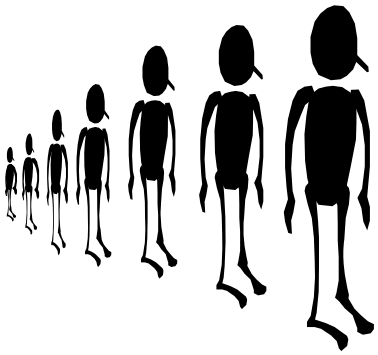
ADTs, Stacks und Queues



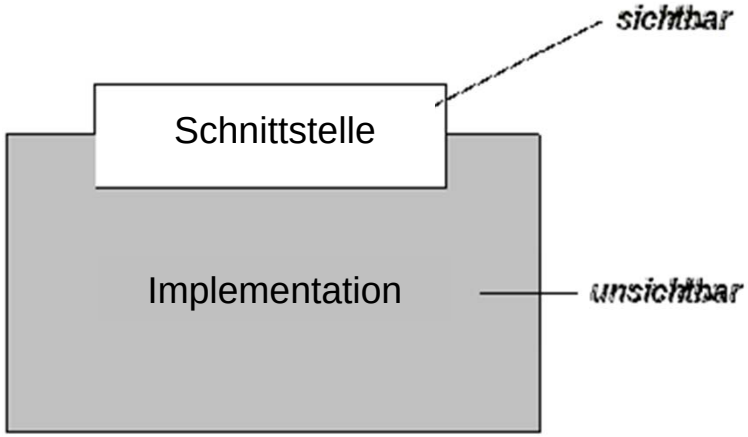
- Sie kennen das Konzept des ADTs
- Sie wissen, wie die Stack (Stapel) Datenstruktur funktioniert
- Sie können diese selber implementieren
- Sie kennen Anwendungen dieser Datenstruktur
- Sie wissen, wie die Queue (Warteschlange) Datenstruktur funktioniert
- Sie können diese implementieren

Abstrakte Datentypen (ADT)

Ein grundlegendes Konzept in der Informatik ist das ***Information Hiding***:

Nur gerade soviel wie für die Verwendung einer Klasse nötig ist, wird auch für andere sichtbar gemacht.

Jede Klasse besteht aus einer von aussen sichtbaren ***Schnittstelle***, und aus einer ausserhalb des Moduls unsichtbaren ***Implementation***:



Schnittstelle

Ein wesentliches Konzept von ADT's ist die

Definition einer Schnittstelle in Form von Zugriffsmethoden

Nur diese **Zugriffsmethoden** können die eigentlichen Daten des ADT's lesen oder verändern.

Dadurch ist sichergestellt, dass die **innere Logik** der Daten erhalten bleibt.

Implementation

Die Implementation eines ADT's kann **verändert werden**, ohne dass dies das verwendende Programm überhaupt merkt.

Man kann auch **verschiedene Implementationen** in Erwägung ziehen, die sich zum Beispiel bezüglich Speicherbedarf und Laufzeit unterscheiden.

ADTs in Java

- ADTs sind ein allgemeines Konzept
- in verschiedenen Sprachen unterschiedlich realisiert
- in Java durch Interfaces & Klassen
  - `interface`: Beschreibung der Schnittstelle (ohne Implementation)
  - `class`: Implementation der Schnittstelle

```
interface Stack<E> {                          // Die Schnittstelle
    void push(E obj);
    E pop();
}

class MyStack<E> implements Stack<E>  {    // Die Implementation
    void push(E obj) {
        // Implementation
    }
}
Stack<E> stack = new MyStack<E>();          // Die Instantiierung
```

- in Java kann eine Klasse mehrere Interfaces implementieren, aber nur von einer Klasse erben (keine Mehrfachvererbung).

Wert-Typen und Referenz-Typen

- einfache Wert-Typen in Java sind:
  - `byte, short, int, long`
  - `float, double`
  - `char`
  - `boolean`

  Beispiel
  - `int a, b;`
  - `a = 3; b = a;`
- eingebaute Referenz-Typen in Java sind
  - `Arrays`
  - `Object`
  - `String (speziell; verhält sich wie Wert-Typ)`

Wert-Typen und Referenz-Typen

- Objekt-Variablen sind lediglich Referenzen (Zeiger) auf Objekte
- `int[] a= {3,1,2};`
- `int[] b;`

- am Anfang zeigen diese nirgendwo hin: `null`

```
int[] a,b;
a[0] = 3; // Fehler
```

- können mittels der Zuweisung gesetzt,

```
a = new int[3];
b = a;
```

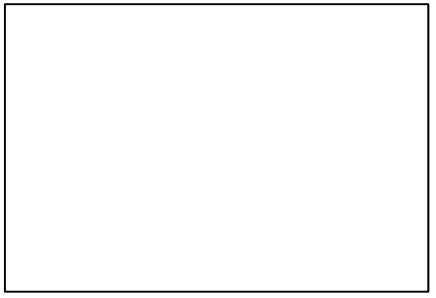
- oder wieder zu null gesetzt werden.

```
a = null;
```

Testfragen

Was wird im folgenden Programm ausgegeben?

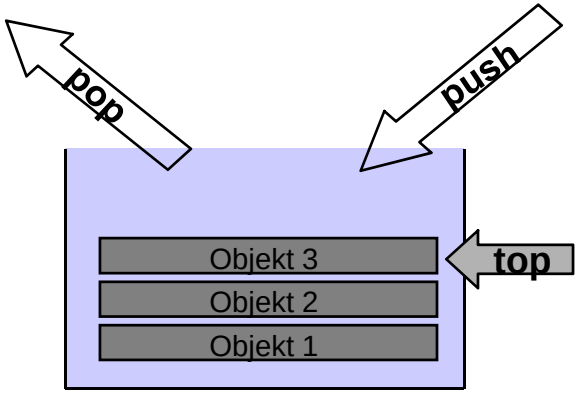
```
class C {
   C(int i) {this.i = i;}
   int i;
}
public class Test{
   static void f (C c) {c.i=25;}
   static void f (String s) {s = s + " du da";}
   static void f (Integer i) {i = 25;}
   public static void main (String[] args ){
      int[] a = {3,1,2};
      int[] b;
      b = a;
      b[0] = -1;
      C c = new C(5);
      String s = "Hallo";
      Integer i = 1;
      f(c); f(s); f(i);
      System.out.println("a[0] = " + a[0]);
      System.out.println("c.i = " + c.i);
      System.out.println("s = " + s);
      System.out.println("i = " + i);
   }
}
```



Stacks

Der Stack ist eine Datenstruktur, die Objekte als Stapel speichert:

- neue Objekte können nur oben auf den Stapel gelegt werden.
- auch das Entfernen von Objekten vom Stapel ist nur oben möglich.



Der Stack ist eine
LIFO (last in first out)
Datenstruktur

Minimale Operationen

Funktionskopf	Beschreibung
`void push (E x)`	Legt x auf den Stack
`E pop ()`	Entfernt das oberste Element und gibt es als Rückgabewert zurück
`boolean isEmpty()`	Gibt true zurück, falls der Stapel leer

Zusätzliche Operationen

Funktionskopf	Beschreibung
`E peek ()`	Gibt das oberste Element als Rückgabewert zurück, ohne es zu entfernen
`void clear ()`	Leert den ganzen Stack
`boolean isFull()`	Gibt true zurück, falls der Stack voll ist

Stack ADT Interface

```
/**
  Interface für Abstrakten Datentyp (ADT) Stack
*/
public interface Stack<E> {
  /**
    Legt eine neues Objekt auf den Stack, falls noch nicht voll.
    @param x ist das Objekt, das dazugelegt wird.
   */
  public void push (E x) throws StackOverflowError;

  /**
    Entfernt das oberste und damit das zuletzt eingefügte Objekt.
    Ist der Stack leer, wird null zurückgegeben.
    @return Gibt das oberste Objekt zurück oder null, falls leer.
  */
  public E pop ();

  /**
    Testet, ob der Stack leer ist.
    @return Gibt true zurück, falls der Stack leer ist.
  */
  public boolean isEmpty();

  /**
    Gibt das oberste Objekt zurück, ohne es zu entfernen.
    Ist der Stack leer, wird null zurückgegeben.
    @return Gibt das oberste Objekt zurück oder null, falls leer.
  */
  public E peek ();

  /**
    Entfernt alle Objekte vom Stack. Ein Aufruf von isEmpty()
    ergibt nachher mit Sicherheit true.
  */
  public void clear ();

  /**
    Testet, ob der Stack voll ist.
    @return Gibt true zurück, falls der Stack voll ist.
  */
  public boolean isFull();
}}
```

Array-Implementation des Stack

```
public class StackArray<E>
    implements Stack<E> {

  public StackArray(int
    capacity){
    data = (E[]) new
      Object[capacity];
    top = -1;
  }

  public void push(E x) throws
    StackOverflowError {
    if (isFull()){
      throw new
          StackOverflowError ();
    }
    top++;
    data[top] = x;
  }

  public E pop() {
    if (isEmpty()){
      return null;
    }
    E topItem = data [top];
    data [top] = null;
    top--;
    return topItem;
  }
```

```
  public boolean isEmpty() {
    return (top < 0);
  }

  public E peek() {
    if (isEmpty()){
      return null;
    }
    return data [top];
  }

  public void clear() {
    int i;
    for (i=0;i<data.length;i++){
      data [i] = null;
    };
    top = -1;
  }

  public boolean isFull() {
    return (top == data.length-1);
  }

  private E[] data;
  private int top;

}
```

Die Array Implementation nutzt einen Array von Objekten als eigentlichen Stapel.

Der *top* Zeiger (zeigt auf das zuletzt eingefügte Element) wird als int realisiert.
Ein Wert von –1 bedeutet, dass der Stack leer ist.

Der Konstruktor erwartet die maximale Anzahl Elemente, die der Stack aufnehmen kann.

Frage: Was sind die Vor- & Nachteile dieser Implementation?

Listen

Abstrakter Datentyp der eine Liste von Objekten verwaltet.

Implementation muss nicht interessieren (kommt später).

Liste ist eine der (wenn nicht die) grundlegenden Datenstrukturen in der Informatik.

Wir können mit Hilfe der Liste einen Stack implementieren.

Definiert in **`java.util.List`**
implementiert in:
`java.util.LinkedList`

Minimale Operationen	Beschreibung
`boolean add (E x)`	Fügt x am Ende der Liste an
`void add (int i, E x)`	Fügt x an der Stelle i in die Liste ein
`E get(int i)`	Gibt das Element an der Stelle i zurück
`E get(0)`	Gibt erstes Element zurück
`E get(size()-1)`	Gibt letztes Element zurück
`E remove(int i)`	Entfernt das Element an der Stelle i und gibt es als Rückgabewert zurück
`E remove(0)`	Entfernt das erste Element
`E remove(size()-1)`	Entfernt das letzte Element
`int size()`	Gibt Anzahl Elemente zurück
`boolean isEmpty()`	Gibt true zurück, falls die Liste leer

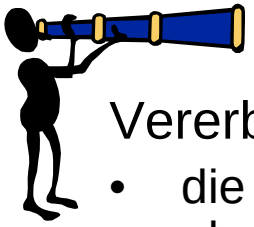
List-Implementation des Stack

```
import java.util.List;
import java.util.LinkedList;

/** Implementation des
    Abstrakten Datentyp (ADT)
    Stack mit Hilfe der
    vordefinierten Klasse
    java.util.LinkedList
*/

  public class
    StackLinkedList<E>
     implements Stack<E> {

  public StackLinkedList() {
    list = new
      LinkedList<E>();
  }

  public void push(E x) {
    list.add(0,x);
  }

  public E pop() {
    if (isEmpty())
      return null;
    return list.remove(0);
  }

  public boolean isEmpty(){
    return list.isEmpty();
  }

  public E peek() {
    if (isEmpty())
      return null;
    return list.get(0);
  }

  public void clear() {
    list.clear();
  }

  public boolean isFull() {
    return false;
  }

  private List<E> list;
}
```

`java.util.LinkedList` verwaltet eine Liste von Objekten und stellt entsprechende Methoden zur Verfügung.

list als **Attribut** der Klasse `StackLinkedList`

Funktion wird "delegiert"



Vererbung vs. Delegation

Vererbung

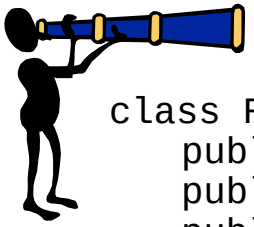
- die erbende Klasse (Subklasse, abgeleitete Klasse) ist eine echte **Erweiterung**
  - z.B. Konto -> Konto mit Zusatzfunktionen
- generell: Vererbung nur verwenden, wenn man sagen kann:
  - "erbende Klasse" gehört zu der Menge der "geerbten Klasse"
    - z.B. LinkedList gehört zu der Menge der Listen
    - Stack gehört aber **nicht** zu den Listen
  - Test: machen alle Methoden die geerbt werden einen Sinn.

Delegation

- die andere Klasse wird lediglich verwendet; Teile der Verarbeitung werden **delegiert.**
- Implementation: die (umschliessende) Klasse hat ein privates Attribut, das auf die eingebettete Klasse zeigt.

```
public class Stack {
  private List list =
      new LinkedList();
  ...
}
```

```
public class StackLinkedList
  extends LinkedList
     implements Stack {..}
```

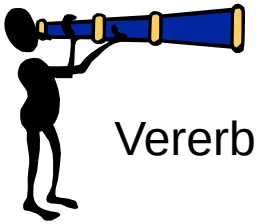


Vererbung: Schlechtes Beispiel

```
class Rechteck {
   public Rechteck (double breite, double hoehe) {b=breite; h=hoehe;}
   public double getBreite () {return (b);}
   public double getHoehe () {return (h);}
   public void setBreite (double breite) {b = breite;}
   public void setHoehe (double hoehe) {h = hoehe;}
   private double h, b;
};

class Quadrat extends Rechteck {
   public Quadrat (double seite) {super (seite, seite);}
   public void setBreite (double breite) {
         super.setBreite (breite);
         super.setHoehe (breite);
   }
   public void setHoehe (double hoehe) {
         super.setBreite (hoehe);
         super.setHoehe (hoehe);
   }
};

class XY {
   public void setSeitenverhaeltnis (Rechteck r, double verhaeltnis) {
         double breite = r.getHoehe() * verhaeltnis;
         r.setBreite (breite);
   }
}
```

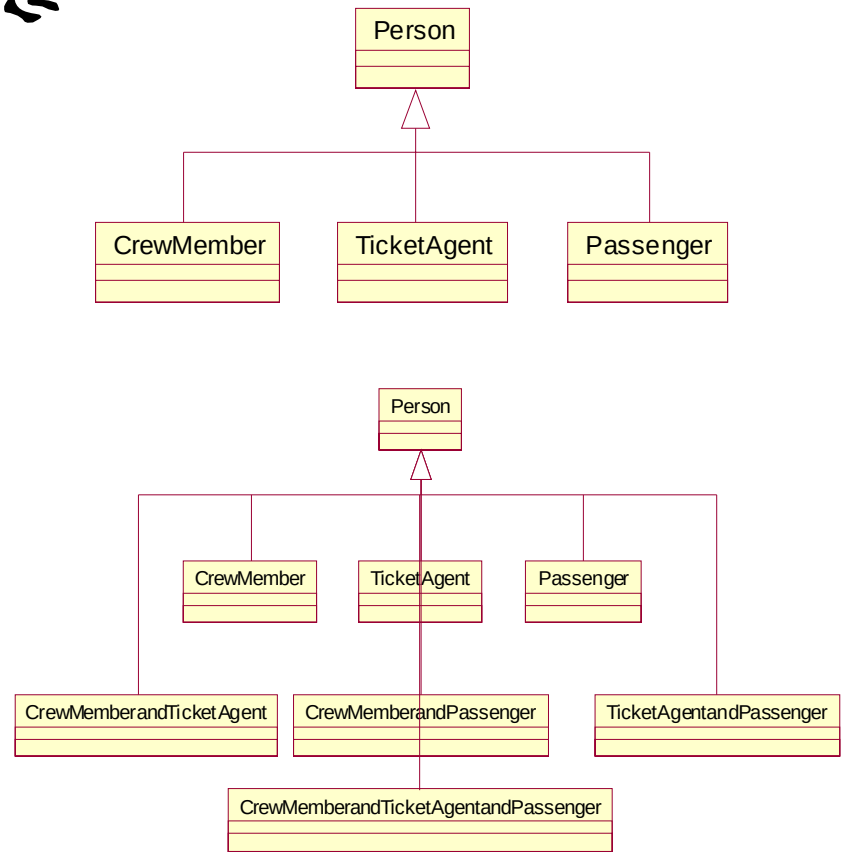
Was passiert, wenn wir hier ein Quadrat übergeben?

Jeder Funktion, die ein Objekt der Basisklasse erwartet, soll auch ein Objekt einer davon abgeleiteten Klasse übergeben werden können, ohne dass sich die Semantik der Funktion ändert

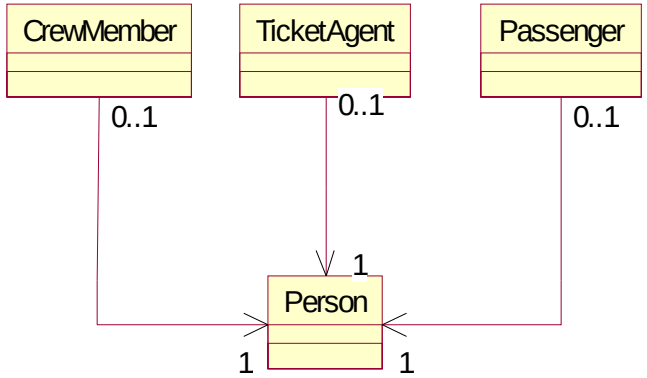


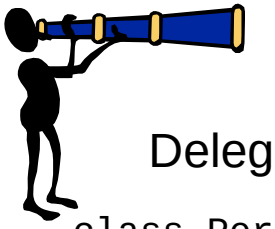
Vererbung vs. Delegation

Vererbung



Delegation





Vererbung vs. Delegation

Delegation

```
class Person {
   private String name;
   ...
   public Person (String n) {name = n;}
   String getName() {return name;}
};

class Passenger {
   private Person thePerson;
   private int seatNo;
   public Passenger (Person p) {
      thePerson = p;
   }
   public void setSeatNo(int no) {
       seatNo = no;
   };
   public String getName() {
       return thePerson.getName();
   }
};
```

```
Person sepp =
   new Person(“Sepp”);

Passenger aPassenger =
   new Passenger(sepp);

TicketAgent anAgent =
   new TicketAgent(sepp);
```

Wo werden Stacks angewendet?

Methoden-Parameter und Rücksprungadressen

Beim Sprung ins Unterprogramm wird der Programmzähler als erstes auf den Stack gelegt, beim Return wird zuletzt dieser Eintrag vom Stack geholt und an der Aufrufstelle weiter gefahren.

Auch Parameter und die lokalen Variablen werden auf dem Stack abgelegt. Damit können Methoden reentrant genutzt werden (Rekursion).

Auswerten von Postfix-Ausdrücken (HP Taschenrechner)

Die Auswertung eines Ausdrucks in Postfix-Notation läuft folgendermassen ab :

- Zahlen werden direkt auf den Stack gelegt
- Operatoren werden sofort ausgewertet, dazu werden 2 Elemente vom Stack geholt
- Das Resultat wird auf den Stack gelegt
- Am Ende enthält der Stack das Schlussresultat
- z.B. (3 + 4) * 2 -> 3 4 + 2 *

Test auf korrekte Klammersetzung

Bei Programmen und arithmetischen Ausdrücken kommen geschachtelte Klammerungen vor. Ein Test, ob diese auch korrekt paarweise gesetzt sind, kann mit Hilfe eines Stacks sehr einfach realisiert werden. Wie?

Beispiel: Auswerten von Postfix-Ausdrücken

Die Auswertung eines Ausdrucks in Postfix-Notation läuft folgendermassen ab:

- Zahlen werden direkt auf den Stack gelegt
- Operatoren werden sofort ausgewertet, dazu werden 2 Elemente vom Stack geholt
- Das Resultat wird auf den Stack gelegt
- Am Ende enthält der Stack das Schlussresultat
- 6* (5 + (2 + 3) * 8 + 3)

Ausdruck	**Aktion**	**Stack**
6 5 2 3 + 8 * + 3 + *	push von 6,5,2 und 3	3 2 5 6
6 5 2 3 **+** 8 * + 3 + *	+ auswerten, 3 + 2 ergibt 5	5 5 6
6 5 2 3 + ***8*** * + 3 + *	push von 8	8 5 5 6
6 5 2 3 + 8 ***** + 3 + *	* auswerten	40 5 6
6 5 2 3 + 8 * **+** 3 + *	+ auswerten	45 6
6 5 2 3 + 8 * + *3* + *	push von 3	3 45 6
6 5 2 3 + 8 * + 3 **+** *	+ auswerten	48 6
6 5 2 3 + 8 * + 3 + *****	* auswerten	288

Beispiel: Test auf korrekte Klammersetzung

Test auf korrekte Klammersetzung :

Der Quelltext wird vom Anfang bis zum Schluss abgearbeitet. Dabei wird jede öffnende Klammer auf den Stack gelegt. Wird eine schliessende Klammer gefunden, wird vom Stack die zugehörende öffnende Klammer geholt. Passen die öffnende und die schliessende Klammer zusammen, kann weitergefahren werden, sonst muss eine Fehlermeldung erzeugt werden;

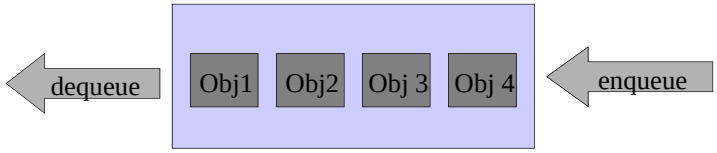
z.B. **XML** zu jedem *opening* Tag: <T> ein entsprechendes *closing* Tag: </T>: eine Regel der sogenannten *Wohlgeformtheit*: (*well formed*)

Ausdruck	**Aktion**	**Stack**
{ a = (b+3) * c[5] }	{ auf den Stack legen	{
{ a = ***(***b+3) * c[5] }	(auf den Stack legen	({
{ a = (b+3***)*** * c[5] }	) gefunden, pop ergibt (, ok, weiterfahren	{
{ a = (b+3) * c***[***5] }	[auf den Stack legen	[{
{ a = (b+3) * c[5***]*** }	] gefunden, pop ergibt [, ok, weiterfahren	{
{ a = (b+3) * c[5] ***}***	} gefunden, pop ergibt {, ok	leer

Warteschlangen, Queues, FIFO-Buffer

Eine Queue speichert Objekte in einer (Warte-) Schlange:

- die Objekte werden in derselben Reihenfolge entfernt wie sie eingefügt werden.
- Dies wird erreicht, indem die Objekte auf der einen Seite des Speicher hinzugefügt und auf der anderen Seite entfernt werden.



Die Queue ist eine
FIFO (first in first out)
Datenstruktur

Minimale Operationen

Funktionskopf	*Beschreibung*
`void enqueue (E x)`	Fügt x in die Queue ein
`E dequeue ()`	Entfernt das älteste Element und gibt es als Rückgabewert zurück
`boolean isEmpty()`	Gibt true zurück, falls die Queue leer ist

Zusätzliche Operationen

Funktionskopf	*Beschreibung*
`E peek ()`	Gibt das älteste Element als Rückgabewert zurück, ohne es zu entfernen
`void clear ()`	Leert die ganze Queue
`boolean isFull()`	Gibt true zurück, falls die Queue voll ist

Queue ADT Interface

```
/**
  Interface für Abstrakten Datentyp (ADT) Queue
*/
public interface Queue {
  /**
    Legt eine neues Objekt in die Queue, falls noch nicht voll.
    @param x ist das Objekt, das dazugelegt wird.
   */
  void enqueue (E x) throws Overflow;

  /**
    Entfernt das "älteste" und damit das zuerst eingefügte Objekt
von
    der Queue. Ist die Queue leer, wird null zurückgegeben.
    @return Gibt das oberste Objekt zurück oder null, falls leer.
  */
  E dequeue ();

  /**
    Testet, ob die Queue leer ist.
    @return Gibt true zurück, falls die Queue leer ist.
  */
  boolean isEmpty();

  /**
    Gibt das "älteste" Objekt zurück, ohne es zu entfernen.
    Ist die Queue leer, wird null zurückgegeben.
    @return Gibt das "älteste" Objekt zurück oder null, falls leer.
  */
  E peek ();

  /**
    Entfernt alle Objekte von der Queue. Ein Aufruf von isEmpty()
    ergibt nachher mit Sicherheit true.
  */
  void clear();

  /**
    Testet, ob die Queue voll ist.
    @return Gibt true zurück, falls der Stack voll ist.
  */
  boolean isFull();
}
```

Array-Implementation der Queue

Zwei **`int`** Variablen, **`outIdx`** und **`inIdx`** bestimmen den momentan gültigen Inhalt der Queue. Dabei zeigt **`outIdx`** auf das "älteste" Element, während **`inIdx`** auf die nächste freie Stelle zeigt. Zusätzlich werden noch die Anzahl Elemente der Queue in **`noOfItems`** gespeichert.

Index	0	1	2	3	4	5	6
Inhalt	?	?	34	27	11	?	?
Zeiger			outIdx			inIdx	

Beim Einfügen wird das Element an der Stelle [`inIdx`] eingefügt und `inIdx` und `noOfItems` um 1 erhöht,

In unserem Beispiel wird neu der Wert 77 eingefügt.

Index	0	1	2	3	4	5	6
Inhalt	?	?	34	27	11	77	?
Zeiger			outIdx				inIdx

Beim Entfernen von Elementen wird das Element an der Stelle [`outIdx`] entfernt, dann wird `outIdx` um 1 erhöht und `nofItems` um 1 reduziert.

es gilt: `noOfItems = inIdx - outIdx`

Wrap-Around bei der Array-Implementation der Queue

Was passiert aber, wenn in unserem Beispiel noch 2 weitere Elemente eingefügt werden?

Index	0	1	2	3	4	5	6	
Inhalt	?	?	34	27	11	77	52	
Zeiger			outIdx					inIdx

Fügen wir zuerst 52 ein.

Spätestens jetzt müssen wir uns fragen, was passiert, wenn wir noch das Element 45 einfügen.

Insbesondere, **wo** wir es einfügen wollen. Die einfachste Lösung wäre, einen **'Overflow'** Fehler zu melden.

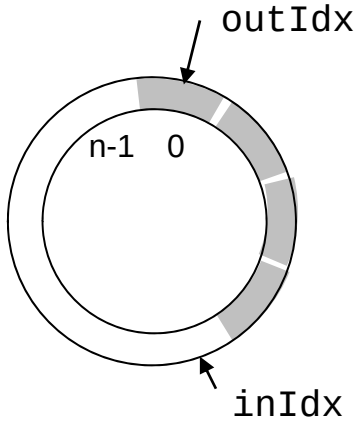
Aber eigentlich haben wir ja noch freien Platz, nämlich am Anfang des Arrays. Der Zeiger `inIdx` springt also um das Ende herum zurück zum Anfang.

Index	0	1	2	3	4	5	6
Inhalt	45	?	34	27	11	77	52
Zeiger	inIdx		outIdx				

es gilt nun nicht mehr: `noOfItems = inIdx - outIdx`

Frage: Wie kann die Operation: "erhöhe inIdx um 1 und setze zu 0 wenn die Array-Grenze erreicht wird" programmiert werden?

Queue Array-Implementation: Der Ringbuffer



Oft verwendete Datenstruktur für Buffer

Initialisierung:

```
private int size;
private int inIdx = 0;
private int outIdx = 0;
private int noOfItems = 0;
private E[] content;
```

```
class Queue<E> /* FIFO */{
  public Queue (int s) {
    size = s;
    content = (E[]) new Object[s];
  }
  public void enqueue(E o) {
    if (noOfItems == size) {
         throw new Exception ("buffer overflow");
    }
    content[inIdx] = o;
    inIdx = (inIdx+1) % size;
    noOfItems++;
  }

  public E dequeue() {
    if (noOfItems == 0) {
         throw new Exception ("buffer underlow");
    }
    E o = content[outIdx];
    outIdx = (outIdx+1) % size;
    noOfItems--;
    return o;
  }
```

List-Implementation der Queue

```
import java.util.List;
import java.util.LinkedList;

/** Implementation des Abstrak-
   en Datentyps (ADT) Queue mit
   Hilfe der vordefinierten
   Klasse LinkedList
*/

public class QueueLinkedList<E>
   implements Queue<E> {

  public QueueLinkedList() {
    list = new LinkedList<E>();
  }

  public void enqueue(E x) {
    list.add(x);
  }

  public E dequeue() {
    if (isEmpty()) return null;
    return list.remove(0);
  }

  public boolean isEmpty() {
    return list.isEmpty();
  }

  public E peek() {
    if (isEmpty()) return null;
    return list.get(0);
  }

  public void makeEmpty() {
    list.clear();
  }

  public boolean isFull() {
    return false;
  }

  private List<E> list;  // Delegation
}
```

Wo werden Queues angewendet?

Warteschlangen

In der Informatik oft eingesetzt.

- z.B. ein Drucker von mehreren Anwendern geteilt und wollen diese gleichzeitig drucken

- Allgemein wird dort wo der Zugriff auf irgendeine Ressource nicht parallel sondern **gestaffelt** erfolgen muss.

Warteschlangen-Theorie

Ein ganzer Zweig der Mathematik beschäftigt sich mit der Theorie von Warteschlangen.

- Fragestellung 1: Kassen in der Migros:
  - gewisses Kundenaufkommen (z.B. 10 pro Minute); bestimmte Verteilung
  - gewisse Verarbeitungszeit (z.B. 1 Minute)
  - wie viele Kassen müssen geöffnet werden, damit die Wartezeit 10 Minuten nicht übersteigt
- Fragestellung 2: Auslegung des Natelnetzes
  - gewisses Kundenaufkommen (z.B. 10 pro Minute); bestimmte Verteilung
  - gewisse Verarbeitungszeit (z.B. 1 Minute)
  - wie dicht muss der Antennenwald sein, wenn von jeder Station maximal 1000 Teilnehmer gleichzeitig bedient werden können.

Priority Queue

- Spezielle Art von Queue
  - Elemente werden anhand ihrer Priorität sortiert eingefügt. Elemente gleicher Priorität behalten ihre Reihenfolge.

- Anwendung:
  - persönliche Aufgabenliste mit Prioritäten geordnet
  - Rechnungen bezahlen nach Rechnungssteller: Staat, Mafia,
  - Scheduling von Prozessen in einem Betriebssystem
  - usw.

Priority Queues

Eine Priority Queue speichert Objekte in einer (Warte-) Schlange:

- Objekte gleicher Priorität werden in derselben Reihenfolge entfernt wie sie eingefügt wurden.
- Dies wird erreicht, in dem die Objekte sortiert eingefügt werden und jeweils das kleinste/grösste entfernt wird -> häufig wird ein Baum für die Implementation verwendet

Minimale Operationen

Funktionskopf	*Beschreibung*
`void enqueue (E x, int priority)`	Fügt x entsprechend priority ein.
`E dequeue ()`	Entfernt das erste Element und gibt es als Rückgabewert zurück
`boolean isEmpty()`	Gibt true zurück, falls die Queue leer ist

Zusätzliche Operationen

Funktionskopf	*Beschreibung*
`E peek ()`	Gibt das erste Element als Rückgabewert zurück, ohne es zu entfernen
`void makeEmpty ()`	Leert die ganze Queue
`boolean isFull()`	Gibt true zurück, falls die Queue voll ist

Zusammenfassung

- Konzept des ADTs in Java
  - Beschreibung der Schnittstelle durch **Interfaces**
  - Implementierung durch **Klasse**
- Referenz Datentypen in Java: Objektvariablen sind Zeiger
- Der Stack als LIFO Datenstruktur
  - als Array Implementation
  - als Listen Implementation
  - Anwendungen von Stack: UPN Rechner; Klammernsetzung
- Vererbung vs. Delegation
- Die Liste als generelle Datenstruktur: ADT der Liste
- Die Queue als FIFO Datenstruktur
  - als Array Implementation: der Ringbuffer
  - als Listen Implementation
  - Anwendungen von Queue: Printerqueue, Warteschlangentheorie
  - Priority Queues