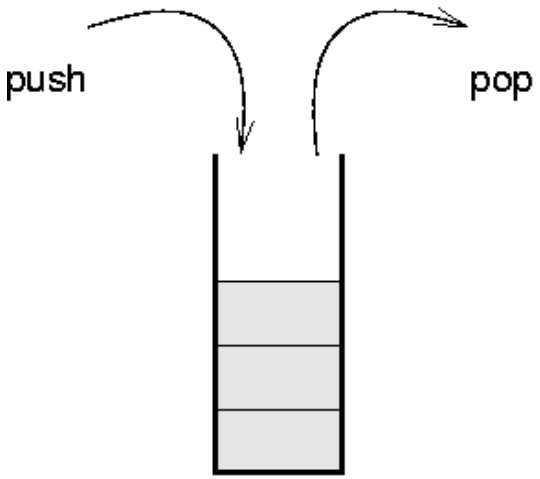
## Schnittstellen, Stack und Queue

- Schnittstelle **Stack**
- Realisierungen des Stacks
- Anwendungen von Stacks
- Schnittstelle **Queue**
- Realisierungen der Queue
- Anwendungen von Queues
- Hinweise zum Üben

**Anmerkung:**

*In den Beispielen werden keine Generics verwendet*

**Stack:** Last In First Out = LIFO




Stack – Stapelspeicher

```
public interface Stack {

    /** Speichert obj oben auf dem Stapel.
     *  Vorbed.: Stapel nicht voll!  Frage: was ist besser?
     *  @param obj zu speicherndes Objekt */
    public void push(Object obj);

    /** Entfernt das oberste Stapelelement und gibt es zurück.
     *  Vorbed.: Stapel nicht leer!
     *  @return oberstes Stapelobjekt
     *  @throws NoSuchElementException, wenn Stack leer */
    public Object pop();

    /** Gibt das oberste Element zurück ohne es zu entfernen.
     *  Vorbed.: Stapel ist nicht leer!
     *  @return oberstes Stapelobjekt
     *  @throws NoSuchElementException, wenn Stack leer */
    public Object peek();

    /** Ist der Stack voll?
     *  @return true, wenn Speicher erschöpft */
    public boolean isFull();

    /** Ist der Stack leer?
     *  @return true, wenn Stack leer */
    public boolean isEmpty();
}
```

```
public interface GenericStack<T> { // ALTERNATIVE

    public void push(T obj);

    public T pop();

    public T peek();

    public boolean isFull();

    public boolean isEmpty();
}
```

```
public class LinkedListStack implements Stack {
    private final SinglyLinkedList data;

    public LinkedListStack() {
        data = new SinglyLinkedList();
    }

    public void push(Object obj) {
        data.addFirst(obj);
    }

    public Object pop() {
        return data.removeFirst();
    }

    public Object peek() {
        return data.getFirst();
    }

    public boolean isFull() {
        return false;
    }

    public boolean isEmpty() {
        return data.isEmpty();
    }
}
```

Die Klasse `LinkedListStack` löst ihre Aufgaben mithilfe der Klasse `SinglyLinkedList`. Man sagt, dass sie ihre Arbeit an die Klasse `SinglyLinkedList` ***delegiert***.

```
public class ArrayStack implements Stack {
    private int top;
    private final Object[] data;

    public ArrayStack(int n) {
        top = 0;              /* naechster freier Platz */
        data = new Object[n];
    }
    public void push(Object obj) {
        data[top++] = obj;    /* bei Überlauf: Exception! */
    }
    public Object pop() {
        if (isEmpty()) throw new NoSuchElementException();
        return data[--top]; // data[top] = null;
    }
    public Object peek() {
        if (isEmpty()) throw new NoSuchElementException();
        return data[top-1];
    }
    public boolean isFull() {
        return top == data.length;
    }
    public boolean isEmpty() {
        return top == 0;
    }
}
```

## Einfache Anwendung der Stackklassen

```
public static void eingabe(Stack s) {
    boolean weiter = true;
    while (weiter) {
        System.out.print("Eingabe: ");
        int zahl = scanner.nextInt();
        if (zahl != 0)
            s.push(Integer.valueOf(zahl));
        else
            weiter = false;
    }
}

public static void ausgabe(Stack s) {
    while (!s.isEmpty()) System.out.print(s.pop() + " ");
    System.out.println();
}

public static void main (String[] args) {
    System.out.println("Bitte Werte für ersten Stack eingeben: ");
    Stack s = new LinkedListStack(); // oder ArrayStack(10);
    eingabe(s);
    System.out.print("Das war: ");
    ausgabe(s);
}
```

## Anwendungen von Stacks

- Laufzeitstack für Methodenaufrufe
- rekursive Algorithmen lassen sich mit Hilfe von Stacks immer durch Schleifen beschreiben (*allerdings ist dies dann viel schlechter lesbar und langsamer als die rekursive Formulierung*)
- UPN-Taschenrechner
- Syntaxanalyse (in Compiler oder Dateieingabe – vgl.: „Kellerautomat“)
- Java-Laufzeitsystem (temporäre Variable)
- Praktikumsaufgabe (`LIFOQueue`)

## Simulation der Rekursion mittels Stack (1)

```
static long fib(int n) {
    if (n <= 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

Die genaue Simulation des rekursiven Ablaufes ist sehr kompliziert und ineffizient!

Eine wesentliche Vereinfachung ist möglich:

- Es wird eine einzige Summe gebildet, bei der es nicht auf die Reihenfolge der Summanden ankommt.
- Diese Summe wird nur in dem Fall n==0 oder n==1 um 1 erhöht.
- Ansonsten muss man sich nur vormerken, dass noch zwei Fibonacci-Werte zu berechnen sind.

## Simulation der Rekursion mittels Stack (2)

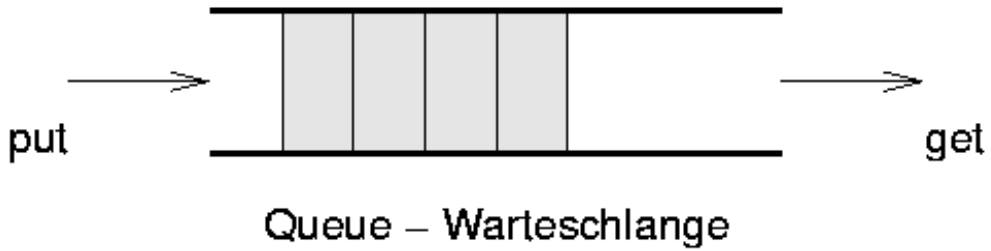
```
static long fib_mitStack(int n) {
    long summe = 0;
    Stack s = new ArrayStack();
    s.push(Integer.valueOf(n));
    while (!s.isEmpty()) {
        int i = ((Integer)s.pop()).intValue();
        if (i <= 1)              // Abbruchbedingung
            summe++;
        else {                   // „rekursive Aufrufe"
            s.push(Integer.valueOf(i-1));
            s.push(Integer.valueOf(i-2));
        }
    return summe;
}
```

**Fazit**: Auch dieser Algorithmus ist sehr langsam!

Es gibt aber einige sinnvolle Anwendungen dieser Idee (z.B. verallgemeinerte Graphsuche).

(Anmerkung: Natürlich hätte man einen speziellen Stack<Integer> verwenden können.)

**Queue:** First In First Out = FIFO



```
public interface Queue {

    /** Hängt obj an die Queue an.
     *  Vorbed.: Die Queue ist nicht voll!
     *  @param obj zu speicherndes Objekt
     */
    public void put(Object obj);

    /** Entnimmt das erste Element aus der Queue.
     *  Vorbed.: Die Queue nicht leer!
     *  @return erstes Element
     *  @throws NoSuchElementException, wenn Queue leer
     */
    public Object get();

    /** Ist die Queue voll?
     *  @return true, wenn Speicher erschöpft
     */
    public boolean isFull();

    /** Ist die Queue leer?
     *  @return true, wenn Queue leer
     */
    public boolean isEmpty();
}
```

```
public class LinkedListQueue implements Queue {
    private final SinglyLinkedList data;

    public LinkedListQueue() {
        data = new SinglyLinkedList();
    }

    public void put(Object obj) {
        data.addLast(obj);
    }

    public Object get() {
        return data.removeFirst();
    }

    public boolean isFull() {
        return false;
    }

    public boolean isEmpty() {
        return data.isEmpty();
    }
}
```

Die Klasse `ListQueue` ist fast identisch zu `ListStack`.

Problem: Komplexität von put() = O(*n*)

Lösung?

```
public class ArrayQueue implements Queue {

    private final Object[] data;  // Array um die Daten zu speichern
    private int size;     // Anzahl der Daten

    ...

    public boolean isFull() {
        return size == data.length;
    }
    public boolean isEmpty() {
        return size == 0;
    }
    ...
```

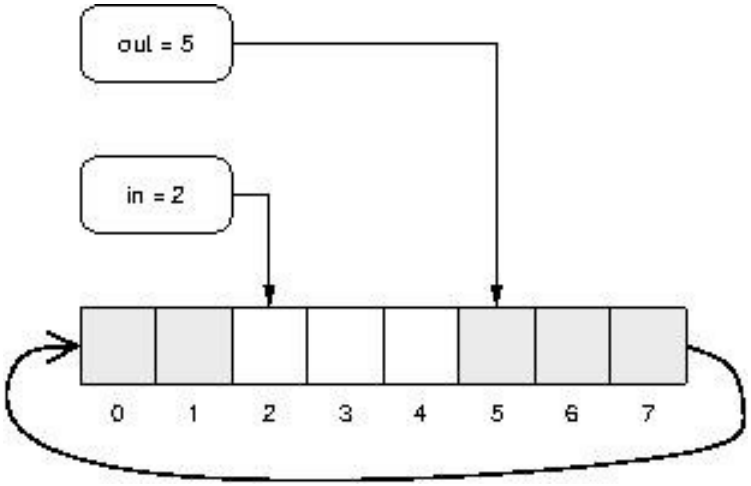
**Wie sollen wir die Queue realisieren?**

```
Queue v = new ArrayQueue(20);

for (int i=0; i<20; i++) q.put(Integer.valueOf(i));
for (int i=0; i<19; i++) System.out.putln(q.get());
```

ist Verschieben der Daten eine Lösung?

## Realisierung eines zirkulären Puffers



```
private final Object[] data;
private int size, in, out;

public ArrayQueue(int n) {
    size = 0;          // Anzahl der Inhalte
    in = 0;            // nächster freier Platz
    out =0;            // nächster Inhalt
    data = new Object[n];
}

public void put(Object obj) {
    if(isFull()) throw new IndexOutOfBoundsException("queue full");
    size++;
    data[in] = obj;
    in = (in+1) % data.length;
}

public Object get() {
    if(!isEmpty()) {
        size--;
        Object result = data[out];
        data[out] = null;
        out = (out+1) % data.length;
        return result;
    }
    else throw new NoSuchElementException();
}
```

## Anwendungen von Queues

- Datenpuffer
- Nachrichtenaustausch (mailbox)
- Ablaufsteuerung (z.B. Printer-Queue, ready queue)
- Algorithmen (s. ebenenweise Baumtraversierung, Breitensuche)
- Abwandlung durch priority queue.

## Hinweise zur direkten Realisierung von Stack / Queue durch verkettete Listen

- Stack und Queue: Operationen nur an den Enden der Datenstruktur
- Stack: Hinzufügen und Entfernen am gleichen Ende der Datenstruktur
- Queue: Hinzufügen und Entfernen an entgegengesetzten Enden der Datenstruktur

- Einfügen am Anfang der verketteten Liste: einfach
- Entfernen am Anfang der verketteten Liste: einfach
- Einfügen am Ende der verketten Liste: einfach, wenn letztes Element bekannt
- Entfernen am Ende der verketteten Liste: aufwändig (man muss das vorletzte Element kennen)