Einführung in die Programmierung I

9. Prozedurale Konzepte in Java, Teil 2

Stefan Zimmer

7.1.2008

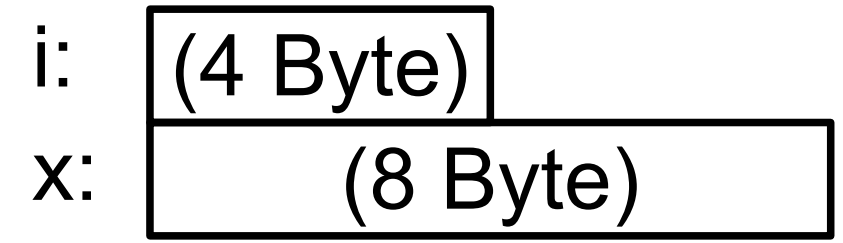
Referenz-Datentypen

- Nächstes Thema unserer Java-Tour wären eigentlich Felder (viele gleiche Dinge zusammenpacken)
- Aber Felder zählen in Java zu den *Referenztypen* (im Gegensatz zu den *primitiven Datentypen*)
- Deswegen machen wir uns erstmal allgemeine Gedanken über Referenzen und Referenztypen (die sind auch für die objektorientierten Aspekte in Java sehr wichtig)
- Und diese Gedanken beginnen wir mit einer genaueren Betrachtung des Gegenstücks, also Variablen für primitive Datentypen

Primitive Datentypen im Speicher (1)

- Für Variablen primitiver Datentypen wird beim Betreten eines Blocks soviel Speicher reserviert, wie das Element belegt – wir können uns diese Speicherstelle (vereinfachend) mit dem Bezeichner beschriftet vorstellen.

```
{ int i; double x;
  // Anweisungen
}
```

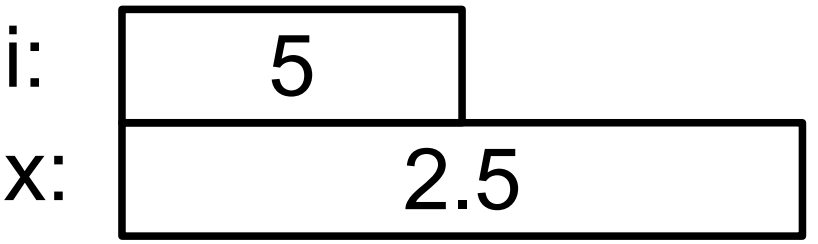


- (In Wirklichkeit gibt es keine Beschriftungen: es reicht, während des Übersetzens für jede Variable die relative Position in Bezug zum Beginn des Speicherplatzes dieses Blocks zu berechnen und bei Verwendung dieser Variablen einzusetzen)

Primitive Datentypen im Speicher (2)

- Wenn wir was in den Variablen speichern, landen die Werte (in Binärdarstellung) im eben reservierten Speicher (wo auch sonst, fragt man sich – noch!):

```
{ int i; double x;
  i = 5;
  x = 2.5;
}
```

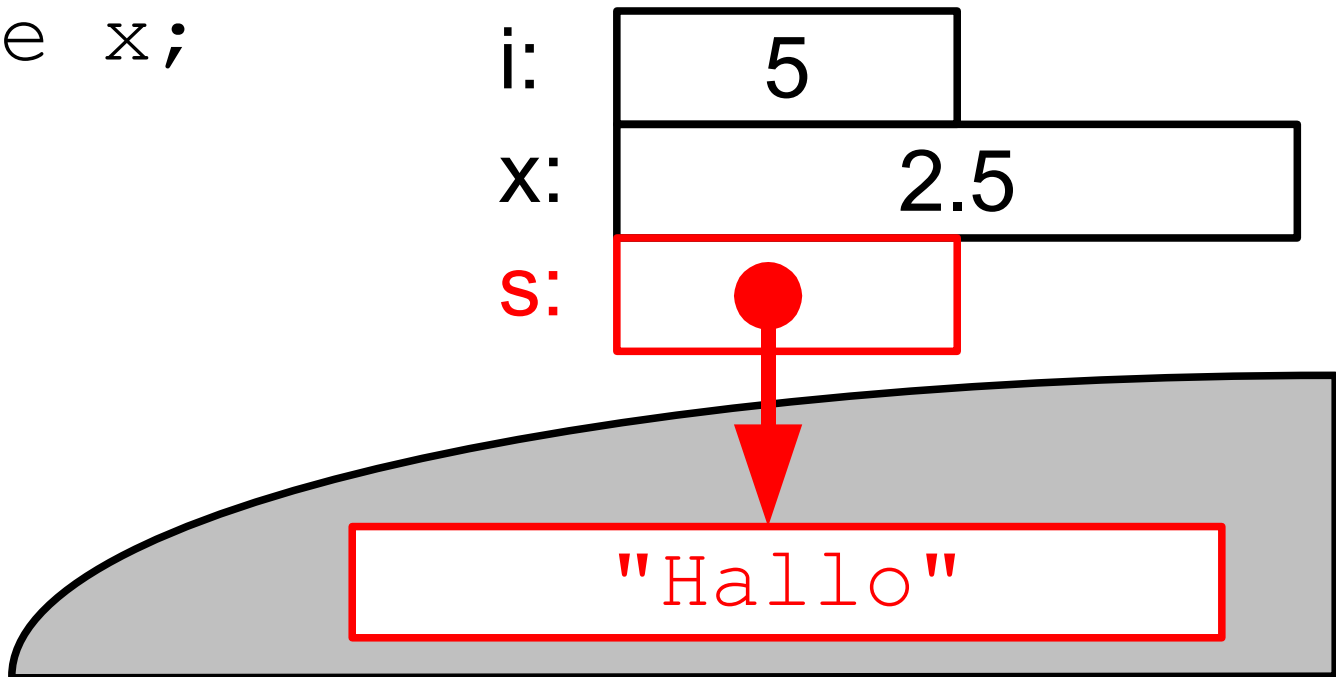


- Beim Verlassen des Blocks wird der Speicherplatz wieder freigegeben
- Dieser Mechanismus lässt sich übrigens sehr effizient mit einem *Kellerspeicher* (*Stack*) realisieren, das kostet kaum Rechenzeit.

Referenz-Datentypen (1)

- Bei Referenzen steht an der Stelle, an der bei primitiven Datentypen der Wert steht, nur ein Verweis auf die eigentlichen Daten (die in einem völlig andern Teil des Speichers liegen).
- Zum Zwecke des Beispiels sei verraten, dass es für Strings einen Referenzdatentyp `String` gibt:

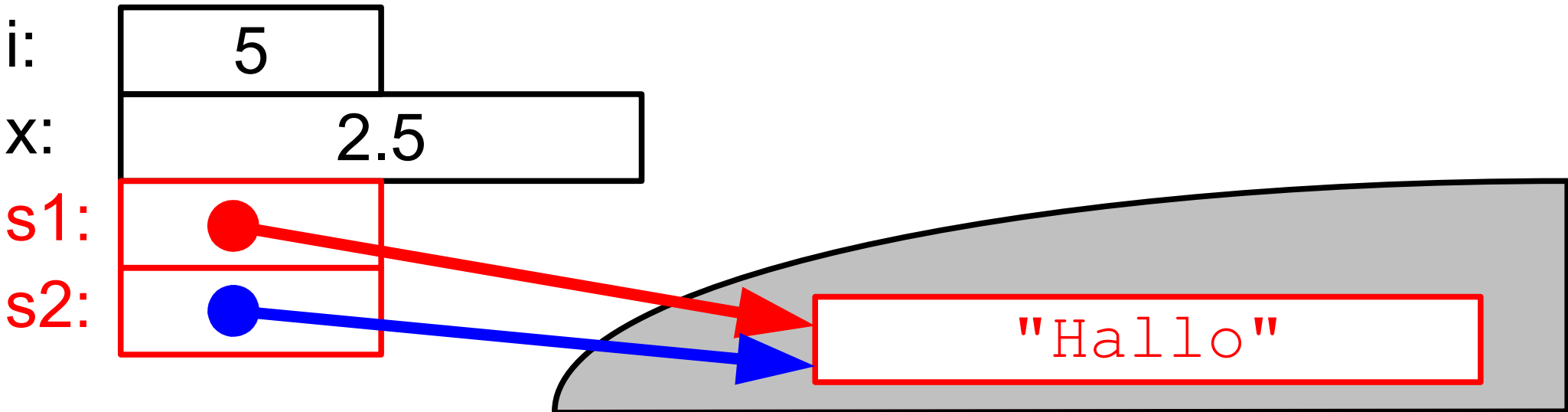
```
{  int i; double x;
   String s;
   i = 5;
   x = 2.5;
   s = "Hallo";
}
```



Referenz-Datentypen (2)

- Bei einer Zuweisung wird eine Kopie der Referenz erstellt, nicht der eigentlichen Daten:

```
{ int i; double x;
  String s1, s2;
  i = 5; x = 2.5;
  s1 = "Hallo";
  s2 = s1;
}
```



Referenz-Datentypen (3)

- In den meisten Programmierprachen kann man zu einem Datentyp T einen neuen Datentyp „Referenz auf ein Objekt vom Typ T“ erzeugen, wodurch Referenzen und referenzierte Objekte unterschieden werden.
- In Java ist das anders:
  - Es gibt keine Referenzen für primitive Datentypen
  - Für alle anderen Datentypen gibt es keine gewöhnlichen Variablen, nur Referenzen
- Ob eine Variable Referenz oder die Daten selber enthält, ergibt sich also aus der Kategorie des Datentyps.

Referenz-Datentypen (4)

☺ Normalerweise müssen wir uns um das Thema „Referenz oder nicht“ keine Gedanken machen.

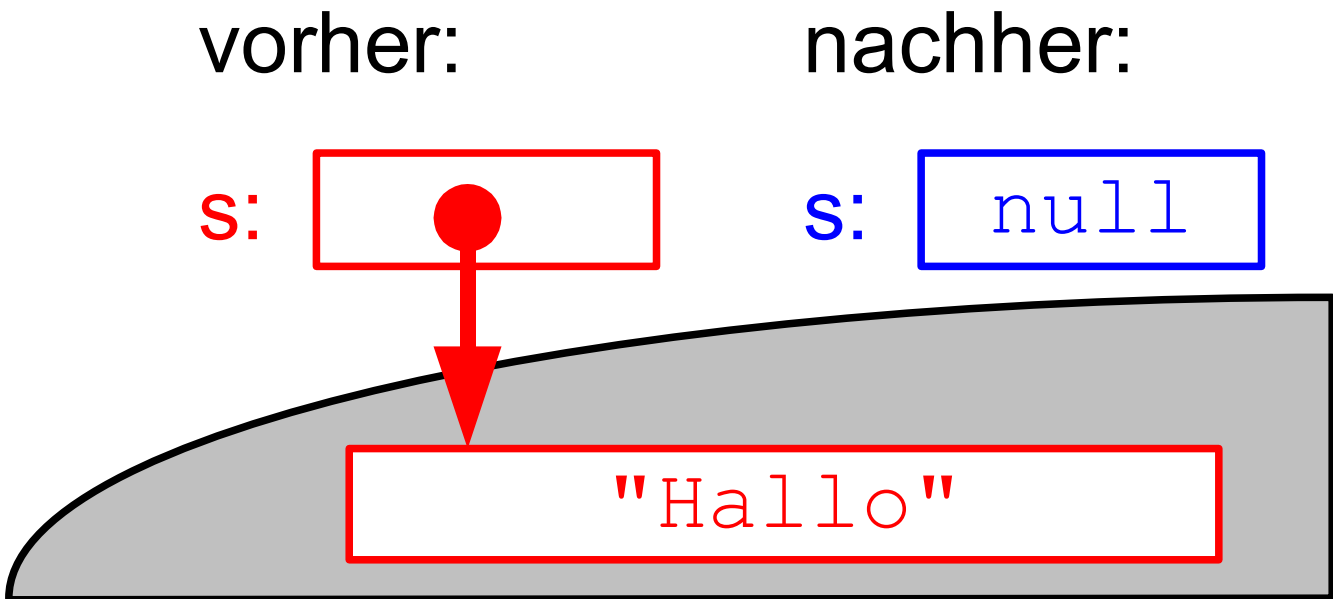
☹ Wenn man das Thema ganz verdrängt, erzeugt man schnell Fehler, die schwierig zu entdecken sind (wir haben z.B. die Referenz kopiert, glauben aber, die Daten kopiert zu haben...)

- In der Summe klar ☺ : das Hantieren mit Referenzen ist in Java weit weniger fehleranfällig als z.B. in C oder C++
- Insbesondere, weil Java sich drum kümmert, wann der Speicherplatz für das referenzierte Objekt freigegeben werden kann: dabei würden wir jede Menge Fehler machen!

Die Referenz `null`

- Gelegentlich hat man eine Referenz, für die man kein Objekt hat – dafür gibt es einen speziellen Wert `null`, der mit allen Referenzdatentypen verträglich ist.
- Ein sinnvolles Beispiel fällt mit mit den bisher vorhandenen Sprachmitteln nicht ein, also kommt ein sinnloses:

```
String s;
s = "Hallo";
// ...
s = null;
// ...
```



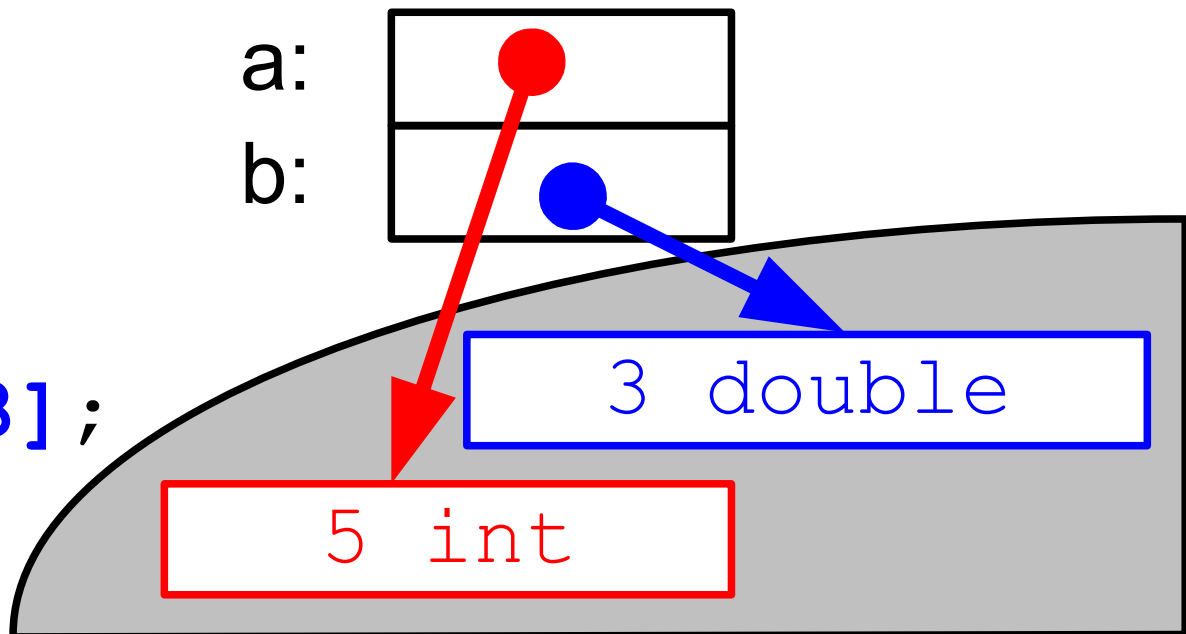
Felder deklarieren

- Für einen beliebigen Datentyp ***T*** bezeichnet ***T*[]** den Datentyp „Feld mit Komponenten vom Typ ***T***“, also die Entsprechung von Listen und Feldern im Maple (nur dass hier alle Komponenten den gleichen, vorher festgelegten Typ haben)
- Über die Zahl der Elemente wird vorerst noch nichts gesagt
- Felder sind Referenz-Datentypen
- Die folgenden Deklarationen reservieren also Speicherplatz für je eine Referenz (!):

```
int[] a;          // Ein Feld aus int
double[] b,c; // Zwei Felder(double)
```

Felder erzeugen (1)

- Nun brauchen wir noch das Feld selbst und dazu
  - den Operator **`new`**,
  - den Datentyp der Feldkomponenten und
  - die Zahl der Komponenten in eckigen Klammern
- Das Ergebnis ist eine Referenz auf den reservierten Speicherplatz, die wir in der Regel abspeichern werden
-
```
int[] a;
double[] b;
a = new int[5];
b = new double[3];
```



Felder erzeugen (2)

- Oft schreibt man das `new` gleich als Anfangswert in die Deklaration:

```
int[] a = new int[5];
```

- Nur in der Deklaration erlaubt ist die Verwendung eines *Literals für Felder* statt der Speicherreservierung mit `new`, wobei die Komponenten des Feldes auch gleich Werte bekommen:

```
int[] a = {1, 1, 2, 3, 5};
```

- Das Literal besteht aus Ausdrücken des passenden Typs, durch Kommas getrennt und in geschweiften Klammern.

Zugriff auf Feldkomponenten (1)

- Der Zugriff auf einzelne Komponenten funktioniert ähnlich wie in Maple (nur dass wir hier bei 0 mit dem Zählen anfangen)
- Mit
  - `x` vom Typ „Feld mit Komponenten vom Typ T“
  - einem ganzzahligen Ausdruck `i`, dessen Wert größer gleich 0 und kleiner der Zahl der Komponenten von `x` ist

  bezeichnet `x[i]` die Komponente Nummer i+1 von `x` und kann wie eine Variable vom Typ `T` verwendet werden (z.B. in Ausdrücken oder auf der linken Seite von Zuweisungen)

Zugriff auf Feldkomponenten (2)

- Eine typische for-Schleife zum Abarbeiten aller Komponenten eines Feldes:

```
int n=10;
int[] feld = new int[n];

for (int i = 0; i<n; i++) {
    feld[i] = i*i;
}

for (int i = 0; i<n; i++) {
    System.out.println(feld[i]);
}
```

Felder als Methodenparameter (1)

- Felder als Parameter (oder Ergebnisse) von Methoden sind kein Problem
- Nützlich ist dabei, dass für ein Feld `x` der Ausdruck `x.length` die Zahl der Komponenten angibt.

```
static void quadrate(int[] arr) {
    for (int i=0; i<arr.length; i++)
        arr[i] = i*i;
}
static void quadrateTest(int n) {
    int[] x = new int[n];
    quadrate(x);
    for (int i=0; i<n; i++)
        System.out.println(x[i]);
}
```

Felder als Methodenparameter (2)

- An vorigem Beispiel ist interessant, dass die Methode `quadrate` das Feld verändern kann, obwohl in Java Parameter als Wert übergeben werden (*Call by value*)
- Dass das geht, liegt daran, dass Felder Referenz-Datentypen sind: es wird die Referenz als Parameter übergeben, nicht das Feld selbst

- Die Variable `x` in der Methode `quadrateTest` kann von der Methode `quadrate` nicht verändert werden
- Das, worauf sie verweist, hingegen schon!

Mehrdimensionale Felder (1)

- Für mehrdimensionale Felder (z.B. Matrizen) mit Komponenten vom Typ `T` stehen bei der Deklaration so viele `[]`-Paare wie Dimensionen:
  ```
  double[][] A; // zweidimensional
  ```
- `A` enthält nun eine Referenz auf ein Feld, das aus Referenzen auf Felder besteht, deren Komponenten vom Typ `double` sind.
- Um ein Feld zu bekommen, kann man `new` mit entsprechend vielen Längenangaben versehen:
  ```
  A = new double[3][2];
  ```
  Das gibt eine Referenz auf ein Feld mit drei Komponenten, jede davon eine Referenz auf ein Feld von zwei `double`.

Mehrdimensionale Felder (2)

- Auf eine einzelne Komponente kann man mit einem doppelten Index zugreifen:

```
for (int i=0; i<A.length; i++) {
    for (int j=0; j<A[i].length; j++) {
        A[i][j] = i*j;
    }
}
```

- Wie man am Ausdruck **`A[i].length`** sieht, kann auch eine Zeile als Ganzes auftreten (`A[i]` kann verwendet werden wie eine Variable vom Typ „Feld mit Komponenten vom Typ `double`“)