Überblick

Motivation

- Klassen stellen ein mächtiges Werkzeug dar, eigene, komplex strukturierte Datentypen zu modellieren und bereit zu stellen.
- Die „Literale“ dieser Datentypen sind die entsprechend strukturierten Objekte.
- In der Praxis hat man es aber auch häufig mit Datentypen zu tun, deren Werte (Literale) aus einem konstanten Wertevorrat stammen (so wie eigentlich auch die primitiven Datentypen).

Motivation

Beispiele für solche Datentypen (Werte-Mengen):

- Die Menge der Jahreszeiten:
  {FRÜHLING, SOMMER, HERBST, WINTER}
- Die Menge der Monate:
  {JANUAR, FEBRUAR, . . . , DEZEMBER}
- Die Menge der Farben:
  {ROT, GRÜN, BLAU, . . . }

Motivation

- Rein vom Konzept her ist das wie gesagt nichts Neues:
- In unseren Modulen, in denen wir die Grunddatentypen und deren Operationen definiert haben, haben wir die Literale ja auch als konstante Funktionen spezifiziert.
- Das ist letztlich auch die Idee, wie man diese Wertemengen definiert.

Motivation

- Die einfachste Möglichkeit solche Wertemengen in einer Programmiersprache zu vereinbaren, wäre, die einzelnen Werte als (globale) Konstanten eines primitiven Typs (typischerweise **`int`** zu vereinbaren.
- Also in Java z.B. für Monat:

```
static final int JANUAR = 1;
static final int FEBRUAR = 2;
static final int MAERZ = 3;
static final int APRIL = 4;
...
```

Motivation

- In einigen Programmiersprachen muss man dies tatsächlich so machen und es ist zunächst auch nicht verkehrt (in Java bis Version 5.0 z.B.).
- Einziges Problem ist, dass man keine *Typsicherheit* mehr hat: Statt einen eigenen Typ „Monat“ zu haben, arbeitet man mit Werten vom Typ `int`.
- Dadurch kann zur Kompilierzeit nicht sicher gestellt werden, dass man z.B. an einen Methodenparameter einen der zulässigen Aufzählungswerte übergibt.

Motivation

- Eine Methode `anZahlTage(int monat, boolean schaltJahr)` nimmt eben nun leider als Eingabe für Monat einen Wert vom Typ **`int`**.
- Würde diese Methode mit `anZahlTage(30, false)` aufgerufen werden, wäre das für den Compiler OK.
- Den potentiellen Fehler würde man dann erst zur Laufzeit feststellen (oder auch nicht, was vermutlich noch schlimmer ist . . .)

Aufzählungstypen

- Für diese Werte-Mengen gibt es in einigen Programmiersprachen (wie gesagt, in Java seit Vers. 5.0) sog. *Aufzählungstypen* (*Enumeration*), die es erlauben, bei der Definition solcher Datentypen die Literale (Werte) explizit festzulegen (aufzuzählen).
- In Java ist dafür ein neues Schlüsselwort verfügbar: **`enum`**.
- Die einfachste Verwendung ist:

```
enum Monat ( JANUAR, FEBRUAR, MAERZ, APRIL, ... )
```

- Dies macht einen neuen Typ `Monat` dem Compiler bekannt, den man nun als Typ für (lokale/globale) Variablen, Eingabeparameter von Methoden und Instanzvariablen (Attribute) verwenden kann.

Aufzählungstypen

- Intern ist dieser Typ in Java als Klasse realisiert, d.h. die einzelnen Literale sind Objekte dieser Klasse.
- Nützliche Eigenschaften von Aufzählungstypen, die standardmäßig durch Java zur Verfügung gestellt werden:
  - Methode `toString`, die den Namen des Literals ausgibt.
    Beispiel: `Monat.JANUAR.toString()` ergibt `"JANUAR"`.
  - Methode `equals` prüft auf Gleichheit.
    Beispiel: `JANUAR.equals(APRIL)` ergibt **`false`**.
  - Variablen von Aufzählungstypen können in **`switch`**-Anweisungen verwendet werden (da es letztlich **`int`**s sind).
  - Methode `values` mit der man die einzelnen Werte der Enumeration durchlaufen kann (mittels eines `Iterator`-Objekts).
  - ...

Aufzählungstypen

- Letztlich steckt dahinter nicht viel Neues, außer, dass die Objekte von der Klasse `Monat` alle intern eine ID besitzen, die aus der Werte-Menge stammt.
- Enumerations können in Java tatsächlich fast genauso wie Klassen um Attribute, Methoden und Konstruktoren ergänzt werden.
- Ein Blick in ein einschlägiges Lehrbuch zu diesem Thema lohnt sich alle mal.
- Zur Veranschaulichung, was man alles u.a. machen kann, noch ein Beispiel:

Aufzählungstypen: Beispiel

```
public enum Farbe
{
  /* ** Die Literal-Namen mit Initialisierung der Attribute durch den Konstruktor ** */
  ROT(255, 0, 0),
  GRUEN(0, 255, 0),
  BLAU(0, 0, 255),
  GELB(255, 255, 0);

  /* ** Attribute, die die RGB-Werte speichern (Konstanten, da nicht veraenderbar) ** */
  private final int r;
  private final int g;
  private final int b;

  /* ** Konstruktor (immer nicht-statisch)** */
  public Farbe(int r, int g, int b)
  {
    this.r = r;
    this.g = g;
    this.b = b;
  }

  /* ** Methoden ** */

  public int getRotAnteil() { return this.r }
  ...
}
```

Überblick

Blick hinter die Kulissen

- Im Folgenden wollen wir kurz die interne Speicherverwaltung von Java (genauer gesagt der JVM) betrachten.
- Wir bleiben hier aber informell und vereinfachend und betrachten die Zusammenhänge nur, soweit wir sie benötigen, um Phänomene zu verstehen, die uns auf der Ebene der Programmierung beschäftigen können.
- Wir werden feststellen, dass diese Aspekte leider tatsächlich einige Effekte haben, die wir bei bei der Programmentwicklung berücksichtigt müssen.
- Ein tieferes Verständnis der hier behandelten Zusammenhänge kann in anderen Vorlesungen erworben werden.

Speicherumgebung

- Wir haben gesehen, dass Programme in einer imperativen Sprache wie Java durch Anweisungen charakterisiert sind, die Zustandsübergänge definieren.
- Zur Erinnerung: ein Zustand ist eine Menge von Paaren (intuitiv: Zetteln) (x, d), die Bindungen (Substitutionen) von Variablen (Bezeichnern) x an ein Literal/Element/Objekt d der Sorte von x spezifizieren (wobei die Zettel auch leer sein dürfen, in diesem Fall ist $d = \omega$).
- Diese Menge von Bindungen (Zettel) muss von einer *Speicherumgebung* entsprechend verwaltet werden.

Speicherumgebung

- Die Speicherumgebung von Java besteht wie erwähnt aus zwei Einheiten, dem *Stack* (*Keller*, *Stapel*) und dem *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).

Speicherumgebung

Bemerkung zur Terminologie:

- Der Stack modelliert die imperative Programmstrukturierung.
- Blöcke und ihre Schachtelung sind das wichtigste Strukturierungselement (Klassen, Methoden, Kontrollstrukturen etc. bilden stets einen Block).
- 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.

Speicherumgebung

Bemerkung zur Terminologie:

- Der Stack erfüllt diese Anforderung:
  1. Neue Bindungen werden oft eingefügt und existierende Bindungen werden oft wieder gelöscht.
  2. Die Bindungen, die zuletzt hinzukamen, werden als erste wieder entfernt (*Last-in-first-out*, *LIFO*).
- Stack heißt ja nicht um sonst „Stapel“ und am besten stapeln lassen sich gleichgroße Objekte (es gibt je einen Stapel für alle 1-Byte, 2-Byte, 4-Byte und 8-Byte Datentypen).

Keller für alle Variablen

- 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 des Stacks übersetzt.
- An dieser Speicheradresse steht deren Inhalt d.
- Das war bisher unser abstraktes Bild der Zettel.

Keller für alle Variablen

- 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*).
- Lokale Variablen unterschiedlicher Methoden(/Blöcke) sind innerhalb des Kellers in Bereiche (sog. *Frames*) angeordnet.
- Ein Frame wird bei einem Methodenaufruf erzeugt und beinhaltet u.a. die lokalen Variablen einer Methode, die übergebenen (aktuellen) Parameter und den Rückgabewert der Methode (wenn er existiert).

Keller vs. Halde

- Der Keller sieht für jeden Eintrag nur begrenzt viel Speicherplatz vor.
- Die Werte von primitiven Typen können direkt im Keller abgelegt werden (in der entsprechenden Speicherzelle).
- Objekte/Arrays sind i.d.R viel größer und teilweise ist deren Größe erst zur Laufzeit bei Erzeugung bekannt.
- Sie werden daher auf dem Heap gespeichert.
- Dazu steht im Keller in der Speicherzelle statt dem Wert (d.h. dem Objekt), die Speicheradresse des Objekts auf dem Heap.

Keller vs. Halde

- Also: für alle lokalen Variablen (Zettel $z = (x, d)$) eines Zustands ist im Keller der Wert d in einer Speicherzelle, deren Adresse sich aus x ergibt, abgelegt.
- Abhängig vom Typ der Variablen x (des Zettels z) gilt:
  - ist der Typ ein primitiver Typ, so ist d der tatsächliche Inhalt/Wert der Variablen, d.h. d repräsentiert das entspr. Literal (das Literal steht im Keller).
  - handelt es sich um einen Referenz-Typ (Array, Objekttyp, damit auch String), so ist der tatsächliche Inhalt/Wert auf der Halde gespeichert und d repräsentiert lediglich die *Referenz* (*Pointer*) auf die entspr. Adresse auf der Halde (die Referenz (=Adresse) steht im Keller, *nicht* das Literal/Objekt).
- So hatten wir unser Zustandsmodell informell erweitert.

Veranschaulichung: Keller vs. Halde

```
char a = 'b';
String gruss1 = "Hi";
String gruss2 = "Hello";
String[] gruesse = {gruss1, gruss2};
int[] zahlen = {1, 2, 3};
boolean b = true;
int i = 42;
```

Stack:

i = 42
b = **true**
zahlen = <adr4>
gruesse = <adr3>
gruss2 = <adr2>
gruss1 = <adr1>
a = 'b'

Heap:

<adr1> : "Hi" <adr2> : "Hallo" <adr3> : { <adr1> , <adr2> } <adr4> : {1, 2, 3 }

Der leere Zettel (revisited)

- Während primitive Typen lediglich deklariert werden, reicht dies bei Referenz-Typen nicht aus, sie müssen mit Hilfe des `new`-Operators oder — im Falle von Arrays und Strings — durch Zuweisung von Literalen zusätzlich noch explizit erzeugt werden.
- Wenn eine Variable angelegt (vereinbart) aber nicht initialisiert wird hatten wir bei primitiven Typen die Intuition eines leeren Zettels/einer leeren Speicherzelle, d.h. es wird „der *leere Wert* (ω) auf den Keller gelegt“.
- Solange dies der Fall ist, kann man einer anderen Variablen nicht den Wert einer leeren Speicherzelle zuweisen.

Der leere Zettel (revisited)

- Bei Referenztypen passiert im Prinzip das Gleiche: im Keller steht (noch) *keine* Speicheradresse, ein sog. **`null`**-Pointer, der in Java durch die Zeichenkette **`null`** repräsentiert wird.
- Eine nicht intitialisierte Variable kann nach wie vor nicht zugewiesen werden, aber die leere Referenz **`null`** kann man schon zuweisen:

```
Punkt p1 = null;

Punkt p2 = p1;
```

- **`null`** wird daher oft auch als Literal bezeichnet (und kann als solches z.B. in Ausdrücken verwendet werden):

```
if(a[0] != null) ...
```

Besonderheiten bei Referenztypen

- Das Verständnis für Referenztypen (Strings, Arrays, Objekte) ist entscheidend für die Programmierung in Java.
- Referenztypen können prinzipiell genauso benutzt werden wie primitive Typen, da sie jedoch lediglich eine Referenz darstellen, ist die Semantik einiger Operatoren anders als bei primitiven Typen!!!
- Ein Beispiel hatten wir schon kennengelernt: Array-Konstanten deren einzelne Array-Komponenten aber veränderbar sind.
- Drei weitere, sehr wichtige Beispiele, die wir im folgenden genauer betrachten:
  - Gleichheit von Objekten
  - Kopieren von Objekten
  - Call-by-reference Effekt bei Methodenaufruf mit Referenztypen

Gleichheit von Objekten

- *Gleichheit* von Objekten bedeutet, dass sie den selben Zustand haben, also alle Attribute haben die selben Werte.
- Beispiel:

```
Punkt p1 = new Punkt(1.0,1.0);
Punkt p2 = new Punkt(1.0,1.0);

boolean vergleich = (p1 == p2);
```

- Was ist der Wert der Variablen `vergleich`?
  Antwort: **`false`**!

Gleichheit von Objekten

- Warum? Sowohl das Objekt `p1` als auch das Objekt `p2` haben doch identische Attributwerte, also haben sie insgesamt denselben Zustand, oder nicht?
- Naja, was macht der Operator „==„?
- Intuitiv: für zwei Zettel $z_1 = (x_1, d_1)$ und $z_2 = (x_2, d_2)$ wird getestet, ob $d_1 = d_2$ gilt.
- Da `p1` und `p2` aber unterschiedliche initialisierte Zettel sind, referenzieren sie unterschiedliche Speicherbereiche auf der Halde (die Objekte dort sind zwar tatsächlich gleich, die Speicheradressen aber nicht!!!)

Gleichheit von Objekten

- Es gibt tatsächlich zwei Arten von Gleichheit bei Referenz-Typen:
  - *Gleichheit*: Der Zustand der entsprechenden Objekte beider Objektvariablen ist gleich, d.h. die Objekte auf der Halde sind identisch.
  - *Identität*: Beide Objektvariablen verweisen auf die gleiche Speicheradresse, d.h. der Wert im Keller ist identisch.
- Der Operator == prüft offenbar den zweiten Fall (Identität).
- Er kann also i.A. nicht dazu benutzt werden, abzufragen, ob die Objekte zweier Objektvariablen gleich bzgl. ihres Zustands sind.
- Dies wird oft übersehen und ist daher eine häufige Fehlerquelle!

Gleichheit von Objekten

- Um die Gleichheit zweier Objekte zu testen muss der Programmierer der entsprechenden Klasse also eine entsprechende Methode bereitstellen.
- In Java ist vorgesehen, dass diese Methode die Signatur `boolean equals(Object obj)` hat[19].
- Tatsächlich ist diese Methode bereits in der Klasse `java.lang.Object` implementiert, allerdings setzt diese zunächst nur die Identität um, d.h. testet auf die Gleichheit der Referenzen.

[19] Achtung: Argument ist tatsächlich vom Typ `java.lang.Objekt`!

Gleichheit von Objekten

- Die `java.lang.Objekt` ist implizit die Oberklasse aller Klassen und vererbt alle ihre Eigenschaften damit implizit auf alle anderen Klassen (siehe dazu auch „Vererbung“ in einem der folgenden Kapitel).
- Wie erwähnt: diese Klasse kennt die Details einer speziellen Klasse nicht und kann deshalb nur die generische Form (Identität) implementieren.
- Um die gewünschte Eigenschaft (Gleichheit) in einer eigenen Klasse zu bekommen, muss der Programmierer die Methode `equals` entsprechend überschreiben (auch dazu später mehr!)

Gleichheit von Objekten

- Die Methode `equals` in der Klasse `Object` ist so spezifiziert, dass sie eine Äquivalenz-Relation auf nicht-**`null`** Objektreferenzen mit speziellen Eigenschaften implementiert, siehe dazu die Dokumentation unter:

  `http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html`
- Diese Vorschrift *soll* eingehalten werden, wenn man in einer Klasse die Methode `equals` überschreibt; das ist aber eine *semantische* Vorschrift, die nicht vom Compiler überprüft, sondern nur vom Programmierer bewiesen werden kann.
- In `Object` ist `equals` durch den Operator `==` implementiert.

Gleichheit von Objekten

- Beispiel: Die Klasse `String` überschreibt `equals` so, dass für einen String `s` und ein Objekt `o` gilt: `s.equals(o)` ergibt **`true`** g.d.w.:
  - `o` ist nicht **`null`**
  - `o` ist vom Typ `String` und
  - `o` repräsentiert genau die gleiche Zeichenkette wie `s` (d.h. `s` und `o` haben gleich Länge und an jeder Stelle steht der gleiche Character)

Gleichheit von Objekten

- Beispiel für die Klasse `Punkt`

```
public boolean equals(Object obj) {
  if (obj instanceof Punkt) {
    Punkt pt = (Punkt) obj;
    return (this.x == pt.x) && (this.y == pt.y);
  } else {
    return false;
  }
```

- Der **`instanceof`**-Operator kann zur Laufzeit prüfen, ob ein von einem Verweis referenziertes Objekt zuweisungskompatibel zu einer Klasse ist.
- Man kann wie bei primitiven Typen auch bei Referenztypen einen expliziten Typcast durchführen (allerdings nur entlang einer Vererbungshierarchie).

Kopieren von Objekten

- Eine Zuweisung mit einem Referenztyp erzeugt eine *Kopie der Referenz* (und *kein neues Objekt*), man spricht auch von *Aliasing*.
- Beispiel:

```
Punkt p1 = new Punkt(1.0,1.0);
Punkt p2 = p1;  // (*)
p1.verschiebe(2.0,2.0);
p2.verschiebe(1.0,1.0);
```

Nach der Zuweisung `(*)` verweisen beide Variablen `p1` und `p2` auf dasselbe Objekt.

Kopieren von Objekten

- Warum? Auf dem Zettel (p_1, d_1), repräsentiert d_1 die Speicheradresse des Punktes `p1` auf dem Heap.
- Die Vereinbarung/Zuweisung `(*)` erzeugt einen zweiten Zettel mit Namen p_2 auf dem der Wert von p_1 geschrieben wird, also (p_2, d_1), d.h. nur der Verweis wurde kopiert, aber nicht das Objekt selber.
- Diese Kopie wird auch *flache* Kopie (*shallow copy*) genannt.
- Problem: es ist nicht sichergestellt, dass die Kopie unabhängig vom ursprünglichen Objekt ist!

Kopieren von Objekten

- Auch hier gibt es einen vorgesehenen Workaround in der Klasse `java.lang.Object`:
  Die Methode `Object clone()` erzeugt ein Objekt vom Typ `Object` als Kopie des aktuellen Objekts.
- In der ursprüngliche Fassung ist das allerdings wiederum die generellste Implementierung: eine flache Kopie.
- Die Methode `clone()` muss dazu entsprechend überschrieben werden.
- Dabei muss man darauf achten, dass die Objekte eines Attributs mit Objekttyp selbst wieder Attribute mit Objekttypen haben können!!!

Beispiel: Erstellen einer tiefen Kopie

```
public class DeepCopy
{
    private int zahl;
    private int[] zahlen;

    public DeepCopy(int zahl, int[] zahlen)
    {
        this.zahl = zahl;
        this.zahlen = zahlen;
    }

    public Object clone()
    {
        int neueZahl = this.zahl;
        int[] neueZahlen = new int[this.zahlen.length];
        for(int i=0; i<this.zahlen.length; i++)
        {
            neueZahlen[i] = this.zahlen[i];
        }
        DeepCopy kopie = new DeepCopy(neueZahl, neueZahlen);
        return kopie;
    }
}
```

Was ist hier nicht optimal gelöst?

Beispiel: Erstellen einer tiefen Kopie

```
public class DeepCopy implements Cloneable
{
    private int zahl;
    private int[] zahlen;

    public DeepCopy(int zahl, int[] zahlen)
    {
        this.zahl = zahl;
        System.arraycopy(zahlen, 0, this.zahlen, 0, zahlen.length);
    }

    public Object clone()
    {
        DeepCopy kopie = new DeepCopy(this.zahl, this.zahlen);
        return kopie;
    }
}
```

Den Zusatz „**`implements`** `Clonable`„ in der ersten Zeile bitte zunächst ignorieren!

Call-by-reference Effekt

- Zur Erinnerung: Java übergibt Parameter bei Methodenaufrufen mit call-by-value.
- In folgendem Beispiel hatte daher der Aufruf von `swap` keinen Einfluss auf `x` und `y` in `main`.

```
public class Exchange {
  public static void swap(int i, int j) {
    int c = i;
    i = j;
    j = c;
  }

  public static void main(String[] args) {
    int x = 1;
    int y = 2;
    swap(x,y);
  }
}
```

Call-by-reference-Effekt

- Bei Objekttypen gibt es dagegen einen unerwarteten Effekt:

```
1  public static void changeValues(int[] zahlen, int index, int wert)
2  {
3    zahlen[index] = wert;
4  }
5
6  public static void main(String[] args)
7  {
8    int[] werte = {0, 1, 2};
9    changeValues(werte, 1, 3);
10 }
```

`main`-Frame nach Zeile 6

args = <adr1>

Heap nach Zeile 6

<adr1> : { }

Call-by-reference-Effekt

- Bei Objekttypen gibt es dagegen einen unerwarteten Effekt:

```
1  public static void changeValues(int[] zahlen, int index, int wert)
2  {
3    zahlen[index] = wert;
4  }
5
6  public static void main(String[] args)
7  {
8    int[] werte = {0, 1, 2};
9    changeValues(werte, 1, 3);
10 }
```

`main`-Frame nach Zeile 8

werte = <adr2>
args = <adr1>

Heap nach Zeile 8

<adr1> : { } <adr2> : { 0, 1, 2 }

Call-by-reference-Effekt

- Bei Objekttypen gibt es dagegen einen unerwarteten Effekt:

```
1  public static void changeValues(int[] zahlen, int index, int wert)
2  {
3    zahlen[index] = wert;
4  }
5
6  public static void main(String[] args)
7  {
8    int[] werte = {0, 1, 2};
9    changeValues(werte, 1, 3);
10 }
```

Heap nach Zeile 8

<adr1> : { } <adr2> : { 0, 1, 2 }

main-Frame nach Zeile 8

werte = <adr2>
args = <adr1>

changeValues-Frame nach Zeile 1

wert = 3
index = 1
zahlen = <adr2>

Call-by-reference-Effekt

- Bei Objekttypen gibt es dagegen einen unerwarteten Effekt:

```
1  public static void changeValues(int[] zahlen, int index, int wert)
2  {
3    zahlen[index] = wert;
4  }
5
6  public static void main(String[] args)
7  {
8    int[] werte = {0, 1, 2};
9    changeValues(werte, 1, 3);
10 }
```

Heap nach Zeile 3

<adr1> : { } <adr2> : { 0, 3, 2 }

main-Frame nach Zeile 8

werte = <adr2>
args = <adr1>

changeValues-Frame nach Zeile 3

wert = 3
index = 1
zahlen = <adr2>

Call-by-reference-Effekt

- Bei Objekttypen gibt es dagegen einen unerwarteten Effekt:

```
1  public static void changeValues(int[] zahlen, int index, int wert)
2  {
3    zahlen[index] = wert;
4  }
5
6  public static void main(String[] args)
7  {
8    int[] werte = {0, 1, 2};
9    changeValues(werte, 1, 3);
10 }
```

`main`-Frame nach Zeile 9

werte = <adr2>
args = <adr1>

Heap nach Zeile 9

<adr1> : { } <adr2> : { 0, 3, 2 }

Call-by-reference Effekt

- Weiteres Beispiel:

```
public class Seiteneffekte {
  public static void swap(int i, int j) {
    int c = i;
    i = j;
    j = c;
  }

  public static void main(String[] args) {
    int x = 1;
    int y = 2;
    swap(x,y);
  }
}
```

Speicherfreigabe

- Anders als in C und C++, wo der *-Operator zur Dereferenzierung eines Zeigers nötig ist, erfolgt in Java der Zugriff auf Referenztypen in der gleichen Weise wie der auf primitive Typen.
- Einen expliziten Dereferenzierungsoperator gibt es in Java (i.Ggs. zu C/C++) nicht (die Methode `finalize` in der Klasse `Object` ist mit Vorsicht zu genießen).

Speicherfreigabe

- Anders als in C/C++ verfügt Java auch über ein automatisches Speichermanagement:
  - Die Keller werden schon von ihrer Struktur her automatisch aufgeräumt, d.h. es gibt in Kellern keine Speicherplätze, die belegt sind, obwohl die Lebensdauer des entsprechenden Namens abgelaufen ist.
  - Für die Halde gilt das nicht: Hier wird im Laufe eines Programmes Speicherplatz zugewiesen und belegt, aber nicht automatisch freigegeben, falls die Lebensdauer eines Namens, der für den gespeicherten Wert steht, abgelaufen ist.

Garbage Collection

- In vielen Sprachen muss der Programmierer dafür sorgen, dass Speicherplatz, der nicht mehr gebraucht wird, freigegeben wird.
- Diese explizite Speicherplatzfreigabe birgt allerdings die Gefahr des Speicherlecks:
- Es könnte passieren, dass Speicherzellen freigegeben werden, die noch benötigt werden, d.h. Werte von Variablen stehen dann nicht mehr zur Verfügung (oder sind vielleicht sogar schon mit anderen Werten überschrieben).

Garbage Collection

- Speziell in modernen Sprachen (auch Java) gibt es dagegen eine automatische Speicherplatzfreigabe (*Garbage Collection*):
  - Der Heap wird immer wieder durchsucht nach Adressen, auf die nicht mehr zugegriffen werden kann (da kein Name diese Adresse als Wert hat).
  - Problem hier: der Programmierer kann nicht oder nur sehr eingeschränkt kontrollieren, wann dieser Reinigungsprozess läuft.
  - Das ist unter Umständen (z.B. in sekundengenauen, empfindlichen Echtzeitsystemen) nicht akzeptabel.