1. Referenzen

2. Arrays (Felder, Reihungen)

2.1 Der Datentyp der Arrays

2.2 Arrays in Java

2.3 Zeichenketten (Strings)

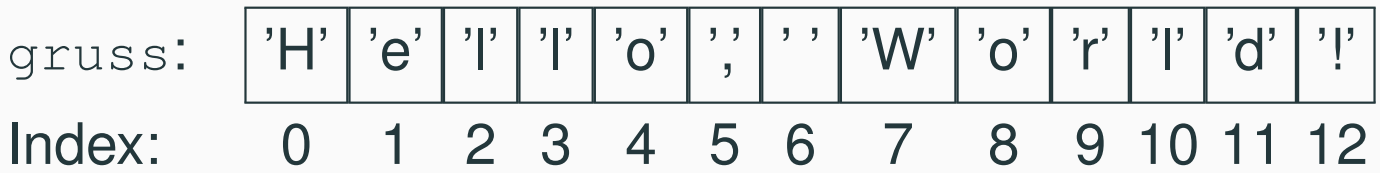
3. Record-Typen

Achtung

- Die Umsetzung in Java erfolgt 1-zu-1, allerdings ist zu beachten:
- Wie bei den meisten Programmiersprachen beginnt die Indexmenge in Java bei 0, d.h. für eine n-elementige Reihung gilt die Indexmenge $I_n = \{0, \ldots, n-1\}$.
  (Die Operation *LEN* würde für ein Array der Länge n also $n-1$ zurückgeben!)
- Wie bei uns sind in Java die Arrays *semidynamisch*, d.h. ihre Größe kann zur Laufzeit (=dynamisch) festgesetzt werden, danach aber nicht mehr geändert werden (=statisch), d.h. dynamisches Wachstum von Arrays ist auch in Java nicht möglich!!!

Beispiel

Ein **`char`**-Array `gruss` der Länge 13:



Arrays in Java

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

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

  (Konstanten wie immer mit dem Zusatz **`final`**)
- Und natürlich: Arrays sind *Referenztypen*, d.h. auf dem Zettel einer Variable steht nicht das Array direkt, sondern die Referenz (auch Java-Arrays sind i.d.R. unterschiedlich groß und werden daher auf der Halde (Heap) verwaltet.

Arrays in Java

Erzeugung (Konstruktions-Operator):

- Es gibt nicht *den* Konstruktions-Operator *INIT*.
- Aber: wie alle Referenztypen werden Arrays bei ihrer Initialisierung *erzeugt*.
- Die Initialisierung (Erzeugung) eines Arrays kann dabei auf verschiedene Arten erfolgen.
- Die einfachste ist, alle Elemente der Reihe nach in geschweiften Klammern `{}` aufzuzählen:

  ```
  <type>[] variablenName = {elem1, elem2, ...}
  ```

  wobei die einzelnen `elem1`, `elem2`, etc. Literale (Werte) oder Variablen vom Typ `<type>` sind.

Zugriff auf das *i*-te Element (Projektion):

- Die Operation *PROJ* für den Zugriff auf das `i`-te Element eines Arrays `a` notiert man in Java durch den Ausdruck `a[i]`.
- Dabei ist `i` vom Typ **`int`** (bzw. vom Compiler implizit in **`int`** cast-bar)
- Der Wert des Ausdrucks `variablenName[i]` ist der Wert des Arrays `variablenName` an der Stelle `i`.
- Der Typ des Ausdrucks `variablenName[i]` ist der Typ, über dem das Array `variablenName` gebildet wurde.
- Beispiel:

  ```
  int[] a = { 1 , 2 , 3 , 4 };
  ```

  Der Ausdruck `a[1]` hat den Wert `2` und den Typ **`int`**.

Verändern des *i*-ten Elements:

- Die Operation *ALT* gibt es nicht explizit.
- Vielmehr ist die Projektion auf das `i`-te Element eines Arrays `a` (`a[i]`) auch für den schreibenden Zugriff gedacht.
- Man kann `a[i]` also nicht nur als Ausdruck verstehen, sondern diesem selbst auch einen Wert zuweisen (es handelt sich nämlich letztlich um eine Variable).
- Beispiel: `int[] a = { 1 , 2 , 3 , 4 };`
  `a[1] = 6;` verändert das Array `a` zu `{ 1 , 6 , 3 , 4 }`.

Zugriff auf die Länge:

- Die Operation *LEN* ist in Java etwas anders umgesetzt: es gibt eine Konstante mit Namen `length`, die anzeigt, wie viele Elemente im Array enthalten sind (Vorsicht: das sind nicht $n-1$ sondern n!!!), d.h. es wird nicht die obere Grenze der Indexmenge sondern die Länge des Arrays gespeichert.
- Der Typ dieser Konstante ist **`int`**.
- Der Name der Konstanten ist zusammengesetzt aus dem Namen des Arrays (also z.B. `a`) und dem Namen `length` (also `a.length`).
- Beispiel:

  ```
  int[] a = { 1 , 2 , 3 , 4 };
  ```

  Der Ausdruck `a.length` hat den Wert `4` und den Typ **`int`**.

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
```

Arrays in Java

- Oft vereinbart man eine Array-Variable, 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 — es ist also so etwas wie ein Konstruktions-Operator).
- Dann kann man das Array im weiteren Programmverlauf (durch Verändern der einzelnen Elemente) füllen:

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

Arrays in Java

- 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 ggfs. noch nicht beim
// Uebersetzen des Programmcodes (Kompilieren)
// (z.B. weil x ein Eingabeparameter ist)
char[] abc = new char[x];
```

Arrays in Java

- Was passiert, wenn man ein Array anlegt

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

  aber nicht füllt? Ist das Array dann leer?
- Nein: 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
```

- Das legt den Schluss nahe, dass die einzelnen Elemente des Arrays wiederum Variablen (Zettel) sind, die ich beschreiben und ablesen kann.
- Das ist genau so, wie wir es theoretisch eingeführt haben und 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.

Arrays in Java

- Genauer gesagt enthält `zahlen` die Adresse der Zettel, der einzelnen Elemente `zahlen[i]` und einen zusätzlichen (nicht-radierbaren) Zettel mit der Länge des Arrays (`zahlen.length`).
- Die einzelnen (radierbaren) Zettel `zahlen[1]`, enthalten die konkreten Werte
- `zahlen` ist im Stack gespeichert, die einzelnen `zahlen[i]` stehen im Heap.

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

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

- Und jetzt eben Achtung: statt einem konkreten Wert wird hier eine Referenz zugewiesen (siehe Implikation “Zuweisung”)!

Implikationen!!!

- Call-by-reference-Effekt:

```
public static void veraendere(int[] a, int i, int wert) {
  a[i] = wert;
}

public static void main(String[] args) {
  int[] werte = {0, 1, 2};
  veraendere(werte, 1, 3);
}
```

Was passiert mit `werte`?

Implikationen!!!

- Gleichheit/Identität:

```
int[] x = {1 , 2};
int[] y = {1 , 2};
boolean gleich = (x==y);  // Welchen Wert hat gleich???
```

Implikationen!!!

- Zuweisung/Kopie:

```
int[] x = {1 , 2};
int[] y = {2 , 3};
x = y;
y[1] = 5;  // Welchen Wert hat x[1]???
x[1] = 10; // Welchen Wert hat y[1]???
```

Workarounds

- Die Klasse `java.util.Arrays` bietet einige (statische) Methoden rund um Arrays an.
- Die (überladen) Methode `equals`, überprüft die Gleichheit zweier Arrays, z.B.

```
static boolean equals(int[] a, int[] a2)
```

für **`int`**-Arrays, also:

```
int[] x = {1 , 2};
int[] y = {1 , 2};
boolean gleich = java.util.Arrays.equals(x,y);
```

- Die überladenen Methoden `copyOf`, erstellt eine tiefe Kopie zu erstellen, z.B.

  ```
  static int[] copyOf(int[] original, int newLength)
  ```

  für **`int`**-Arrays (die neue Kopie kann dabei abgeschnitten oder mit zusätzlichen 0-Werten aufgefüllt werden — je nach `newLength`), also

  ```
  int[] x = {1 , 2};
  int[] y = {2 , 3};
  x = java.util.Arrays.copyOf(y, y.length);
  // x und y sind nun unabhaengig
  int[] z = java.util.Arrays.copyOf(y, y.length+2);
  // z ist um 2 Elemente laenger, die zusaetzlichen Elemente
  // sind jeweils 0
  ```

- 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 (Zettel), 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
```

Arrays in Java

- 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 die Referenz zu weiteren Zetteln (die Elemente) steht.
- Konstanten sind nicht radierbare Zettel, d.h. der Zettel `<type>[]` auf dem die Referenz zu den anderen Zettel steht, ist dann nicht radierbar; die Zettel, die referenziert werden, aber natürlich schon.

- Mit Arrays kann man also wunderbar programmieren:
- Beispiel: der Algorithmus *enthalten* von oben

```
public static boolean enthalten(int[] x, int a) {
      boolean gefunden = false;
      int i = 0;
      while(!gefunden && i < x.length) {
            if(x[i] == a) {
                  gefunden = true;
            }
            i++;
      }
      return gefunden;
}
```

Versuchen Sie es selbst mal mit einer **`for`**-Schleife!

- Beispiel: Summe der Elemente in einem **`int`**-Array

```
public static int summeElemente(int[] x) {
      int erg = 0;
      for(int i = 0; i < x.length; i++) {
          erg = erg + x[i];
      }
      return erg;
}
```

- Beispiel: Wechselgeldalgorithmus

  Zur Erinnerung:
  - Eingabe eines Rechnungsbetrags $1 \leq r \leq 100$.
  - Gesucht ist das Wechselgeld zu einer Bezahlung von r mit einem 100-EUR-Schein als Menge an 1 EUR, 2 EUR Münzen sowie 5 EUR Scheinen (mit dem Ziel möglichst wenige Münzen/Scheine auszugeben).
  - Als Ergbnis wollten wir eine Folge an 1er, 2er und 5er ausgeben. Das könnten wir jetzt mit einem Array implementieren.

  Dadurch handeln wir uns allerdings ein Problem ein:
  - Arrays sind ja semi-dynamisch, d.h. wir müssen in Abhängigkeit von r zunächst bestimmen, wieviel Scheine/Münzen auszugeben sind (d.h. wie lang das Ergebnis-Array wird).

- Beispiel: Wechselgeldalgorithmus (cont.)

  Aber die Lösung hatten wir wenigstens schon:

  - Der ganzzahlige Quotient $q_1 = DIV(100 - r, 5)$ ist die Anzahl der 5-EUR-Scheine im Wechselgeld.
  - Der Rest $r_1 = MOD(100 - r, 5)$ ist der noch zu verarbeitende Wechselbetrag. Offensichtlich gilt $r_1 < 5$.
  - r_1 muss nun auf 1 und 2 aufgeteilt werden, d.h. analog bilden wir $q_2 = DIV(r_1, 2)$ und $r_2 = MOD(r_1, 2)$.
  - q_2 bestimmt die Anzahl der 2-EUR-Münzen und r_2 die Anzahl der 1-EUR-Münzen.

  D.h. die Länge des Ergebnis-Arrays ist $q_1 + q_2 + r_2$.

- Beispiel: Wechselgeldalgorithmus (cont.)

  Fragt sich nur noch, wie wir das Ergebnis-Array zu befüllen haben:

  - q_1 ist die Anzahl der 5-EUR-Scheine, d.h. die Stellen $0, \ldots, (q_1 - 1)$ sind mit der Zahl 5 zu belegen.
  - q_2 ist die Anzahl der 2-EUR-Münzen, d.h. die Stellen $q_1, \ldots, (q_1 + q_2) - 1$ sind mit der Zahl 2 zu belegen.
  - r_2 ist die Anzahl der 1-EUR-Münzen, d.h. die Stellen $(q_1 + q_2), \ldots, (q_1 + q_2 + r_2) - 1$ sind mit der Zahl 1 zu belegen.

- Beispiel: Wechselgeldalgorithmus (cont.)

```
public static int[] wechselGeld(int r) {
    int q1 = (100 - r) / 5;
    int q2 = ((100 - r) % 5) / 2;
    int r2 = ((100 - r) % 5) % 2;
    int[] erg = new int[q1 + q2 + r2];

    for(int i=0; i<q1; i++) {
        erg[i] = 5;
    }
    for(int i=q1; i<q1+q2; i++) {
        erg[i] = 2;
    }
    for(int i=q1+q2; i<erg.length; i++) {
        erg[i] = 1;
    }
    return erg;
}
```

Indexmenge von Arrays

- Wir haben übrigens immer noch nicht geklärt, was eigentlich passiert, wenn wir eine Indexposition abrufen wollen, die es gar nicht gibt, also z.B.

```
int[] a = {1, 2, 3}; a[10] = 11;
```

- In diesem Beispiel hat der Compiler die Chance, das theoretisch abzufangen (was er aber nicht tun wird).
- Ganz allgemein ist das sowieso schwierig:

```
int x; // Wert erst zur Laufzeit klar ...
int[] a = new int[x]; //Laenge erst zur Laufzeit klar!
```

- Der Fehler wird erst zur Laufzeit behandelt durch eine sog. *Ausnahme* (*Exception*).
- Was das ist, lernen wir leider erst später kennen, aber schon mal soviel: das Programm wird mit einer Fehlermeldung beendet.

Arrays in Java

- Da auch Arrays einen bestimmten Typ haben z.B. `gruss : char[]` kann man auch Reihungen von Reihungen bilden.
- Diese Gebilde heißen auch *mehrdimensionale* Arrays.
- 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};
```

- Man ist dabei nicht auf „rechteckige“ Arrays beschränkt:

```
int[] m0 = {0};
int[] m1 = {1, 2};
int[] m2 = {3, 4, 5};
int[][] m = {m0, m1, m2};
```

1. Referenzen

2. Arrays (Felder, Reihungen)

2.1 Der Datentyp der Arrays

2.2 Arrays in Java

2.3 Zeichenketten (Strings)

3. Record-Typen

Zeichenketten

- Wir hatten bereits diskutiert, dass Zeichenketten nicht nur zur Darstellung von Daten benutzt werden können; sie können selbst Gegenstand der Datenverarbeitung sein.
- *Zeichenketten* (*Strings*) sind in Java 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 wieder ein Referenztyp, genauer eine Klasse von Objekten, ein sog. *Objekttyp*.

Zeichenketten

- Betrachten wir folgendes 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.

Zeichenketten

- Achtung: Die Komponenten dieses Arrays können nicht mehr (durch Neuzuweisung) geändert werden.
- D.h. in diesem Beispiel

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

  ist nicht nur die Referenz eine Konstante, sondern auch die einzelnen Komponenten des Arrays.
- Die Objekte (Literale) der Klasse `String` sind sog. *immutable* Objekte. Was das genau heißt sehen wir leider erst wieder später.

Zeichenketten

- Obwohl `String` (wie Arrays) kein primitiver Typ ist, wird dieser Typ in Java sehr ähnlich wie ein primitiver Typ behandelt:
- Z.B. können Werte dieses Typs (wie bei primitiven Typen) durch Literale gebildet werden (in `" "` eingeschlossen).
- Beispiele für Literale der Sorte `String` in Java:
  - `"Hello World!"`
  - `"Kroeger"`
  - `"Guten Morgen"`
  - `"42"`
- Literale und komplexere Ausdrücke vom Typ `String` können durch den (überladenen) Operator `+` konkateniert werden:
  - `"Guten Morgen, "+"Kroeger"` ergibt die Zeichenkette `"Guten Morgen, Kroeger"`

Zeichenketten

- Bei der Konkatenation ist zu beachten, dass ein neues Objekt vom Typ `String` erzeugt wird, das unabhängig von den bisherigen Strings ist:

```
String gm = "Guten Morgen, ";
String k = "Kroeger";
String gmk = gm + k;
}
```

- In diesem Beispiel wird durch `gm + k;` ein neuer String erzeugt (mit einer neuen Speicheradresse) und die deren Referenz `gmk` zugeordnet, d.h. alle drei Variablen haben unterschiedliche Werte (Referenzen).

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:
- Eine überladene Typcast-Operation, um aus primitiven Typen Strings zu erzeugen

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

  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()` (Achtung, anders als bei Arrays ist das tatsächlich eine Methode mit “Funktionsklammern”!!!)
- 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` (ist ja eigentlich ein Array).
- 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.
- 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-Variable `args` (könnte auch anders benannt werden, wichtig ist nur der Typ).
- Beispiele:

```
public static void main(String[] args)   // OK
public static void main(String[] name)   // OK
public static void main(String args)   // nicht OK, Typ!!!
```

Zeichenketten

- Dies ermöglicht das Verarbeiten von Argumenten, die über die Kommandozeile übergeben werden.
- Der Aufruf

  `java KlassenName <Ein1> <Ein2> ... <Ein_n>`

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

  ```
  args[0] = <Ein1>
  args[1] = <Ein2>
  ...
  args[n-1] = <Ein_n>
  ```

  d.h. zur Laufzeit wird entschieden, wie lang das Array `args` ist.

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

```
public class Gruss {
      public static void gruessen(String gruss) {
            System.out.println("Der Gruss ist: "+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!"`

Zeichenketten

- Natürlich können wir die Input-Strings auch in andere (insbs. primitive) Typen casten, dies geht allerdings etwas aufwendiger (über die Wrapper-Klassen der primitiven Typen).
- Hä????
- Tut mir leid, auch das lernen wir noch kennen, aber mal wieder später.
- Aber immerhin jetzt schon: damit können Sie dann auch z.B. numerische Daten über die Kommandozeile eingeben und entsprechend verarbeiten.

Endlich: der Kalender

- Mit Strings und Arrays können wir nun auch endlich unser Kalenderprojekt etwas cleverer implementieren.
- Statt je eine Variable für die einzelnen Termin-Einträge zu verschwenden, verwalten wir diese jetzt in einem Array.
- Statt einem einzigen **`char`**-Eintrag als Abkürzung können wir jetzt den Eintrag als `String`-Objekt speichern.
- Die Termine an einem Tag (einfach erweiterbar auf eine ganze Woche bzw. auf ganze Jahre):

```
...
String[] termine = new String[24]
...
```

Endlich: der Kalender

- Die Termine sollten zwar noch initialisiert werden (z.B. mit dem leeren String), dies geht aber auch einfach mit einer Schleife:

```
for(int i = 0; i < termine.length; i++) {
  termine[i] = "";
}
```

- Die Ausgabe benötigt auch nur solch eine Schleife.
- Einfach geht auch die Eintragung eines neuen Termins:

```
// Termin-Eingabe
int stunde = ...
Stringe eingabe = ...

termine[stunde] = eingabe;
```

Endlich: der Kalender

- Die Erweiterung auf eine ganze Woche kann dann z.B. über ein mehrdimensionales Array erfolgen:

```
String[][] termine = new String[7][];

// die einzelnen Tage mit 24 Stunden anlegen
// und gleich initialisieren:
for(int i = 0; i < termine.length; i++) {
  // anlegen
  termine[i] = new String[24];
  // initialisieren
  for(int j=0; j < termine[i].length; j++) {
    termine[i][j] = "";
  }
}
```

- Frühstück am Dienstag (Tag 2), 8 Uhr tragen wir wie folgt ein:

```
termine [1][7] = "Fruehstueck";
```

Kapitel 3: Record-Typen i

1. Referenzen

2. Arrays (Felder, Reihungen)

3. Record-Typen

3.1 Der Datentyp Record

3.2 Records in Java

1. Referenzen

2. Arrays (Felder, Reihungen)

3. Record-Typen

3.1 Der Datentyp Record

3.2 Records in Java

Wozu Records?

- So, nachdem wir denken, dass wir das mit dem Kalender im Griff haben: warum nicht auch noch Personen (Adressen/ Telefonnummern) verwalten.
- Für Personen wollen wir insbesondere folgende Infos verwalten:
  - Name (vermutlich vom Typ `String`)
  - E-Mailadresse (vermutlich vom Typ `String`)
  - Telefonnummer (vermutlich vom Typ **`int`**)
  - optionaler Kommentar (vermutlich vom Typ `String`)
- Das wird mit einem mehrdimensionalen Array schwierig (Mix aus `String` und **`int`**).

Wozu Records?

- Macht aber nix, dann nehmen wir halt mehrere Arrays (ist ja quasi dasselbe):

```
// wir verwalten mal maximal 100 Freunde:
String[] name = new String[100];
String[] mail = new String[100];
int[] tel = new int[100];
String[] comment = new String[100];
```

- Der Eintrag einer neuen Person an Stelle `i`:

```
name[i] = "Pepper Wutz";
mail[i] = "pepper@wutz.de";
tel[i] = 015112;
comment[i] = "Unbedingt daten!!!";
```

Wozu Records?

- Nehmen wir an, wir wollen Index 3 und Index 4 vertauschen (z.B. weil wir die Personen sortieren wollen):

```
String s;   int i   // Zwischenspeicher

// Name
s = name[3];   name[3] = name[4];   name[4] = s;
// Mail
s = mail[3];   mail[3] = mail[4];   mail[4] = s;
// Telefonnr
i = tel[3];   tel[3] = tel[4];   tel[4] = i;
// Kommentar
s = comment[3];   comment[3] = comment[4];   comment[4] = s;
```

Wozu Records?

- Was ist blöd bei dieser Datenmodellierung?
- Wir verwalten einen Datensatz (Person) in mehreren Arrays, dadurch geht die Übersicht schnell verloren.
- Bereits vermeintlich einfache Operationen sind relativ komplex.
- Wir bräuchten ein Konstrukt, mit dem wir verschiedene Werte zu einem Datum zusammen fassen können.

Wozu Records?

- Dieses Konstrukt ist mathematisch gesehen ein Tupel (ein Element des kartesischen Produkts von verschiedenen Mengen).
- In den meisten Programmiersprachen heißen diese Konstrukte *Records*.
- Arrays und Records sind also grundsätzlich gleich (Elemente eines Kreuzprodukts).
- Der Unterschied ist, dass bei Arrays die Grundmenge nur S ist, bei Records können es unterschiedliche Sorten S_i sein (wobei diese nicht notwendigerweise verschieden sein müssen).

Was sind Records

- Ein *Record* (Tupel) über Datentypen (Sorten) $S_1, \ldots S_n$ ist ein Element aus der Menge $S_1 \times \ldots \times S_n$.
- Der Datentyp "Record über $S_1, \ldots S_n$" stellt also die Elemente dieses Kreuzprodukts zur Verfügung.
- Der Datentyp sollte entsprechend Grundoperationen bereitstellen, z.B. sollte, ähnlich wie bei einem Array, man auf die einzelnen *Komponenten* eines Records zugreifen können.
- Entsprechend erlauben wir auch Variablen vom Typ Record über $S_1, \ldots S_n$ und können damit Ausdrücke mit diesen Variablen und den Grundoperationen bilden.

Grundoperationen von Records

- Die Grundoperationen beschränken sich dabei typischerweise auf:
  - Direkter Zugriff (lesend/schreibend) auf die einzelnen Komponenten.
  - Erzeugung/Initialisierung eines Records.
- Diese könnten wir wie gewohnt in ein entsprechendes Modul schreiben und damit allgemein zur Verfügung stellen.
- Auch das Verhalten dieser Operationen ließe z.B. durch axiomatische Spezifikation definieren.
- Im Gegensatz zu Arrays verzichten wir allerdings auf die Theorie und schauen uns Records gleich in Java an.
- Vorher allerdings noch der Hinweis, dass es sich bei Records natürlich wieder um Referenztypen handelt.

1. Referenzen

2. Arrays (Felder, Reihungen)

3. Record-Typen

3.1 Der Datentyp Record

3.2 Records in Java

Records als Klassen

- Tatsächlich gibt es in Java den Datentyp Record über $S_1, \ldots S_n$ nicht explizit (wie Arrays), sondern es ist eigentlich ein "Abfallprodukt" der Klassen, die im oo Paradigma ganz entscheidend sind (dazu aber später mehr).
- Daher muss man jeden speziellen Record-Typ zunächst explizit vereinbaren und ihm auch einen Namen geben.
- Das tut man in Java durch eine sog. Klassen-Vereinbarung.
- Klassen ermöglichen uns also, einen eigenen Datentyp zu definieren, bei dem es sich um ein Record-Typ handelt.

Records als Klassen

- Klassen können grundsätzlich in eigenständigen Programmdateien vereinbart werden (das haben wir schon kennen gelernt), aber auch innerhalb eines Programms (dann heißen sie auch *innere* Klassen).
- Bei inneren Klassen handelt es sich typischerweise um klassische Record-Typen, die man lokal als Record-Datentyp vereinbart.
- Klassen können also dazu verwendet werden, verschiedene Werte zu einem einzigen Datentyp zusammen zu fassen.
- Dies geschieht über entsprechende Variablen, die sog. *Komponenten-* oder *Instanz-Variablen*.

Klassendefinition

- Eine Klassendefinition, die einen entsprechenden Record-Typen für unser Adressbuch vereinbart wäre z.B.:

```
public static class Person {
  public String name;
  public String mail;
  public int tel;
  public String comment;
}
```

- Wie gesagt, diese Klassendefinition kann entweder in einer eigenen Datei stehen (dann ohne **`static`**), oder innerhalb einer anderen Klasse, z.B. einer ausführbaren Klasse mit `main`-Methode.

Klassendefinition

- Mit dieser Definition machen wir den Typ `Person` bekannt.
- Der Typ besteht aus einer Menge von Instanzvariablen (teilweise unterschiedlichen Typs).
- Literale vom Typ `Person` werden nun *Objekte* genannt und unterschieden sich durch die Werte der Instanzvariablen.
- Diese Objekte haben aber alle dieselbe Struktur: für den Typ `Person` sind alle Literale (Objekte) Elemente aus

  `String` × `String` × **`int`** × `String`

Klassendefinition

- Nochmal: das ist eigentlich genau dasselbe wie bei den Arrays.
- Die einzelnen Komponenten des Arrays sind Variablen (desselben Typs, gespeichert auf der Halde).
- Die einzelnen Komponenten des Records sind ebenfalls Variablen (ggfls. unterschiedlichen Typs, ebenfalls auf der Halde abgelegt).
- Da beim Array der Typ gleich ist, müssen die Komponenten-Variablen nicht explizit angegeben werden, beim Record entsprechend schon.

Instanziierung

- Wollen wir nun Variablen vom Typ `Person` vereinbaren und instantiieren, geht es jetzt nur mit dem Operator **`new`** gefolgt vom Namen der Klasse (des Record-Typs) mit Funktionsklammern:

```
Person person1 = new Person();
// alternativ:
Person person2;
person2 = new Person();
```

- Achtung, die Variablen `person1` und `person2` sind natürlich Referenz-Variablen: die Referenzen "zeigen" auf den Speicherbereich (auf der Halde), wo die Komponenten (Instanz-Variablen mit Namen und Inhalten) abgelegt sind.

Zugriff auf Komponenten

- Der Zugriff erfolgt ähnlich wie bei Arrays über den Index, nur jetzt über den Namen der Instanzvariablen mit Punktnotation.
- Z.B. bezeichnet der Ausdruck `person1.tel` die Instanzvariable `tel` der Variable `person1`.
- Die ist zunächst mit einem Standard-Wert gesetzt, kann aber wie bei den Arrays direkt verändert werden:
  Die Anweisung `person1.tel = 01234;` weist ihr entsprechend eine neue Telefonnummer zu, d.h. die Komponente von `person1` hat nun einen neuen Wert (die entsprechende Telefonnummer).

Beispiel

- Entsprechend kann nun Personen verwaltet werden.
- Z.B. Eingabe einer neuen Person:

```
Person person = new Person();

// Eingabe der Werte
String name = ...
String mail = ...
int tel = ...
String comment = ...

// Setzen der Werte von person:
person.name = name;
person.mail = mail;
person.tel = tel;
person.comment = comment;
```

Implikationen: Call-by-reference-Effekt

- Aufpassen, dagegen gibt es keinen Workaround:

```
public static void einBisschenSpassMussSein(Person p) {
  p.name = "Roberto Blanco";
}
...
  Person person = new Person();
  person.name = "Peer Kroeger";
  person.mail = "";
  person.tel = 9306;
  person.comment = "ziehmlich verplant";
  einBisschenSpassMussSein(person);
...
```

Implikationen: Gleichheit/Identität

- Hier gibt es nun keinen “einfachen” Workaround, sondern man muss die (Komponenten-weise) Gleichheit selbst implementieren.
- Dabei müssen wir alle Komponenten der beiden Records auf Gleichheit (Vorsicht, nicht Identität) überprüfen.
- Leider können wir das noch nicht wirklich: es gibt zwar einen Weg, Strings auf Gleichheit (Vorsicht, sind ja Referenztypen!!!) zu testen (es sind ja “nur” Arrays), die Methode `equals` der Klasse `String` verstehen wir aber eigentlich noch nicht.

Implikationen: Gleichheit/Identität

- Die Lösung (die wir teilweise noch nicht verstehen) sieht “ungefähr” so aus:

```
public static boolean equals(Person p1, Person p2) {
  return (
    p1.name.equals(p2.name)   &&
    p1.mail.equals(p2.mail)   &&
    p1.tel == p2.tel   &&
    p1.comment.equals(p2.comment)
  );
}
```

d.h. hier wird Komponenten-weise abgeglichen.

Implikationen: Kopie

- Auch für die Erstellung einer tiefen Kopie müssen wir selbst aktiv werden, und auch hier wieder die Problematik, dass einige Komponenten selbst Referenztypen sind.

```
public static Person deepCopy(Person p) {
  Person erg = new Person();
  erg.name = p.name+"";
  erg.mail = p.mail+"";
  erg.tel = p.tel;
  erg.comment = p.comment+"";
  return erg;
}
```

Warum +""???

Mit Records programmieren

- Ein Record-Typ eignet sich i.Ü auch sehr gut für unser Wechselgeld Problem.
- Wir könnten das Wechselgeld nicht als Folge der auszuzahlenden Scheine/Münzen modellieren, sondern als Recordtyp mit den drei Komponenten:
  - “Einer” gibt die Anzahl an 1 EUR Münzen an
  - “Zweier” gibt die Anzahl an 2 EUR Münzen an
  - “Fuenfer” gibt die Anzahl an 5 EUR Scheinen an
- Der entsprechende Typ “Wechselgeld” wäre also ein Record-Typ.

Implikationen: Kopie

```
public class Wechselgeld {
  public static class WG {
    public int einer;
    public int zweier;
    public int fuenfer;
  }

  public static WG wechelgeld(int r) {
    WG w = new WG();
    w.fuenfer = (100 - r) / 5;
    w.zweier = ((100 - r) % 5) / 2;
    w.einer = ((100 - r) % 5) % 2;
    return w;
  }

...
```

Implikationen: Kopie

```
...

  public static void main(String[] args) {
    // Eingabe Rechnungsbetrag
    int r = ...
    WG wg = wechselgeld(r);

    System.out.println("Das Wechselgeld enthaelt:");
    System.out.println(wg.fuenfer+" 5 EUR Scheine");
    System.out.println(wg.zweier+" 2 EUR Muenzen");
    System.out.println(wg.einer+" 1 EUR Muenzen");
  }
}
```