Dokumentation für den Leistungskurs

1. Basis-Sprachelemente und -Datentypen in Java

Kenntnisse über Java-spezifische Klassen insbesondere zur Gestaltung einer grafischen Benutzungsoberfläche werden bei den Aufgaben für das Zentralabitur nicht vorausgesetzt.

Sprachelemente

- Klassen
- Beziehungen zwischen Klassen
  - gerichtete Assoziation
  - Vererbung
  - Interfaces
- Attribute und Methoden (mit Parametern und Rückgabewerten)
- Wertzuweisungen
- Verzweigungen (if - , 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(int 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

2. SQL-Sprachelemente

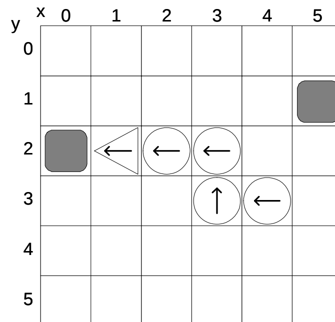
```
SELECT (DISTINCT) … FROM
WHERE
GROUP BY
ORDER BY
ASC, DESC
(LEFT / RIGHT / INNER) JOIN … ON
UNION
AS
NULL
Vergleichsoperatoren: =, <>, >, <, >=, <=, LIKE, BETWEEN, IN, IS
NULL
Arithmetische Operatoren: +, -, *, /, (…)
Logische Verknüpfungen: AND, OR, NOT
Funktionen: COUNT, SUM, MAX, MIN, AVG
```

SQL-Abfragen über eine und mehrere verknüpfte Tabellen

Geschachtelte SQL-Ausdrücke

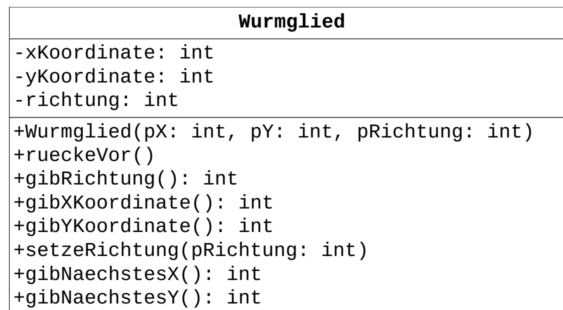
3. Klassendiagramme

Klassendiagramme beschreiben die vorhandenen Klassen mit ihren Attributen und Methoden sowie die Beziehungen der Klassen untereinander.
Im Folgenden soll eine Variante eines Computerspiels (Darstellung: Schlange auf einem Spielfeld) modelliert werden:

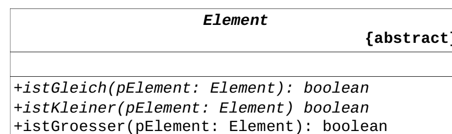


Klassen

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.

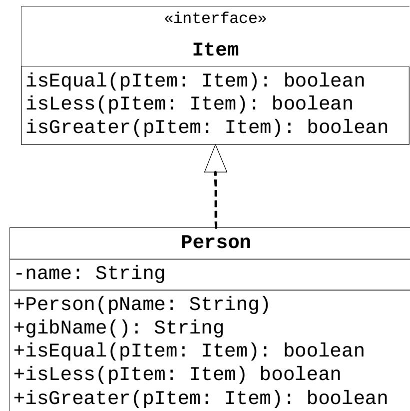


Bei **abstrakten Klassen**, also Klassen, von denen kein Objekt erzeugt werden kann, wird unter den Klassennamen im Diagramm `{abstract}` geschrieben. Abstrakte Methoden, also Methoden, für die keine Implementierungen angegeben werden und die nicht aufgerufen werden können, werden in Kursivschrift dargestellt.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



Bei diesem Beispiel sind die Methoden `istGleich` und `istKleiner` abstrakt. Die Methode `istGroesser` ist mit Hilfe der abstrakten Methoden implementiert.

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



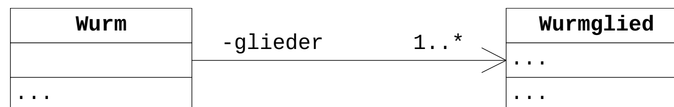
Beziehungen zwischen Klassen

Assoziation

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

Mögliche Multiplizitäten:

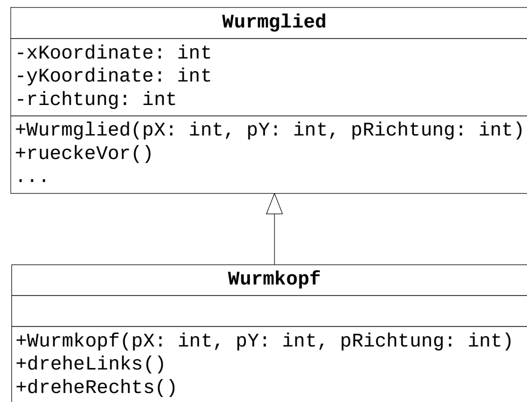
- `1` genau ein assoziiertes Objekt
- `0..1` kein oder ein assoziiertes Objekt
- `0..*` beliebig viele assoziierte Objekte
- `1..*` mindestens ein, beliebig viele assoziierte Objekte



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 Attribute und alle öffentlichen Methoden der Oberklasse zur Verfügung. In der Unterklasse können Attribute und Methoden ergänzt oder auch Methoden der Oberklasse überschrieben werden.

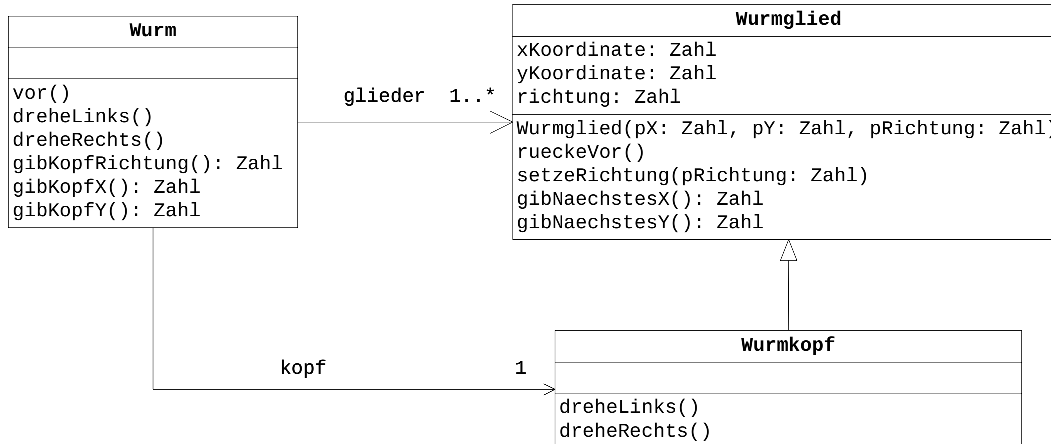


Die Klasse `Wurmkopf` spezialisiert hier die Klasse `Wurmglied`.

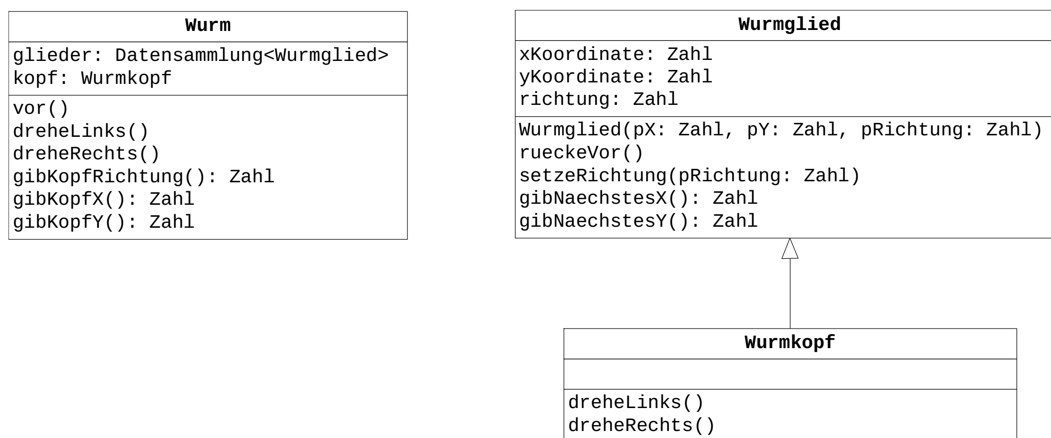
Entwurfsdiagramm

Bei einem Entwurf werden 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 Klammer der Datentyp oder die Klassenbezeichnung der Elemente, die dort verwaltet werden. Anfragen werden durch den Datentyp des Rückgabewertes gekennzeichnet.

Version 1: Assoziationspfeile mit Multiplizitäten



Version 2: Attribute mit Datensammlung

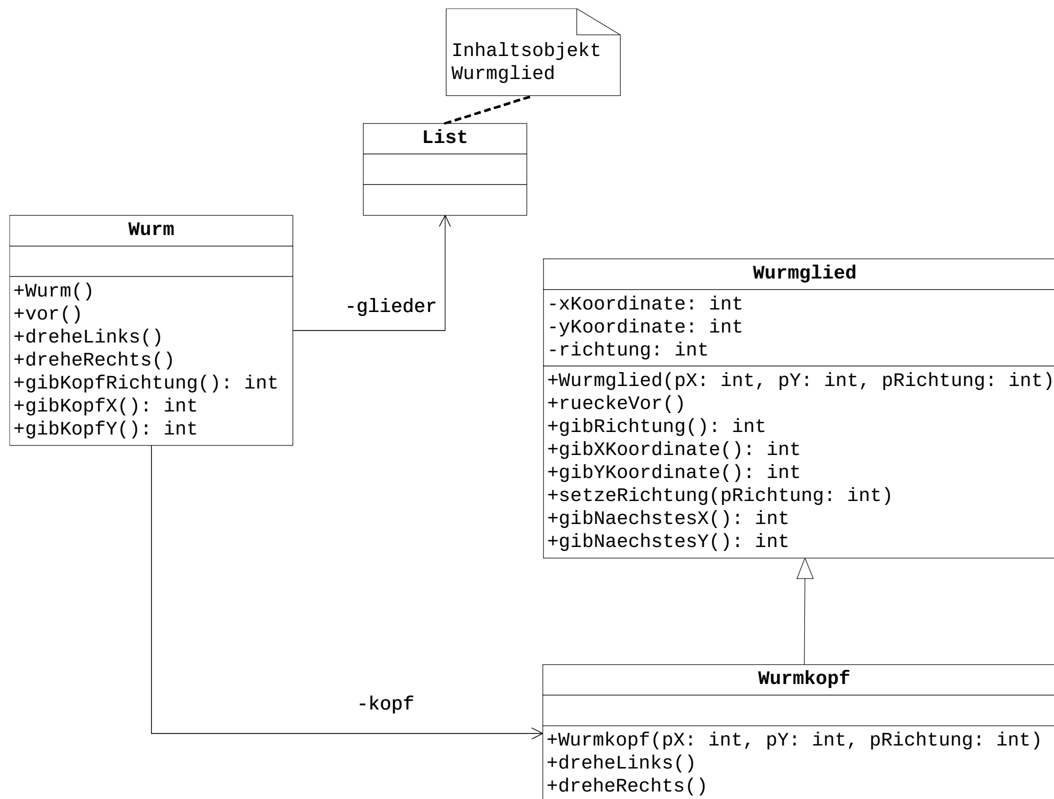


Beide Darstellungen drücken den gleichen Sachverhalt aus.

Implementationsdiagramm

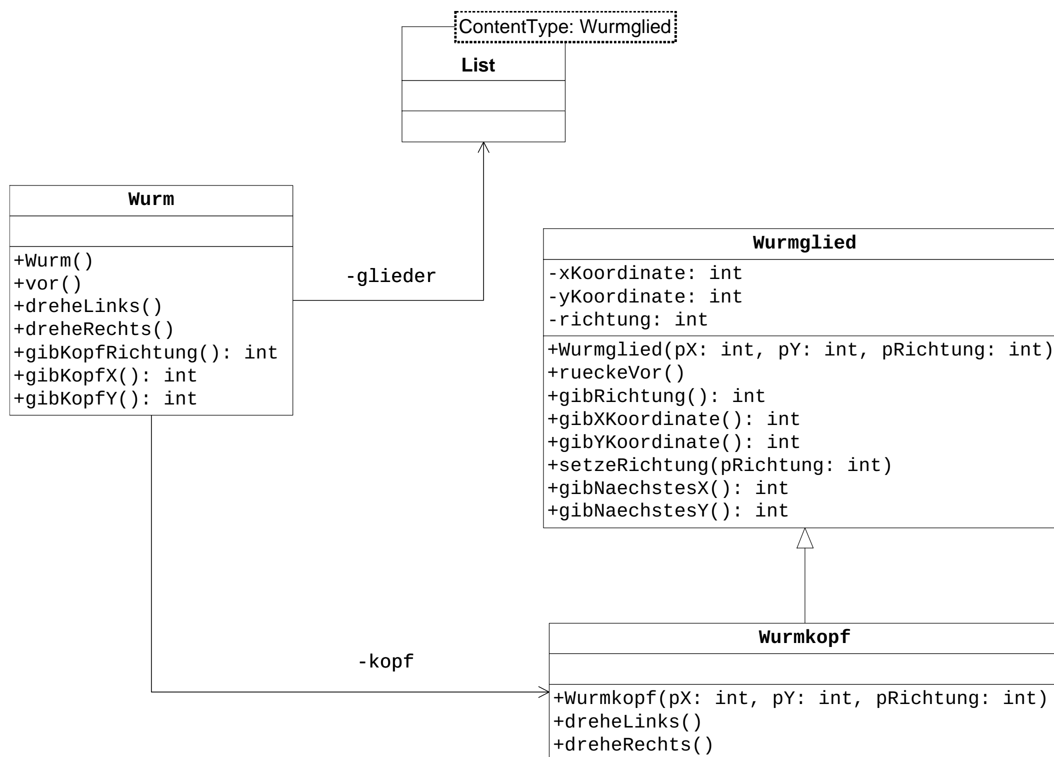
Ein Implementationsdiagramm ergibt sich durch Präzisierung eines Entwurfsdiagramms und orientiert sich stärker an der verwendeten Programmiersprache. Für die im Entwurfsdiagramm angegebenen Datensammlungen werden konkrete Datenstrukturen gewählt, deren Inhaltstypen in Form von Kommentaren oder als Parameter angegeben 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.

Beispiel für ein Implementationsdiagramm mit Assoziationen und Vererbung
Version 1: Die Klasse `List` verwaltet allgemein Inhaltsobjekte der Klasse `Object`.



Erläuterung:
Bezeichner der Attribute, durch die die Assoziationen realisiert werden, stehen an den Pfeilen.
Objekte der Klasse `List` verwalten Objekte der Klasse `Object`. Der Kommentar an der Klasse `List` verdeutlicht, welche Inhaltsobjekte in der Klasse `List` hier verwendet werden sollen.

Version 2: Die Klasse `List` ist eine generische Klasse (Template).



Erläuterung:
Bezeichner der Attribute, durch die die Assoziationen realisiert werden, stehen an den Pfeilen.
Bei der Klasse `List` handelt es sich um eine generische Klasse, über Parameter wird der Datentyp der Inhaltsobjekte angegeben. In der Klasse `List` werden Objekte der Klasse `Wurmglied` verwaltet.

4. Klassendokumentationen

In Java werden Objekte über Referenzen verwaltet, d.h., eine Variable `pObject` von 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.

4.1. Lineare Strukturen

Die generische Klasse Queue<ContentType>

Objekte der generischen Klasse **Queue** (Warteschlange) 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 generischen Klasse Queue<ContentType>

Konstruktor **`Queue()`**
Eine leere Schlange wird erzeugt. Objekte, die in dieser Schlange verwaltet werden, müssen vom Typ **ContentType** sein.

Anfrage **`boolean isEmpty()`**
Die Anfrage liefert den Wert `true`, wenn die Schlange keine Objekte enthält, sonst liefert sie den Wert `false`.

Auftrag **`void enqueue(ContentType pContent)`**
Das Objekt `pContent` wird an die Schlange angehängt. Falls `pContent` gleich `null` ist, bleibt die Schlange unverändert.

Auftrag **`void dequeue()`**
Das erste Objekt wird aus der Schlange entfernt. Falls die Schlange leer ist, wird sie nicht verändert.

Anfrage **`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<ContentType>`

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 generischen Klasse `Stack<ContentType>`

Konstruktor	**`Stack()`** Ein leerer Stapel wird erzeugt. Objekte, die in diesem Stapel verwaltet werden, müssen vom Typ **ContentType** sein.
Anfrage	**`boolean isEmpty()`** Die Anfrage liefert den Wert `true`, wenn der Stapel keine Objekte enthält, sonst liefert sie den Wert `false`.
Auftrag	**`void push(ContentType pContent)`** Das Objekt `pContent` wird oben auf den Stapel gelegt. Falls `pContent` gleich `null` ist, bleibt der Stapel unverändert.
Auftrag	**`void pop()`** Das zuletzt eingefügte Objekt wird von dem Stapel entfernt. Falls der Stapel leer ist, bleibt er unverändert.
Anfrage	**`ContentType top()`** Die Anfrage liefert das oberste Stapelobjekt. Der Stapel bleibt unverändert. Falls der Stapel leer ist, wird `null` zurückgegeben.

Die generische Klasse `List<ContentType>`

Objekte der generischen Klasse **`List`**verwalten beliebig viele, linear angeordnete Objekte vom Typ **`ContentType`**. Auf höchstens ein Listenobjekt, 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>`

Konstruktor	**`List()`** Eine leere Liste wird erzeugt.
Anfrage	**`boolean isEmpty()`** Die Anfrage liefert den Wert `true`, wenn die Liste keine Objekte enthält, sonst liefert sie den Wert `false`.
Anfrage	**`boolean hasAccess()`** Die Anfrage liefert den Wert `true`, wenn es ein aktuelles Objekt gibt, sonst liefert sie den Wert `false`.
Auftrag	**`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`.
Auftrag	**`void toFirst()`** Falls die Liste nicht leer ist, wird das erste Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.
Auftrag	**`void toLast()`** Falls die Liste nicht leer ist, wird das letzte Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.
Anfrage	**`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.

Auftrag **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.

Auftrag **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.

Auftrag **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.

Auftrag **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.

Auftrag **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.

4.2 Nicht-lineare Strukturen

Die Klasse `BinaryTree<ContentType>`

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>`

Konstruktor **`BinaryTree<ContentType>()`**
Nach dem Aufruf des Konstruktors existiert ein leerer Binärbaum.

Konstruktor **`BinaryTree<ContentType>(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.

Konstruktor **`BinaryTree<ContentType>(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.

Anfrage **`boolean isEmpty()`**
Diese Anfrage liefert den Wahrheitswert `true`, wenn der Binärbaum leer ist, sonst liefert sie den Wert `false`.

Auftrag **`void setContent(ContentType pContent)`**
Wenn der Binärbaum leer ist, wird 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.

Anfrage **`ContentType getContent()`**
Diese Anfrage liefert das Inhaltsobjekt des Binärbaums. Wenn der Binärbaum leer ist, wird `null` zurückgegeben.

Auftrag **`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.

Auftrag **`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.

Anfrage **`BinaryTree<ContentType> getLeftTree()`**
Diese Anfrage liefert den linken Teilbaum des Binärbaumes. Der Binärbaum ändert sich nicht. Wenn der Binärbaum leer ist, wird `null` zurückgegeben.

Anfrage **`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 Klasse `BinarySearchTree<ContentType extends ComparableContent<ContentType>>`

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` (s. Dokumentation des Interfaces) eine eindeutige Ordnungsrelation festgelegt sein.

Beispiel einer solchen Klasse:

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

  int wert;
  // diverse weitere Attribute

  public boolean isLess(Entry pContent) {
    return this.getWert() < pContent.getWert();
  }

  public boolean isEqual(Entry pContent) {
    return this.getWert() == pContent.getWert();
  }

  public boolean isGreater(Entry pContent) {
    return this.getWert() > pContent.getWert();
  }

  public int getWert() {
    return this.wert;
  }
```

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 generischen Klasse BinarySearchTree<ContentType extends ComparableContent<ContentType>>

Konstruktor **BinarySearchTree()**

Der Konstruktor erzeugt einen leeren Suchbaum.

Anfrage **boolean isEmpty()**

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

Auftrag **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.

Anfrage **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.

Auftrag **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.

Anfrage **ContentType getContent()**

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

Anfrage **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.

Anfrage **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 (Schnittstelle) `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:

Anfrage	**`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.
Anfrage	**`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.
Anfrage	**`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.

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

Konstruktor **Graph()**

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

Auftrag **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.

Auftrag **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.

Auftrag **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.

Auftrag **void removeEdge(Edge pEdge)**

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

Anfrage **Vertex getVertex(String pID)**

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

Anfrage **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.

Anfrage **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.

Anfrage **`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.

Anfrage **`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.

Anfrage **`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.

Auftrag **`void setAllVertexMarks(boolean pMark)`**

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

Anfrage **`boolean allVerticesMarked()`**

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

Auftrag **`void setAllEdgeMarks(boolean pMark)`**

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

Anfrage **`boolean allEdgesMarked()`**

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

Anfrage **`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`

Konstruktor	**`Vertex(String pID)`** Ein neues Objekt vom Typ `Vertex` mit der ID `pID` wird erstellt. Seine Markierung hat den Wert `false`.
Anfrage	**`String getID()`** Die Anfrage liefert die ID des Knotens als `String`.
Auftrag	**`void setMark(boolean pMark)`** Der Auftrag setzt die Markierung des Knotens auf den Wert `pMark`.
Anfrage	**`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

Konstruktor **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.

Auftrag **void setWeight(double pWeight)**

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

Anfrage **double getWeight()**

Die Anfrage liefert das Gewicht der Kante als double.

Anfrage **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.

Auftrag **void setMark(boolean pMark)**

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

Auftrag **void setWeight(double pWeight)**

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

Anfrage **boolean isMarked()**

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

4.3 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`

Konstruktor **`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.

Auftrag **`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.

Anfrage **`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.

Anfrage **`String getErrorMessage()`**

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

Auftrag **`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`

Anfrage **`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]).

Anfrage **`String[] getColumnNames()`**

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

Anfrage **`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.

Anfrage **`int getRowCount()`**

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

Anfrage **`int getColumnCount()`**

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

4.4 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

Konstruktor **`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.

Auftrag **`void send(String pMessage)`**

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

Anfrage **`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.

Auftrag **`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

Konstruktor **`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.

Anfrage **`boolean isConnected()`**

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

Auftrag **`void send(String pMessage)`**

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

Auftrag **`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.

Auftrag **`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

Konstruktor **`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.

Anfrage **`boolean isOpen()`**

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

Anfrage **`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`.

Auftrag **`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.

Auftrag **`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.

Auftrag **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.

Auftrag **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.

Auftrag **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.

Auftrag **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.

Auftrag **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.