Dynamische Arrays

Die im vorigen Kapitel beschriebenen Statischen Arrays haben einen schwerwiegenden Nachteil: ihre Größe kann zur Programmlaufzeit nicht angepasst werden. In der Programmierpraxis verleitet dies oft dazu, mehr Speicherplatz zu reservieren als benötigt.

Dynamische Arrays bieten demgegenüber folgende Vorteile:

- Die Größe passt sich automatisch an, wenn neue Elemente hinzukommen.
- Sie sind nicht typgebunden und können daher verschiedenartige Objekttypen aufnehmen.

Es gibt mehrere Arten von dynamischen Datenbehältern, die in der Java-Klassenbibliothek als `Collections` (Sammlungen) bezeichnet werden. Die häufig benutzte Klasse `Vector` wird im Folgenden beschrieben.

Klasse `Vector`

Die Benutzung der Klasse `Vector` erfordert die Einbindung von

```
import java.util.Vector;
```

Die Implementierung eines Vectors ist kein Array, sondern eine verkettete Liste – eine Datenstruktur die zur Laufzeit ihre Größe verändern kann und außerdem unterschiedliche Objekt-Typen aufnehmen kann.

Die Klasse Vector bietet vier Konstruktoren:

```
1. public Vector( )
2. public Vector(int anfangsGroesse)
3. public Vector(int anfangsGroesse, int inkrement)
4. public Vector(Collection coll)
```

Beim Erzeugen eines Vectors kann die Anfangsgröße angegeben werden. Der Speicherplatz für die Elemente wird reserviert, ein Wachstumsfaktor (der bei Überschreiten der Anfangsgröße ausgeführt wird) oder eine bereits existierende Collection (die in einen dynamischen Vektor verwandelt wird) kann optional übergeben werden. In den meisten Fällen wird der parameterlose Konstruktoraufruf genügen.

```
Vector stVek = new Vector( ); // Neuer dynamischer Vektor
```

Elementzugriff

- Elemente eines Vectors werden prinzipiell als Objekte der Klasse **`Object`** (Wurzelklasse der Javahierarchie) behandelt. Es können keine elementaren Datentypen (int, double, char) gespeichert werden - gegebenenfalls müssen Hüllklassen (Integer, Double etc) verwendet werden (siehe Beispiel unten).

- Beim Wiederauslesen eines Elements aus einem Vector muss eine Abwärts-Konvertierung (*downcast*) in den entsprechenden Ausgabetyp vorgenommen werden. Da ein *downcast* prinzipiell typunsicher ist (siehe Kapitel Klassenvererbung / Polymorphie), müssen die Konvertierungsmittel sorgsam angewendet werden (z.B. Typprüfung mit **`instanceof`**).

- Der Elementzugriff erfolgt immer mit Methoden und ist deshalb nicht ganz so komfortabel wie bei einem statischen Array, dessen Elemente über den Index (in eckigen Klammern) angesprochen werden. Die Zugriffsmethoden sind synchronisiert (gegen gleichzeitigen Zugriff abgesichert), was die Zugriffsgeschwindigkeit etwas verringert.

Speichern von Elementen in einem Vector

Es gibt fünf Methoden, die zum Speichern von Elementen in einem Vector eingesetzt werden können.

void	**add** (int index, Object element) Fügt das angegebene Objekt an der angegebenen Index-Position in den Vector ein.
boolean	**add**(Object element) Hängt das angegebene Objekt an das Ende des Vectors an.
void	**addElement**(Object element) Hängt das angegebene Objekt an das Ende des Vectors an und vergrössert den Vector.
boolean	**addAll**(Collection c) Fügt alle Elemente der angegebenen Collection an das Ende des Vectors an.
boolean	**addAll**(int index, Collection c) Fügt alle Elemente der angegebenen Collection an der angegebenen Position des Vectors an.

Die Einfüge-Methoden unterscheiden sich durch die Möglichkeit, eine (nullbasierte) Index-Position anzugeben, an der das Element eingefügt wird, bzw. durch die Möglichkeit, eine vorhandene Sammlung (Collection) von Objekten auf einen Schlag in einen Vector einzufügen.

Die am leichtesten benutzbare Methode ist **`add( Object )`**, die das angegebene Objekt an das Ende des Vectors anhängt. Wenn das Objekt an eine bestimmte Stelle des Vectors eingefügt werden soll, wird die Methode **`add( index, object)`** benutzt. Die Methode **`addElement(Object)`** hängt das Objekt an das Ende des Vectors an und vergrössert den Vector zugleich um den beim Konstruktoraufruf (optional) angegebenen Inkrementwert.

Verschiedene Objekttypen in einem Vector

Im folgenden Beispiel wird gezeigt, wie unterschiedliche Objekttypen in einen **`Vector`** eingefügt werden. Zunächst werden Objekte vom Typ Student erzeugt und mit der Methode **`add( )`** an das Ende des Vectors angehängt. Die Objekte werden dabei automatisch in Objekte der Klasse **`Object`** konvertiert. Dann werden mehrere Objekte des Typs **`Integer`** an bestimmten Index-Positionen eingefügt..

```
//////////////////////////

import java.util.Vector;

class Student {

     private int    matrikelnr;
     private double einkommen;
     private String vorname, nachname;

     public Student(String v, String n, int m, double e) {
        vorname    = v;
        nachname   = n;
        einkommen  = e;
        matrikelnr = m;
     }
     public void gibAus() {
        System.out.println("Student:    "
                           + this.vorname + " " + nachname);
        System.out.println("Einkommen:  " + einkommen);
        System.out.println("MatrikelNr: " + matrikelnr);
     }
}
//////////////////////////////////////////////////////

class DynVektor1 {

 public static void main(String args[]) {

   Student eins = new Student("Hanna", "Mueller", 1223, 2000.0);
   Student zwei = new Student("Hans", "Meier", 2331, 3000.0);

    Vector stVek = new Vector( );

    stVek.add( eins );
    stVek.add( zwei );

   for(int j = 0; j < 12; j++)
       stVek.add( j, new Integer (j) );

    System.out.println("Laenge: "+ stVek.size()); // 14
  }
}
```

Der Vector besteht nun aus vierzehn Elementen.

Auslesen von Elementen aus einem Vector

Zum Auslesen gespeicherter Objekte aus einem Vector gibt es in der Java-API zahlreiche Methoden, die hier nicht alle erläutert werden können; die API-Dokumentation (siehe obigen Link) listet diese vollständig auf. Die am häufigsten und am leichtesten zu merkenden Methoden sind:

Object	get(int index) Liefert das Element an der angegebenen Index-Position.
Object	elementAt(int index) Liefert das Element am angegebenen Index.
Enumeration	elements() Gibt eine Enumeration (Aufzählungs-Container) mit allen Elementen zurück.

Beim Auslesen ist zu beachten, dass jedes aus einem Vector entnommene Element stets vom Basis-Typ Object ist.

```
Object o = stVek.get(i); // oder:
Object o = stVek.elementAt(i);
```

Dieses Objekt kann natürlich erst dann sinnvoll verwendet werden, nachdem es in wieder in den konkreten Klassentyp seiner Ursprungsklasse (im obigen Beispiel Student) umgewandelt worden ist. Dies ist durch einen downcast möglich.

Ein downcast muss immer abgesichert werden, damit keine nicht-implementierten Attribute oder Methoden angesprochen werden können (siehe ausführlich Kapitel Polymorphie). Dies kann am einfachsten durch den Operator instanceof erfolgen:

```
if(o instanceof STUDENT) { ... }
```

Dieser Operator (keine Methode!) liefert den bool'schen Wert true, falls das Objekt vom Typ der angegebenen Klasse ist. Falls nicht, wird die folgende Konvertierungsanweisung übersprungen. Falls doch, dann kann bedenkenlos konvertiert,

```
STUDENT t = (STUDENT)o;
```

und dann eine klassenspezifische Methode aufgerufen werden:

```
t.gibAus();
```

Im folgenden Beispiel wird der Vector elementweise durchlaufen (for-Schleife), dabei wird jedes Element extrahiert, auf seinen Klassentyp geprüft (Student oder INTEGER) und dann entsprechend ausgegeben.

Auslesen eines Elements aus einem Vector mit Typprüfung und Umwandlung in den korrekten Ursprungstyp (downcast):

```
for (int i = 0; i < stVek.size(); i++) {

            Object o = stVek.get(i);

            if( o instanceof Student) {

                  Student t = (Student)o;
                  t.gibAus();

                  ((Student)o).gibAus(); //Alternativ!
            }

            if( o instanceof Integer) {

                  Integer k = (Integer)o;

                  System.out.println("Integer: " + k);

                  System.out.println("Integer: " + (Integer)o);
            }
  }
```