Materialien zu den zentralen Abiturprüfungen im Fach Informatik

Grundkurs / Leistungskurs
(ab 2018)

Inhaltsverzeichnis

1. Vorwort

Grundlage für die zentral gestellten schriftlichen Aufgaben der Abiturprüfung sind in allen Fächern die Kernlehrpläne für die gymnasiale Oberstufe (Kernlehrplan für die Sekundarstufe II – Gymnasium/Gesamtschule in Nordrhein-Westfalen, Frechen 2013). Die im jeweiligen Kernlehrplan in Kapitel 2 festgeschriebenen Kompetenzbereiche (Prozesse) und Inhaltsfelder (Gegenstände) sind für den Unterricht obligatorisch. In der Abiturprüfung werden daher grundsätzlich alle Kompetenzen vorausgesetzt, die der Lehrplan für das Ende der Qualifikationsphase vorsieht.
Das hier vorliegende Dokument soll dazu dienen, die Vorgaben zu präzisieren und zu fokussieren, um Fehlinterpretationen zu vermeiden, Fragen zu klären und weitere fachliche Hinweise und Materialien zur unterrichtlichen Vorbereitung der Schülerinnen und Schüler auf die zentrale Abiturprüfung an die Hand zu geben.[1]

Michael Hypius, Fachkoordinator für Informatik

Änderungen (Changelog 25.10.2021)

Das vorliegende Dokument entspricht inhaltlich den vorangegangenen Dokumenten zu den zentralen Prüfungen (Dokumentation für den Grundkurs / Leistungskurs) ab dem Abitur 2018. Es wurden lediglich **redaktionelle** Überarbeitungen durchgeführt und Abschnitte zu Automaten und Datenschutz **integriert**, die bisher separat verfügbar waren.

1. Redaktionelle Überarbeitungen:
   - Deckblatt, Inhaltsverzeichnis, Vorwort und Changelog hinzugefügt
   - Einige Abbildungen im Abschnitt zu Klassendiagrammen hinzugefügt
   - Formatierung der Klassendokumentationen angepasst
   - Fehlerkorrekturen und Optimierungen von Abbildungen, Formatierungen und Formulierungen

2. Aus anderen Dokumenten zum Zentralabitur integriert:
   - Abschnitt zu endlichen Automaten
   - Abschnitt zu Kellerautomaten
   - Abschnitt zu Grundprinzipien des Datenschutzes

[1] Das vorliegende Dokument unterscheidet nicht zwischen Inhalten für den Grundkurs und Inhalten für den Leistungskurs. Diese sind den jeweils aktuellen Zentralabiturvorgaben zu entnehmen.

2. Klassendiagramme

Klassendiagramme beschreiben die vorhandenen Klassen mit ihren Attributen und Methoden sowie die Beziehungen der Klassen untereinander. Dabei ist zwischen Implementationsdiagrammen und Entwurfsdiagrammen zu unterscheiden.

Im Folgenden soll ein Computerspiel in Teilen modelliert werden, in dem ein Wurm, der aus mehreren Elementen bzw. Gliedern besteht, über ein Spielfeld gesteuert wird.

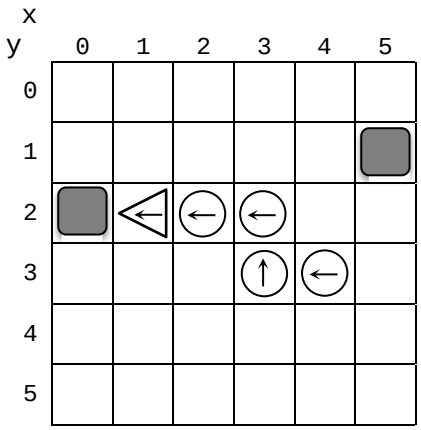


Abbildung 1: Beispiel für ein Spielfeld mit Wurm

Klassen
Klassen werden durch Rechtecke dargestellt, die entweder nur den Namen der Klasse tragen oder zusätzlich auch Attribute und / oder Methoden enthalten. Attribute und Methoden können zusätzliche Angaben zu Parametern und Sichtbarkeit (`public` (+), `private` (-)) besitzen.

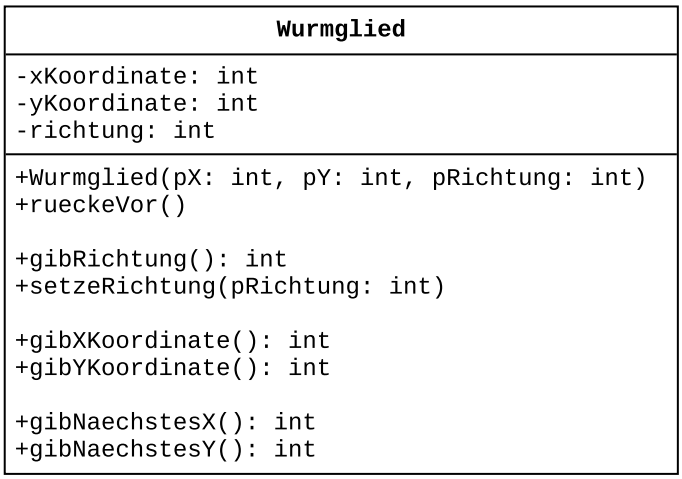


Abbildung 2: Beispielhafte Darstellung einer Klasse als Implementationsdiagramm

Bei *abstrakten Klassen*, also Klassen, von denen kein Objekt erzeugt werden kann, wird unter den Klassennamen im Diagramm rechtsbündig `{abstract}` geschrieben. Abstrakte Methoden, also Methoden, für die keine Implementierungen angegeben werden und die nicht aufgerufen werden können, werden in Kursivschrift dargestellt. Bei einer handschriftlichen Darstellung werden sie mit einer Wellenlinie unterschlängelt.

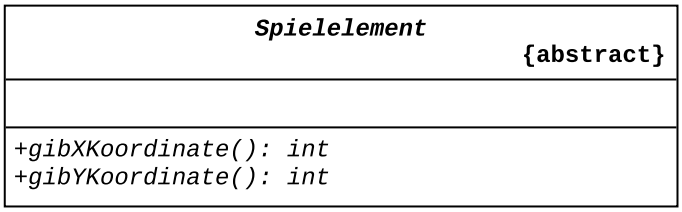


Abbildung 3: Beispieldarstellung einer abstrakten Klasse

Bei Klassen, die generische Datentypen verwenden (vgl. Abschnitt 7), werden der Typparameter und seine Belegung in einem mit gestrichelter Linie umrandeten Feld am oberen rechten Ende der Klassendarstellung angegeben.

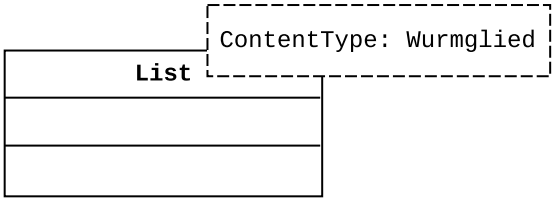


Abbildung 4: Beispieldarstellung einer Klasse mit generischem Datentyp

Schnittstellen

Ein *Interface* (Schnittstelle) enthält nur die Deklarationen von Methoden, nicht aber deren Implementation. Die Implementation einer Schnittstelle wird durch eine gestrichelte Linie mit geschlossener, nicht ausgefüllter Pfeilspitze dargestellt.

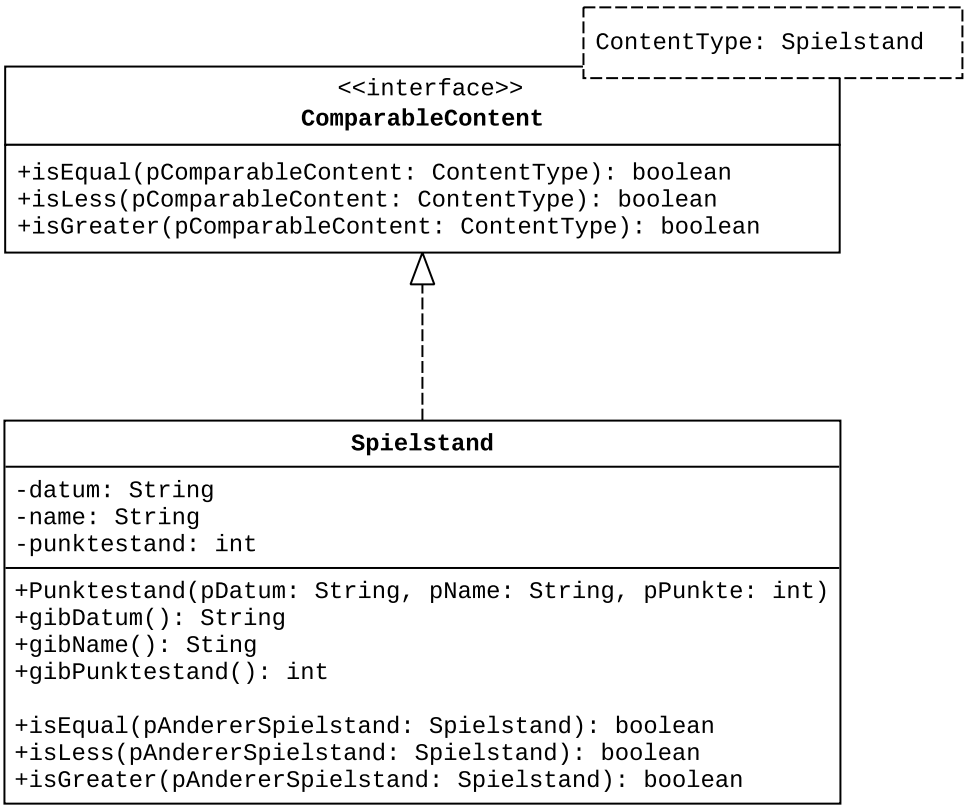


Abbildung 5: Beispieldarstellung einer Schnittstelle mit implementierender Klasse

Beziehungen zwischen Klassen

Assoziation

Eine gerichtete Assoziation von einer Klasse `A` zu einer Klasse `B` modelliert, dass Objekte der Klasse `B` in einer Zugriffsbeziehung zu Objekten der Klasse `A` stehen bzw. stehen können. Anders ausgedrückt: Objekte der Klasse A können Objekte der Klasse B verwalten.
Bei einer Assoziation kann man angeben, wie viele Objekte der Klasse `B` in einer solchen Zugriffsbeziehung zu einem Objekt der Klasse `A` stehen bzw. stehen können. Die Zahl nennt man *Multiplizität*.

Beispiele für Multiplizitäten:

- `1` genau ein assoziiertes Objekt
- `0..1` kein oder ein assoziiertes Objekt
- `0..*` beliebig viele assoziierte Objekte
- `1..*` mindestens ein, beliebig viele assoziierte Objekte

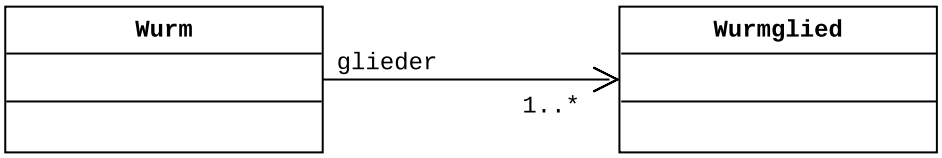


Abbildung 6: Beispieldarstellung einer Assoziation mit Multiplizität

Ein Objekt der Klasse `Wurm` steht zu mindestens einem oder beliebig vielen Objekten der Klasse `Wurmglied` in Beziehung.

Vererbung

Die Vererbung beschreibt die Beziehung zwischen einer allgemeineren Klasse (Oberklasse) und einer spezialisierten Klasse (Unterklasse). Der Unterklasse stehen alle öffentlichen (`public`) Attribute und Methoden der Oberklasse zur Verfügung. In der Unterklasse können Attribute und Methoden ergänzt oder auch Methoden der Oberklasse überschrieben werden.

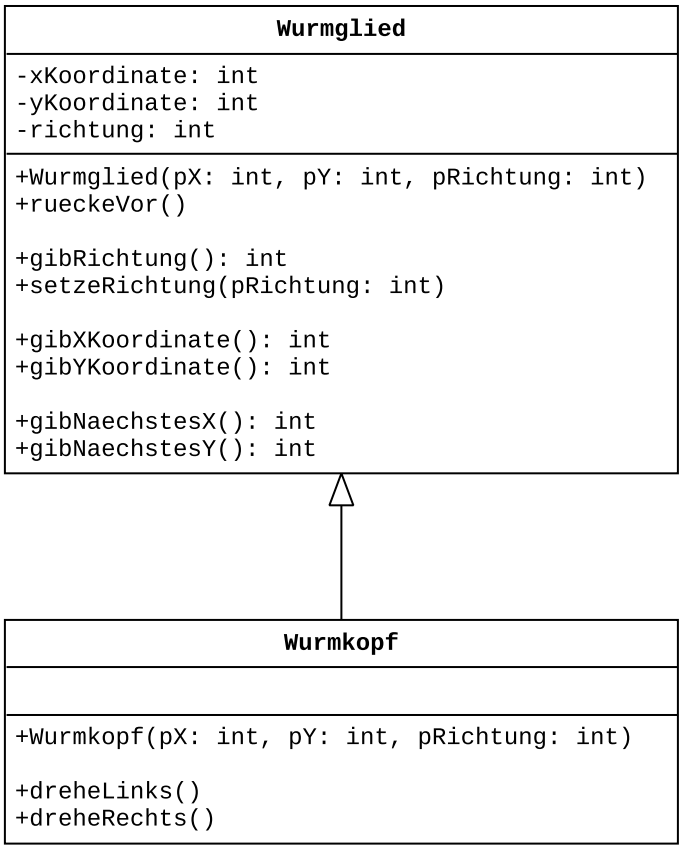


Abbildung 7: Die Klasse `Wurmkopf` spezialisiert die Klasse `Wurmglied`.

Diagrammtyp: Entwurfsdiagramm

Bei einem ersten groben Entwurf werden zunächst die in der Auftragssituation vorkommenden Objekte identifiziert und ihren Klassen zugeordnet.

Das Entwurfsdiagramm enthält Klassen und ihre Beziehungen mit Multiplizitäten. Als Beziehungen können Vererbung und gerichtete Assoziationen gekennzeichnet werden. Gegebenenfalls werden wesentliche Attribute und / oder Methoden angegeben.

Die Darstellung ist programmiersprachenunabhängig ohne Angabe eines konkreten Datentyps, es werden lediglich `Zahl`, `Text`, `Wahrheitswert` und `Datensammlung<...>` unterschieden. Bei der Datensammlung steht in Klammern der Datentyp oder die Klassenbezeichnung der Elemente, die dort verwaltet werden.

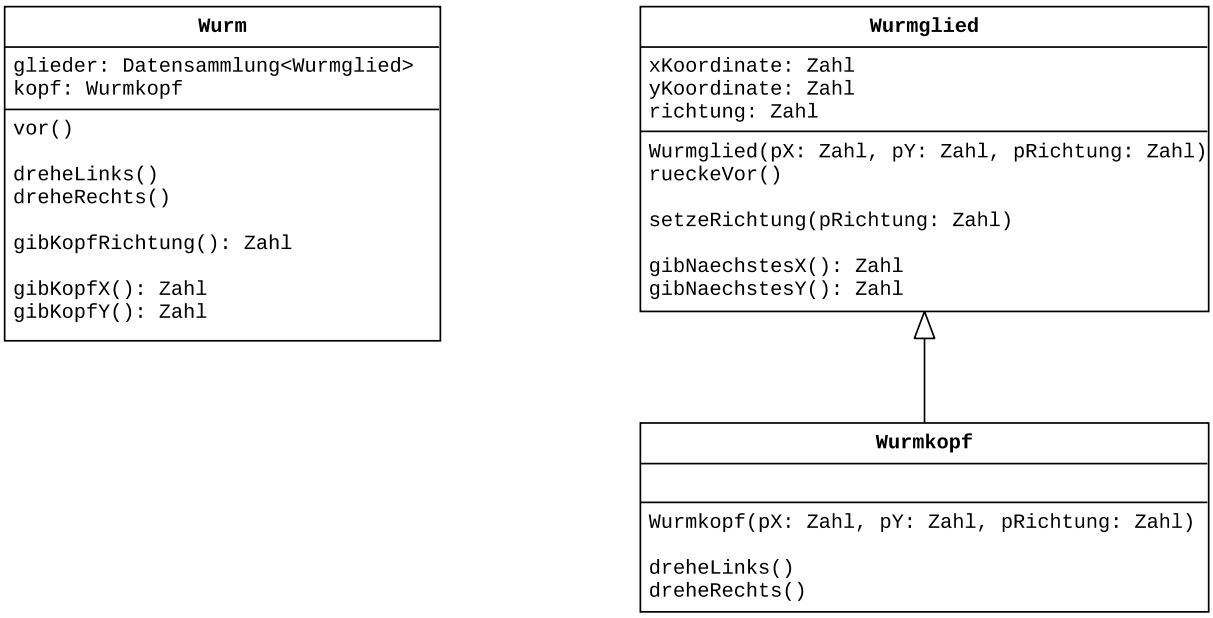


Abbildung 8: Beispiel für ein Entwurfsdiagramm ohne Assoziationspfeile

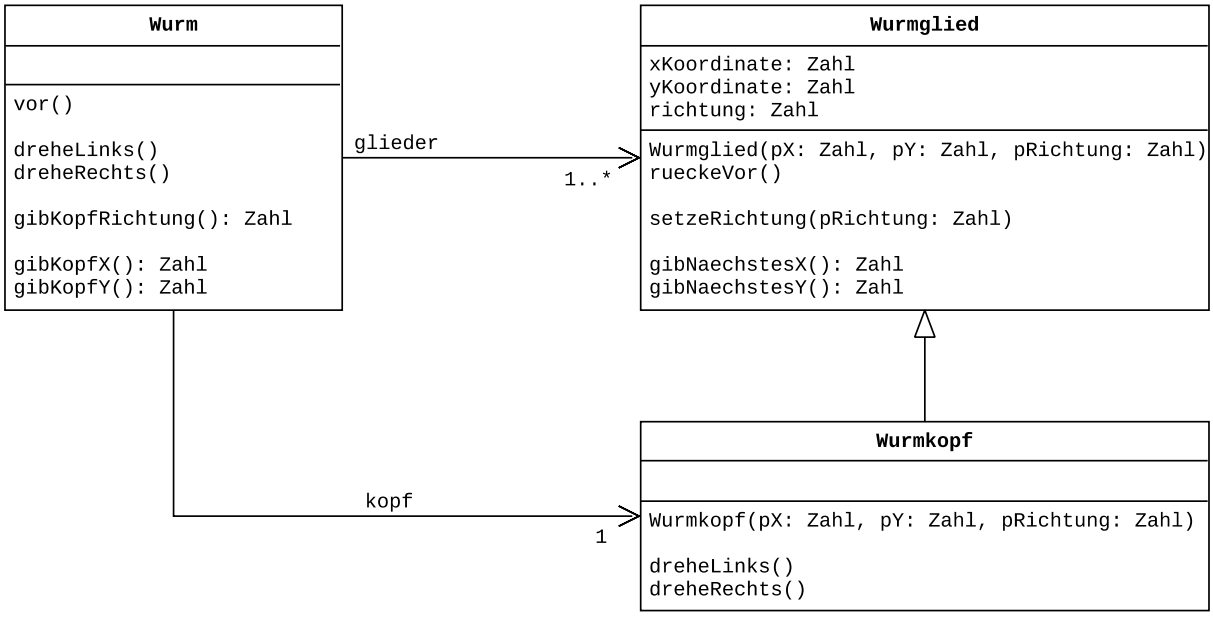


Abbildung 9: Beispiel für ein Entwurfsdiagramm mit Assoziationspfeilen

Beide Darstellungen (Abbildung 8, Abbildung 9) drücken den gleichen Sachverhalt aus.

Diagrammtyp: Implementationsdiagramm

Ein Implementationsdiagramm ergibt sich durch Präzisierung eines Entwurfsdiagramms und orientiert sich stärker an der verwendeten Programmiersprache. Für die im Entwurfsdiagramm angegebenen Datensammlungen werden konkrete Datenstrukturen gewählt, deren Inhalte ggf. über einen Typparameter spezifiziert werden. Die Attribute werden mit den in der Programmiersprache (hier Java) verfügbaren Datentypen versehen und die Methoden mit Parametern einschließlich ihrer Datentypen.

Bei den für das Zentralabitur dokumentierten Klassen (`List`, `BinaryTree`, …) wird auf die Angabe der Attribute und der Methoden verzichtet. Auf die Angabe von Multiplizitäten wird zugunsten von konkreten Bezeichnern und Datenstrukturen verzichtet.

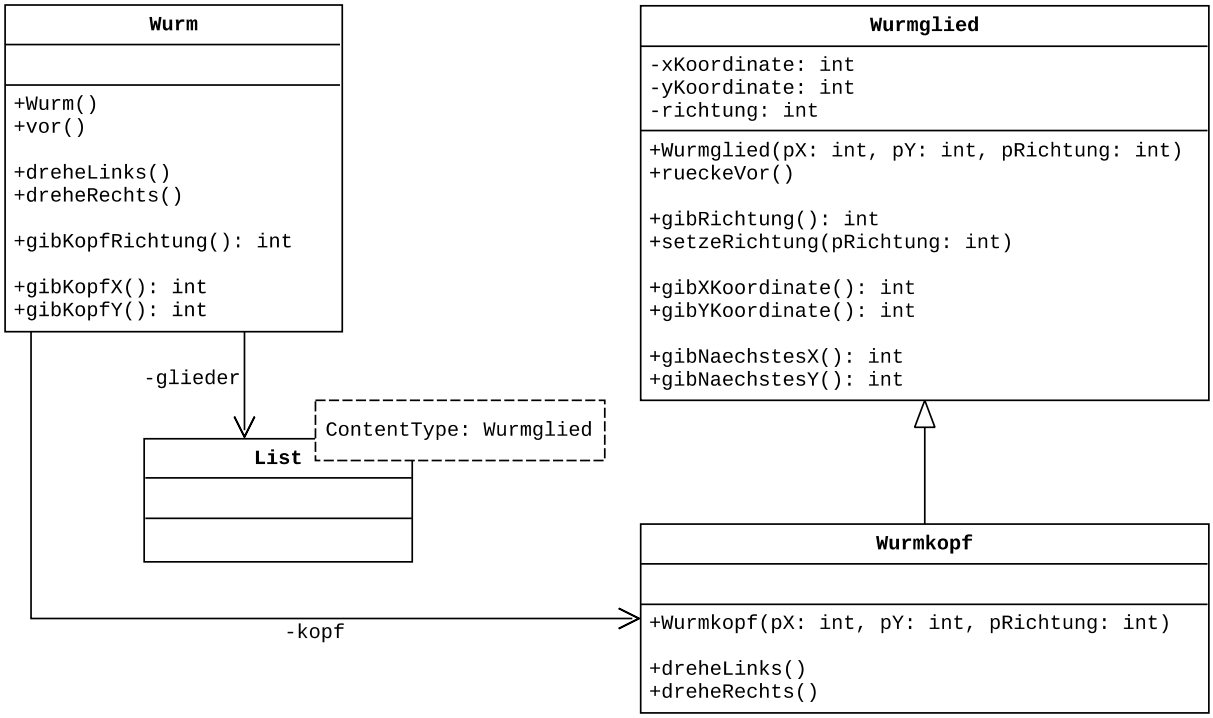


Abbildung 10: Mögliches Implementationsdiagramm zu den Entwurfsdiagrammen in den Abbildungen 8 und 9

Erläuterung:

Die Bezeichner von Attributen, durch die Assoziationen realisiert werden, stehen an den Pfeilen. Bei der Klasse `List` handelt es sich um eine generische Klasse, über einen Typparameter wird der Datentyp der Inhaltsobjekte angegeben. In der Klasse `List` werden Objekte der Klasse `Wurmglied` verwaltet.

3. Elemente von Java

Die folgenden Elemente von Java werden für die zentralen Abiturprüfungen vorausgesetzt. Kenntnisse über weitere javaspezifische Klassen insbesondere zur Gestaltung einer grafischen Benutzungsoberfläche sind für die Bearbeitung der Aufgaben im Zentralabitur nicht erforderlich.

Sprachelemente

- Klasse
- Assoziation
- Vererbung
- Schnittstelle (Interface)
- Attribute und Methode (mit Parametern und Rückgabewerten)
- Wertzuweisung
- Verzweigungen (`if`, `if … else`, `switch`)
- Schleifen (`while`, `for`, `do … while`)

Datentypen

Datentyp	Operationen	Methoden

`int`
Klasse `Integer`
	`+ -`
	`* /`
	`%`
	`< >`
	`<= >=`
	`== !=`
		statische Methoden der Klasse `Math`:
		`abs(int a)`
		`min(int a, int b)`
		`max(int a, int b)`
		
		statische Konstanten der Klasse `Integer`:
		`MIN_VALUE, MAX_VALUE`
		
		statische Methoden der Klasse `Integer`:
		`toString(int i)`
		`parseInt(String s)`

`double`
Klasse `Double`
	`+ -`
	`* /`
	`< >`
	`<= >=`
	`== !=`
		statische Methoden der Klasse `Math`:
		`abs(double a)`
		`min(double a, double b)`
		`max(double a, double b)`
		`sqrt(double a)`
		`pow(double a, double b)`
		`round(double a)`
		`random()`
		
		statische Konstanten der Klasse `Double`:
		`NaN, MIN_VALUE, MAX_VALUE`
		
		statische Methoden der Klasse `Double`:
		`toString(double d)`
		`parseDouble(String s)`
		`isNaN(double a)`

`boolean`
Klasse `Boolean`
	`&& ||`
	`!`
	`== !=`
		statische Methoden der Klasse `Boolean`:
		`toString(boolean b)`
		`parseBoolean(String s)`

`char`
Klasse `Character`
	`< >`
	`<= >=`
	`== !=`
		statische Methoden der Klasse `Character`:
		`toString(char c)`

Klasse `String`		Methoden der Klasse `String`
		`length()`
		`indexOf(String str)`
		`substring(int beginIndex)`
		`substring(int beginIndex,`
		`int endIndex)`
		`charAt(int index)`
		`equals(Object anObject)`
		`compareTo(String anotherString)`
		`startsWith(String prefix)`

Statische Strukturen
Ein- und zweidimensionale Felder (Arrays) von einfachen Datentypen und Objekten

4. Datenbankabfragen in SQL

```
SELECT (DISTINCT)
FROM
WHERE
GROUP BY
ORDER BY
ASC, DESC
(LEFT / RIGHT / INNER) JOIN … ON
UNION
AS
NULL
```

Vergleichsoperatoren: `=, <>, >, <, >=, <=, LIKE, BETWEEN, IN, IS NULL`
Arithmetische Operatoren: `+, -, *, /, (…)`
Logische Verknüpfungen: `AND, OR, NOT`
Funktionen: `COUNT, SUM, MAX, MIN, AVG`

(Geschachtelte) SQL-Abfragen über eine und mehrere verknüpfte Relationen

5. Grundprinzipien des Datenschutzes

Unter Datenschutz versteht man den Schutz des Rechts *auf informationelle Selbstbestimmung*, d. h. des Rechts jeder Person, über die Preisgabe und Verwendung seiner eigenen personenbezogenen Daten selbst zu bestimmen.

Personenbezogene Daten sind solche Daten, die sich einer Person eindeutig zuordnen lassen. Es spielt dabei keine Rolle, ob diese Zuordnung jedermann möglich ist oder ob zur eindeutigen Zuordnung weitere Hilfsinformationen benötigt werden. Daten, die keiner speziellen Person zuzuordnen sind, fallen nicht unter den Datenschutz, wobei es andere Gründe geben kann, sie nicht allgemein zugänglich zu machen.

Die folgenden Grundprinzipien des Datenschutzes basieren auf der *Europäischen Datenschutzgrundverordnung* und liegen hier in reduzierter Form vor. Sie erheben in dieser Formulierung keinen Anspruch auf juristische Verbindlichkeit.

Verbot mit Erlaubnisvorbehalt
Die Verarbeitung, d. h. zum Beispiel die Erhebung, Speicherung, Weitergabe oder allgemeine Verwendung personenbezogener Daten ist grundsätzlich verboten – es sei denn, die betroffene Person hat der Verarbeitung für einen konkreten Zweck zugestimmt oder es gibt eine explizite gesetzliche Regelung, die eine Verarbeitung für einen konkreten Zweck erlaubt.

Datenminimierung
Die Verarbeitung personenbezogener Daten ist an dem Ziel auszurichten, so wenig personenbezogene Daten wie möglich zu verarbeiten. Insbesondere sind personenbezogene Daten nach Möglichkeit zu anonymisieren bzw. zu pseudonymisieren. Sie sind zu löschen, sobald sie nicht mehr benötigt werden.

Zweckbindung
Personenbezogene Daten dürfen nur zu einem konkreten Zweck erhoben und verarbeitet werden. Eine weitergehende Verarbeitung zu einem anderen Zweck ist in der Regel nicht erlaubt.

Transparenz
Werden personenbezogene Daten verarbeitet, so ist die betroffene Person, sofern sie nicht bereits auf andere Weise Kenntnis erlangt hat, darüber zu informieren. Dazu gehören unter anderem Informationen zum Zweck der Erhebung, zur erhebenden Stelle bzw. Institution und zu der Frage, wie lange die Daten gespeichert werden. Ausnahmen gibt es z. B. im Bereich der Strafverfolgung.

Erforderlichkeit
Personenbezogene Daten dürfen nur dann verarbeitet werden, wenn sie für den Zweck, zu dem sie verarbeitet werden, wirklich benötigt werden oder die Aufgabenerfüllung der verantwortlichen Stelle zumindest erheblich erleichtert wird.

6. Automaten

Endliche Automaten

Ein deterministischer endlicher Automat (DEA) erhält ein Wort als Eingabe und erkennt (oder akzeptiert) dieses oder nicht. Häufig wird er daher auch als Akzeptor bezeichnet. Die Menge der akzeptierten Wörter bildet die durch den Automaten dargestellte oder definierte Sprache. Die von endlichen Automaten erkannten Sprachen sind *regulär*.

Definition

Ein deterministischer endlicher Automat wird durch ein 5-Tupel (Z, A, d, q_0, E) spezifiziert:

- Z ist eine endliche, nicht leere Menge von Zuständen.
- A ist das Eingabealphabet, eine endliche, nicht leere Menge von Symbolen.
- $d: Z \times A \to Z$ ist die Zustandsübergangsfunktion.
- $q_0 \in Z$ ist der Anfangszustand.
- $E \subseteq Z$ ist die Menge der Endzustände.

Die Zustandsübergangsfunktion kann durch einen Übergangsgraphen oder eine Tabelle dargestellt werden. Sie kann dabei partiell angegeben werden. Alle nicht angegebenen Übergänge führen dann in einen Fehlerzustand.

Verdeutlichung an einem Beispiel

Genau die Binärzahlen, die durch vier teilbar sind, sollen durch einen deterministischen endlichen Automaten akzeptiert werden. Solche Binärzahlen enden auf zwei Nullen.

Deterministischer endlicher Automat M = (Z, A, d, q_0, E)

Zustandsmenge:	$Z = \{q_0, q_1, q_2\}$
Eingabealphabet:	$A = \{0, 1\}$
Anfangszustand:	q_0
Menge der Endzustände:	$E = \{q_2\}$

Die Übergangsfunktion d wird mit Hilfe eines Zustandsübergangsgraphen visualisiert.

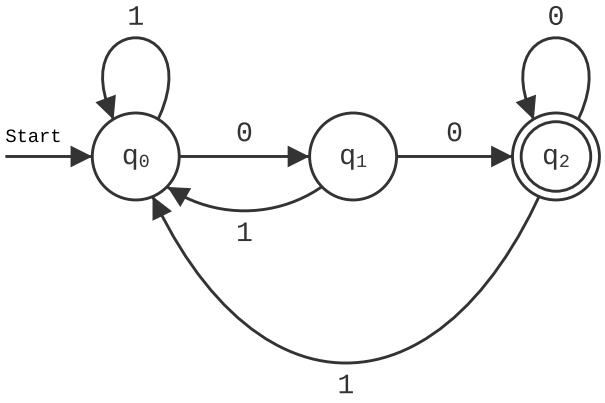


Abbildung 11: Zustandsübergangsgraph zu M

Ein nichtdeterministischer endlicher Automat (NEA) ist ein endlicher Automat, bei dem es in jedem Zustand für jedes Zeichen des Eingabealphabets mehr als einen Übergang geben kann. Außerdem sind sogenannte ε-Übergänge möglich, bei denen ohne Abarbeitung eines Zeichens der Eingabe in einen anderen Zustand gewechselt werden kann.

Definition
Ein nichtdeterministischer endlicher Automat wird durch ein 5-Tupel (Z, A, d, q_0, E) spezifiziert:

- Z ist eine endliche, nicht leere Menge von Zuständen.
- A ist das Eingabealphabet, eine endliche, nicht leere Menge von Symbolen.
- d ist die Zustandsübergangsfunktion, die jeder Kombination aus Zustand und Eingabesymbol eine Menge von Folgezuständen zuordnet.
  Formal ausgedrückt:
  $d: Z \times (A \cup \{\varepsilon\}) \rightarrow P(Z)$
- $q_0 \in Z$ ist der Anfangszustand.
- $E \subseteq Z$ ist die Menge der Endzustände.

Auch hier kann die Zustandsübergangsfunktion durch einen Übergangsgraphen oder eine Tabelle dargestellt werden.

Verdeutlichung an einem Beispiel
Der nichtdeterministische endliche Automat N erkennt dieselbe Sprache wie der deterministische endliche Automat M.

Nichtdeterministischer endlicher Automat N = (Z, A, d, q_0, E)

Zustandsmenge:	$Z = \{q_0, q_1, q_2\}$
Eingabealphabet:	$A = \{0, 1\}$
Anfangszustand:	q_0
Menge der Endzustände:	$E = \{q_2\}$

Die Übergangsfunktion d wird mit Hilfe eines Zustandsübergangsgraphen visualisiert.

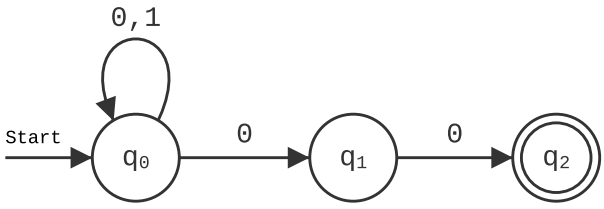


Abbildung 12: Zustandsübergangsgraph zu N

Kellerautomaten

Der Kellerautomat[2] ist ein endlicher Automat, der um einen Kellerspeicher (Stapel) erweitert wurde. Die möglichen Aktionen eines Kellerautomaten hängen nicht nur vom Zustand und den gelesenen Eingabezeichen ab, sondern auch vom obersten Kellersymbol. Jeder Schritt kann durch ein Tripel aus Zustand, Eingabesymbol und Kellerzeichen beschrieben werden.

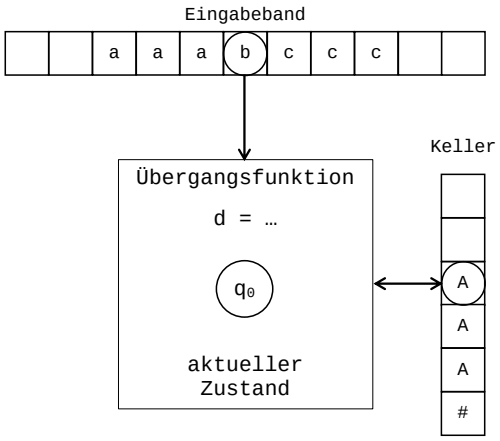


Abbildung 13: Schematische Darstellung eines Kellerautomaten

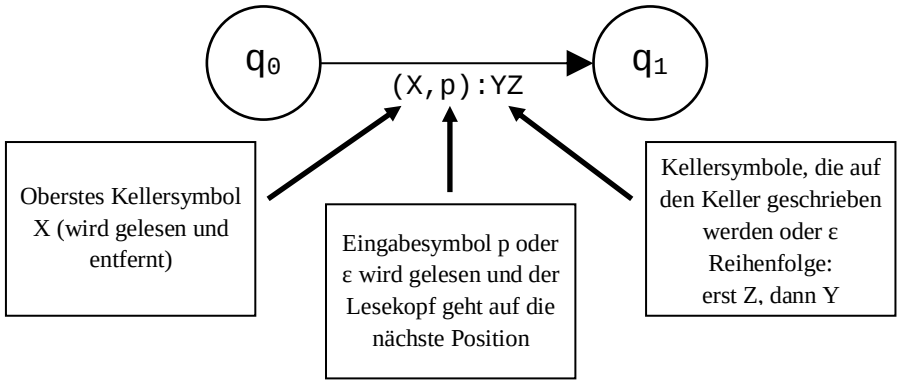
Definition eines Kellerautomaten

Ein nichtdeterministischer Kellerautomat (NKA) ist ein 7-Tupel (Z, A, K, d, q_0, #, E)

- Z ist die Zustandsmenge, eine nicht leere Menge von Zuständen.
- A ist das Eingabealphabet, eine endliche, nicht leere Menge von Symbolen.
- K ist das Kelleralphabet, eine endliche, nicht leere Menge von Symbolen.
- d ist die Übergangsfunktion, die jeder Kombination aus Zustand, Kellersymbol und Eingabesymbol eine Menge von Kombinationen aus Folgezustand und einer (ggf. leeren) endlichen Folge von Kellersymbolen zuordnet.
  Formal ausgedrückt:
  $d: Z \times K \times (A \cup \{\varepsilon\}) \rightarrow P_e(Z \times K^*)$ mit P_e = Menge aller endlichen Teilmengen.
- q_0 ist der Anfangszustand; q_0 ist ein Element aus Z.
- # ist das Kellerstartsymbol; # ist ein Element aus K.
- E ist die Menge der Endzustände, eine Teilmenge von Z.

Der Automat terminiert, wenn er sich nach Abarbeitung des Eingabewortes in einem Endzustand befindet, unabhängig von der dann aktuellen Kellerbelegung.

Grafische Darstellung von Zustandsübergängen:



Anmerkung:

ε ist kein Symbol des Eingabe-/Kelleralphabets, sondern kennzeichnet das leere Wort.

[2] Es werden nur nichtdeterministische Kellerautomaten (NKA) betrachtet.

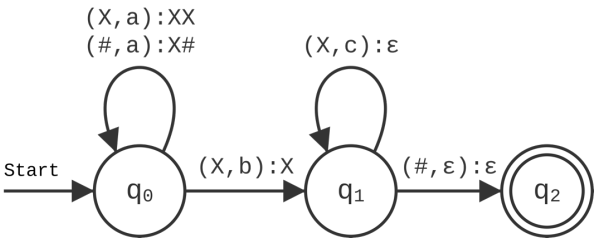
Verdeutlichung an einem Beispiel

Genau die Worte der Form $a^n bc^n$ (z. B. aaabccc) sollen von einem NKA akzeptiert werden.

Kellerautomat $M = (Z, A, K, d, q_0, \#, E)$

Zustandsmenge:	$Z = \{q_0, q_1, q_2\}$
Eingabealphabet:	$A = \{a, b, c\}$
Kelleralphabet:	$K = \{X, \#\}$
Startzustand:	q_0
Kellerstartsymbol:	#
Menge der Endzustände:	$E = \{q_2\}$

Die Übergangsfunktion d kann mit Hilfe eines Zustandsübergangsgraphen visualisiert oder tabellarisch dargestellt werden:



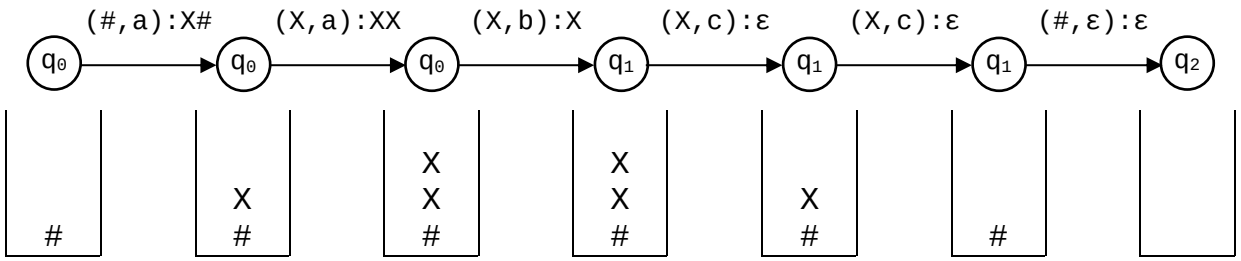
aktueller **Zustand**	gelesenes **Eingabe-symbol**	oberstes **Keller-symbol** (wird entfernt)	**Folge-zustand**	zu schreibende **Keller-symbole**
q_0	a	#	q_0	X#
q_0	a	X	q_0	XX
q_0	b	X	q_1	X
q_1	c	X	q_1	ε
q_1	ε	#	q_2	ε

Abbildung 14: Übergangsfunktion d als Zustandsübergangsgraph und als Zustandsübergangstabelle

Worterkennung durch den Kellerautomaten M

Erkennung des Wortes aabcc in grafischer Darstellung:

Beispiel:



Erkennung des Wortes `aabcc` in tabellarischer Darstellung:

Zustand	gelesenes Eingabe-symbol	Keller-inhalt	Neuer Keller-inhalt	Neuer Zustand
q_0	a	#	X #	q_0
q_0	a	X #	X X #	q_0
q_0	b	X X #	X X #	q_1
q_1	c	X X #	X #	q_1
q_1	c	X #	#	q_1
q_1		#		q_2

7. Klassendokumentationen

In Java werden Objekte über Referenzen verwaltet, d. h., eine Variable `pObject` der Klasse `Object` enthält eine Referenz auf das entsprechende Objekt. Zur Vereinfachung der Sprechweise werden jedoch im Folgenden die Referenz und das referenzierte Objekt sprachlich nicht unterschieden.

Lineare Strukturen

Die generische Klasse `Queue`

Objekte der generischen Klasse `Queue` (Schlange) verwalten beliebige Objekte vom Typ `ContentType` nach dem First-In-First-Out-Prinzip, d. h., das zuerst abgelegte Objekt wird als erstes wieder entnommen. Alle Methoden haben eine konstante Laufzeit, unabhängig von der Anzahl der verwalteten Objekte.

Dokumentation der Klasse `Queue<ContentType>`

`Queue()`

Eine leere Schlange wird erzeugt. Objekte, die in dieser Schlange verwaltet werden, müssen vom Typ `ContentType` sein.

`boolean isEmpty()`

Die Anfrage liefert den Wert `true`, wenn die Schlange keine Objekte enthält, sonst liefert sie den Wert `false`.

`void enqueue(ContentType pContent)`

Das Objekt `pContent` wird an die Schlange angehängt. Falls `pContent` gleich `null` ist, bleibt die Schlange unverändert.

`void dequeue()`

Das erste Objekt wird aus der Schlange entfernt. Falls die Schlange leer ist, wird sie nicht verändert.

`ContentType front()`

Die Anfrage liefert das erste Objekt der Schlange. Die Schlange bleibt unverändert. Falls die Schlange leer ist, wird `null` zurückgegeben.

Die generische Klasse `Stack`

Objekte der generischen Klasse `Stack` (Keller, Stapel*)* verwalten beliebige Objekte vom Typ `ContentType` nach dem Last-In-First-Out-Prinzip, d. h., das zuletzt abgelegte Objekt wird als erstes wieder entnommen. Alle Methoden haben eine konstante Laufzeit, unabhängig von der Anzahl der verwalteten Objekte.

Dokumentation der Klasse `Stack<ContentType>`

`Stack()`

Ein leerer Stapel wird erzeugt. Objekte, die in diesem Stapel verwaltet werden, müssen vom Typ `ContentType` sein.

`boolean isEmpty()`

Die Anfrage liefert den Wert `true`, wenn der Stapel keine Objekte enthält, sonst liefert sie den Wert `false`.

`void push(ContentType pContent)`

Das Objekt `pContent` wird oben auf den Stapel gelegt. Falls `pContent` gleich `null` ist, bleibt der Stapel unverändert.

`void pop()`

Das zuletzt eingefügte Objekt wird von dem Stapel entfernt. Falls der Stapel leer ist, bleibt er unverändert.

`ContentType top()`

Die Anfrage liefert das oberste Stapelobjekt. Der Stapel bleibt unverändert. Falls der Stapel leer ist, wird `null` zurückgegeben.

Die generische Klasse `List`

Objekte der generischen Klasse `List` verwalten beliebig viele, linear angeordnete Objekte vom Typ `ContentType`. Auf höchstens eins der verwalteten Objekte, aktuelles Objekt genannt, kann jeweils zugegriffen werden. Wenn eine Liste leer ist, vollständig durchlaufen wurde oder das aktuelle Objekt am Ende der Liste gelöscht wurde, gibt es kein aktuelles Objekt. Das erste oder das letzte Objekt einer Liste können durch einen Auftrag zum aktuellen Objekt gemacht werden. Außerdem kann das dem aktuellen Objekt folgende Listenobjekt zum neuen aktuellen Objekt werden.
Das aktuelle Objekt kann gelesen, verändert oder gelöscht werden. Außerdem kann vor dem aktuellen Objekt ein Listenobjekt eingefügt werden.

Dokumentation der Klasse `List<ContentType>`

`List()`

Eine leere Liste wird erzeugt.

`boolean isEmpty()`

Die Anfrage liefert den Wert `true`, wenn die Liste keine Objekte enthält, sonst liefert sie den Wert `false`.

`boolean hasAccess()`

Die Anfrage liefert den Wert `true`, wenn es ein aktuelles Objekt gibt, sonst liefert sie den Wert `false`.

`void next()`

Falls die Liste nicht leer ist, es ein aktuelles Objekt gibt und dieses nicht das letzte Objekt der Liste ist, wird das dem aktuellen Objekt in der Liste folgende Objekt zum aktuellen Objekt, andernfalls gibt es nach Ausführung des Auftrags kein aktuelles Objekt, d. h. `hasAccess()` liefert den Wert `false`.

`void toFirst()`

Falls die Liste nicht leer ist, wird das erste Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.

`void toLast()`

Falls die Liste nicht leer ist, wird das letzte Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.

`ContentType getContent()`

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`), wird das aktuelle Objekt zurückgegeben. Andernfalls (`hasAccess() == false`) gibt die Anfrage den Wert `null` zurück.

`void setContent(ContentType pContent)`

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`) und `pContent` ungleich `null` ist, wird das aktuelle Objekt durch `pContent` ersetzt. Sonst bleibt die Liste unverändert.

`void append(ContentType pContent)`

Ein neues Objekt `pContent` wird am Ende der Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Wenn die Liste leer ist, wird das Objekt `pContent` in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt (`hasAccess() == false`). Falls `pContent` gleich `null` ist, bleibt die Liste unverändert.

`void insert(ContentType pContent)`

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`), wird ein neues Objekt `pContent` vor dem aktuellen Objekt in die Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Falls die Liste leer ist und es somit kein aktuelles Objekt gibt (`hasAccess() == false`), wird `pContent` in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt. Falls es kein aktuelles Objekt gibt (`hasAccess() == false`) und die Liste nicht leer ist oder `pContent == null` ist, bleibt die Liste unverändert.

`void concat(List<ContentType> pList)`

Die Liste `pList` wird an die Liste angehängt. Anschließend wird `pList` eine leere Liste. Das aktuelle Objekt bleibt unverändert. Falls es sich bei der Liste und `pList` um dasselbe Objekt handelt, `pList == null` oder eine leere Liste ist, bleibt die Liste unverändert.

`void remove()`

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`), wird das aktuelle Objekt gelöscht und das Objekt hinter dem gelöschten Objekt wird zum aktuellen Objekt. Wird das Objekt, das am Ende der Liste steht, gelöscht, gibt es kein aktuelles Objekt mehr (`hasAccess() == false`). Wenn die Liste leer ist oder es kein aktuelles Objekt gibt (`hasAccess() == false`), bleibt die Liste unverändert.

Nicht-lineare Strukturen

Baumklassen

Die folgenden Klassen `BinaryTree` und `BinarySearchTree` modellieren Binärbäume als nicht-lineare dynamische Datenstrukturen.

Die generische Klasse `BinaryTree`

Mithilfe der generischen Klasse `BinaryTree` können beliebig viele Objekte vom Typ `ContentType` in einem Binärbaum verwaltet werden. Ein Objekt der Klasse stellt entweder einen leeren Baum dar oder verwaltet ein Inhaltsobjekt sowie einen linken und einen rechten Teilbaum, die ebenfalls Objekte der generischen Klasse `BinaryTree` sind.

Dokumentation der Klasse `BinaryTree<ContentType>`

`BinaryTree()`

Nach dem Aufruf des Konstruktors existiert ein leerer Binärbaum.

`BinaryTree(ContentType pContent)`

Wenn der Parameter `pContent` ungleich `null` ist, existiert nach dem Aufruf des Konstruktors der Binärbaum und hat `pContent` als Inhaltsobjekt und zwei leere Teilbäume. Falls der Parameter `null` ist, wird ein leerer Binärbaum erzeugt.

```
BinaryTree(ContentType pContent,
           BinaryTree<ContentType> pLeftTree,
           BinaryTree<ContentType> pRightTree)
```

Wenn der Parameter `pContent` ungleich `null` ist, wird ein Binärbaum mit `pContent` als Inhaltsobjekt und den beiden Teilbäume `pLeftTree` und `pRightTree` erzeugt. Sind `pLeftTree` oder `pRightTree` gleich `null`, wird der entsprechende Teilbaum als leerer Binärbaum eingefügt. Wenn der Parameter `pContent` gleich `null` ist, wird ein leerer Binärbaum erzeugt.

`boolean isEmpty()`

Diese Anfrage liefert den Wahrheitswert `true`, wenn der Binärbaum leer ist, sonst liefert sie den Wert `false`.

`void setContent(ContentType pContent)`

Wenn der Binärbaum leer ist, werden der Parameter `pContent` als Inhaltsobjekt sowie ein leerer linker und rechter Teilbaum eingefügt. Ist der Binärbaum nicht leer, wird das Inhaltsobjekt durch `pContent` ersetzt. Die Teilbäume werden nicht geändert. Wenn `pContent` `null` ist, bleibt der Binärbaum unverändert.

`ContentType getContent()`

Diese Anfrage liefert das Inhaltsobjekt des Binärbaums. Wenn der Binärbaum leer ist, wird `null` zurückgegeben.

`void setLeftTree(BinaryTree<ContentType> pTree)`

Wenn der Binärbaum leer ist, wird `pTree` nicht angehängt. Andernfalls erhält der Binärbaum den übergebenen Baum als linken Teilbaum. Falls der Parameter `null` ist, ändert sich nichts.

`void setRightTree(BinaryTree<ContentType> pTree)`

Wenn der Binärbaum leer ist, wird `pTree` nicht angehängt. Andernfalls erhält der Binärbaum den übergebenen Baum als rechten Teilbaum. Falls der Parameter `null` ist, ändert sich nichts.

`BinaryTree<ContentType> getLeftTree()`

Diese Anfrage liefert den linken Teilbaum des Binärbaumes. Der Binärbaum ändert sich nicht. Wenn der Binärbaum leer ist, wird `null` zurückgegeben.

`BinaryTree<ContentType> getRightTree()`

Diese Anfrage liefert den rechten Teilbaum des Binärbaumes. Der Binärbaum ändert sich nicht. Wenn der Binärbaum leer ist, wird `null` zurückgegeben.

Die generische Klasse `BinarySearchTree`

Mithilfe der generischen Klasse `BinarySearchTree` können beliebig viele Objekte des Typs `ContentType` in einem Binärbaum (binärer Suchbaum) entsprechend einer Ordnungsrelation verwaltet werden.
Ein Objekt der Klasse `BinarySearchTree` stellt entweder einen leeren Baum dar oder verwaltet ein Inhaltsobjekt vom Typ `ContentType` sowie einen linken und einen rechten Teilbaum, die ebenfalls Objekte der Klasse `BinarySearchTree` sind.
Die Klasse der Objekte, die in dem Suchbaum verwaltet werden sollen, muss das generische Interface `ComparableContent` implementieren. Dabei muss durch Überschreiben der drei Vergleichsmethoden `isLess`, `isEqual`, `isGreater` (siehe Dokumentation dieses Interfaces) eine eindeutige Ordnungsrelation festgelegt sein.

Beispiel einer Klasse, die das Interface `ComparableContent` implementiert:

```
public class Eintrag implements ComparableContent<Eintrag> {

  private int wert;
  // weitere Attribute

  public boolean isLess(Eintrag pEintrag) {
    return this.gibWert() < pEintrag.gibWert();
  }

  public boolean isEqual(Eintrag pEintrag) {
    return this.gibWert() == pEintrag.gibWert();
  }

  public boolean isGreater(Eintrag pEintrag) {
    return this.gibWert() > pEintrag.gibWert();
  }

  public int gibWert() {
    return this.wert;
  }

  // weitere Methoden
}
```

Die Objekte der Klasse `ContentType` sind damit vollständig geordnet. Für je zwei Objekte `c1` und `c2` vom Typ `ContentType` gilt also insbesondere genau eine der drei Aussagen:

- `c1.isLess(c2)` (Sprechweise: `c1` ist kleiner als `c2`)
- `c1.isEqual(c2)` (Sprechweise: `c1` ist gleichgroß wie `c2`)
- `c1.isGreater(c2)` (Sprechweise: `c1` ist größer als `c2`)

Alle Objekte im linken Teilbaum sind kleiner als das Inhaltsobjekt des Binärbaumes. Alle Objekte im rechten Teilbaum sind größer als das Inhaltsobjekt des Binärbaumes. Diese Bedingung gilt auch in beiden Teilbäumen.

Dokumentation der Klasse `BinarySearchTree<ContentType extends ComparableContent<ContentType>>`

`BinarySearchTree()`

Der Konstruktor erzeugt einen leeren Suchbaum.

`boolean isEmpty()`

Diese Anfrage liefert den Wahrheitswert `true`, wenn der Suchbaum leer ist, sonst liefert sie den Wert `false`.

`void insert(ContentType pContent)`

Falls bereits ein Objekt in dem Suchbaum vorhanden ist, das gleichgroß ist wie `pContent`, passiert nichts. Andernfalls wird das Objekt `pContent` entsprechend der Ordnungsrelation in den Baum eingeordnet. Falls der Parameter `null` ist, ändert sich nichts.

`ContentType search(ContentType pContent)`

Falls ein Objekt im binären Suchbaum enthalten ist, das gleichgroß ist wie `pContent`, liefert die Anfrage dieses, ansonsten wird `null` zurückgegeben. Falls der Parameter `null` ist, wird `null` zurückgegeben.

`void remove(ContentType pContent)`

Falls ein Objekt im binären Suchbaum enthalten ist, das gleichgroß ist wie `pContent`, wird dieses entfernt. Falls der Parameter `null` ist, ändert sich nichts.

`ContentType getContent()`

Diese Anfrage liefert das Inhaltsobjekt des Suchbaumes. Wenn der Suchbaum leer ist, wird `null` zurückgegeben.

`BinarySearchTree<ContentType> getLeftTree()`

Diese Anfrage liefert den linken Teilbaum des binären Suchbaumes. Der binäre Suchbaum ändert sich nicht. Wenn er leer ist, wird `null` zurückgegeben.

`BinarySearchTree<ContentType> getRightTree()`

Diese Anfrage liefert den rechten Teilbaum des Suchbaumes. Der Suchbaum ändert sich nicht. Wenn er leer ist, wird `null` zurückgegeben.

Das generische Interface `ComparableContent<ContentType>`

Das generische Interface `ComparableContent` muss von Klassen implementiert werden, deren Objekte in einen Suchbaum (`BinarySearchTree`) eingefügt werden sollen. Die Ordnungsrelation wird in diesen Klassen durch Überschreiben der drei implizit abstrakten Methoden `isGreater`, `isEqual` und `isLess` festgelegt.

Das Interface `ComparableContent` gibt folgende implizit abstrakte Methoden vor:

```
boolean isGreater(ContentType pComparableContent)
```

Wenn festgestellt wird, dass das Objekt, von dem die Methode aufgerufen wird, bzgl. der gewünschten Ordnungsrelation größer als das Objekt `pComparableContent` ist, wird `true` geliefert. Sonst wird `false` geliefert.

```
boolean isEqual(ContentType pComparableContent)
```

Wenn festgestellt wird, dass das Objekt, von dem die Methode aufgerufen wird, bzgl. der gewünschten Ordnungsrelation gleich dem Objekt `pComparableContent` ist, wird `true` geliefert. Sonst wird `false` geliefert.

```
boolean isLess(ContentType pComparableContent)
```

Wenn festgestellt wird, dass das Objekt, von dem die Methode aufgerufen wird, bzgl. der gewünschten Ordnungsrelation kleiner als das Objekt `pComparableContent` ist, wird `true` geliefert. Sonst wird `false` geliefert.

Graphenklassen

Die folgenden Klassen `Graph`, `Vertex` und `Edge` werden verwendet, um die nicht-lineare dynamische Datenstruktur Graph zu realisieren.

Die Klasse `Graph`

Die Klasse `Graph` stellt einen ungerichteten, kantengewichteten Graphen dar. Es können Knoten- und Kantenobjekte hinzugefügt und entfernt, flache Kopien der Knoten- und Kantenlisten des Graphen angefragt und Markierungen von Knoten und Kanten gesetzt und überprüft werden. Des Weiteren kann eine Liste der Nachbarn eines bestimmten Knoten, eine Liste der inzidenten Kanten eines bestimmten Knoten und die Kante von einem bestimmten Knoten zu einem anderen Knoten angefragt werden. Abgesehen davon kann abgefragt werden, welches Knotenobjekt zu einer bestimmten ID gehört und ob der Graph leer ist.

Dokumentation der Klasse `Graph`

`Graph()`

Ein Objekt vom Typ `Graph` wird erstellt. Der von diesem Objekt repräsentierte Graph ist leer.

`void addVertex(Vertex pVertex)`

Der Auftrag fügt den Knoten `pVertex` vom Typ `Vertex` in den Graphen ein, sofern es noch keinen Knoten mit demselben ID-Eintrag wie `pVertex` im Graphen gibt und `pVertex` eine ID ungleich `null` hat. Ansonsten passiert nichts.

`void addEdge(Edge pEdge)`

Der Auftrag fügt die Kante `pEdge` in den Graphen ein, sofern beide durch die Kante verbundenen Knoten im Graphen enthalten sind, nicht identisch sind und noch keine Kante zwischen den beiden Knoten existiert. Ansonsten passiert nichts.

`void removeVertex(Vertex pVertex)`

Der Auftrag entfernt den Knoten `pVertex` aus dem Graphen und löscht alle Kanten, die mit ihm inzident sind. Ist der Knoten `pVertex` nicht im Graphen enthalten, passiert nichts.

`void removeEdge(Edge pEdge)`

Der Auftrag entfernt die Kante `pEdge` aus dem Graphen. Ist die Kante `pEdge` nicht im Graphen enthalten, passiert nichts.

`Vertex getVertex(String pID)`

Die Anfrage liefert das Knotenobjekt mit `pID` als ID. Ist ein solches Knotenobjekt nicht im Graphen enthalten, wird `null` zurückgeliefert.

`List<Vertex> getVertices()`

Die Anfrage liefert eine neue Liste aller Knotenobjekte vom Typ `List<Vertex>`. Enthält der Graph keine Knotenobjekte, so wird eine leere Liste zurückgeliefert.

`List<Vertex> getNeighbours(Vertex pVertex)`

Die Anfrage liefert alle Nachbarn des Knotens `pVertex` als neue Liste vom Typ `List<Vertex>`. Hat der Knoten `pVertex` keine Nachbarn in diesem Graphen oder ist gar nicht in diesem Graphen enthalten, so wird eine leere Liste zurückgeliefert.

`List<Edge> getEdges()`

Die Anfrage liefert eine neue Liste aller Kantenobjekte vom Typ `List<Edge>`. Enthält der Graph keine Kantenobjekte, so wird eine leere Liste zurückgeliefert.

`List<Edge> getEdges(Vertex pVertex)`

Die Anfrage liefert eine neue Liste aller inzidenten Kanten zum Knoten `pVertex`. Hat der Knoten `pVertex` keine inzidenten Kanten in diesem Graphen oder ist gar nicht in diesem Graphen enthalten, so wird eine leere Liste zurückgeliefert.

`Edge getEdge(Vertex pVertex, Vertex pAnotherVertex)`

Die Anfrage liefert die Kante, welche die Knoten `pVertex` und `pAnotherVertex` verbindet, als Objekt vom Typ `Edge`. Ist der Knoten `pVertex` oder der Knoten `pAnotherVertex` nicht im Graphen enthalten oder gibt es keine Kante, die beide Knoten verbindet, so wird `null` zurückgeliefert.

`void setAllVertexMarks(boolean pMark)`

Der Auftrag setzt die Markierungen aller Knoten des Graphen auf den Wert `pMark`.

`boolean allVerticesMarked()`

Die Anfrage liefert `true`, wenn die Markierungen aller Knoten des Graphen den Wert `true` haben, ansonsten `false`.

`void setAllEdgeMarks(boolean pMark)`

Der Auftrag setzt die Markierungen aller Kanten des Graphen auf den Wert `pMark`.

`boolean allEdgesMarked()`

Die Anfrage liefert `true`, wenn die Markierungen aller Kanten des Graphen den Wert `true` haben, ansonsten `false`.

`boolean isEmpty()`

Die Anfrage liefert `true`, wenn der Graph keine Knoten enthält, ansonsten `false`.

Die Klasse `Vertex`

Die Klasse `Vertex` stellt einen einzelnen Knoten eines Graphen dar. Jedes Objekt dieser Klasse verfügt über eine im Graphen eindeutige ID als `String` und kann diese ID zurückliefern. Darüber hinaus kann eine Markierung gesetzt und abgefragt werden.

Dokumentation der Klasse `Vertex`

`Vertex(String pID)`

Ein neues Objekt vom Typ `Vertex` mit der ID `pID` wird erstellt. Seine Markierung hat den Wert `false`.

`String getID()`

Die Anfrage liefert die ID des Knotens als `String`.

`void setMark(boolean pMark)`

Der Auftrag setzt die Markierung des Knotens auf den Wert `pMark`.

`boolean isMarked()`

Die Anfrage liefert `true`, wenn die Markierung des Knotens den Wert `true` hat, ansonsten `false`.

Die Klasse `Edge`

Die Klasse `Edge` stellt eine einzelne, ungerichtete Kante eines Graphen dar. Beim Erstellen werden die beiden durch sie zu verbindenden Knotenobjekte und eine Gewichtung als `double` übergeben. Beide Knotenobjekte können abgefragt werden. Des Weiteren können die Gewichtung und eine Markierung gesetzt und abgefragt werden.

Dokumentation der Klasse `Edge`

`Edge(Vertex pVertex, Vertex pAnotherVertex, double pWeight)`

Ein neues Objekt vom Typ `Edge` wird erstellt. Die von diesem Objekt repräsentierte Kante verbindet die Knoten `pVertex` und `pAnotherVertex` mit der Gewichtung `pWeight`. Ihre Markierung hat den Wert `false`.

`void setWeight(double pWeight)`

Der Auftrag setzt das Gewicht der Kante auf den Wert `pWeight`.

`double getWeight()`

Die Anfrage liefert das Gewicht der Kante als `double`.

`Vertex[] getVertices()`

Die Anfrage gibt die beiden Knoten, die durch die Kante verbunden werden, als neues Feld vom Typ `Vertex` zurück. Das Feld hat genau zwei Einträge mit den Indexwerten `0` und `1`.

`void setMark(boolean pMark)`

Der Auftrag setzt die Markierung der Kante auf den Wert `pMark`.

`void setWeight(double pWeight)`

Der Auftrag setzt das Gewicht der Kante auf den Wert `pWeight`.

`boolean isMarked()`

Die Anfrage liefert `true`, wenn die Markierung der Kante den Wert `true` hat, ansonsten `false`.

Datenbankklassen

Die Klasse `DatabaseConnector`

Ein Objekt der Klasse `DatabaseConnector` ermöglicht die Abfrage und Manipulation einer relationalen Datenbank. Beim Erzeugen des Objekts wird eine Datenbankverbindung aufgebaut, so dass anschließend SQL-Anweisungen an diese Datenbank gerichtet werden können.

Dokumentation der Klasse `DatabaseConnector`

`DatabaseConnector(String pIP, int pPort, String pDatabase, String pUsername, String pPassword)`

Ein Objekt vom Typ `DatabaseConnector` wird erstellt, und eine Verbindung zur Datenbank wird aufgebaut. Mit den Parametern `pIP` und `pPort` werden die IP-Adresse und die Port-Nummer übergeben, unter denen die Datenbank mit Namen `pDatabase` zu erreichen ist. Mit den Parametern `pUsername` und `pPassword` werden Benutzername und Passwort für die Datenbank übergeben.

`void executeStatement(String pSQLStatement)`

Der Auftrag schickt den im Parameter `pSQLStatement` enthaltenen SQL-Befehl an die Datenbank ab.
Handelt es sich bei `pSQLStatement` um einen SQL-Befehl, der eine Ergebnismenge liefert, so kann dieses Ergebnis anschließend mit der Methode `getCurrentQueryResult` abgerufen werden.

`QueryResult getCurrentQueryResult()`

Die Anfrage liefert das Ergebnis des letzten mit der Methode `executeStatement` an die Datenbank geschickten SQL-Befehls als Objekt vom Typ `QueryResult` zurück.
Wurde bisher kein SQL-Befehl abgeschickt oder ergab der letzte Aufruf von `executeStatement` keine Ergebnismenge (z. B. bei einem `INSERT`-Befehl oder einem Syntaxfehler), so wird `null` geliefert.

`String getErrorMessage()`

Die Anfrage liefert `null` oder eine Fehlermeldung, die sich jeweils auf die letzte zuvor ausgeführte Datenbankoperation bezieht.

`void close()`

Die Datenbankverbindung wird geschlossen.

Die Klasse `QueryResult`

Ein Objekt der Klasse `QueryResult` stellt die Ergebnistabelle einer Datenbankanfrage mit Hilfe der Klasse `DatabaseConnector` dar. Objekte dieser Klasse werden nur von der Klasse `DatabaseConnector` erstellt. Die Klasse verfügt über keinen öffentlichen Konstruktor.

Dokumentation der Klasse `QueryResult`

`String[][] getData()`

Die Anfrage liefert die Einträge der Ergebnistabelle als zweidimensionales Feld vom Typ String. Der erste Index des Feldes stellt die Zeile und der zweite die Spalte dar (d. h. String[zeile][spalte]).

`String[] getColumnNames()`

Die Anfrage liefert die Bezeichner der Spalten der Ergebnistabelle als Feld vom Typ `String` zurück.

`String[] getColumnTypes()`

Die Anfrage liefert die Typenbezeichnung der Spalten der Ergebnistabelle als Feld vom Typ `String` zurück. Die Bezeichnungen entsprechen den Angaben in der Datenbank.

`int getRowCount()`

Die Anfrage liefert die Anzahl der Zeilen der Ergebnistabelle als `int`.

`int getColumnCount()`

Die Anfrage liefert die Anzahl der Spalten der Ergebnistabelle als `int`.

Netzklassen

Die Klasse `Connection`

Objekte der Klasse `Connection` ermöglichen eine Netzwerkverbindung zu einem Server mittels TCP/IP-Protokoll. Nach Verbindungsaufbau können Zeichenketten (`Strings`) zum Server gesendet und von diesem empfangen werden. Zur Vereinfachung geschieht dies zeilenweise, d. h., beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfang wird dieser entfernt. Es findet nur eine rudimentäre Fehlerbehandlung statt, so dass z. B. der Zugriff auf unterbrochene oder bereits getrennte Verbindungen nicht zu einem Programmabbruch führt. Eine einmal getrennte Verbindung kann nicht reaktiviert werden.

Dokumentation der Klasse `Connection`

`Connection(String pServerIP, int pServerPort)`

Ein Objekt vom Typ `Connection` wird erstellt. Dadurch wird eine Verbindung zum durch `pServerIP` und `pServerPort` spezifizierten Server aufgebaut, so dass Daten (Zeichenketten) gesendet und empfangen werden können. Kann die Verbindung nicht hergestellt werden, kann die Instanz von `Connection` nicht mehr verwendet werden.

`void send(String pMessage)`

Die Nachricht `pMessage` wird – um einen Zeilentrenner ergänzt – an den Server gesendet. Schlägt der Versand fehl, geschieht nichts.

`String receive()`

Es wird beliebig lange auf eine eingehende Nachricht vom Server gewartet und diese Nachricht anschließend zurückgegeben. Der vom Server angehängte Zeilentrenner wird zuvor entfernt. Während des Wartens ist der ausführende Prozess blockiert. Wurde die Verbindung unterbrochen oder durch den Server unvermittelt geschlossen, wird `null` zurückgegeben.

`void close()`

Die Verbindung zum Server wird getrennt und kann nicht mehr verwendet werden. War die Verbindung bereits getrennt, geschieht nichts.

Die Klasse `Client`

Objekte von Unterklassen der abstrakten Klasse `Client` ermöglichen Netzwerkverbindungen zu einem Server mittels TCP/IP-Protokoll. Nach Verbindungsaufbau können Zeichenketten (`Strings`) zum Server gesendet und von diesem empfangen werden, wobei der Nachrichtenempfang nebenläufig geschieht. Zur Vereinfachung finden Nachrichtenversand und -empfang zeilenweise statt, d. h., beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfang wird dieser entfernt. Jede empfangene Nachricht wird einer Ereignisbehandlungsmethode übergeben, die in Unterklassen implementiert werden muss. Es findet nur eine rudimentäre Fehlerbehandlung statt, so dass z. B. Verbindungsabbrüche nicht zu einem Programmabbruch führen. Eine einmal unterbrochene oder getrennte Verbindung kann nicht reaktiviert werden.

Dokumentation der Klasse `Client`

`Client(String pServerIP, int pServerPort)`

Es wird eine Verbindung zum durch `pServerIP` und `pServerPort` spezifizierten Server aufgebaut, so dass Daten (Zeichenketten) gesendet und empfangen werden können. Kann die Verbindung nicht hergestellt werden, kann der Client nicht zum Datenaustausch verwendet werden.

`boolean isConnected()`

Die Anfrage liefert den Wert `true`, wenn der Client mit dem Server aktuell verbunden ist. Ansonsten liefert sie den Wert `false`.

`void send(String pMessage)`

Die Nachricht `pMessage` wird – um einen Zeilentrenner ergänzt – an den Server gesendet. Schlägt der Versand fehl, geschieht nichts.

`void close()`

Die Verbindung zum Server wird getrennt und der Client kann nicht mehr verwendet werden. Ist Client bereits vor Aufruf der Methode in diesem Zustand, geschieht nichts.

`void processMessage(String pMessage)`

Diese Methode wird aufgerufen, wenn der Client die Nachricht `pMessage` vom Server empfangen hat. Der vom Server ergänzte Zeilentrenner wurde zuvor entfernt. Die Methode ist abstrakt und muss in einer Unterklasse der Klasse `Client` überschrieben werden, so dass auf den Empfang der Nachricht reagiert wird. Der Aufruf der Methode erfolgt nicht synchronisiert.

Die Klasse `Server`

Objekte von Unterklassen der abstrakten Klasse `Server` ermöglichen das Anbieten von Serverdiensten, so dass Clients Verbindungen zum Server mittels TCP/IP-Protokoll aufbauen können. Zur Vereinfachung finden Nachrichtenversand und -empfang zeilenweise statt, d. h., beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfang wird dieser entfernt. Verbindungsannahme, Nachrichtenempfang und Verbindungsende geschehen nebenläufig. Auf diese Ereignisse muss durch Überschreiben der entsprechenden Ereignisbehandlungsmethoden reagiert werden. Es findet nur eine rudimentäre Fehlerbehandlung statt, so dass z. B. Verbindungsabbrüche nicht zu einem Programmabbruch führen. Einmal unterbrochene oder getrennte Verbindungen können nicht reaktiviert werden.

Dokumentation der Klasse `Server`

`Server(int pPort)`

Ein Objekt vom Typ `Server` wird erstellt, das über die angegebene Portnummer einen Dienst anbietet an. Clients können sich mit dem Server verbinden, so dass Daten (Zeichenketten) zu diesen gesendet und von diesen empfangen werden können. Kann der Server unter der angegebenen Portnummer keinen Dienst anbieten (z. B. weil die Portnummer bereits belegt ist), ist keine Verbindungsaufnahme zum Server und kein Datenaustausch möglich.

`boolean isOpen()`

Die Anfrage liefert den Wert `true`, wenn der Server auf Port `pPort` einen Dienst anbietet. Ansonsten liefert die Methode den Wert `false`.

`boolean isConnectedTo(String pClientIP, int pClientPort)`

Die Anfrage liefert den Wert `true`, wenn der Server mit dem durch `pClientIP` und `pClientPort` spezifizierten Client aktuell verbunden ist. Ansonsten liefert die Methode den Wert `false`.

`void send(String pClientIP, int pClientPort, String pMessage)`

Die Nachricht `pMessage` wird – um einen Zeilentrenner erweitert – an den durch `pClientIP` und `pClientPort` spezifizierten Client gesendet. Schlägt der Versand fehl, geschieht nichts.

`void sendToAll(String pMessage)`

Die Nachricht `pMessage` wird – um einen Zeilentrenner erweitert – an alle mit dem Server verbundenen Clients gesendet. Schlägt der Versand an einen Client fehl, wird dieser Client übersprungen.

`void closeConnection(String pClientIP, int pClientPort)`

Die Verbindung des Servers zu dem durch `pClientIP` und `pClientPort` spezifizierten Client wird getrennt. Zuvor wird die Methode `processClosingConnection` mit IP-Adresse und Port des jeweiligen Clients aufgerufen. Ist der Server nicht mit dem in der Parameterliste spezifizierten Client verbunden, geschieht nichts.

`void close()`

Alle bestehenden Verbindungen zu Clients werden getrennt und der Server kann nicht mehr verwendet werden. Ist der Server bereits vor Aufruf der Methode in diesem Zustand, geschieht nichts.

`void processNewConnection(String pClientIP, int pClientPort)`

Diese Ereignisbehandlungsmethode wird aufgerufen, wenn sich ein Client mit IP-Adresse `pClientIP` und Portnummer `pClientPort` mit dem Server verbunden hat. Die Methode ist abstrakt und muss in einer Unterklasse der Klasse `Server` überschrieben werden, so dass auf den Neuaufbau der Verbindung reagiert wird. Der Aufruf der Methode erfolgt nicht synchronisiert.

`void processMessage(String pClientIP, int pClientPort, String pMessage)`

Diese Ereignisbehandlungsmethode wird aufgerufen, wenn der Server die Nachricht `pMessage` von dem durch `pClientIP` und `pClientPort` spezifizierten Client empfangen hat. Der vom Client hinzugefügte Zeilentrenner wurde zuvor entfernt. Die Methode ist abstrakt und muss in einer Unterklasse der Klasse `Server` überschrieben werden, so dass auf den Empfang der Nachricht reagiert wird. Der Aufruf der Methode erfolgt nicht synchronisiert.

`void processClosingConnection(String pClientIP, int pClientPort)`

Sofern der Server die Verbindung zu dem durch `pClientIP` und `pClientPort` spezifizierten Client trennt, wird diese Ereignisbehandlungsmethode aufgerufen, unmittelbar bevor die Verbindungstrennung tatsächlich erfolgt. Wird die Verbindung unvermittelt unterbrochen oder hat der in der Parameterliste spezifizierte Client die Verbindung zum Server unvermittelt getrennt, erfolgt der Methodenaufruf nach der Unterbrechung / Trennung der Verbindung. Die Methode ist abstrakt und muss in einer Unterklasse der Klasse `Server` überschrieben werden, so dass auf das Ende der Verbindung zum angegebenen Client reagiert wird. Der Aufruf der Methode erfolgt nicht synchronisiert.