Programmieren I in Java

Dr. Klaus Höppner

1. Zusammenfassung

Inhaltsverzeichnis

1 Grundlagen der objektorientierten Programmierung

1.1 Prinzip

In der objektorientierten Programmierung werden Daten und die zugehörigen Funktionen als Einheit betrachtet.

Grundeinheiten der objektorientierten Programmierung sind die *Klassen*. Hierbei wird eine Menge von Objekten mit gleichen Eigenschaften und Fähigkeiten durch die Definition einer Klasse beschrieben.

Die einzelnen Objekte, die einer bestimmten Klasse angehören, nennt man dann Instanz einer Klasse.

1.2 Grundlegende Eigenschaften der objektorientierten Programmierung

Abstraktion Ein Objekt kann Aufträge ausführen, über seinen Zustand berichten und mit anderen Objekten kommunizieren, ohne die eigentliche Implementation offen zu legen.

Kapselung Durch das Verstecken der internen Struktur ist nur ein Zugriff auf das Objekt über definierte Schnittstellen möglich. Hierdurch kann sicher gestellt werden, dass ein Objekt intern immer in einem konsistenten Zustand bleibt.

Vererbung Klassen können von anderen Klassen erben.

Dies bedeutet, dass eine neue Klasse die Eigenschaften und Methoden der Elternklasse *erbt*, d. h. diese übernimmt, ohne dass sie noch einmal neu implementiert werden müssen.

In der neuen Klasse können dann neue Methoden und Eigenschaften definiert werden, weiterhin können z. B. Methoden der Ursprungsklasse überschrieben oder erweitert werden.

Bei der Vererbung entsteht in der Regel aus einer allgemeinen Klasse eine spezialisiertere Klasse

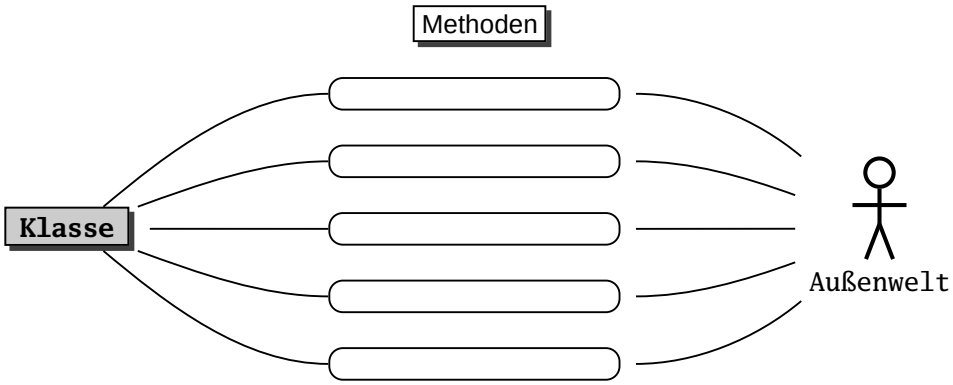


Abbildung 1: Kapselung einer Klasse: Zugriff von außen findet nur über öffentliche Methoden statt.

Polymorphie (*späte Bindung*) Eine Methode einer Klasse ist polymorph, wenn sie in verschiedenen, durch Vererbung oder durch die Implementierung eines Interfaces verwandten Klassen die gleiche Spezifikation hat, aber unterschiedlich implementiert ist.

Polymorphie (oder späte Bindung) bedeutet hierbei, dass erst zur Laufzeit bestimmt wird, welche Methode aufzurufen ist. Es kann also vom Programmablauf abhängig sein, welche Methode in Anwendung kommt. Polymorphie ist einer der wichtigsten Bestandteile der objektorientierten Programmierung.

Diese Vorteile dieser grundlegenden Eigenschaften wirken sich insbesondere bei der modernen Softwareentwicklung im Team aus. Hier steht die Einigung auf eine Struktur der Klassen mit definierten Schnittstellen (so genannte Interfaces) im Vordergrund. Der *interne* Ablauf in einer Klasse ist dann nur für diejenigen Personen des Teams relevant, die an einer konkreten Klasse arbeiten.

2 Grundsätzlicher Ablauf

Zum Erstellen einer Java-Anwendung schreibt man mit einem beliebigen Editor – dies können normale Texteditoren wie Notepad unter Windows oder Emacs auf Unix-Systemen sein, oder spezielle Entwicklungsumgebungen wie Eclipse oder NetBeans – Quelltext-Dateien (eine pro Klasse).

Der Quelltext wird von dem Java-Compiler (`javac`) in einen Byte-Code übersetzt, der sich je Klasse in einer Datei mit der Endung `.class` befindet. Dieser ist (unabhängig vom verwendeten Betriebssystem) in der *Java Virtual Machine* ausführbar, indem der Java-VM der Name der Klasse übergeben wird, in der sich das Hauptprogramm befindet (**`public static void`** `main`).

Der grundsätzliche Ablauf bei der Programmentwicklung in der Kommandozeile sieht dabei so aus:

```
> notepad HelloWorld.java
> javac HelloWorld.java
```

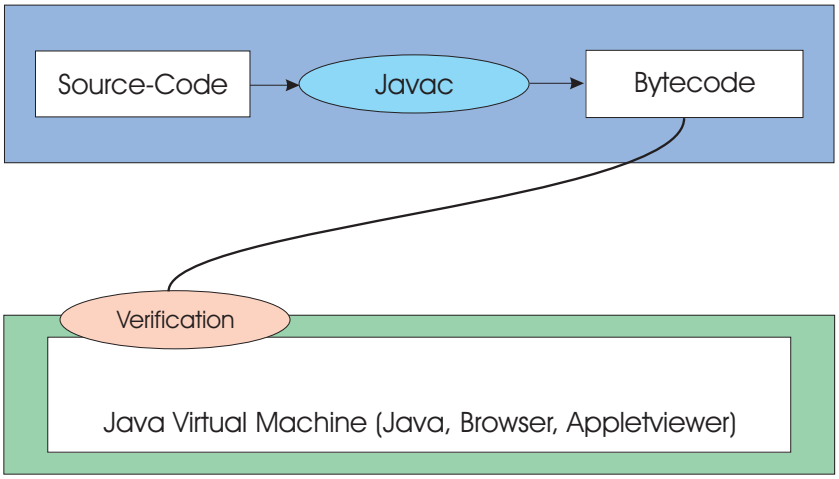


Abbildung 2: Erstellen und Benutzen einer Java-Anwendung

```
> java HelloWorld
```

Während man zum Übersetzen die Java-Entwicklungsumgebung benötigt (Java Development Kit – JDK), reicht zum Ausführen des Byte-Codes die Java-Laufzeitumgebung (Java Runtime Environment – JRE).

3 Definition von Klassen in Java

3.1 Grundgerüst

Klassen werden in Java mit dem Schlüsselwort **`class`** definiert.

Der Grundaufbau lautet:

```
Sichtbarkeit class Klassenname {
    ...
}
```

In der Klassendefinition werden dann die zur Klasse gehörigen Daten (Attribute, auch Member-Variablen genannt) und Funktionen (Methoden) definiert.

Konventionen:

- In Java muss sich jede Klasse in einer eigenen Quelltext-Datei befinden, die den Namen `Klassenname.java` hat.
- Die Namen von Klassen beginnen mit einem Großbuchstaben, die Namen von Attributen und Methoden mit einem Kleinbuchstaben.

- Die Klassendefinition besteht aus dem Kopf (Sichtbarkeit, Klassenname, evtl. Angaben zu Klassen, von denen geerbt wird, und Interfaces, die implementiert werden) und dem Körper, der Attribute und Methoden enthält, und von geschweiften Klammern umschlossen wird.

3.2 Pakete (*packages*)

Um größere Projekte überschaubar zu halten, können Klassen in Java zu einem Modul zusammengefasst werden, das als *package* bezeichnet wird.

Packages definieren einen eigenen Namensraum, d. h. es ist möglich, dass Klassen in verschiedenen Paketen gleich heißen, ohne dass es dabei zu Namenskonflikten kommt. Weiterhin kann die Sichtbarkeit von Klassen, Methoden oder Attributen auf ein Package beschränkt werden.

Java-Klassen, die zu einem Packages gehören

- haben als erste Code-Zeile in der Quelltext-Datei die Zeile

  package *Paketname*;

  und
- befinden sich in einem Unterverzeichnis, das wie das Package heißt.

3.3 Sichtbarkeit von Klassen und deren Elemente

Für Klassen selber und deren Attribute und Methoden kann bestimmt werden, in welchem Kontext sie sichtbar sind. Hierbei gibt es in Java vier Sichtbarkeitsstufen:

private Sichtbarkeit nur innerhalb derselben Klasse,

(keine Angabe) Sichtbarkeit innerhalb desselben Paketes (**package**),

protected Sichtbarkeit in allen erbenden Klassen (zusätzlich zur Sichtbarkeit im selben Paket!)

public Sichtbarkeit in allen Klassen

Tipp:

- In der Regel sind die Attribute einer Klasse privat.
- Die Schnittstellen nach außen werden als öffentliche Methoden implementiert.
- Häufig wird es auch private Methoden geben, die für den internen Ablauf in der Klasse zuständig sind.

3.4 Definition von Attributen und Methoden

Attribute und Methoden werden innerhalb des Körpers der Klasse definiert:

```
Sichtbarkeit class Klassenname {
    Sichtbarkeit typ1 name1;

    Sichtbarkeit typ2 name2(parameter) {
        ...
        return wert;
    }
}
```

Java ist eine streng typisierte Sprache, d. h. für jedes Attribut und jede lokale Variable innerhalb von Methoden muss ein Typ deklariert werden. Hierbei sind als Typen neben den elementaren Datentypen Objektreferenzen erlaubt. Dasselbe gilt für den Typ des Rückgabewertes von Methoden. Methoden, die *keinen* Wert zurück geben, werden mit dem Rückgabetyp **void** definiert.

Methoden mit einem Rückgabetyp, der nicht **void** ist, müssen mit dem Befehl **return** einen Wert zurück geben. Hierbei dürfen im Code der Methode mehrere **return** vorkommen. Beim ersten **return**, das ausgeführt wird der dort angegebene Wert zurück gegeben und das Ausführen der Methode abgebrochen! Die Parameterliste bei Methoden muss immer angegeben werden, selbst wenn eine Methode keine Parameter hat. In diesem Fall wird also ein leeres rundes Klammerpaar `()` angegeben.

Attributen kann ein Vorgabewert zugewiesen werden.

Zusätzliche Qualifizierer

Neben dem Qualifizierer, der die Sichtbarkeit von Attributen bzw. Methoden bestimmt, sind noch folgende Qualifizierer zulässig:

final Bei Attributen ist deren Wert endgültig festgelegt, es kann ihnen also kein neuer Wert zugewiesen werden. Finalen Attributen muss bei der Definition ein Wert zugewiesen werden.

Bei Methoden bedeutet **final**, dass diese in erbenden Klassen nicht überladen werden kann.

static Mit diesem Schlüsselwort werden *Klassen*attribute bzw. *Klassen*methoden gekennzeichnet. Ein statisches Attribut existiert somit nur ein einziges Mal für die Klasse als ganzes, unabhängig von der Zahl der existierenden Instanzen der Klasse. Eine statische Methode wird für die Klasse als ganzes aufgerufen, daher existiert in diesen keine Objektreferenz **this** und es können nur statische Attribute und Methoden der Klasse verwendet werden.

abstract kennzeichnet eine Methode als abstrakt, d. h. es fehlt eine konkrete Implementierung der Methode, die dann in einer Kindklasse nachgeholt werden muss. Abstrakte Methoden sind nur in abstrakten Klassen und in Interfaces zulässig.

3.5 Die `main`-Methode

Ausführbare Programme enthalten ein Hauptprogramm, das beim Start der Anwendung ausgeführt wird. Hierfür befindet sich in einer Klasse eine statische Methode `main`, z. B.:

```
public class Anwendung {
    public static void main(String[] args) {
        System.out.println("Hallo Welt");
    }
}
```

In dem Array `args` befinden sich die beim Start des Programms angegebenen Kommandozeilen-Parameter.

3.6 Das Speichermodell von Java

Die *Java virtual machine* verwendet zwei Arten von dynamisch verwalteten Speicher, in denen zur Laufzeit Speicher belegt und wieder freigegeben werden kann: den Stack und den Heap.

Zwischen beiden Speicherarten 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, und statische Attribute von Klassen.

- 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 (nicht-statischen) Attribute angelegt und zerstört.

Eine schematische Darstellung von Stack und Heap befindet sich in Abb. 3.

4 Sprachelemente von Java

4.1 Anweisungen und Anweisungsblöcke, Kommentare

Alle Anweisungen in Java enden mit einem Semikolon.

Mehrere Anweisungen können zu einem Anweisungsblock zusammengefasst werden, indem sie in geschweiften Klammern eingeschlossen werden.

Java kennt einzeilige und Blockkommentare. Einzeilige Kommentare werden mit `//` eingeleitet und gelten bis zum Ende der Zeile. Blockkommentare werden von `/* ... */` umschlossen.

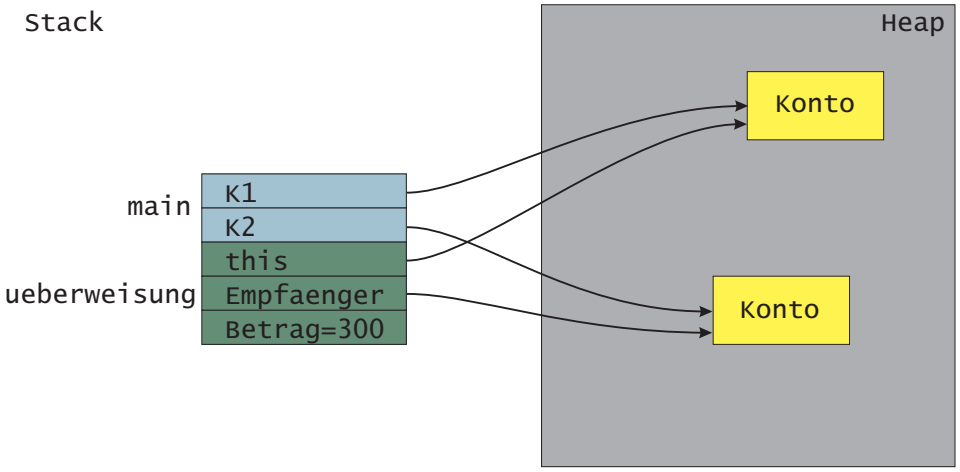


Abbildung 3: Stack und Heap

4.2 Variablen

Neben Attributen in Klassen können in jeder Methode lokale Variablen definiert werden. Dies geschieht analog zur Definition von Attributen, allerdings sind die Schlüsselwörter für Sichtbarkeit und **`static`** nicht zulässig (und sinnlos). Die Definition einer Variablen lautet also:

```
typ name;
```
oder
```
typ name = wert;
```

Parameter von Methoden gelten ebenfalls als lokale Variablen.

Als Typen sind jeweils elementare Datentypen und Referenzen auf Instanzen einer Klasse zulässig.

4.3 Elementare Datentypen

Java kennt vier Arten von elementaren Datentypen: Wahrheitswerte, Zeichen, ganze Zahlen und Gleitkommazahlen. Diese werden in Tabelle 1 aufgelistet.

4.4 Arrays

Arrays sind Felder von Elementen gleichen Typs, wobei Arrays eine feste Länge haben. Die Definition eines Arrays in Java sieht dabei folgendermaßen aus:

```
typ[] name = new typ[anzahl];
```

Auf die einzelnen Elemente eines Array wird mit dem Indexoperator `[]` zugegriffen, z. B. `a[i]`. In Java findet eine Bereichsüberprufung statt, ob der Index innerhalb der Grenzen des Arrays liegt, ansonsten wird eine *Exception* geworfen.

Tabelle 1: Elementare Datentypen in Java

Typ	Bedeutung	Beispiele
boolean	Wahrheitswert	`true`, `false`
char	Zeichen	`'A'`, `'m'`
short int long	ganze Zahl	`-3`, `17`
float double	Gleitkommazahl	`2.718`

Achtung: Die Zählung der Array-Elemente beginnt bei 0. Gültige Indizes für einen Array der Länge n sind als $0 \ldots n-1$.

4.5 Operatoren

Java kennt insbesondere folgende arithmetische und logische Operatoren:

Zuweisungsoperator `=`

Vorzeichenoperator `+`, `-`

Arithmetische Operatoren `+` (Addition), `-` (Subtraktion), `*` (Multiplikation), `/` (Division), `%` (Modulo)

Vergleichsoperatoren `==` (gleich), `!=` (ungleich), `<`, `>`, `<=` (kleiner oder gleich), `>=` (größer oder gleich)

logische Operatoren `&&` (logisches und), `||` (logisches oder), `!` (logisches nicht)

zusammengesetzte Operatoren `+=`, `-=`, `*=`, `/=`, `%=`

Inkrement und Dekrement `++`, `--`

Diese Operatoren gibt es in Postfix- und Präfix-Schreibweise.

Bei der Präfix-Schreibweise ist der Rückgabewert der Wert der Variablen *nach* dem Inkrementieren bzw. Dekrementieren; bei der Postfix-Schreibweise ist der Rückgabewert der Wert der Variablen *vor* dem Inkrementieren bzw. Dekrementieren.

Die Priorität der Operatoren ergibt sich aus Tabelle 2

Tabelle 2: Priorität der Operatoren

Priorität	Operator
1.	++ (Postfix), -- (Postfix)
2.	! (logisches nicht) + (Vorzeichen), - (Vorzeichen) ++ (Präfix), -- (Präfix) & (Adresse), * (Zeiger)
3.	*, /, %
4.	+ (Addition), - (Subtraktion)
5.	&& (logisches und)
6.	|| (logisches oder)
7.	=, +=, -=, *=, /=, %=

4.6 Operatoren im Zusammenhang mit Klassen

Neue Instanzen einer Klasse werden mit dem Operator **new** angelegt.

Auf Attribute und Methoden einer Instanz der Klasse wird mit dem Punktoperator zugegriffen.

Beispiel:

```
Klassenname k1 = new Klassenname();

int i = k1.methode1();

System.out.println(k1.attribut1);
// Hierfür muss das Attribut natürlich öffentlich sein.
```

Ein weiterer wichtiger Operator ist **instanceof**, mit dem geprüft werden kann, ob es sich bei einem Objekt um die Instanz einer bestimmten Klasse handelt:

```
objektname instanceof Klassenname
```

4.7 Der Cast-Operator

In einigen Fällen ist in Java eine explizite Typumwandlung erforderlich, z. B.

```
int i = 127;
short s = (short) i;
```

Der Cast-Operator muss auch beim so genannten *down cast* bei Objekten verwendet werden.

4.8 Verzweigungen

Mit der `if`-Anweisung kann erreicht werden, dass Anweisungen nur ausgeführt werden, wenn eine Bedingung erfüllt ist. Die allgemeine Form lautet:

```
if ( Bedingung ) {
    Anweisung1;
    Anweisung2;
    ...
}
```

Es kann auch ein `else`-Zweig angegeben werden, der ausgeführt wird, wenn die Bedingung nicht erfüllt ist:

```
if ( Bedingung ) {
    Anweisung1;
    Anweisung2;
    ...
} else {
    Anweisung3;
    Anweisung4;
    ...
}
```

`if`-Anweisungen können auch verkettet werden:

```
if ( Bedingung1 ) {
    Anweisung1;
    Anweisung2;
    ...
} else if ( Bedingung2 ) {
    Anweisung3;
    Anweisung4;
    ...
} else {
    Anweisung5;
    Anweisung6;
    ...
}
```

4.9 Schleifen

Java kennt drei Arten von Schleifen:

`while`-Schleife Die `while`-Schleife prüft *vor* dem Abarbeiten der Anweisungen in der Schleife, ob die Bedingung zum Durchlaufen der Schleife erfüllt ist. Es kann also sein, dass die Schleife *gar nicht* durchlaufen wird.

Allgemeine Form:

```
while ( Bedingung ) {
    Anweisung1;
    Anweisung2;
    ...
}
```

do-while-Schleife Die do-while-Schleife prüft die Bedingung erst nach dem Abarbeiten der Anweisungen in der Schleife. Sie wird also mindestens einmal durchlaufen.

Allgemeine Form:

```
do {
    Anweisung1;
    Anweisung2;
    ...
} while ( Bedingung );
```

for-Schleife Eine for-Schleife hat die allgemeine Form:

```
for ( Anweisung1; Bedingung; Anweisung2 ) {
    Schleifenanweisung1;
    Schleifenanweisung2;
    ...
}
```

Zu Beginn der for-Schleife wird *einmalig* Anweisung1 ausgeführt. Anschließend wird vor jedem Durchlaufen der Schleife die Bedingung geprüft. Nach Abarbeiten der Anweisungen in der Schleife wird dann jeweils Anweisung2 ausgeführt.

Erweiterte for-Schleife Eine Iteration über alle Elemente eines Arrays (oder eines Daten-Containers, der das Interface `Iterable` implementiert) kann man ab Java-Version 5 mit der erweiterten for-Schleife erreichen.

Beispiel:

```
int[] feld = new int[10];
...
for (int wert : feld) {
    System.out.println(wert);
}
```

Tipps

- Bei der do-while-Schleife folgt nach dem `while (...)` ein Semikolon.
- Bei den beiden anderen Schleifentypen steht hinter dem `while (...)` bzw. `for (...)` in aller Regel *kein* Semikolon (und ein dort versehentlich stehendes Semikolon führt gerne zu Endlosschleifen).

5 Objektorientierte Programmierung in Java

5.1 Konstruktoren

Grundlegendes

Im Programmtext selber werden Instanzen der Klasse mit dem **new**-Operator angelegt (und dann meist einer Objektreferenz zugewiesen):

```
Klassenname ref = new Klassenname();
```

Beim Anlegen einer Instanz der Klasse wird jeweils eine besondere Methode der Klasse aufgerufen: der *Konstruktor*.

Wie bei anderen Methoden können beim Aufruf des Konstruktors auch Parameter angegeben werden:

```
Klassenname ref = new Klassenname(Parameter);
```

Definition von Konstruktoren

Der Konstruktor ist eine Methode der Klasse, die genauso heißt wie die Klasse. Bei der Definition des Konstruktors wird kein Rückgabetyp angegeben:

```
public class Klassenname {
    public Klassenname(Parameter) {
        ...
    }
}
```

Der Standardkonstruktor

Wird *kein* eigener Konstruktor definiert, existiert der so genannte Standardkonstruktor implizit:

```
public Klassenname() {
}
```

Tipps:

- Es können auch mehrere Konstruktoren für unterschiedliche Parameterkombinationen definiert werden.
- Der Compiler muss bei jedem Anlegen einer Instanz in der Lage sein, den passenden Konstruktor anhand der Parameterliste eindeutig auswählen zu können.
- Werden eigene Konstruktoren definiert, gibt es keinen automatischen Standardkonstruktor mehr. Das Anlegen von Instanzen ohne Parameter

  ```
  Klassenname ref = new Klassenname();
  ```

  ist dann nur noch möglich, wenn der Standardkonstruktor explizit definiert wird.
- Als *erste* Zeile im Konstruktor darf entweder mit **super**(*parameter*) angegeben werden, wie der Konstruktor der Elternklasse aufgerufen werden soll, oder mit **this**(*parameter*) ein anderer Konstruktor derselben Klasse aufgerufen werden.

5.2 Finalisierung

Da die Speicherverwalung von Java sich eigenständig um das Zerstören von Objekten kümmert, gibt es im Gegensatz zu anderen objektorientierten Sprachen keinen Operator, mit dem explizit ein Objekt zerstört werden kann. Der Prozess der Zerstörung von Objekten durch die Speicherverwaltung (*garbage collection*) wird als Finalisierung bezeichnet.

Nach Wunsch kann in einer Klasse die Methode **public void** `finalize()` überladen werden. Dies ist sinnvoll, wenn vor dem Zerstören einer Instanz noch Aktionen ausgeführt werden sollen, z B. offene Dateien oder Netzwerkverbindungen geschlossen werden sollen.

Beispiel In einem statischen Attribut soll die Zahl der Instanzen einer Klasse mitgezählt werden:

```
public class Klassenname {
    private static int instanzzahl = 0;
    public Klassenname() {
        ++instanzzahl;
        ...
    }

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

Achtung: Sobald keine Referenz mehr auf ein Objekt verweist, kann dieses zerstört werden. Es bleibt aber der Speicherverwaltung überlassen, wann dies geschieht. Durch den Befehl `System.gc()` kann eine garbage collection erzwungen werden.

5.3 Die Referenz `this`

Damit man in einer Methode auf die Attribute und anderen Methoden der aktuellen Instanz (also die Instanz der Klasse, für die die Methode aufgerufen wird) zugreifen kann, existiert in jeder (nicht statischen) Methode die Referenz `this` auf das aktuelle Objekt.

Somit kann mit

```
this.attribut1
```

auf ein Attribut der aktuellen Instanz zugegriffen und mit

```
this.methode1(...)
```

eine Methode für die aktuelle Instanz aufgerufen werden.

Das **`this.`** kann weggelassen werden, wenn der Name des Attributs bzw. der Methode eindeutig ist. Dies geht also solange, wie es keine lokale Variable mit dem gleichen Namen wie das Attribut gibt.

5.4 Zugriffsmethoden

Häufig benötigt man Methoden, mit denen man den Wert von Attributen lesen oder sogar ändern kann. Solche Methoden heißen Zugriffsmethoden und haben eine besonders einfache Struktur.

```
public class Klassenname {
    private typ1 attribut1;

    public typ1 getAttribut1() {
        return attribut1;
    }

    public void setAttribut1(typ1 attribut1) {
        this.attribut1 = attribut1;
    }
};
```

6 Beispiel für eine einfache Klasse

6.1 Modell

Als Beispiel soll hier ein (vereinfachtes) Radio dienen.

Ein Radio hat interne Attribute (z. B die eingestellte Lautstärke und Frequenz).

Für den Besitzer des Radios spielt der interne Aufbau des Radios keine Rolle. Er »kommuniziert« mit dem Radio über die Bedienelemente. Die Bedienelemente repräsentieren also die

öffentlichen Methoden des Radios, beispielsweise leiser und lauter stellen oder Wahl einer neuen Frequenz.

Eine schematische Darstellung des Radiomodells findet sich in Abbildung 4.

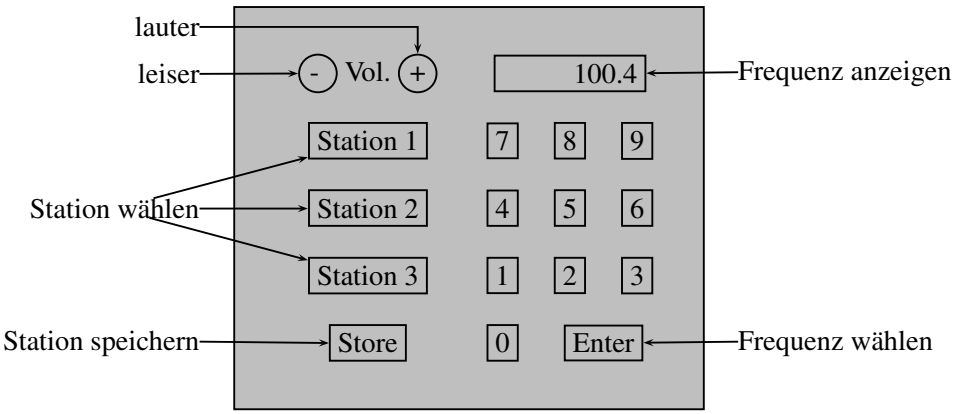


Abbildung 4: Modellansicht eines Radios

6.2 Implementierung

Die Definition der Klasse wird in der Datei `Radio.java` vorgenommen.

```
package radio;
/**
 * Klasse zur Darstellung eines Radios.
 *
 * @author klaus
 *
 */
public class Radio {
    private double frequenz;
    private int lautstaerke;
    private double[] stationen;

    private final static double fmin = 87.6;
    private final static double fmax = 108;
    private final static int lmin = 0;
    private final static int lmax = 10;

    /**
     * Standardkonstruktor der Klasse Radio.
     * Hierin werden die Attribute für Frequenz, Lautstärke und
     * Stationstasten initialisiert.
     */
    public Radio() {
        this.frequenz = Radio.fmin;
        this.lautstaerke = (Radio.lmin+Radio.lmax)/2; // Mittlere Lautstärke
        this.stationen = new double[5];
        for (int index=0; index<this.stationen.length; index++) {
            this.stationen[index] = Radio.fmin;
        }
```

```
}

/**
 * Get-Methode für die Frequenz.
 *
 * @return aktuelle Frequenz
 */
public double getFrequenz() {
    return frequenz;
}
/**
 * Set-Methode für die Frequenz.
 * Hierbei wird der zuläassige Frequenzbereich berücksichtigt.
 *
 * @param frequenz zu setzenden Frequenz
 */
public void setFrequenz(double frequenz) {
    if (frequenz>=fmin && frequenz<=fmax) {
        this.frequenz = frequenz;
    }
}
/**
 * Get-Methode für die Lautstärke
 * @return aktuelle Lautstärke
 */
public int getLautstaerke() {
    return lautstaerke;
}
/**
 * Lautstärke erhöhen.
 * Diese Methode erhöht (so möglich) die Lautstärke um eine Stufe.
 */
public void lauter() {
    if (lautstaerke<lmax) {
        lautstaerke++;
    }
}
/**
 * Lautstärke reduzieren.
 * Diese Methode erniedrigt (so möglich) die Lautstärke um eine Stufe.
 */
public void leiser() {
    if (lautstaerke>lmin) {
        lautstaerke--;
    }
}
/**
 * Aktuelle Frequenz auf einer Stationstaste speichern.
 * Diese wird auf der Stationstaste mit dem als Parameter übergebenen
 * Index abgespeichert.
 * @param index Nummer der Stationstaste
 */
public void speicherStation(int index) {
    stationen[index] = frequenz;
}
/**
 * Eine abgespeicherte Frequenz zur aktuellen Frequenz machen.
```

```
     * Hier wird die auf der Stationstaste mit dem übergebenen Index
     * abgespeicherte Frequenz als aktuelle Frequenz übernommen.
     *
     * @param index Nummer der Stationstaste
     */
    public void waehleStation(int index) {
        frequenz = stationen[index];
    }
}
```

Nun kann man in einem Hauptprogramm eine Instanz der Klasse anlegen und das Radio »bedienen«:

```
/**
 * Beispielanwendung für ein Radio.
 * Diese Klasse enthält als einzige Methode {@link #main(String[])} mit
 * dem Hauptprogramm.
 */

import radio.*;

public class RadioAnwendung {

    /**
     * Hauptprogramm.
     * Hier wird eine Instanz von {@link Radio} angelegt und einige Aktionen mit ihm durchgeführt.
     *
     * @param args nicht benutzt
     */
    public static void main(String[] args) {
        Radio r = new Radio();

        r.lauter();
        r.setFrequenz(89.3);
        r.speicherStation(0);
        r.setFrequenz(103.4);
        r.speicherStation(1);

        System.out.println("Aktuelle Frequenz: "+r.getFrequenz());

        r.waehleStation(0);

        System.out.println("Aktuelle Frequenz: "+r.getFrequenz());

    }

}
```

6.3 Statische Attribute und Methoden

Statische Attribute

Ein statisches Attribut ist ein Attribut, das für alle Instanzen einer Klasse gleich ist. Dieses Attribut darf in allen Methoden der Klasse ganz normal verwendet werden. Aber jede Änderung

des statischen Attributs wirkt sich für alle Instanzen aus!

Beispiel

Bei dem Klassenmodell für ein Radio werden die statischen Attribute `minFrequenz`, `maxFrequenz` usw. innerhalb der Klassendefinition in der Datei `radio.h` bekannt gemacht:

```
public class Radio {
    private final static double fmin = 87.6;
    ...
};
```

Klassenspezifische Konstanten wie oben sind ein häufiges Beispiel für statische Attribute.

Ein anderes Beispiel für ein statisches Attribut wurde im Abschnitt 5.2 gezeigt.

Soll außerhalb einer Methode auf ein statisches Attribut zugegriffen werden, so kann dies über `Klassenname.attribut` geschehen.

Statische Methoden

Statische Methoden sind Methoden, die nicht für eine einzelne Instanz der Klasse aufgerufen werden, sondern für die Klasse an sich. In statischen Methoden steht daher die Referenz `this` nicht zur Verfügung. Aus diesem Grund kann in einer statischen Methode auch nur auf statische Attribute uns andere statische Methoden der Klasse zugegriffen werden.

Die Deklaration einer statischen Methode erfolgt ebenfalls mit dem Schlüsselwort `static`:

```
public class Klassenname {
    private static int attribut1;
    public static int getAttribut1() {
        return attribut1;
    }
}
```

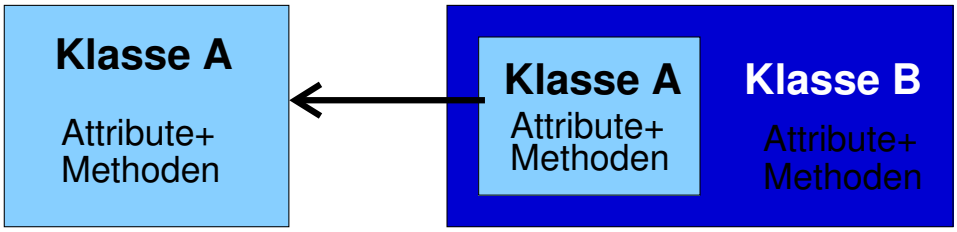
Der Zugriff auf diese statische Methode erfolgt dann mit

```
Klassenname.getAttribut1()
```

7 Vererbung

7.1 Grundsätzliches

In allen objektorientierten Sprachen – also auch in Java – gibt es die Möglichkeit der Vererbung. Vererbung bedeutet, dass eine neue Klasse die Attribute und Methoden einer anderen Klasse erbt, ohne dass diese neu definiert werden müssen.



Die Syntax beim Vererben in Java lautet:

```
public class KlasseB extends KlasseA {
    ...
};
```

7.2 Vererbung und Konstruktoren

Beim Anlegen einer Instanz der Kindklasse wird *vor* dem Konstruktor der Kindklasse der Standard-Konstruktor der Elternklasse aufgerufen.

Beispiel

```
public class Base {
    public Base() {
        System.out.println("Konstruktor von Base");
    }
}

public class Ext extends Base {
    public Ext() {
        System.out.println("Konstruktor von Ext");
    }
}
```

Dieses Beispiel ergibt als Ausgabe:

```
Konstruktor von Base
Konstruktor von Ext
```

Ein Problem tritt auf, wenn eine Instanz der Kindklasse angelegt werden soll, aber der Konstruktor der Elternklasse mit Parameter(n) aufgerufen werden soll. Dies gilt insbesondere dann, wenn gar kein Standardkonstruktor der Elternklasse existiert.

In diesem Fall kann im Konstruktor der Kindklasse als erster Befehl mit **super**(*parameter*) angegeben werden, wie der Konstruktor der Elternklasse aufgerufen werden soll.

Beispiel

```
public class Base {
    public Base(int i) {
```

```
        ...
    }
}

public class Ext extends Base {
    public Ext(int i, int j) {
        super(i);
        ...
    }
};
```

7.3 Vererbung und Typumwandlung

Da eine Kindklasse alle Attribute und Methoden der Elternklasse erbt, ist jede Instanz der Kindklasse automatisch auch eine Instanz der Elternklasse.

Daher ist bei Objektreferenzen folgender Quelltext zulässig:

```
Ext e = new Ext();
Base b = e;
```

Die hier implizit durchgeführte Typumwandlung von der Kind- zur Elternklasse wird als *up cast* bezeichnet. Dieser ist problemlos möglich.

Eine Typumwandlung von der Eltern- zur Kindklasse nennt man *down cast*. Dieser ist nur dann zulässig, wenn die Referenz vom Typ der Elternklasse in Wirklichkeit auf eine Instanz der Kindklasse verweist. Ein down cast muss explizit mit dem Cast-Operator durchgeführt werden.

7.4 Polymorphie

In Java können Methoden überladen werden, d. h. sie können mit identischer Signatur sowohl in der Eltern- wie in der Kindklasse definiert sein.

Beim Auswählen, welche Implementierung einer Methode gewählt wird, gilt in Java das Prinzip der späten Bindung. Die Auswahl der Methode findet somit erst zur Laufzeit statt und hängt davon ab, von welcher Klasse das Objekt instanziert wurde.

7.5 Abstrakte Klassen und Interfaces

Häufig steht für Methoden in der Elternklasse noch kein Code fest, da dieser erst in den Kindklassen implementiert werden soll. In diesem Fall kann die Methode als abstrakt deklariert und der Code weggelassen werden.

Abstrakte Methoden sind nur in Klassen zulässig, die als abstrakt deklariert sind. Die allgemeine Form lautet:

```
public abstract class Klassenname {
    public abstract int methode();
}
```

Von abstrakten Klassen können keine Instanzen angelegt werden!

Eng verwandt mit abstrakten Klassen sind Interfaces. Diese enthalten nur abstrakte Methoden und dürfen nur statische Attribute besitzen. Ein Vergleich von abstrakten Klassen findet sich in Tabelle 3.

Tabelle 3: Vergleich: Abstrakte Klassen und Interfaces

	Abstrakte Klassen	Interfaces
Methoden	mindestens eine abstrakte Methode, aber zusätzlich nicht-abstrakte erlaubt	nur abstrakte Methoden erlaubt
Attribute	beliebig	nur statisch
Vererbung	eine Klasse kann nur von einer einzigen abstrakten Klasse erben	eine Klasse darf beliebig viele Interfaces implementieren (und zusätzlich von einer Klasse erben)

Der Vorteil von Interfaces liegt darin, dass in Java eine Klasse zwar nur von maximal einer Klasse erben, aber beliebig viele Interfaces implementieren darf. Hierbei wird von Java bereits ein Satz von Interfaces definiert, z. B. `Cloneable`, `Comparable` oder `Iterable`.

Die allgemeine Syntax lautet:

```
public class Ext extends Base implements Interface1, Interface2 {
    ...
}
```