Programmieren I

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2009/2010

Objekte und Referenzen

Garbage Collector

Destruktor/Finalisierung

Rückblick

In der letzten Vorlesung wurde eine Klasse `Buch` eingeführt, die u. a. ein Attribut `ausleiher` besitzt, auf das über öffentliche Methoden zugegriffen werden kann:

```
public class Buch {
    private String ausleiher;

    public String getAusleiher() {
        return ausleiher;
    }
    public void ausleihe(String ausleiher) {
        this.ausleiher = ausleiher;
    }
}
```

Anwendung

Weiterhin werden in einer Instanz der Klasse `Bibliothek` Bücher verwaltet, auf die dann über einen Index zugegriffen werden kann, z. B.

```
Bibliothek bib = new Bibliothek();

// ...

Buch b1 = bib.getBuch(0);
Buch b2 = bib.getBuch(0); // Zeitpunkt 1

b1.ausleihe("Karl Meier"); // Zeitpunkt 2
b2.ausleihe("Erika Schmidt"); // Zeitpunkt 3
```

Fragen

Fragen:

- Zu welchem Zeitpunkt ist welches Buch von wem ausgeliehen?
- Wie viele Objekte vom Typ „Buch“ gibt es eigentlich?

Antwort:

	Ausleiher `b1`	Ausleiher `b2`
Zeitpunkt 1	–	–
Zeitpunkt 2	Karl Meier	Karl Meier
Zeitpunkt 3	Erika Schmidt	Erika Schmidt

`b1` und `b2` sind nur verschiedene Namen für die einzige Instanz von `Buch`!

Weiteres Beispiel

Im Praktikum wurde ein Klasse `Konto` definiert, die u. a. Überweisungen auf andere Konten enthält:

```
public class Konto {
    private double saldo;
    public void ueberweisung(Konto empfaenger, double betrag) {
        if (betrag<=saldo) {
            saldo -= betrag;
            empfaenger.saldo += betrag;
        }
    }
    public static void main(String[] args) {
        Konto k1 = new Konto("Karl Meier");
        Konto k2 = new Konto("Erika Schmidt");
        // ...
        k1.ueberweisung(k2, 100);
    }
}
```

Ablauf

- In der Methode `main` wird `ueberweisung` für `k1` mit `k2` als Parameter aufgerufen.
- Innerhalb der Methode `ueberweisung` ist der Parameter als lokale Variable `empfaenger` bekannt.
- Für `empfaenger` wird dann der Saldo geändert, indem der Überweisungsbetrag hinzu addiert wird.
- Anschließend in `main` hat trotzdem `k2` einen höheren Kontostand.
- Offensichtlich sind `empfaenger` und `k2` also verschiedene Namen für dieselbe Instanz von `Konto`.

Objekte und Referenzen

In Java werden Variablen, die sich auf die elementaren Datentypen beziehen (z. B. `int`, . . .), durch einen festgelegten Speicherbereich repräsentiert, in dem der Wert der Variablen abgelegt wird (ein `int` z. B. durch 4 Byte).

Variablen vom Typ `Objekt` (und indirekt aller Klassen der Systembibliothek und aller selbst geschriebenen Klassen) enthalten jedoch nur eine *Objektreferenz*, also einen *Verweis* auf ein Objekt.

Hierbei benutzt die JVM zwei Speicherbereiche:

- den Stack und
- den Heap

Stack und Heap

Stack und Heap sind beides dynamische Speicherbereiche, in denen zur Laufzeit Speicher belegt und wieder freigegeben werden kann.

Allerdings gibt es einen wesentlichen Unterschied:

- Die Reihenfolge, in der im Stack Speicher angefordert und wieder freigegeben wird, ist festgelegt. Der zuletzt angeforderte Speicher muss zuerst wieder freigegeben werden. Im Stack befinden sich lokale Variablen und Parameter von Methoden, entweder als elementare Datenwerte oder Objektreferenzen.
- Der Heap-Speicher kann zur Laufzeit frei reserviert und wieder frei gegeben werden, ohne dabei eine vorgegebene Reihenfolge einhalten zu müssen. Hier werden zur Laufzeit dynamisch Instanzen von Klassen (also Objekte) samt ihrer Attribute angelegt und zerstört.

Analyse des Bibliothek-Beispiels

Im ersten Beispiel wurden (in einer Methode `main`) eine Instanz der Klasse `Bibliothek` angelegt und dieser ein oder mehrere Bücher hinzugefügt.

- Im Heap existieren eine Instanz von Bibliothek (u. a. mit einem Feld von Referenzen auf Buch als Attribut) und ein oder mehrere Buch-Instanzen (samt deren Attribute).
- Im Stack existieren als lokale Variablen die Referenz auf `Bibliothek` mit dem Namen `bib` und zwei Referenzen auf `Buch` mit den Namen `b1` und `b2`.

Analyse des Bibliothek-Beispiels (Forts.)

Es existieren im Beispiel also zwei lokale Variablen vom Typ `Buch`, da diese aber Objektreferenzen sind, können sie auf ein und dasselbe Objekt verweisen.

Insgesamt existieren im Beispiel drei Objektreferenzen auf das Buch mit dem Index 0:

- als `bib.buecher[0]`, also als Element mit dem Index 0 im Feld `buecher` (Attribut des Objekts, auf das `bib` verweist),
- als `b1` und
- als `b2`.

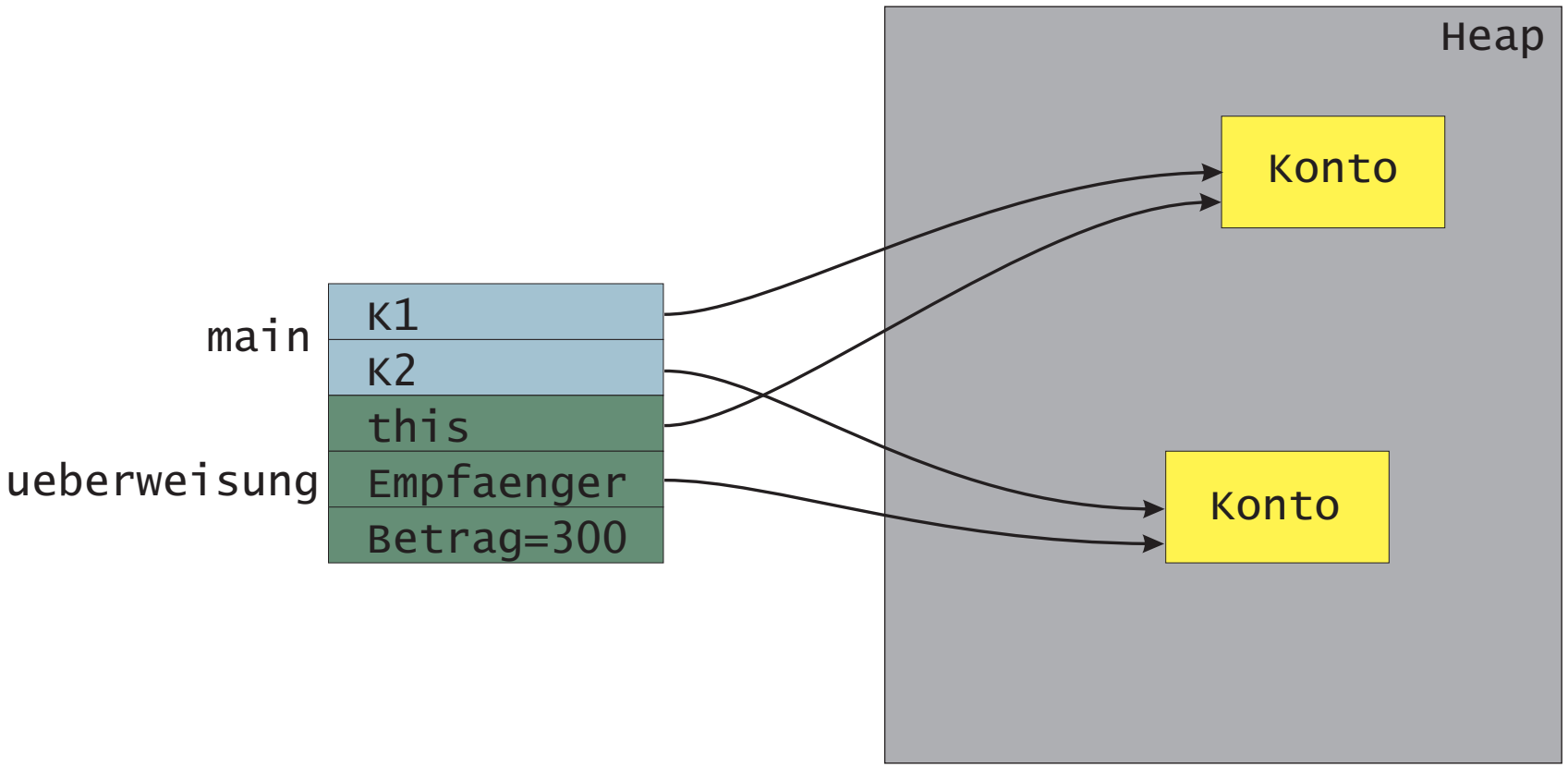
Also drei Referenzen, aber nur ein einziges Objekt!

Analyse des Konto-Beispiels

Im Beispiel mit der Überweisung liegt folgender Ablauf vor:

- Im Heap werden zwei `Konto`-Objekte angelegt.
- Im Stack befinden sich für `main` zwei Referenzen `k1` und `k2`, die auf diese Objekte verweisen.
- Beim Aufruf von `k1.ueberweisung(k2,100)` werden auf dem Stack für die Methode `ueberweisung` angelegt:
  1. die Referenz `this`, die auf dasselbe Objekt wie `k1` verweist.
  2. die Referenz `empfaenger`, die auf dasselbe Objekt wie `k2` verweist.
  3. die Variable `betrag` mit dem Wert 100.
- Nach dem Ende von `ueberweisung` werden diese drei Variablen wieder vom Stack entfernt.
- Wegen des Referenz-Modells ist der neue Kontostand auch für `k2` sichtbar.

Schaubild


Heap
Konto
Konto
main
K1
K2
this
ueberweisung
Empfaenger
Betrag=300

Call by Value/Call by Reference

In Programmiersprachen wird zwischen zwei Methoden zur Parameterübergabe an Methoden und Funktionen unterschieden:

Call by Value: Hier wird der Parameter beim Aufruf ausgewertet und *als Kopie* übergeben.

Call by Reference: Hier wird im Code der Methode/Funktion der Parameter als *Alias* auf die Variable im aufrufenden Code angelegt. Alle Aktionen, die mit dem Parameter in der Methode/Funktion durchgeführt werden, passieren also tatsächlich mit der Variablen, die beim Aufruf angegeben wurde.

Sonderfall Java

Vereinfacht ausgedrückt gilt bei Java

- für elementare Datentypen *Call by Value* und
- für Objekte *Call by Reference*

Diese Sichtweise ist jedoch nicht ganz korrekt, denn formal existiert in Java nur *Call by Value*. Selbst im zweiten der beiden o. a. Fälle wird eine Kopie übergeben, nämlich eine *Kopie der Objektreferenz*.

Beispiel zur Verdeutlichung: Wenn man im Code der Methode eine als Parameter übergebenen Objektreferenz per Zuweisung auf ein anderes Objekt verweisen lässt, beeinflusst dies nicht die Objektreferenz, die beim Aufruf angegeben wurde.

Zerstören von Objekten

Im Gegensatz zu anderen Programmiersprachen werden Objekte in Java nicht explizit zerstört.

In Java zählt die Virtual Machine die Referenzen auf die Objekte mit. Sobald es keine Referenz mehr auf ein Objekt gibt, *kann* es zerstört werden, da es im Programm nicht mehr verwendbar ist.

Hierfür gibt es in der JVM einen Mechanismus, der automatisch solche als unbenutzt identifizierte Objekte zerstört und ihrem Speicherplatz wieder freigibt: den *Garbage Collector*.

Dies ist eine der Stärken von Java gegenüber anderen Sprachen, wo der Programmierer für die Zerstörung dynamisch angelegter Objekte selbst zuständig ist, und bei nachlässiger Programmierung Speicherlecks entstehen können.

Destruktor/`finalize`

Die meisten objektorientierten Programmiersprachen kennen als Gegenstück zum Konstruktor den *Destruktor*, der ausgeführt wird, wenn ein Objekt zerstört wird. Hier können abschließende Arbeiten wie z. B. Trennen von Netzwerk- oder Datenbankverbindungen erledigt werden.

Da es in Java dem Garbage Collector überlassen bleibt, wann Objekte zerstört werden, kennt Java stattdessen das Konzept der *Finalisierung* (analoges gilt für C#/.NET). Eigener Finalisierungs-Code kann dabei in Form einer Methode

```
protected void finalize()
```

definiert werden. Allerdings muss der Programmierer berücksichtigen, dass er wenig bis keinen Einfluss auf Zeitpunkt und Reihenfolge der Finalisierung hat.

Beispiel

Zur Illustration hier eine Klasse, die in einem statischen Attribut die Zahl der Instanzen der Klasse mitzählen soll:

```
public class Test {
    private static int count = 0;

    public static int getCount() {
        return count;
    }

    public Test() {
        count++;
    }

    protected void finalize() {
        count--;
    }
}
```

```
public static void main(String[] args) throws Exception {
    Test t1 = new Test();
    Test t2 = new Test();

    // Wieviele Instanzen existieren?
    System.out.println(Test.getCount());

    t1 = t2 = null;

    // Wieviele Instanzen existieren jetzt?
    System.out.println(Test.getCount());
    Thread.sleep(5000);
    // Und nun?
    System.out.println(Test.getCount());

    System.gc(); // Garbage Collection erzwingen
    // Wieviele Instanzen existieren jetzt?
    System.out.println(Test.getCount());
}
```

Analyse

Im vorigen Beispiel werden zwei Instanzen angelegt. Somit hat das statische Attribut `count`, das vom Standardkonstruktor jeweils hoch gezählt wird, den Wert 2.

Wenn den beiden Variablen `t1` und `t2` der Null-Pointer zugewiesen wird, sind die beiden angelegten Instanzen unbenutzt und *können* bei der Garbage Collection zerstört werden.

Im Allgemeinen arbeitet der Garbage Collector nur bei Bedarf, daher wird meistens der Zähler `count` auf dem Wert 2 bleiben, bis durch `System.gc()` die Garbage Collection erzwungen wird.

Anschließend hat `count` den Wert 0, da bei der Zähler bei der Finalisierung jeweils herunter gezählt wurde.