Einführung in die Programmierung

Teil 5: Referenzdatentypen

Prof. Dr. Peer Kröger,
Florian Richter, Michael Fromm

Wintersemester 2018/2019

Übersicht

Kapitel 1: Referenzen i

Motivation

- Das Verständnis von Referenz(daten-)typen ist zentral für die objekt-orientierte Programmierung.
- Bisher haben wir nur mit einfachen Datentypen gearbeitet.
- *Referenzdatentypen* sind Konstrukte, mit deren Hilfe wir aus einfachen Datentypen “eigene”, komplexere Typen definieren und erzeugen können.
- Damit können wir z.B. Mengen von Werten verwalten, aber auch komplexere Datentypen (später z.B. oo Klassen).

Referenzen

- Das Konzept der Referenzen ist dabei nichts oo-spezifisches, das Konzept gab es bereits in “alten” Programmiersprachen und ist z.B. auch in C fest verwurzelt.
- Um es zu verstehen, müssen wir einen tieferen Blick in die Speicherverwaltung von Programmiersprachen unternehmen.
- Wir orientieren uns dabei an der Speicherverwaltung von Java (genauer gesagt der JVM), die meisten Konzepte gelten aber analog für andere Programmiersprachen.

Daten im Speicher

- Die Speicherumgebung besteht typischerweise aus zwei getrennten Bereichen im (Haupt-)Speicher, dem sog. *Stack* (*Keller*, *Stapel*) und dem sog. *Heap* (*Halde*).
- Ein Stack kann Objekte gleicher Größe dynamisch verwalten.
- Ein Heap dagegen verwaltet Objekte sehr unterschiedlicher Größe dynamisch in einem gemeinsamen Speicherbereich (*dynamic storage allocation*).
- Die Speicherumgebung muss Zustandsübergänge (so wie wir sie in Kapitel 5 eingeführt haben) effizient durchführen.
- Dazu gehört das Anlegen neuer sowie das Ablesen und Verändern bestehender Variablen (Zettel), also Paaren (x, d).

Daten im Speicher

- Der Stack modelliert die imperative Programmstrukturierung durch verschachtelte (Anweisungs-)Blöcke wie Klassen, Methoden, Kontrollstrukturen etc.
- Ein Block führt Namensbindungen (Zettel) ein, die zusätzlich zu den Bindungen (Zetteln) außerhalb des Blocks gelten.
- Nach Verlassen des Blocks gelten wieder nur noch jene Bindungen, die bereits vor Betreten des Blockes galten.
- Der Stack erlaubt (wie bei einem echten “Stapel”) entsprechend effizient die Bindungen, die zuletzt hinzukamen, als erste wieder zu entfernen (*Last-in-first-out*, *LIFO*).
- Am besten stapeln lassen sich i.Ü. gleichgroße Objekte (also z.B. die Werte der primitiven Datentypen).

Daten im Speicher

- Alle lokalen Variablen, d.h. Paare (Zettel) $z = (x, d)$, eines Zustands $\mathcal{S}$ werden grundsätzlich im Keller verwaltet.
- Der Name einer Variablen x wird intern in eine Speicheradresse[1] des Stacks übersetzt.
- An dieser Speicheradresse steht deren Inhalt d als Binärdarstellung (dessen Länge fix ist).
- Globalen Variablen sind (zunächst) in jedem Block sichtbar und sind daher in einem eigenen Bereich, auf den von anderen Bereichen zugegriffen werden kann, abgelegt (sog. *Constant Pool*).
- Das war bisher unser abstraktes Bild der Zettel.

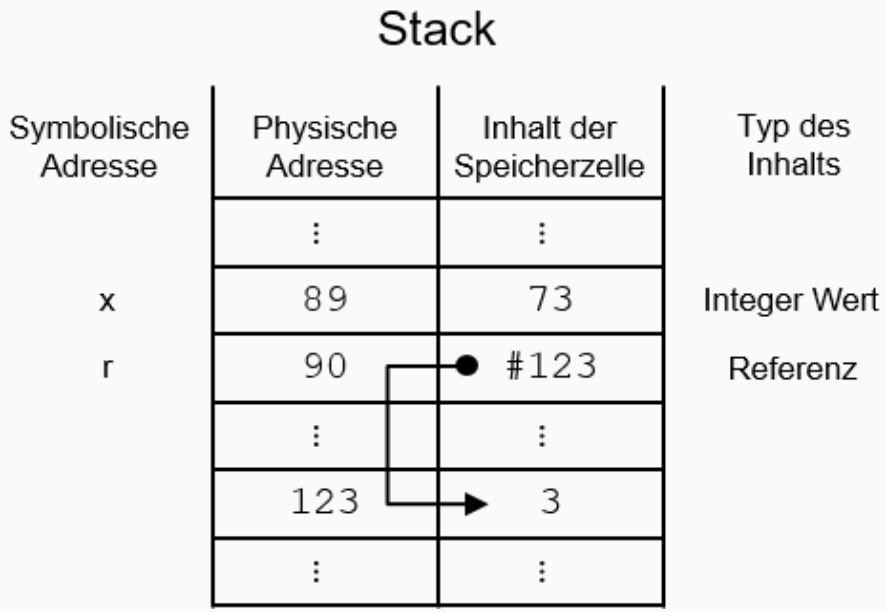
[1] Die Speicheradressen (in Binärdarstellung) sind übrigens auch alle gleich lang

Referenzen im Speicher

- So eine (physische) Speicheradresse nennt sich *Referenz*.
- Der Bezeichner der Variable (Zettel) ist quasi eine symbolische Referenz: da es zu mühsam ist, mit echten Speicheradressen zu arbeiten, arbeiten wir lieber mit Variablennamen (tatsächlich war das gaaaanz früher nicht so, einer der ersten großen Errungenschaften höherer imperativer Sprachen).
- Java ist hier sehr angenehm, da wir uns mit der physischen Speicheradresse nicht rumärgern müssen, und nicht mal Zugriff auf deren “Wert” haben (z.B. bei C mit dem Pointer-Konzept gibt es dagegen explizit die Möglichkeit auch auf die physische Adresse zuzugreifen).

Referenzen und Daten im Speicher

- Prinzipiell könnte man aber natürlich auch eine Referenz auf eine andere Speicheradresse in einer Speicheradresse ablegen:



Bemerkung: Adressen und Inhalte sind natürlich eigentlich in Binärdarstellung.

Referenzen und Daten im Speicher

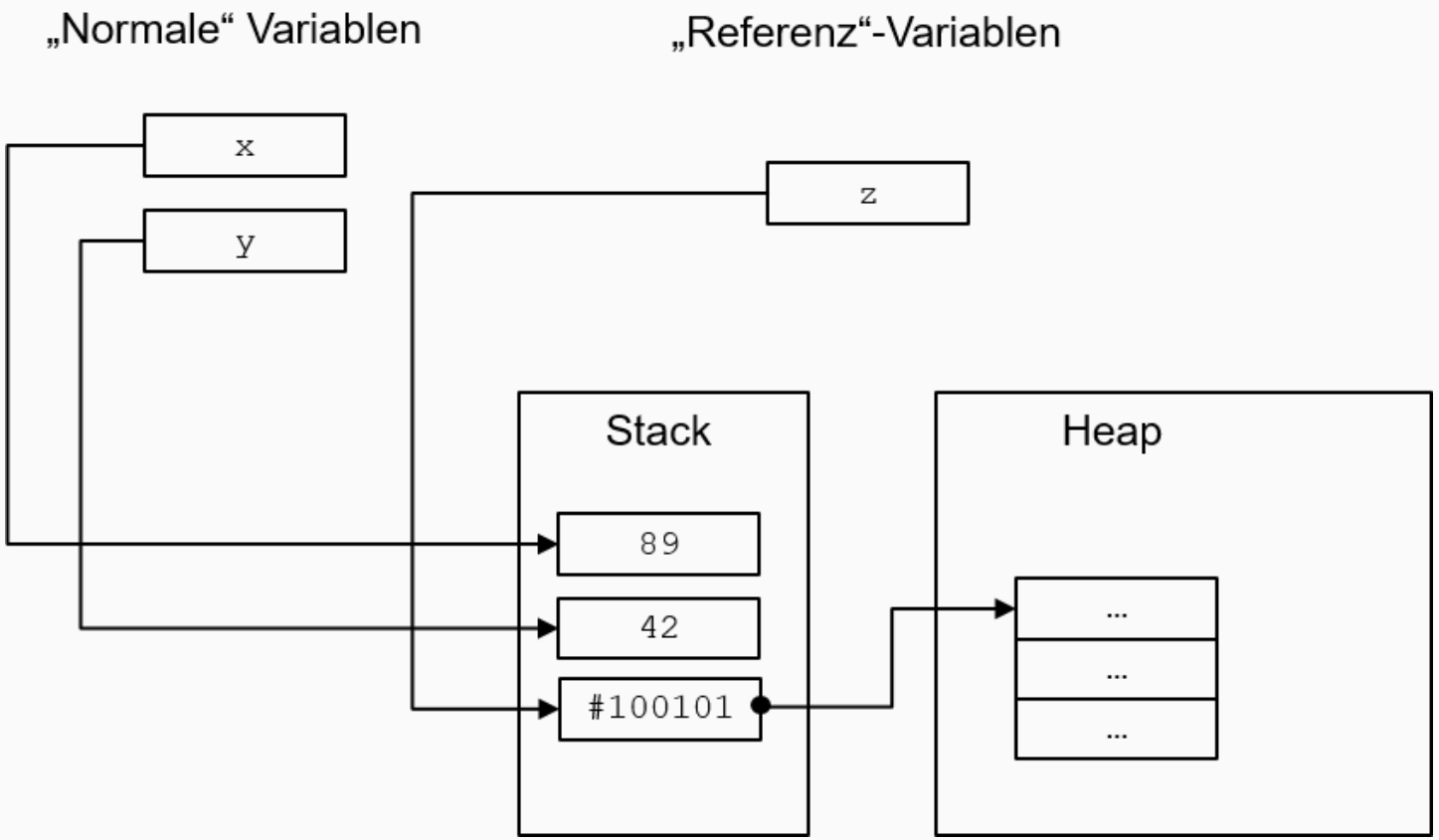
- In diesem Beispiel ist also *x* eine “normale” Variable, auf dessen Zettel als Wert tatsächlich ein Element aus dem Wertebereich der Sorte von *x* steht (hier 73).
- Dagegen ist *r* eine “Referenz”-Variable, die auf eine Speicheradresse referenziert, wo dann der tatsächliche Wert (hier 3) steht.
- Aber brauchen wir solche Referenz-Variablen überhaupt???

Referenzen und Daten im Speicher

- Naja, der Stack ist leider nur gut darin Sachen (genauer: Objekte, Werte) zu verwalten, die gleich groß sind.
- Häufig (speziell in oo Programmen) muss man aber Objekte (Werte) verwalten, die unterschiedlich groß sind oder sein können.
- Dazu dient der Heap (ach ja, den gibt's ja auch noch ...), d.h. diese Objekte werden tatsächlich "auf Halde gelegt".
- Verwaltet werden sie (in Java) aber als "normale" Variablen im Stack, d.h. dort stehen jetzt Speicheradressen (Referenzen) des Heaps.

Referenzen und Daten im Speicher

- Der Vergleich schematisch:



Implikationen

- Wie gesagt, wir werden zwei unterschiedliche Arten von Variablen haben:
  - Variablen mit primitivem Daten-Typ: so wie bisher ist hier auf dem Zettel der Wert des “Objekts” gespeichert, das diese Variable repräsentiert.
  - Variablen mit Referenz-Typ: hier ist auf dem Zettel nur die Referenz auf den Speicherort (typw. auf dem Heap), wo wir den eigentlichen Wert finden.
- Dies hat weitreichende Folgen auf die Eigenschaften bzw. den Umgang mit diesen Variablen im Kontext der Programmierung

  ...

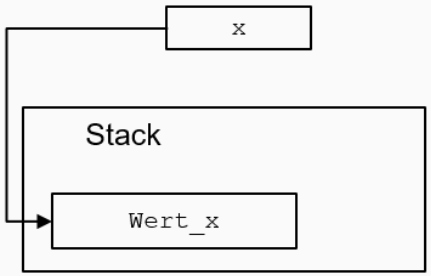
Implikation 1: Call-by-reference-Effekt

- Call-by-reference-Effekt troz call-by-value-Übergabe:
  - Bei Übergabe einer lokalen Variablen x an eine Prozedur *veraendere*(e) (z.B. durch Aufruf von *veraendere*(x)) wird für den Eingabeparameter (e) ein eigener Zettel angelegt und der Wert $W(x)$ an e übergeben.
  - Ist x eine normalen Variable, bekommt e quasi eine Kopie, *veraendere*(x) arbeitet entsprechend auf dieser Kopie, Veränderungen an der Kopie ändert nicht den urspr. Wert von x (alles wie gehabt).
  - Ist x eine Referenz-Variable ist $W(x)$ allerdings die Referenz auf den tatsächlichen Wert von "Objekt" x (der z.B. auf der Halde gespeichert ist), d.h. e enthält eine Kopie der Referenz auf den urspr. Speicherort, *veraendere*(x) arbeitet damit auf dem selben Objekt (Änderungen sind anschließend sichtbar).

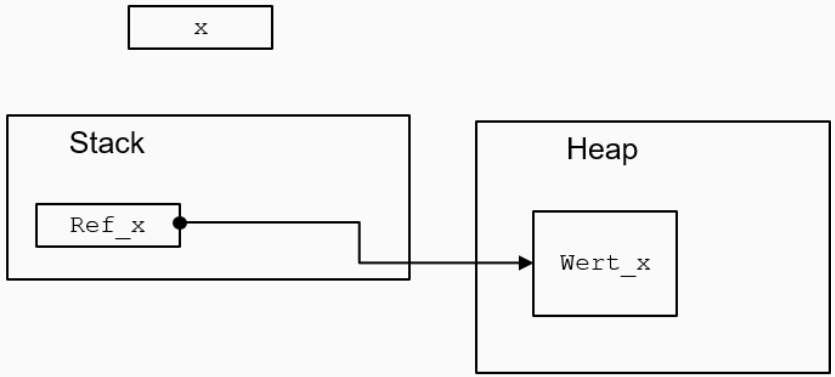
Implikation 1: Call-by-reference-Effekt

- Schematisch:

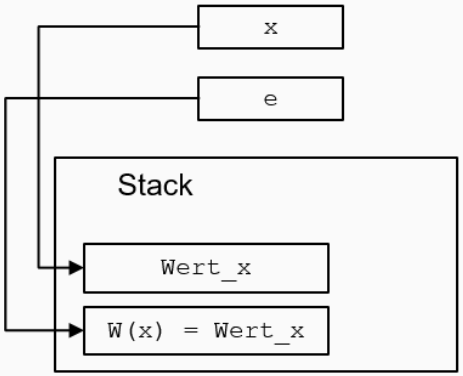
„Normale“ Variable

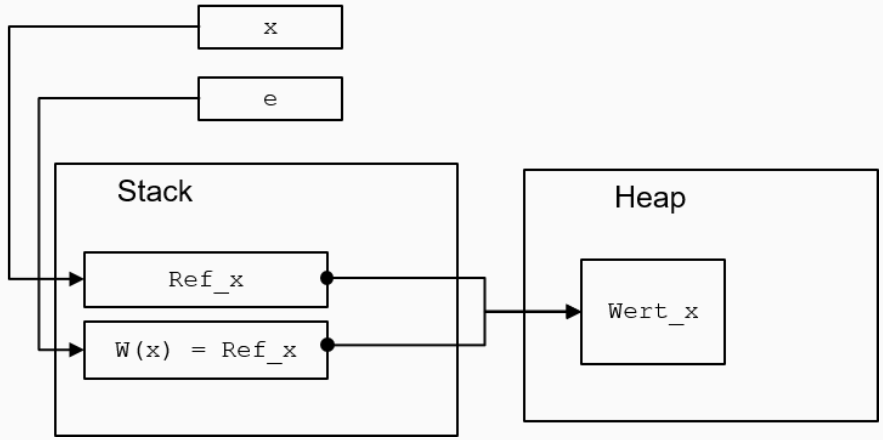


„Referenz“-Variable



Übergabe einer lokalen Variable x an eine Prozedur `veraendere(e)` durch Aufruf von `veraendere(x)`



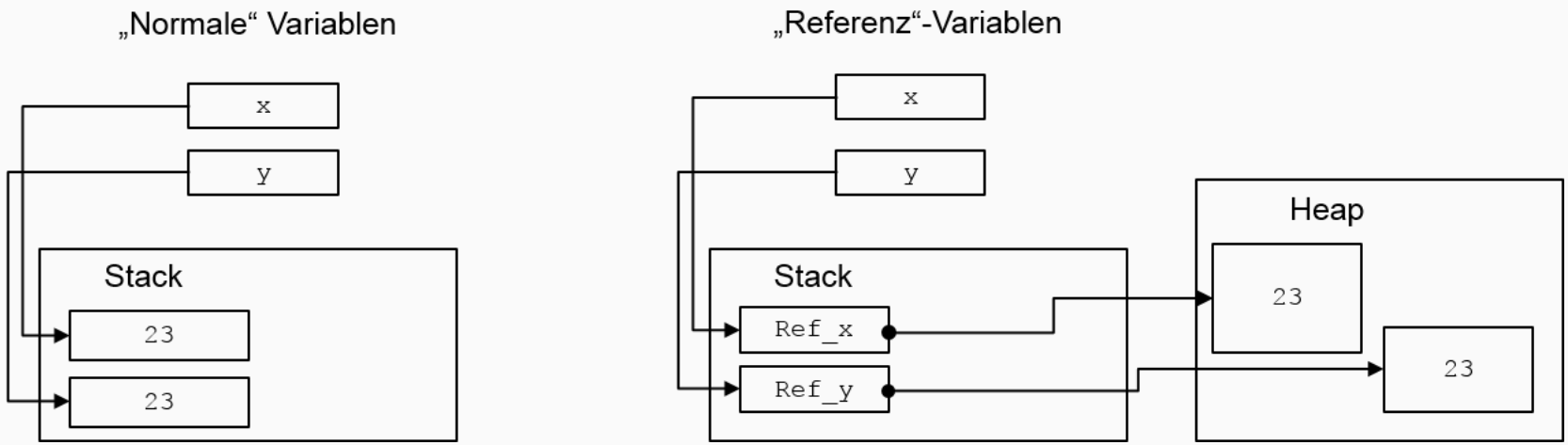


Implikation 2: Gleichheit vs.Identität

- Gleichheit von Variablen ist nicht gleich Gleichheit:
  - Die Gleichheit von Variablen x und y haben wir bisher mit dem Gleichheits-Operator (-Prädikat) überprüft: der Ausdruck $x = y$ prüft, ob x und y den gleichen Wert haben, d.h. $W(x = y)$ hängt von $W(x$ und $W(y)$ ab.
  - Bei normalen Variablen ist alles wie gehabt.
  - Bei Referenz-Variablen sind $W(x)$ und $W(y)$ allerdings wiederum nur die Referenze auf die tatsächlichen Werte, d.h. $x = y$ prüft nur auf die Gleichheit der Referenzen (“Identität”).
  - x und y können theoretisch zwar unterschiedliche Speicheradressen haben, aber dieselben Werte besitzen. In diesem Fall wäre aber $W(x = y) = FALSE$.

Implikation 2: Gleichheit vs. Identität

- Schematisch:



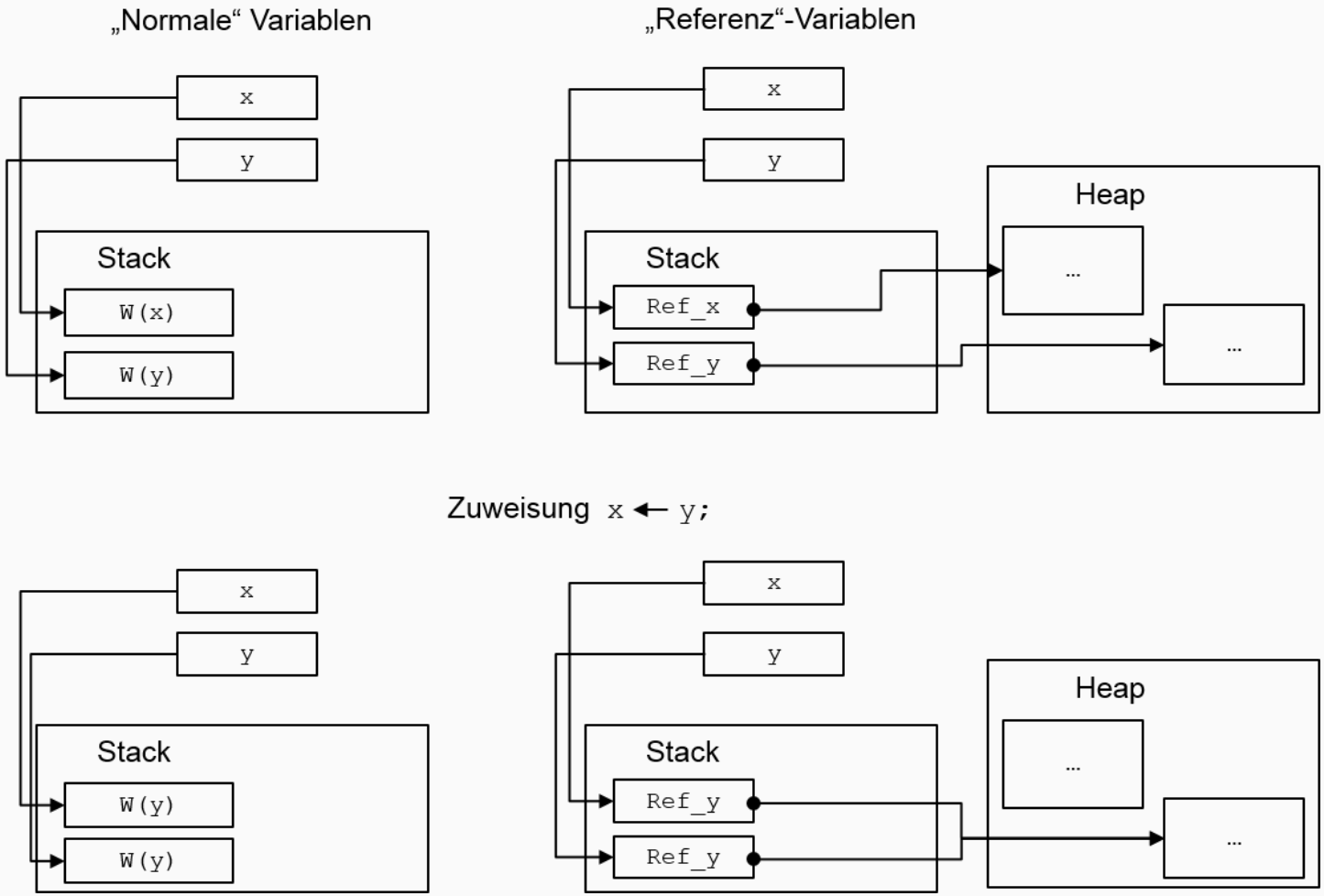
- In beiden Fällen haben *x* und *y* den selben Wert; dadurch, dass sie aber unterschiedliche Referenzen haben, werden sie im rechten Fall nicht als gleich betrachtet.

Implikation 3: Zuweisung/Kopie

- Bei der Zuweisung müssen wir aufpassen:
  - Wenn wir x den Wert einer anderen Variablen y zuweisen ($x \leftarrow y$), substituieren wir x durch $W(y)$.
  - Bisher "kopieren" wir dadurch den Wert von y, falls wir später x verändern würden, bleibt y davon unberührt.
  - Sind x und y Referenz-Variablen ist das nicht mehr so: $x \leftarrow y$ weist x die Referenz des Objekts zu, das von y referenziert wird, d.h. beide Variablen referenzieren jetzt das selbe Objekt auf der Halde.
  - Änderungen an x ändern damit das gemeinsam Objekt, also auch y (und andersrum).
- Dieses "Kopieren" der Referenz (anstelle des tatsächlichen Objekts/Werts) nennt man *flache Kopie* (shallow copy). Wird wirklich auch das Objekt auf der Halde kopiert, ist das eine *tiefe Kopie*.

Implikation 3: Zuweisung/Kopie

- Schematisch:



Road Map

- Variablen von primitive Datentypen und Referenztypen müssen also evtl. unterschiedlich behandelt werden, bzw. haben unterschiedliche Auswirkungen auf unsere Programme.
- Das behalten wir im Kopf und schauen uns jetzt verschiedene Referenztypen aus dem Blick der Daten(-Modellierung an).
- D.h., wir kümmern uns um die Frage, wozu Referenztypen gut sind und wo bzw. wie sie uns bei der Verarbeitung von Daten helfen.

Motivation

- Bisher haben wir einzelne Objekte der Grunddatentypen (Menge der bereitgestellten Sorten) verarbeitet.
- Oft werden aber auch Mengen bzw. Multimengen von Objekten verarbeitet (z.B. bei unseren Wechselgeldalgorithmen, die die Ausgabe als Folge von EUR-Münzen/-Scheinen modelliert hatten).
- Unsere bisherigen Konzepte geben die Verarbeitung von (Multi-)Mengen von Objekten allerdings nicht her.
- In einer Programmiersprache sind daher üblicherweise einfache *Datenstruktur* eingebaut, die es ermöglicht, eine Reihe von Werten (gleichen Typs) zu modellieren, oder Möglichkeiten vorgesehen, diese selbst zu definieren.

- In Java gibt es beide Möglichkeiten:
  1. Arrays (Felder):
     Der eingebaute Datentyp der Arrays entspricht dem Konzept der Folgen über Elemente einer Menge. Eine spezielle Form von Arrays sind *Zeichenketten* (*Strings*), bei denen die Objekte der Menge vom Typ *CHAR* (bzw. in Java **`char`**) sind.
  2. Klassen:
     Klassen bieten die Möglichkeit Tupel (Elemente des kartesischen Produkts) aus unterschiedlichen Mengen zu bilden. Diese werden (speziell in älteren Programmiersprachen) oft auch *Records* genannt. Wir werden sehen, dass das oo Klassenkonzept aber noch viel mehr als die ursprünglichen Records zu bieten hat. Die o.a. Strings sind in Java als Klasse implementiert.

Kapitel 2: Arrays (Felder, Reihungen) i

Wozu brauchen wir Arrays

- Nehmen wir an, wir wollen unseren Kalender mit dem Computer verwalten und wir wollen das natürlich selbst programmieren.
- Unser Kalender soll für jede Stunde eines Tags einen Eintrag ermöglichen.
- Als Eintrag vergeben wir Buchstaben als Kürzel für die Termine (z.B. Kürzel für die Vorlesungen, Mittagessen, etc.).
- Wenn wir davon ausgehen, dass wir entsprechende Methoden in Java zur Verfügung haben, wollen wir außerdem noch einzelne Einträge über die Kommandozeile eingeben und den gesamten Kalender ausgeben können.

Wozu brauchen wir Arrays

- Die Verarbeitung der Daten mit unseren bisherigen Mitteln wäre leider extrem aufwendig.
- Wir müssten für jeden möglichen Eintrag eine eigene Variable vom Typ `char` vereinbaren.
- Wenn wir davon ausgehen, dass der Tag 24 Stunden und die Woche 7 Tage hat, müssten wir alleine für eine Woche 168 Variablen vereinbaren.

Wozu brauchen wir Arrays

Hier ein paar Java-Code-Schnippsel:

```
public class KalenderDummy {
  public static void main
    (String[] args) {

    // Eine Variable
    // pro Stunde und Tag
    char mo_00 = '-';
    char mo_01 = '-';
    ...
    char so_23 = '-';

    // Termin-Eingabe
   int tag = ...
   int stunde = ...
   char eintrag = ...
    ...
```

```
...
// Termin eintragen
switch(tag) {
 case 1: switch(stunde) {
   case 0 : mo_00 = eintrag;
   break;
   case 1 : mo_01 = eintrag;
   break;
   ...
 }
 case 2: switch(stunde) {
 ...
}

// Termine anzeigen
...
```

Wozu brauchen wir Arrays

- Die aufwendige Fallunterscheidung ist leider notwendig, um aus den 168 Variablen die richtige zu wählen, in die wir einen neuen Eintrag schreiben müssen.
- Die Ausgabe wäre ähnlich aufwendig.
- Für ein ganzes Jahr wäre das Programm der totale Overkill ...

Wozu brauchen wir Arrays

- Woran hängt es?
- Wir verwalten eine Menge von gleichartigen Werten (alle vom Typ **`char`**) aber wir haben kein Konstrukt für Mengen von Werten.
- Daher verwalten wir eine Menge von Variablen mit dem selben Typ (und sogar bis auf die Nummerierung und den Namen des Tags mit “ähnlichen” Bezeichnungen).
- Apropos Bezeichnungen der Variablen: in denen steckt so etwas wie ein Index (`tag_stunde`)! Wie wäre es, eine Variable zu haben und über einen Index auf den gewünschten Inhalt zuzugreifen ...?

Was sind Arrays

- Ein *Array* (Reihung, Feld) ist genau so ein Konstrukt: es ist eine Menge von Werten gleichen Typs, auf die über einen Index direkt zugegriffen werden kann.
- Formal handelt es sich dabei um eine Folge, d.h. ein Array von Objekten einer bestimmten Sorte S ist ein Element aus S^*.
- Ein Array mit n Komponenten über einer Sorte S kann als Abbildung von der Indexmenge I_n auf die Menge S interpretiert werden.

Arrays als Datentyp

- Ein Array über einer Sorte S ist ein eigener Datentyp.
- Was bedeutet das?
- Ein Datentyp sollte vielleicht ein paar passende “Grundoperationen” bereit stellen, damit man mit ihm vernünftig arbeiten (programmieren) kann.
- Ein Datentyp kann der Typ einer Variablen sein, d.h. wir können Variablen dieses Typs vereinbaren und vielleicht auch (mit entsprechenden Operationen) initialisieren, etc.
- Über diese Variablen und mit den Grundoperationen kann ich Ausdrücke bilden, etc. (also ganz allgemein: programmieren).

- Die Operationen sollten die wesentliche Eigenschaften von Arrays, die sie von anderen Datentypen unterscheiden, widerspiegeln, insbesondere:
  - Die Länge eines Arrays ist fix, d.h. sie können nicht dynamisch wachsen oder schrumpfen (d.h. die Menge I_n verändert sich nicht).
  - Arrays erlauben direkten Zugriff (lesend/schreibend) auf ihr i-tes Element ($i \in I_n$).
- Diese könnten wir wie gewohnt in ein entsprechendes Modul schreiben und damit allgemein zur Verfügung stellen.
- Das Verhalten dieser Operationen ließe z.B. durch axiomatische Spezifikation definieren.

Arrays: Grundoperationen

- Die typischen Grundoperationen von Arrays sind:
  - Der Zugriff auf die obere Grenze des Indexbereichs (Länge).
  - Der Zugriff auf die *i*-te Komponente (Projektion).
  - Ein Konstruktions-Operator, der ein Array einer gewissen Länge (d.h. die Indexmenge ist damit festgelegt) erzeugt. Wie besprochen ist die Länge des Arrays dann nicht mehr veränderbar (statisch). Diese Operation legt also insbesondere die Indexmenge fest und erzeugt ein Array mit "Dummy"-Werten.
  - Das Verändern des *i*-ten Elements.

Arrays: Grundoperationen

- Um ihre Signaturen anzugeben, bezeichnen wir die Menge aller Arrays über der Sorte S mit **`ARRAY`** S und nehmen an, dass wie bei den Folgen gilt: $I_n \subset \mathbb{N}^2$.
- Außerdem nehmen wir an, dass jede Sorte S noch ein spezielles Element ω_S enthält, dass für "undefiniert" steht (oder alternativ für ein Default-Element).
- Dieses Element brauchen wir für den Konstruktions-Operator, also zur Erzeugung eines neuen Arrays.
- Dabei soll ja die Indexmenge festgelegt werden und ein neues ("leeres"?) Array erzeugt werden.

[2] Achtung: $I_n = \{1, \ldots, n\}$. Dazu später mehr!!!

Arrays: Grundoperationen

- Somit wären:
  - Die Konstruktion (Erzeugung) eines Arrays der Lände n:
    $INIT_S : \mathbb{N} \to \texttt{ARRAY}\ S$
    mit $INIT_S(n) = (\omega_S, \omega_S, \ldots, \omega_S) \in S^n$
  - Die Länge des Arrays:
    $LEN : \texttt{ARRAY}\ S \to \mathbb{N}$
  - Der Zugriff auf die i-te Komponente (Projektion):
    $PROJ : \texttt{ARRAY}\ S \times \mathbb{N} \to S$
  - Das Verändern des i-ten Elements
    $ALT : \texttt{ARRAY}\ S \times \mathbb{N} \times S \to \texttt{ARRAY}\ S$
    mit $ALT(x, i, a) = (x_1, \ldots, x_{n-1}, a, x_{n+1}, \ldots, x_n)$
    für $x = (x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n)$.

- Mit diesen Grundoperationen könnten wir jetzt weitere Algorithmen entwickeln, wie z.B. die Frage, ob einesObjekts $a \in S$ in einem Array $x \in$ **array** S enthalten ist:

```
PROCEDURE enthalten(x : ARRAY S, a : S) → 𝔹
    OUTPUT  gibt TRUE falls a in x enthalten, sonst FALSE
    BODY
        VAR  gefunden : 𝔹, i : ℕ;
        gefunden ← FALSE;
        i ← 1;
        WHILE  ¬gefunden ∧ i ≤ UP(x)  DO
            IF  PROJ(x, i) = a  THEN  gefunden ← TRUE;  ENDIF
            i ← i + 1;
        ENDDO
        RETURN  gefunden;
```

- Oder auch in anderen Algorithmen verwenden, z.B. eine natürliche Zahl in ein Array aus Ziffern umwandeln:

PROCEDURE *natToArray*$(n : \mathbb{N}_0) \rightarrow$ **ARRAY** $\mathbb{N}_0$
  OUTPUT gibt die Zifferen der Zahl n als Array zurück
  BODY
    VAR $stellen, zaehler : \mathbb{N}$, a : **ARRAY** $\mathbb{N}_0$;
    $stellen \leftarrow stellen(n)$; *(– Funktion aus Teil4, Kapitel 3 –)*
    $a \leftarrow INIT_{\mathbb{N}_0}(stellen)$;
    $zaehler \leftarrow stellen$;
    WHILE $zahler > 0$ **DO**
      ALT(a, zaehler, MOD(x,10));
      $x \leftarrow DIV(x, 10)$;
      $zaehler \leftarrow zaehler - 1$;
    ENDDO
    RETURN *gefunden*;

Array-Variablen

- Dabei haben wir nun noch nicht genau geklärt, wie wir mit den Variablen vom Typ **`ARRAY`** *S* umzugehen haben.
- Wenn wir so tun, als passt auf den entsprechenden Zettel immer das Array komplett drauf, d.h. auf dem Zettel steht immer noch ganz normal der Wert der Variable und nicht eine Referenz (in diesem fall wäre alles OK).
- Da die Größe eines Arrays aber möglicherweise erst dann bekannt ist, wenn der Algorithmus ausgeführt wird (also zur Laufzeit)[3], ist es für einen Rechner schwierig, zu antizipieren, wie groß die Speicherzelle (der Zettel) sein muss.

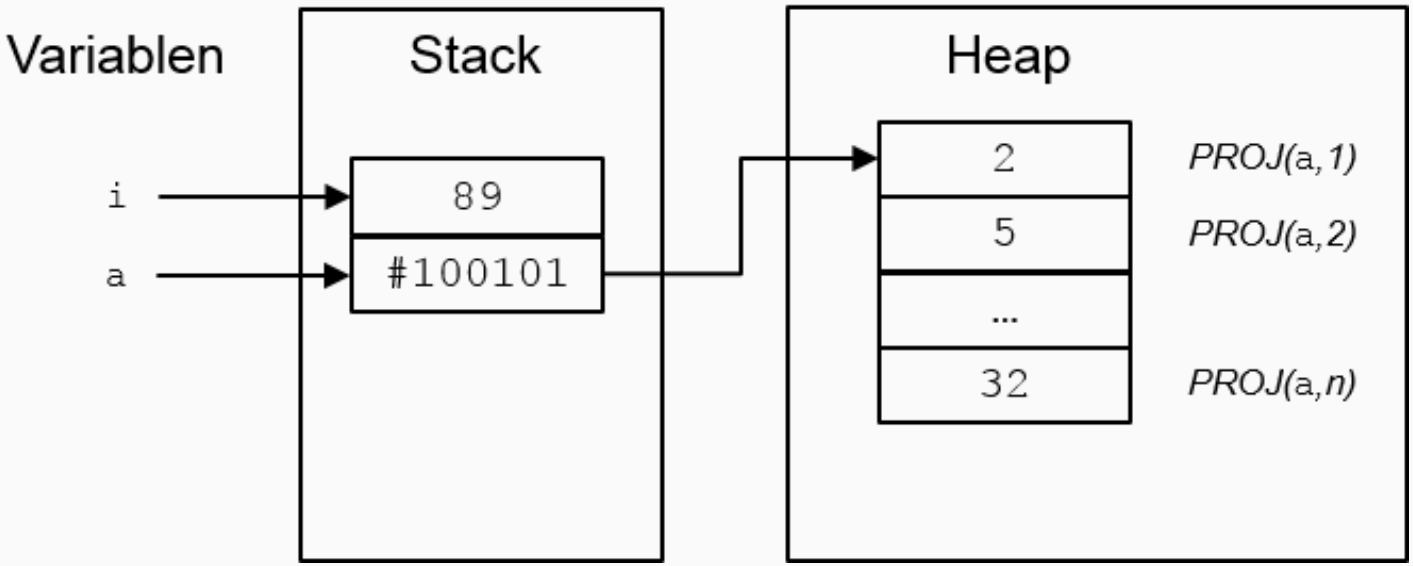
[3] Im Algorithmus *natToArray* hängt die Länge z.B. von *n* ab.

Arrays als Referenztypen

- Deshalb ist in den meisten Programmiersprachen ein Array ein Referenztyp, d.h. eine Variable vom Typ **`ARRAY`** *S* speichert nicht das Array direkt, sondern eine Referenz auf den Ort, wo die einzelnen Elemente gespeichert sind (meist die Halde).
- Das Array selbst (also die einzelnen Elemente) kann man als Sammlung von einzelnen Zetteln auffassen (wir wissen nur nicht immer vorab, wie viele Zettel gebraucht werden).
- Die Halde ist der Ort, wo Objekte unterschiedlicher Größe gespeichert werden können, also insbesondere Arrays.

Arrays als Referenztypen

- Bildlich:



Dabei ist *i* eine “normale” Variable, z.B. vom Typ $\mathbb{N}_0$ und *a* eine Array-Variable z.B. vom Typ **`ARRAY`** $\mathbb{N}_0$.

Array-Variablen

- Wenn wir das berücksichtigen wollen, müssen wir wie besprochen “normale” Variablen und “Referenz”-Variablen unterscheiden!
- Wir bleiben intuitiv (und ersparen uns eine genaue Formalisierung):
- Für eine Variable der Sorte S hatten wir das Paar (x, d) mit $d = W(x) \in S$ verwaltet.
- Für eine Variable vom Typ **`ARRAY`** S müssen wir nun das Paar (x, r) verwalten, wobei r die Referenz auf $W(x) \in$ **`ARRAY`** S darstellt, bezeichnet als $Ref(x)$, d.h. $W(x)$ ist nun eine Referenz (auf ein Array).

Array-Variablen

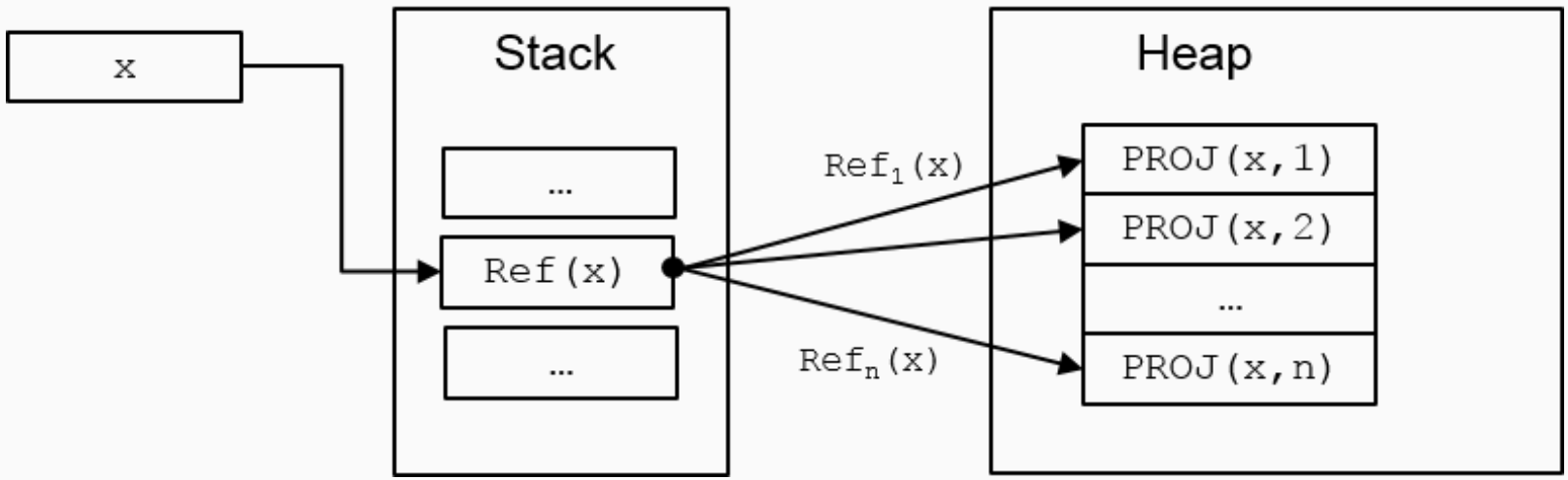
- Dementsprechend sind die einzelnen Elemente des Arrays x wiederum “interne” Variablen die wir über die Referenz $Ref(x)$ ansprechen können:
  - $Ref_1(x)$ bezeichnet den Zettel des ersten Elements
  - $Ref_2(x)$ bezeichnet den Zettel des zweiten Elements
  - ...
  - $Ref_n(x)$ bezeichnet den Zettel des letzten Elements

  d.h. $Ref(x)$ ist eigentlich so etwas wie eine Sammlung von Referenzen[4].
- Ein “Literal” aus **`ARRAY`** S ist also genau genommen jetzt eine Referenz auf das konkrete Objekt (hier: Array) und es gibt einen “Mechanismus” (die $Ref_i(x)$), um auf die einzelnen Komponenten zuzugreifen.

[4] Das ist aber ein schiefes Bild, auf dem Zettel im Stack steht nur eine Referenz.

Array-Variablen

- Bildlich:



d.h. z.B. $W(PROJ(x,1)) = W(Ref_1(x))$.

Array-Variablen

- Diese “Formalisierung” entspricht in etwa der Implementierung von Arrays in Java (und den meisten anderen Sprachen).
- Die zuvor diskutierten Implikationen (call-by-reference-Effekt, Zuweisung, Gleichheit, ...) gelten entsprechend.
- Apropos Java: lassen Sie uns doch gleich einen Blick auf die Umsetzung von Arrays in Java werfen.