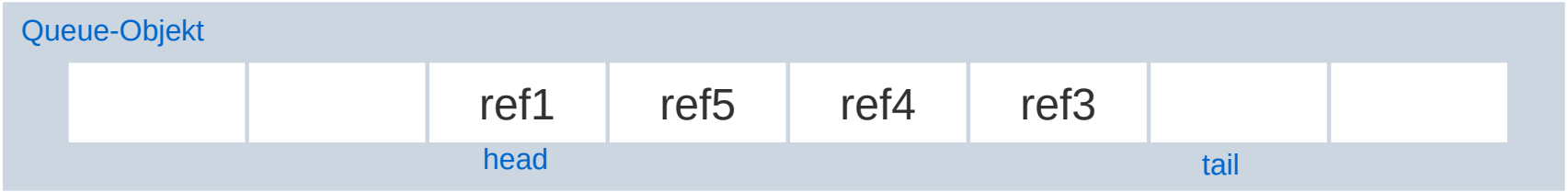
```
public boolean empty () {
    return head == tail;
}
```

Die full-Hilfsmethode

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

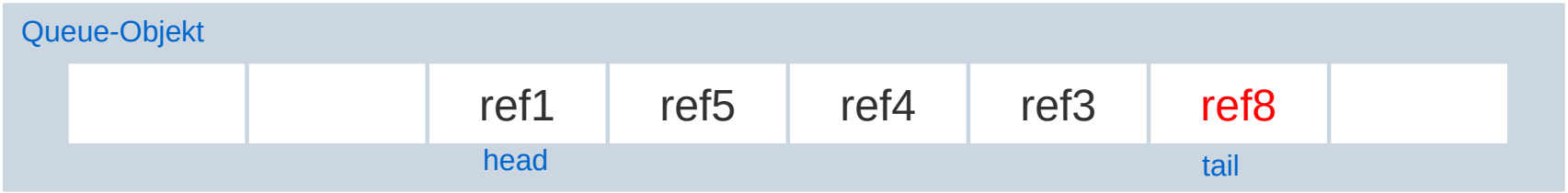


Die enqueue-Operation



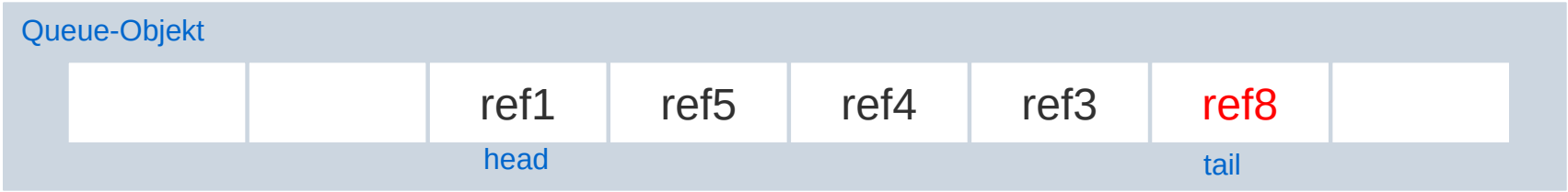

```
public void enqueue ( E elem ) throws FullQueueException {
   if ( !full() ) {
      queue[tail] = elem;
      if ( tail == (queue.length-1) )
          tail = 0;
      else tail++;
   } else
     throw new FullQueueException();
}
```

Die enqueue-Operation




```
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();
}
```

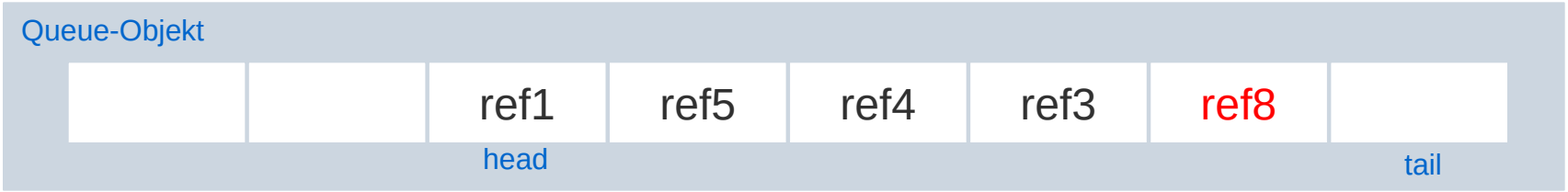
Die enqueue-Operation




```
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();
}
```

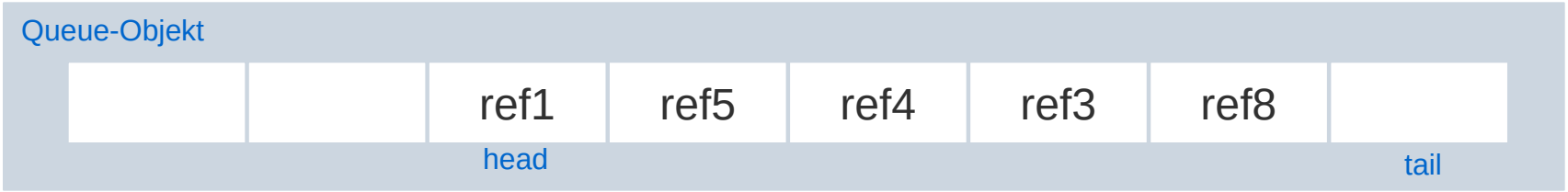
Hier wird geprüft, ob tail am Ende des Feldes ist.

Die enqueue-Operation



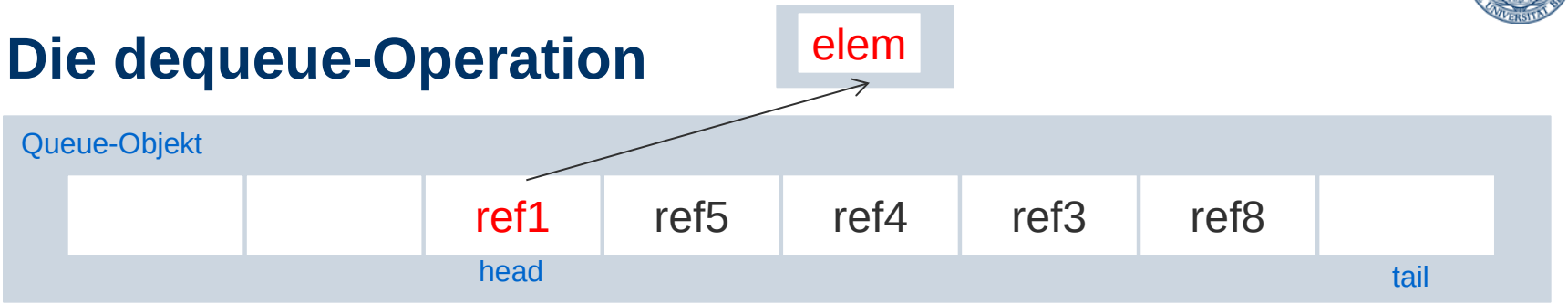

```
public void enqueue ( E elem ) throws FullQueueException {
   if ( !full() ) {
      queue[tail] = elem;
      if ( tail == (queue.length-1) )
          tail = 0;
      else tail++;
   } else
     throw new FullQueueException();
}
```

Die dequeue-Operation



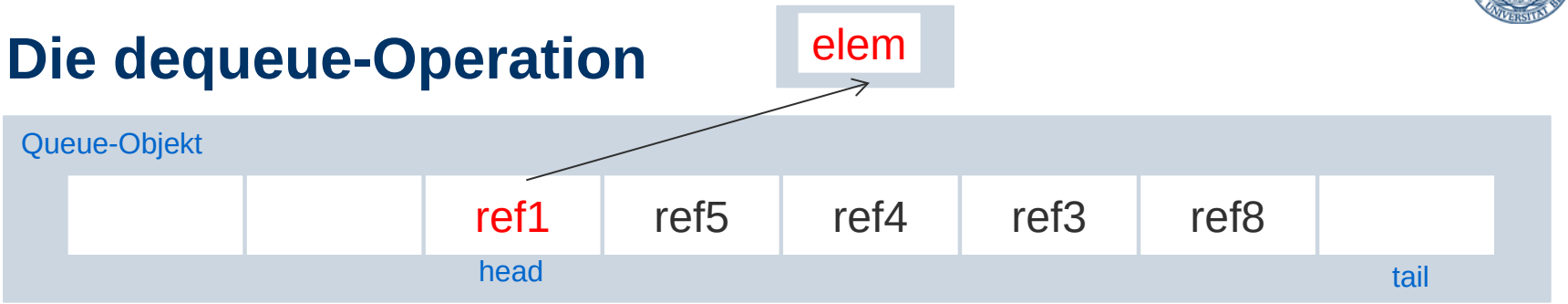

```
public E dequeue() throws EmptyQueueException {
   if ( !empty() ) {
      E elem = queue[head];
      if ( head == (queue.length-1) )
          head = 0;
      else head++;
      return elem; }
   else
      throw new EmptyQueueException(); }
```

Die dequeue-Operation



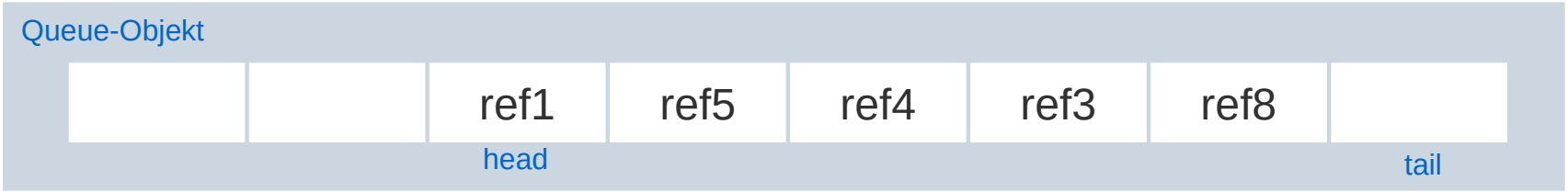

```
public E dequeue() throws EmptyQueueException {
   if ( !empty() ) {
      E elem = queue[head];
      if ( head == (queue.length-1) )
          head = 0;
      else head++;
      return elem; }
   else
      throw new EmptyQueueException(); }
```

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(); }
```

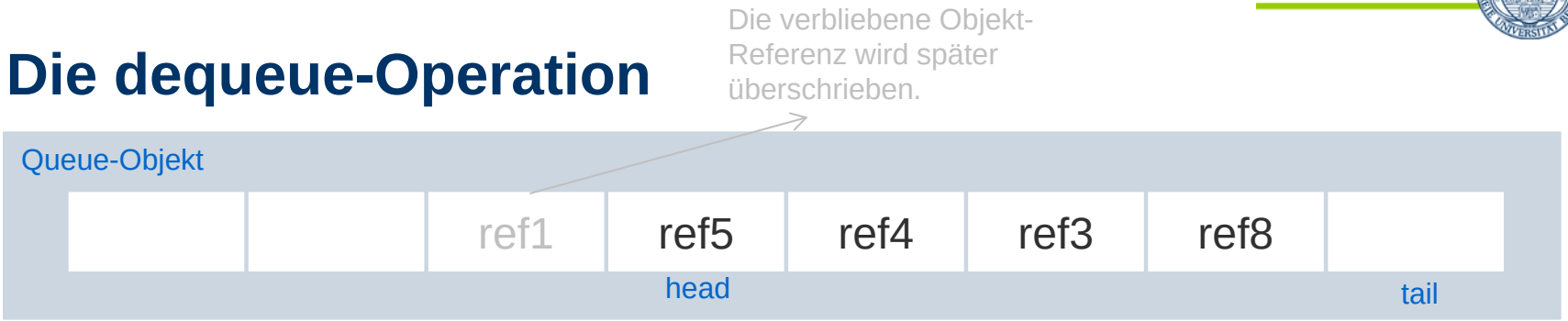
Die dequeue-Operation




```
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




```
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(); }
```

Freie Universität Berlin

Elementare Datenstrukturen

ZUSAMMENFASSUNG

Was haben wir besprochen

- Stapel (Stacks) und Warteschlangen (Queues) sind abstrakte Datentypen.

- Sie können mit Hilfe von Arrays implementiert werden. Das 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, usw.)...