ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Klassen und Objekte

Prof. Dr. Helmut G. Folz

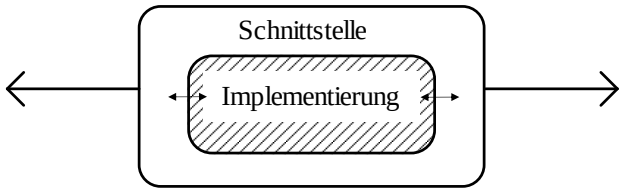
Objekte und Klassen

Definition:

Eine ***Klasse*** beschreibt eine Menge von Objekten mit gleichen Eigenschaften, gleichem Verhalten, gemeinsamen Beziehungen zu anderen Objekten und gemeinsamer Semantik. Eine Klasse definiert also die Eigenschaften und das Verhalten ihrer Objekte.

Grundprinzipien

- **Datenkapselung**: Die Daten und die dazugehörigen Operationen sind gekapselt in einer Programmeinheit
- **Geheimnisprinzip**: Die Daten sind nur innerhalb des Objekts bekannt. Der Zugriff erfolgt über die öffentlichen Operationen.



Aufbau von Klassen

Eine **Klasse** beschreibt eine Menge von Objekten und besteht aus

- **Attributen**
  - ⇨ Statische Eigenschaften
  - ⇨ Die eigentlichen Daten, die nicht direkt von außen zugreifbar sein sollen.
- **Methoden**
  - ⇨ Dynamische Eigenschaften
  - ⇨ Funktionen, Operationen der Klasse
- **Konstruktoren**
  - ⇨ Sind spezielle Methoden zum Erzeugen von Objekten
  - ⇨ Es kann beliebig viele Konstruktoren geben.

Objekt

- Ein **Objekt** ist ein konkretes "Exemplar" einer Klasse, oft wird missverständlich von einer "Instanz" gesprochen (engl. instance)
- Ein Objekt wird mit Hilfe eines Konstruktors der Klasse erzeugt.
- Jedes Objekt besitzt einen kompletten Satz an Attributen mit jeweils eigenen Werten.

Klasse Punkt : UML-Notation

Punkt
-x : double = 0.0 -y : double = 0.0
+getX() +getY() +verschiebe(deltaX : double, deltaY : double) +skaliere(faktor : double) +abstand(p : Punkt)

Beispiel: Klasse Punkt (1)

```
public class Punkt {
    /**
     * Erzeuge Punkt-Objekt mit beiden Koordinaten
     *
     * @param x X-Koordinate
     * @param y y-Koordinate
     */
    public Punkt (double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() { return x; }
    public double getY() { return y; }
```


Beispiel: Klasse Punkt (2)

```
   /**
    * Verschieben eines Punktes
    *
    * @param deltaX um so viel wird die X-Koordinate verschoben
    * @param deltaY um so viel wird die Y-Koordinate verschoben
    */
    public void verschiebe (double deltaX, double deltaY) {
        this.x += deltaX;
        this.y += deltaY;
    }

    /**
     * Skalieren eines Punktes um einen Faktor
     *
     *   @param faktor Koordinaten werden damit multipliziert
     */
    public void skaliere (double faktor){
        x *= faktor;
        y *= faktor;
    }
```

Beispiel: Klasse Punkt (3)

```
    /**
     * Abstand zu einem anderen Punkt ermitteln
     *
     * @param p anderer Punkt (muss != null sein)
     * @return berechteter euklidischer Abstand
     */
    public double abstand (Punkt p) {
        return Math.sqrt((x - p.x)*(x - p.x)
                         + (y - p.y)*(y - p.y));
    }

    private double x = 0.0;
    private double y = 0.0;
}
```


Klassendefinition (1)

- Innerhalb des Klassenblocks sind alle Merkmale der Klasse definiert.
- Die Attribute sind hier jeweils mit dem Modifikator **`private`**, die Methoden mit dem Modifikator **`public`** versehen.
- Die Attribute sind damit vor Zugriffen von außerhalb der Klasse geschützt, während die Methoden öffentlich zugänglich sind.
- Die Methode **`Punkt()`** ist ein sogenannter Konstruktor und dafür zuständig, beim Anlegen eines Objektes die Attribute zu initialisieren.

Klassendefinition (2)

- Eine Klasse definiert automatisch einen neuen Datentyp, einen Referenztyp, d. h. Objekte von Klassen müssen generell dynamisch allokiert werden.
- Beim Anlegen eines Objektes erhält das Objekt Speicherplatz für alle in der Klasse definierten (Objekt-) Attribute (Zusatz `static` fehlt).
- Ein Objekt ist im Prinzip ein Modul im Sinne der modularen Programmierung.
  - Seine enthaltenen Attribute sind die privaten Daten,
  - die öffentlich zugängliche Schnittstelle wird durch die `public-Methoden` repräsentiert.
- Bei der Definition von Klassen gibt es noch weitere Möglichkeiten, Zugriffsrechte an Attribute und Methoden zu verteilen.

Übersicht über Zugriffsrechte

		Modifikator	*Zugriff erlaubt für*
Klasse	*öffentlich*	**`public`**	alle Klassen
	Paket	**`--`**	alle Klassen des gleichen Paketes
Attribute, Methoden	*öffentlich*	**`public`**	alle Klassen, die auf die zugehörige Klasse zugreifen dürfen
	privat	**`private`**	die eigene Klasse
	geschützt	**`protected`**	Klassen des Paketes und für Unterklassen (auch außerhalb des Paketes)
	Paket	**`--`**	alle Klassen des gleichen Paketes

Klasse: Zusammenfassung

- Eine Klasse besteht aus Attributen und Methoden, die Daten und Funktionen der von einer Klasse erzeugbaren Objekte.
- Jedes erzeugte Objekt erhält einen eigenen Satz an Attributen, auf die mit Hilfe der Methoden zugegriffen werden kann.
- Eine Klasse kann ebenfalls beinhalten:
  - ⇨ **Klassenattribute** (Zusatz `static`), die nur einmal je Klasse vorhanden sind, und
  - ⇨ **Klassenmethoden** (Zusatz `static`), die unabhängig von einem bestimmten Objekt aufrufbar sind.

Konstruktoren (1)

- Konstruktoren sind spezielle Methoden zum Konstruieren und Vorbelegen von Objekten.
- Jede Klasse hat mindestens einen Konstruktor. Dieser heißt genau wie die Klasse. Konstruktoren haben keinen Rückgabewert.
- Ist kein Konstruktor explizit definiert, so wird vom Compiler ein parameterloser Konstruktor, der sogenannte **Standard-Konstruktor** (default constructor) erzeugt.
- Ist ein Konstruktor vorhanden, so wird nicht automatisch ein Standard-Konstruktor angelegt

Konstruktoren (2)

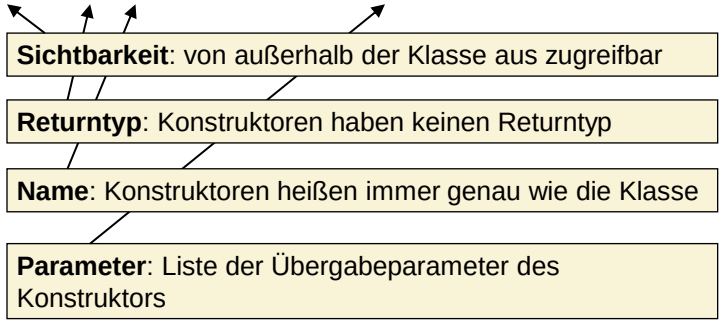
- Definition eines zusätzlichen Standard-Konstruktors für unser Beispiel:

```
public Punkt () {
   this(0.0, 0.0);
}
```

- Eine Klasse kann theoretisch beliebig viele Konstruktoren besitzen. Diese müssen sich durch Anzahl und Typ ihrer Argumente unterscheiden.
- Konstruktoren werden normalerweise `public` angelegt.

Konstruktor

```
public Konto(String inhaber, int kontonr){…}
```



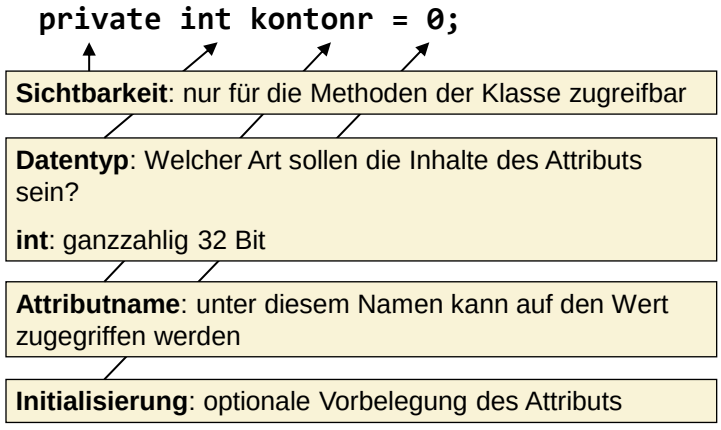
Attribute

- Attribute können von beliebigem Typ sein.
- Attribute von Referenztyp sind natürlich zunächst Referenzvariablen und müssen explizit allokiert werden.
- Attribute werden automatisch initialisiert beim Allokieren des Objektes und sind beim Eintritt in den Konstruktor bereits mit sinnvollen Werten versehen.
- Attribute können auch explizit in der Klasse initialisiert werden, wie das Beispiel gezeigt hat:

```
private double x = 0.0, y = 0.0;
```

- Klassenattribute (Zusatz `static`) werden bereits direkt nach dem Laden der Klasse angelegt und initialisiert.

(Objekt-)Attribut



Initialisierung von Attributen

Typ	Initialwert
`boolean`	`false`
`char`	`'\u0000'`
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0`
`float`	`0.0f`
`double`	`0.0`
`beliebiger Referenztyp`	`null`

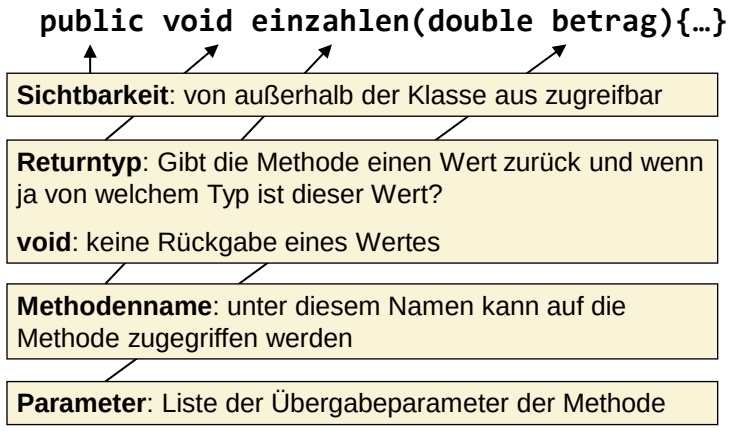
Die this-Referenz

- Wie ist es möglich, dass eine Methode immer den direkten Zugriff auf die passenden Objekt-Attribute hat?
- Die Lösung ist die sogenannte `this`-Referenz.
- Ähnlich wie bei C++ wird bei Java einer Methode als unsichtbarer erster Parameter eine Referenz auf das zugehörige Objekt übergeben.

```
public Punkt (double x, double y) {
    this.x = x;
    this.y = y;
}

public double getX() {
    return x;
}
public double getY() {
    return y;
}
```

(Objekt-)Methode



Überladen von Methoden

- Namen von Methoden müssen nicht eindeutig sein, wie dies bei vielen klassischen Programmiersprachen der Fall sein muss.
- Die Unterscheidung von Methoden erfolgt aufgrund ihrer sogenannten Signatur, d. h. durch Name, Anzahl und Typ der Parameter.
- Beispiel: die Klasse `PrintStream`

```
void println (int arg)        { .... }
void println (double arg)     { .... }
void println (String arg)     { .... }
void println (Object arg)     { .... }
...
```

Die spezielle Methode toString()

- Jede Klasse erbt direkt oder indirekt von der Basisklasse `java.lang.Object` die Methode

  ```
  String toString()
  ```

- Diese Methode ist dazu gedacht, von einem Objekt eine textuelle Aufbereitung seines Inhaltes machen zu können. Z.B.:

```
public class Punkt {
...
    public String toString() {
        return new StringBuffer()
                .append('(').append(x)
                .append(", ").append(y)
                .append(')')
                .toString();
    }
```


Die spezielle Methode equals()

- In fast jeder Klasse gibt es auch die Notwendigkeit einer Methode, die die Gleichheit zweier Objekte überprüft.
- Diese Methode heißt, wie bei der `String`-Klasse, `equals()` und würde in unserer `Punkt`-Klasse so aussehen:

```
public boolean equals(Object o) {
   if (o instanceof Punkt) {
       Punkt p = (Punkt)o;
       return (x == p.x && y == p.y);
   } else
       return false;
}
```

Der Operator instanceof

- Syntax:

  a `instanceof` *Klassenname*

- Dieser Ausdruck hat den Wert `true`, wenn die Referenz *a* auf ein Objekt der Klasse *Klassenname* bzw. auf ein Objekt, dessen Klasse von der Klasse *Klassenname* abgeleitet ist, zeigt.
- Wenn a die Nullreferenz ist, hat der Ausdruck immer den Wert `false`
- `if (o instanceof Punkt)`
  überprüft also ob die Referenz o auf ein Punkt-Objekt verweist.

Beispiel: Klasse Punkt (1)

```
/**
 * Definition eines Punktes mit x- und y-Koordinate
 */
public class Punkt {
    /**
     * Erzeuge Punkt-Objekt mit beiden Koordinaten
     *
     * @param x X-Koordinate
     * @param y y-Koordinate
     */
    public Punkt (double x, double y) {
        this.x = x;
        this.y = y;
    }
    /**
     * Standard-Konstruktor, initialisiert die Koordinaten mit 0.0
     */
    public Punkt () {
        this(0.0, 0.0);
    }
```

Beispiel: Klasse Punkt (2)

```
public double getX() { return x; }
public double getY() { return y; }
/**
 * Verschieben eines Punktes
 *
 * @param deltaX um so viel wird die X-Koordinate verschoben
 * @param deltaY um so viel wird die Y-Koordinate verschoben
 */
public void verschiebe (double deltaX, double deltaY) {
    this.x += deltaX;
    this.y += deltaY;
}
/**
 * Skalieren eines Punktes um einen Faktor
 *
 *   @param faktor Koordinaten werden damit multipliziert
 */
public void skaliere (double faktor){
    x *= faktor;
    y *= faktor;
}
```


Beispiel: Klasse Punkt (3)

```
/**
 * Abstand zu einem anderen Punkt ermitteln
 *
 * @param p anderer Punkt (muss != null sein)
 * @return berechteter euklidischer Abstand
 */
public double abstand (Punkt p) {
    return Math.sqrt((x - p.x)*(x - p.x)
                   + (y - p.y)*(y - p.y));
}

public String toString() {
    return new StringBuffer()
           .append('(').append(x)
           .append(", ").append(y)
           .append(')')
           .toString();
}
```

Beispiel: Klasse Punkt (4)

```
    /**
     * Zwei Punkte auf Gleichheit pruefen
     *
     * @param o muss ein gültiges Punkt-Objekt referenzieren
     * @return true, falls o ein Punkt-Objekt ist und inhaltlich
     *         gleich zu this
     *         false sonst
     */
    public boolean equals(Object o) {
        if (o instanceof Punkt) {
            Punkt p = (Punkt)o;
            return (x == p.x && y == p.y);
        } else
            return false;
    }

    private double x = 0.0;
    private double y = 0.0;
}
```


Beispiel: Klasse PunktTest

```
public class PunktTest {

    public void start() {
        Punkt p1 = new Punkt(1.0,2.0);
        Punkt p2 = new Punkt(1.0,5.0);
        System.out.println("p1: " + p1);
        System.out.println("p2: " + p2);
        System.out.println("Abstand zwischen p1 und p2 : "
                           + p1.abstand(p2));
        p1.skaliere(3.0);
        System.out.println("p1: " + p1);
        if (p1.equals(p2))
            System.out.println("p1 ist gleich p2");
        else
            System.out.println("p1 ist nicht gleich p2");
    }

    public static void main(String[] args) {
        new PunktTest().start();
    }
}
```

Beispiel: Klasse Strecke

- Eine Strecke besteht aus zwei Punkten
- Eine solche Beziehung nennt man auch Assoziation, hier speziell auch Kompositon oder starke Aggregation (Ganzes-Teil-Beziehung)

Beispiel: Klasse Strecke (1)

```
/** Definition einer Strecke im 2-dimensionalen Raum     */
public class Strecke {
    public Strecke (double x1, double y1,
                    double x2, double y2) {
        p = new Punkt(x1, y1);
        q = new Punkt(x2, y2);
    }
    public Strecke (Punkt p1, Punkt p2) {
        p = p1;
        q = p2;
    }
    public Punkt getP() { return p; }
    public Punkt getQ() { return q; }

    public void verschiebe (double a, double b){
        p.verschiebe(a,b);
        q.verschiebe(a,b);
    }
```

Dieser Konstruktor ist falsch! Warum?

Beispiel: Klasse Strecke (2)

```
    /**
     * Laenge der Strecke, d. h. Abstand zwischen Anfangs-
     * und Endpunkt
     * @return errechnete Laenge
     */
    public double getLaenge() {
        return p.abstand(q);
    }

    public String toString() {
        return new StringBuffer()
                .append('(') .append(p)
                .append(", ").append(q)
                .append(')')
                .toString();
    }
    /** Anfangs- und Endpunkt   */
    private Punkt p, q;
}
```


Beispiel: Klasse StreckeTest

```
/** Testprogramm fuer Strecke   */
public class StreckeTest {
   public void start() {
       Strecke s1 = new Strecke(1.0, 2.0, 3.0, 4.0);
       System.out.println("s1 = " + s1);
       s1.verschiebe(1.0,2.0);
       System.out.println("s1 = " + s1);
       System.out.println("Laenge(s1) = "
                          + s1.getLaenge());
   }

   public static void main(String args[]) {
       new StreckeTest().start();
   }
}
```

Objektbezogene Initialisierungsblöcke

- Für komplexere Initialisierung von Objekten können auch sogenannte objektbezogene Initialisierungsblöcke definiert werden.
- Diese werden als Block außerhalb von Elementfunktionen definiert und automatisch bei der Objektkonstruktion durchlaufen.
- Erst nach dem Durchlaufen aller Initialisierungsblöcke wird der Konstruktorrumpf ausgeführt.
- Empfehlung: Normalerweise sollten objektbezogene Initialisierungsblöcke nicht verwendet werden. Wenn doch, dann immer höchstens einer und der sollte bei den Konstruktoren stehen.

Objektbezogene Initialisierungsblöcke

```
public class ObjektInit {
    private int[] ia;
    private int max = 20;

    {   // Initialisierung des Arrays ia
        System.out.println("Beginn Initialisierung");
        ia = new int [max];
        for (int i = 0; i < ia.length; i++)
            ia[i] = i*i;
        System.out.println("Ende   Initialisierung");
    }

    public ObjektInit () {
        System.out.println("Beginn Konstruktor");
        for (int zahl : ia)
            System.out.printf("%d  ", zahl);
        System.out.println();
        System.out.println("Ende   Konstruktor");
    }
```

Objektbezogene Initialisierungsblöcke

```
Beginn Initialisierung
Ende   Initialisierung
Beginn Konstruktor
0  1  4  9  16  25  36  49  64  81  100  121  144  169  196
225  256  289  324  361
Ende   Konstruktor
```


Klassenattribute und Klassenmethoden

- **Klassenattribute** sind Attribute, die nur einmal je Klasse vorhanden sind, und gehören nicht zu speziellen Objekten, z.B. klassenweit eindeutige Konstanten.

```
class Zaehl {
    private static int anzObjekte; ...
}
```

- Klassenattribute werden initialisiert bevor Objekte der Klasse erzeugt werden und bevor irgendwelche Methoden der Klasse aufgerufen werden.

```
class Math {
   public static final double PI =
                                3.14159265358979323846;
 ...
```

Klassenbezogene Initialisierungsblöcke

```
/**
 * Einfache, nicht effiziente Ermittlung von Primzahlen
 *
 * @author Folz
 * @version 2015
 */
public class Primes {
    // die ersten 100 Primzahlen
    private static int[] knownPrimes = new int[100];
    private static int candidate = 3;

    static {
        System.out.println("Start static-Block");
        knownPrimes[0] = 2;
        knownPrimes[1] = 3;
        for (int i = 2; i < knownPrimes.length; ++i)
            knownPrimes[i] = nextPrime();
        System.out.println("\nEnde static-Block");
    }
```


Klassenbezogene Initialisierungsblöcke

```
    private static int nextPrime() {
        do {
            candidate += 2;
        } while (!isPrime(candidate));
        return candidate;
    }

    private static boolean isPrime(int candidate) {
        boolean prime = true;
        int i = 0;
        int divisor = knownPrimes[i];
        while (prime && divisor * divisor <= candidate) {
            if (candidate % divisor == 0) {
                prime = false;
            } else {
                divisor = knownPrimes[++i];
            }
        }
        return prime;
    }
```

Klassenbezogene Initialisierungsblöcke

```
    public static void main(String[] args) {
        System.out.println("Start main");
        System.out.println("Liste der ersten Primzahlen");
        for (int i = 0; i < knownPrimes.length; ++i) {
            System.out.printf("%4d ", knownPrimes[i]);
            if ((i+1) % 10 == 0)
                System.out.println();
        }
        System.out.println("Ende main");
    }
}
```

Das Array ist also mit dem Start von main bereits gefüllt!

Klassenbezogene Initialisierungsblöcke

```
Start static-Block

Ende static-Block
Start main
Liste der ersten Primzahlen
   2    3    5    7   11   13   17   19   23   29
  31   37   41   43   47   53   59   61   67   71
  73   79   83   89   97  101  103  107  109  113
 127  131  137  139  149  151  157  163  167  173
 179  181  191  193  197  199  211  223  227  229
 233  239  241  251  257  263  269  271  277  281
 283  293  307  311  313  317  331  337  347  349
 353  359  367  373  379  383  389  397  401  409
 419  421  431  433  439  443  449  457  461  463
 467  479  487  491  499  503  509  521  523  541
Ende main
```

Klassenmethoden

- **Klassenmethoden** sind Methoden, die nur innerhalb der Klasse operieren können, nicht aber innerhalb von einzelnen Objekten.

```
class Math {
   public static double sin (double x);
...
```

- Klassenmethoden sind nur für die Klasse aufrufbar
- Klassenmethoden haben keine `this`-Referenz
- Klassenmethoden können nur auf Klassenattribute zugreifen nicht auf objektbezogene Attribute
- Aufruf einer Klassenmethode von außerhalb

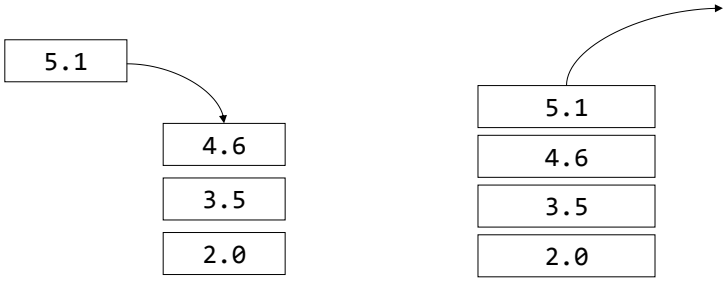
```
Primes.nextPrime (); // Primes = Klassenname
```


Beispiel: Stack (Stapel)

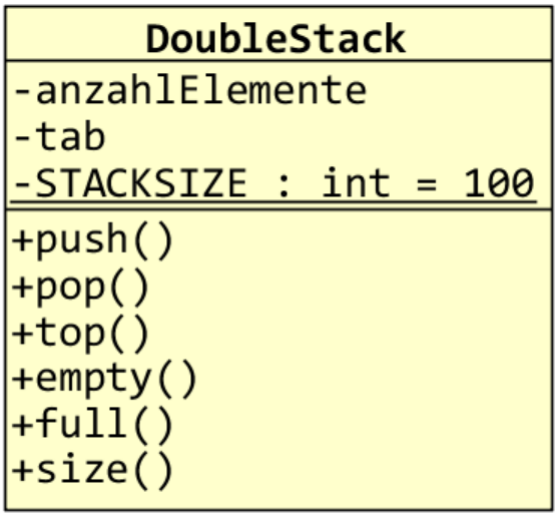
- Stacks (Stapel) sind Datenstrukturen, die Werte aufnehmen können mit der Möglichkeit, Elemente hinzuzufügen oder wegzunehmen.
- Dabei kann aber immer nur das zuletzt hinzugefügte Element wieder weggenommen werden.
- **LIFO-Prinzip (*Last in first out*)**
- Standardfunktionen eines Stack:
  - ⇨ **push**: Ein Element auf den Stack legen
  - ⇨ **pop**: Das oberste Element vom Stack wegnehmen

Beispiel: Stack (Stapel)

- **push**: Ein Element auf den Stack legen
- **pop**: Das oberste Element vom Stack wegnehmen



Beispiel: Stack (Stapel)



DoubleStack (1)

```
public class DoubleStack {
    private int anzElemente; // Anzahl Elemente des Stacks
    private double[] tab;    // Implementierungs-Array
    private static final int STACKSIZE = 100;
                             // Voreinstellung Stackgroesse
    private static final String MSG_SIZE =
                                 "Stackgröße muss > 0 sein!";
    private static final String MSG_NOT_FULL =
                                 "Stack darf nicht voll sein!";
    private static final String MSG_NOT_EMPTY =
                                 "Stack darf nicht leer sein!";

    public DoubleStack() {
        this(STACKSIZE);
    }
```


DoubleStack (2)

```
    /**
     * DoubleStack Konstruktor
     *
     * @param n Vorgabe für die Größe des internen Arrays
     */
    public DoubleStack(int n) {
        check(n > 0, MSG_SIZE);
        tab = new double[n];
        anzElemente = 0;
    }

    public boolean empty () {
        return (anzElemente == 0);
    }

    public boolean full  () {
        return (tab.length <= anzElemente);
    }
```

DoubleStack (3)

```
/**
 * Ein Element in den Stack einfuegen
 * Vorbedingung: !full()
 *
 * @param x Wert, der auf den Stack zu legen ist
 */
public void push (double x) {
    check(!full(), MSG_NOT_FULL);
    tab[anzElemente] = x;
    anzElemente++;
}

/**
 * Oberstes Element Entfernen
 * Vorbedingung: !empty()
 */
public void pop () {
    check(!empty(), MSG_NOT_EMPTY);
    anzElemente--;
}
```


DoubleStack (4)

```
/**
 * Wert des obersten Elementes zurueckgeben
 * Vorbedingung: !empty()
 * @return oberster Wert des Stacks
 */
public double top () {
    check(!empty(), MSG_NOT_EMPTY);
    return tab[anzElemente - 1];
}

public int size() {
    return anzElemente;
}

public String toString() {
    StringBuffer sb = new StringBuffer();
    for (int i = anzElemente - 1; i >= 0; --i)
        sb.append(tab[i]).append("   ");
    return sb.toString();
}
```

DoubleStackTest (1)

```
public class DoubleStackDialog {
    private Scanner input;
    private DoubleStack s1;
    private String menue;
    enum Funktion { POP, PUSH, EMPTY, FULL, SIZE, KONSTR, ENDE };

    public DoubleStackDialog() {
        input = new Scanner(System.in);
        // Menü zusammenbauen
        StringBuffer sb = new StringBuffer();
        for (Funktion f : Funktion.values()) {
            sb.append(f.ordinal()).append(": ")
              .append(f).append("; ");
        }
        sb.append(" -> ");
        menue = sb.toString();
    }
```


DoubleStackTest (2)

```
    public void start() {
        s1 = new DoubleStack();
        Funktion funktion = null;

        while (funktion != Funktion.ENDE) {
            try {
                funktion = einlesenFunktion();
                ausfuehrenFunktion(funktion);
            } catch (IllegalArgumentException e) {
                System.out.println("Ausnahme gefangen: " + e);
            } catch (java.util.InputMismatchException e) {
                System.out.println(e);
                input.next();
            } catch (Exception e) {
                System.out.println("Ausnahme gefangen: " + e);
                e.printStackTrace(System.out);
            }
        }
    }
```

DoubleStackTest (3)

```
    /**
     * Menü ausgeben und Funktion einlesen.
     *
     * @return eingelesene Funktion als ganzzahliger Wert
     */
    private Funktion einlesenFunktion() {
        System.out.print(menue);
        int eingabe = input.nextInt();
        if (eingabe >= 0 && eingabe < Funktion.values().length)
            return Funktion.values()[eingabe];
        else
            throw new IllegalArgumentException("Falsche Funktion: "
                                               + eingabe);
    }
```


DoubleStackTest (4)

```
private void ausfuehrenFunktion(Funktion funktion) {
       double wert;
       int groesse;

       switch (funktion) {
           case POP  : System.out.println("pop: " + s1.top());
                       s1.pop();
                       break;

           case PUSH : System.out.println("Wert: ");
                       wert = input.nextDouble();
                       s1.push(wert);
                       break;

           case EMPTY: System.out.println("empty: " + s1.empty());
                       break;

           case FULL : System.out.println("full: " + s1.full());
                       break;

           case SIZE : System.out.println("size: " + s1.size());
                       break;
```

DoubleStackTest (5)

```
            case KONSTR: System.out.println("Groesse: ");
                         groesse = input.nextInt();
                         s1 = new DoubleStack(groesse);
                         break;

            case ENDE : break;

            default   : System.out.println("Falsche Funktion!");
        }
        System.out.println("s1: " + s1);
    }

    public static void main (String[] args) {
        try {
            new DoubleStackDialog().start();
        } catch(Throwable e) {
            System.out.println("main: Ausnahme gefangen: " + e);
        }
    }
}
```


Speicherbereinigung und finalize

- Ein Objekt wird dann freigegeben, wenn es nicht mehr referenziert wird.
- Die Speicherbereinigung erfolgt automatisch durch den **Garbage Collector** und läuft in einem eigenen Thread im Hintergrund ab.
- Bevor jedoch ein Objekt tatsächlich freigegeben wird, wird vom Garbage-Collector die Standardmethode `finalize()`, die jede Klasse von der Klasse `java.lang.Object` erbt, aufgerufen.
- `finalize` ist seit Java 9 als **deprecated** gekennzeichnet, weil sich der Finalisierungs-mechanismus als problematisch herausgestellt hat!

Speicherbereinigung und finalize

```
public Punkt f() {
    Punkt a = new Punkt();
    Punkt b = new Punkt();
    Punkt c = new Punkt();
    Punkt d = new Punkt();
    a = null;       // erstes Objekt ist freigegeben
    b = c;          // zweites Objekt ist freigegeben
    return d;
}                   // drittes Objekt ist freigegeben
                    // viertes Objekt noch referenziert
```


Die finalize- Methode

- Die sogenannte `finalize`-Methode wird automatisch aufgerufen, bevor der Garbage Collector den Speicher für das Objekt freigibt

```
protected void finalize () throws Throwable {
    super.finalize();
    close();     // z.B. Schließen der Datei
}
```

- Regel ab Java 9: finalize soll nicht mehr benutzt werden. Statt dessen soll die Freigabe von Ressourcen mit Hilfe des Interface **`AutoCloseable`** und des **try-with-resources**-Statements gelöst werden (mehr dazu später!)

Die finalize- Methode

- Die finalize-Methode kann zu Aufräumarbeiten herangezogen werden, wie z.B. Schließen von Dateien, Unterbrechen von Netzverbindungen usw. Da es leider keine Garantie des Aufrufes gibt, empfiehlt sich die folgende Strategie.

```
private boolean cleanedup = false;
public void cleanup() {/* Notwendige Bereinigungen */
}
public void dispose() {
    cleanup();
    cleanedup = true;
}
protected void finalize() throws Throwable {
    super.finalize();
    if (!cleanedup)
         cleanup();
}
```


Testprogramm für finalize

```
public class FinalizeTest {
    private byte[] btab;
    private int objektId = 0;
    private static int objektAnzahl = 0;

    public FinalizeTest() {
        btab = new byte[2_000_000];
        objektId = ++objektAnzahl;
        System.out.println("Objekt " + objektId);
    }

    protected void finalize() throws Throwable {
       super.finalize();
       System.out.println("Objekt " + objektId
                                    + " freigeben");
    }
```

Testprogramm für finalize

```
    public static void main(String args[]) {
        FinalizeTest f;
        for (int i = 0; i < 20; i++) {
            f = new FinalizeTest();
        }
    }
}
```


Klasse Ratio: Rationalzahlen

- Klasse `Ratio` für den Umgang mit rationalen Zahlen entwickelt. Die Klasse sollte folgende Möglichkeiten bieten:
- Arithmetik
  Addition, Subtraktion, Multiplikation, Division
- Vergleiche
  Gleichheit, Größer oder Kleiner
- Attribute

```
private long zaehler, nenner  = 1L;
```

Klasse Ratio: Rationalzahlen

- Für Ratio-Objekte sollen immer folgende Bedingungen erfüllt sein:
  - Zähler und Nenner müssen gekürzt sein
  - Der Nenner ist positiv (> 0)
  - Die Zahl 0 wird durch 0/1 dargestellt
- Konstruktor
  - ⇨ Man braucht folgende Möglichkeiten der Konstruktion:

```
Ratio a = new Ratio(1, 2);     // 1/2
Ratio b = new Ratio(2);        // 2/1
Ratio c = new Ratio();         // 0/1
```


Klasse Ratio: Konstruktoren

```
public Ratio(long zaehler, long nenner) {
    if (nenner == 0)
        throw new ArithmeticException(MSG_NENNER_NULL);
    if (nenner > 0) {
        this.zaehler = zaehler;
        this.nenner  = nenner;
    } else {
        this.zaehler = -zaehler;
        this.nenner  = -nenner;
    }
    kuerzen();
}
public Ratio (long n) {
    this(n, 1L);
}
public Ratio() {
    this(0L, 1L);
}
```

Klasse Ratio: Kürzen

```
private Ratio kuerzen () {
    long teiler = ggT(zaehler, nenner);
    if (teiler > 1) {
        zaehler = zaehler / teiler;
        nenner  = nenner  / teiler;
    }
    return this;
}

private static long ggT(long m, long n)  {
    m = Math.abs(m);
    n = Math.abs(n);
    long r = m % n;
    while (r > 0){
        m = n;
        n = r;
        r = m % n;
    }
    return n;
}
```


Klasse Ratio: Kopierkonstruktor

```
/**
 * Kopierkonstruktor
 *
 * @param r  Konstruiere Ratio-Objekt als Kopie
 *           eines übergebenen Ratio-Objektes
 */
public Ratio (Ratio r) {
    zaehler = r.zaehler;
    nenner  = r.nenner;
}
```

Klasse Ratio: Addition 1. Version

```
public Ratio add (Ratio r) {
    Ratio tmp = new Ratio(this);
    tmp.zaehler = tmp.zaehler * r.nenner
                + tmp.nenner  * r.zaehler;
    tmp.nenner  = tmp.nenner  * r.nenner;
    tmp.kuerzen();
    return tmp;
}
```


Klasse Ratio: Addition 2. Version

```
public Ratio add (Ratio r) {
    return new Ratio(zaehler * r.nenner
                   + nenner  * r.zaehler,
                     nenner  * r.nenner);
}
```

Klasse Ratio: Multiplikation/Division

```
public Ratio multiply (Ratio r) {
    return new Ratio(zaehler * r.zaehler,
                     nenner  * r.nenner);
}

public Ratio divide (Ratio r) {
    return new Ratio(zaehler * r.nenner,
                     nenner  * r.zaehler);
}
```


Klasse Ratio: Gleichheit

```
public boolean equals (Object o) {
    Ratio r;
    if (o instanceof Ratio) {
        r = (Ratio)o;
        return nenner  == r.nenner
        && zaehler == r.zaehler;
    } else
        return false;
}
```

Klasse Ratio: Größenvergleich (1)

```
public int compareTo (Ratio r) {
    Ratio tmp = this.subtract(r);
    if (tmp.zaehler < 0)
        return -1;
    else if (tmp.zaehler > 0)
        return 1;
    else
        return 0;
}
```


Klasse Ratio: Größenvergleich (2)

```
/**
 * Groessenvergleich
 *  @param ein anderes Ratio-Objekt, mit dem
 *         verglichen wird
 *  @return liefert 0 bei Gleichheit, kleiner 0, wenn
 *          der erste Bruch kleiner ist und groesser
 *          Null sonst
 */
public int compareTo (Ratio r) {
    long tmp = zaehler * r.nenner
             - nenner * r.zaehler;
    if (tmp < 0)
        return -1;
    else if (tmp > 0)
        return 1;
    else
        return 0;
}
```