Kapitel 12.
Referenzen

Kapitel 12: Referenzen

Ziel: Die Wahrheit über Objekte

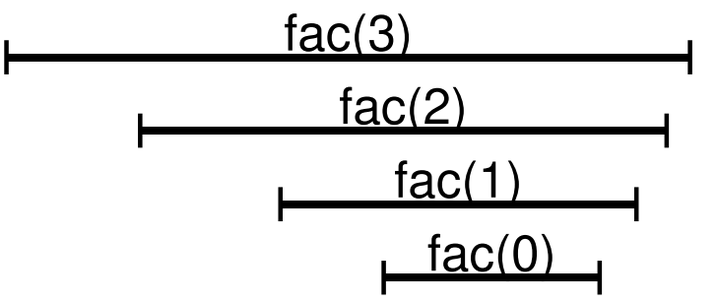
- Lebensdauer
- Speicherverwaltung
- Parameterübergabemechanismen in Methoden
- Gleichheiten, Kopien
- Arrays
- Speicherbereinigung

Lebensdauer

- dynamisches Konzept: Lebensdauer einer *Inkarnation*
- Beispiele: Lebensdauer
  - eines Programms: seine Laufzeit vom Start bis Ende
  - eines Objekts: vom Generieren mit new bis zum (automatischen) Löschen
  - einer Methode: vom Aufruf bis zur Rückkehr

  Beispiel:

```
int fac(int n) {
    if(n==0) {return 1;}
    else {return n*fac(n-1);}
}
```



Lebensdauer von lokalen Variablen vs. Objekten

```
Point foo() {
    Point p = new Point(1,1);
    Point q = new Point(2,2);
    return q;
}
```

- Point (1,1) lebt bis zum Methodenende
  - einzige Variable p, in die er gespeichert wird, lebt nur bis zum Methodenende
- Point (2,2) lebt darüber hinaus
  - die Variable q, in die er gespeichert wird, lebt zwar nur bis zum Methodenende, aber
  - da er als Ergebnis zurückgegeben wird, muss das Objekt auch nach dem Methodenende noch erhalten bleiben / leben

Mehr Wahrheit über den Speicher

- Der Speicher ist (mindestens) in zwei Teile aufgeteilt:
  - den Stack und
  - den Heap
- lokale Variablen und Parameter liegen am Stack
- Objekte liegen auf dem Heap
- Folge: die Vorstellung, dass durch
  `Point p = new Point(1,1);`
  ein Objekt kreiert und auf dem Speicherplatz p abgelegt wird, ist ***falsch***!

Mehr Wahrheit über den Speicher

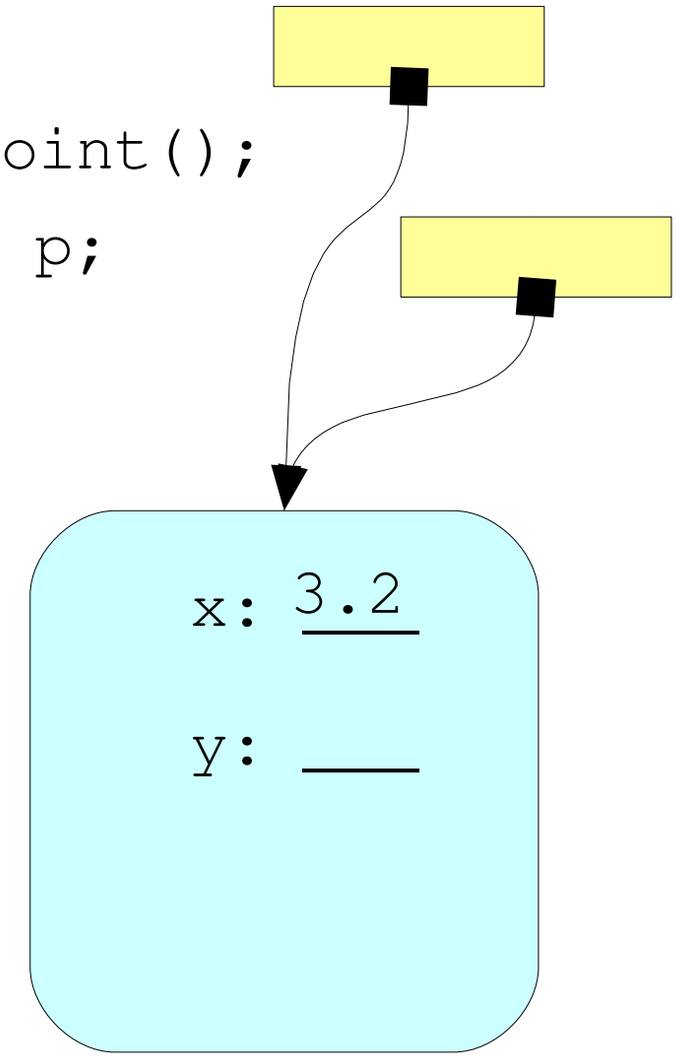
- Richtig ist, dass das Objekt kreiert und irgendwo an einer Speicherstelle ab Adresse *x* im Heap gespeichert wird.
- In der lokalen Variablen `p` steht dagegen nur die Adresse *x*. Dort steht also nur ein Zeiger, Pointer, Verweis oder – in Java-Sprache – eine ***Referenz*** auf das Point(1,1)-Objekt
- Unterschied zischen „guten“ Java-Referenzen und „bösen“ C/C++-Pointern: mit letzteren kann man rechnen um sie zu manipulieren, wogegen man in Java nur immer Referenzen auf bereits kreierte Objekte setzen kann.
- Vorteil: Sicherheit und Typisierung bei Java-Referenzen

Beurteilung von Referenzen

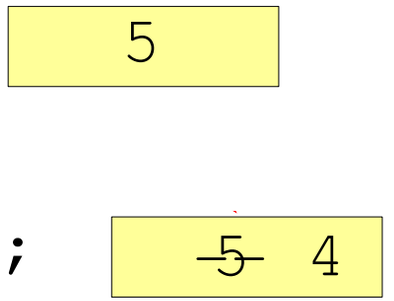
- Positiv:
  - speicher- und laufzeitsparend, weil nur die Referenzen und nicht die „großen Objekte“ beim Programmablauf weitergegeben werden
  - keine Kopien, die Speicher kosten und veralten
  - Zugriff auf dasselbe Objekt einfach von mehreren Stellen aus möglich (je nach Anwendung nötig)
- Negativ:
  - Zugriff auf dasselbe Objekt einfach von mehreren Stellen aus möglich (Sicherheit)
  - ein Indirektionsschritt beim Zugriff nötig

Funktionsweise von Referenzen

```
Point p;
p = new Point();
Point q = p;
q.x = 3.2
```

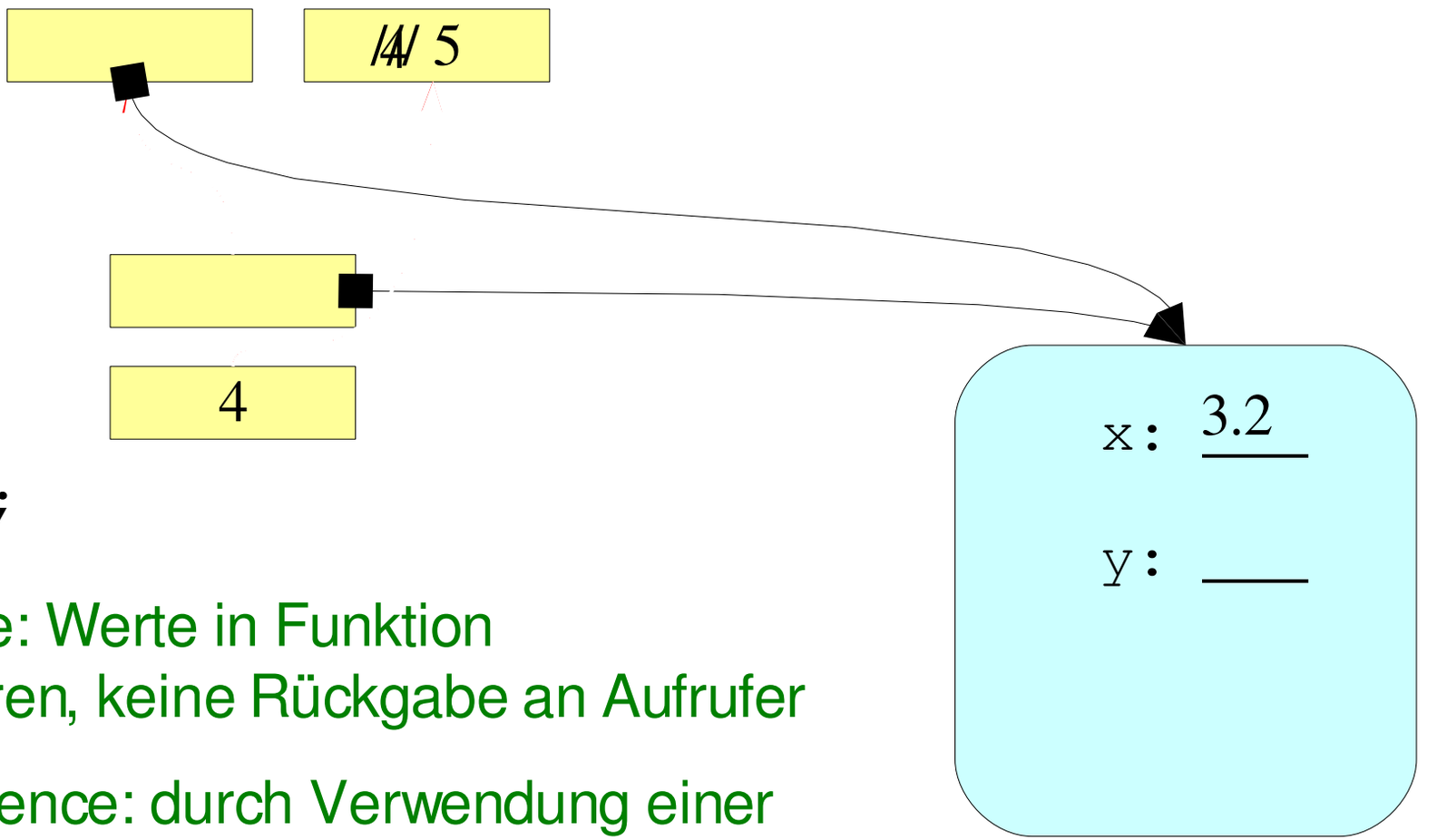



```
int n;
n = 5;
int k = n;
k = 4;
```



Referenzen und Methodenaufrufe

```
void foo(Point s, int i) { s.x=3.2; i=5;  }
```



call-by-value: Werte in Funktion hineinkopieren, keine Rückgabe an Aufrufer

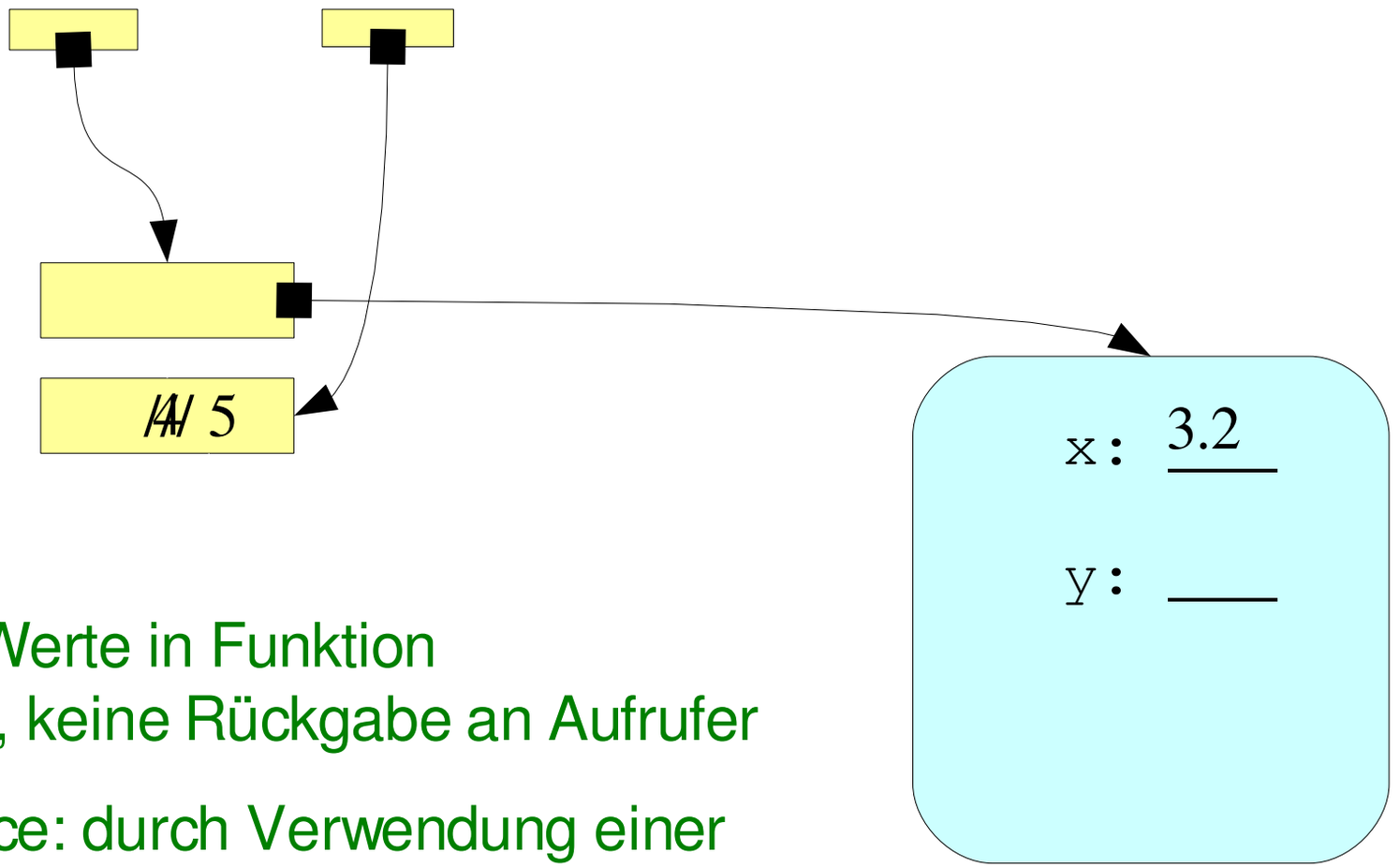
call-by-reference: durch Verwendung einer Referenz: Änderungen beim Aufrufer möglich

Referenzen und Methodenaufrufe

- **Java macht immer call-by-value.**
- Da Objekte aber immer Referenzen sind, *scheint* Java call-by-reference für die Objekte zu machen.
- Würde Java call-by-reference machen, könnte man auch ein komplett neues Objekt zurückgeben, **aber:** Änderungen des Wertes von `s` wirken sich in Java *nicht* beim Aufrufer aus => Java hat nicht CbR.
- Nebenbei: auch bei `foo(final Point s,...)` wäre der Zugriff im Beispiel erlaubt; nur eine (ohnehin nur lokal sichtbare) Änderung von `s` selber ist damit verboten

Nebenbei: call by reference

```
void foo(Point s, int i) { s.x=3.2; i=5;  }
```



call-by-value: Werte in Funktion hineinkopieren, keine Rückgabe an Aufrufer

call-by-reference: durch Verwendung einer Referenz: Änderungen beim Aufrufer möglich

this

- this ist eine Referenz auf sich selbst, also das eigene Objekt, die eigene Speicheradresse
- Es ermöglicht das Zurückgeben von z.B. einem neuen Objekt in den einen Fällen oder von dem alten (this) in anderen Fällen.
- Es ermöglicht Abfragen, ob ein Objekt o == this ist
- Damit klar: die Gleichheit auf Objekten ist keine semantische Gleichheit (womöglich in Abhängigkeit von den Eigenschaften der gespeicherten Attribute), sondern die reine Objektidendentität (Speicheradressengleichheit).

null

- `null` ist eine ausgezeichnete Referenz, die auf **kein** Objekt verweist.
- `null` ist der Defaultwert für automatisch initialisierte Pointer.
- Wenn `null` (aus Versehen) dereferenziert wird, liefert das Programm zur Laufzeit eine Fehlermeldung, eine sog. NullPointerException.
- Manchmal setzt man eine Referenz-Variable `p` absichtlich auf `null`, um damit klarzumachen, dass diese Referenz `p` auf das Objekt nicht mehr benötigt wird (siehe Speicherbereinigung).

Gleichheiten/Äquivalenzen

- Um zwei Objekte als semantisch gleich (besser äquivalent) zu definieren, kann man die vordefinierte Methode `equals` überschreiben.
- Typ: `public boolean equals(Object o)` kann *genau mit diesem Typ überschrieben* werden.
- Dabei muss im Rumpf natürlich `o` in den eigentlichen Typen gecastet werden, um die nötigen Attributzugriffe zu ermöglichen.
- Beachte: Definition als `public boolean equals(Point p)` liefert eine neue, überladene Methode.

Kopien

- Kopien kann man nicht einfach durch `q=p` realisieren.
- Es muss ein neues Objekt kreiert und mit denselben Attributen belegt werden.
- In Java geschieht das mit der vordefinierten Methode `clone()`, die ein Objekt liefert, das eine Bitkopie des Originals ist.
- Dazu muss man aber für die zu klonende Klasse angeben ... `implements Cloneable` ...
- Anwendung: `Point r = (Point)(p.clone());`
- Warnung: clone ist keine „deep copy“ (Objektreferenzen als Attribute ⇨ nur Referenzkopien!)

Arrays

- Arrays sind Objekte und Array-Variablen damit nur Referenzen.
- Folge: nach

```
String[] a = { “a0“, “a1“, “a2“, “a3“ };
String[] b = a;
b[0] = “b0“;
```

ist `a[0]` mit “`b0`“ belegt.

- Jedes mehrdimensionale Array ist ein eindimensionales Array (der äußersten Dimension), dessen Einträge die Referenzen auf die (evtl. mehrdimensionalen) Arrays der zweitäußersten Dimension sind.
- Folge: „flexible Längen“ und „Querverweise“

Garbage Collection (Speicherbereinigung)

- Definition: die Beseitigung nicht mehr benötigter Objekte aus dem Speicher
- Wenn die letzte Referenz auf ein Objekt gelöscht wird (durch Zuweisung auf ein anderes Objekt oder auf `null`, oder durch Ende des Gültigkeitsbereichs der lokalen Variablen), kann auf das Objekt nicht mehr zugegriffen werden.
- Im Gegensatz zu C, C++ oder Pascal macht der Compiler die Buchführung (Korrektheitsgründe!), d.h., die Freigabe des Speichers geschieht in Java automatisch.

Zusammenfassung

- Lebensdauern
  - von Programmen, Methoden, Objekten, Variablen
- Speicherverwaltung
  - Heap/Stack, Referenzen
- Parameterübergabemechanismen
  - call-by-value, call-by-reference
- Gleichheiten, Kopien
  - equals, clone
- Arrays
  - auch mehrdimensional
- Garbage Collection