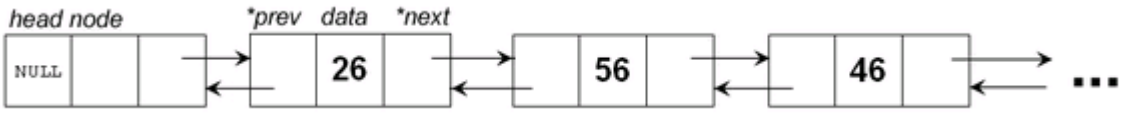
1 (Verkette) Listen

Verkettete Listen gehören zu den dynamischen Datenstrukturen, die eine Speicherung von einer im Vorhinein nicht bestimmten Anzahl von miteinander in Beziehung stehenden Werten einfacher oder zusammengesetzter Datentypen erlauben.

http://de.wikipedia.org/wiki/Liste_(Datenstruktur)

Eine Liste hat ein erstes **Element** (auch Knoten), dieses wird **Kopf** genannt. Eine Liste hat auch immer ein letztes Element. Im Gegensatz zum Array muss jedoch nicht festgelegt werden, wie groß die Liste letztlich sein wird. Ein Element einer Liste enthält Daten. Im folgenden Bild sind dies Zahlen. In Java können nicht nur Zahlen in Listen gespeichert werden, **beliebige Objekte können in Listen strukturiert** werden.

Die Listenart, die uns vornehmlich interessiert, ist eine **doppelt verkettete Liste**[1]. Das bedeutet, dass von jedem Knoten aus das vorherige und das nachfolgende Element erreicht werden kann.



1.1 Beispiel Quiz

Ein nettes Beispiel für eine Liste ist ein Quiz. Auf Karteikarten stehen jeweils eine Frage und eine Antwort. Quiz legt im Konstruktor 4 Beispielfragen und legt sie als Liste ab.

Die Methoden der Klasse Quiz verdeutlichen, wie man mit einer Liste umgehen kann.

QuizGUI ist eine freiwillige Vertiefung, auf die nicht näher eingegangen wird.

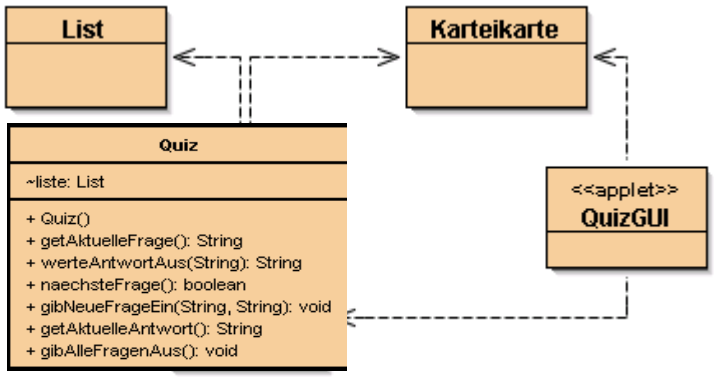


Abbildung 1: Übersicht über die Klassen Quizprojekt

1.2 Typecast: Object in Karteikarten wandeln

In der Klasse Quiz gibt es folgende Formulierung:

Karteikarte karte=(Karteikarte) liste.getItem();

oder ausführlich:

Karteikarte karte;
karte=(Karteikarte) liste.getItem();

Die Klammern um die Klasse Karteikarte drücken aus, dass der geholte Inhalt der Liste, der grundsätzlich vom Typ Objekt (in Java: Object) ist, in eine Karteikarte umgewandelt wird.

Es geschieht ein „Typecasting“ von Object zu Karteikarte.

In der Liste lassen sich beliebige Objekte speichern. Das kann String, int, Konto, Karteikarte... sein. Möglich ist auch, dass in einer Liste verschiedene Objekte gespeichert werden (ein Konto und zwei Karteikarten). Allerdings muss man aufgrund dieser Flexibilität selbst dafür sorgen, dass die zurückgegebenen Objekte zu den gewünschten Typen umgewandelt werden, was im obigen Beispiel Karteikarte ist.

Arbeitshinweise	
1.	*Nennen Sie Methoden, die Sie selbst zum Führen von Listen benötigen würden (z.B. auf einer Adressliste streicheDurch()). Vergleichen Sie Ihre benötigten Methoden mit den Methoden der Klasse „List“, die auf den folgenden Seiten beschrieben werden.*
2.	*Analysieren Sie das Klassendiagramm und die Beschreibung zu „List“.*

1 Die Zentralabikonforme Liste ist doppelt verknüpft, das sie den Nachfolger und auch den Vorgänger kennt.

Arbeitshinweise	
	Veranschaulichen Sie die Methoden, indem Sie sich eine Liste als Güterzug vorstellen und den Positionszeiger als Kran, der immer nur auf einen Waggon zugreifen kann.
3.	*Analysieren Sie das Quiz-Projekt. Schauen Sie sich zunächst die einfache Klasse Karteikarte an. Analysieren Sie anschließend die Klasse Quiz.* *Hinweis: werteFragenAus() sorgt dafür, dass Fragen mit falschen Antworten ans Ende der Liste gehängt werden.*
4.	*Schreiben Sie die Methode „steckeAktuelleKarteGanzNachHinten()“, die die aktuelle Karte ganz nach hinten schiebt. (analog: nach vorne)*
5.	*Schreiben Sie die Methode „bringeKarteUmEinePositionNachVorne()“, die die aktuelle Karte um eins nach vorne schiebt.*
6.	*Implementieren Sie die folgenden Methoden:* *1. gibRueckwaertsAus(), die ähnlich wie die Ausgabe alle Fragen auf der Konsole ausgibt, diesmal aber rückwärts (letzte Frage zuerst).* *2. gibLaengeDerListe() , die eine int-Zahl mit der Anzahl der Listenelemente zurückgibt.* *3. geheZuFrageNr(int x), die den Positionszeiger auf die x-te Frage setzt.*
7.	*Schreiben Sie die Methode „sucheFrage(String Antwort)“, die die Frage zu einer Antwort zurückgibt.*

2 Modellieren einer Zugübersicht (vgl. Bahnhof)

An größeren Bahnhöfen oder U-Bahn haltestellen gibt es seit einigen Jahren einen besonderen Service: Elektronische Hinweisschilder zeigen die Züge an, die als nächstes abfahren werden. Dabei wird neben dem Ziel auch das Gleis, die Abfahrtszeit inkl. Verspätung erfasst. Diese Servicetafeln benötigen folgende Funktionalität:

Es gibt 3 verschiedene Ausgaberoutinen

a) Alle erfassten Züge müssen angezeigt werden können

b) Auf spezielle Tafeln können nur die nächsten 10 Züge angezeigt werden.

c) Alle Züge, die an ein Gleis einfahren (z.B. Gleis 5), können angezeigt werden

Zur Verwaltung gehören die folgenden Routinen

1) Die Verspätungszeit und die Gleisnummer können verändert werden

2) Ein Zug kann komplett gestrichen werden (wenn er abgefahren ist oder ausfällt)

3) Ein neuer Zug kann eingefügt werden, dabei wird automatisch erkannt, an welche Position er gehört. Fährt er später ab als alle anderen, so wird er ans Ende der Liste gehängt.

Zusatz

4) Ist für ein Gleis die identische Abfahrtszeit angegeben, soll ein Warnhinweis angezeigt werden.

5) Der nächste Zug, der abfährt, soll bei der Ausgabe mit einem Stern versehen werden.

Arbeitshinweise	
1.	*Lesen Sie zunächst alleine die Aufgabenbeschreibung und überlegen Sie sich grob eine Realisierung.*
2.	*Diskutieren Sie in Ihrer Entwicklergruppe, welche Eigenschaften und Methoden die Klasse „Zug“ enthalten muss. Einigen Sie sich auf eine Variante, die dann für alle gilt. Fixieren Sie diese als UML-Diagramm.*
	Teilen Sie sich die Arbeit auf. Wählen Sie einen Projektkoordinator, der die Übersicht über das Projekt erhält. An seinem Rechner werden die einzelnen Programmteile zusammengefügt. Implementieren Sie das Projekt.

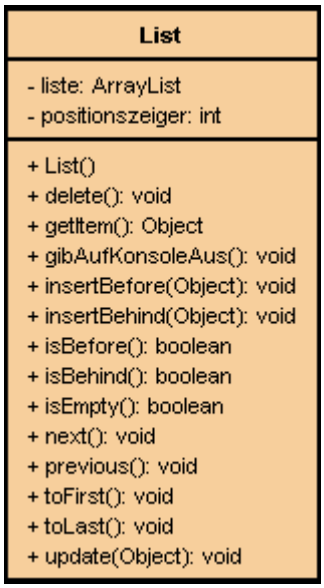
3 Anhang: Die Klasse List

Die Klasse List

Objekte der Klasse *List* verwalten beliebige Objekte nach einem Listenprinzip. Ein interner Positionszeiger wird durch die Listenstruktur bewegt, seine Position markiert ein aktuelles Objekt. Die Lage des Positionszeigers kann abgefragt, verändert und die Objektinhalte an den Positionen können gelesen oder verändert werden.

Die Klasse *List* stellt Methoden in folgender Syntax zur Verfügung:

```
public List()

public boolean isEmpty()

public boolean isBefore()

public boolean isBehind()

public void next()

public void previous()

public void toFirst()

public void toLast()

public Object getItem()

public void update (Object pObject)

public void insertBefore (Object pObject)

public void insertBehind (Object pObject)

public void delete()
```



Text 1: Methoden und Eigenschaften der „NRW-Zentralabi-Vorgaben-Liste“

5

Auftrag	`insertBefore (Object pObject)`
Vorher	Der Positionszeiger steht nicht vor der Liste.
Nachher	Ein neues Listenelement mit dem entsprechenden Objekt ist angelegt und vor der aktuellen Position in die Liste eingefügt worden. Der Positionszeiger steht hinter dem eingefügten Element.

Auftrag	`insertBehind (Object pObject)`
Vorher	Der Positionszeiger steht nicht hinter der Liste.
Nachher	Ein neues Listenelement mit dem entsprechenden Objekt ist angelegt und hinter der aktuellen Position in die Liste eingefügt worden. Der Positionszeiger steht vor dem eingefügten Element.

Auftrag	`delete()`
Vorher	Der Positionszeiger steht nicht vor oder hinter der Liste.
Nachher	Das aktuelle Listenelement ist gelöscht. Der Positionszeiger steht auf dem Element hinter dem gelöschten Element, bzw. hinter der Liste, wenn das gelöschte Element das letzte Listenelement war.

Dokumentation der Methoden der Klasse List

Konstruktor	`List()`
Nachher	Eine leere Liste ist angelegt. Der interne Positionszeiger steht vor der leeren Liste
Anfrage	`isEmpty(): boolean`
Nachher	Die Anfrage liefert den Wert ***true***, wenn die Liste keine Elemente enthält, sonst liefert sie den Wert ***false***.
Anfrage	`isBefore(): boolean`
Nachher	Die Anfrage liefert den Wert ***true***, wenn der Positionszeiger vor dem ersten Listenelement oder vor der leeren Liste steht, sonst liefert sie den Wert ***false***.
Anfrage	`isBehind(): boolean`
Nachher	Die Anfrage liefert den Wert ***true***, wenn der Positionszeiger hinter dem letzten Listenelement oder hinter der leeren Liste steht, sonst liefert sie den Wert ***false***.
Auftrag	`next()`
Nachher	Der Positionszeiger ist um eine Position in Richtung Listenende weitergerückt, d.h. wenn er vor der Liste stand, wird das Element am Listenanfang zum aktuellen Element, ansonsten das jeweils nachfolgende Listenelement. Stand der Positionszeiger auf dem letzten Listenelement, befindet er sich jetzt hinter der Liste. Befand er sich hinter der Liste, hat er sich nicht verändert.
Auftrag	`previous()`
Nachher	Der Positionszeiger ist um eine Position in Richtung Listenanfang weitergerückt, d.h. wenn er hinter der Liste stand, wird das Element am Listenende zum aktuellen Element, ansonsten das jeweils vorhergehende Listenelement. Stand der Positionszeiger auf dem ersten Listenelement, befindet er sich jetzt vor der Liste. Befand er sich vor der Liste, hat er sich nicht verändert.
Auftrag	`toFirst()`
Nachher	Der Positionszeiger steht auf dem ersten Listenelement. Falls die Liste leer ist befindet er sich jetzt hinter der Liste.
Auftrag	`toLast()`
Nachher	Der Positionszeiger steht auf dem letzten Listenelement. Falls die Liste leer ist befindet er sich jetzt vor der Liste.
Anfrage	`getItem(): Object`
Nachher	Die Anfrage liefert den Wert des aktuellen Listenelements bzw. *null*, wenn die Liste keine Elemente enthält, bzw. der Positionszeiger vor oder hinter der Liste steht.
Auftrag	`update (Object pObject)`
Vorher	Die Liste ist nicht leer. Der Positionszeiger steht nicht vor oder hinter der Liste.
Nachher	Der Wert des Listenelements an der aktuellen Position ist durch *pObject* ersetzt.

3.1 Quelle Anhang

- http://www.learn-line.nrw.de/angebote/abitur-wbk-08/download/if-datentypen-opjetk-ansatz-java.pdf bzw. http://www.u-helmich.de/inf/BlueJ/kurs121/seite15/abi-list.html

3.2 Vertiefende Links

- http://en.wikipedia.org/wiki/Linked_list
- http://de.wikipedia.org/wiki/Typecast