Technische Hochschule Ingolstadt
Fakultät Informatik

Programmierung 2 (KI-BA)
Programmieren in Java II (WINF-BA)

Prof. Dr. Hans-Michael Windisch

Inhalt der Vorlesung

1. Klassen und Objekte
2. Vererbung
3. Enums, Wrapper und Autoboxing
4. Interfaces
5. Generics
6. Exceptions
7. Polymorphismus
8. Grafische Oberflächen
9. Streams und Lambda Expressions
10. Leichtgewichtige Prozesse – Threads

Kapitel 1: Klassen und Objekte

Inhalt

1.1 Das Objektorientierte Programmiermodell
1.2 Definition einer Klasse
1.3 Objekterzeugung
1.4 Überladen von Methodennamen
1.5 Referenzen und „this“
1.6 Parameterübergabe
1.7 Objektauflösung
1.8 Sichtbarkeitsstufen
1.9 Arrays
1.10 Strings
1.11 Überprüfung auf Gleichheit
1.12 Statische Elemente
1.13 Getter- und Setter-Methoden
1.14 Pakete

Lernziele

[LZ 1.1] Das Objektorientierte Programmiermodell erklären können.
[LZ 1.2] Einfache UML-Klassenmodelle verstehen können
[LZ 1.3] Java-Klassen definieren und nutzen können
[LZ 1.4] Möglichkeiten der Objekterzeugung kennen und anwenden können
[LZ 1.5] Wissen, was Methodenüberladung bedeutet und wann Sie sinnvoll ist
[LZ 1.6] Wissen, was Referenzen sind und wie sie verwendet werden
[LZ 1.7] Die this-Referenz kennen und ihre Einsatzmöglichkeiten beherrschen
[LZ 1.8] Den Java-Parameterübergabemechanismus kennen und einsetzen können
[LZ 1.9] Die Funktionsweise des Garbage Collectors in Java erläutern können
[LZ 1.10] Die Sichtbarkeitsstufen kennen und deren Anwendung beherrschen
[LZ 1.11] Die Array-Syntax kennen und Java-Arrays einsetzen können
[LZ 1.12] Die Java-String API kennen und diese in Java-Programmen einsetzen können
[LZ 1.13] Wissen, wie Java-Objekte inhaltlich verglichen werden können
[LZ 1.14] Motivation und Syntax statischer Elemente kennen und anwenden
[LZ 1.15] Die Konvention für Getter und Setter kennen und diese anwenden können
[LZ 1.16] Das Java package-Konzept kennen und es anwenden können

Grundideen der Objektorientierung

Ein **Programm** besteht aus Objekten, die gemeinsam die Funktionalität des Programms erbringen.

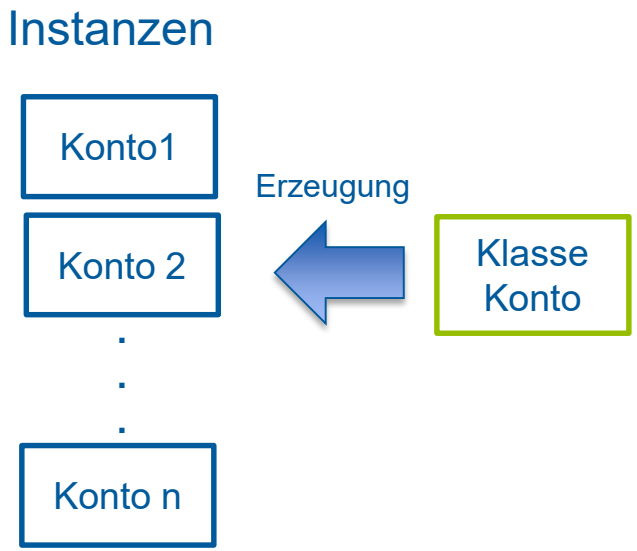
Ein **Objekt** besteht aus **Attributen** (=Variablen mit aktuellen Werten) und bietet für andere Objekte Dienstleistungen, **Methoden** genannt, an. Die Methoden können über Aufrufe von anderen Objekten genutzt werden. Jedes Objekt hat eine eindeutige ID, welche für die Nutzung von außen benötigt wird.

Jedes Objekt folgt einem „Bauplan“, dieser wird in einer sogenannten **Klasse** vorgegeben. Die Klasse ist ein Datentyp (kurz: Typ) und legt Attribute und Methoden aller Objekte der Klasse fest. Jedes Objekt ist „Instanz“ einer Klasse.

Die Objekte zu einer Klasse unterscheiden sich durch ihren **Zustand** (=Werte der Attribute).

Beispiel: Girokonto in Java

Girokonten sind Konto-Objekte, die Konto-Klasse ist der Datentyp zur Erzeugung der Konto-Objekte.



Beispiel: Girokonto in Java
Konkret sieht die Konto-Klasse in Java mit Attributen und Methoden so aus:
(hier nur zur Verdeutlichung der Grundideen)

Datentyp Konto

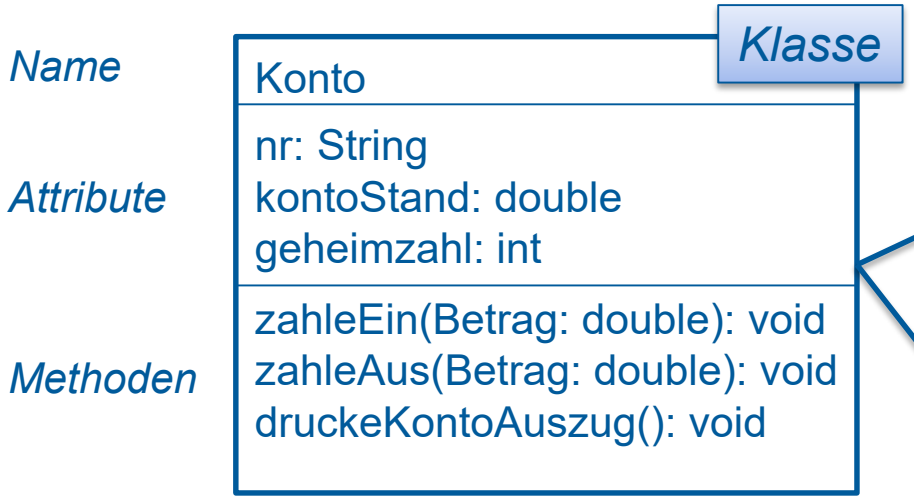
```
class Konto {
    String nr;
    double kontoStand;
    int geheimzahl;

    void zahleEin(double betrag) {
        kontoStand += betrag;
    }
    void zahleAus(double betrag) {
        kontoStand -= betrag;
    }
    void druckeKontoAuszug() {
        System.out.println("Konto " + nr + ": Kontostand = " + kontoStand);
    }
}
```

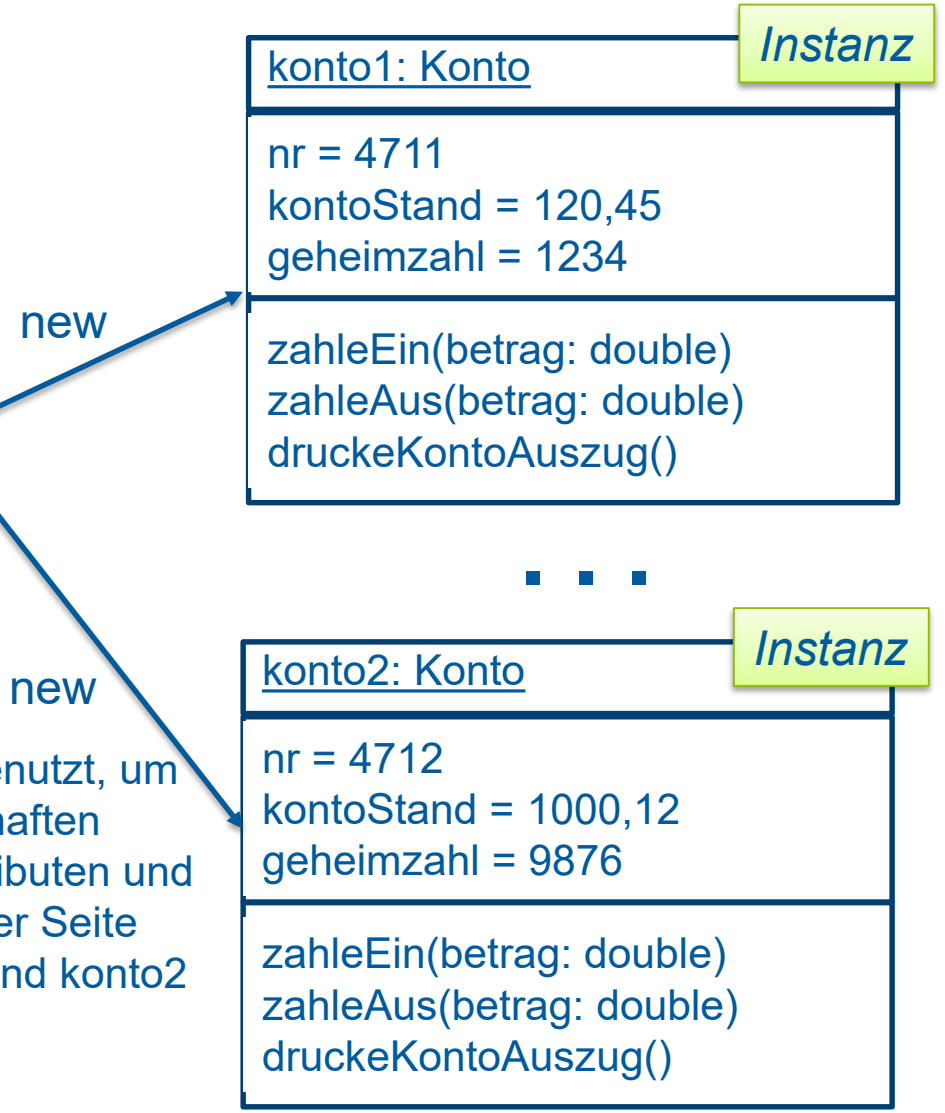
Attribute

Operationen

UML-Darstellung der Klasse

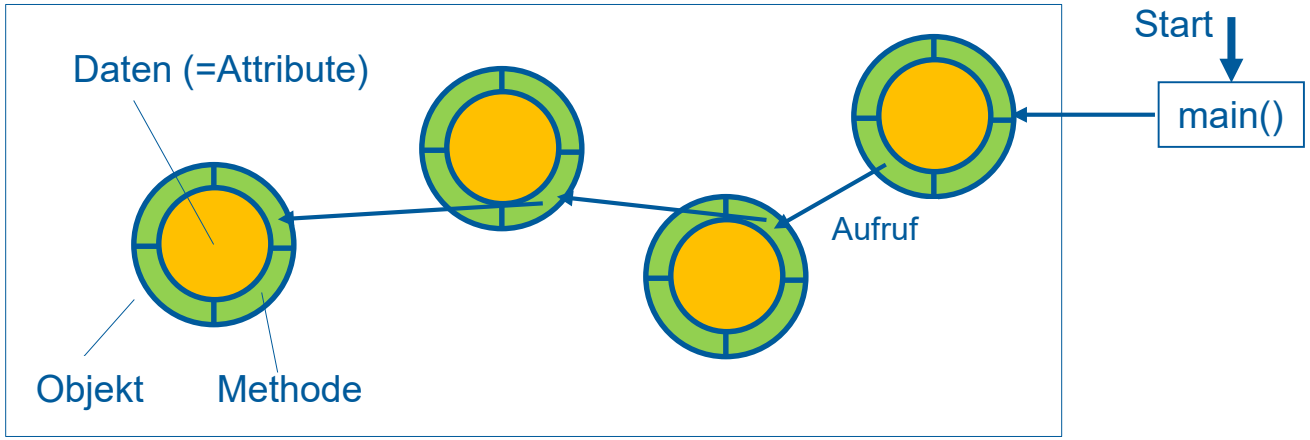


UML-Darstellung der Instanzen



Die Unified Modelling Language (UML) wird häufig genutzt, um Klassen und Objekte mit ihren wesentlichen Eigenschaften darzustellen. Oben ist die Klasse „Konto“ mit den Attributen und Methoden dargestellt (vgl. die Java-Umsetzung auf der Seite davor!). Rechts erkennt man zwei Instanzen konto1 und konto2 mit ihren unterschiedlichen Attributwerten.

Grundprinzip der Ausführung eines OO-Programms (OO = Objektorientierung)
Jedes objektorientiert programmierte Programm beginn mit einer main-Funktion, die ohne ein dazugehöriges Objekt aufgerufen wird (mehr dazu später). In main() werden typischerweise Objekte erzeugt und deren Methoden werden aufgerufen, was weitere Objekterzeugungen und Methodenaufrufe nach sich zieht, um die Funktionalität des Programms zu erbringen.



In Java (Erläuterung später):

```
class Konto {
    …
   static public void main(String[] args) {
      Konto k = new Konto(100.0); // Konto mit Kontostand erzeugen
      k.zahleAus(50.0);                  // 50 EURO auszahlen
   }
}
```

Eine **Klasse** ist ein benutzerdefinierter Datentyp und legt Eigenschaften fest, die allen Instanzen (=Objekten) der Klasse gemeinsam sind. Dies sind Attribute und Methoden. Jede Klasse hat in ihrem *package* (vergleichbar mit einem Ordner im Dateisystem, s. Abschnitt 1.14) einen eindeutigen Namen. Die **Reihenfolge** von Klassen-, Methoden- und Attributdefinitionen ist in Java beliebig!

Attribute legen die Variablen fest, die ein Objekt besitzt. Alle Methoden können die Attributwerte lesen und verändern. Von außerhalb des Objekts sollten Attribute i.allg. nicht zugreifbar sein, der Zugriff auf Attribute ist durch Methoden zu gewährleisten. Innerhalb einer Klasse sind Attribute über ihren Namen nutzbar.
Beispiel: Attribut „seitenlaenge“ der Quadrat-Klasse (s. nächste Seite)

Methoden bieten „Dienstleistungen“ für andere Objekte an und legen das Verhalten von Objekten einer Klasse fest. Eine Methode wird in Java von außen – d.h. durch Methoden anderer Objekte – mithilfe des „.“-Operators aufgerufen. Methoden sind analog zu Funktionen parametrisierbar und besitzen einen Ergebnistyp (inkl. „void“). Innerhalb einer Klasse werden Methoden anhand ihres Namens aufgerufen.
Beispiel: Methode „berechneFlaeche()“ der Quadrat-Klasse (s. nächste Seite)

Beispiel: Definition der Klasse Quadrat

```
public class Quadrat {
        double seitenLaenge;

        Quadrat(double pSeitenlaenge) {
                seitenLaenge = pSeitenlaenge;
        }

        double berechneFlaeche() {
                return seitenLaenge * seitenLaenge;
        }

        public static void main(String args[]) {
                Quadrat q = new Quadrat(1.5);
                System.out.println("Fläche: " +
                   q.berechneFlaeche());
        }
}
```

Beginn der Klassendefinition

seitenLaenge ist ein Attribut vom Typ double

Konstruktor initialisiert Attribut

Methode zur Flächenberechnung

Erzeugung eines Quadrat-Objekts

Berechnen und Ausgeben der Fläche

Anmerkungen zum Quadrat-Beispiel
Ein Objekt speichert Daten in Form von **Attributen** und **lokalen Variablen**:

- Die Attribute bestimmen i.W. den Speicherbedarf eines Objekts und werden auf dem Heap angelegt. Ihre Lebenszeit endet mit der Auflösung des Objekts (vgl. Abschnitt 1.7). Attribute werden durch den Compiler automatisch initialisiert. Da sie die Variable eines Objekts sind, werden sie auch Instanzvariable genannt. Beispiel: seitenlaenge
- Lokale Variable und Parameter existieren nur solange der Block (z.B. Methodenkörper, Schleifenkörper, then-Zweig), in dem Sie definiert sind, existiert. Sie werden auf dem Stack angelegt und nicht automatisch initialisiert.

main()-Methode
main() gehört als statische Methode zur Klasse Quadrat und ist damit eine Eigenschaft der Quadrat-Klasse, nicht der Quadrat-Objekte. Sie dient zum parametrisierten Starten einer Anwendung. Jede Klasse kann höchstens eine *main()*-Methode enthalten, eine Anwendung jedoch beliebig viele!

Java-Konvention zur Namensgebung:
Klassen beginnen mit Großbuchstaben. Lokale Variablen, Attribute und Methoden beginnen mit Kleinbuchstaben. Namensbestandteile werden durch Großbuchstaben getrennt (Camel Notation).
Also: *seitenLaenge* und nicht *seiten_laenge*

Java Programmierkonventionen sind z.B. hier zu finden: https://google.github.io/styleguide/javaguide.html

Instanzen primitiver Datentypen (boolean, char, int, …)
Aus Performanz- und Speicherplatz-Gründen werden Instanzen primitiver Datentypen nicht als Objekte realisiert. Wird beispielsweise mit „int eineGanzeZahl;“ ein Attribut oder eine lokale Variable deklariert, so bewirkt dies zur Laufzeit eine automatische Reservierung von Speicherplatz (4 oder 8 Byte, je nach Hardware) für eine ganze Zahl.

Objekte werden mit Hilfe des vordefinierten Operators „new“ erzeugt. Dieser reserviert Speicherplatz für die Attribute des Objekts und initialisiert die Attribute. Anschließend wird die passende Konstruktor-Methode ausgeführt, die zusammen mit „new“ aufgerufen wurde. Der Java-Compiler ermittelt den korrekten Konstruktor anhand der Parametertypen des new-Aufrufs.
„new“ liefert eine Referenz auf das erzeugte Objekt, welche typischerweise in einem Attribut/einer Variable gespeichert wird, um das Objekt später aufrufen zu können.

Beispiel:

```
Quadrat q = new Quadrat(1.4);          // Quadrat erzeugen, Referenz in „q“ speichern
double flaeche = q.berechneFlaeche();  // berechneFlaeche() mithilfe von „q“ aufrufen, Ergebnis in Variable „flaeche“ speichern
System.out.println(flaeche);           // „flaeche“ ausgeben
```

Konstruktoren (engl. *Constructor*, kurz CTOR)

Konstruktoren sind spezielle Methoden einer Klasse zur Initialisierung von Objekten. Ein Konstruktor besitzt den gleichen Namen wie die Klasse und kann Parameter besitzen, jedoch keinen Rückgabewert. Mehrere Konstruktoren unterscheiden sich in ihren Parametertypen.

Beispiel:

```
class Rechteck {
    double hoehe, breite;  // jedes Rechteck hat Höhe und Breite

    Rechteck(double wert1, double wert2) {  // Konstruktor: initialisiert die Attribute aus den Parametern
            hoehe = wert1;
            breite = wert2;
    }
    Rechteck(double wert) { // Konstruktor: nutzt den 1. CTOR, wir erhalten ein Quadrat
        this(wert, wert);
    }
    public static void main(String[] args) {
        Rechteck r1 = new Rechteck(3.0, 4.0); // Rechteck mit Seiten 3.0 und 4.0
        Rechteck r2 = new Rechteck(1.5);        // Quadrat mit Seitenlänge 1.5
    }
}
```

Erläuterung des Beispiels
Der erste Konstruktor besitzt zwei Parameter zur Initialisierung der beiden Seiten. Der zweite Konstruktor soll ebenfalls beide Seiten initialisieren, besitzt aber nur einen Parameter. Dieser wird zur Initialisierung beider Seiten verwendet, indem der erste Konstruktor mittels der vordefinierten Variable „this“ (siehe 1.5) aufgerufen wird. Anhand der beiden Parameter erkennt der Compiler, dass der erste Konstruktor aufgerufen werden soll.
Diese Art des Aufrufs eines CTORs durch einen anderen nennt man Konstruktorenverkettung.

Einschränkungen bei Konstruktorenverkettung
Konstruktoren können andere Konstruktoren mit folgenden Einschränkungen aufrufen
- Konstruktoren dürfen nur von anderen Konstruktoren (oder implizit durch „new“) aufgerufen werden
- Der Konstruktor-Aufruf muss die erste Anweisung in einem Konstruktor sein.
- Es kann höchstens ein anderer Konstruktor aufgerufen werden.

Default-Konstruktor
Wird kein Konstruktor definiert, so definiert der Compiler implizit einen parameterlosen Default-Konstruktor. Dieser reserviert Speicher für die Attribute des Objekts.

Neben Konstruktoren können auch Methoden mit gleichem Namen und unterschiedlicher Parameterliste definiert werden. Man nennt dies „Überladen“ von Methoden.

Beachte: der Ergebnistyp muss dabei stets der gleich sein!

Beispiel:

```
class Rechteck {
    double hoehe;
    double breite;
    ... // Konstruktor usw.
    void setMasse (double h, double b) {
        hoehe = h;
        breite = b;
    }
    void setMasse (double l) {
        setMasse(l, l);
    }
}
```

Überladen:
gleicher Methodenname und Ergebnistyp,
jedoch unterschiedliche Parameter!

Begriff „Referenz“
Eine Referenz ist ein Verweis (Aliasname) auf ein Objekt im Speicher. Es handelt sich nicht um die Speicheradresse! Eine Referenz hat einen bestimmten Typ (=Klasse) und verweist entweder auf ein Objekt vom bezeichneten Typ oder hat den vordefinierten Wert „null“.

Beachte: der Zugriff auf ein Objekt mittels einer Null-Referenz liefert eine sogenannte „Nullpointer-Exception“!

Beispiel:

```
Quadrat ref1 = new Quadrat(1.0);  // ref1 verweist auf Quadrat mit Seite 1.0
Quadrat ref2;                     // ref2 zeigt auf kein Objekt
Quadrat ref3 = new Quadrat(3.0);  // ref3 verweist auf Quadrat mit Seite 3.0

ref2 = ref1;                      // ref2 verweist auf Quadrat mit Seite 1.0
if (ref1 == ref2) {               // verweisen beide auf das selbe Quadrat?
   System.out.println(“gleich!“); // Ja! Ausgabe „gleich!“
}

ref2 = ref3;                      // ref2 verweist auf Quadrat mit Seite 3.0
```

Die „this“-Referenz

Objekte erhalten ein implizites Attribut namens „this“ als Referenz auf das Objekt selbst. Für Klassen macht „this“ keinen Sinn.
Anmerkung: dies ist die zweite Anwendung des „this“-Schlüsselwortes. Es wurde bereits für die Konstruktorenverkettung genutzt.

Anwendungsbeispiele

```
this.einAttribut = 3; // identisch mit einAttribut = 3
this.eineMethode(); // identisch mit eineMethode();
```

Zugriff auf Attribut
Aufruf einer Methode

```
Rechteck(double hoehe, double breite) {
    this.hoehe = hoehe;
    this.breite = breite;
}
```

Unterscheidung zwischen Parameter und Attribut: hoehe ist der Parameter, this.hoehe bezeichnet das Attribut

Methoden können mehrere Parameter besitzen. Bzgl. der Datentypen der Parameter unterscheidet man analog zu Attributen

- Primitive Typen

Wofür	Datentyp(en)	Vorbelegung (Attribute und Arrays)
Zeichen:	char (8 Bit)	'\0‘
Ganze Zahlen:	int, short(16 Bit), long (64 Bit)	0
Wahrheitswert:	boolean (Werte true bzw. false)	false
Gleitpunktzahlen:	float (*single precision*), double (*double precision*)	0.0f bzw. 0.0d (= 0.0)

- Objektreferenzen: Der jeweilige Parameter ermöglicht die Nutzung eines Objekts, falls der Wert des Parameters nicht *null* ist

Parameter werden in Java stets als Kopie (engl. *by value*) übergeben. Die Übergabe einer Adresse (=Referenz mit „&“-Operator) wie in „C“ gibt es nicht!

Entsprechend gibt es bei Parametern primitiver Datentypen keine Möglichkeit, die entsprechenden Variablen beim Aufrufer durch die aufgerufene Funktion zu verändern. Wird dagegen eine Objektreferenz übergeben, so kann das betreffende Objekt über die Referenz genutzt und ggf. durch den Aufruf entsprechender Methoden geändert werden.

Beispiel 1:

```
void passeLaengeAn(Quadrat p) {
    p.seitenLaenge = seitenLaenge;
}

... (in main) …
Quadrat a = new Quadrat(1.);
Quadrat b = new Quadrat(3.);
a.passeLaengeAn(b);
```

Das Attribut seitenLaenge des 2. Quadrats wird auf den Wert des seitenLaenge-Attributs des 1. Quadrats gesetzt.

Beispiel 2:

```
void ueberschreibe(Quadrat p) {
    p = this; // this: Referenz auf aufgerufenes Objekt
}

... (in main) …
Quadrat a = new Quadrat(1.);
Quadrat b = new Quadrat(3.);
a.ueberschreibe(b);
```

Der Wert des Parameters „p“ wird verändert, er verweist nun auf das 1. Quadrat. Nach außen, also zum Aufrufer hin, hat dies keine Wirkung!

Funktionsprinzip Garbage Collector

Anders als in „C“ oder C++ können Java-Objekte nicht explizit aufgelöst werden.
Stattdessen erkennt der sog. Garbage Collector (GC) nicht mehr referenzierte Objekte und löst diese auf. Der GC ist Bestandteil des Java Laufzeitsystems (Java Runtime) und wird periodisch im Hintergrund aktiv.

Prinzipielle Funktionsweise des GC

Für jedes Objekt o gibt es einen Zähler z

- Objekterzeugung: o.z = 1
- Zuweisung der Referenz: o.z += 1
- Referenz-Variable wird auf anderes Objekt oder null gesetzt: o.z -= 1
- Referenz-Variable verlässt Gültigkeitsbereich (Block): o.z -= 1
- falls o.z == 0 gilt, kann o aufgelöst werden (d.h. der Speicherplatz wird freigegeben)

Beispiel:

```
{
    Quadrat r = new Quadrat(1.);   // z==1
    Quadrat s = r;                 // z==2
    r = null;                      // z==1
}                                  // z==0 wg. Auflösung der Variablen -> Freigabe möglich
```

Zur Durchsetzung des **Kapselungsprinzips** sollen Objekt-Attribute von außen nie direkt, sondern nur über Methoden genutzt werden können. Diese Vorgehensweise reduziert Abhängigkeiten zwischen Klassen und verringert damit Seiteneffekte und Testaufwände bei Änderungen. Java bietet zur Umsetzung folgende Sichtbarkeitsstufen an:

Sichtbarkeitsstufen für Attribute bzw. Methoden

Stufe	Java	UML	Bedeutung
öffentlich	*public*	+	für alle Objekte nutzbar
package	(keine)	(keine)	nur von Klassen aus dem selben Package nutzbar
privat	*protected*	#	*nutzbar durch* Objekte derselben Klasse oder einer Unterklasse (s. Kapitel 3) sowie im selben Package
privat	*private*	-	nur innerhalb eines Objekts bzw. von Objekten der selben Klasse nutzbar

Sichtbarkeitsstufen für Klassen

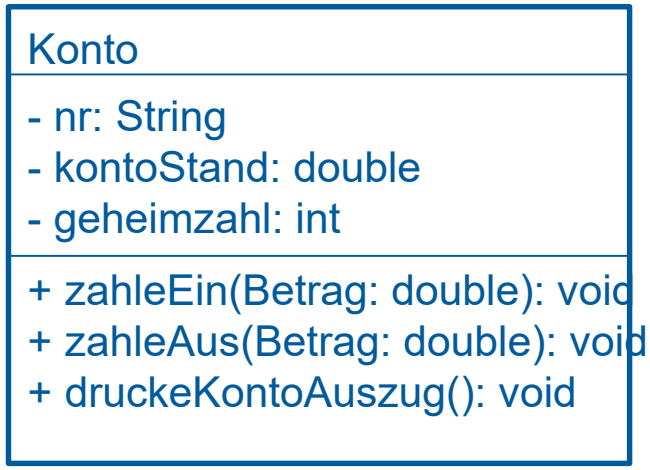
public: für alle Objekte nutzbar

keine Kennzeichnung: nur im selben Package (vgl. 1. 13) nutzbar

Dringende Empfehlung: Attribute (mit Ausnahme konstanter Attr.) nie *public* definieren!

Beispiel: Klasse Girokonto mit Sichtbarkeitsstufen

Umsetzung vom UML-Modell zur Java-Klasse




```
public class Girokonto {
    private String nr;
    private double kontoStand;
    private int geheimzahl;

    public void zahleEin(double Betrag) { … }
    public void zahleAus(double Betrag) { … }
    public void druckeKontoAuszug() { … }
};
```

Nutzung (in Methode einer anderen Klasse):

```
Girokonto k = new Girokonto();

k.zahleEin(100.0); // zahleEin ist public → OK
k.geheimzahl = 1234; // geheimzahl ist private → Fehler: geheimzahl ist private!
```

Arrays sind in Java Objekte, d.h. sie müssen mit „new“ erzeugt und anschließend initialisiert werden. Array-Elemente werden wie Attribute zur Laufzeit automatisch initialisiert, d.h. Elemente primitiver Typen werden mit

- Null-Character '\0' (char)
- 0 (int, long)
- 0.0 bzw. 0.0f (double, float)
- false (boolean)

belegt. Objektreferenzen werden mit „null“ initialisiert.
Als Konsequenz sind zwei Schritte nötig, um ein Feld in Java zu erzeugen und zu belegen:
1. Erzeugung des Array-Objekts, dabei Initialisierung wie oben beschrieben
2. Belegung der Array-Elemente

Beispiel:

```
int[] intArray = new int[2];          // Erzeugung eines Array-Objekts für 2 int-Werte
intArray[0] = 14;                     // Belegung des 1. Wertes
intArray[1] = intArray[0] + 1;        // Belegung des 2. Wertes
```

Weitere Array-Eigenschaften
- Gültige Index-Werte sind im Bereich 0, ..,Array.length-1 (Beispiel siehe nächste Seite)
- Ungültige Index-Werte führen zu einer Exception (IndexOutOfBounds) zur Laufzeit
- Performante Implementierung komplexer Array-Operationen gibt es in der Klasse *Arrays* (s. Java-API https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/Arrays.html)
  Beispiel: Array-Sortierung, s. nächste Seite

Beispiele zur Verdeutlichung der Syntax:

```
int[] intArray1; // Array-Referenz mit Wert null
int[] intArray2 = new int[10]; // Array-Referenz verweist auf Array-Objekt mit 10 int-Werten, alle =0
intArray1 = intArray2; // beide Referenzen verweisen auf das selbe Array, kein Kopiervorgang!
// statische Initialisierung der Array-Elemente
// beachte: statische Initialisierung nur bei Var.-Deklaration möglich!
int intArray3[] = new int[] { 1, 2, 3 };
for (int i = 0; i < intArray1.length; i++) // Schleife über Array-Größe (length-Attribut)
    intArray1[i] = i;
// Summenberechnung mit neuer for-each-artiger Schleife
// Syntax: for (<Elementtyp> <Laufvariable> : <Array-Referenz>)
// Achtung: es werden stets alle Elemente des Arrays durchlaufen, keine Veränderung des Arrays!
int summe = 0;
for (int wert : intArray1)
    summe += wert;

// Alternative mit Standard-For-Schleife
summe = 0;
for (int i = 0; i < intArray1.length; ++i)
    summe += intArray1[i];
```

Mehrdimensionale Arrays

Arrays mit Dimension > 1 werden durch Hinzufügen weiterer Klammerpaare „[ ]“ deklariert. Eine 2 x 2-Matrix ganzer Zahlen kann bspw. so deklariert werden:

```
int[][] intMatrix = new int[2][2];
```

Auch mehrdimensionale Arrays können bei der Deklaration statisch initialisiert werden, eine 2 x 3-Matrix aus Quadrat-Referenzen kann etwa folgendermaßen initialisiert werden:

```
Quadrat[][] quadratMatrix = new Quadrat[][] {
    { new Quadrat(1.0), new Quadrat(1.0), new Quadrat(3.0) },
    { new Quadrat(4.0), new Quadrat(5.0), new Quadrat(6.0) }
};
```

Zur Iteration kann die length-Eigenschaft für jede Dimension genutzt werden:

```
for (int zeile = 0; zeile < quadratMatrix.length; ++zeile)
  for (int spalte = 0; spalte < quadratMatrix[zeile].length; ++spalte)
    System.out.println(quadratMatrix[zeile][spalte].toString());
```

Sortieren von Arrays

Die Klasse java.util.Arrays definiert u.a. eine **`sort`**-Methode, mit der Arrays aufsteigend sortiert werden können.
Beispiel für ein int-Array:

```
int[] intArray = new int[] { 31, 22, 13 };
java.util.Arrays.sort(intArray1); // Reihenfolge jetzt: 13, 22, 31
```

Beachte: Zum Sortieren von Arrays mit Elementen von benutzerdefinierten Datentypen muss der Datentyp (z.B. Quadrat) das Comparable-Interface implementieren (s. Abschnitt 5)!

Java bietet eine vordefinierte Lösung für die Verarbeitung von Zeichenketten – die Klasse java.lang.String. Die String-Klasse ist wie alle Klassen aus `java.lang` standardmäßig importiert und kann folgendermaßen instanziiert werden:

```
String s1 = "Hallo";              // gleichbedeutend mit new String("Hallo")
String s2 = new String();         // leerer String
String s3 = new String("Hallo");  // vgl. erste Anweisung
```

Strings können nach ihrer Erzeugung nicht verändert werden, d.h. jede Änderung bewirkt die Erzeugung eines neuen String-Objekts. Möchte man aus Effizienzgründen anders vorgehen, so bietet sich z.B. die Klasse `StringBuilder` an.

Beachte:
String-Konstanten werden intern nur einmal im Speicher gehalten, daher wird für s3 im nachfolgenden Beispiel „new“ verwendet, um ein neues String-Objekt zu erzeugen:

```
String s1 = "Hallo";
String s2 = "Hallo";
String s3 = new String("Hallo");
System.out.println(s1 == s2); // true
System.out.println(s1 == s3); // false
```

Wichtige String-Methoden:

int compareTo(String s)	this < s: liefert <0, this == s: liefert 0, this > s: liefert >0
boolean equals(String s)	liefert true, falls this und s inhaltlich gleich sind
int length()	liefert die Länge des Strings
int indexOf(String s)	liefert die erste Position von s in String oder -1 (nicht gef.)
int lastIndexOf(String s)	liefert die letzte Position von s in String oder -1 (nicht gef.)
char charAt(int i)	liefert das Zeichen an der i. Position (ab 0...)
String trim()	entfernt Whitespace-Zeichen am Anfang und am Ende
String substring(int i1, int i2)	liefert den Teilstring von Position i1 bis i2-1
String substring(int i1)	liefert den Teilstring von Position i1 bis zum Ende
String replace(char x, char y)	ersetzt alle x durch y
String replace(String x, String y)	ersetzt alle x durch y
boolean startsWith(String s)	liefert true, falls this mit s beginnt
boolean endsWith(String s)	liefert true, falls this mit s endet
boolean contains(String s)	liefert true, falls this s enthält

Mehr Information siehe Java-API zur Klasse String:
https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/String.html

Beispiele zur Nutzung von Strings:

```
String s1 = "Mein";
String s2 = "Beispiel";
// Konkatenation mit "+"-Operator
String s3 = s1 + " " + s2;
s3 += " Nummer " + 1; // "Mein Beispiel Nummer 1"
System.out.println(s3);
// Teil-Strings bilden
System.out.print(s1.substring(1,3)); // "ei"
System.out.print(s1.substring(3)); // "spiel"
// Suche nach der Zeichenkette "spiel"
int pos = s1.indexOf("spiel"); // pos == 3
// einzelne Zeichen lesen
char c = s1.charAt(2); // c == 'i'
```

Die toString()-Methode

In Java hat jedes Objekt eine toString()-Methode, die eine String-Darstellung des Objekts erzeugen kann. Damit ist z.B. sichergestellt, dass jedes Objekt auf die Konsole ausgegeben werden kann. Für selbstdefinierte Klassen sollte die toString()-Methode überschrieben werden, um eine geeignete String-Repräsentation der Objekte zu erhalten.

Beispiel Quadrat-Klasse

```
public class Quadrat {
  private double seitenlaenge;

  public String toString() {
    return "Quadrat(seitenlaenge = " + seitenLaenge + ")";
  }

  static public void main(String[] args) {
    Quadrat q = new Quadrat(1.5);
    double flaeche = q.berechneFlaeche();
    System.out.printf("Fläche = %f\n", flaeche);
    System.out.println(q); // hier wird q.toString() aufgerufen!
  }
}
```

Es gibt zwei Arten der Gleichheitsprüfung:

1. Identität oder referentielle Gleichheit:
   Gegeben seien zwei Referenzen, ref1 und ref2, beide seien != null
   Falls gilt: ref1 == ref2 verweisen ref1 und ref2 auf das selbe Objekt

2. Wertmäßige Gleichheit oder Inhaltsgleichheit
   Zur Prüfung der Inhaltsgleichheit kann die für alle Objekte vordefinierte *equals()*-Methode verwendet werden.
   Diese ist z.B. für String definiert:

```
boolean istGleich = "Hallo".equals("hallo"); // istGleich hat den Wert false
```

Achtung:

```
String s1 = "Hallo";
String s2 = new String("Hallo");
if (s1 == s2)
        System.out.println("gleiches Stringobjekt!");
if (s1.equals(s2))
        System.out.println("s1 und s2 beinhalten den gleichen String!");
```

Ausgabe: s1 und s2 beinhalten den gleichen String!

Für **eigene Klassen** muss die equals()-Methode überschrieben werden, da die Default-Implementierung lediglich die beiden Referenzen auf Gleichheit prüft.

Beispiel für die Quadrat-Klasse: Quadrate sind gleich, wenn ihre Seitenlängen gleich sind

```
class Quadrat {
        private int seitenlaenge;
        //... weitere Definitionen
        public boolean equals(Object o) {
                return seitenlaenge == ((Quadrat)o).seitenlaenge;
        }
}
```

In gewissen Fällen möchte man Attribute bzw. Methoden für alle Objekte einer Klasse festlegen. In Java können daher Attribute und Methoden als statisch definiert werden.
Der Zugriff erfolgt dann über den Klassennamen, es ist kein Objekt der Klasse notwendig!
Beachte: Der Zugriff von statischen Methoden auf nicht-statische Elemente liefert einen Compiler-Fehler!

Beispiele

```
// 1. main-Methode zum Start
static public void main (String args[]) { … }
// 2. Ausgabe auf die Konsole, out ist static-Attribut von System
System.out.println("Hello, world!");
```

Verständnisfrage:
Welche Bedeutung hat zaehler im Beispiel rechts?

Antwort: zaehler enthält stets die Anzahl bereits erzeugter Objekte der Klasse X

```
public class X {
    private static int zaehler;
    public X() {
        zaehler++;
    }
}
```

Attribute sind typischerweise nicht *public* definiert, um unkontrollierte Zugriffe durch andere Objekte zu verhindern. Für den kontrollierten Zugriff definiert man (soweit nötig!) je Attribut zwei Methoden, eine für den Lese- und eine für den Schreibzugriff.
Folgende **Namenskonvention** stellt eine einheitliche Verwendung sicher:
Gegeben sei das Attribut namens „test“

private T test; // T sei irgendein Datentyp, z.B. int

→ Getter für „test“: public T getTest() { return test }

→ Setter für „test“: public void setTest(T wert) { test = wert; }

Beispiel:

```
public class Quadrat {
        // Attribut für Seitenlänge
        private double seitenLaenge;

        public double getSeitenLaenge() {
                return seitenLaenge;
        }
        public void setSeitenLaenge(double seitenLaenge) {
                this.seitenLaenge = seitenLaenge;
        }
}
```

Große Java-Programme bestehen aus sehr vielen Klassen. Um die Übersicht zu behalten, werden Klassen in ***packages*** (dt. Paket) organisiert. Analog zum Dateisystem mit Ordnern ergibt sich eine baumartige Struktur, da ein package Klassen, aber auch weitere packages enthalten kann. Die Java-Laufzeitbibliothek enthält z.B. das package „java.lang“ für Standard-Klassen wie die Klasse String, der vollständige Pfad zur Nutzung der String-Klasse ist damit: java.lang.String. Der vollständige Aufruf der println-Methode lautet:

```
java.lang.System.out.println(<irgendein String>);
```

Importieren von Klassen

Die Nutzung der Paketpfade ist umständlich. Daher können Klassen durch „import“-Anweisungen in einer Java-Quelldatei bekannt gemacht werden. Damit genügt die Angabe des Klassennamens.

Beispiele:

```
import java.util.Date; // importiere Date zur Speicherung von Zeitangaben
import java.util.*;    // importiere alle Klassen aus java.util
```

Eigene Pakete definieren

Die erste Anweisung in einer Klassendefinition ist typischerweise eine package-Definition, z.B.

```
package paket1.paket2; // nachfolgende Klassen liegen in paket2 (welches in paket1 liegt)
```

Ohne eine package-Anweisung wird das default-package (Baumwurzel) genutzt.

Statische Imports von Klassen

In Java kann man auch auf statische Attribute und Methoden ohne Voranstellung des Klassennamens zugreifen, indem man die entsprechende Klasse statisch importiert:

```
import static java.lang.Math.*;
import static java.lang.System.*;

class MathTester {
     public static void main(String[] args) {
          System.out.println(Math.sin(3.14));  // Langversion
          out.println(sin(3.14));              // Kurzversion
     }
}
```

Im Beispiel oben kann die letzte Zeile sehr kurz geschrieben werden, da `out` ein statisches Attribut der Klasse `System` ist bzw. `sin` eine statische Methode der Klasse `Math` ist und beide Eigenschaften öffentlich (public) sind.