

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

		<code>length() indexOf(String str) substring(int beginIndex) substring(int beginIndex, int endIndex) charAt(int index) equals(Object anObject) compareTo(String anotherString) startsWith(String prefix)</code>
--	--	---

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) JOIN ... ON

UNION

AS

NULL

Vergleichsoperatoren: =, <>, >, <, >=, <=, LIKE, BETWEEN, IN, IS

NULL

Arithmetische Operatoren: +, -, *, /, (...)

Logische Verknüpfungen: AND, OR, NOT

Funktionen: COUNT, SUM, MAX, MIN

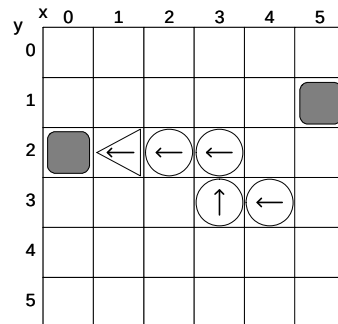
SQL-Abfragen über eine und mehrere verknüpfte Tabellen

Verschachtelte SQL-Ausdrücke vorkommen.

3. Klassendiagramme

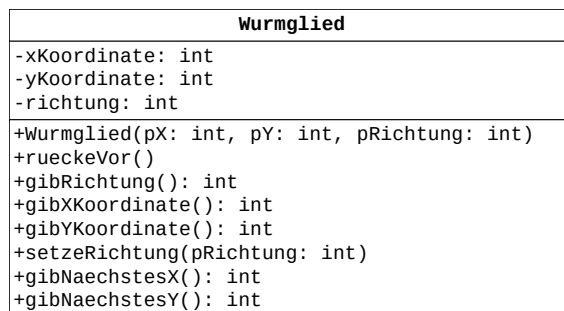
Klassendiagramme beschreiben die vorhandenen Klassen mit ihren Attributen und Methoden sowie die Beziehungen der Klassen untereinander.

Im Folgenden soll eine Variante eines Computerspiels (Darstellung: Schlange auf einem Spielfeld) modelliert werden:

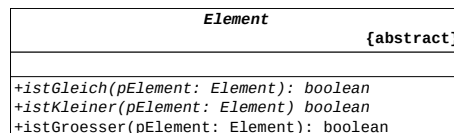


Klassen

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.

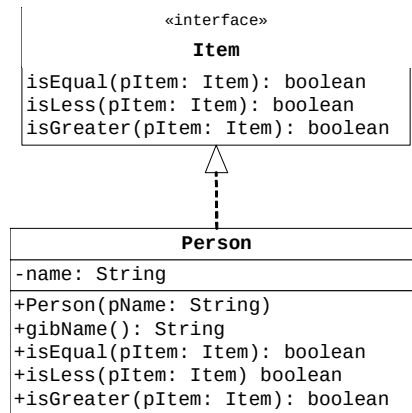


Bei **abstrakten Klassen**, also Klassen, von denen kein Objekt erzeugt werden kann, wird unter den Klassennamen im Diagramm {abstract} geschrieben. Abstrakte Methoden, also Methoden, für die keine Implementierungen angegeben werden und die nicht aufgerufen werden können, werden in Kursivschrift dargestellt.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

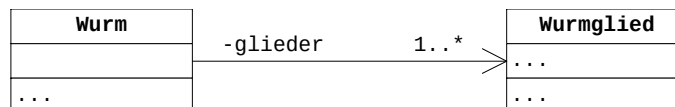
Assoziation

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:

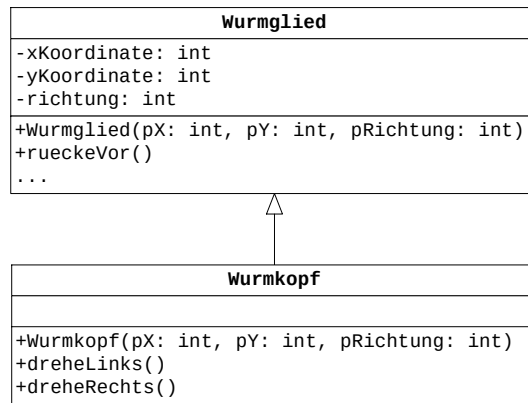
- 1 genau ein assoziiertes Objekt
- 0..1 kein oder ein assoziiertes Objekt
- 0..* beliebig viele assoziierte Objekte
- 1..* mindestens ein, beliebig viele assoziierte Objekte



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.



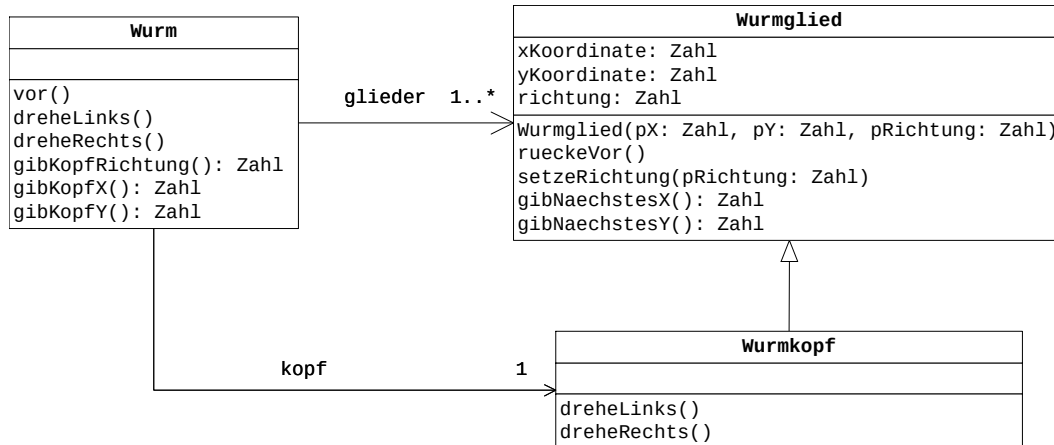
Die Klasse **Wurmkopf** spezialisiert hier die Klasse **Wurmglied**.

Entwurfsdiagramm

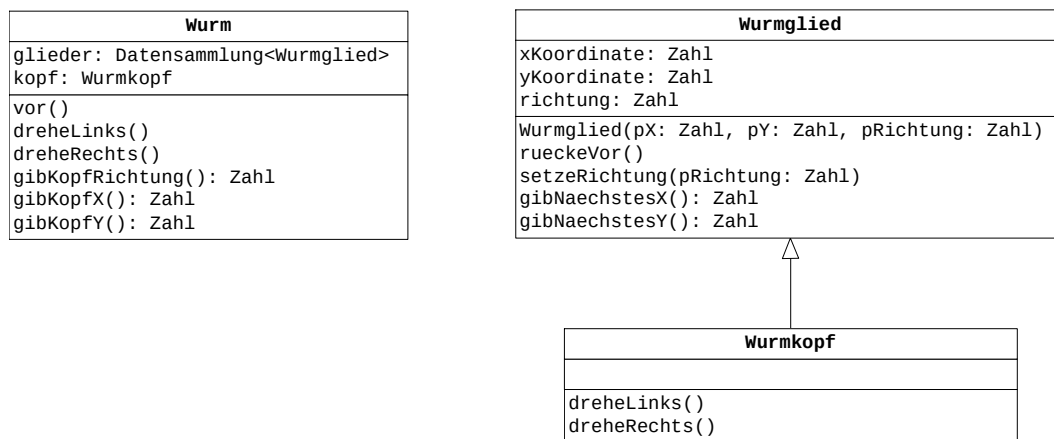
Bei einem Entwurf werden die in der Auftragssituation vorkommenden Objekte identifiziert und ihren Klassen zugeordnet.

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

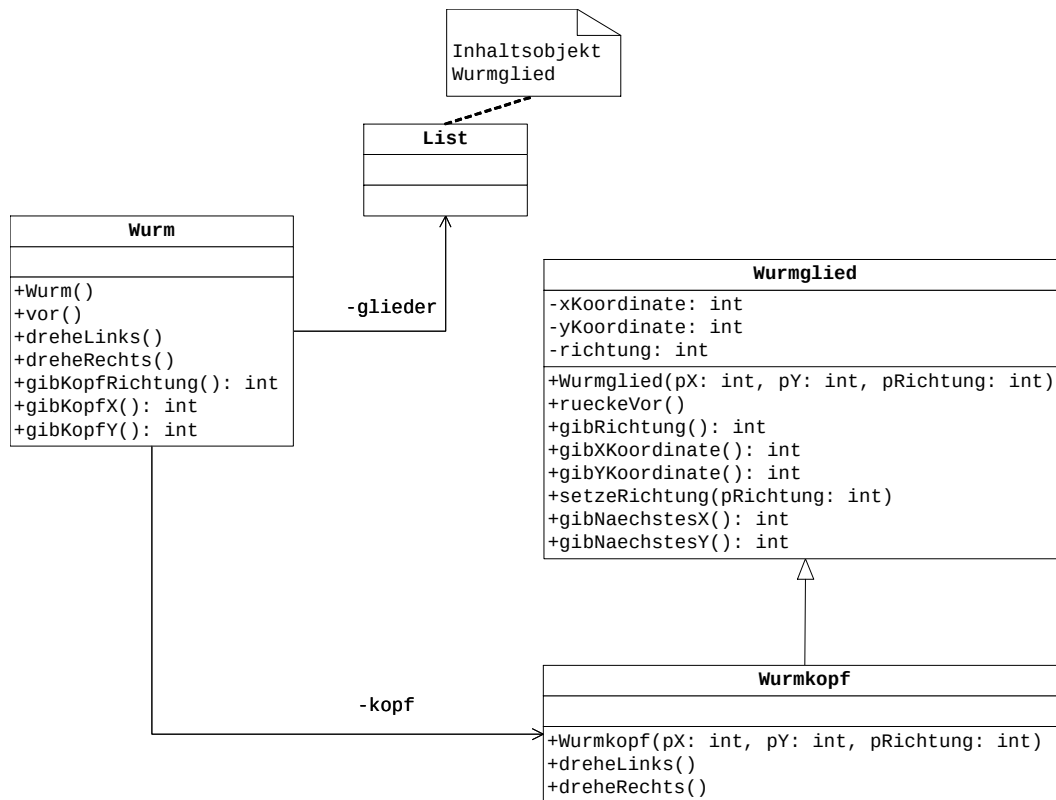


Beide Darstellungen drücken den gleichen Sachverhalt aus.

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.

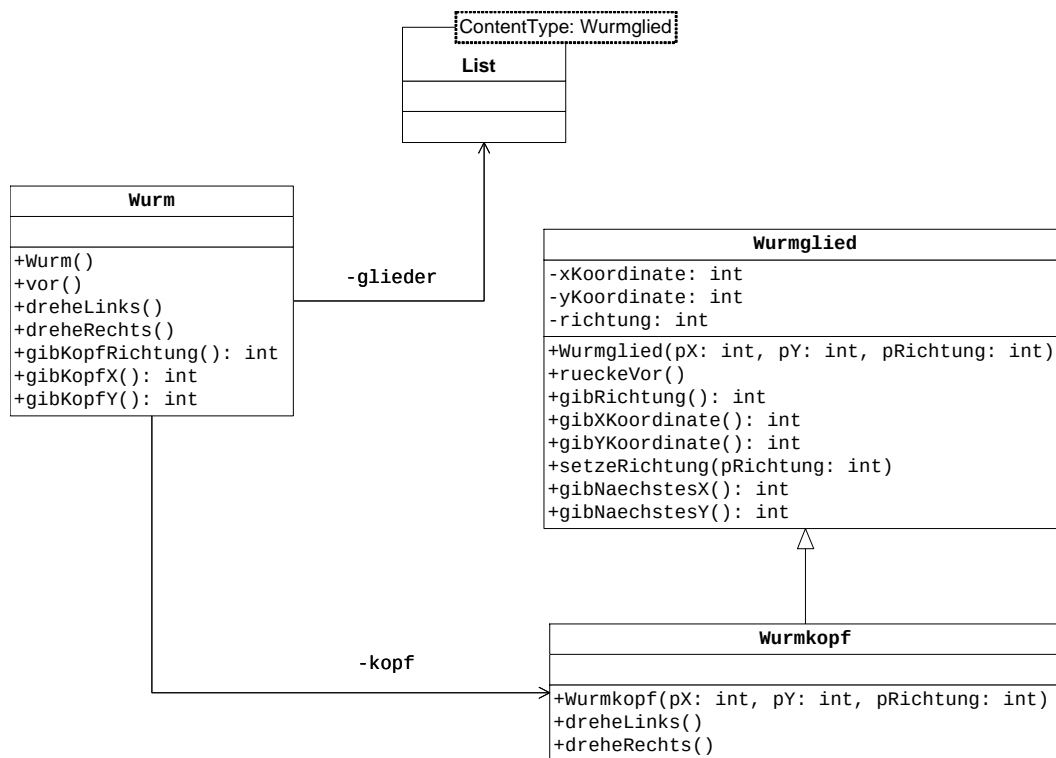


Erläuterung:

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<ContentType>()`

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<ContentType>()**

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<ContentType>()`

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 `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.

[illegible]

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>**

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>`

Konstruktor `BinarySearchTree<ContentType>()`

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:

Auftrag **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.

Auftrag **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.

Auftrag **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.

Graphen

Graphen

Ein ungerichteter Graph besteht aus einer Menge von Knoten und einer Menge von Kanten. Die Kanten verbinden jeweils zwei Knoten und können ein Gewicht haben.

Die Klasse GraphNode

Objekte der Klasse GraphNode sind Knoten eines Graphen. Ein Knoten hat einen Namen und kann markiert werden.

Dokumentation der Klasse GraphNode

Konstruktor **GraphNode(String pName)**

Ein Knoten mit dem Namen pName wird erzeugt. Der Knoten ist nicht markiert.

Auftrag **void mark()**

Der Knoten wird markiert, falls er nicht markiert ist, sonst bleibt er unverändert.

Auftrag **void unmark()**

Die Markierung des Knotens wird entfernt, falls er markiert ist, sonst bleibt er unverändert.

Anfrage **boolean isMarked()**

Die Anfrage liefert den Wert `true`, wenn der Knoten markiert ist, sonst liefert sie den Wert `false`.

Anfrage **String getName()**

Die Anfrage liefert den Namen des Knotens.

Die Klasse Graph

Objekte der Klasse Graph sind ungerichtete, gewichtete Graphen. Der Graph besteht aus Knoten, die Objekte der Klasse GraphNode sind, und Kanten, die Knoten miteinander verbinden. Die Knoten werden über ihren Namen eindeutig identifiziert.

Dokumentation der Klasse Graph

Konstruktor	Graph() Ein neuer Graph wird erzeugt. Er enthält noch keine Knoten.
Anfrage	boolean isEmpty() Die Anfrage liefert <code>true</code> , wenn der Graph keine Knoten enthält, andernfalls liefert die Anfrage <code>false</code> .
Auftrag	void addNode(GraphNode pNode) Der Knoten <code>pNode</code> wird dem Graphen hinzugefügt. Falls bereits ein Knoten mit gleichem Namen im Graphen existiert, wird dieser Knoten nicht eingefügt. Falls <code>pNode</code> <code>null</code> ist, verändert sich der Graph nicht.
Anfrage	boolean hasNode(String pName) Die Anfrage liefert <code>true</code> , wenn ein Knoten mit dem Namen <code>pName</code> im Graphen existiert. Sonst wird <code>false</code> zurück gegeben.
Anfrage	GraphNode getNode(String pName) Die Anfrage liefert den Knoten mit dem Namen <code>pName</code> zurück. Falls es keinen Knoten mit dem Namen im Graphen gibt, wird <code>null</code> zurück gegeben.
Auftrag	void removeNode(GraphNode pNode) Falls <code>pNode</code> ein Knoten des Graphen ist, so werden er und alle mit ihm verbundenen Kanten aus dem Graphen entfernt. Sonst wird der Graph nicht verändert.
Auftrag	void addEdge(GraphNode pNode1, GraphNode pNode2, double pWeight) Falls eine Kante zwischen <code>pNode1</code> und <code>pNode2</code> noch nicht existiert, werden die Knoten <code>pNode1</code> und <code>pNode2</code> durch eine Kante verbunden, die das Gewicht <code>pWeight</code> hat. <code>pNode1</code> ist also Nachbarknoten von <code>pNode2</code> und umgekehrt. Falls eine Kante zwischen <code>pNode1</code> und <code>pNode2</code> bereits existiert, erhält sie das Gewicht <code>pWeight</code> . Falls einer der Knoten <code>pNode1</code> oder <code>pNode2</code> im Graphen nicht existiert oder <code>null</code> ist, verändert sich der Graph nicht.
Anfrage	boolean hasEdge(GraphNode pNode1, GraphNode pNode2) Die Anfrage liefert <code>true</code> , falls eine Kante zwischen <code>pNode1</code> und <code>pNode2</code> existiert, sonst liefert die Anfrage <code>false</code> .

Anfrage	void removeEdge(GraphNode pNode1, GraphNode pNode2) Falls pNode1 und pNode2 nicht null sind und eine Kante zwischen pNode1 und pNode2 existiert, wird die Kante gelöscht. Sonst bleibt der Graph unverändert.
Anfrage	double getEdgeWeight(GraphNode pNode1, GraphNode pNode2) Die Anfrage liefert das Gewicht der Kante zwischen pNode1 und pNode2. Falls die Kante nicht existiert, wird Double.NaN (not a number) zurück gegeben.
Auftrag	void resetMarks() Alle Knoten des Graphen werden als unmarkiert gekennzeichnet.
Anfrage	boolean allNodesMarked() Die Anfrage liefert den Wert true, wenn alle Knoten des Graphen markiert sind, sonst liefert sie den Wert false. Wenn der Graph leer ist, wird true zurückgegeben.
Anfrage	List<GraphNode> getNodes() Die Anfrage liefert eine Liste, die alle Knoten des Graphen enthält.
Anfrage	List<GraphNode> getNeighbours(GraphNode pNode) Die Anfrage liefert eine Liste, die alle Nachbarknoten des Knotens pNode enthält.

4.3 Netzstrukturen

Die Klasse Connection

Objekte der Klasse `Connection` ermöglichen eine Netzwerkverbindung mit dem TCP/IP-Protokoll. Es können nach Verbindungsaufbau zu einem Server Zeichenketten (Strings) gesendet und empfangen werden. Zur Vereinfachung geschieht dies zeilenweise, d. h., beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfangen wird er entfernt.

Eine Fehlerbehandlung, z.B. ein Zugriff auf eine bereits geschlossene Verbindung, ist in dieser Klasse aus Gründen der Vereinfachung nicht vorgesehen.

Dokumentation der Klasse Connection

Konstruktor **`Connection(String pServerIP, int pServerPort)`**

Es wird eine Verbindung zum durch IP-Adresse und Portnummer angegebenen Server aufgebaut, so dass Daten gesendet und empfangen werden können.

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

Die angegebene Nachricht `pMessage` wird - um einen Zeilentrenner erweitert - an den Server versandt.

Anfrage **`String receive()`**

Es wird auf eine eingehende Nachricht vom Server gewartet und diese Nachricht zurückgegeben, wobei der vom Server angehängte Zeilentrenner entfernt wird. Während des Wartens ist der ausführende Prozess blockiert.

Auftrag **`void close()`**

Die Verbindung wird getrennt und kann nicht mehr verwendet werden.

Die Klasse Client

Über die Klasse Client werden Netzwerkverbindungen mit dem TCP/IP-Protokoll ermöglicht. Es können - nach Verbindungsaufbau zu einem Server - Zeichenketten (Strings) gesendet und empfangen werden, wobei der Empfang nebenläufig geschieht. Zur Vereinfachung geschieht dies zeilenweise, d. h., beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfangen wird er entfernt.

Die empfangene Nachricht wird durch eine Ereignisbehandlungsmethode verarbeitet, die in Unterklassen überschrieben werden muss.

Eine Fehlerbehandlung ist in dieser Klasse aus Gründen der Vereinfachung nicht vorgesehen.

Dokumentation der Klasse Client

Konstruktor **Client(String pServerIP, int pServerPort)**

Es wird eine Verbindung zum durch IP-Adresse und Portnummer angegebenen Server aufgebaut, so dass Zeichenketten gesendet und empfangen werden können.

Auftrag **void send(String pMessage)**

Die angegebene Nachricht pMessage wird - um einen Zeilentrenner erweitert - an den Server versandt.

Auftrag **void processMessage(String pMessage)**

Nachdem der Server die angegebene Nachricht pMessage gesendet hat wurde der Zeilentrenner entfernt. Der Client kann auf die Nachricht pMessage in dieser Methode reagieren. Allerdings enthält diese Methode keine Anweisungen und muss in Unterklassen überschrieben werden, damit die Nachricht verarbeitet werden kann.

Auftrag **void close()**

Die Verbindung zum Server wird getrennt und kann nicht mehr verwendet werden.

Die Klasse Server

Über die Klasse Server ist es möglich, eigene Serverdienste anzubieten, so dass Clients Verbindungen gemäß dem TCP/IP-Protokoll hierzu aufbauen können. Nachrichten werden grundsätzlich zeilenweise verarbeitet, d. h., beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfangen wird er entfernt.

Verbindungsaufbau, Nachrichtenempfang und Verbindungsende geschehen nebenläufig. Durch Überschreiben der entsprechenden Methoden kann der Server auf diese Ereignisse reagieren.

Eine Fehlerbehandlung ist in dieser Klasse aus Gründen der Vereinfachung nicht vorgesehen.

Dokumentation der Klasse Server

Konstruktor	Server(int pPortNr) Nach dem Aufruf dieses Konstruktors bietet ein Server seinen Dienst über die angegebene Portnummer an. Clients können sich nun mit dem Server verbinden.
Auftrag	void closeConnection(String pClientIP, int pClientPort) Unter der Voraussetzung, dass eine Verbindung mit dem angegebenen Client existiert, wird diese beendet. Der Server sendet sich die Nachricht processClosedConnection.
Auftrag	void processClosedConnection(String pClientIP, int pClientPort) Diese Methode ohne Anweisungen wird aufgerufen, bevor der Server die Verbindung zu dem in der Parameterliste spezifizierten Client schließt. Durch das Überschreiben in Unterklassen kann auf die Schließung der Verbindung zum angegebenen Client reagiert werden.
Auftrag	void processMessage(String pClientIP, int pClientPort, String pMessage) Der Client mit der angegebenen IP und der angegebenen Portnummer hat dem Server eine Nachricht gesendet. Dieser ruft daraufhin diese Methode ohne Anweisungen auf. Durch das Überschreiben in Unterklassen kann auf diese Nachricht des angegebenen Client reagiert werden.

Auftrag

**void processNewConnection(String pClientIP,
int pClientPort)**

Der Client mit der angegebenen IP-Adresse und der angegebenen Portnummer hat eine Verbindung zum Server aufgebaut. Der Server hat daraufhin diese Methode aufgerufen, die in dieser Klasse keine Anweisungen enthält. Durch das Überschreiben in Unterklassen kann auf diesen Neuaufbau einer Verbindung von dem angegebenen Client zum Server reagiert werden.

Auftrag

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

Wenn eine Verbindung zum angegebenen Client besteht, dann wird diesem Client die angegebene Nachricht - um einen Zeilentrenner erweitert - gesendet.

Auftrag

void sendToAll(String pMessage)

Die angegebene Nachricht wird - um einen Zeilentrenner erweitert - an alle verbundenen Clients gesendet.

Auftrag

void close()

Alle bestehenden Verbindungen werden getrennt. Der Server kann nicht mehr verwendet werden.