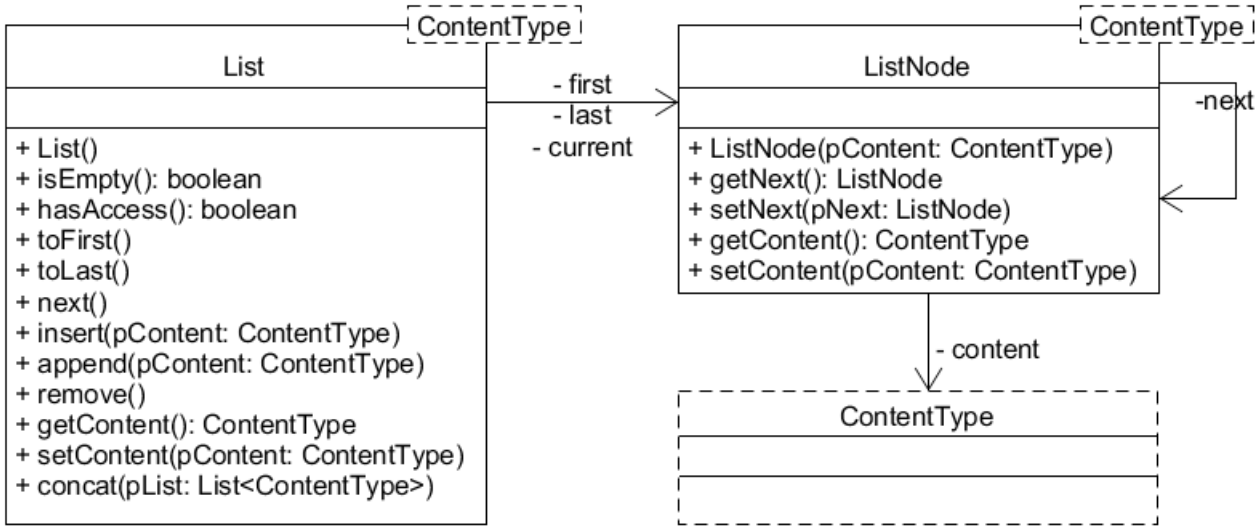
Die Klasse „List“

Die Klasse „List“ ist eine Verallgemeinerung der Klassen „Queue“ und „Stack“. Die einzelnen Elemente der Liste werden ebenfalls durch Knoten („Nodes“) hintereinander gehängt. Anders als bei der Queue oder beim Stack kann man jedoch Elemente **an beliebiger Stelle einfügen** oder löschen.

Um den Zugriff auf die Elemente der Liste an beliebiger Stelle zu ermöglichen, gibt es neben „first“ und „last“ eine weitere Referenz „**current**“. Diese Referenz kann man mit den Methoden **toFirst()** und **next()** an jede beliebige Stelle der Liste bewegen. An dem Knoten, auf dem die current-Referenz gerade steht, kann man verschiedene Aktionen ausführen:
den Inhalt an diesem Knoten abfragen, ein neues Element einfügen oder den Knoten löschen.

Wenn „current“ auf das letzte Element zeigt, und man ihn mit next() noch eine Position weiter schiebt, zeigt „current“ nicht mehr auf ein gültiges Element. Zu dieser Zeit ist kein Zugriff auf die Elemente der Liste möglich. Um diesen Fall abzufragen hat die Liste die Funktion **hasAccess()**. Diese gibt true zurück, wenn „current“ auf einen gültigen Knoten zeigt, ansonsten false.

Die Klasse „List“ ist im folgenden **Klassendiagramm** dargestellt. Sie ist generisch (wie Queue und Stack), d.h. man kann Listen für beliebige Inhalte deklarieren.
Für die Anwendung sind nur die Methoden der Klasse List interessant.
Wichtig sind auch die verschiedenen Referenzen: first, last, current.
Next und content haben die gleiche Funktion wie bei der Queue und dem Stack.



Methoden der Klasse List

`List()` (Konstruktor)
Eine leere Liste wird erzeugt. D.h. first, last und current zeigen jeweils auf null.

`boolean isEmpty()`
Gibt true zurück, falls es keine Elemente in der Liste gibt, sonst false.

`boolean hasAccess()`
Gibt true zurück, falls current auf einen gültigen Knoten zeigt.
D.h. bei einer leeren List ist hasAccess() == false. Erst wenn es mindestens ein Element in der Liste gibt und toFirst() aufgerufen wird, zeigt current auf einen Knoten, und hasAccess() gibt true zurück.

`void toFirst()`
Setzt current auf das erste Element der Liste. Wenn die Liste leer ist, bleibt current = null.

`void next()`
Falls current auf einen gültigen Knoten zeigt (hasAccess() == true), rückt current weiter auf seinen Nachfolger. Sonst passiert nichts.
Falls current auf den letzten Knoten zeigt, zeigt er anschließend auf null, d.h. hasAccess() wird false.

`void insert(ContentType pContent)`
Falls current auf einen gültigen Knoten zeigt (hasAccess() == true), wird ein neuer Knoten mit pContent vor current eingefügt. Current zeigt weiterhin auf den Knoten, auf den es vorher gezeigt hat.
Falls die Liste leer ist, wird ebenfalls ein neuer Knoten eingefügt. Current bleibt dann null (wie bei append())
Falls die Liste nicht leer ist UND hasAccess() == false ist, passiert nichts (es wird kein Knoten eingefügt).
Sonderfall: Falls der Parameter pContent null ist, wird kein neuer Knoten eingefügt.

`void append(ContentType pContent)`
Hängt ein neues Element ans Ende der Liste.
Current wird dadurch nicht beeinflusst (wenn current vorher den Wert null hatte, bleibt es weiterhin null).
Sonderfall: Falls der Parameter pContent null ist, wird kein neuer Knoten angehängt.

`void remove()`
Falls current auf einen gültigen Knoten zeigt (hasAccess() == true), wird dieser Knoten gelöscht.
Current wandert dann zu dessen Nachfolger.
Falls es keinen Nachfolger gibt, ist anschließend hasAccess() == false.
Falls current nicht auf einen gültigen Knoten zeigt, passiert nichts.

`ContentType getContent()`
Falls current auf einen gültigen Knoten zeigt (hasAccess() == true), wird das Objekt an diesem Knoten zurückgegeben. Sonst wird null zurückgegeben.

`void setContent(ContentType pContent)`
Falls current auf einen gültigen Knoten zeigt (hasAccess() == true), wird das Inhaltsobjekt an diesem Knoten durch pContent ausgetauscht. Das Inhaltsobjekt, das vorher an diesem Knoten hing, wird gelöscht (falls es nicht vorher an andere Stelle gesichert wurde).

`void concat(List<ContentType> pList)`
Eine zweite Liste (pList) wird ans Ende dieser Liste angehängt. Die Liste muss den gleichen Typ von Inhaltsobjekten haben. Auf diese Weise können zwei Listen miteinander verbunden werden.
Current bleibt davon unberührt (zeigt weiterhin auf den gleichen Knoten bzw. auf null).
Die Liste, auf die der Parameter pList zeigt, wird zu einer leeren Liste.
Das bedeutet dass man mit pList anschließend keinen Zugriff mehr auf die Elemente der Liste hat.

Aufgabe 1

Beantworte die folgenden Verständnisfragen (ohne Programmierung).

a) In eine Liste wurden bereits drei Elemente eingefügt. Wenn man das zweite Element in der Liste löschen möchte, welche Methoden (der Klasse List) ruft man dazu nacheinander auf?

b) Eine Liste hat drei Elemente. Wenn man zwischen dem ersten und zweiten Element der Liste ein weiteres Element einfügen möchte, welche Methoden ruft man dazu nacheinander auf?

c) In welchen Fällen liefert hasAccess() den Wert false? In welchen Fällen den Wert true? Liste alle denkbaren Fälle auf.

d) Warum kann man mit insert() kein Element am Ende der Liste einfügen? Wie kann man trotzdem ein Objekt ans Ende der Liste hängen?

Aufgabe 2

Die folgende Beispielprogramm erzeugt eine Liste und fügt einige Elemente ein. Zeichne je ein Objektdiagramm der Situation 1. nach Zeile 07, 2. nach Zeile 15, 3. nach Zeile 22.

```
01  public class Beispiel
02  {
03     private List<String> farbenListe;
04
05     public Beispiel()
06     {
07        farbenListe = new List();                    // 1. Diagramm
08     }
09
10     public void main()
11     {
12        farbenListe.append("Rot");
13        farbenListe.append("Blau");
15        farbenListe.append("Gelb");                  // 2. Diagramm
16
17        farbenListe.toFirst();
18        farbenListe.next();
19        farbenListe.remove();
20        farbenListe.next();
21        farbenListe.insert("Grün");
22        farbenListe.append("Orange");                // 3. Diagramm
23     }
24  }
```

Aufgabe 3

Verwende die ausgeteilte BlueJ-Vorlage.
Sie enthält die Klasse List, die vom Schulministerium zur Verfügung gestellt wird.
Implementiere die Klasse Aufgabe3.

a) Deklariere und erzeuge eine Referenz für ein Objekt der Klasse List, das Strings enthalten soll. Das Objekt wird im Konstruktor erzeugt.

b) Implementiere die Methode `fuellen()`.
Sie fügt die Strings „Rot“, „Blau“, „Gelb“ und „Grün“ (in dieser Reihenfolge) in die Liste ein.

c) Implementiere die Methode `vertauschen1()`.
Die Methode soll das zweite und das vierte Element der Liste vertauschen, ohne die Methode setContent() zu benutzen – getContent() ist jedoch erlaubt und erforderlich.
Annahme: die Methode weiß nichts über die einzelnen Inhaltsobjekte der Liste.
D.h. sie kann nicht annehmen, dass an der zweiten Stelle „Blau“ oder „Gelb“ steht.
Es wird zur Vereinfachung allerdings angenommen, dass es mindestens vier Elemente gibt.

d) Implementiere eine neue Version der Methode: `vertauschen2()`.
Diese macht keinerlei Annahmen (d.h. es kann sein, dass es weniger als vier Elemente gibt).
Falls die Liste nicht genügend Elemente hat, soll die Methode nichts tun.

e) Zusatz: Alternative Implementierung `vertauschen3(int pos1, int pos2)`
Die Methode vertauscht die Elemente der Liste an den Stellen pos1 und pos2.
Beispiel: pos1 = 0 und pos2 = 2 vertauscht das erste und das dritte Element der Liste.
Für diese Aufgabe darfst du die Methode setContent() benutzen.
Falls die Liste nicht genügend Elemente hat, geschieht nichts.
Beachte auch andere mögliche Sonderfälle für die Parameter pos1 und pos2.

Achtung, für Aufgabe e) wird die Konsole nicht benötigt (auch für keine andere Teilaufgabe).
Woher die Methode die Werte für pos1 und pos2 erhält, ist für die Aufgabe nicht relevant.