```
 /**
 * <p>
 * Materialien zu den zentralen NRW-Abiturpruefungen im Fach Informatik ab 2017.
 * </p>
 * <p>
 * Generische Klasse List<ContentType>
 * </p>
 * <p>
 * Objekt der generischen Klasse List verwalten beliebig viele linear
 * angeordnete Objekte vom Typ ContentType. Auf hoechstens ein Listenobjekt,
 * aktuellesObjekt genannt, kann jeweils zugegriffen werden.<br />
 * Wenn eine Liste leer ist, vollstaendig durchlaufen wurde oder das aktuelle
 * Objekt am Ende der Liste geloescht wurde, gibt es kein aktuelles Objekt.<br
/>
 * Das erste oder das letzte Objekt einer Liste koennen durch einen Auftrag zum
 * aktuellen Objekt gemacht werden. Ausserdem kann das dem aktuellen Objekt
 * folgende Listenobjekt zum neuen aktuellen Objekt werden. <br />
 * Das aktuelle Objekt kann gelesen, veraendert oder geloescht werden. Ausserdem
 * kann vor dem aktuellen Objekt ein Listenobjekt eingefuegt werden.
 * </p>
 *
 * @author Qualitaets- und UnterstuetzungsAgentur - Landesinstitut fuer Schule,
Materialien zum schulinternen Lehrplan Informatik SII
 * @version Generisch_06 2015-10-25
 */
public class List<ContentType> {

  /* --------- Anfang der privaten inneren Klasse -------------- */

  private class ListNode {

    private ContentType contentObject;
    private ListNode next;

    /**
     * Ein neues Objekt wird erschaffen. Der Verweis ist leer.
     *
     * @param pContent das Inhaltsobjekt vom Typ ContentType
     */
    private ListNode(ContentType pContent) {
      contentObject = pContent;
      next = null;
    }

    /**
     * Der Inhalt des Knotens wird zurueckgeliefert.
     *
     * @return das Inhaltsobjekt des Knotens
     */
    public ContentType getContentObject() {
      return contentObject;
    }

    /**
     * Der Inhalt dieses Kontens wird gesetzt.
     *
     * @param pContent das Inhaltsobjekt vom Typ ContentType
     */
    public void setContentObject(ContentType pContent) {
      contentObject = pContent;
    }

    /**
     * Der Nachfolgeknoten wird zurueckgeliefert.
```

```
     *
     * @return das Objekt, auf das der aktuelle Verweis zeigt
     */
    public ListNode getNextNode() {
      return this.next;
    }

    /**
     * Der Verweis wird auf das Objekt, das als Parameter uebergeben
     * wird, gesetzt.
     *
     * @param pNext der Nachfolger des Knotens
     */
    public void setNextNode(ListNode pNext) {
      this.next = pNext;
    }

  }

  /* ----------- Ende der privaten inneren Klasse -------------- */

  // erstes Element der Liste
  ListNode first;

  // letztes Element der Liste
  ListNode last;

  // aktuelles Element der Liste
  ListNode current;

  /**
   * Eine leere Liste wird erzeugt.
   */
  public List() {
    first = null;
    last = null;
    current = null;
  }

  /**
   * Die Anfrage liefert den Wert true, wenn die Liste keine Objekte enthaelt,
   * sonst liefert sie den Wert false.
   *
   * @return true, wenn die Liste leer ist, sonst false
   */
  public boolean isEmpty() {
    // Die Liste ist leer, wenn es kein erstes Element gibt.
    return first == null;
  }

  /**
   * Die Anfrage liefert den Wert true, wenn es ein aktuelles Objekt gibt,
   * sonst liefert sie den Wert false.
   *
   * @return true, falls Zugriff moeglich, sonst false
   */
  public boolean hasAccess() {
    // Es gibt keinen Zugriff, wenn current auf kein Element verweist.
    return current != null;
  }

  /**
   * 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
 * Ausfuehrung des Auftrags kein aktuelles Objekt, d.h. hasAccess() liefert
 * den Wert false.
 */
public void next() {
  if (this.hasAccess()) {
    current = current.getNextNode();
  }
}

/**
 * Falls die Liste nicht leer ist, wird das erste Objekt der Liste aktuelles
 * Objekt. Ist die Liste leer, geschieht nichts.
 */
public void toFirst() {
  if (!isEmpty()) {
    current = first;
  }
}

/**
 * Falls die Liste nicht leer ist, wird das letzte Objekt der Liste
 * aktuelles Objekt. Ist die Liste leer, geschieht nichts.
 */
public void toLast() {
  if (!isEmpty()) {
    current = last;
  }
}

/**
 * Falls es ein aktuelles Objekt gibt (hasAccess() == true), wird das
 * aktuelle Objekt zurueckgegeben, andernfalls (hasAccess() == false) gibt
 * die Anfrage den Wert null zurueck.
 *
 * @return das aktuelle Objekt (vom Typ ContentType) oder null, wenn es
 *         kein aktuelles Objekt gibt
 */
public ContentType getContent() {
  if (this.hasAccess()) {
    return current.getContentObject();
  } else {
    return null;
  }
}

/**
 * Falls es ein aktuelles Objekt gibt (hasAccess() == true) und pContent
 * ungleich null ist, wird das aktuelle Objekt durch pContent ersetzt. Sonst
 * geschieht nichts.
 *
 * @param pContent
 *            das zu schreibende Objekt vom Typ ContentType
 */
public void setContent(ContentType pContent) {
  // Nichts tun, wenn es keinen Inhalt oder kein aktuelles Element gibt.
  if (pContent != null && this.hasAccess()) {
    current.setContentObject(pContent);
  }
}

/**
 * Falls es ein aktuelles Objekt gibt (hasAccess() == true), wird ein neues
 * Objekt vor dem aktuellen Objekt in die Liste eingefuegt. Das aktuelle
```

```
   * Objekt bleibt unveraendert. <br />
   * Wenn die Liste leer ist, wird pContent in die Liste eingefuegt und es
   * gibt weiterhin kein aktuelles Objekt (hasAccess() == false). <br />
   * Falls es kein aktuelles Objekt gibt (hasAccess() == false) und die Liste
   * nicht leer ist oder pContent gleich null ist, geschieht nichts.
   *
   * @param pContent
   *            das einzufuegende Objekt vom Typ ContentType
   */
  public void insert(ContentType pContent) {
    if (pContent != null) { // Nichts tun, wenn es keinen Inhalt gibt.
      if (this.hasAccess()) { // Fall: Es gibt ein aktuelles Element.

        // Neuen Knoten erstellen.
        ListNode newNode = new ListNode(pContent);

        if (current != first) { // Fall: Nicht an erster Stelle einfuegen.
          ListNode previous = this.getPrevious(current);
          newNode.setNextNode(previous.getNextNode());
          previous.setNextNode(newNode);
        } else { // Fall: An erster Stelle einfuegen.
          newNode.setNextNode(first);
          first = newNode;
        }

      } else { //Fall: Es gibt kein aktuelles Element.

        if (this.isEmpty()) { // Fall: In leere Liste einfuegen.

          // Neuen Knoten erstellen.
          ListNode newNode = new ListNode(pContent);

          first = newNode;
          last = newNode;
        }

      }
    }
  }

  /**
   * Falls pContent gleich null ist, geschieht nichts.<br />
   * Ansonsten wird ein neues Objekt pContent am Ende der Liste eingefuegt.
   * Das aktuelle Objekt bleibt unveraendert. <br />
   * Wenn die Liste leer ist, wird das Objekt pContent in die Liste eingefuegt
   * und es gibt weiterhin kein aktuelles Objekt (hasAccess() == false).
   *
   * @param pContent
   *            das anzuhaengende Objekt vom Typ ContentType
   */
  public void append(ContentType pContent) {
    if (pContent != null) { // Nichts tun, wenn es keine Inhalt gibt.

      if (this.isEmpty()) { // Fall: An leere Liste anfuegen.
        this.insert(pContent);
      } else { // Fall: An nicht-leere Liste anfuegen.

        // Neuen Knoten erstellen.
        ListNode newNode = new ListNode(pContent);

        last.setNextNode(newNode);
        last = newNode; // Letzten Knoten aktualisieren.
      }
```

```
  }
}

/**
 * Falls es sich bei der Liste und pList um dasselbe Objekt handelt,
 * pList null oder eine leere Liste ist, geschieht nichts.<br />
 * Ansonsten wird die Liste pList an die aktuelle Liste angehaengt.
 * Anschliessend wird pList eine leere Liste. Das aktuelle Objekt bleibt
 * unveraendert. Insbesondere bleibt hasAccess identisch.
 *
 * @param pList
 *            die am Ende anzuhaengende Liste vom Typ List<ContentType>
 */
public void concat(List<ContentType> pList) {
  if (pList != this && pList != null && !pList.isEmpty()) { // Nichts tun,
  // wenn pList und this identisch, pList leer oder nicht existent.

    if (this.isEmpty()) { // Fall: An leere Liste anfuegen.
      this.first = pList.first;
      this.last = pList.last;
    } else { // Fall: An nicht-leere Liste anfuegen.
      this.last.setNextNode(pList.first);
      this.last = pList.last;
    }

    // Liste pList loeschen.
    pList.first = null;
    pList.last = null;
    pList.current = null;
  }
}

/**
 * Wenn die Liste leer ist oder es kein aktuelles Objekt gibt (hasAccess()
 * == false), geschieht nichts.<br />
 * Falls es ein aktuelles Objekt gibt (hasAccess() == true), wird das
 * aktuelle Objekt geloescht und das Objekt hinter dem geloeschten Objekt
 * wird zum aktuellen Objekt. <br />
 * Wird das Objekt, das am Ende der Liste steht, geloescht, gibt es kein
 * aktuelles Objekt mehr.
 */
public void remove() {
  // Nichts tun, wenn es kein aktuelle Element gibt oder die Liste leer ist.
  if (this.hasAccess() && !this.isEmpty()) {

    if (current == first) {
      first = first.getNextNode();
    } else {
      ListNode previous = this.getPrevious(current);
      if (current == last) {
        last = previous;
      }
      previous.setNextNode(current.getNextNode());
    }

    ListNode temp = current.getNextNode();
    current.setContentObject(null);
    current.setNextNode(null);
    current = temp;

    //Beim loeschen des letzten Elements last auf null setzen.
    if (this.isEmpty()) {
      last = null;
    }
```

```
    }
  }

  /**
   * Liefert den Vorgaengerknoten des Knotens pNode. Ist die Liste leer, pNode
   * == null, pNode nicht in der Liste oder pNode der erste Knoten der Liste,
   * wird null zurueckgegeben.
   *
   * @param pNode
   *          der Knoten, dessen Vorgaenger zurueckgegeben werden soll
   * @return der Vorgaenger des Knotens pNode oder null, falls die Liste leer
ist,
   *          pNode == null ist, pNode nicht in der Liste ist oder pNode der
erste Knoten
   *          der Liste ist
   */
  private ListNode getPrevious(ListNode pNode) {
    if (pNode != null && pNode != first && !this.isEmpty()) {
      ListNode temp = first;
      while (temp != null && temp.getNextNode() != pNode) {
        temp = temp.getNextNode();
      }
      return temp;
    } else {
      return null;
    }
  }

}
```