

Java — ein Überblick in 21 Minuten

Berthold Hoffmann • Universität Bremen

Einleitung

Dieses kurze Papier soll einen ersten Einblick in die Konzepte der objektorientierten Sprache Java geben, die für netzwerkweites Programmieren entworfen wurde. Wie in dem Lehrbuch Watt (1990) steht dabei die *Semantik* im Vordergrund, und syntaktische Fragen werden weitgehend vernachlässigt.

Der Überblick basiert auf Version 1.0 Beta der Sprachdefinition von SUN Microsystems (1995) und dem Handbuch von Flanagan (1996), das Version 1.0 der Sprache beschreibt, und ist vorerst durch keinerlei praktische Arbeit mit Java getrübt.

Entwurfsziele

SUN Microsystems schreiben der von ihnen selbst entwickelten Sprache Java bescheiden folgende Eigenschaften zu:

- einfach
- objektorientiert
- verteilt
- interpretiert
- robust
- sicher
- Architektur-neutral
- portabel
- effizient
- nebenläufig (*multi-threaded*)
- dynamisch

Es muß sich erst noch zeigen, ob das alles stimmt, jedenfalls kann das schon als ein erster Hinweis auf die Ausrichtung der Sprache genommen werden.

Konzepte von Java

Wir geben einen knappen Überblick über die Konzepte der Sprache. In der Reihenfolge richten wir uns nach Watt (1990), indem wir zunächst die funktionale Facette (Werte), dann die imperative Facette (Speicher) und endlich Bindungen betrachten, bevor wie Abstraktionen untersuchen sowie die fortgeschrittenen Konzepte des Programmieren im Großen: Kapselung, Typisierung und Ausnahmen. Nebenläufigkeit wird (noch) nicht vollständig beschrieben.

Werte

Werte sind die Daten, die in einem Java-Programm berechnet werden können, indem Ausdrücke ausgewertet werden. Java ist *getypt*: Die Menge aller Werte wird in Datentypen eingeteilt. Die Zuordnung zu einem Datentyp ist größtenteils *statisch*, d. h. schon für die Größen im Programm gegeben und wird vom Übersetzer überprüft. Jeder Datentyp ist gegeben durch eine Wertemenge (wie `int`), und die dazugehörenden Operationen wie `+`).

Datentypen. Die *eingebauten einfachen Datentypen* ähneln denen von C; dazu kommt noch der Typ `boolean` für die Wahrheitswerte `true` und `false`.

Die Typen `byte`, `short`, `int` und `long` enthalten ganze Zahlen verschiedener Genauigkeiten. Die Typen `float` und `double` enthalten Fließpunktzahlen verschiedener Genauigkeit. Der Typ `char` enthält Zeichen in der universellen *Unicode*-Codierung, die alle weltweit benutzten Alphabete darstellen kann (auch Chinesisch).

Die Wertemengen (Größen) der Datentypen sind alle *exakt* vorgegeben, um Plattform-übergreifende Portabilität zu erreichen. *Unicode*s können übrigens auch für Bezeichner im Programmtext selbst verwendet werden, zur Freude für die chinesischen Kollegen.

Nur wenige *zusammengesetzte Datentypen* sind eingebaut:

- *Felder* sind flexibel, so daß Variablen Felder beliebiger Größe zugewiesen werden können. Sie werden wie Objekte als Referenzen abgespeichert, bei der Vereinbarung explizit erzeugt, und automatisch wieder freigegeben. Sonst ähnelt die Syntax der von C. (Die in C so beliebte Verwendung von Zeigern für Feldelemente ist aber nicht möglich.)
- *Zeichenketten (strings)* sind Objekte vordefinierter Klassen, für deren Benutzung es praktische Schreibabkürzungen gibt, u.a. die Verkettungsoperation `+`. Es wird zwischen zwei Arten von Zeichenketten unterschieden:
 - `String` für unveränderliche Zeichenketten.
 - `StringBuffer` für veränderliche Zeichenketten.

Es gibt weder Mengen, noch Verbunde (*record*, *struct*) oder disjunkte Vereinigungen (*union*). Alle Benutzerdefinierten Datentypen werden als *Klassen* definiert. Die Objekte (Werte) einer Klasse sind Zeigern auf Verbunde, wobei die Zeiger vorm Benutzer verborgen bleiben, damit sie nicht explizit manipuliert werden können.

Ausdrücke. Dieser Teil der Sprache ähnelt C sehr stark; weil der Typ `boolean` hinzugenommen wurde, bezeichnen die Operatoren wie `&`, `^`, `|`, `&&` und `||` jedoch sowohl logische als auch die von C geläufigen ganzzahligen Operationen.

Die Auswertungsreihenfolge von Operationen und Funktionen in Ausdrücken ist strikt vorgeschrieben: Vor dem Aufruf einer Funktion oder Operation werden alle Argumente von links nach rechts ausgewertet. Ausgenommen davon sind die bedingte Auswahl `? :`, wobei `B ? T : E` dem Ausdruck

`if B then T else E`

entspricht. Hier wird nur das erste Argument, und dann entweder nur das zweite oder nur das dritte ausgewertet. Das *bedingte Und* `&&` und das *bedingte Oder* `||` lassen sich dann so definieren:

$$\begin{aligned} E \ \&\& \ F &\equiv E \ ? \ F : \ \mathbf{false} \\ E \ || \ F &\equiv E \ ? \ \mathbf{true} : \ F \end{aligned}$$

Speicher

Java benutzt – wie alle imperativen und objektorientierten Sprachen – das Konzept des *Speichers*: Variablen sind benannte Speicherzellen, in deren Inhalt mit Befehlen gesetzt und überschrieben werden kann.

Variablen. In Java sind Variablen entweder Komponenten von Objekten, oder lokal in Methoden.

Hinsichtlich der Lebensdauer unterschieden sie sich so:

- *Klassenvariablen* werden mit dem Zusatz `static` vereinbart und sind *global*; sie werden einmal beim Laden einer Klasse angelegt, sind allen Objekten der Klasse gemeinsam, und existieren für den Rest des Programms. (Sie sind aber nicht unbedingt überall *sichtbar*, siehe unter Bindungen.)
- *Instanzvariablen* sind *dynamisch*; sie werden beim Instanzieren eines Objekts angelegt und gelten solange, wie dieses Objekt noch benutzt wird. Der von einer Instanzvariablen belegte Speicherplatz kann danach durch automatische Speicherbereinigung (*garbage collection*) wieder freigegeben werden.
- Lokale Variablen in Methodenrümpfen verschwinden nach Abarbeitung der Methode, sind also *lokal* wie in imperativen Sprachen. (Wenn diese Variablen Objekte sind, existieren sie weiter, solange es noch weitere Verweise auf sie gibt.)

[*Persistente* Variablen sind anscheinend für eine spätere Entwicklungsstufe von Java vorgesehen.]

Feldvariablen und Felder von Objekten können auch *selektiv* überschrieben werden; bei Zuweisungen von Objekten werden immer nur die *Referenzen* auf die Objekte kopiert (analog bei Feldern); so können Aliase entstehen. Die allgemeine Methode `clone` (der Wurzelklasse `Object`) erlaubt es aber auch, alle Komponenten eines Objektes zu kopieren (aber nicht rekursiv die Komponenten der Komponenten).

Befehle. Die Kontrollstrukturen von Java sind die von C (bis auf Sprünge). Numerische Fallunterscheidungen müssen also mit Sprungtabellen (`switch`) implementiert werden, und die Zählschleife behält eine etwas bizarre Syntax.

Java ist ausdrucksorientiert wie C, macht also keinen (großen) Unterschied zwischen Ausdrücken und Befehlen. Insbesondere werden die Zuweisungsoperatoren als Operationen aufgefaßt, nicht als Befehle.

Bindung

Unter Bindung versteht man die Benennung von Programmgrößen (wie Objekten, Klassen, Methoden) mit Namen (Bezeichner = *identifizier*), die in *Vereinbarungen* getroffen werden und bestimmte *Gültigkeitsbereiche* haben.

Gültigkeitsbereiche. Es gibt vier Ebenen der Gültigkeit von Bindungen:

- das Programm
- ein Paket
- eine Klassen- oder Schnittstellendefinition (evtl. mit deren Unterklassen)
- eine Methodendefinition (oder ein Block in einer Methodendefinition oder eine `for`-Schleife)

Das Programm ist hierbei die Menge aller zur Verfügung stehenden Klassen, und Pakete sind Übersetzungseinheiten, die beliebige Ansammlungen von miteinander “befreundeten” Klassen enthalten, von denen höchstens eine öffentlich ist.

Klassendefinitionen und Schnittstellendefinitionen können öffentlich (`public`) oder für das Paket getroffen werden, in dem sie stehen.

Innerhalb von *Klassendefinitionen* werden die Eigenschaften einer Klasse *C* festgelegt. Als Gültigkeit für die Eigenschaften kann dabei angegeben werden:

- allgemein (`public`)
- das ganze Paket, in dem die Klasse definiert ist (*default*)
- das ganze Paket, und alle weiteren Unterklassen (`protected`) der Klasse *C*
- die Klasse *C* mit ihren Unterklassen (`private protected`)

- nur die Klasse `C` selbst (**private**)

public-Variablen sind also global und auch *global veränderbar*!

Die innerhalb von *Schnittstellendefinitionen* festgelegten Eigenschaften haben die gleiche Gültigkeit wie die Schnittstelle als Ganze.

In *Methodendefinitionen* können lokal nur Variablen und Konstanten vereinbart werden, die aber auch lokal in jeder geklammerten Anweisung (`{...}`) und jeder **for**-Schleife.

Namensüberdeckung. Variablen einer Klasse *überschatten* Variablen desselben Namens in einer Überklasse. Methodennamen überschatten einander nicht, sondern *überschreiben* einander (siehe Typisierung).

Variablen in inneren Blöcken (d. h. in Methodenrümpfen) überschatten vermutlich Vereinbarungen desselben Namens in äußeren Blöcken. (Dieser Punkt konnte nicht geklärt werden.)

Globale Namen. In Java-Programmen können *absolute Namen* für Klassen und Methoden verwendet, die irgendwo im Internet stehen. Es gibt eine Konvention für das Umsetzen von Internet-Adressen und Verzeichnisnamen in Qualifizierungen von Java-Namen.

Beispielsweise wäre ein Paket `graph.class` in einem Verzeichnis `lib` des Autors (das nach Konvention eine öffentliche Klasse `graphs` definieren sollte), unter dem absoluten Namen

```
DE.Uni-Bremen.informatik.hof.lib.graph
```

im gesamten Internet bekannt und benutzbar.

Relative Namen nehmen das Verzeichnis des Pakets als Anfang, in dem dieser Name steht; auch hier können Klassen in Unterverzeichnissen über qualifizierende Präfixe angesprochen werden.

Vereinbarungen. *Typvereinbarungen* gibt es eigentlich nicht. Jede Klasse definiert jedoch einen Typ, und wird benutzt, um den Typ von Variablen, Konstanten oder Parametern anzugeben.

Variablenvereinbarungen initialisieren immer (zur Not implizit mit einem neutralen Wert wie `0`, `\u0000` oder `null`). Variablenvereinbarungen mit dem Zusatz **final** kennzeichnen *Konstanten*, deren Wert nicht mehr geändert werden darf. Im allgemeinen braucht dieser Wert nicht schon zur Übersetzungszeit festzustehen (wohl aber bei Konstanten in einer Schnittstellendefinition).

Operatoren können (noch) nicht vom Benutzer definiert werden.

Methoden unterteilen sich in Klassenmethoden (**static**), die nur auf Klassenvariablen zugreifen können, und in *Instanzenmethoden*, die auch implizit auf die Instanzvariablen der Klasse der Klasse zugreifen können. *Konstruktor-methode* erlauben es, ein Objekt auf verschiedene Arten zu initialisieren. Mit einer (nicht statischen) Abschlußmethode `finalize` kann angegeben werden, was vor dem Freigeben eines Objektes bei der automatischen Speicherbereinigung getan werden soll.

Abstraktion

Konstrukte zur Abstraktion erlauben, Programmteile zu benennen und zu parameterisieren, um sie dann mit verschiedenen Argumenten beliebig oft im Programm zu verwenden. Prozeduren und Funktionen sind die geläufigsten Arten der Abstraktion.

Funktionen und **Prozeduren** werden *Methoden* genannt, wie in OO-Sprachen üblich. Sie liefern immer einen Wert (wie in C). Bei "eigentlichen" Prozeduren wird der Ergebnistyp mit `void` vereinbart.

Parameter sind einfache Werte oder Objekte. Parameter werden immer als Wert übergeben, also entweder als Kopie eines einfachen Wertes oder als Kopie des Verweises auf ein Objekt (oder ein Feld). Objektparameter entsprechen also den Referenzparametern imperativer Sprachen und können benutzt werden, um Ergebnisse einer Methode zu liefern. Methoden können also nur Objekte *verändern*, nicht Parameter einfachen Typs.

Methoden (also Funktionen oder Prozeduren) können selbst nicht als Parameter übergeben werden.

Kapselung

Es gibt drei Konstruktionen zur Kapselung von Bindungen:

- *Pakete* sind Übersetzungseinheiten, in denen miteinander befreundete Klassen und Schnittstellen zusammengefaßt werden. Nach außen ist nur die einzige öffentliche Klasse oder Schnittstelle des Pakets sichtbar. Pakete können andere Pakete *importieren*, d.h. die öffentliche Klasse eines anderen Paketes unter dem letzten Teil ihres globalen Namens benutzbar machen.
- *Klassen* kapseln die Eigenschaften von gleichartigen Objekten. Die Objekte können durch Instanziierung der Klasse erzeugt werden.
- *Schnittstellen* kapseln Anforderungen an Klassen hinsichtlich der von ihnen zu definierenden Methoden. Sie definieren nur die Signaturen dieser Methoden und ggf. einige statisch bestimmte Konstanten.

Generische Klassen sind nicht möglich. Felder, die in einer statisch getypten Sprache eigentlich als generische Klassen

aufgefaßt werden müßten, sind in der Sprache fest eingebaut, sicher auch aus Gründen der Verträglichkeit mit C++.

Typisierung

Für Variablen und Konstanten gibt es jeweils nur eine gültige Vereinbarung, d.h. sie müssen monomorph sein.

Überladen gibt es (noch) nur für eingebaute Operatoren, aber allgemein für Methoden. Die überladenen Methoden müssen sich in Anzahl oder Typ der Parameter unterscheiden. Methodendefinitionen können auch überschrieben werden, siehe unter Vererbung.

Anpassungen sind implizite Konversionen von Typen, die in bestimmten Kontexten vorgenommen werden können. Die einfachen Typen sind fast ausnahmslos aneinander anpaßbar (außer `char` auf `byte`), mit der wichtigen Ausnahme `boolean`. *Ausweiten* ist dabei bei Zuweisungen immer möglich, z. B. von `int` an `double`. *Einschränkungen* (*casts*) sind nur durch explizites Angeben einer Umwandlung (eines sogenannten *casts*) möglich.

Polymorphie gibt es nicht.

Vererbung bezeichnet die Möglichkeit, Typen zu *spezialisieren*, so daß Methoden des Basistyps auch auf die Spezialisierungen angewendet werden können. Damit wird eine Untertyprelation geschaffen.

In Java gibt es einfaches Vererben (*single inheritance*), d. h. eine Klasse kann als Spezialisierung *einer* anderen definiert werden. Eine Klasse kann also von ihrer Basis-klasse erben. Die Klassen `String` und `StringBuffer` sowie alle Benutzer-definierten Klassen werden als Unter-klasse der Wurzelklasse `Object` aufgefaßt.

Mehrfachvererbung wird nicht allgemein unterstützt. Dies wird mit der schwierigen Implementierung begründet. Als Alternative werden Schnittstellen (*interface*) angeboten.

Eine Schnittstellendefinition spezifiziert Signaturen von Methoden (und statisch auswertbare Konstanten). Eine Schnittstelle kann *mehrere* Schnittstellen erweitern (*extend*), und eine Klasse kann *mehrere* Schnittstellen implementieren.

Da Schnittstellen selbst keine Implementierung haben, können sie die Mehrfachvererbung nicht ersetzen. Sie ähneln (eingeschränkten) vertagten Klassen (*deferred classes*) und können in erster Linie dazu dienen, Überladen von Methoden zu erreichen, indem die zu überladenen Methoden in einer Schnittstelle zusammengefaßt werden und ansonsten nicht in Beziehung stehende Klassen so vereinbart werden, daß sie diese Schnittstelle implementieren.

Überschreiben (*overriding*) bezeichnet die Möglichkeit, das Methoden einer Klasse Methoden einer (direkt oder indirekt) geerbten Oberklasse ersetzen können. In diesem Fall müssen beide Methoden die gleiche Anzahl und Typen von Parametern haben; der Ergebnistyp der überschreibenden Methode muß an den der überschriebenen Methode *zuweisbar* (d. h. eine Erweiterung dieses Typs) sein. Die Gültigkeitsbereich der überschreibenden Methode muß den der überschriebenen enthalten.

Finale Klassen können nicht mehr durch Vererbung erweitert werden. Auch einzelne Methoden einer Klasse können als **final** vereinbart werden und dürfen dann nicht mehr überschrieben werden. Diese Angabe vereinfacht die Methodenaufrufe, weil dann kein dynamisches Binden notwendig ist.

Ablaufsteuerung

Java kennt dieselben Auswege wie C, mit denen vom normalen Kontrollfluß abgewichen werden kann:

- Unterbrechungen (**break**) erlauben den Abbruch einer Schleife oder einer Sprungtabelle (**switch**)
- Fortsetzungen (**continue**) erlauben die Unterbrechung des aktuellen Schleifendurchlaufes.
- Rückgabeanweisungen (**return**) beenden Methoden und definieren ggf. einen Ergebniswert.

Ausnahmen (*exceptions*) werden von Bibliotheksmethoden geworfen. Der Benutzer kann neue Ausnahmen angeben und in Methoden werfen.

Mit der Anweisung

```
throw (e)
```

kann der Benutzer eine Ausnahme wecken (oder werfen). Dabei ist `e` ein Objekt, typischerweise von einem Untertyp des vordefinierten Typs `Exception`. Wecken einer Ausnahme beendet die normale Ausführung, bis ein Block kommt, der diese Ausnahme behandelt (oder einfängt, **catch**).

Mit **try** werden Blöcke eingeleitet, die eine Ausnahmebehandlung enthalten.

Die Behandlung wird mit Anweisungen der Form

```
catch (e) Block
```

spezifiziert, wobei die aufgetretene Ausnahme an `e` zuweisbar sein muß, damit *Block* ausgeführt wird. Jede Ausnahmebehandlung kann mit einer Anweisung

```
finally Block
```

abgeschlossen werden. Dann wird *Block* immer ausgewertet, wenn eine Ausnahme auftrat, auch wenn sie nicht zu den vorher mit `catch` abgefangenen paßte.

Nebenläufigkeit

Java hat Nebenläufigkeit in Form von leichten Prozessen (*light-weighted processes*) oder Fäden (*threads*).

Es gibt auch Pakete, die verteilte Ausführung ermöglichen.

Java und C

Java übernimmt nur kleinere Teile der Sprache C; viele der bekannten Unsicherheiten werden ausgeräumt:

- Übernahme der Konzepte für das “Programmieren im Kleinen” (Vereinbarungen, Anweisungen und Ausdrücke)
- Verbannen unsicherer maschinennaher Konzepte: Zeiger, Sprünge)
- Verzicht auf den Präprozessor zugunsten echter Modularisierung

Java und C++

Die Ähnlichkeit ist eher oberflächlich, denn die Syntax (und teilweise auch die Terminologie) für die objektorientierten Erweiterungen von C++ gegenüber C wird übernommen, die Semantik ist aber erheblich reiner:

- Alle Benutzer-definierten Typen sind Klassen (ein Reinheitsgebot für OO-Sprachen)
- Zeiger werden verborgen
- Aus Sicherheitsgründen gibt es eine automatische Speichereinigung (*garbage collection*)
- Es gibt keine Mehrfachvererbung. Dies soll durch Schnittstellen (*interfaces*) kompensiert werden.

Java auf dem Weg ins objektorientierte Glück?

Bertrand Meyer (1988) beschreibt sieben Kriterien für den “weg ins objektorientierte Glück”. Inwieweit folgt Java Meyer auf diesem Weg?

1. **Objekte.** Programme werden anhand der Daten und nicht anhand der Funktionen modularisiert.
2. **Datenabstraktion** (*information hiding*). Objekte sind abstrakte Datentypen; Variablen bleiben verborgen.
3. **Automatische Speicherverwaltung** (*garbage collection*). Unbenutzte Objekte werden automatisch wieder freigegeben; es gibt kein explizites Freigeben von Speicher.
4. **Klassen.** Alle zusammengesetzte Typen sind Moduln, und alle Moduln definieren Typen. Objekte einer Klasse können in beliebiger Zahl geschaffen werden.

5. **Vererbung.** Klassen können Erweiterungen oder Beschränkungen anderer Klassen sein.

6. **Polymorphie und dynamisches Binden.** Größen können sich auf Objekte mehrerer Klassen beziehen; Operationen können in verschiedenen Klassen verschieden realisiert sein.

7. **Mehrfache und wiederholte Vererbung.** Eine Klasse kann von mehreren Klassen, und mehrmals von einer Klasse erben.

Java folgt diesem Weg ziemlich weit: 1, 3, 4, 5, 6 sind ohne Einschränkungen erfüllt. Im Gegensatz zu 2. kann man aber globalen Variablen definieren, wenn man unbedingt will. Mehrfachvererbung gibt es auch nicht.

Zusammenfassung

Java kann als eine “ziemlich reine” objektorientierte Sprache bezeichnet werden (im Gegensatz zu C++), in der man eigentlich objektorientiert programmieren *muß*. Dazu sind im Wesentlichen auch alle Konzepte in adäquater Form vorhanden.

Einige trotzdem bestehenden Schwachpunkte werden im folgenden angesprochen.

Globale Variablen (**public**) sind auch *global veränderbar*. Dies verletzt den Anspruch der *Sicherheit*. Oft wird das damit begründet, daß der globale Zugriff auf die Werte einer verborgenen Variable mit einer Funktion oder Prozedur zu ineffizient sei, aber ein lesender Export wie in Oberon oder Eiffel zeigt, daß dies nicht zwingend der Fall ist.

Schnittstellen sind eigentlich nur dafür gut, Überladen unter dem Dach der Vererbung zu beschreiben. *Mehrfachvererbung* kann damit nicht ersetzt werden. (Eine andere Frage ist, wie wichtig die Mehrfachvererbung ist.)

Das *Typsystem* hätte etwas einheitlicher gestaltet werden können, wenn auch die eingebauten einfachen Datentypen eingegliedert worden wären, und das dort definierte Ausweiten als die Verträglichkeit von Typen mit ihren Untertypen aufgefaßt worden wäre.

Daß die *Befehle* ausgerechnet aus einer so veralteten und vergleichsweise barocken Sprache wie C entlehnt werden, macht die Sprache sicher nicht einfacher, aber natürlich weniger gewöhnungsbedürftig.

Literatur

- David FLANAGAN (1996). *Java in a Nutshell*. Sebastopol, California: O'Reilly & Associates. [30 DM]
- Laura LEMAY, Charles L. PERKINS (1996). *Teach Yourself Java in 21 Days*. Indianapolis, IN: Sams.net Publishing [85 DM]
- Bertrand MEYER (1988): *Object-Oriented Software Construction*. Hemel Hempstead: Prentice Hall International Series in Computer Science.
- SUN Microsystems (1995): *The Java Language Specification, Version 1.0 Beta*. Mountainview, 30.10.1995.
- David A. WATT (1990) *Programming Language – Concepts and Paradigms*. Hemel Hempstead: Prentice Hall International Series in Computer Science. 332 Seiten. [Bibliothek: 19 h inf 330 e/290] *Deutsche Fassung*: Programmiersprachen — Konzepte und Paradigmen (bearbeitet von Berthold HOFFMANN), Hanser-Verlag München Wien (1996).