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 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
- 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 = Stapel; und am besten stapeln lassen sich gleichgroße Objekte

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 x wird intern in eine Speicheradresse des Stacks übersetzt
- An dieser Speicheradresse steht der Inhalt d
- 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 *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

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 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)
- Der Name *Referenz-Typ* macht in diesem Zusammenhang nun auch sehr viel Sinn

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
- Bei Referenztypen passiert im Prinzip das Gleiche: im Keller steht (noch) *keine* Speicheradresse, ein sog. **`null`**-Pointer, der in Java durch die Zeichenkette **`null`** representiert wird
- **`null`** wird 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 (schon damals haben wir die Wirkung von Referenzen diskutiert)
- 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

```
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 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 stellt die Klasse `Object` die Methode **`boolean`** `equals(Object obj)` zur Verfügung
- Die Implementierung in der Klasse `Object` setzt zunächst die Identität um, d.h. testet auf die Gleichheit der Referenzen
- Da jede Klasse implizit Unterklasse von `Object` ist, steht diese Methode für alle Objekte in Java zur Verfügung
- Um die Gleichheit zu testen, muss diese Methode entsprechend überschrieben werden

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
- 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 gleiche Länge und an jeder Stelle steht der gleiche Character)

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
  - 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

- Die Methode `clone` in der Klasse `Object` erzeugt eine Kopie des aktuellen Objekts, in der ursprüngliche Fassung ist das allerdings eine flache Kopie
- Die Java-API stellt das Interface `Cloneable` zur Verfügung, das für die Methode `clone` (die jede Klasse von `Object` erbt) eine spezielle Eigenschaft spezifiziert
- Implementiert eine Klasse das Interface `Cloneable`, so garantiert der Implementierer laut Spezifikation, dass die Methode `clone` eine sog. *tiefe Kopie* (*deep copy*) erzeugt: Für alle Attribute mit Objekttypen müssen Kopien der entsprechenden Objekte angelegt werden
- Die Methode `clone` muss dazu entsprechend überschrieben werden
- Achtung: Beim Erstellen einer tiefen Kopie muss man darauf achten, dass die Objekte eines Attributs mit Objekttyp selbst wieder Attribute mit Objekttypen haben können

EXKURS: Marker-Intertfaces

- Das Interface `Cloneable` spezifiziert keine Methode (und auch keine Konstanten)
- Solche Interfaces, die weder Methoden noch Konstanten definieren, werden *Marker-Interfaces* genannt
- Marker-Interfaces sind dazu gedacht, gewisse (teilweise abstrakte) Eigenschaften von Objekten sicher zu stellen, die typischerweise im Kommentar des Interfaces spezifiziert sind
- Implementiert eine Klasse ein Marker-Interface, sollte sich der Implementierer an diese Spezifikationen halten
- Dies wird aber wiederum nirgends automatisch (z.B. vom Compiler) abgeprüft, d.h. es liegt alleine in der Verantwortung des Programierers, diese Eigenschaften zu garantieren

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;
        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;
    }
}
```

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 }

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)
- 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 (explizite Speicherplatzfreigabe – Gefahr des Speicherlecks)
- In vielen 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