Überblick

Reihungen

- Ein Array (Reihung, Feld) ist ein Tupel von Komponentengliedern gleichen Typs, auf die über einen Index direkt zugegriffen werden kann
- Formal handelt es sich um eine Folge, d.h. ein Array von Objekten eines bestimmten Typs `type` ist ein Element aus `type*`
- Der Zugriff auf das i-te Element kann durch die Projektion formalisiert formalisiert werden
- Eine Reihung mit n Komponenten vom Typ `<type>` ist eine Abbildung von der Indexmenge I_n auf die Menge `<type>`

Achtung

- Achtung 1:
  Bei den meisten Programmiersprachen beginnt die Indexmenge bei 0, d.h. für eine n-elementige Reihung gilt die Indexmenge $I_n = \{0, \ldots, n-1\}$
- Achtung 2:
  In Java sind Arrays *semidynamisch*, d.h., ihre Größe kann zur Laufzeit (=dynamisch) festgesetzt werden, danach aber nicht mehr geändert werden (=statisch), d.h. dynamisches Wachstum ist nicht möglich!!!

Reihungen

- Beispiel: Eine **`char`**-Reihung `gruss` der Länge 13:

`gruss`:	'H'	'e'	'l'	'l'	'o'	','	' '	'W'	'o'	'r'	'l'	'd'	'!'
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12

ist formal eine Abbildung

$$\texttt{gruss} : \{0, 1, \ldots, 12\} \to \textbf{char}$$

$$i \mapsto \begin{cases} \text{'H'} & \text{falls} \quad i = 0 \\ \text{'e'} & \text{falls} \quad i = 1 \\ & \vdots \\ \text{'!'} & \text{falls} \quad i = 12 \end{cases}$$

Reihungen

- Der Typ eines Arrays, das den Typ `<type>` enthält, wird in Java als `<type>[]` notiert
- Beispiel: ein Array mit Objekten vom Typ **`int`** ist vom Typ **`int`**`[]`
- Variablen und Konstanten vom Typ `<type>[]` können wie gewohnt vereinbart werden:

```
<type>[] variablenName;
```

  (Konstanten wie immer mit dem Zusatz **`final`**)
- Vorsicht: Arrays sind sog. *Referenztypen*, d.h. auf dem Zettel einer Variable steht nicht der Wert (also das Array direkt) sondern eine Referenz (Arrays werden nämlich, da sie i.d.R. unterschiedlich groß sind, auf einem besonderen Bereich im Speicher verwaltet – dazu aber später noch mehr)
- Dies führt zu einem call-by-reference-Effekt (auch dazu später mehr)

Reihungen

- Referenztypen werden bei Ihrer Initialisierung *erzeugt*
- Die Initialisierung (Erzeugung) eines Arrays kann auf verschieden Art und Weisen erfolgen
- Die einfachste ist, alle Elemente der Reihe nach aufzuzählen:

```
<type>[] variablenName = {element1, element2, ...}
```

wobei die einzelnen `element1`, `element2`, etc. Literale (Werte) oder Variablen vom Typ `<type>` sind

Reihungen

- *Zugriff* (Projektion) auf das `i`-te Element eines Arrays `a` notiert man in Java durch den Ausdruck `a[i]`, d.h. der Wert des Ausdrucks `variablenName[i]` ist der Wert des Arrays `variablenName` an der Stelle `i`
- Ein Array hat immer eine feste *Länge*, die in einer Konstanten `length` festgehalten wird
- Diese Konstante wird mit einem Array immer automatisch miterzeugt
- Der Name der Konstanten ist zusammengesetzt aus dem Namen des Arrays und dem Namen `length`
- Beispiel: die Länge eines Arrays `a` ist der Wert des Ausdrucks `a.length`

Reihungen

- Beispiel

```
char a = 'a';
char b = 'b';
char c = 'c';
char[] abc = {a, b, c};
System.out.print(abc[0]); // gibt den Character 'a' aus,
                          // den Wert des Array-Feldes
                          // mit Index 0. Allgemein: array[i]
                          // ist Zugriff auf das i-te Element
System.out.print(abc.length);      // gibt 3 aus
int[] zahlen = {1, 3, 5, 7, 9};
System.out.print(zahlen[3]);       // gibt die Zahl 7 aus
System.out.print(zahlen.length);   // gibt 5 aus
```

Reihungen

- Oft legt man ein Array an, bevor man die einzelnen Elemente kennt
- Die Länge muss man dabei angeben: `char[] abc = new char[3];`
- Das Schlüsselwort **new** ist hier verlangt (es bedeutet in diesem Fall, dass eine neue (leere) Referenz angelegt wird)
- Dann kann man das Array im weiteren Programmverlauf füllen:

```
abc[0] = 'a';
abc[1] = 'b';
abc[2] = 'c';
```

Reihungen

- Dass Arrays in Java *semidynamisch* sind, bedeutet: Es ist möglich, die Länge erst zur Laufzeit festzulegen
- Beispiel

```
// x ist eine Variable vom Typ int
// deren Wert bei der Ausfuehrung
// feststeht, aber nicht beim
// Festlegen bzw. Uebersetzen des Programmcodes
// (z.B. weil x ein Eingabeparameter ist)
char[] abc = new char[x];
```

Reihungen

- Was passiert, wenn man ein Array anlegt

```
int[] zahlen = new int[10];
```

  aber nicht füllt (ist das Array dann leer?)
- Es gibt in Java keine leeren Arrays
- Ein Array wird immer mit den Standardwerten des jeweiligen Typs initialisiert
- Das spätere Belegen einzelner Array-Zellen ist also immer eine Änderung eines Wertes (durch eine Wertzuweisung)

```
int[] zahlen = new int[10];
System.out.print(zahlen[3]);    // gibt 0 aus
zahlen[3] = 4;
System.out.print(zahlen[3]);    // gibt 4 aus
```

Reihungen

- Das legt den Schluss nahe, dass die einzelnen Element des Arrays wiederum Variablen (Zettel) sind, die ich beschreiben und ablesen kann
- So in etwa kann man sich das auch tatsächlich vorstellen
- In

```
int[] zahlen = { 1 , 2 , 3 }
```

ist die Variable `zahlen` ein (radierbarer) Zettel, auf dem ein Haufen weitere Zettel, nämlich die der einzelnen Elemente `zahlen[i]`, liegen (genauer die Adresse der Zettel, der einzelnen Elemente `zahlen[i]` und einem zusätzlichen (nicht-radierbaren) Zettel mit der Länge des Arrays: `zahlen.length`)

- Die einzelnen (radierbaren) Zettel auf diesem Haufen, also die Elemente, z.B. `zahlen[1]`, enthalten die konkreten Werten (z.B. `1`) bzw. sind leer, wenn das Array noch nicht initialisiert worden ist

Reihungen

- Es gelten die üblichen Eigenschaften für Zettel (Variablen/Konstanten), also z.B. ist auch soetwas erlaubt:

```
char[] abc = \{ 'a',  'b',  'c'\};
char[] de = \{ 'e',  'e'\};
abc = de; // d.h. de wird der Wert von abc zugewiesen
```

- Naja, statt einem konkreten Wert wird hier eben eine Referenz zugewiesen

Reihungen

- Auch Array-Variablen kann man als Konstanten deklarieren
- Dann kann man der Variablen keinen neuen Wert zuweisen

```
final char[] ABC = \{ 'a',  'b',  'c'\};
final char[] DE = \{ 'd',  'e'\};
ABC = DE; // ungueltige Anweisung: Compilerfehler
```

- Aber Achtung: einzelne Array-Komponenten sind normale Variablen, man kann ihnen also einen neuen Wert zuweisen

```
ABC[0] = 'd';                          // erlaubt
ABC[1] = 'e';                          // erlaubt
System.out.print(ABC.length);  // gibt 3 aus
System.out.print(ABC[0]);      // gibt 'd' aus
System.out.print(ABC[1]);      // gibt 'e' aus
System.out.print(ABC[2]);      // gibt 'c' aus
```

Reihungen

- Hä? Wie passt denn das mit unserer Intuition der Zettel(-wirtschaft) zusammen?
- Sehr gut sogar:
- Wie gesagt, eine Variable vom Typ `<type>[]` ist ein Zettel auf dem wiederum weitere Zettel (die Elemente) liegen
- Konstanten sind nicht radierbare Zettel, d.h. der Zettel `<type>[]` auf dem die anderen Zettel liegen ist dann nicht radierbar, die Zettel, die darauf liegen aber natürlich schon
- Natürlich

Reihungen

- Beispiel für einen Algorithmus mit Arrays
  - Suche ein Element `q`∈**`int`** in einem Array `a`∈**`int`**`[]` (gib an, ob das Element im Array vorhanden ist)
  - Lösungsidee 1: sequentielle Suche:

    Durchlaufe das Array `a` von Position `i = 0` bist `a.length-1`

    Falls `a[i] == q`, gibt **`true`** aus

    Falls `q` nicht gefunden wurde, gib **`false`** aus

```
public static bolean seqSearch(int[] a, int q) {
      for(i=0;i<a.length;i++) {
            if(a[i] == q) {
                  return true;
            }
      }
      return false;
}
```

Reihungen

- Da auch Arrays einen bestimmten Typ haben z.B. `gruss : char[]` kann man auch Reihungen von Reihungen bilden
- Mit einem Array von Arrays lassen sich z.B. Matrizen modellieren

```
int[] m0 = {1, 2, 3};
int[] m1 = {4, 5, 6};
int[][] m = {m0, m1};
```

- Daher heißen diese Gebilde auch *mehrdimensionale* Arrays

Überblick

Zeichenketten

- Wir hatten bereits diskutiert, dass Zeichenketten nicht nur zur Darstellung von daten benutzt werden könne; sie können selbst Gegenstand der Datenverarbeitung sein
- *Zeichenketten* (*Strings*) sind letztlich Arrays über dem Typ **`char`** (Folgen über der Menge der druckbaren Zeichen)
- Java stellt einen eigenen Typ `String` für Zeichenketten zur Verfügung, d.h. es gibt eine eigene Sorte (mit Operationen) für Zeichenketten in Java, wir können mit diesem Typ ganz normal „arbeiten“
- Der Typ `String` ist kein primitiver Typ, sondern ein Referenztyp (siehe Arrays), genauer eine Klasse von Objekten, ein sog. *Objekttyp*

Zeichenketten

- Betrachten wir nochmal das obige Beispiel

```
public class HelloWorld {
     public static final String GRUSS = "Hello World";

     public static void main(String[] args) {
          System.out.println(GRUSS);
     }
}
```

- In der Deklaration und Initialisierung

  `public static final String GRUSS = "Hello, World!";`

  entspricht der Ausdruck `"Hello, World!"` einer speziellen Schreibweise für ein konstantes Array `char[13]`, das in einen Typ `String` *gekapselt* ist
- Achtung: Die Komponenten dieses Arrays können nicht mehr (durch Neuzuweisung) geändert werden

Zeichenketten

- Obwohl `String` kein primitiver Typ ist, können Werte dieses Typs (wie bei primitiven Typen) durch Literale gebildet werden (in `" "` eingeschlossen)
- Beispiele für Literate der Sorte `String` in Java
  - `"Hello World!"`
  - `"Kroeger"`
  - `"Guten Morgen"`
  - `"42"`
- Literale und komplexere Ausdrücke vom Typ `String` können durch den (abermals überladenen!) Operator `+` konkateniert werden
  - `"Guten Morgen,"+""+"Kroeger"` ergibt die Zeichenkette `"Guten Morgen, Kroeger"`

Zeichenketten

- `String` ist in der Tat ein klassisches Modul, dass verschiedene Operationen (statische Methoden) über dieser (und weiterer) Sorte(n) bereitstellt
- Ein paar davon, schauen wir uns im folgenden an, ansonsten sei wieder auf die Dokumentation der API verwiesen
  - Typcast, um aus primitiven Typen Strings zu erzeugen

    ```
    static String valueOf(<type> input)
    ```

    wobei u.a. `<type>` $\in \{$ **`boolean`**, **`char`**, **`char[]`**, **`int`**, **`long`**, **`float`**, **`double`** $\}$
    Bei der Konkatenation eines Strings mit einem Literal eines primitiven Typs (z.B. `"Note: "+1.0`) werden diese Methoden (hier: **`static`** `String valueOf(`**`double`**` d)`) implizit verwendet

Zeichenketten

- `String` ist in der Tat ein klassisches Modul, dass verschiedene Operationen (statische Methoden) über dieser (und weiterer) Sorte(n) bereitstellt
- Ein paar davon, schauen wir uns im folgenden an (einige später), ansonsten sei wieder auf die Dokumentation der API verwiesen:
- Typcast, um aus primitiven Typen Strings zu erzeugen

  ```
  static String valueOf(<type> input)
  ```

  wobei u.a. `<type>` $\in \{$ **`boolean, char, char[], int, long, float, double`** $\}$

  Bei der Konkatenation eines Strings mit einem Literal eines primitiven Typs (z.B. `"Note: "+1.0`) werden diese Methoden (hier: **`static`** `String valueOf(`**`double`**` d)`) implizit verwendet

Zeichenketten

- Länge der Zeichenkette durch die Methode `int length()`
- Die Methode `char charAt(int index)` liefert das Zeichen an der gegebenen Stelle des Strings
  Dabei hat das erste Element den Index 0 und das letzte Element den Index `length() - 1`
- Beispiele:
  - der Ausdruck `"Hello, World!".length()` hat den Wert: `13`
  - der Ausdruck `"Hello, World!".charAt(10)` hat den Wert: `'l'`

Zeichenketten

- Nun verstehen Sie auch endlich offiziell den Parameter der `main`-Methode eines Java Programms
- In Programmbeispielen haben wir bereits die `main`-Methode gesehen, die das selbständige Ausführen eines Programmes ermöglicht
- Der Aufruf `java KlassenName` führt die `main`-Methode der Klasse `KlassenName` aus (bzw. gibt eine Fehlermeldung falls, diese Methode dort nicht existiert)
- Die `main`-Methode hat immer einen Parameter, ein `String`-Array, meist als Eingabe-Variablen `args`
- Dies ermöglicht das Verarbeiten von Argumenten, die über die Kommandozeile übergeben werden

Zeichenketten

- Der Aufruf

  `java KlassenName <Eingabe1> <Eingabe2> ... <Eingabe_n>`

  füllt das `String`-Array (Annahme, der Eingabeparameter heißt `args`) automatisch mit den Eingabe

```
args[0] = <Eingabe1>
args[1] = <Eingabe2>
...
args[n-1] = <Eingabe_n>
```

Zeichenketten

- Beispiel für einen Zugriff der main-Methode auf das Parameterarray

```
public class Gruss {

      public static void gruessen(String gruss) {
            System.out.println(gruss);
      }

      public static void main(String[] args) {
           gruessen(args[0]);
      }
}
```

Dadurch ist eine vielfältigere Verwendung möglich:

- `java Gruss "Hello, World!"`
- `java Gruss "Hallo, Welt!"`
- `java Gruss "Servus!"`

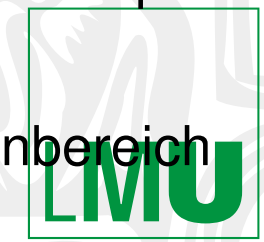
Überblick

Module

- Klassen (so wie wir sie bisher kennen: mit statischen Methoden und Klassenvariablen) sind ausführbar, wenn sie eine `main`-Methode enthalten
- Klassen können aber auch keine `main`-Methode enthalten, dann stellen sie verschiedene Algorithmen (in Form anderer statischer Methoden) und globale Größen wie Konstanten zur Verfügung
- Letztere Variante hatten wir Modul genannt
- Wir hatten ebenfalls diskutiert, dass Module ein wichtiges Strukturierungskonzept darstellen
- Funktionalitäten (insbesondere Methoden), die semantisch zusammen gehören (also z.B. in einem engen Kontext der realen Welt stehen), kann man gut in einem Modul bündeln

Packages

- Bei großen Programmen entstehen allerdings viele Klassen/Module (und sog. Schnittstellen, die wir später kennenlernen)
- Um einen Überblick über diese Menge zu bewahren, wird ein zusätzliches Strukturierungskonzept benötigt, das von den Details abstrahiert und die übergeordnete Struktur der Module/Klassen verdeutlicht
- Ein solches weiteres, übergeordnetes Strukturierungskonzept stellen die *Pakete* (*packages*) dar
- Packages erlauben es, Komponenten zu größeren Einheiten zusammenzufassen
- Die meisten Programmiersprachen bieten diese Strukturierungskonzepte (Module bzw. Packages, teilweise unter anderen Namen) an
- Packages gruppieren Klassen, die einen gemeinsamen Aufgabenbereich haben.



Packages in Java

- In Java können Klassen zu Packages zusammengefasst werden
- Packages dienen in Java dazu,
  - große Gruppen von Klassen, die zu einem gemeinsamen Aufgabenbereich gehören, zu bündeln,
  - potentielle Namenskonflikte zu vermeiden,
  - Zugriffe und Sichtbarkeit zu definieren und kontrollieren,
  - eine Hierarchie von verfügbaren Komponenten aufzustellen.

Packages in Java

- Jede Klasse in Java ist Bestandteil von genau einem Package
- Ist eine Klasse nicht explizit einem Package zugeordnet, dann gehört es implizit zu einem *Default*-Package
- Packages sind hierarchisch gegliedert, können also Unterpackages enthalten, die selbst wieder Unterpackages enthalten, usw.
- Die Package-Hierarchie wird durch Punktnotation ausgedrückt:

```
package.unterpackage1.unterpackage2. ... .Klasse
```

Import von Packages

- Der vollständige Name einer Klasse besteht wie bereits angedeutet aus dem Klassen-Namen *und* dem Package-Namen:

  `packagename.KlassenName`
- Package-Namen bestehen nach Konvention immer aus Kleinbuchstaben
- Um eine Klasse verwenden zu können, muss angegeben werden, in welchem Package sie sich befindet

Import von Packages

- Dies kann auf zwei Arten geschehen:
  1. Die Klasse wird an der entsprechenden Stelle im Programmtext über den vollen Namen angesprochen:
     ```
     packagename.KlassenName.methode();
     ```
  2. Am Anfang des Programms werden die gewünschten Klassen mit Hilfe einer **`import`**-Anweisung eingebunden:
     ```
     import packagename.KlassenName;
     ...
     methode();
     ```
- Achtung: werden zwei Klassen gleichen Namens aus unterschiedlichen Packages importiert, müssen die Klassen trotz **`import`**-Anweisung mit vollem Namen aufgerufen werden!

Import von Packages

- Klassen des Default-Packages können ohne explizite **`import`**-Anweisung bzw. ohne vollen Namen verwendet werden
- Wird in der **`import`**-Anweisung eine Klasse angegeben, wird genau diese Klasse importiert. Alle anderen Klassen des entsprechenden Packages bleiben unsichtbar
- Will man alle Klassen eines Packages auf einmal importieren, kann man dies mit der folgenden **`import`**-Anweisung:

  ```
  import packagename.*;
  ```

- Achtung: es werden dabei wirklich nur die Klassen aus dem Package `packagename` eingebunden und etwa auch die Klassen aus Unter-Packages von `packagename`

Packages aus der Java-Klassenbibliothek

- Die Java-Klassenbibliothek bietet bereits eine Vielzahl von Klassen an, die alle in Packages gegliedert sind.
- Beispiele für vordefinierte Packages:

`java.io`	Ein- und Ausgabe
`java.util`	nützliche Sprach-Werkzeuge
`java.awt`	Abstract Window Toolkit
`java.lang`	Elementare Sprachunterstützung

  usw.
- Die Klassen im Package `java.lang` sind so elementar (z.B. enthält `java.lang` die Klasse `System`), dass sie von jeder Klasse automatisch importiert werden. Ein expliziter Import mit **`import`** `java.lang.*;` ist also nicht erforderlich.

Deklaration eigener Packages

- Ein eigenes Package `mypackage` wird angelegt, indem man vor eine Klassendeklaration und vor den **`import`**-Anweisungen die Anweisung **`package`** `mypackage;` platziert.
- Es können beliebig viele Klassen (jeweils aber mit unterschiedlichen Namen) mit der Anweisung **`package`** `mypackage;` im selben Package gruppiert werden.
- Um Namenskollisionen bei der Verwendung von Klassenbibliotheken unterschiedlicher Hersteller zu vermeiden, ist es Konvention, für Packages die URL-Domain der Hersteller in umgekehrter Reihenfolge zu verwenden, z.B.

  `com.sun. ...` für die Firma Sun,

  `de.lmu.ifi.dbs. ...` für den DBS-Lehrstuhl an der LMU.

Deklaration eigener Packages

- Wie bereits erwähnt, muss die Deklaration einer Klasse `X` in eine Datei `X.java` geschrieben werden
- Darüberhinaus müssen alle Klassendeklarationen (also die entsprechenden `.java`-Dateien) eines Packages `p` in einem Verzeichnis `p` liegen
- Beispiel:
  - Die Datei `Klasse1.java` mit der Deklaration der Klasse `package1.Klasse1` liegt im Verzeichnis `package1`
  - Die Datei `Klasse2.java` mit der Deklaration der Klasse `package1.unterpackage1.Klasse2` liegt im Verzeichnis `package1/unterpackage1`

Zugriffsrechte und Sichtbarkeit

- Wir hatten bereits ein Schlüsselwort zur Spezifikation der Sichtbarkeit von Klassen und Klassen-Elementen (globale Größen/statische Methoden) kennengelernt: **`public`**
- Klassen und Elemente mit der Sichtbarkeit **`public`** sind von allen anderen Klassen (insbesondere auch Klassen anderer Packages) sichtbar und zugreifbar
- Darüberhinaus gibt es weitere Möglichkeiten, eine davon ist **`private`**
- Klassen und Elemente mit der Sichtbarkeit **`private`** sind nur innerhalb der eigenen Klasse (also auch *nicht* innerhalb möglicher Unterklassen oder Klassen des selben Packages) sichtbar und zugreifbar

Zugriffsrechte und Sichtbarkeit

- Klassen und Elemente, deren Sichtbarkeit *nicht* durch ein entsprechendes Schlüsselwort spezifiziert ist, erhalten per Default die sogenannte *package scoped* (*friendly*) Sichtbarkeit: diese Elemente sind nur für Klassen innerhalb des selben Packages sichtbar und zugreifbar
- Es gibt dann noch eine vierte Möglichkeit (`protected`), die wir mal wieder erst später kennen lernen