Sammlungsklassen Übersicht

Sammlungstrukturen allgemein

Wozu?

Sammlungsklassen nutzt man, wenn man mehrere Objekte gleichen Typs unter einem Namen ansprechen will.

Arrays

Wie?

Deklaration des Array-Objektes	Moebel[] moebel;
Definition [Erzeugen] des Array-Objektes	moebel= new Moebel[3]
Definition [Erzeugen] der Objekte im Array	moebel[0] = new Schrank(…); ...

Bild

Index [Position]	0	1	2
Inhalt	erstes Schrankobjekt	zweites Schrankobjekt	drittes Schrankobjekt

schnell

Arrays bieten einen schnellen Zugriff auf die in ihnen abgespeicherten Objekte. Das gilt sowohl für den lesenden als auch für den schreibenden Zugriff.

statische Länge

Bei den Arrays sind zwar die Inhalte veränderbar, sie haben aber eine statische Länge. Wenn man also während der Arbeit ein weiteres, zusätzliches Objekt hinzufügen will, geht das nur mit einem neuen Array, das dann ausreichende Länge haben muss und in das man die schon vorhandenen Inhalte hinüber kopieren muss.

unkomfortabel

Arrays bieten keine besondere Funktionalität an. [Was damit gemeint ist: s.u.]

Warum gibt es das dann noch?

Arrays sind eine sehr alte Datenstruktur, die von vielen Programmierern als vorhanden erwartet wird. Sie ist sehr viel mehr an der Hardware orientiert, als die anderen strukturierten Datentypen. Im Hintergrund arbeiten diese [bei Java] mit Arrays.

ArrayList – zunächst untypisiert

Wie?

Deklaration des ArrayList-Objektes	ArrayList moebel;
Definition [Erzeugen] des ArrayList-Objektes	moebel= new ArrayList()
Definition [Erzeugen] der Objekte in der ArrayList	moebel.add(new Schrank(…)); ...

Bild

Index [Position]	0	1	2
Inhalt	erstes Objekt	zweites Objekt	drittes Objekt

Die bildliche Darstellung sieht also gar nicht so anders aus. Zum Typ s.u.

flexibel

Die Länge von ArrayList – Objekten ist dynamisch. Ist der bisherige Umfang nicht ausreichend, wird ohne Zutun des Programmierers und des Anwenders im Hintergrund mehr Platz geschaffen.

komfortabel

Für die üblicherweise mit Sammlungsstrukturen durchzuführenden Aufgaben gibt es jeweils passende Methoden [wichtig: add, contains, get, set, remove, size, isEmpty, equals].
Auf Objekte in einer ArrayList kann ebenfalls indiziert [also z.B. get(1)] zugegriffen werden.
Aber:

Iterator

Echte Sammlungsstrukturen stellen eine Methode iterator() bereit, mit der man auf die Elemente der Sammlung eines nach dem anderen zugreifen kann ohne dass man sich um diesen Zugriff selbst kümmern muss.

Typ ist Object

Wenn man nichts anderes vorgibt [s.u.], dann ist der Typ der Objekte in der Sammlung Object, selbst dann wenn man sicher weiß, dass es alles z.B. Moebel-Objekte sind.
Daher wurden typisierte ArrayLists eingeführt:

ArrayList – typisiert

Wie?

Deklaration des typisierten ArrayList-Objektes	ArrayList**<Moebel>** moebel;
Definition [Erzeugen] des ArrayList-Objektes	moebel= new ArrayList<Moebel>()
Definition [Erzeugen] der Objekte in der ArrayList	moebel.add(new Schrank(…)); ...

Bild

Index [Position]	0	1	2
Inhalt	erstes Moebelobjekt	zweites Moebelobjekt	drittes Moebelobjekt

Da Schrank von Moebel erbt, kann man sagen:"ein Schrankobjekt ***ist ein*** Moebelobjekt."
Daher könnte oben auch jeweils Schrankobjekt stehen

flexibel

s.o.

komfortabel

s.o. und:
Bei einer typisierten ArrayList ist bei allen Zugriffen sichergestellt, dass sie ein Objekt des vorgegebenen Typs erwarten bzw. liefern.

Iterator

s.o.
Beschafft man sich auch einen typisierten Iterator, dann liefert er auch diesen Typ wirklich zurück.

LinkedList – gleich der typisierte Fall

Wie?

Deklaration des LinkedList-Objektes	LinkedList<Moebel> moebel;
Definition [Erzeugen] des LinkedList-Objektes	moebel= new LinkedList<Moebel>()
Definition [Erzeugen] der Objekte in der LinkedList	moebel.add(new Schrank(…)); ...

Bild

Index [Position]	0			1			2		
Inhalt	*-> null*	**erstes Moebelobjekt**	*-> Nachfolger*	*-> Vorgänger*	**zweites Moebelobjekt**	*-> Nachfolger*	*-> Vorgänger*	**drittes Moebelobjekt**	*-> null*

Die bildliche Darstellung sieht hier deutlich anders aus.

flexibel

Bei einer LinkedList geht alles prinzipiell genau so flexibel. Besonders ist hier nur, dass in den einzelnen Elementen jeweils ein Verweis auf den Vorgänger und einer auf den Nachfolger vermerkt sind. So kann es sein, dass die Art der Ablage nicht der hier dargestellten Reihenfolge entspricht.

komfortabel

Für die üblicherweise mit Sammlungsstrukturen durchzuführenden Aufgaben gibt es auch im Fall der LinkedList jeweils passende Methoden [wichtig: add, contains, get, set, remove, size, isEmpty, equals].

Iterator

Echte Sammlungsstrukturen stellen eine Methode iterator() bereit, also auch die LinkedList. Entsprechendes gilt für die Typisierung.

Weitere Typen von Sammlungsstrukturen

Es gibt weitere Typen von Sammlungsstrukturen, die hier aber nicht dargestellt sind.