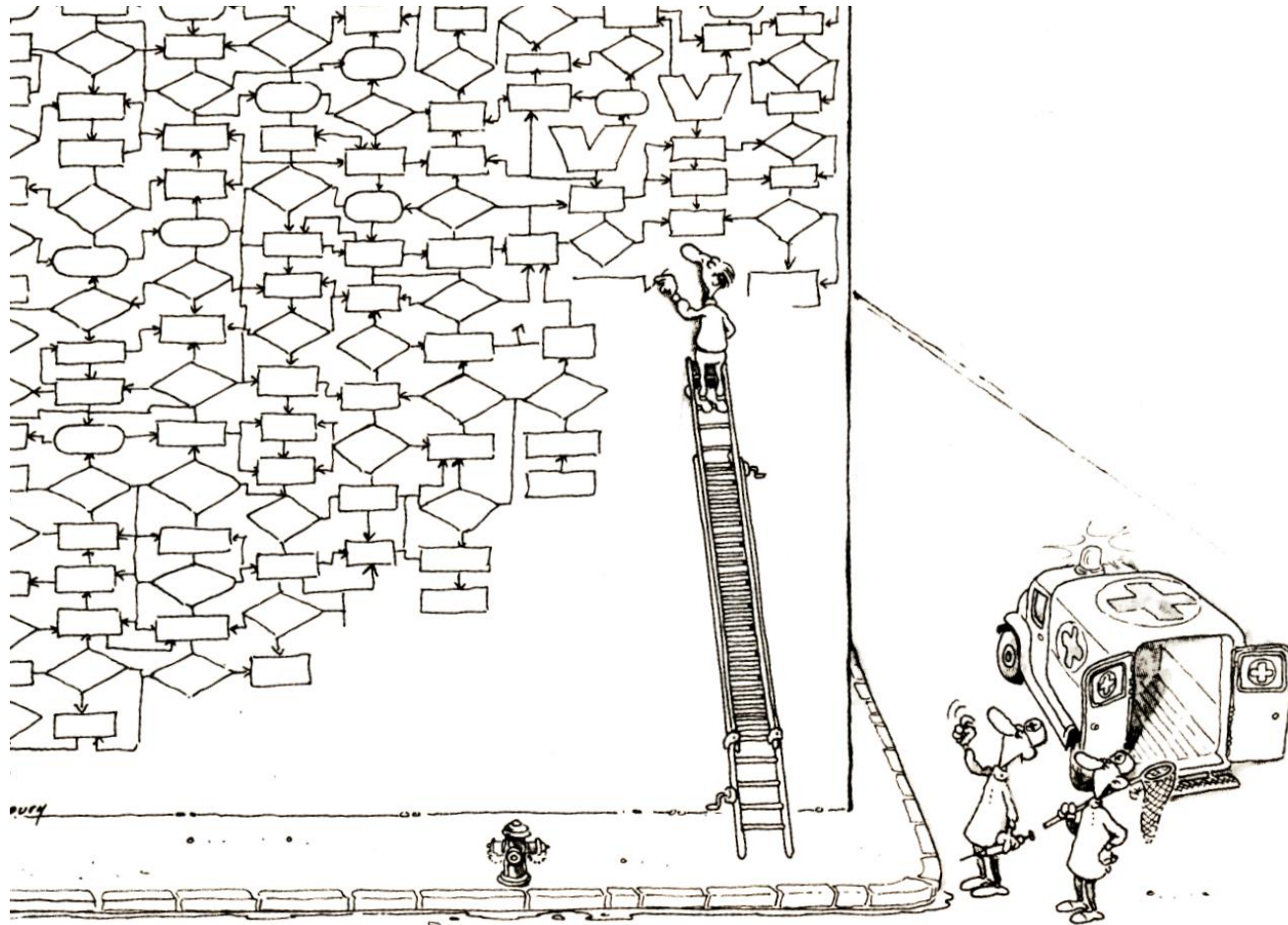


□ Wieso lernen wir Algorithmen?



- Algorithmen und Datentypen/Strukturen sind zentral in jedem IT System
- Früher musste man noch vieles selber ausprogrammieren, z.B. Sortierung
- Heute meist Bestandteil der Programmiersprache (Datentypen) oder Teil der Bibliothek (Algorithmen)
- Aber:
 - Nicht alle Aufgaben gelöst
 - Verwendung von Bibliotheks-Algorithmen setzt ein (Grund-) Verständnis voraus
- Viele der "alten" Algorithmen sind gut beschrieben und verstanden

- **Zweck:**
 - 1. Füllen des persönlichen Werkzeugkastens**
 - 2. Anhand einfacher, guter Beispielen sehen, wie man programmiert**
- und

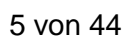
Semesterplan (ohne Gewähr)

1	ADT, Stacks, Queues	GUI Setup
2	Listen	Lineare Listen
3	Generics	Generics
4	Rekursion	Rekursion
5	Bäume	Bäume
6	Sortiert Bäume, Suchen	Sortiert Bäume, Suchen
7	Graphen, Topologien	Graphen, Topologien
8	Backtracking	Backtracking
9	Algorithmen Aufwand	Algorithmen Aufwand
10	Suchen, Hashing	Hashing
11	Sortieren 1	Sortierbeispiel
12	Sortieren 2	Sortierbeispiel 2
13	Speicherverwaltung	Speicherverwaltung
14	Simulated Annealing	Simulated Annealing

Vornote: 15 % Praktika und 15% Zwischenprüfung - 4 Seiten A4 erlaubt

SEP: 70% - 4 Seiten A4 erlaubt

www.zhaw.ch/~rege



- -300: Euklids Algorithmus zur Bestimmung des grössten gemeinsamen Teilers ggT in seinem 7. Buch der Elemente
- 800: Mohammed ibn Musa abu Djafar al Choresmi (الخوارزمي أبو عبد الله محمد بن موسى)
 - Übernahm indische Zahlen 1..9 ein und führte die 0 ein, 10er-System
 - Verschiedene Lehrbücher: bis 16. Jh in Europa als Standard verwendet (Bildung des Begriffs Algorithmus aus seinem Namen und dem griechischen "arithmo" für Zahl)
- 1574: das Rechenbuch von Adam Ries
- 1614: erste Logarithmentafel (in ca. 30 Jahren erstellt)
- 1703: binäres Zahlensystem von Leibniz
- 1931: Gödels Unvollständigkeitssatz: Bei hinreichend starken widerspruchsfreien Systemen gibt es immer unbeweisbare Aussagen.
- 1936: Church'sche These: stellt die Behauptung auf, dass eine Turingmaschine alle von Menschen berechenbaren mathematischen Funktionen lösen kann (<http://de.wikipedia.org/wiki/Turingmaschine>)
- Danach umfangreicher Ausbau der Algorithmentheorie



Was ist ein Algorithmus: Beispiel

□ Aufgabenstellung

- Gegeben: sind zwei ganze Zahlen a , b
- Gesucht: c soll der grösste gemeinsame Teiler von a und b sein

□ Einfacher Algorithmus:

- setze c zum Minimum der beiden Zahlen
- subtrahiere von c solange 1, bis die Divisionen a/c und b/c keinen Rest mehr ergeben

□ in Java

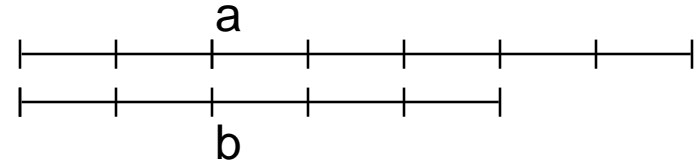
```
c = Math.min(a,b);  
while ((a % c != 0) || (b % c != 0)) c--;
```

□ Lösung wird gefunden, aber ...

Algorithmus nach Euklid (3. Jh v. Chr.)

□ Satz: der GGT zweier Zahlen ist auch GGT ihrer Differenz

□ Also: $\text{ggt}(a,b) = \begin{cases} a & \text{falls } a = b \\ \text{ggt}(a-b,b) & \text{falls } a > b \\ \text{ggt}(b-a,a) & \text{falls } a < b \end{cases}$



```
int ggt(int a, int b) {  
    if (a > b) return ggt(a-b,b);  
    else if (a < b) return ggt(a,b-a);  
    else return a;  
}
```

□ Algorithmus Iterativ

- subtrahiere von der grösseren Zahl die kleinere und weise den Wert der grösseren zu
- wiederhole dies, bis die Zahlen gleich sind

```
while (a != b) {  
    if (a > b) a = a - b;  
    else b = b - a;  
}  
c = a;
```


Ein **Algorithmus** ist eine *Anleitung zur Lösung einer Aufgabenstellung*, die so präzise formuliert ist, dass sie “mechanisch” ausgeführt werden kann:

- Mögliche Beschreibungen sind
 - Prosa, Pseudocode, Programmiersprache

Ein algorithmisches Problem:

- Problem das mit einem Algorithmus gelöst werden kann

□ Determiniertheit:

- Identische Eingaben führen stets zu identischen Ergebnissen.

□ Determinismus:

- Ablauf des Verfahrens ist an jedem Punkt fest vorgeschrieben (keine Wahlfreiheit).

□ Terminierung:

- Für jede Eingabe liegt Ergebnis nach endlich vielen Schritten vor.

□ Effizienz:

- "Wirtschaftlichkeit" des Aufwands relativ zu einem vorgegebenen Massstab (z.B. Laufzeit, Speicherplatzverbrauch).

- aber: Es darf auch nicht-terminierende, ineffiziente, nicht-deterministische und nicht-determinierte Algorithmen geben!

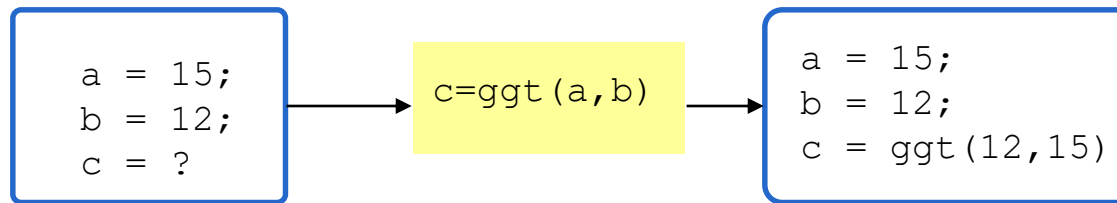
- grundlegender Zusammenhang zwischen den zentralen Begriffen:



- Jedes Programm repräsentiert einen bestimmten Algorithmus.
- Ein Algorithmus wird durch viele verschiedene Programme repräsentiert.
- Programmieren setzt Algorithmenentwicklung voraus:
- Kein Programm ohne Algorithmus !

- Programme bzw. Anweisungen können auch als Zustandstransformatoren betrachtet werden

- Eingangszustand : $a = \text{Wert1}, b = \text{Wert2}$
- Ausgangszustand: $a = \text{Wert1}, b = \text{Wert2}, c = \text{ggt}(\text{Wert1}, \text{Wert2})$



- Schreibweise: Hoare Tripple

- $\{a = 15, b = 12\} c = \text{ggt}(a,b) \{a = 15, b = 12, c = \text{ggt}(a,b)\}$
- {Eingangszustand} Anweisung {Ausgangszustand}

- $\{a = 5\} a = a + 1 \{a = 6\}$
- $\{a = \text{Wert}\} a = a + 1 \{a = \text{Wert} + 1\}$
- $\{a = 5\} b = \text{sqrt}(a) \{b = \sqrt{5}\}$
- $\{a \geq 0\} b = \text{sqrt}(a) \{b = \sqrt{a}\}$
- $\{\text{Vorbedingung}\} \text{Anweisung} \{\text{Nachbedingung}\}$

Ein Programm (= Folge von Anweisungen) wird als **korrekt** bezeichnet, wenn unter der Annahme, dass die Vorbedingung erfüllt ist, unter Anwendung des Programms die Nachbedingung erfüllt wird.

Man sagt auch: „die Vorbedingung impliziert die Nachbedingung“

korrektes Programme als Sequenz von korrekten Anweisungen

$\{P_0\} S_1 \{P_1\} S_2 \{P_2\}$

Man kann so die (partielle) Korrektheit von Programmen beweisen !

Wenn es terminiert dann ergibt es korrektes Resultat

□ Was Java zur Verfügung stellt:

- Einfache Datentypen
 - *byte, short, int, long*
 - *float, double*
 - *char*
 - *boolean*
- Referenz Datentypen
 - *array*
 - *string*
 - *objects*

□ Was der JDK zur Verfügung stellt

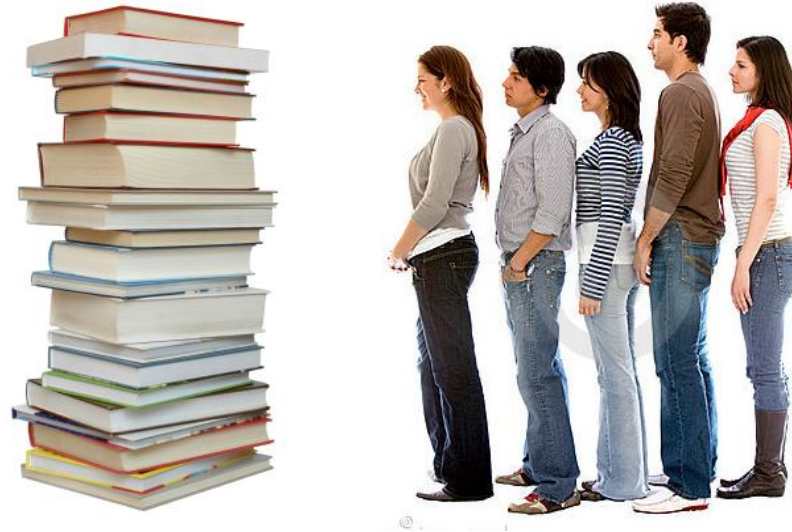
- *List*
- *Hashtable*
- *Collections*
-

□ Was man z.T. selbst machen aber sicher verstehen muss

- Komplexe Datenstrukturen
 - *Stacks*
 - *Queues*
 - *Trees*
 - ...



Thema dieser Vorlesung



- 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

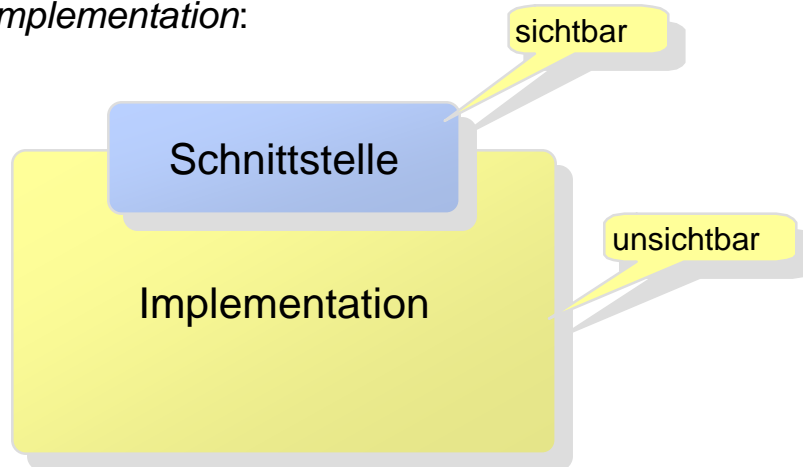
Abstrakter Datentyp

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 merkt.

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

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

```
interface Stack {  
    void push(Object obj);  
    Object pop();  
}  
  
class MyStack implements Stack {  
    void push(Object obj) {  
        // Implementation  
    }  
}  
  
Stack stack = new MyStack();
```

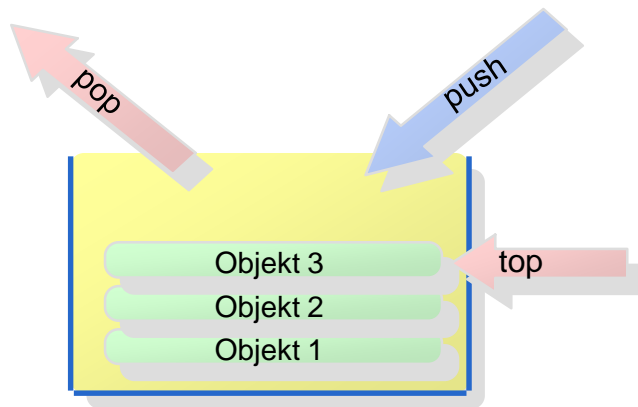
Variable ist vom Typ
des Interfaces

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

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

`void push (Object x)`

`Object pop ()`

`boolean isEmpty()`

Beschreibung

Legt x auf den Stapel

Entfernt das oberste Element und gibt es als Rückgabewert zurück

Gibt true zurück, falls der Stapel leer

Zusätzliche

Funktionskopf

`Object peek ()`

`void removeAll ()`

`boolean isFull()`

Operationen

Beschreibung

Gibt das oberste Element als Rückgabewert zurück, ohne es zu entfernen

Leert den ganzen Stapel

Gibt true zurück, falls der Stapel voll ist

Stack ADT als Java Interface

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

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

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

Vor- und Nachbedingungen in Interfaces

```
public int foo (short x, @NotNull String s) throws AnyException;
```

Compiler hilft bei der Fehlererkennung:
Compilerfehler sind "gute" Fehler: schnell gefunden, schnell korrigiert

Vorbedingung (Precondition)

- die Anzahl und Wertebereiche der Parameter, die durch Datentyp **formal (von Compiler überprüfbar)** bestimmt sind → Clean Code: so exakt bzw. einschränkend wie möglich (JavaScript, Python, Basic 😊)
- Weitere formale Einschränkungen z.B. @NotNull
- Weitere **informelle Einschränkungen** des Wertebereichs im Kommentar

Vorsicht mit "Spezialwerten" z.B. null, 31.12.1999 oder 42: dürfen nicht Teil des Rückgabe-Wertebereichs sein/werden.

Nachbedingung (Postcondition)

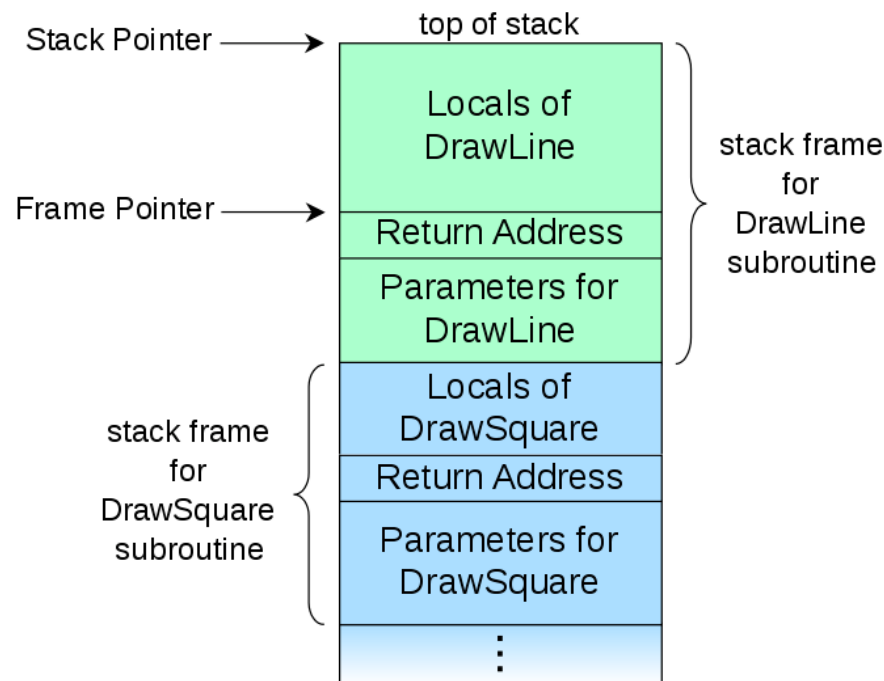
- die Wertebereiche der Rückgabeparameter
- **Fehlerfälle**: formal durch checked Exceptions oder Spezialwert signalisiert, informell durch Kommentar

Implementationen und (später) neue Versionen dürfen

- Vorbedingung **abschwächen** i.e. Wertebereich erweitern (z. B. short → int).
- Nachbedingung **verstärken** i.e. Wertebereich verkleinern (z.B. int → short).

□ 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 (später unter Rekursion).*



... wo werden Stacks angewendet?

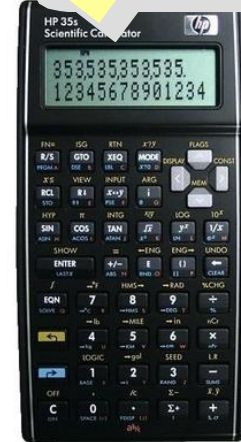
□ Die Auswertung eines Ausdrucks in Postfix-Notation

- läuft folgendermassen ab:
- Zahlen werden auf den Stack gelegt <Enter>
- Operatoren werden sofort ausgewertet, dazu werden 2 Elemente vom Stack geholt
- Das Resultat wird wieder auf den Stack gelegt
- Am Ende enthält der Stack das Schlussresultat
- Der Ausdruck: $6 * (5 + (2 + 3) * 8 + 3)$ wird wie folgt ausgewertet

HP 35 1972



HP 35s 2007



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

... wo werden Stacks angewendet?

□ Test auf korrekte Kammersetzung :

- Der Programm-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 zugehörige öffnende Klammer geholt.
- Passen die öffnende und die schliessende Klammer zusammen, kann weitergefahren werden, sonst muss eine Fehlermeldung erzeugt werden;

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

□ XML Wohlgeformtheit

- **XML** zu jedem *opening* Tag: <T> ein entsprechendes *closing* Tag: </T>
- dies ist die wichtigste Regel der sogenannten *Wohlgeformtheit von XML*: (*well formed*)

Array-Implementation des Stack

```
public class StackArray implements
    Stack {

    Object[] data;
    private int top;
    public StackArray(int capacity){
        removeAll();
    }

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

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

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

public Object peek() {
    if (isEmpty()){return null;}
    return data [top-1];
}

public void removeAll() {
    data = new Object[capacity];
    top = 0;
}

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

- 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 0 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?

- Abstrakter Datentyp, der eine Liste von Objekten verwaltet.



- Implementation muss nicht interessieren (kommt später).

- Liste ist eine der grundlegenden Datenstrukturen in der Informatik.

- Wir können mit Hilfe der Liste einen Stack implementieren.

Schnittstelle: `java.util.List`

Impl.: `java.util.LinkedList`

Minimale Operationen

Funktionskopf

`void add (Object x)`

`void add (int pos, Object x)`

`Object get(int pos)`

`Object remove(int pos)`

`int size()`

`boolean isEmpty()`

Beschreibung

Fügt x am Schluss der Liste an

Fügt x an der pos in die Liste ein

Gibt Element an pos zurück

Entfernt das pos Element und gibt es als Rückgabewert zurück

Gibt Anzahl Element zurück

Gibt true zurück, falls die Liste leer

List-Implementation des Stack

```
import java.util.*;

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

public class StackLinkedList
    implements Stack{

    public StackLinkedList() {
        removeAll()
    }

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

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

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

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

public void removeAll() {
    list = new LinkedList()
}

public boolean isFull() {
    return false;
}

private List list;
}
```

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

list als **Attribut** der Klasse StackLinkedList

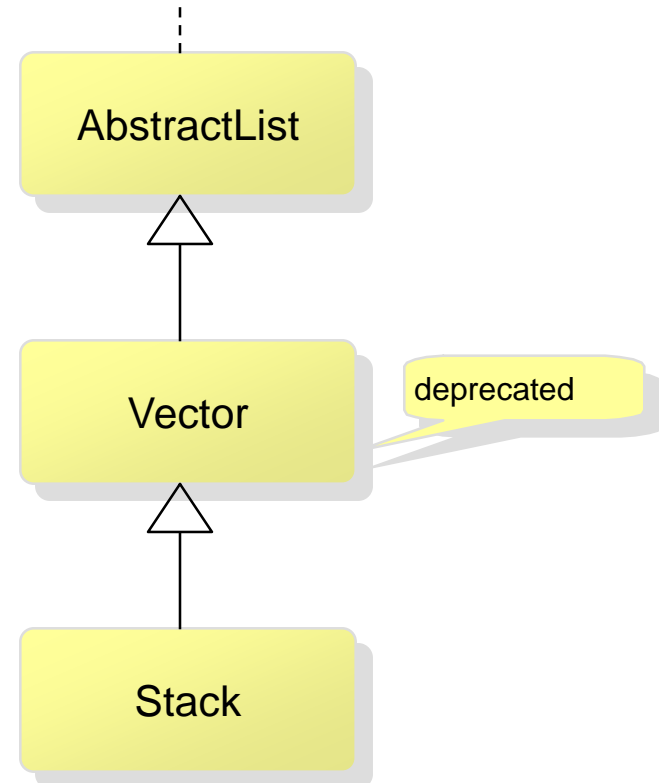
Funktion wird "delegiert"

immer false

Stack Implementation im JDK

- ❑ Es existiert eine Stack Implementation im java.util package
- ❑ Schönes Beispiel schlechter Anwendung von Vererbung
- ❑ Verletzung der Namenskonventionen
 - ❑ isEmpty -> empty
- ❑ Information Hiding Prinzip verletzt
 - ❑ Implementation ist immer ein Vector
- ❑ Encapsulation Prinzip verletzt
 - ❑ Muss die eigentlich deprecated Vector Klasse verwenden
 - ❑ Man kann mit Vector Methoden die Konsistenz des Stacks zerstören

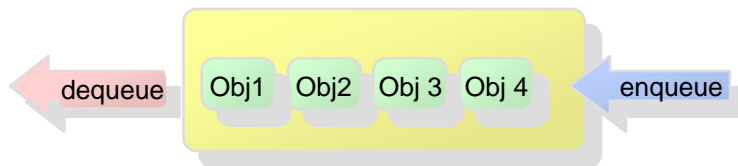
Nur wenn man "ist ein" sagen kann, darf man Vererbung verwenden



Queues

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

```
void enqueue (Object x)  
Object dequeue ()
```

```
boolean isEmpty()
```

Beschreibung

Fügt x in die Queue ein
Entfernt das älteste
Element und gibt es als
Rückgabewert zurück

Gibt true zurück,
falls die Queue leer ist

Zusätzliche Operationen

Funktionskopf

```
Object peek ()
```

```
void removeAll ()
```

```
boolean isFull()
```

Beschreibung

Gibt das älteste Element
als Rückgabewert
zurück, ohne es
zu entfernen

Leert die ganze Queue
Gibt true zurück, falls die
Queue voll ist

Queue ADT Interface

```
public interface Queue {
    /**
     * Legt eine neues Objekt in die Queue, falls noch nicht voll.
     * @param x ist das Objekt, das dazugelegt wird. */
    void enqueue (Object 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.
     */

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

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


Wo werden Queues angewendet?

- **Warteschlangen** werden in der Informatik oft eingesetzt.
- z.B. ein Drucker von mehreren Anwendern geteilt

- 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.
 - Fragestellungen 1: Anzahl benötigter Kassen in einem Geschäft:
 - *gewisses Kundenaufkommens (z.B. 10 pro Minute); bestimmte Verteilung*
 - *gewisses Verarbeitungszeit (z.B. 1 Minute)*
 - *wie viele Kassen müssen geöffnet werden, damit die mittlere Wartezeit 10 Minuten nicht übersteigt*
 - Fragestellung 2: Auslegung des Natelnetzes
 - *gewisses Anzahl Natelbenutzer*
 - *Benutzungswahrscheinlichkeit und Dauer*
 - *wie viele Antennen notwendig, damit in 99% der Fälle eine Verbindung zustande kommt.*

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 **noItems** gespeichert.

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



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



Beim Einfügen wird das Element an der Stelle [inIdx] eingefügt und inIdx und noItems um 1 erhöht, In unserem Beispiel wird neu der Wert 77 eingefügt.

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

es gilt: $\text{noItems} = \text{inIdx} - \text{outIdx}$

Wrap-Around der Array-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	
			outIdx					inIdx



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



Fügen wir zuerst 52 ein.

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

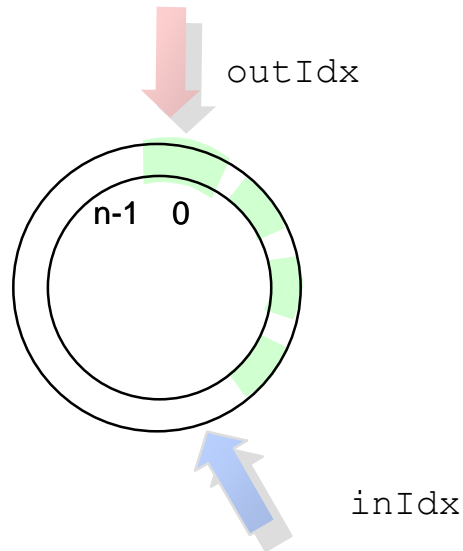
Insbesondere, **wo** wir es einfügen wollen. Die einfachste Lösung wäre, einen '**Overflow**' Fehler zu melden.

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

es gilt nun nicht mehr: $\text{noItems} = \text{inIdx} - \text{outIdx}$

Aufgabe: 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 inIdx = 0;  
private int outIdx = 0;  
private int noOfItems = 0;  
private Object[] content;  
private int size;
```

```
class Queue /* FIFO */{  
    public Queue (int s) {size = s; content = new  
        Object[s];  
    }  
    public void enqueue(Object o) {  
        if (noOfItems == size) {  
            throw new Exception ("buffer overflow");  
        }  
        content[inIdx] = o;  
        inIdx = (inIdx+1) % size;  
        noOfItems++;  
    }  
  
    public Object dequeue() {  
        if (noOfItems == 0) {  
            throw new Exception ("buffer underflow");  
        }  
        Object o = content[outIdx];  
        outIdx = (outIdx+1) % size;  
        noOfItems--;  
        return o;  
    }  
}
```

List-Implementation der Queue

```
import java.util.*;

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

public class QueueLinkedList
    implements Queue {

    public QueueLinkedList() {
        removeAll()
    }

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

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

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

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

    public void removeAll() {
        list = new LinkedList();
    }

    public boolean isFull() {
        return false;
    }

    private List list; // Delegation
}
```

Priority Queue

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

- Objekte hoher Priorität wandern nach vorne
- Objekte gleicher Priorität werden in derselben Reihenfolge entfernt wie sie eingefügt wurden.



Minimale Operationen

Funktionskopf

```
void enqueue (Object x,  
              int priority)  
Object dequeue ()
```

```
boolean isEmpty()
```

Beschreibung

Fügt x in die Queue ein
Entfernt das erste Element und gibt es als Rückgabewert zurück

Gibt true zurück, falls die Queue leer ist

Zusätzliche Operationen

Funktionskopf

```
Object peek ()
```

```
void removeAll ()
```

```
boolean isFull()
```

Beschreibung

Gibt das erste Element als Rückgabewert zurück, ohne es zu entfernen

Leert die ganze Queue
Gibt true zurück, falls die Queue voll ist

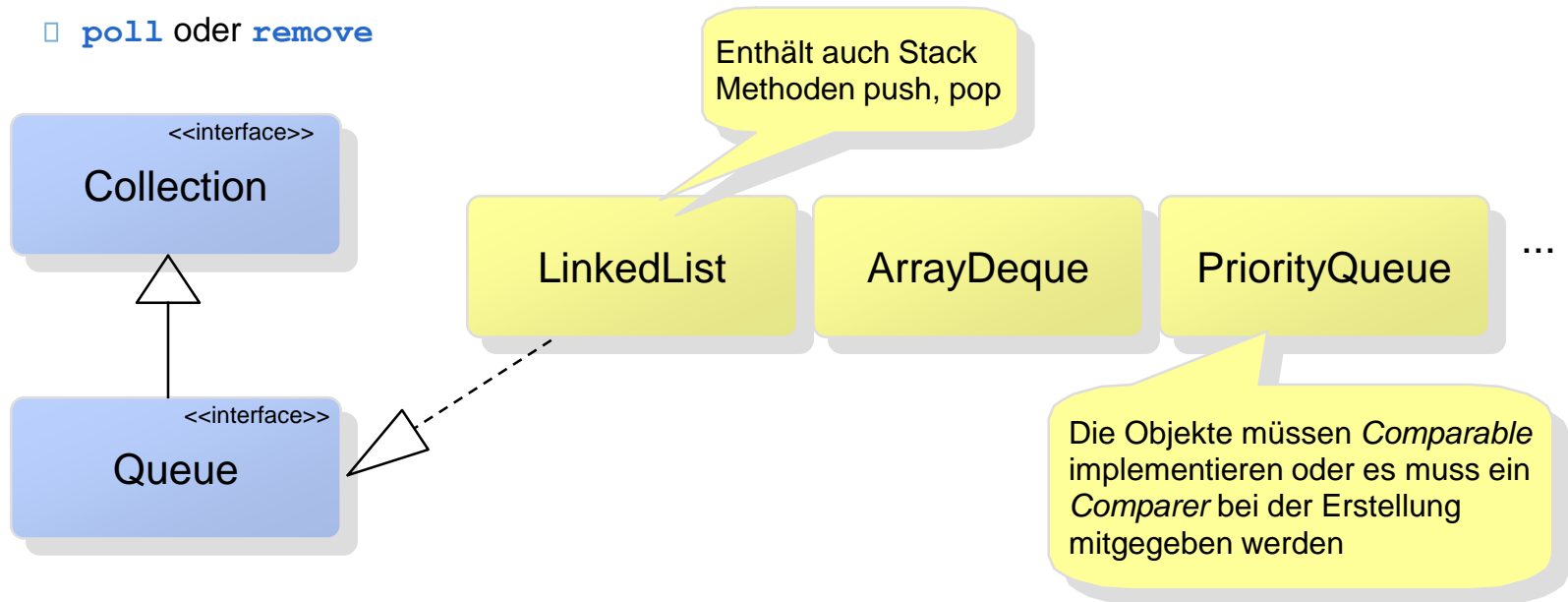
□ Anwendung:

- Scheduling von Prozessen in Betriebssystemen
- Taskliste nach Prioritäten geordnet
- Rechnungen bezahlen nach Rechnungssteller: Mafia zuerst
- usw.

Queue Implementation im JDK



- Queue Interface in java.util
- Einfügen in Queue (enqueue)
 - `offer` oder `add`
- Holen des ersten Elements
 - `poll` oder `remove`



- Konzept des ADTs in Java
 - Beschreibung der Schnittstelle durch **Interfaces**
 - Implementierung durch **Klasse**
- Der Stack als LIFO Datenstruktur
 - als Array Implementation
 - als Listen Implementation
 - Anwendungen von Stack: UPN Rechner; Klammernsetzung
- Listen zur Implementation von andern Datenstrukturen
- Die Queue als FIFO Datenstruktur
 - als Array Implementation: der Ringbuffer
 - als Listen Implementation
 - Anwendungen von Queue: Printerqueue, Warteschlangentheorie
- Priority Queue

Anhang - Best Developers

Best Developers?

Ranked by Average Score Across

Rank	Country	Score Index
1	China	100.0
2	Russia	99.9
3	Poland	98.0
4	Switzerland	97.9
5	Hungary	93.9
6	Japan	92.1
7	Taiwan	91.2
8	France	91.2
9	Czech Republic	90.7
10	Italy	90.2
11	Ukraine	88.7
12	Bulgaria	87.2
13	Singapore	87.1
14	Germany	84.3

Never Gives Up?

% Which Scored a Zero

Rank	Country	Score Index
1	Switzerland	2.5%
2	Hungary	2.7%
3	Poland	2.7%
4	Spain	3.0%
5	Bulgaria	3.0%
6	Argentina	3.2%
7	Russia	3.2%
8	Italy	3.2%
9	Czech Republic	3.2%
10	Germany	3.3%

Which Challenges Are the Most Popular?

Percentage of HackerRank Challenges Solved By Type

Rank	Domain	Percent of Challenges Solved
1	Algorithms	39.5%
2	Java	9.3%
3	Data Structures	9.1%
4	C++	6.6%
5	Tutorials	6.5%
6	Mathematics	6.1%
7	Python	5.3%
8	SQL	4.6%
9	Shell	3.1%
10	Artificial Intelligence	2.9%
11	Functional Programming	2.5%
12	Databases	1.5%
13	Ruby	1.0%
14	Distributed Systems	1.0%
15	Security	0.9%
Total		100.0%