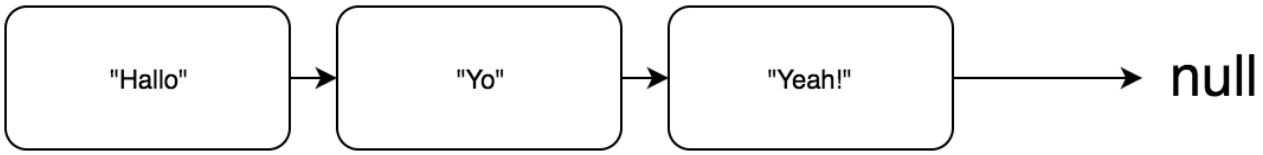
Einfach verkettete Listen

Einfach verkettete Listen und Bäume sind Datenstrukturen, die in Java auf einfache Weise abgebildet werden können.

Zunächst ist es hilfreich, sich einfach verkettete Listen vorstellen zu können:



Dies ist eine einfach verkettete Liste aus Strings. Ein Element zeigt immer auf das nächste. Wenn ein Element auf `null` zeigt, bedeutet das, dass die Liste zu Ende ist.

In Java ist es immer so, dass für ein einzelnes Listenelement eine Klasse gegeben ist oder angelegt wird. Beispiel:

```
// Ob diese Klasse Element oder Knoten oder sonst wie heißt, ist egal.
class Element{
	// In diesem Attribut wird die Information abgelegt, die ein Listenelement
	// trägt
	private String name;
	// Hier könnten noch mehr oder andere Attribute stehen

	// Das hier ist das wichtigste Attribut:
	private Element next;
	// Es ist so wichtig, da man mit dieser Klasse sonst keine einfach
	// verkettete Liste abbilden kann

	// Der Konstruktor ist fast immer so
	public Element(String name, Element next){
		this.name = name;
		this.next = next;
	}

	// Dieser Konstruktor ist optional, aber bequemer an manchen Stellen:
	public Element(String name){
		this.name = name;
		this.next = null;
	}

	// Setter- und Getter
	public void setNext(Element e){
		next = e;
	}

	public Element getNext(){
		return next;
	}

	public void setName(String name){
		this.name = name;
	}
```

```
	public String getName(){
		return name;
	}
}
```

Das ist die Grundvoraussetzung, um die Datenstruktur einfach verkettete Liste verwenden zu können. Es ist wichtig, zu verstehen, dass eine Instanz davon sowohl ein einzelnes Listenelement als auch eine ganze Liste sein kann. Das ist davon abhängig, was der Wert des Attributs next ist.

Wenn das Attribut next den Wert null hat, handelt es sich um ein einzelnes Listenelement oder je nach Betrachtung um eine Liste der Länge 1. Wenn das Attribut next der Instanz **a** als Wert eine andere Instanz von Element hat, ist **a** Teil einer Liste.

Wofür braucht man das überhaupt?

Wir kennen bereits Arrays. Arrays sind praktisch, wenn man viele Werte abspeichern möchte und vorher auch schon weiß, wie viele Elemente abgespeichert werden. Und sie sind praktisch, wenn man schnell auf einzelne Elemente darin zugreifen möchte. Leider sind Arrays sehr unpraktisch, wenn es darum geht, Elemente dazwischen zu löschen (also zu entfernen) oder neue Elemente dazwischen einzufügen. Oder überhaupt auch am Anfang oder am Ende einzufügen.

Bei einer einfach verketteten Liste ist das hingegen viel einfacher (Elemente einfügen und entfernen). Nachteil von einfach verketteten Listen ist allerdings, dass es damit schwieriger ist, auf ein bestimmtes Element zuzugreifen. Um beispielsweise auf das vierte Listenelement zuzugreifen, muss man zunächst die ersten drei Listenelemente aufrufen und über ihr next eben zum vierten Listenelement kommen. Bei Arrays sagt man da einfach feld[3].

Das heißt also: in der „realen Welt“ (für Programmierer) muss immer entschieden werden, welche Datenstruktur für welche Aufgabenstellung am besten geeignet ist. In der Prüfung nimmt man einfach das, was vorgegeben ist. Hier muss man sich selten Gedanken darüber machen, welche Datenstruktur am besten geeignet ist.

Methoden

In diesem Abschnitt werden einzelne Methoden erklärt, die in der Klasse Element eingefügt werden, um mit der Datenstruktur zu arbeiten. Diese Methoden sind normalerweise in Aufgabenstellungen verlangt. Es geht hier darum, dass du sie alle schon mal gesehen hast, um sie dann in der Prüfung einfach aus dem Gedächtnis hinschreiben zu können.

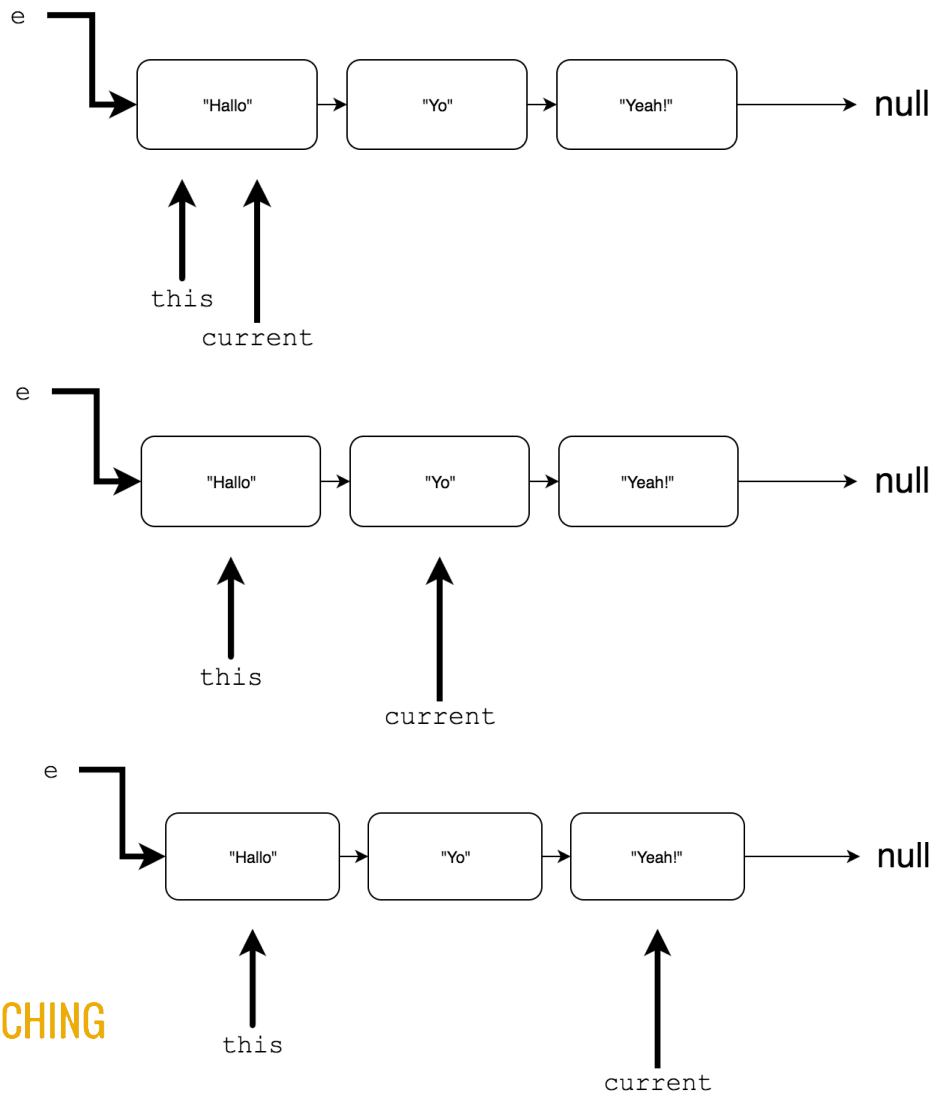
int getLength()

Beispiel für die Aufgabenstellung: schreibe eine Methode, die die Länge der Liste zurückgibt.

```
// innerhalb der Klasse Element
public int getLength(){
	int result = 1;

	Element current = this;

	while(current.getNext() != null){
		current = current.getNext();
		result++;
	}

	return result;
}
```

Erklärung des Ablaufs:

```
Element e = new Element("Hallo", new Element("Yo", new Element("Yeah")));
System.out.println(e.getLength());
```



Element getLast()

In Aufgabenstellungen wird nach dieser Methode verlangt, wenn es heißt, man soll das letzte Element ermitteln.

```
public Element getLast(){
	Element current = this;

	while(current.getNext() != null){
		current = current.getNext();
	}

	return current;
}
```

void append(Element toInsert)

In Aufgabenstellungen wird nach dieser Methode verlangt, wenn es heißt, man soll ein Element am Ende der Liste einfügen / anfügen

```
public void append(Element toInsert){
	// toInsert könnte natürlich eine gesamte Liste sein, nicht nur ein
	// Element
	getLast().setNext(toInsert);
}
```

Es ist häufig hilfreich, innerhalb von Methoden eigene selbst programmierte Methoden zu verwenden.

Element get(int index)

Es sollen alle Elemente ab einer bestimmten Stelle ermittelt werden? Dann das hier machen:

```
public Element get(int index){
	if(index < 0 || index >= getLength()){
		return null;
	}

	int currentIndex = 0;
	Element current = this;

	while(currentIndex < index){
		current = current.getNext();
		currentIndex++;
	}

	return current;
}
```

Wenn man nur ein Element möchte, sollte man das hier in der letzten Zeile machen:

```
return new Element(current.getName(), null);
```

public String toString()

Die Liste soll ausgegeben werden können? Dann diese Methode implementieren:

```
// nur innerhalb der Element-Klasse
public String toString(){
	String result = "(";

	Element current = this;

	// Achtung: Reihenfolge der Anweisungen innerhalb der Schleife
	while(current.getNext() != null){
		result += current.getName() + ", ";

		current = current.getNext();
	}

	return result + current.getName() + ")";
}

Beispiel für die Verwendung:
System.out.println(list);
//--> (a, b, c, d)
```

public Element getByName(String name)

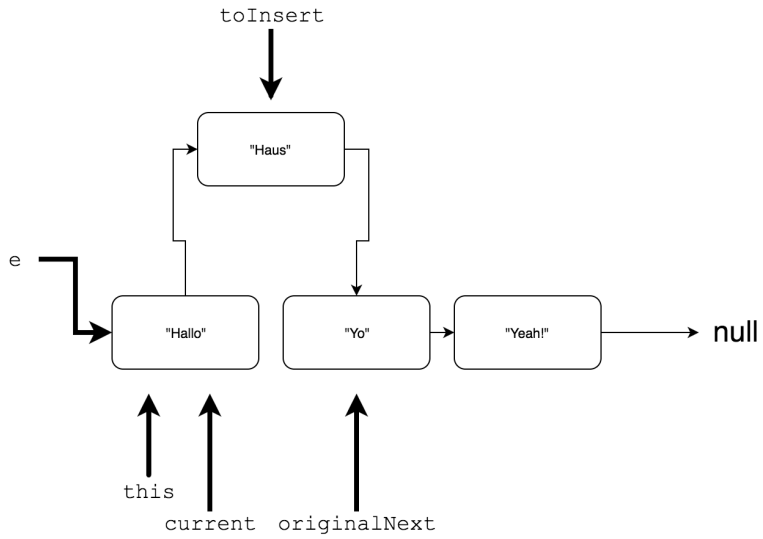
Nach dieser Methode ist in der Aufgabenstellung gefragt, wenn man ein Element in der Liste anhand eines Attributs finden soll.

```
public Element getByName(String name){
	Element current = this;

	while(current.getNext() != null){
		if(current.getName().equals(name)){
			return current;
		}

		current = current.getNext();
	}

	// nicht gefunden
	return null;
}

// ganz anderes Beispiel, weil das Attribut anders heißt und andere Klasse (Node)
public Node getByNummer(int nummer){
	Node current = this;

	while(current.getNext() != null){
		if(current.getNummer() == nummer){
			return current;
		}

		current = current.getNext();
	}

	// nicht gefunden
	return null;
}
```

void insertAt(int destinationIndex, Element toInsert)

Diese Methode ist dafür geeignet, ein Element an einer vorgegebenen Position (Index) in der Liste einzufügen.

```
// Innerhalb der Klasse Element
public void insertAt(int destinationIndex, Element toInsert){
	// Dadurch, dass diese Methode innerhalb der Klasse Element steht, kann
	// kein Element an der Stelle 0 eingefügt werden. Dafür braucht man aber
	// auch diese Methode nicht. Das geht so:
	// Element newList = new Element("neues Element ganz vorn", alteListe);

	if(toInsert == null || toInsert.getLength() != 1 || destinationIndex == 0){
		// diese Methode funktioniert nur richtig, wenn ein
		// einziges Element übergeben wird
		return;
	}

	int currentIndex = 1;

	Element current = this;

	// Achtung: Reihenfolge der Anweisungen innerhalb der Schleife
	while(currentIndex < destinationIndex){
		current = current.getNext();

		currentIndex++;
	}

	Element originalNext = current.getNext();
	current.setNext(toInsert);
	toInsert.setNext(originalNext);
}
```



Beispielaufruf:

```
e.insertAt(1, new Element("Haus"));
```

void remove(int index)

Mit dieser Methode kann ein Element aus der Liste an der angegebenen Stelle entfernt werden.

```
// Innerhalb der Klasse Element
public void remove(int index){
	// Dadurch, dass diese Methode innerhalb der Klasse Element steht, kann
	// kein Element an der Stelle 0 entfernt werden. Dafür braucht man aber
	// auch diese Methode nicht. Das geht so:
	// Element newList = alteListe.getNext();

	if(index == 0 || index >= getLength()){
		return;
	}

	int currentIndex = 1;

	Element current = this;

	// Achtung: Reihenfolge der Anweisungen innerhalb der Schleife
	while(currentIndex < index){
		current = current.getNext();

		currentIndex++;
	}

	current.setNext(current.getNext().getNext());
}
```

Anwendung

In einer main-Methode …

```
Element e = new Element("A", null);
// e ist nun eine Liste mit einem Element

// Das hier geht aber genauso, weil wir ja einen zweiten Konstruktor haben, der
// nur einen Parameter hat und next dann standardmäßig auf null setzt:
e = new Element("A");

// hier eine längere Liste
Element list = new Element("a", new Element("b", new Element("c",
	new Element("d"))));

System.out.println(list);
// (a, b, c, d)
System.out.println(list.getLength());
// 4
System.out.println(list.getLast());
// (d)

list.insertAt(1, new Element("a2"));
System.out.println(list);
// (a, a2, b, c, d)

list.append(new Element("z"));
System.out.println(list);
// (a, a2, b, c, d, z)
```

```
// Unterschiede der folgenden Zeilen ganz genau beachten und nachvollziehen!

Element list2 = new Element("Auto", new Element("Baustelle", new
Element("Dachdecker")));

System.out.println(list2);
// (Auto, Baustelle, Dachdecker)

Element result = list2.sortedInsert(new Element("Ahne"));
System.out.println(result);
// (Ahne, Auto, Baustelle, Dachdecker)

System.out.println(list2);
// (Auto, Baustelle, Dachdecker)

result = list2.sortedInsert(new Element("Holz"));
System.out.println(result);
// (Auto, Baustelle, Dachdecker, Holz)

System.out.println(list2);
// (Auto, Baustelle, Dachdecker, Holz)

list2 = list2.sortedInsert(new Element("Beamter"));
System.out.println(list2);
// (Auto, Baustelle, Beamter, Dachdecker, Holz)
```

Und wenn es eine zweite Klasse gibt?

Häufig soll eine zweite Klasse programmiert werden, die etwa so aussieht:

```
class Liste{
	private Element start;

	public Liste(Element start){
		this.start = start;
	}

	// hier sollen Methoden wie getLast etc. programmiert werden
}
```

Dann sollte bedacht werden, dass in den Methoden nicht mehr **this** benutzt werden kann. Es sollte dann mit **start** gearbeitet werden, weil diese Variable den Start der Liste enthält.