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 Klasse Queue

Objekte der Klasse **`Queue`** (Warteschlange) verwalten beliebige Objekte nach dem First-In-First-Out-Prinzip, d.h., das zuerst abgelegte Objekt wird als erstes wieder entnommen.

Dokumentation der Klasse Queue

Konstruktor **`Queue()`**
Eine leere Schlange wird erzeugt.

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

Auftrag **`void enqueue(Object pObject)`**
Das Objekt `pObject` wird an die Schlange angehängt. Falls `pObject` 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 **`Object front()`**
Die Anfrage liefert das erste Objekt der Schlange. Die Schlange bleibt unverändert. Falls die Schlange leer ist, wird `null` zurück gegeben.

Die Klasse Stack

Objekte der Klasse **`Stack`** (Keller, Stapel*)* verwalten beliebige Objekte nach dem Last-In-First-Out-Prinzip, d.h., das zuletzt abgelegte Objekt wird als erstes wieder entnommen.

Dokumentation der Klasse Stack

Konstruktor **`Stack()`**
Ein leerer Stapel wird erzeugt.

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

Auftrag **`void push(Object pObject)`**
Das Objekt `pObject` wird oben auf den Stapel gelegt. Falls `pObject` 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 **`Object top()`**
Die Anfrage liefert das oberste Stapelobjekt. Der Stapel bleibt unverändert. Falls der Stapel leer ist, wird `null` zurück gegeben.

Die Klasse List

Objekte der Klasse **`List`** verwalten beliebig viele, linear angeordnete Objekte. 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

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

Auftrag **`void append(Object pObject)`**

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

Auftrag **`void insert(Object pObject)`**

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`), wird ein neues Objekt 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 `pObject` 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 `pObject` gleich `null` ist, bleibt die Liste unverändert.

Auftrag **`void concat(List 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.