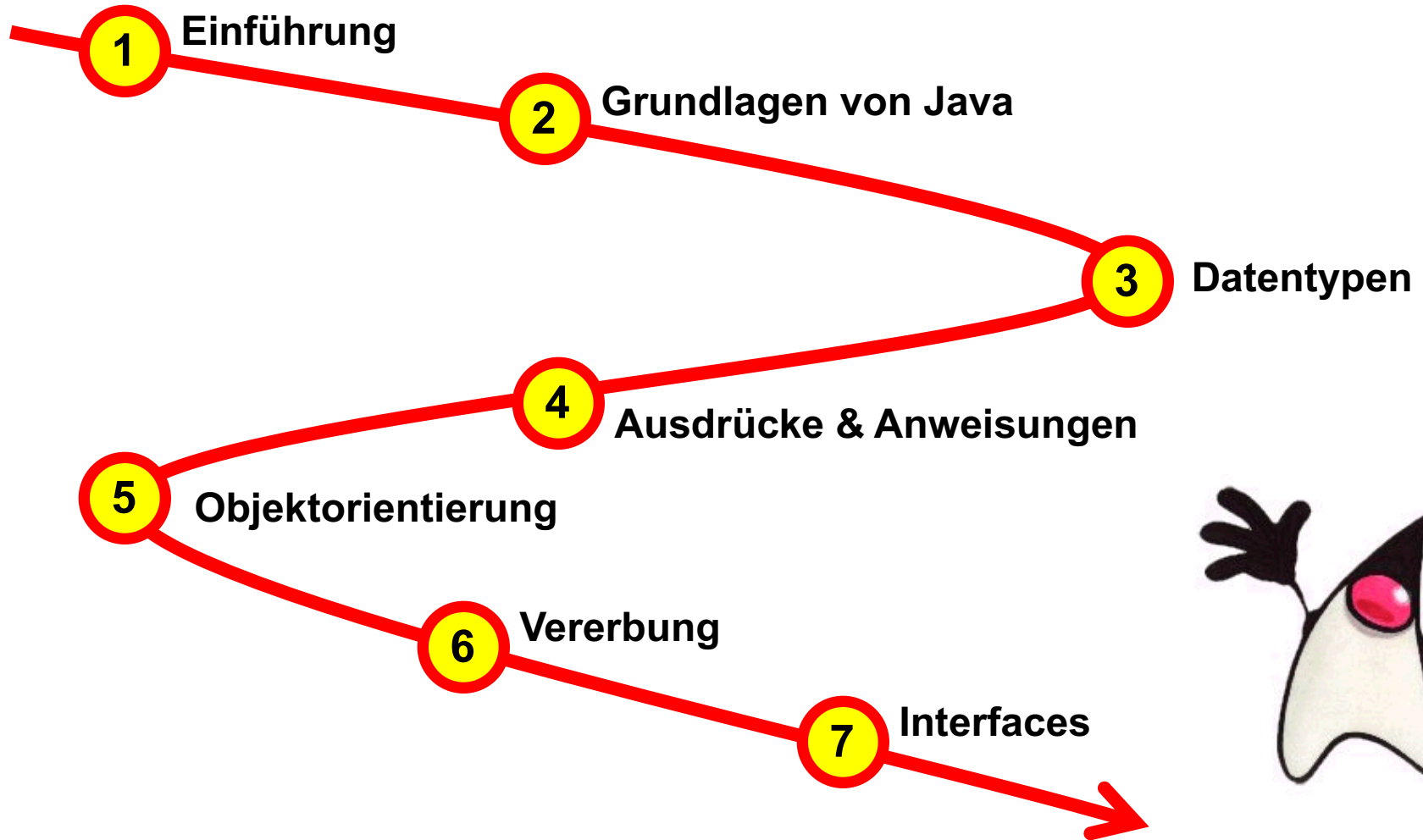


Programmierung 1

Michael Lang



Themenüberblick

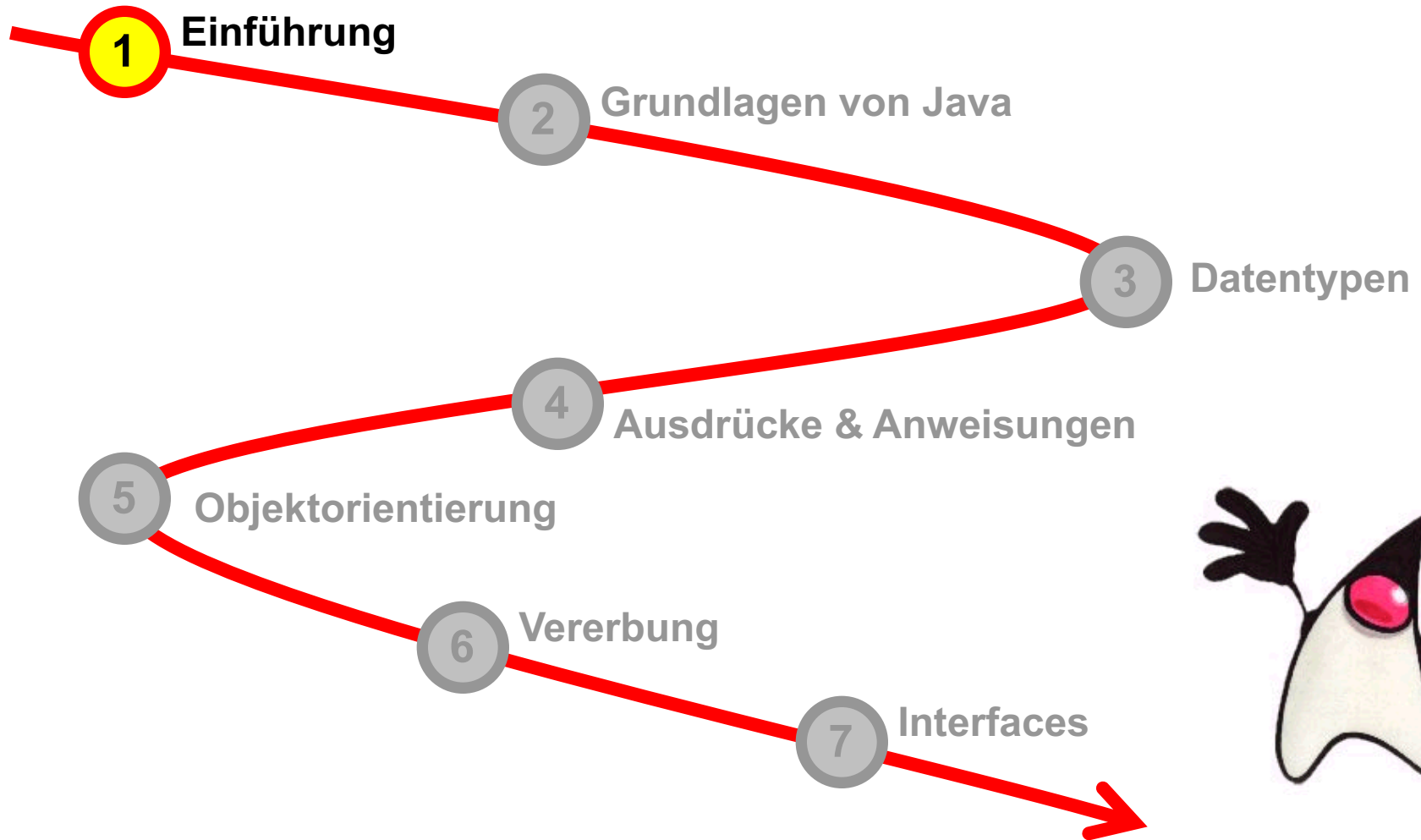




Programmierung 1

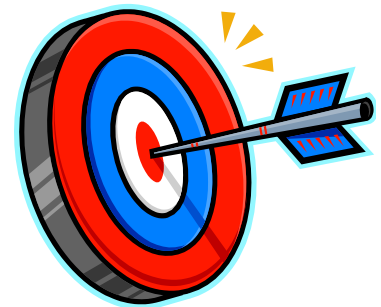
Kapitel 1
Einführung

Themenüberblick



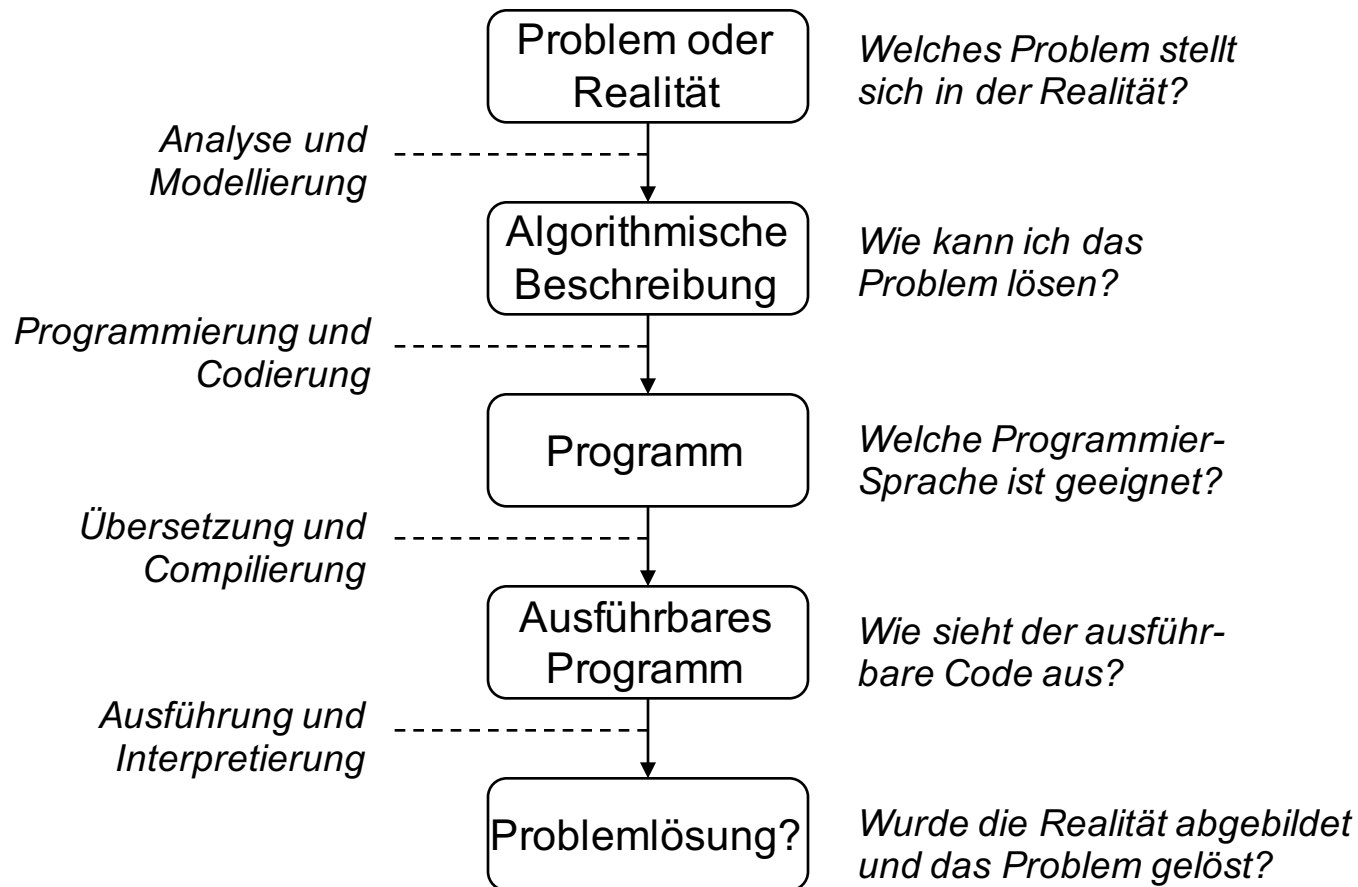
Lernziele

- Sie können den Begriff Algorithmus definieren
- Sie kennen die Eigenschaften und Bestandteile von Algorithmen
- Sie können die Grundbegriffe der Programmierung nennen und einsetzen
- Sie kennen unterschiedliche Darstellungsformen von Algorithmen
- Sie können einfache Algorithmen in Form von Pseudocode, Programmablaufplänen und Struktogrammen darstellen



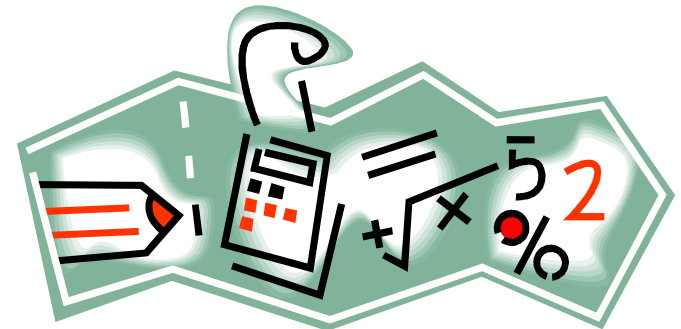
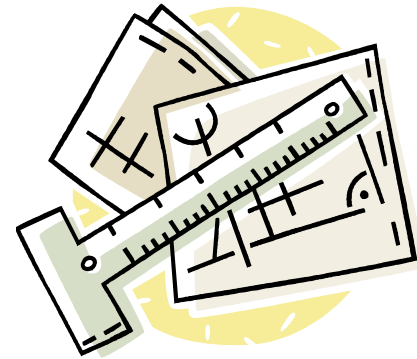
Ziel der Programmierung

Umsetzung eines gegebenen oder selbstentwickelten Algorithmus in ein lauffähiges Computerprogramm



Beispiele für Algorithmen

- Bedienungsanleitungen
- Bauanleitungen
- Kochrezepte
- mathematische Problemstellungen
- Such- und Sortieralgorithmen



Definition des Algorithmusbegriffs

Duden

Al|go|rith|mus, der; , ...men [mlat. algorismus = Art der indischen Rechenkunst, in Anlehnung an griech. arithmós = Zahl entsteht aus dem Namen des pers.- arab. Mathematikers Al-Hwarizmi, gest. nach 846] (Math., Datenverarb.): Verfahren zur schrittweisen Umformung von Zeichenreihen; Rechenvorgang nach einem bestimmten [sich wiederholenden] Schema.

Informatik

Ein Algorithmus ist eine präzise (d.h. in einer festgelegten Sprache abgefasste) endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer (Verarbeitungs-)Schritte.

Beispiel: Ein Kochrezept

Zutaten

500 g Hackfleisch vom Rind
2 große Zwiebeln, fein gehackt
4 Stangen Staudensellerie, fein gehackt
150 g Speck, fein gehackt
2 Zehen Knoblauch, fein gehackt
400 g geschälte Tomaten
300 ml Fond (Bratenfond)
1 Paprikaschote, rot, klein gewürfelt
1 Gewürznelke
1 Lorbeerblatt
½ TL Oregano
½ TL Muskat (gemahlen)
3 EL Öl (Oliven)
40 g Butter
150 ml Wein, rot, trocken
1 Chilischote, zerkleinert, getrocknet

Zubereitung

Butter und Öl in einer Pfanne (besser Bräter) erhitzen. Zwiebeln, Sellerie, Speck und Paprika gut anbraten. Die Sachen aus der Pfanne nehmen und zur Seite stellen. Den Knoblauch leicht anrösten und dann das Hackfleisch dazugeben, alles gut anbraten. Die Sachen wieder dazugeben und alles ca. 10 Min. köcheln lassen. Jetzt alle übrigen Zutaten und Gewürze dazugeben und das Ganze ca. 45 Min. (bei geschlossenem Deckel) köcheln lassen (gelegentlich abschmecken und ggf. nachwürzen).
Über die fertig gegarten Spaghetti geben.



Probleme bei diesem Beispiel

- Zutaten sind bereits vorbereitet
- Spaghetti fehlen bei Zutaten
- Beschreibung für Zubereitung der Spaghetti fehlt
- ungenaue Aussagen
 - ◆ fein gehackt
 - ◆ klein gewürfelt
 - ◆ erhitzen
 - ◆ gut anbraten
 - ◆ leicht anrösten
 - ◆ ...
- Pfanne (besser Bräter)



KURZ: Die Beschreibung lässt Raum für individuelle Entscheidungen und Interpretationen.

Verbesserungsmöglichkeiten des Rezepts

- komplette Eliminierung von individuellen Interpretationsspielräumen
- vollständige Beschreibung der Arbeitsschritte inkl. Vorbereitung und der Kochanweisung für die Spaghetti
- vollständige Angabe der Zutaten
- präzise Angaben bei den Aussagen
 - ◆ fein gehackt → gewürfelt, $2 \text{ mm} \leq \text{Kantenlänge} \leq 3 \text{ mm}$ (Zwiebeln und Speck)
 - ◆ fein gehackt → gewürfelt, $0,8 \text{ mm} \leq \text{Kantenlänge} \leq 1 \text{ mm}$ (Knoblauch)
 - ◆ klein gewürfelt → $4 \text{ mm} \leq \text{Kantenlänge} \leq 6 \text{ mm}$
 - ◆ erhitzen → $120 \text{ °C} < \text{Temperatur} < 125 \text{ °C}$
 - ◆ gut anbraten → Farbe der Zwiebeln entspricht dem RGB-Wert CC3300 
 - ◆ leicht anrösten → Farbe des Knoblauchs entspricht dem RGB-Wert CC6600 



Eigenschaften von Algorithmen

Terminierung

- bricht nach endlich vielen Schritten ab

Determinismus

- legt die „Wahlfreiheit“ fest
- deterministischer Ablauf
 - ◆ legt eindeutige Vorgabe der Schrittfolge der auszuführenden Schritte fest
- determiniertes Ergebnis
 - ◆ wird immer dann geliefert, wenn bei vorgegebener Eingabe ein eindeutiges Ergebnis geliefert wird - auch bei mehrfacher Durchführung mit denselben Eingabeparametern

Bestandteile von Algorithmen

- elementare Operationen (Ausdrücke und Anweisungen)
Berechne 5 plus 7
- sequenzielle Ausführung
Berechne 10 minus 3, dann multipliziere das Ergebnis mit 4
- parallele Ausführung
Du rechnest Aufgabe 1 und ich rechne Aufgabe 2
- bedingte Ausführung
Wenn Du Aufgabe 1 gelöst hast, dann beginne mit Aufgabe 2
- Schleife
Rechne Aufgabe 1, bis Du das richtige Ergebnis bekommst
- Unterprogramm
Rechne Aufgabe 1 anhand der Lösung auf Seite 106
- Variablen und Konstanten

Grundbegriffe der Programmierung

Ausdruck

- Kombination von **Operanden** und **Operatoren** als "Vorschrift" zur Berechnung eines **Werts**
- liefert immer einen Wert (Ergebniswert) ab
- Beispiel:
 $1 / x$

Anweisung

- Kombination von Ausdrücken und Methoden als "Vorschrift" zur Ausführung einer Aktion
- Beispiele:
 $x = 5$ Wertzuweisung
 $y = 1 / x$ Wertzuweisung
`print(x)` Ausgabeanweisung (Methodenaufruf "Drucke x")

Grundbegriffe der Programmierung

Sequenz

- bildet eine zeitliche Abfolge von Anweisungen
- einzelne Schritte werden durchnummeriert oder es wird zum Abschluss der Sequenz ein Semikolon gesetzt

Bedingte Anweisung

- es werden Bedingungen auf Ihre Richtigkeit geprüft
- für wahre und falsche Aussagen in der Bedingung können unterschiedliche Anweisungen ausgeführt werden

Schleifen

- bestimmte Anweisungen werden wiederholt, bis eine definierte Endbedingung erfüllt wird
- Unterscheidung in drei Schleifenarten



Grundbegriffe der Programmierung

Unterprogramme

- beinhaltet einen Teilalgorithmus
- dieser Teilalgorithmus kann in mehreren Algorithmen wieder verwendet werden

Variablen

- „Platzhalter“ für einen konkreten Wert
- sind von einem bestimmten Datentyp
- können ihren Wert ändern

Konstanten

- haben einen festen Wert
- sind von einem bestimmten Datentyp
- können ihren Wert NICHT ändern

Darstellungsformen von Algorithmen

Pseudocode

- nahe an den Konstrukten verbreiteter Programmiersprachen
- Verwendung spezieller englischer Begriffe aus dem Alltag
- Begriffe haben eine festgelegte Bedeutung

Programmablaufpläne

- genormt nach DIN 66001
- Ursprung in der linearen Programmierung
- nur für kleinere Programme geeignet (Übersichtlichkeit)

Nassi-Schneiderman-Diagramme (Struktogramme)

- Entstehung 1973
- Darstellung genormt nach DIN 66261 im Jahr 1985
- überwiegender Einsatz in der prozeduralen Programmierung

Pseudocode

Sequenz

- Alternative 1: Schritte werden durchnummeriert: 1, 2, 3, ...
- Alternative 2: Abschluss der Sequenz durch Semikolon
- Vorteil Alternative 1: Verfeinerung einzelner Schritte: 2.1, 2.2, ...

Bedingte Anweisung

- es werden Bedingungen auf Ihre Richtigkeit geprüft
- Alternative 1: **falls** Bedingung **dann** Schritt
- Alternative 2: **falls** Bedingung **dann** Schritt A **sonst** Schritt B

Schleifen

- kopfgesteuert: **solange** Bedingung wahr **führe aus** Schritte
- fußgesteuert: **wiederhole** Schritte **bis** Bedingung wahr
- Zählschleife: **wiederhole für** Zahlenbereich Arbeitsschritte

Beispiel: Pseudocode

Sequenz

- (1) Koche Wasser
- (2) Gib Kaffeepulver in Tasse
- (3) Fülle Wasser in Tasse

Bedingte Anweisung

falls Ampel rot oder gelb
 dann stoppe
 sonst fahre weiter

falls Ampel ausgefallen
 dann fahre vorsichtig weiter
sonst falls Ampel grün
 dann fahre weiter
sonst stoppe

Schleifen

solange Liste nicht erschöpft
 führe aus
 Gib nächste Zahl aus der
 Liste aus

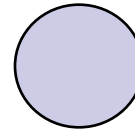
wiederhole
 Gib nächste Zahl aus der
 Liste aus
bis Liste erschöpft

wiederhole für 5 bis 10
 Gib nächste Zahl aus der
 Liste aus

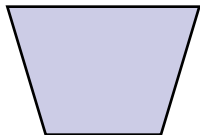
Programmablaufpläne



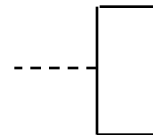
**Allgemeine Verarbeitung
(einschl. Ein- und Ausgabe)**



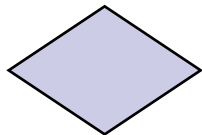
Verbindungsstelle



**manuelle Verarbeitung
(einschl. Ein- und Ausgabe)**



**Bemerkung
(erläuternder Text)**



Verzweigung

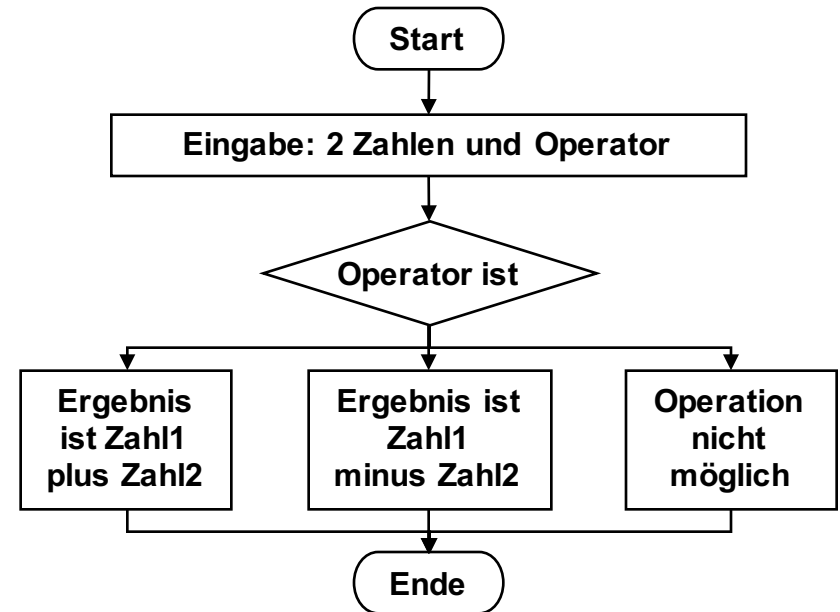
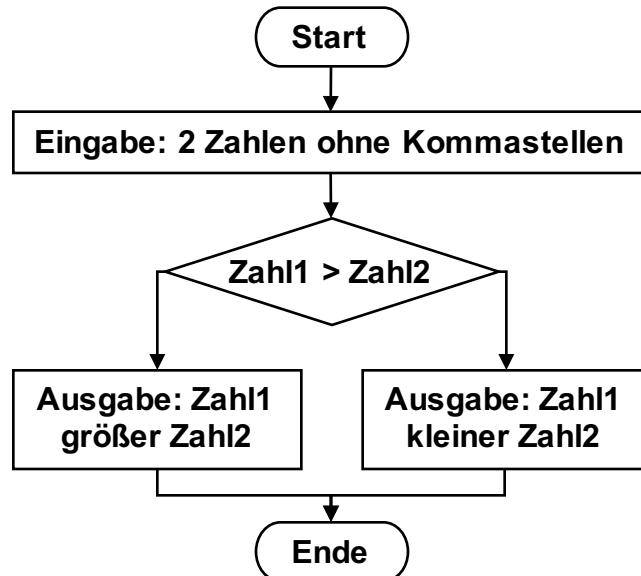
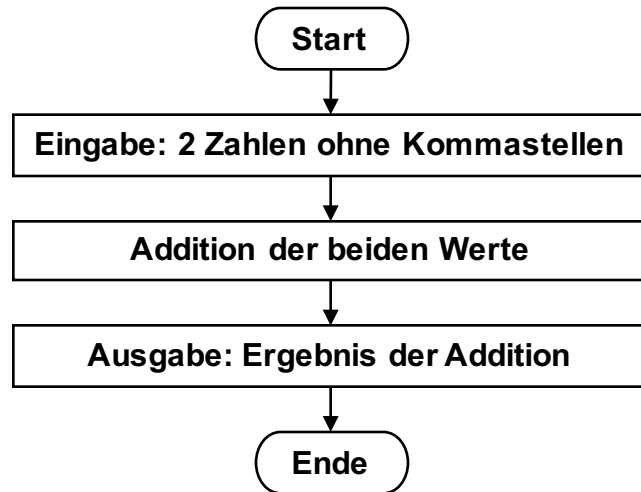


**Verbindung
(Verarbeitungsfolge)**



Grenzstelle (zur Umwelt)

Beispiel: Programmablaufplan



Struktogramme

Sequenz

Eingabe: 2 Zahlen ohne Kommastellen
Addiere die beiden Zahlen
Ausgabe: Ergebnis der Addition

Kopfgesteuerte Schleife

Zähler auf 0 setzen
Solange Zähler kleiner 100 ist
Zähler um 1 erhöhen

Bedingte Anweisung

Eingabe: 2 Zahlen ohne Kommastellen	
Zahl 1 > Zahl 2	
Ausgabe: Zahl 1 ist größer als Zahl 2	Ausgabe: Zahl 2 ist größer als Zahl 1

Fußgesteuerte Schleife

Zähler auf 0 setzen
Zähler um 1 erhöhen
bis Zähler gleich 100 ist

Mehrfachverzweigung

Eingabe: 2 Zahlen und 1 Operator		
Operator ist		
+	-	
Ergebnis = Zahl 1 + Zahl 2	Ergebnis = Zahl 1 - Zahl 2	Operation nicht möglich
Ausgabe: 2 Zahlen, Operator, Ergebnis		

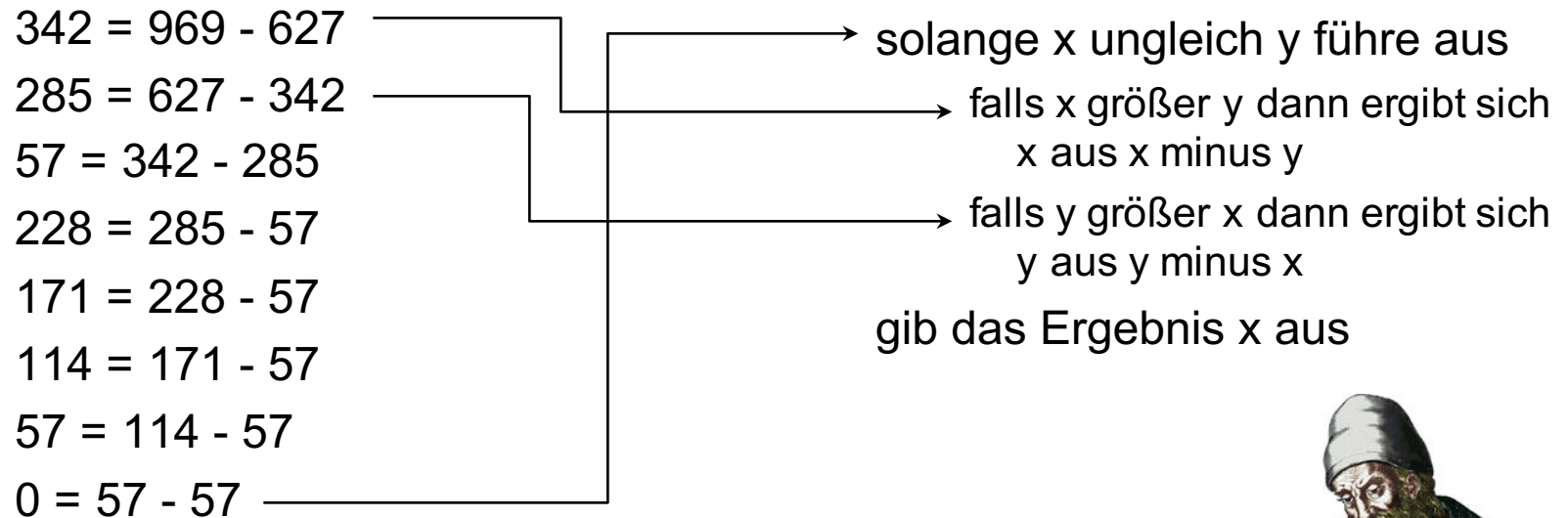
Zählschleife

Für Zähler von 1 bis 100
Ausgabe: Zähler
Ausgabe: „Unser Rechner kann Zählen“

Beispiel: Mathematische Problemstellung

gesucht sei $\text{ggT}(969, 627)$ durch
Anwendung des Euklidschen
Algorithmus

allgemeingültige Formulierung mit
den Variablen x und y : $\text{ggT}(x, y)$

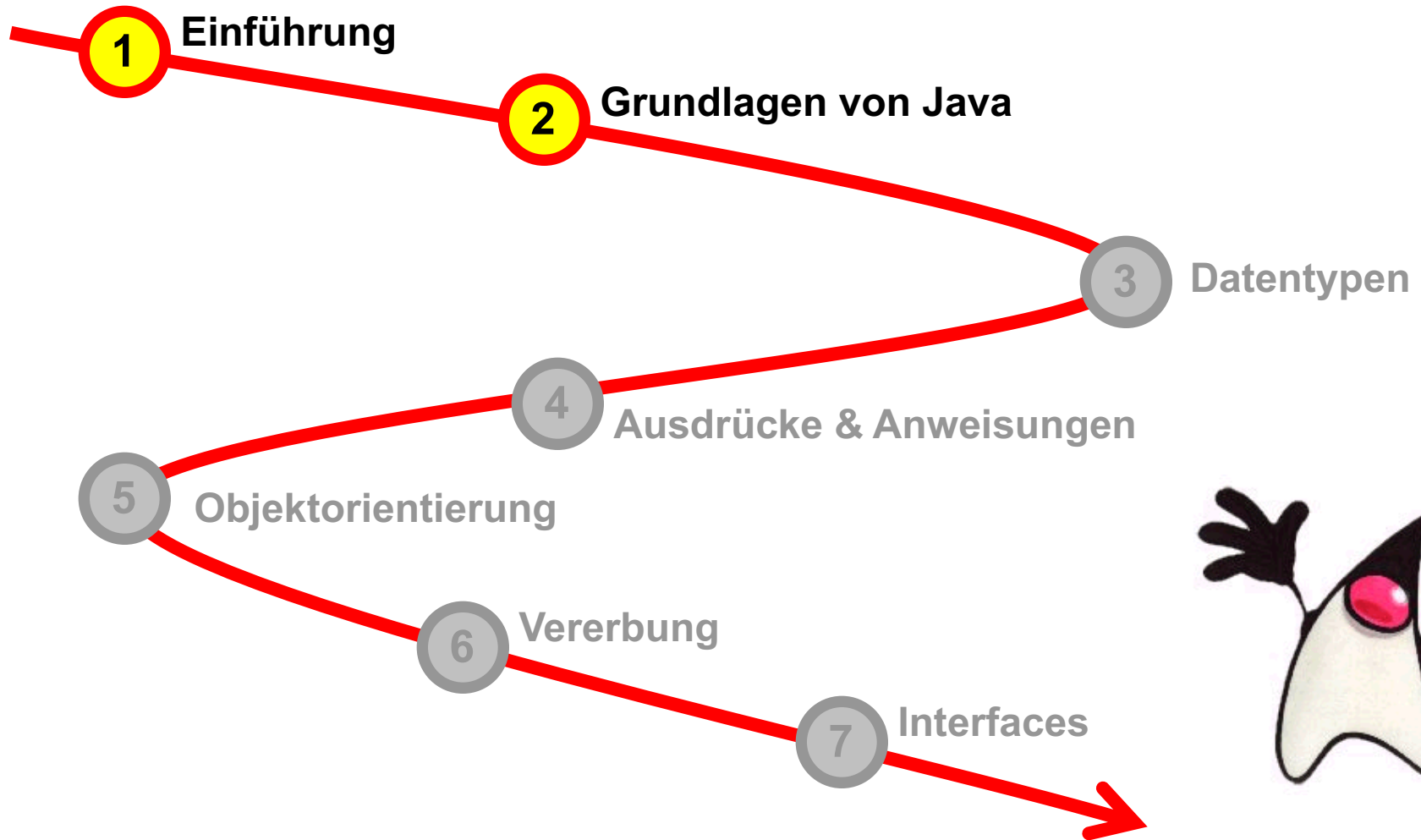




Programmierung 1

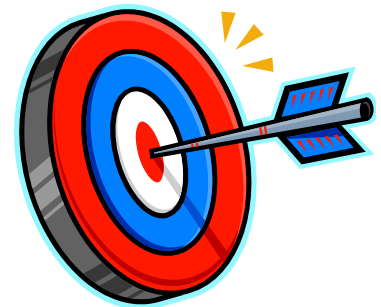
Kapitel 2 Grundlagen von Java

Themenüberblick



Lernziele

- Sie können die Eigenschaften der Programmiersprache Java beschreiben
- Sie kennen die Aufgaben von Compiler, Linker und Interpreter
- Sie können das Zusammenspiel von Bytecode und der Java Virtual Machine erläutern
- Sie können die wesentlichen Java-Tools benennen und ihre Aufgabe beschreiben
- Sie können das Paketkonzept in Java erläutern
- Sie kennen die wesentlichen Systemvariablen im Umfeld von Java
- Sie kennen den Funktionsumfang der verschiedenen Java- Editionen
- Sie können Eclipse als Java-Entwicklungsumgebung einsetzen



Klassifizierung von Software

- Programme zur Steuerung der Verarbeitungsprozesse, der Übertragungsprozesse und der Speicherungsprozesse in Computern
- unterschiedliche Softwarearten

- ◆ Systemsoftware

- * grundlegende Dienste für andere Programme
- * Steuerung des Computersystems (Hardware)
- * Ablaufsteuerung anderer Programme



- ◆ Entwicklungssoftware

- * Erstellung und Modifikation von Programmen
- * Übersetzungsprogramme für Programmiersprachen



- ◆ Anwendungssoftware

- * Programme zur Verarbeitung der Daten
- * Unterhaltungssoftware
- * Spiele



Eigenschaften von Java

vollständig objektorientiert

- ohne prozedurale Altlasten

unkompliziert - einfach und leicht erlernbar

- Syntax ähnlich zu C, C++
- Beschränkung auf das Notwendigste
 - ◆ keine Pointer
 - ◆ keine Header-Dateien
 - ◆ keine Präprozessor-Anweisungen
 - ◆ keine Mehrfachvererbung

plattformunabhängig (architekturneutral)

- übersetzte Java-Programme (Bytecode) sind auf jeder javafähigen Plattform ausführbar
- fest definierter Wertebereich für Zahlen



Eigenschaften von Java

sicher

- keine direkten Speicherzugriffe mit Pointerarithmetik
- strenge Typüberprüfung
- keine Programmierung mit Sprachverletzung

robust

- keine Rechnerabstürze durch Programmierfehler
- Überprüfung der Speicherzugriffe
- Ausnahmeroutinen zur Fehlerbehandlung

multithreaded

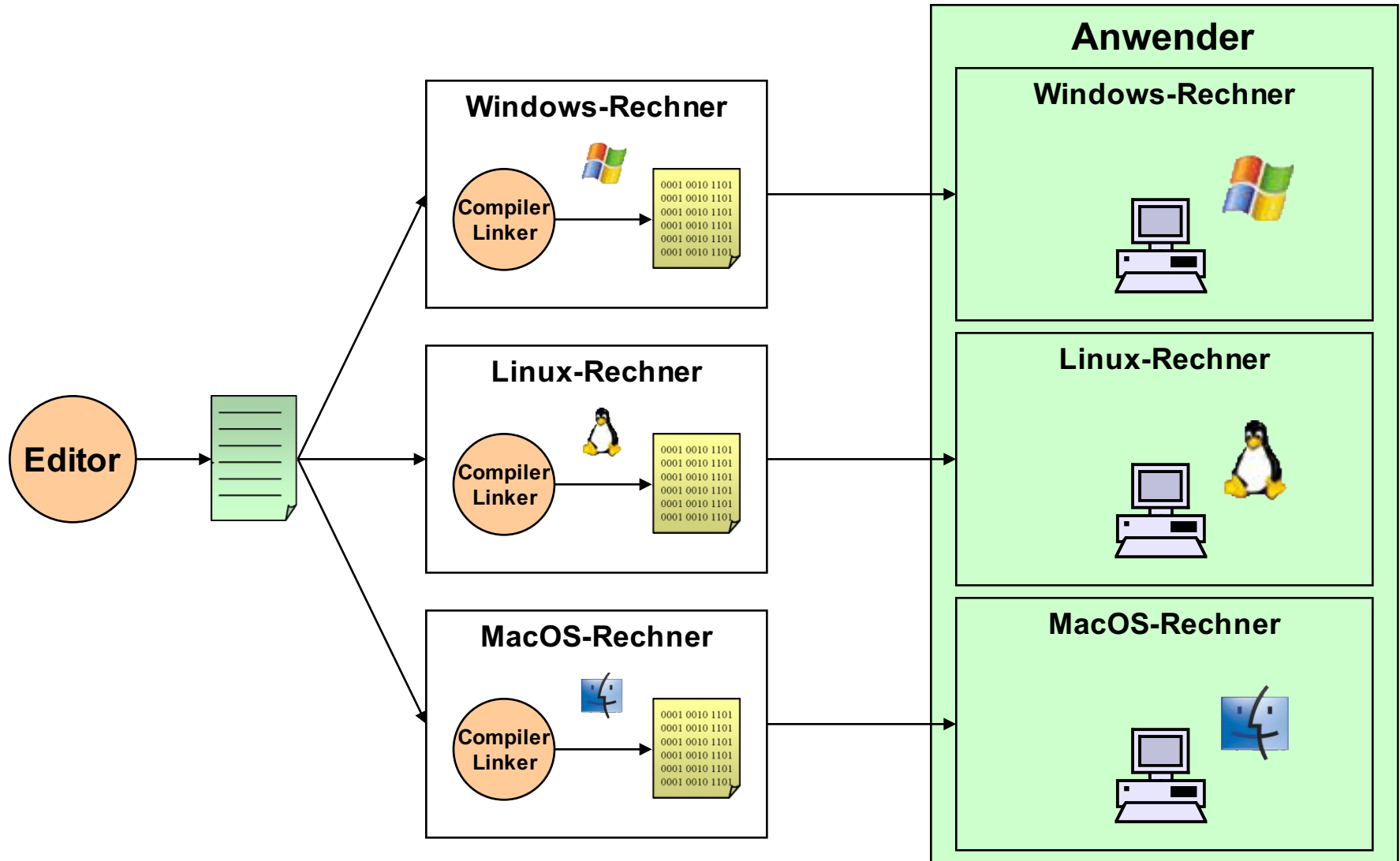
- parallele Ausführung von Programmteilen

internetfähig

- Java-Applets sind über das Internet verteilbar und können lokal auf dem Rechner ausgeführt werden



Compiler



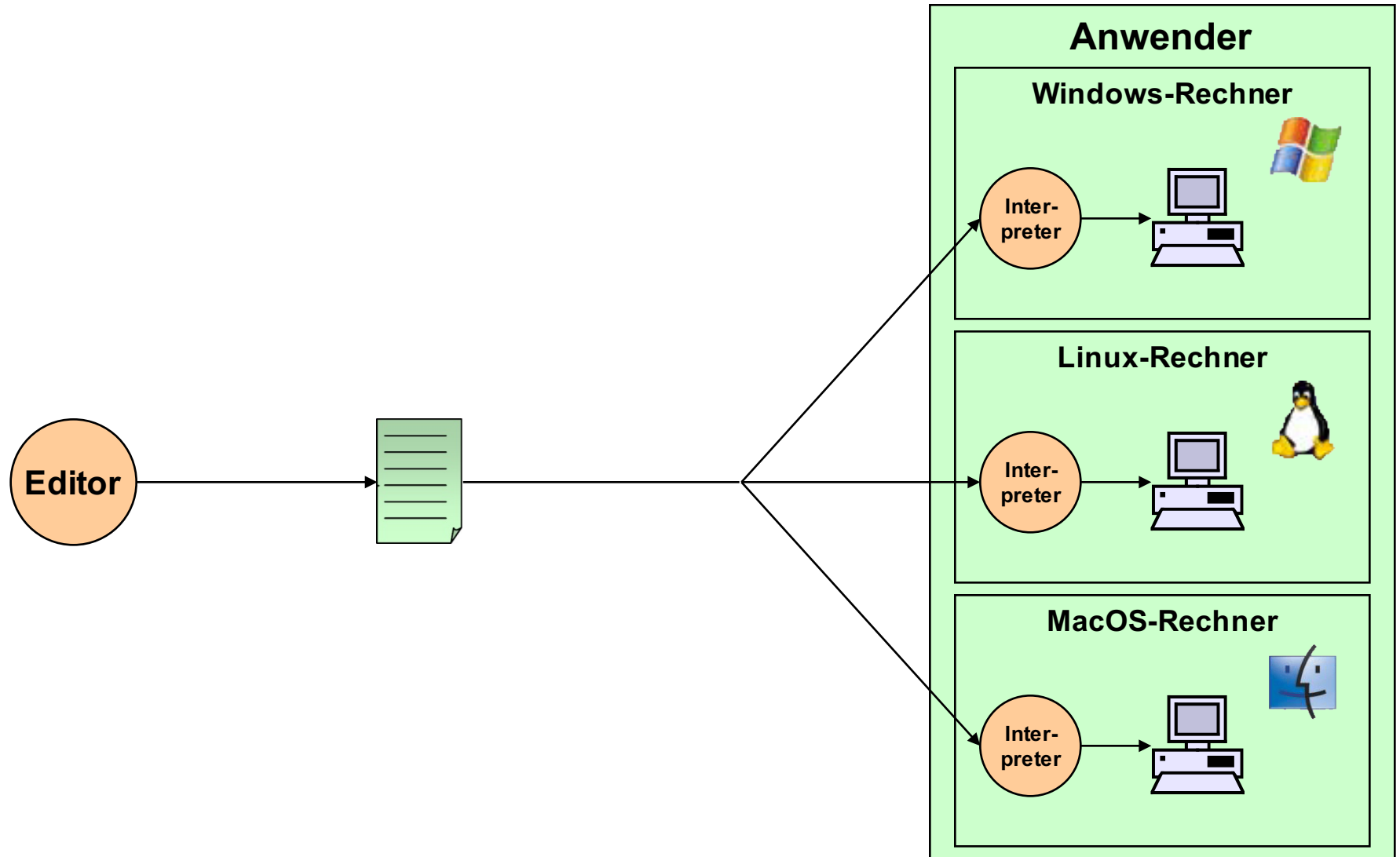
Compiler

- übersetzt ein Computerprogramm aus einer Quellsprache in ein semantisch äquivalentes Programm einer Zielsprache
- Aufbau eines Compilers in 2 Phasen
 - ◆ Analysephase
 - * lexikalische Analyse zerteilt Code in zusammengehörende Token
 - * syntaktische Analyse überprüft auf formale Richtigkeit
 - * semantische Analyse überprüft die logischen Rahmenbedingungen
 - ◆ Synthesephase
 - * Zwischencodeerzeugung liefert die Basis für die Optimierung
 - * Programmoptimierung auf Basis des maschinennahen Zwischencodes
 - * Codegenerierung erzeugt den Programmcode der Zielsprache
- verschiedene Arten von Compilern
 - ◆ Native Compiler erzeugt Code für die Plattform, auf der er läuft
 - ◆ Cross-Compiler erzeugt Code für andere Plattformen
 - ◆ One-pass-Compiler erzeugt den Code in einem Durchlauf
 - ◆ Multi-pass-Compiler erzeugt den Code in mehreren Durchläufen

Linker

- stellt einzelne Programmmodule zu einem ausführbaren Programm zusammen
- fügt benötigten Code aus Funktionsbibliotheken zum Code des Hauptprogramms hinzu
- statisches Linken
 - ◆ beim kompilieren werden benötigte Codings aus Funktionsbibliotheken dem Coding des Hauptprogramms hinzugefügt
- dynamisches Linken
 - ◆ zur Laufzeit werden die Codings aus Funktionsbibliotheken o.ä. zum Hauptprogramm hinzugelinkt
 - * DLL-Konzept von Windows
 - * auch bei Java findet dynamisches Linken statt

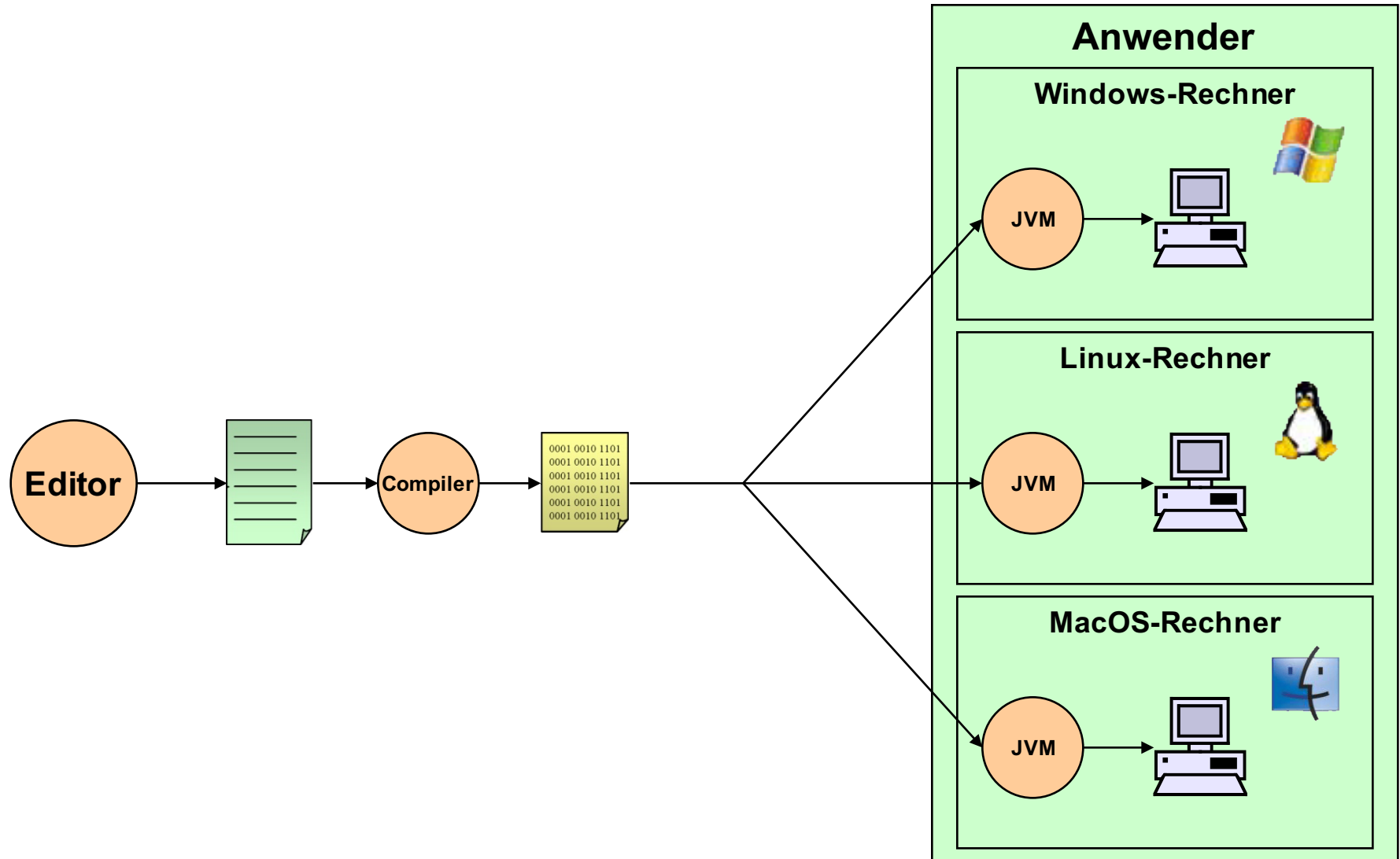
Interpreter



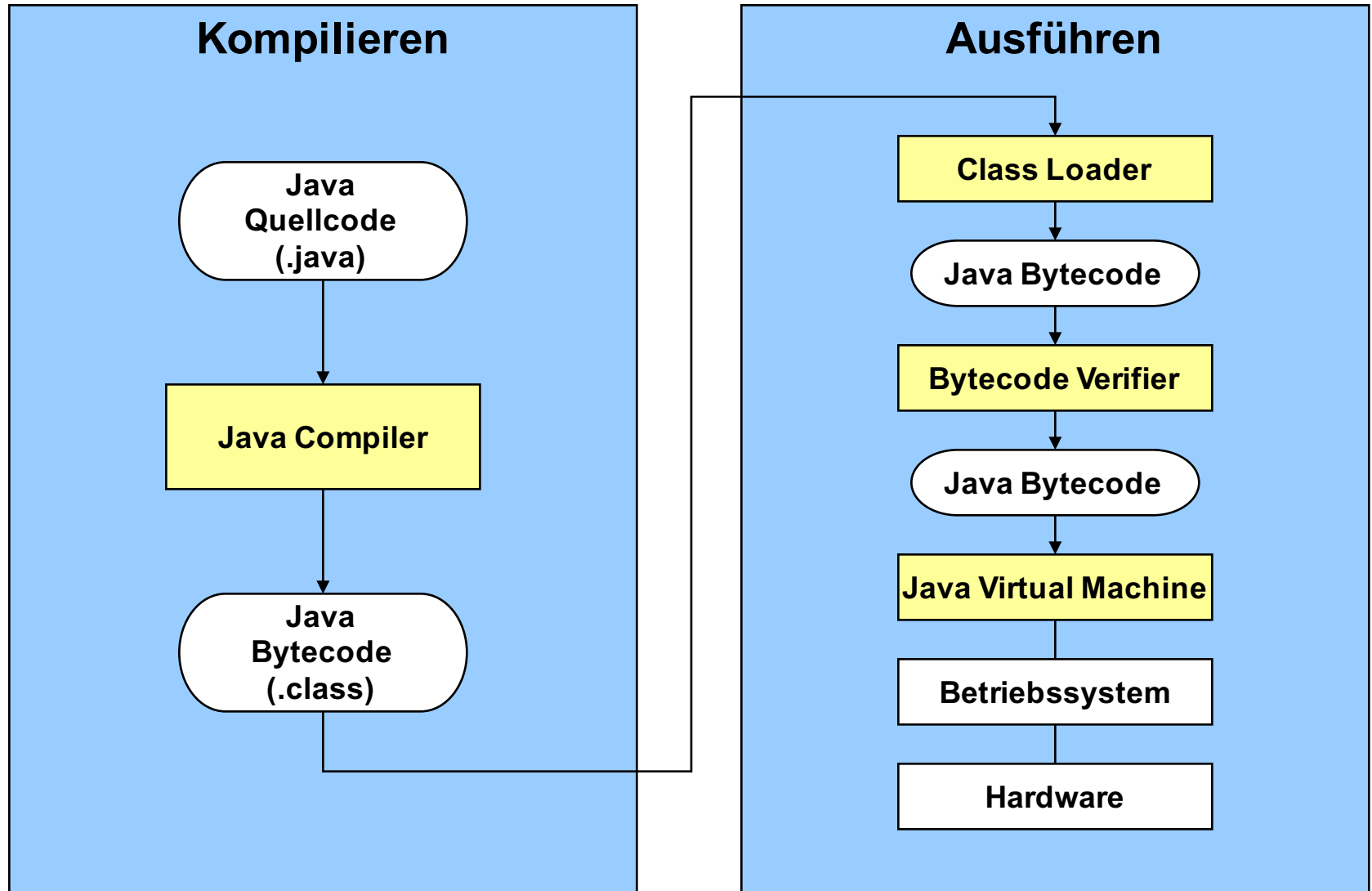
Interpreter

- übersetzt den Quellcode nicht in eine direkt ausführbare Datei
- liest den Quellcode ein, analysiert diesen und führt ihn dann aus
- die Analyse erfolgt also zur Laufzeit des Programms
- auf jeder Rechnerarchitektur lauffähig
- deutlich langsamer als kompilierte Programme
- Möglichkeiten zur Steigerung der Geschwindigkeit
 - ◆ Just-In-Time-Compiler
 - * zur Laufzeit wird der Quellcode in einen Maschinencode übersetzt
 - * Maschinencode wird direkt vom Prozessor ausgeführt
 - * mehrfach durchlaufene Programmteile müssen nur ein Mal übersetzt werden
 - * nur auf einer bestimmten Rechnerarchitektur lauffähig
 - ◆ Bytecode-Interpreter
 - * Quellcode wird zur Laufzeit in sog. Bytecode übersetzt
 - * Bytecode wird von einem Interpreter (Virtual Machine) ausgeführt
 - * Bytecode kann auf verschiedenen Plattformen ausgeführt werden

Bytecode und virtual Machine



Bytecode und virtual Machine





Wesentliche Java-Tools

Java-Compiler (javac.exe)

Java virtual Machine (java.exe)

Javadoc

- dient der automatischen Erstellung von Dokumentationen

Class Loader

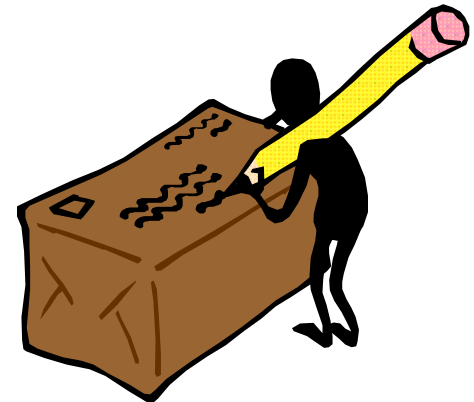
- lädt Java-Klassen in den Arbeitsspeicher
- bereitet Ausführung von Java-Applikationen vor

Bytecode Verifier

- Prüfung auf syntaktische Korrektheit
- Überprüfung der Klassenhierarchie
- Überprüfung jeder Methode auf strukturelle Gültigkeit
- Datenflussanalyse, um u.a. Typfehler zu vermeiden

Das Paketkonzept in Java

- Möglichkeit zur Strukturierung bzw. sinnvollen Sortierung von Klassen
- Pakete sind somit Sammlungen von Klassen
- die Klassen verfolgen einen gemeinsamen Zweck
- jede Klasse ist genau einem Paket zugeordnet
- Paketnamen können aus mehreren Teilen bestehen und hierarchisch aufgebaut sein (vergleichbar mit der Ordnerstruktur im Windows-Explorer)
- eine Klasse kann eindeutig über das Paket und ihren Namen identifiziert werden



Wichtige Systemvariablen

PATH

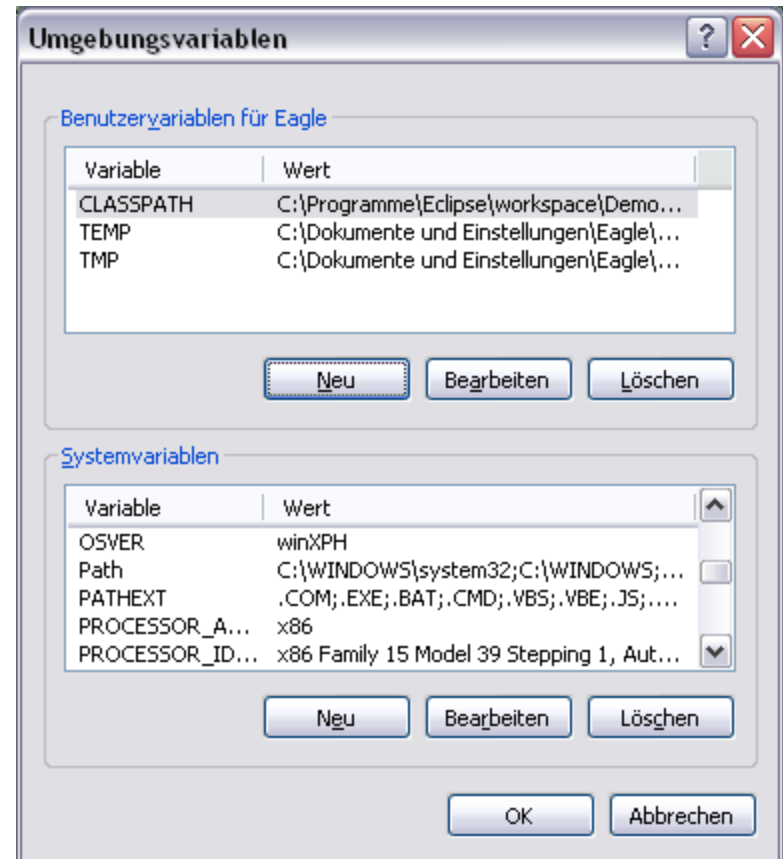
- Verzeichnis, in dem sich die JAVA-Tools befinden

CLASSPATH

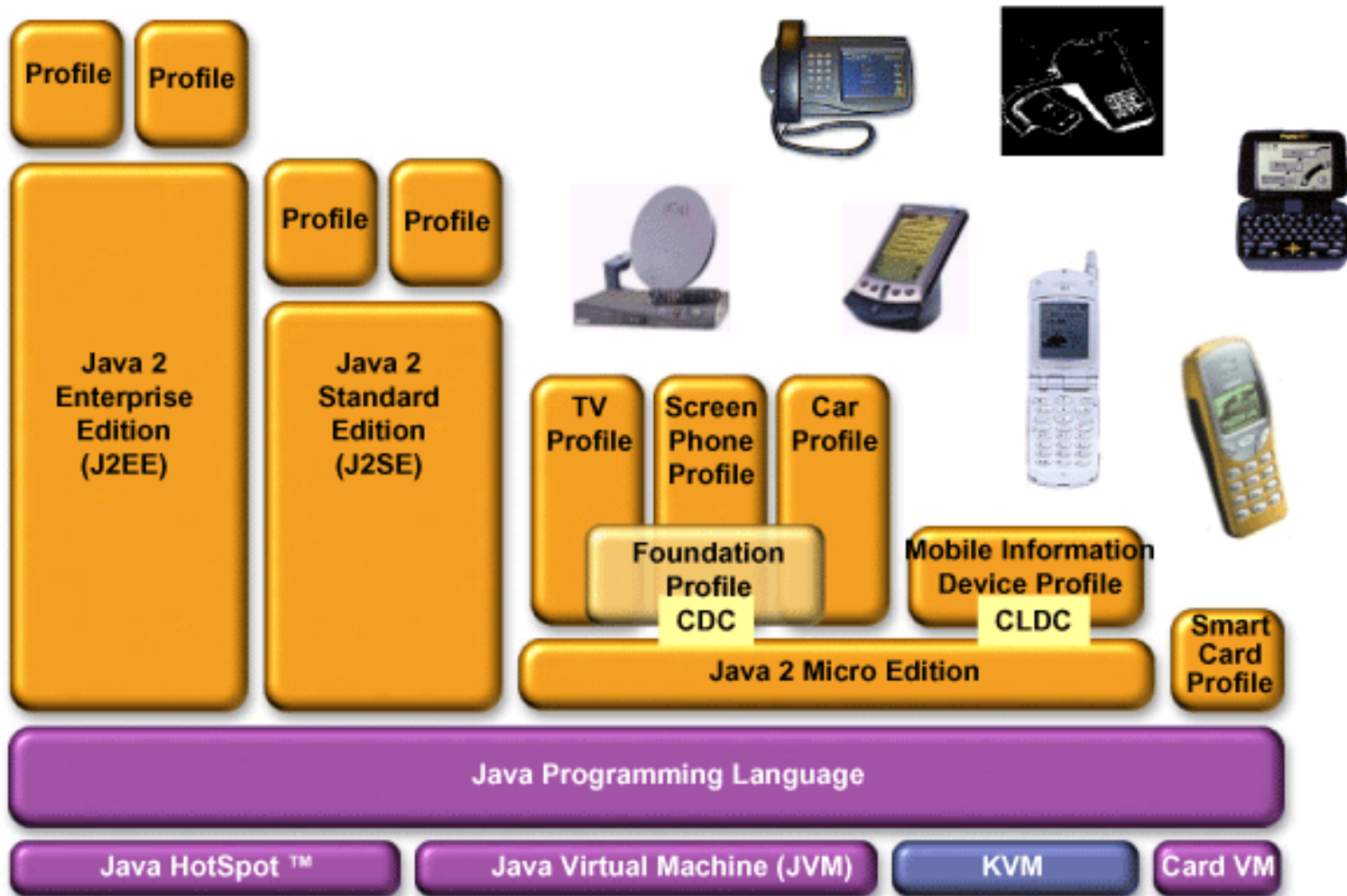
- Verzeichnisse, in denen nach Klassen und Paketen gesucht werden soll

JAVA_HOME

- Verzeichnis, in dem das J2SDK installiert wurde



Funktionsumfang der Java 2™ Plattform



Funktionsumfang der Java 2™ Plattform

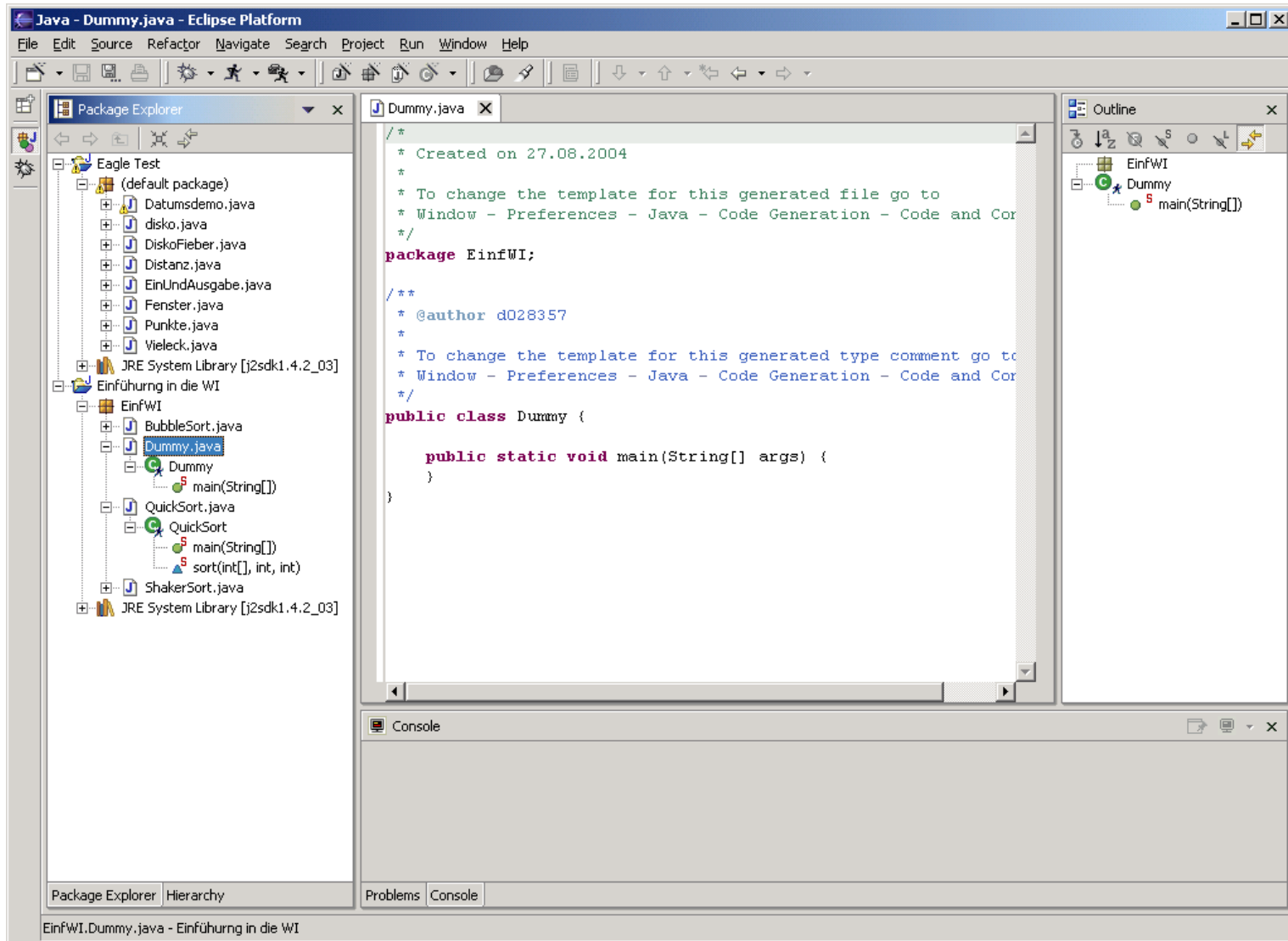
- Java 2 Standard Edition beinhaltet das Software Development Kit mit der Standard API zur Entwicklung von Java-Applikationen
- Java 2 Enterprise Edition umfasst neben der J2SE weitere Packages zur serverseitigen Programmierung (Enterprise Java Beans, Servlets, JSP, Java-Mail-API, etc.)
- Java 2 Micro Edition stellt eine funktional kleinere Laufzeitumgebung für mobile Endgeräte (PDAs, Handys, Navigationssysteme, etc.) dar
 - ◆ Connected Device Configuration (CDC) beinhaltet die komplette JVM und ist in mobilen Systemen integriert
 - ◆ Connected Limited Device Configurations (CLDC) ist eine J2ME-Bibliothek zur Abdeckung gerätespezifischer Funktionen; ermöglicht die Zusammenarbeit verschiedener Geräte der gleichen Kategorie
 - ◆ KVM ist die kleinste Laufzeitumgebung und wird für den Einsatz auf Geräten mit beschränkter Speicherkapazität und CPU-Leistung verwendet
- Java Card APIs definieren eine minimale Laufzeitumgebung auf SmartCards

Eclipse als Entwicklungsumgebung

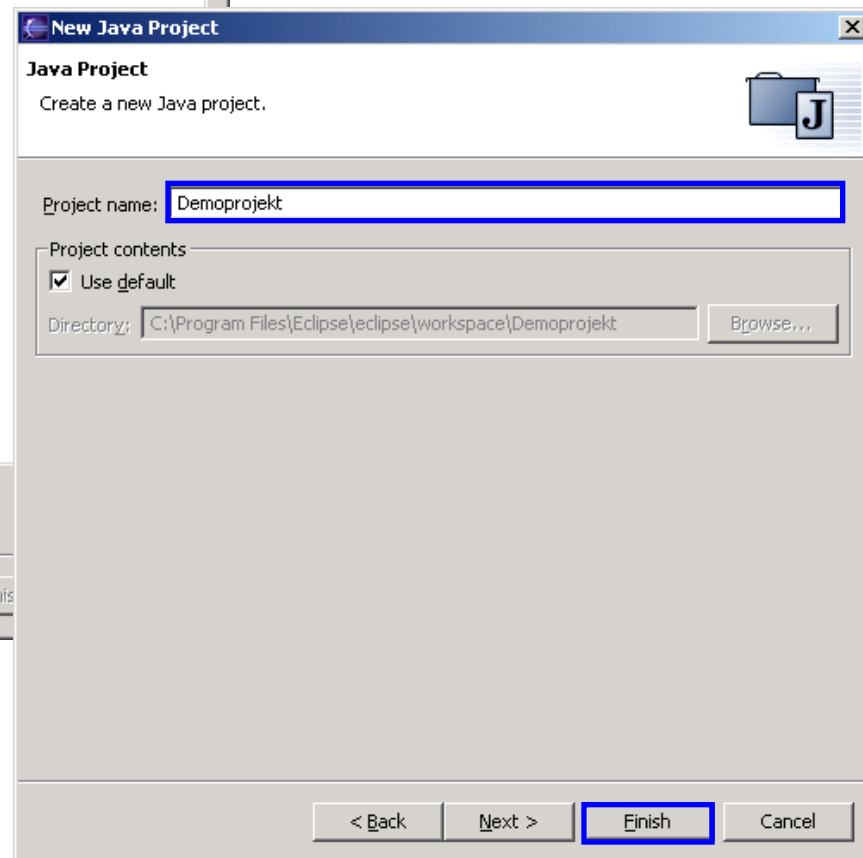
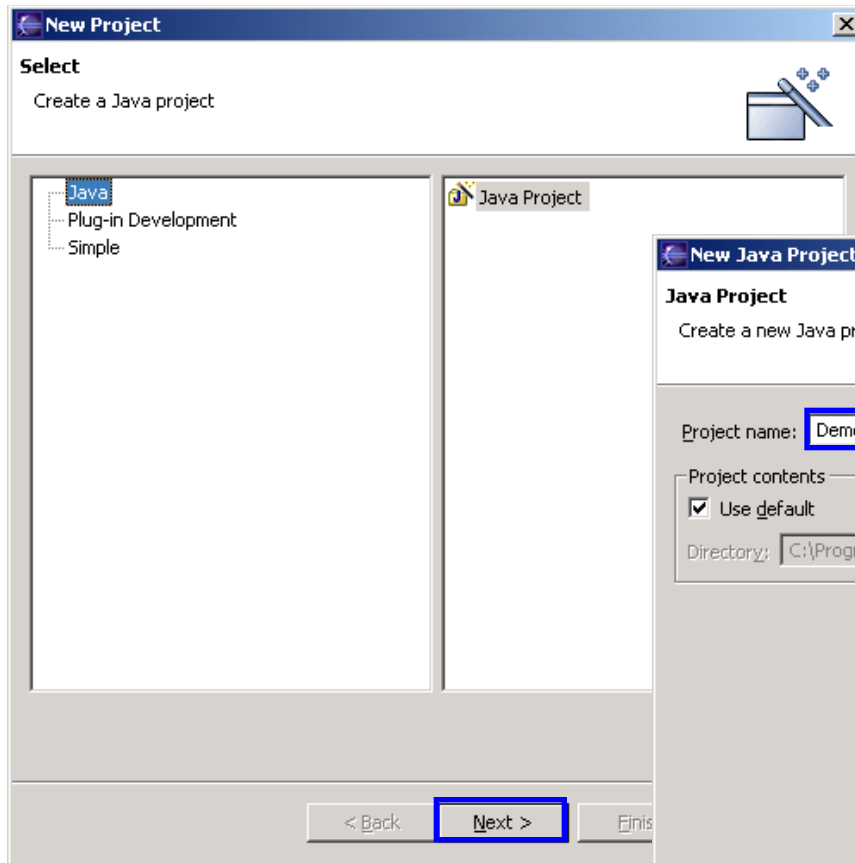
- Initiative kam von IBM, Quellcode wurde im Jahr 2001 freigegeben
- Eclipse ist seitdem ein Open-Source-Projekt
- komplett in Java implementiert
- kostenloser Download unter <http://www.eclipse.org>
- bietet die Möglichkeit, Tools von anderen Herstellern zu integrieren -> Plug-In-Technologie
- bietet die Möglichkeit, Programme im Team zu entwickeln
- beinhaltet einen inkrementellen Compiler, d.h. beim Speichern des Quellcodes wird der Byte-Code automatisch erzeugt
- **NACHTEIL:** hohe Systemanforderungen



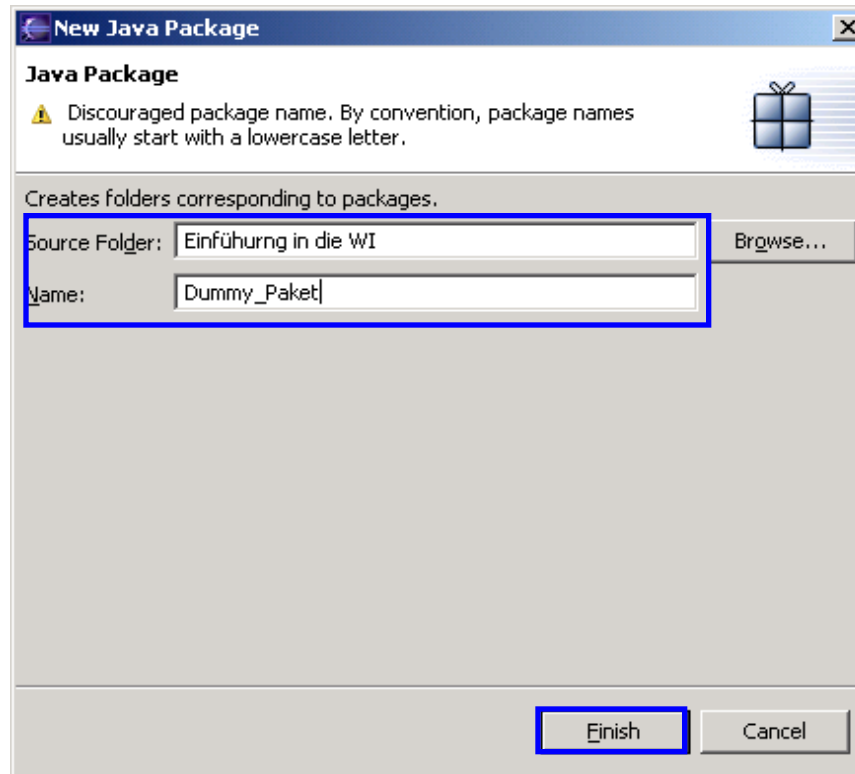
Eclipse als Entwicklungsumgebung



Ein neues Projekt anlegen



Ein neues Paket anlegen



Eine neue Klasse anlegen

New Java Class

Java Class
Create a new Java class.

Source Folder: Einführung in die WI Browse...

Package: EinfWI Browse...

☐ Enclosing type: Browse...

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object Browse...

Interfaces: Add...
Remove

Which method stubs would you like to create?

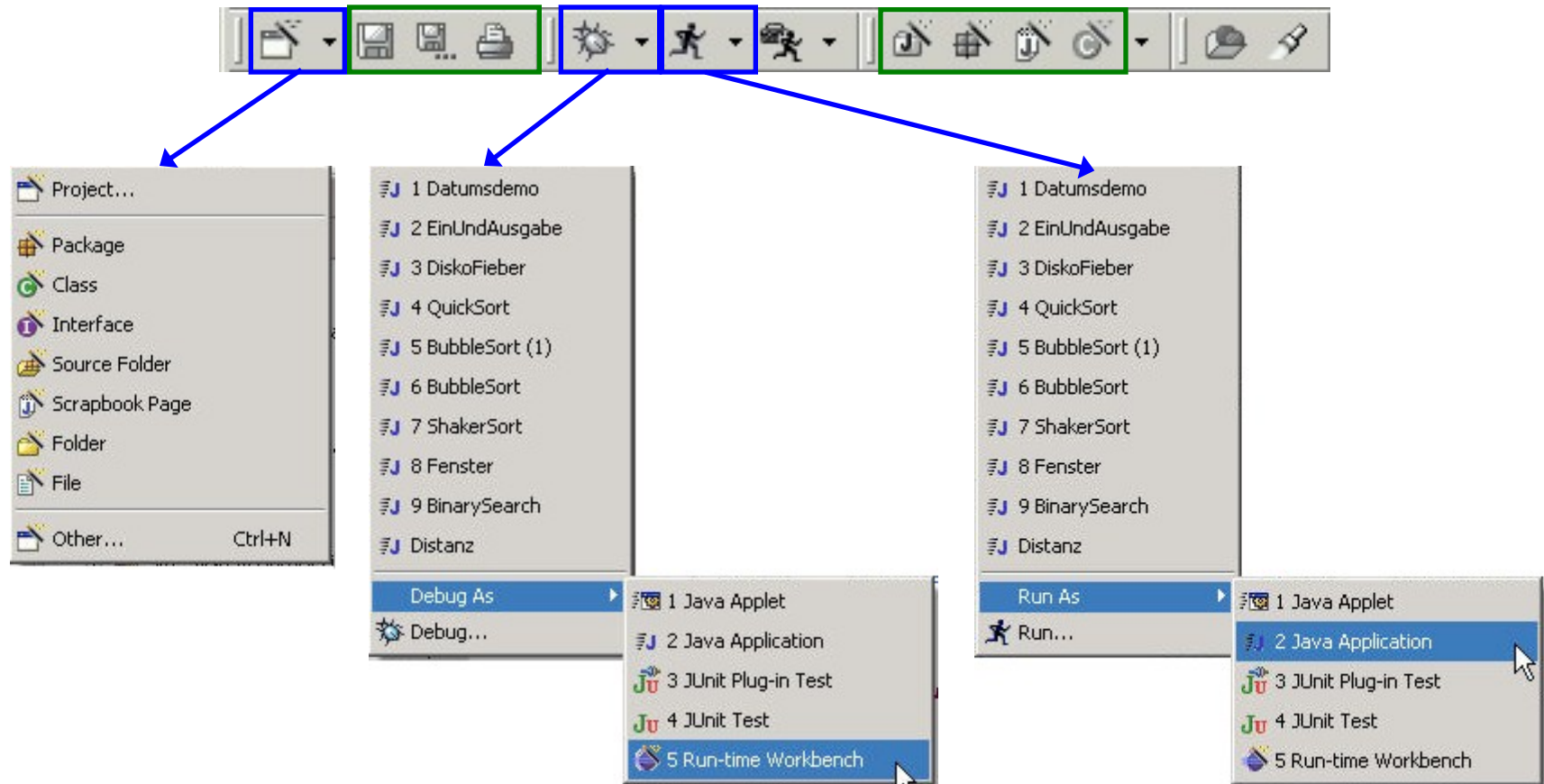
☒ public static void main(String[] args)

☐ Constructors from superclass

☒ Inherited abstract methods

Finish Cancel

Die Symbolleiste

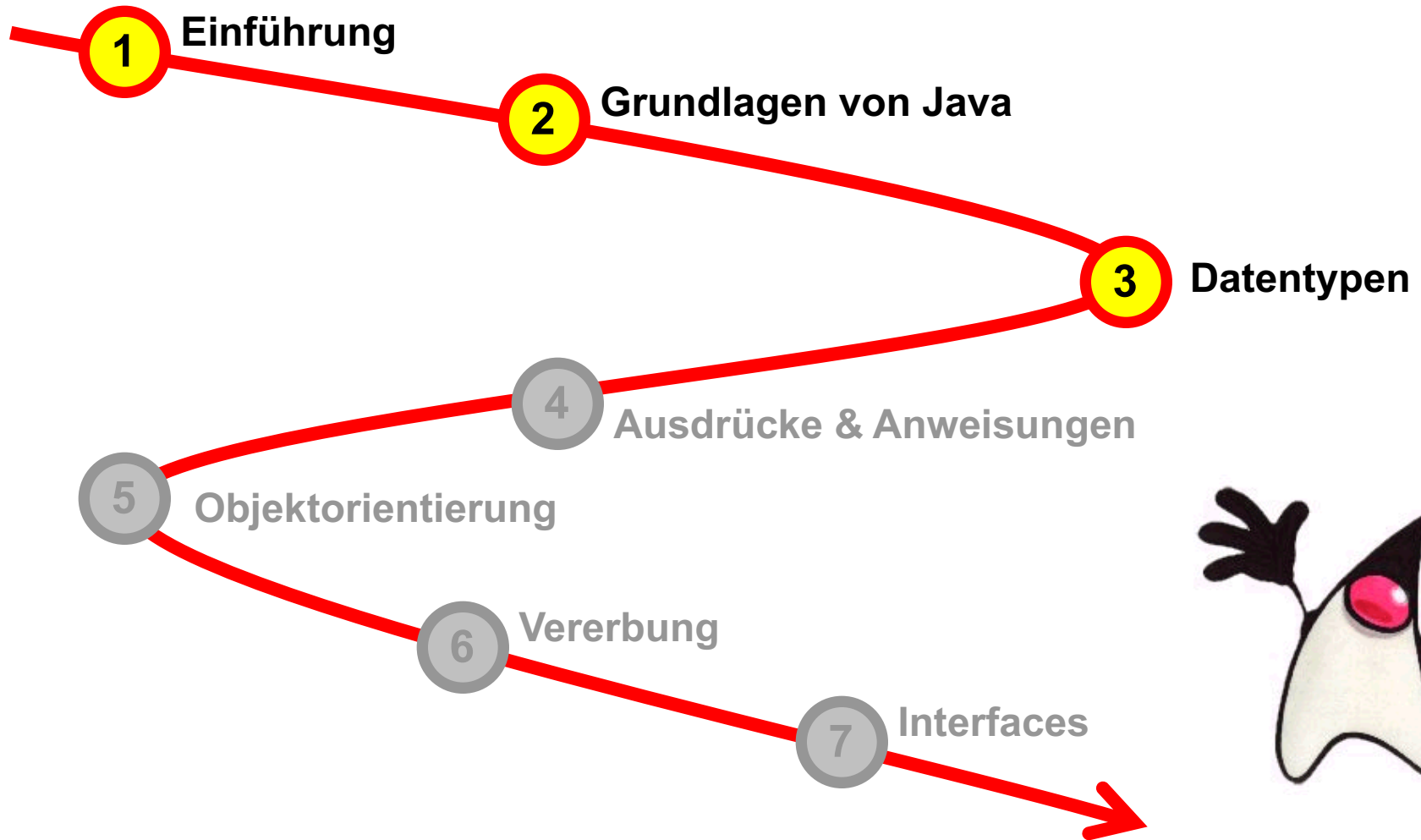




Programmierung 1

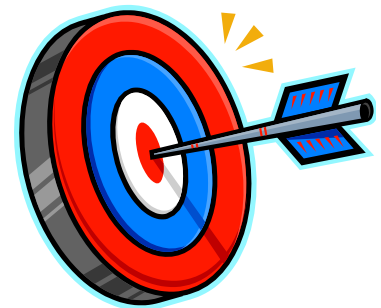
Kapitel 3 Datentypen

Themenüberblick

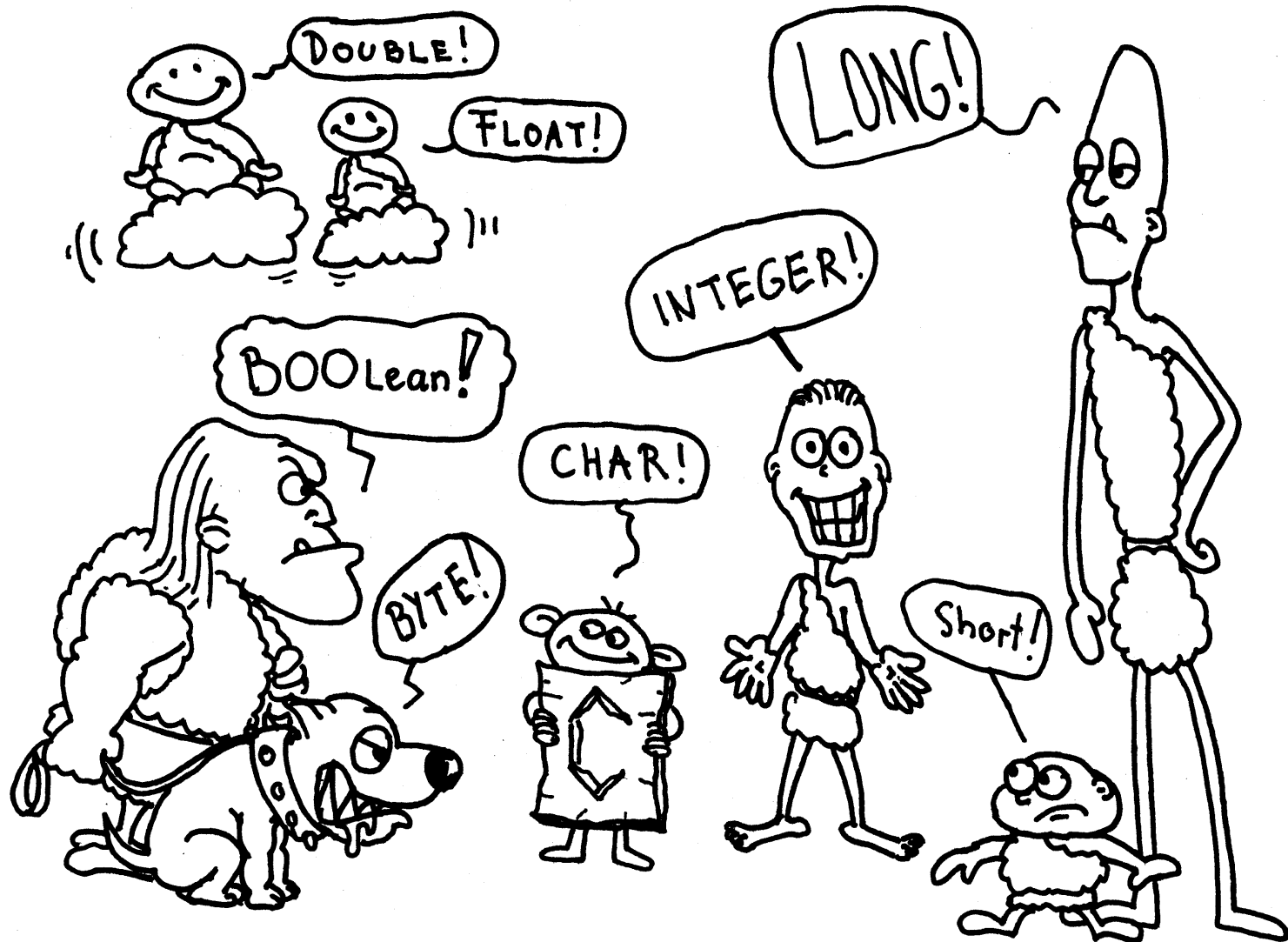


Lernziele

- Sie kennen die unterschiedlichen einfachen Datentypen in Java
- Sie kennen die Wertebereiche der jeweiligen Datentypen
- Sie können Variablen und Konstanten in Java deklarieren
- Sie kennen die unterschiedlichen Literale
- Sie kennen unterschiedliche Standard-Escape-Sequenzen und können diese einsetzen
- Sie kennen die Konvertierungsvorschriften bezüglich der einfachen Datentypen
- Sie können den Aufbau von Arrays beschreiben
- Sie können Arrays in Java deklarieren
- Sie kennen den theoretischen Umgang mit Referenzdatentypen
- Sie kennen die speziellen Referenzdatentypen Array und String



Variablen und Datentypen



Variablen und Datentypen

- bool'sche Typen sind Wahrheitswerte: *True* und *False*
- numerische Typen
 - ◆ Byte, Short, Integer und Long: Menge der *ganzen Zahlen*
 - ◆ Double und Float: Menge der *reellen Zahlen*
- alphanumerisch
 - ◆ Char: Menge der *vereinbarten Schriftzeichen*, z.B. ASCII-Zeichensatz
- komplexe Datentypen
 - ◆ String: Verkettung von Character-Werten → Zeichenketten
 - ◆ Array: Definition eines n-dimensionalen Feldes
 - ◆ Referenzdatentypen: Individuell definierte Datentypen

Wertebereiche und Speicherverbrauch

Typname	Länge in Byte	Wertebereich
boolean	1	true, false
char	2	Alle Unicode-Zeichen
byte	1	-2^7 bis 2^7-1 (-128 bis 127)
short	2	-2^{15} bis $2^{15}-1$ (-32.768 bis 32.767)
int	4	-2^{31} bis $2^{31}-1$ (-2.147.483.648 bis 2.147.483.647)
long	8	-2^{63} bis $2^{63}-1$ (-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807)
float	4	$\pm 3.40282347 \cdot 10^{38}$
double	8	$\pm 1.79769313486231570 \cdot 10^{308}$

Deklaration von Variablen und Konstanten

- Variablen werden durch Datentyp und Variablennamen deklariert
Typname Variablenname;

- Beispiele für Variablendeklarationen

boolean muede;

int a;

- Beispiele für Deklarationen mit Initialisierung

char buchstabe = 'a';

long zahl = 45768;

- Konstanten werden ähnlich wie Variablen deklariert
- vor den Typnamen wird das Schlüsselwort *final* gesetzt

final Typname Konstantenname;

- Konstanten können nach der Deklaration einmalig mit einem Wert initialisiert werden

- Beispiele

final double pi = 3.1415926535897932384626433832795;

Numerische Literale

- Literale für die Familie der Integer-Werte (byte, short, int, long) können in Dezimal-, Oktal- oder Hexadezimalform geschrieben werden
 - ◆ Dezimalliterale bestehen aus den Ziffern 0 bis 9
 - ◆ Oktalliterale bestehen aus dem Präfix 0 und den Ziffern 0 bis 7
 - ◆ Hexadezimalliterale bestehen aus dem Präfix 0x, den Ziffern 0 bis 9 und den Buchstaben A bis F bzw. a bis f
 - ◆ durch das Suffix l bzw. L wird ein Literal vom Typ long erzeugt
- Literale für Fließkommazahlen float und double
 - ◆ bestehen aus einem Vorkommateil, dem Dezimalpunkt, einem Nachkommateil, einem Exponenten und einem Suffix
 - ◆ Unterscheidung zwischen float und double durch das Suffix f bzw. d
 - ◆ Exponent wird durch ein e bzw. E eingeleitet
 - ◆ Vorkomma- oder Nachkommateil darf ausgelassen werden, der Exponent und das Suffix sind optional
 - ◆ ohne Suffix sind Fließkommazahlen immer vom Typ double

Alphanumerische Literale

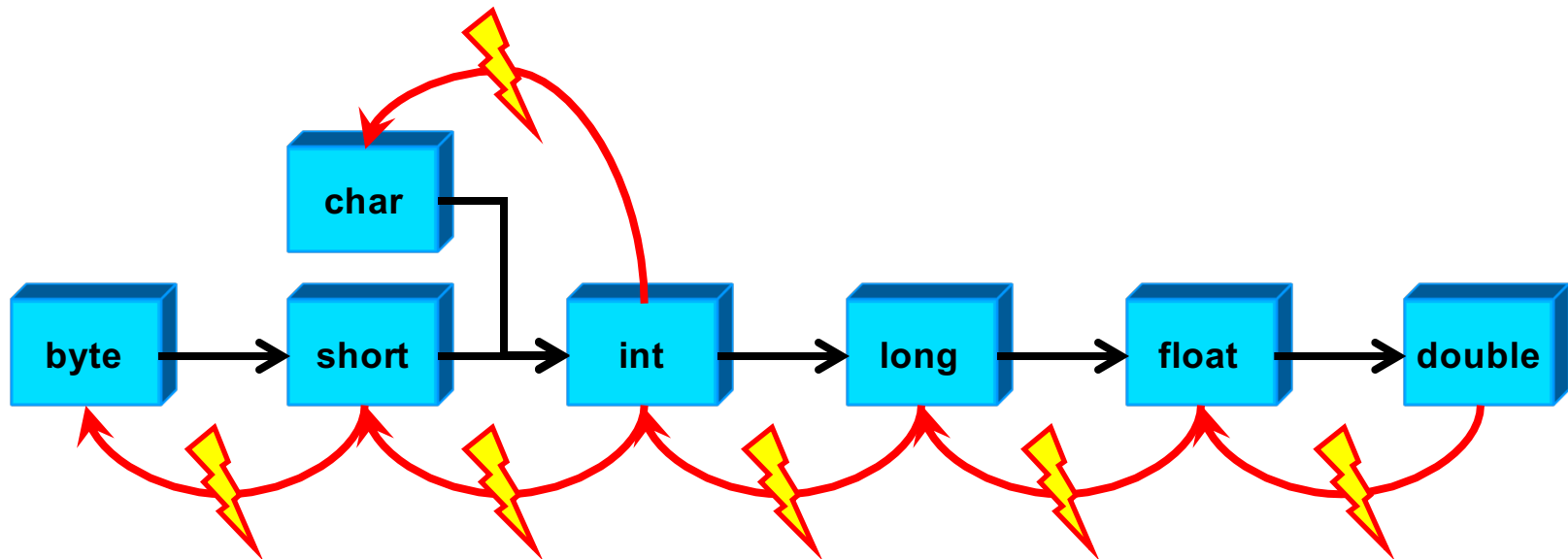
- besondere Literale für den Datentyp boolean
true und false
- Char-Literale werden grundsätzlich in einfache Hochkommata gesetzt
- Char-Variablen sind prinzipiell 2 Byte lang, da in char Unicode-Werte gespeichert werden können
- besondere Zeichenliterale stellen die String-Literale dar
- String-Literale stehen in doppelten Hochkommata
- Zeichenliterale können Standard-Escape-Sequenzen zur Darstellung von Sonderzeichen beinhalten

Standard-Escape-Sequenzen

Zeichen	Bedeutung
\b	Rückschritt (Backspace)
\t	Horizontaler Tabulator
\n	Zeilenschaltung (Newline)
\f	Seitenumbruch (Formfeed)
\r	Wagenrücklauf (Carriage return)
\“	Doppeltes Anführungszeichen
\‘	Einfaches Anführungszeichen
\\	Backslash
\nnn	Oktalzahl nnn (nicht größer 377)
\uxxxx	Unicodezeichen (xxxx steht für den Hexadezimalwert des Unicodezeichens)

Konvertierungsregeln

- generelle Unterscheidung in erweiternde und einschränkende Konvertierung
- es existiert keine Konvertierungsvorschrift für den Datentyp boolean und für die Konvertierung zwischen einfachen und Referenzdatentypen



Aufbau eines Arrays

- Arrays sind (mehrdimensionale) Feldvariablen, die aus mehreren Elementen bestehen
- alle Elemente eines Arrays gehören dem gleichen Datentyp an
- in Java sind Arrays semidynamisch, d.h. ihre Größe kann zur Laufzeit festgelegt, aber danach nicht mehr verändert werden
- Arrays in Java sind Objekte und bieten verschiedene Methoden
- die Elemente eines Arrays sind bei n Elementen von 0 bis $n-1$ durchnummeriert
- Zugriffe auf einzelne Elemente von Arrays erfolgen über den numerischen Index
- das Attribut `length` ist vom Typ `Integer` und gibt die Anzahl der Elemente eines Arrays zurück

Deklaration von Arrays

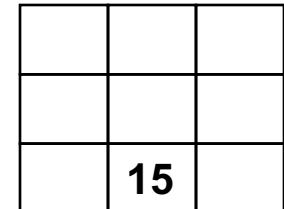
- die Deklaration eines Arrays entspricht syntaktisch der einer einfachen Variablen
- Unterschied: an den Typnamen oder an den Variablennamen werden eckige Klammern angefügt
- die Initialisierung erfolgt mithilfe des new-Operators oder durch Zuweisung eines Array-Literals

- Beispiele

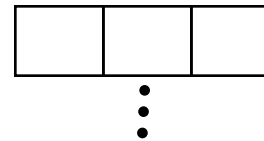
- ◆ `int [] zahl = {1, 2, 3, 4, 5};`
`zahl[1] = 7;`



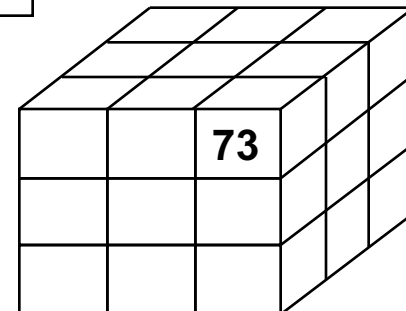
- ◆ `int [][] matrix = new int[3][3];`
`matrix[1][2] = 15;`



- ◆ `int [][] matrix = new int[3][];`
`matrix[0] = new int[4];`



- ◆ `int [][][] wuerfel = new int[3][3][3];`
`wuerfel[2][0][0] = 73;`



Umgang mit Referenzdatentypen

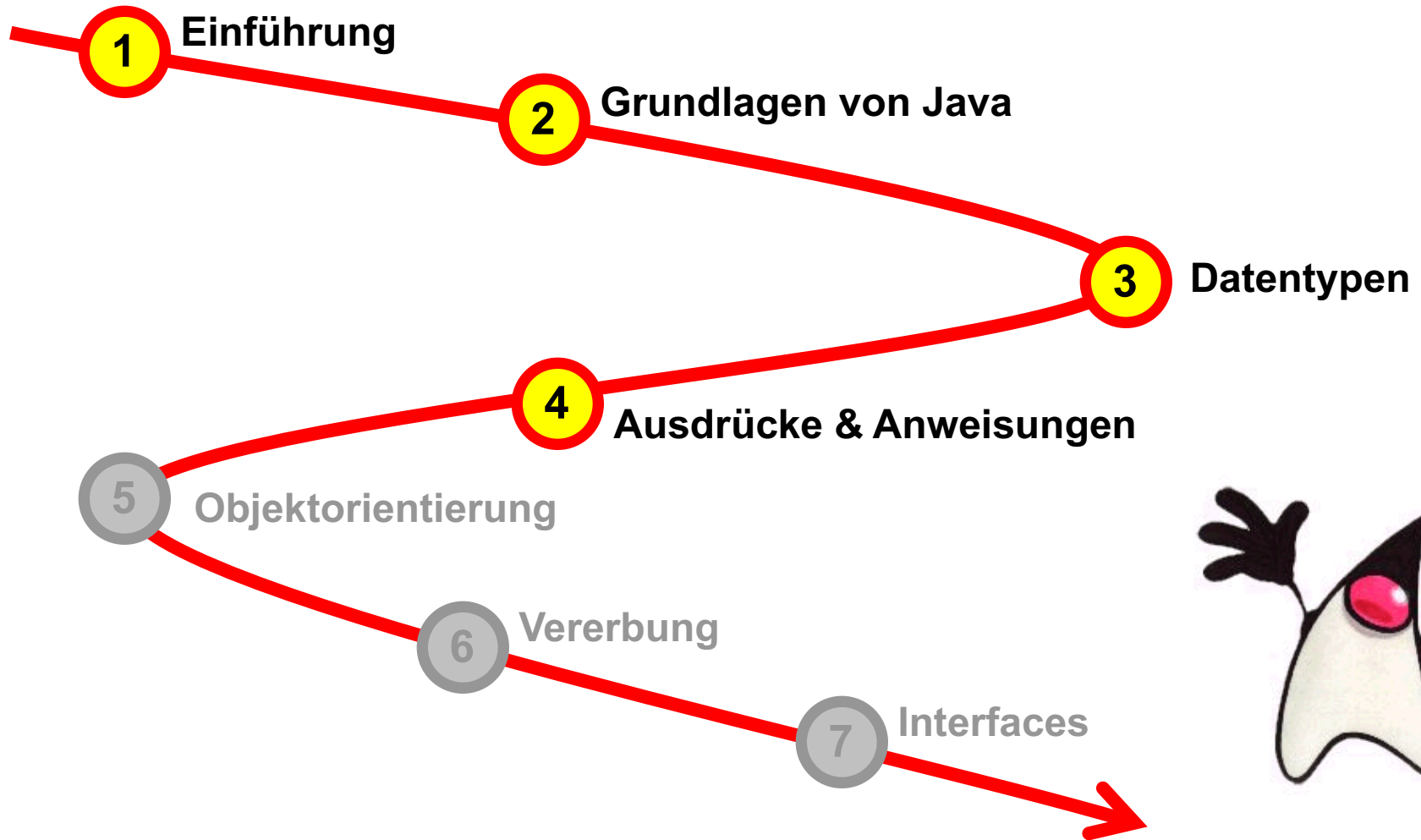
- im Gegensatz zu einfachen Datentypen handelt es sich bei Referenzdatentypen um komplexe Typen
- zu den Referenztypen zählen
 - ◆ Objekte (näheres im Kapitel 5: Objektorientierung)
 - ◆ Strings
 - ◆ Arrays
- Strings und Arrays sind besondere Objekte
- sowohl bei Strings als auch bei Arrays kennt der Compiler Literale, die den Aufruf des Operators `new` überflüssig machen
- bei einfachen Datentypen reicht die Deklaration der Variablen aus, während Referenztypen mittels des `new`-Operators noch erzeugt werden müssen (Ausnahmen Strings und Arrays)
- Referenztypen stellen lediglich einen Verweis auf Objekte dar



Programmierung 1

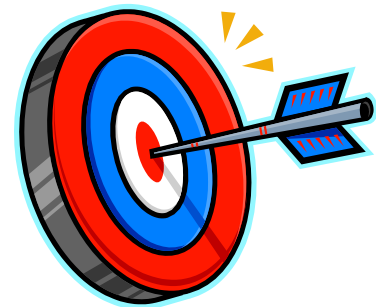
Kapitel 4 Ausdrücke & Anweisungen

Themenüberblick



Lernziele

- Sie kennen die Begriffe Ausdrücke und Anweisungen
- Sie kennen die unterschiedlichen Arten von Operatoren
 - ◆ arithmetisch
 - ◆ relational
 - ◆ logisch
 - ◆ bitweise
 - ◆ Zuweisung
 - ◆ sonstige
- Sie kennen die Verzweigungsmöglichkeiten mit der IF- und der SWITCH-Anweisung
- Sie kennen die unterschiedlichen Schleifentypen
 - ◆ kopfgesteuert
 - ◆ fußgesteuert
 - ◆ zählend
- Sie kennen die break- und continue-Anweisung



Ausdrücke

- kleinste ausführbare Einheit in einem Programm
- Ausdrücke haben folgende Aufgaben
 - ◆ Variablen Werte zuzuweisen
 - ◆ numerische Berechnungen durchzuführen
 - ◆ logische Bedingungen zu formulieren
- Ausdrücke bestehen aus mindestens einem Operator und einem oder mehreren Operanden, auf die der Operator angewendet wird
- Unterscheidung der Operatoren nach dem Typ der Operanden
 - ◆ arithmetische Operatoren
 - ◆ relationale Operatoren
 - ◆ logische Operatoren
 - ◆ bitweise Operatoren
 - ◆ Zuweisungsoperatoren
 - ◆ sonstige Operatoren

Arithmetische Operatoren

Operator	Bezeichnung	Bedeutung
+	Positives Vorzeichen	$+n$ ist gleichbedeutend mit n
-	Negatives Vorzeichen	$-n$ kehrt das Vorzeichen von n um
+	Summe	$a + b$ ergibt die Summe von a und b
-	Differenz	$a - b$ ergibt die Differenz
*	Produkt	$a * b$ ergibt das Produkt
/	Quotient	a / b ergibt den Quotienten
%	Restwert	$a \% b$ ergibt den Rest der ganzzahligen Division von a durch b
++	Präinkrement	$++a$ erhöht a um 1 und ergibt a
++	Postinkrement	$a++$ ergibt a und erhöht a um 1
--	Prädecrement	$--a$ verringert a um 1 und ergibt a
--	Postdecrement	$a--$ ergibt a und verringert a um 1

Relationale Operatoren

Operator	Bezeichnung	Bedeutung
==	Gleich	$a == b$ ergibt true, wenn a gleich b ist. Sind a und b Referenztypen, so ist der Rückgabewert true, wenn beide Werte auf dasselbe Objekt zeigen.
!=	Ungleich	$a != b$ ergibt true, wenn a ungleich b ist. Sind a und b Referenztypen, so ist der Rückgabewert true, wenn beide Werte auf verschiedene Objekt zeigen.
<	Kleiner	$a < b$ ergibt true, wenn a kleiner b ist.
<=	Kleiner gleich	$a <= b$ ergibt true, wenn a kleiner oder gleich b ist.
>	Größer	$a > b$ ergibt true, wenn a größer b ist.
>=	Größer gleich	$a >= b$ ergibt true, wenn a größer oder gleich b ist.

Logische Operatoren

Operator	Bezeichnung	Bedeutung
!	Logisches NICHT	!a ergibt false, wenn a wahr ist, und true, wenn a falsch ist.
&&	UND mit Short-Circuit-Evaluation	a && b ergibt true, wenn sowohl a als auch b wahr sind. Ist a bereits falsch, so wird false zurückgegeben und b nicht mehr ausgewertet.
	ODER mit Short-Circuit-Evaluation	a b ergibt true, wenn mindestens einer der beiden Ausdrücke a oder b wahr ist. Ist a bereits wahr, so wird true zurückgegeben und b nicht mehr ausgewertet.
&	UND ohne Short-Circuit-Evaluation	a & b ergibt true, wenn sowohl a als auch b wahr sind. Beide Teilausdrücke werden ausgewertet.
	ODER ohne Short-Circuit-Evaluation	a b ergibt true, wenn mindestens einer der beiden Ausdrücke a oder b wahr ist. Beide Teilausdrücke werden ausgewertet.
^	Exklusiv-ODER	a ^ b ergibt true, wenn beide Ausdrücke einen unterschiedlichen Wahrheitswert haben.

Bitweise Operatoren

Operator	Bezeichnung	Bedeutung
~	Einerkomplement	$\sim a$ entsteht aus a , indem alle Bits von a invertiert werden
	Bitweises ODER	$a b$ ergibt den Wert, der entsteht, wenn die korrespondierenden Bits von a und b miteinander ODER-verknüpft werden.
&	Bitweises UND	$a \& b$ ergibt den Wert, der entsteht, wenn die korrespondierenden Bits von a und b miteinander UND-verknüpft werden.
^	Bitweises Exklusiv-ODER	$a \wedge b$ ergibt den Wert, der entsteht, wenn die korrespondierenden Bits von a und b miteinander Exklusiv-ODER-verknüpft werden.

Bitweise Operatoren

Operator	Bezeichnung	Bedeutung
>>	Rechtsschieben mit Vorzeichen	$a \gg b$ ergibt den Wert, der entsteht, wenn alle Bits von a um b Positionen nach rechts geschoben werden. Falls das höchstwertige Bit gesetzt ist (a also negativ ist), wird auch das höchstwertige Bit des Resultats gesetzt.
>>>	Rechtsschieben ohne Vorzeichen	$a \ggg b$ ergibt den Wert, der entsteht, wenn alle Bits von a um b Positionen nach rechts geschoben werden. Dabei wird das höchstwertige Bit des Resultats immer auf 0 gesetzt.
<<	Linksschieben	$a \ll b$ ergibt den Wert, der entsteht, wenn alle Bits von a um b Positionen nach links geschoben werden. Das höchstwertige Bit (also das Vorzeichen) erfährt keine besondere Behandlung.

Zuweisungsoperatoren

Operator	Bezeichnung	Bedeutung
=	Einfache Zuweisung	$a = b$ weist a den Wert von b zu und liefert b als Rückgabewert.
+=	Additionszuweisung	$a += b$ weist a den Wert von $a + b$ zu und liefert $a + b$ als Rückgabewert.
-=	Subtraktionszuweisung	$a -= b$ weist a den Wert von $a - b$ zu und liefert $a - b$ als Rückgabewert.
*=	Multiplikationszuweisung	$a *= b$ weist a den Wert von $a * b$ zu und liefert $a * b$ als Rückgabewert.
/=	Divisionszuweisung	$a /= b$ weist a den Wert von a / b zu und liefert a / b als Rückgabewert.
%=	Modulozuweisung	$a \% = b$ weist a den Wert von $a \% b$ zu und liefert $a \% b$ als Rückgabewert.

Zuweisungsoperatoren

Operator	Bezeichnung	Bedeutung
<code>&=</code>	UND-Zuweisung	<code>a &= b</code> weist a den Wert von <code>a & b</code> zu und liefert <code>a & b</code> als Rückgabewert.
<code> =</code>	ODER-Zuweisung	<code>a = b</code> weist a den Wert von <code>a b</code> zu und liefert <code>a b</code> als Rückgabewert.
<code>^=</code>	Exklusiv-ODER-Zuweisung	<code>a ^= b</code> weist a den Wert von <code>a ^ b</code> zu und liefert <code>a ^ b</code> als Rückgabewert.
<code><<=</code>	Linksschiebezuweisung	<code>a <<= b</code> weist a den Wert von <code>a << b</code> zu und liefert <code>a << b</code> als Rückgabewert.
<code>>>=</code>	Rechtsschiebezuweisung	<code>a >>= b</code> weist a den Wert von <code>a >> b</code> zu und liefert <code>a >> b</code> als Rückgabewert.
<code>>>>=</code>	Rechtsschiebezuweisung mit Nullexpansion	<code>a >>>= b</code> weist a den Wert von <code>a >>> b</code> zu und liefert <code>a >>> b</code> als Rückgabewert.

Sonstige Operatoren

- der Fragezeichenoperator ?:
 - ◆ ist der einzige dreistellige Operator in Java
 - ◆ er besteht aus einem logischen Ausdruck und zwei weiteren Ausdrücken, die entweder numerisch, boolean oder von einem Referenztyp sind.
 - ◆ $a ? b : c$ ergibt b , wenn a wahr ist und c , wenn a falsch ist.
- Type-Cast-Operator
 - ◆ dient der expliziten Typumwandlung bei einer einschränkenden Konvertierungen
 - ◆ $(\text{type}) a$ wandelt den Ausdruck a in einen Ausdruck vom Typ type um
- String-Verkettung
 - ◆ mit dem $+$ -Operator können nicht nur mit numerische Operanden verwendet werden, sondern auch zur Verkettung von Strings
 - ◆ $a + b$ ergibt einen verketteten String, wenn einer der Operanden a oder b ein Objekt vom Typ String ist

Anweisungen

- Anweisungen sind elementare Arbeitsschritte
- die leere Anweisung
 - ◆ Syntax
;
 - ◆ die einfachste Anweisung in Java
 - ◆ kann dort verwendet werden, wo syntaktisch eine Anweisung erforderlich, aber von der Programmlogik nicht sinnvoll ist
- der Anweisungsblock
 - ◆ Syntax

```
{  
Anweisung 1;  
Anweisung 2;  
  
...  
}
```
 - ◆ ist eine Zusammenfassung von Anweisungen
 - ◆ wird als einzelne Anweisung behandelt und wird dort verwendet, wo syntaktisch nur eine Anweisung erlaubt ist

Verzweigungen mit der if-Anweisung

- die einfache if-Anweisung
 - ◆ Syntax
if (ausdruck)
 anweisung;
 - ◆ *ausdruck* kann aus einem relationalen Operator oder aus mehreren relationalen Operatoren bestehen, die mit logischen Operatoren verknüpft sind
 - ◆ *anweisung* wird genau dann ausgeführt, wenn *ausdruck true* ist
- die if-else-Anweisung
 - ◆ Syntax
if (ausdruck)
 anweisung1;
else
 anweisung2;
 - ◆ *anweisung1* wird ausgeführt, wenn *ausdruck true* ist
 - ◆ ist *ausdruck false*, wird *anweisung2* ausgeführt

Verzweigungen mit der if-Anweisung

- dangling else

- ◆ betrifft die Zuordnung des else-Zweiges zur innersten if-Verzweigung

- ◆ Syntax

```
if (ausdruck1)  
    if (ausdruck2)
```



```
        anweisung1;  
else
```

```
        anweisung2;
```

- ◆ im Beispiel gehört der *else*-Zweig zu der Anweisung *if* (*ausdruck2*) und nicht, wie es die Einrückung andeutet zu *if* (*ausdruck1*)

- die if-else if-Anweisung

- ◆ Syntax

```
if (ausdruck1)  
    anweisung1;  
else if (ausdruck2)  
    anweisung2;  
else  
    anweisung3;
```

- ◆ *anweisung1* wird ausgeführt, wenn *ausdruck1 true* ist
- ◆ ist *ausdruck1 false*, wird *ausdruck2* ausgewertet
- ◆ ist *ausdruck2 true*, wird *anweisung2* ausgeführt
- ◆ ist auch *ausdruck2 false*, wird *anweisung3* ausgeführt

Verzweigungen mit der switch-Anweisung

- bietet die Möglichkeit der Mehrfachverzweigung
 - ◆ Syntax

```
switch (ausdruck) {  
    case constant1: anweisung1;  
    case constant2: anweisung2;  
    ...  
    default: default_anweisung;  
}
```
 - ◆ der *ausdruck* wird zunächst ausgewertet
 - ◆ das Ergebnis wird mit den Konstanten der *case*-Label verglichen und bei Gleichheit die Anweisung und alle nachstehenden Anweisungen ausgeführt
 - ◆ dies kann durch Einsatz einer *break*-Anweisung verhindert werden
 - ◆ jede Konstante darf nur einmal auftauchen
 - ◆ gibt es keine passende Konstante, wird das optionale *default*-Label am Ende der *switch*-Anweisung angesprungen

Unterschiedliche Schleifenarten

- kopfgesteuerte Schleifen
 - ◆ die Abbruchbedingung steht im Schleifenkopf
 - ◆ wird auch als abweisende Schleife bezeichnet
 - ◆ wird 0, 1 oder n Mal durchlaufen
- fußgesteuerte Schleifen
 - ◆ die Abbruchbedingung steht im Schleifenfuß
 - ◆ wird auch als offene oder nicht abweisende Schleife bezeichnet
 - ◆ wird mind. 1 Mal durchlaufen
- zählende Schleifen
 - ◆ besitzt einen Zähler
 - ◆ hat i.d.R. eine feste Anzahl an Durchläufen
 - ◆ die Abbruchbedingung kann sich in Java nicht nur auf den Zähler beziehen

Kopfgesteuerte Schleifen

- in Java realisiert durch die while-Schleife
 - ◆ Syntax
while (ausdruck)
 anweisung;
 - ◆ die Abbruchbedingung *ausdruck* muss als Ergebnis einen Wert vom Typ boolean zurückliefern
 - ◆ ist das Ergebnis von *ausdruck true*, wird die Anweisung *anweisung*; ausgeführt
 - ◆ liefert *ausdruck* als Ergebnis *false*, wird mit der ersten Anweisung nach der Schleife fortgefahren
 - ◆ ist das Ergebnis von *ausdruck* gleich zu Beginn *false*, werden die Anweisungen in der Schleife nie ausgeführt
➔ daher der Begriff: abweisende Schleife

Fußgesteuerte Schleifen

- in Java realisiert durch die do-while-Schleife
 - ◆ Syntax

```
do
    anweisung1;
while (ausdruck);
```
 - ◆ zunächst wird *anweisung1*; ausgeführt
 - ◆ erst danach wird das Ergebnis der Abbruchbedingung *ausdruck* geprüft
 - ◆ ist das Ergebnis von *ausdruck true*, wird die Schleife ein weiteres Mal durchlaufen
 - ◆ ist das Ergebnis von *ausdruck false*, wird die Schleifenverarbeitung beendet
 - ◆ da die Bedingung erst nach dem ersten Durchlauf geprüft wird, muss die Schleife mindestens einmal durchlaufen werden
➔ daher der Begriff: nicht abweisende Schleife

Zählende Schleifen

- in Java realisiert durch die for-Schleife
 - ◆ Syntax

```
for (init; test; update)
    anweisung1;
```
 - ◆ der Schleifenkopf besteht aus drei optionalen Ausdrücken
 - ◆ der *init*-Teil wird i.d.R. zur Initialisierung von einem oder mehreren Schleifenzählern verwendet, die nur lokal innerhalb der Schleife bekannt sind
 - ◆ der *test*-Teil bildet die Abbruchbedingung der Schleife
 - ◆ *anweisung1* wird nur dann ausgeführt, wenn *test* als Ergebnis *true* liefert
 - ◆ fehlt der *test*-Teil, setzt der Compiler an diese Stelle die Konstante *true*
 - ◆ im *update*-Teil werden die Schleifenzähler verändert
 - ◆ besondere for-Schleife zum Durchlaufen von Feldern (Arrays)

```
for ( Typ Bezeichner : Feld )
```

Die break- und continue-Anweisung

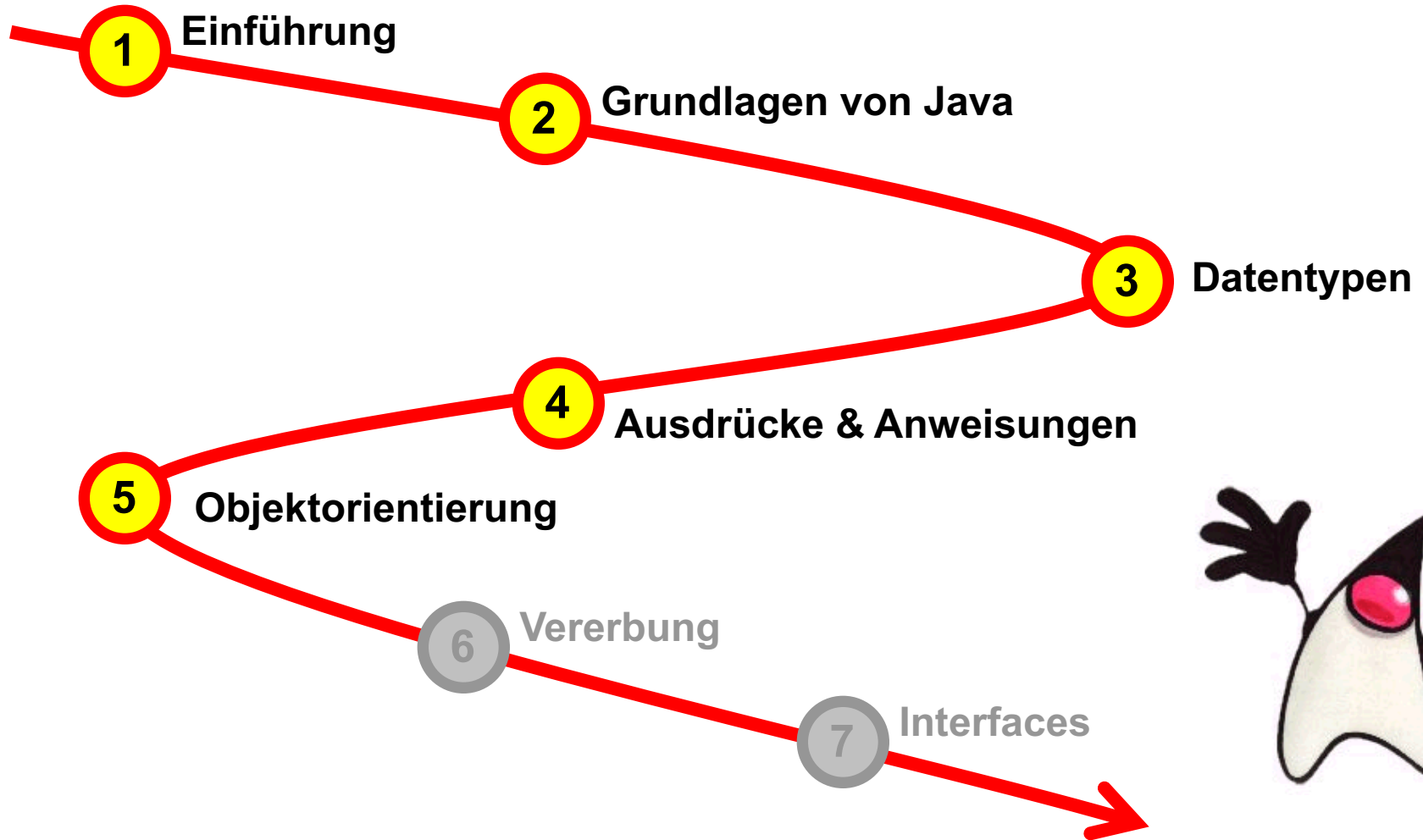
- beide Anweisungen dienen der Änderung des Ablaufs innerhalb von Schleifen
- die break-Anweisung
 - ◆ eine break-Anweisung innerhalb einer Schleife führt zum Verlassen der Schleife
 - ◆ das Programm wird mit der ersten Anweisung nach der Schleife fortgesetzt
- die continue-Anweisung
 - ◆ der aktuelle Schleifendurchlauf wird beendet und das Programm springt zum Ende der Schleife
 - ◆ ein neuer Durchlauf der Schleife mit Prüfung der Abbruchbedingung beginnt



Programmierung 1

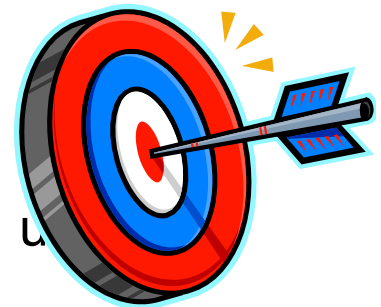
Kapitel 5 Objektorientierung

Themenüberblick

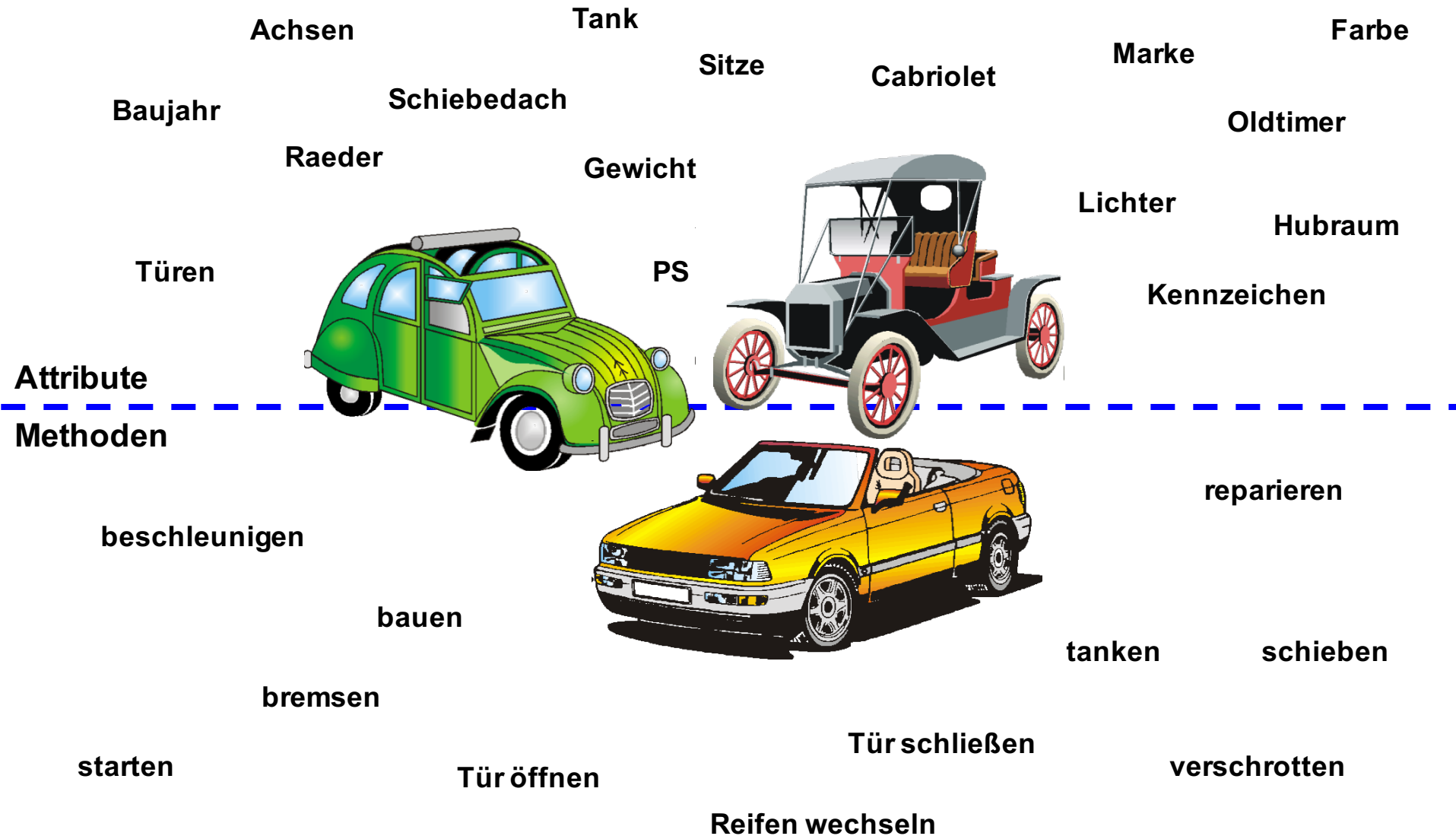


Lernziele

- Sie kennen die Begriffe Klasse, Objekt, Attribut und Methode..
- Sie kennen die Eigenschaften von Attributen und Methoden.
- Sie können Klassen, Attribute und Methoden in der Unified Modelling Language (UML) darstellen.
- Sie können Objekte mithilfe von Konstruktoren erzeugen.
- Sie können das Prinzip der Kapselung im Zusammenhang mit Getter- und Setter-Methoden erläutern.
- Sie kennen die unterschiedlichen Sichtbarkeiten von Attributen und Methoden.
- Sie können Methoden überladen.
- Sie kennen die Begriffe der Klassenattribute und –methoden.
- Sie können Objekte mit dem Garbage Collector zerstören.
- Sie kennen die Begriffe Assoziation, Aggregation und Komposition.



Einführung in die Objektorientierung





Einführung in die Objektorientierung

Klassen

- Klassen bilden den Bauplan (die Schablone) für Objekte
- nach dem Bauplan können mehrere Objekte erzeugt werden

Objekte

- stellen die sog. Instanz einer Klasse dar
- können erzeugt, verändert und zerstört werden
- haben eine Identität und einen Zustand

Attribute

- beschreiben die Eigenschaften von Objekten
- beschreiben den Zustand eines Objektes

Methoden

- beschreiben das Verhalten von Objekten
- stellen die Funktionalität von Objekten dar

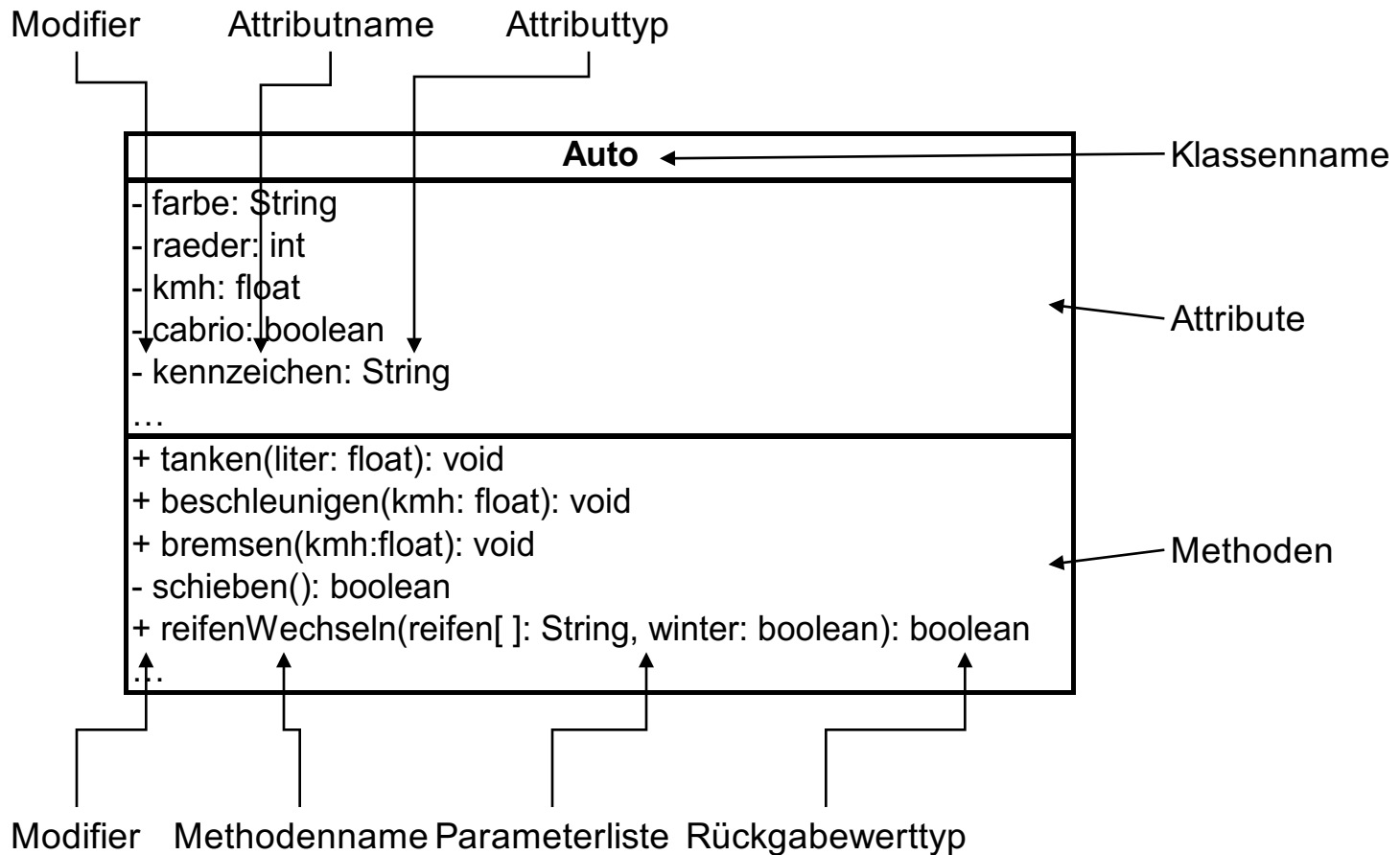
Attribute und Methoden

- Attribute sind Variable oder Konstanten innerhalb eines Objektes
- Attribute werden wie Variablen oder Konstanten deklariert
- in den Methoden werden die Algorithmen zu einem Objekt implementiert
- Methoden können einen Rückgabewert haben
- der Rückgabewert einer Methode kann von einem einfachen oder einem Referenzdatentyp oder vom Typ void sein
- Methoden vom Rückgabewerttyp void haben keinen Rückgabewert
- die Deklaration von Methoden erfolgt nach folgender Syntax
[Modifier] Typ Name ([Übergabeparameter]) {
 Anweisungen;
}
- sowohl Attribute und Methoden haben einen Modifier, der die Sichtbarkeit außerhalb der Klassen festlegt

Darstellung von Klassen in der UML

- UML steht für Unified Modelling Language
- wird im Rahmen der objektorientierten Analyse und im objektorientierten Design eingesetzt
- beschreibt eine standardisierte Sammlung von Diagrammen
- die Diagramme werden unterschieden in
 - ◆ statische Diagramme
 - ◆ dynamische Diagramme
- Klassen werden durch Klassendiagramme dargestellt
- Klassendiagramme zählen zu den statischen Diagrammen
- in den Klassendiagrammen werden festgelegt
 - ◆ der Name der Klasse
 - ◆ die Namen und Datentypen der Attribute
 - ◆ die Namen und Rückgabewerte der Methoden
 - ◆ die Sichtbarkeit von Attributen und Methoden

Darstellung der Klassen in der UML



Objekte erzeugen

- Objekte sind Instanzen einer Klasse
- zum Erzeugen von Objekten muss zunächst eine Klasse definiert werden
 - ◆ die Definition einer Klasse beginnt mit dem Schlüsselwort `class`
 - ◆ innerhalb der Klasse werden die entsprechenden Attribute und Methoden zu der Klasse deklariert und implementiert
 - ◆ die Definition einer Klasse beschreibt also den Aufbau und das Verhalten einer Klasse
- Objekte werden wie folgt erzeugt
 - ◆ zunächst wird eine Variable vom Typ der Klasse deklariert
 - ◆ mit Hilfe des `new`-Operators wird ein neues Objekt der Klasse instanziiert und muss der Variable zugewiesen werden
 - ◆ durch Verwendung des `new`-Operators wird gleichzeitig für das neue Objekt Speicher allokiert
 - ◆ mit dem `new`-Operator wird eine spezielle Methode der Klasse implizit aufgerufen, der Konstruktor

Konstruktoren

- sind spezielle Methoden zum Erzeugen von neuen Objekten einer Klasse
- Konstruktoren tragen den gleichen Namen wie die Klasse
- werden ausschließlich über den new-Operator aufgerufen
- liefern als Ergebnis die Referenz auf das neu geschaffene Objekt zurück
- es existiert ein Standardkonstruktor
 - ◆ dieser wird automatisch vom Compiler zur Verfügung gestellt, wenn in der Klassendefinition kein Konstruktor implementiert wurde
 - ◆ besitzt keine Übergabeparameter
 - ◆ sobald ein Konstruktor in der Klasse implementiert wurde, erzeugt der Compiler keinen Standardkonstruktor mehr
- Konstruktoren können dazu verwendet werden, um die Attribute zu initialisieren

Beispiel für die Implementierung einer Klasse

```
package prog1.demos.objekt;
class Auto {

    // Deklaration der Attribute
    int ps;
    float kmh;
    String kfzKZ;
    String marke;

    // Implementierung des Konstruktors
    Auto(){
        ps = 75;
        kmh = 0;
        kfzKZ = "XX-XX 0000";
        marke = "Eigenbau";
    }

    // Implementierung der Methoden
    void beschleunigen(float pluskmh){
        kmh += pluskmh;
    }
    void bremsen(float minuskmh){
        if (kmh - minuskmh >= 0){
            kmh -= minuskmh;
        }
    }
}
```

Prinzip der Kapselung

- die Kapselung ist ein Programmiergrundsatz in der objektorientierten Programmierung
- es handelt sich dabei um keine Eigenschaft der Programmiersprache Java
- Idee: auf die Attribute eines Objektes wird nicht direkt, sondern über spezielle Methoden zugegriffen
- dazu müssen die Attribute vor der Außenwelt verborgen und vor unerlaubtem Zugriff geschützt werden
- dies geschieht durch die Sichtbarkeitsmodifizier
- Ziel: die Werte der Attribute dürfen nur im Sinne des Entwicklers sinnvoll verändert werden -> Plausibilitätsprüfung
- die speziellen Änderungsmethoden werden als Getter- und Setter-Methoden bezeichnet
 - ◆ Getter-Methoden dienen zum Auslesen der Attribute
 - ◆ Setter-Methoden werden zum Verändern der Attribute verwendet

Sichtbarkeit von Attributen und Methoden

- **private**
 - ◆ nur innerhalb der Klasse sichtbar
 - ◆ stärkste Form der Kapselung, da auf die Attribute und Methoden nur innerhalb der Klasse zugegriffen werden kann
- **protected**
 - ◆ nur innerhalb eines Paketes und in Subklassen (siehe Kapitel 6: Vererbung) sichtbar
- **default**
 - ◆ nur innerhalb des Paketes sichtbar
- **public**
 - ◆ von überall sichtbar
 - ◆ normalerweise sind die Getter- und Setter-Methoden öffentlich, um auf private Attribute zuzugreifen

Beispiel für Getter- und Setter-Methoden

```
package prog1.demos.objekt;
class Auto {

    // Deklaration der gekapselten Attribute
    private int ps;
    private float kmh;
    private String kfzKZ;
    private String marke;

    // Getter- und Setter-Methoden
    public String getKfzKZ() {
        return kfzKZ;
    }

    public void setKfzKZ(String kfzKZ) {
        this.kfzKZ = kfzKZ;
    }

    public float getKmh() {
        return kmh;
    }

    public void setKmh(float kmh) {
        this.kmh = kmh;
    }
}
```


Zugriff auf Attribute und Methoden

- auf Methoden und Attribute wird in der Punktnotation zugegriffen
objektname.methode(Übergabeparameter);
objektname.attribut;
- Voraussetzung: die Sichtbarkeit der Methoden und Attribute lässt den direkten Zugriff zu
- call by value
 - ◆ alle Parameter werden in Java mit call by value übergeben
 - ◆ Änderungen der Werte innerhalb der Methoden haben keinen Einfluss auf den Übergabeparameter
- call by reference
 - ◆ werden Referenzdatentypen (Objekte) übergeben, steht in der Methode die Referenz auf das Originalobjekt zur Verfügung
 - ◆ alle Methoden und Attribute zum Originalobjekt stehen in der Methode zur Verfügung
 - ◆ Änderungen der Attribute finden somit innerhalb des Originalobjektes statt

Beispiel für Methoden- und Attributzugriffe

```
package prog1.demos.objekt;
class Auto {

    // Deklaration der Attribute
    private int ps;
    private float kmh;
    private String kfzKZ;
    private String marke;
    public int tueren;

    // Getter- und Setter-Methoden
    public String getKfzKZ() {
        return kfzKZ;
    }

    public void setKfzKZ(String kfzKZ) {
        this.kfzKZ = kfzKZ;
    }

    public float getKmh() {
        return kmh;
    }

    public void setKmh(float kmh) {
        this.kmh = kmh;
    }

}
```

```
package prog1.demos.objekt;
class AutoTest {
    public static void main(String[] args) {

        Auto bmw = new Auto();
        Auto audi = new Auto();

        bmw.tueren = 5;
        audi.tueren = 3;

        bmw.setKfzKZ("HD-XX 321");
        audi.setKmh(135.7f);

        System.out.println("Der BMW hat das  
Kennzeichen: " + bmw.getKfzKZ());

        System.out.println("Der Audi fährt: " +  
audi.getKmh());

    }
}
```

Überladene Methoden

- Methoden mit dem gleichen Namen werden innerhalb einer Klasse mehrmals definiert
- die Methoden unterscheiden sich dabei in der Anzahl und/oder in der Typisierung der Übergabeparameter
- die Modifier können dabei unterschiedliche Sichtbarkeiten zulassen
- Ziel: gleichnamige Operationen können mit unterschiedlichen Datentypen als Parameter versorgt werden
- gleichnamige Operationen können unterschiedliche Funktionalität zur Verfügung stellen
- auch Konstruktoren können überladen werden

Beispiel für überladene Methoden

```
package prog1.demos.objekt;
class Auto {

    // Deklaration der gekapselten Attribute
    private int ps;
    private float kmh;
    private String kfzKZ;
    private String marke;

    // überladene Konstruktoren und Methoden
    Auto(){
        ps = 75; kmh = 0;
        kfzKZ = "XX-XX 0000"; marke = "Eigenbau";
    }

    public Auto(int ps, float kmh, String kfzKZ, String marke){
        this.ps = ps; this.kmh = kmh;
        this.kfzKZ = kfzKZ; this.marke = marke;
    }

    void beschleunigen(float pluskmh){
        kmh += pluskmh;
    }
    protected void beschleunigen(){
        kmh += 10;
    }
}
```

Klassenattribute und –methoden

- Klassenattribute und –methoden existieren unabhängig von einer Instanz einer Klasse
- ohne das ein Objekt existiert können diese verwendet werden
- sie werden durch den Modifier static als Klassenattribut bzw. –methode definiert
- Problem: static-Methoden können normalerweise nur auf static-Attribute zugreifen, und nicht auf Instanzattribute und -methoden
- Lösungen
 - ◆ Alternative 1
Erzeugen einer statischen Referenzvariable, mit der auf den Instanzkontext zugegriffen werden kann
 - ◆ Alternative 2
Erzeugen eine lokale Referenzvariable innerhalb einer statischen Methode, mit der auf den Instanzkontext zugegriffen werden kann
- auf Klassenattribute und –methoden wird ebenfalls in der Punktnotation über den Klassennamen zugegriffen

Beispiel für Klassenattribute und -methoden

```
package prog1.demos.objekt;
class Auto {

    // Deklaration der gekapselten Attribute
    private int ps;
    private float kmh;
    private String kfzKZ;
    private String marke;
    private static int autoZaehler = 0;

    // Getter- und Setter-Methoden
    Auto(){
        autoZaehler++;
        ps = 75;
        kmh = 0;
        kfzKZ = "XX-XX 0000";
        marke = "Eigenbau";
    }

    public static int getAutoZaehler() {
        return autoZaehler;
    }

    public static void setAutoZaehler(int autoZaehler) {
        Auto.autoZaehler = autoZaehler;
    }

}
```

```
package prog1.demos.objekt;
class AutoTest {
    public static void main(String[] args) {

        System.out.println(Auto.getAutoZaehler());

        Auto bmw = new Auto();
        Auto audi = new Auto();

        System.out.println(Auto.getAutoZaehler());

        bmw.tueren = 5;
        audi.tueren = 3;

        bmw.setKfzKZ("HD-XX 321");
        audi.setKmh(135.7f);

        System.out.println("Der BMW hat das  
Kennzeichen: " + bmw.getKfzKZ());

        System.out.println("Der Audi fährt: " +  
audi.getKmh());

    }
}
```

Objekte löschen mit dem Garbage-Collector

- nicht mehr benötigte Objekte belasten den Speicher und müssen daher „eingesammelt“ werden
- diese Aufgabe übernimmt der Garbage Collector
- benötigte Objekte sind dadurch gekennzeichnet, dass sie noch referenziert sind (reachable)
- sobald keine Referenzvariable mehr auf das Objekt verweist, wird das Objekt vom Garbage Collector zerstört (unreachable)
- Java verfügt über eine automatische Garbage Collection
- der Garbage Collector in Java startet, wenn die Virtual Machine für neue Objekte Speicherplatz benötigt
- explizit kann der Garbage Collector auch über den Befehl `System.gc();` gestartet werden
- der Garbage Collector ruft den Destruktor eines Objektes
- Destruktoren sind parameterlose Methoden mit dem Namen `finalize();`

Weitere Klassenarten

- neben den „normalen“ Klassen können in Java auch innere, lokale und anonyme Klassen definiert werden
- sie dienen u.a. zur Strukturierung von Applikationen
- der Zugriff von außen kann durch die bekannten Modifier beschränkt werden
- werden innerhalb der Java-Bibliotheken intensiv genutzt
- diese werden als eigene Byte-Code-Dateien gespeichert
- innere Klassen werden innerhalb einer Klasse parallel zu den Attributen und Methoden deklariert
- lokale Klassen werden innerhalb eines Anweisungsblocks in einer Methode deklariert und sind nur innerhalb dieser Methode verfügbar
- anonyme Klassen werden innerhalb eines Anweisungsblocks in einer Methode deklariert, instanziiert und besitzt keinen eigenen Namen

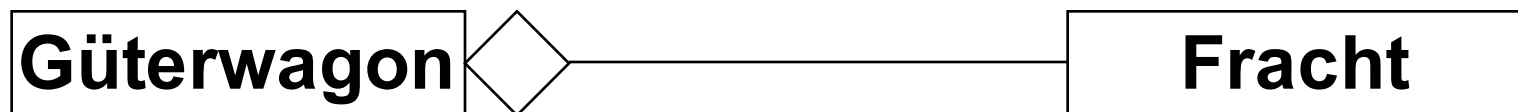
Assoziation

- beschreibt die Struktur einer Menge von Beziehungen zwischen Objekten
- binäre Assoziationen beschreiben die Beziehung zwischen je zwei Klassen
- Assoziationen können benannt werden, durch eine verbale Beschreibung der Beziehung
- die Darstellung in der UML erfolgt durch eine einfache Verbindungslinie zwischen den Klassen



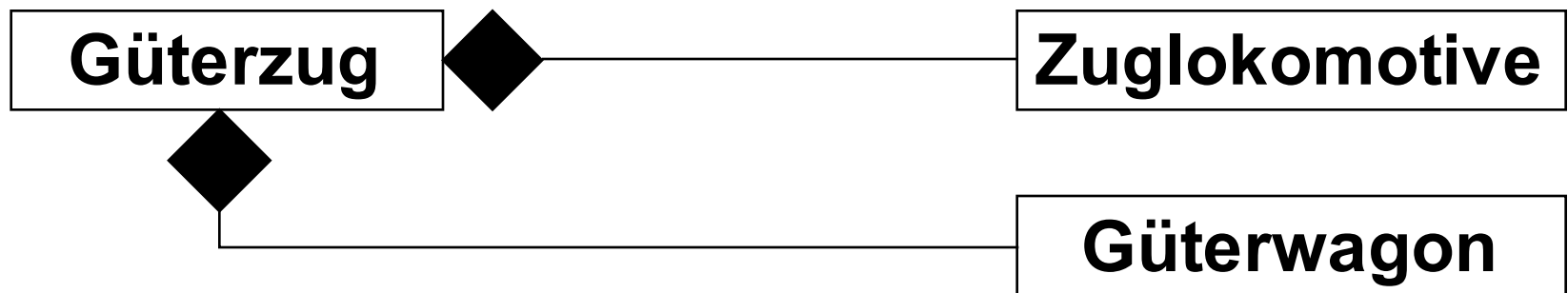
Aggregation

- Aggregationen stellen eine spezielle Form der Assoziation dar
- Aggregationen stellen eine „Teile-Ganzes-Beziehung“ oder „Besteht-aus-Beziehung“ dar
- Aggregationen beschreiben keine existentielle Abhängigkeiten
- Beispiel: die Beziehung zwischen einem Güterwagon und der Fracht; der Wagon existiert auch ohne Fracht
- die Notation in der UML erfolgt durch eine Linie mit einer Raute am Ende, wobei die Raute bei der Aggregatsklasse (dem Ganzen) angesiedelt ist



Komposition

- die Komposition ist eine spezielle Form der Aggregation
- Kompositionen sind „strenger“ als Aggregationen
- es handelt sich ebenfalls um eine „Teile-Ganzes-Beziehung“
- die Existenz des Ganzen ist im Unterschied zur Aggregation von der Existenz des einzelnen Teils abhängig
- Beispiel: ein Güterzug kann nur dann existieren, wenn mind. eine Zuglokomotive und mind. ein Güterwagen existiert
- die Darstellung erfolgt analog der Aggregation nur mit einer ausgefüllten Raute

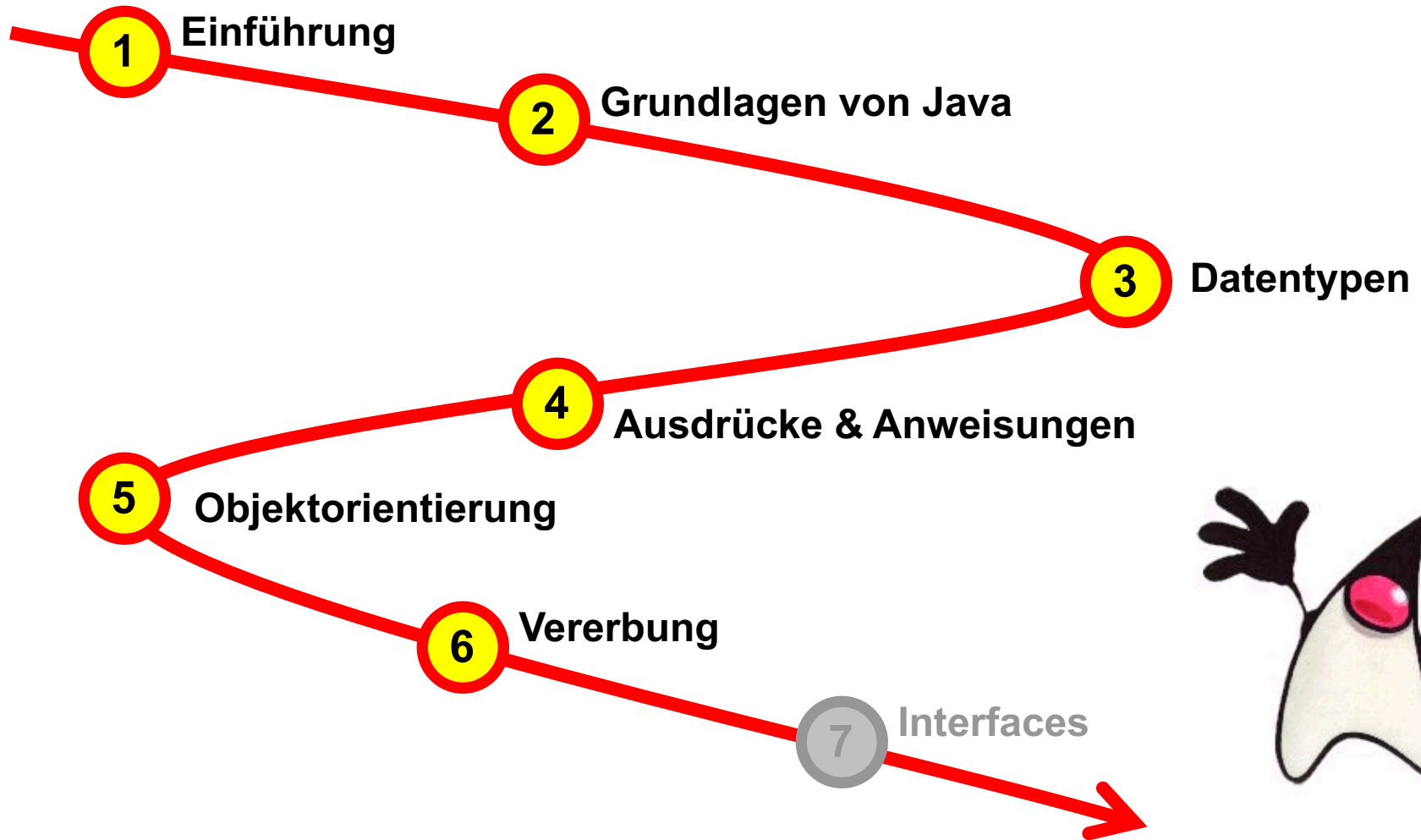




Programmierung 1

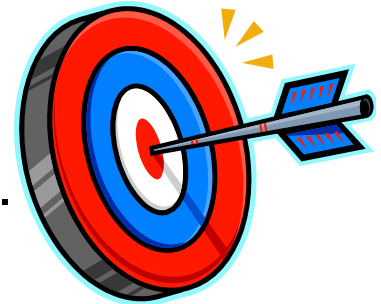
Kapitel 6 Vererbung

Themenüberblick



Lernziele

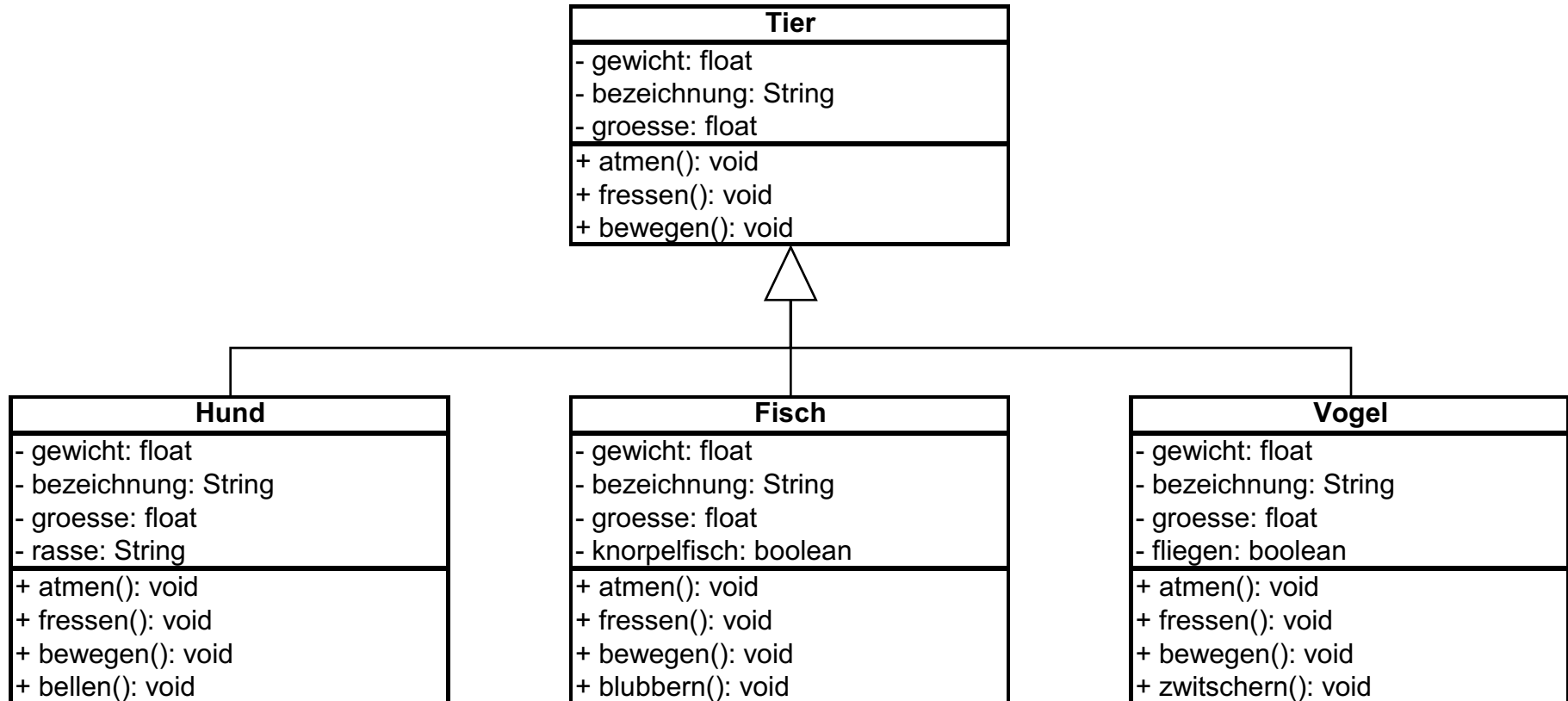
- Sie können die Eigenschaften der Vererbung beschreiben.
- Sie können den Unterschied von Super- und Subklassen beschreiben und den Begriff der Vererbungshierarchie erläutern.
- Sie können die Begriffe Generalisierung und Spezialisierung im Zusammenhang mit der Vererbung erklären.
- Sie können Vererbungsbeziehungen in der UML darstellen.
- Sie können das Vererbungskonzept in Java beschreiben.
- Sie kennen die Eigenschaften der Klasse Object.
- Sie können Methoden überschreiben.
- Sie kennen die Bedeutung der Modifier abstract und final.
- Sie können die Bedeutung von this und super erklären.
- Sie kennen die Konzepte des narrowing und widening casts.
- Sie können den Begriff der Polymorphie erklären.



Eigenschaften der Vererbung

- eine Unterklasse (Subklasse) erbt die Attribute und Methoden einer Oberklasse (Superklasse)
- statt Vererbung spricht man auch vom Ableiten von Klassen
→ eine Subklasse leitet sich aus einer oder mehreren Superklassen ab
- es entsteht eine Vererbungshierarchie, die theoretisch beliebig tief geschachtelt werden kann
- i.d.R. besitzt die Subklasse noch zusätzliche Attribute und Methoden
- Ziel und Vorteil: bestehender Programmcode kann wieder verwendet werden (Reuse)
- Mehrfachvererbung besagt, dass eine Subklasse von mehreren Superklassen erbt
- Superklassen stellen eine Generalisierung ihrer Subklassen dar
- umgekehrt sind Subklassen Spezialisierungen ihrer Superklassen

Darstellung der Vererbung in der UML



Vererbung in Java

- Java unterstützt nur die Einfach- und nicht die Mehrfachvererbung
- über Interfaces (Kapitel 7) kann auf mehrere Klassen Bezug genommen werden
- die Vererbung ist durch folgende Syntax definiert:
`class Subklasse extends Superklasse { }`
- mit `extends` verweist die Subklasse auf ihre Superklasse
- nach `extends` darf lediglich eine Superklasse angegeben werden
- alle Attribute und Methoden mit Ausnahme der Konstruktoren der Superklasse werden an die Subklasse vererbt
- in der Subklasse kann die Funktionalität der abgeleiteten Klasse erweitert und verändert werden
 - ◆ Hinzufügen von Methoden und Attributen
 - ◆ Überladen von Methoden
 - ◆ Überschreiben von Methoden

Die Superklasse Object

- ist die Wurzel der Klassenhierarchie in Java
- liegt im Paket `java.lang`
- von ihr sind alle Klassen explizit oder implizit abgeleitet, d.h. jede eigene Klasse ist immer eine Subklasse von `Object`
- damit stehen alle Methoden der Klasse `Object` in allen abgeleiteten Klassen zur Verfügung
 - ◆ `Object()` – parameterloser Konstruktor
 - ◆ `toString()` – Konvertierung eines Objekts in einen String ausgeführt
 - ◆ `equals()` – vergleicht zwei Objekte miteinander
 - ◆ `hashCode()` – berechnet den Hashwert eines Objektes
 - ◆ `clone()` – kopiert zwei Objekte
 - ◆ `finalize()` – ist der Destruktor
- in Referenzvariablen vom Typ `Object` können alle beliebigen Objektreferenzen gespeichert werden (vgl. Narrowing cast)

Sichtbarkeit von Methoden und Attributen

- **private**
 - ◆ nur innerhalb der Klasse sichtbar
 - ◆ stärkste Form der Kapselung, da auf die Attribute und Methoden nur innerhalb der Klasse zugegriffen werden kann
- **protected**
 - ◆ nur innerhalb eines Paketes und in Subklassen (siehe Kapitel 6: Vererbung) sichtbar
- **default**
 - ◆ nur innerhalb des Paketes sichtbar
- **public**
 - ◆ von überall sichtbar
 - ◆ normalerweise sind die Getter- und Setter-Methoden öffentlich, um auf private Attribute zuzugreifen

Methoden überschreiben

- Methoden, die bereits in der Superklasse implementiert sind, können in Subklassen neu definiert werden (Überschreiben)
- zur Laufzeit wird entschieden, welche Methode konkret ausgeführt wird (dynamisches Binden)
 - ◆ dabei sucht die JVM zunächst in der Klasse des Referenzdatentyps nach einer passenden Methode
 - ◆ wird dort keine passende Methode gefunden, wird die Vererbungshierarchie von unten nach oben nach einer passenden Methode durchsucht
- dies erfordert zusätzliche Rechnerkapazität und wirkt sich daher negativ auf die Performance aus
- das Überschreiben kann durch zwei Modifier vermieden werden
 - ◆ `private` → Methode ist in Subklasse nicht sichtbar
 - ◆ `final` → verhindert explizit das Überschreiben
- die Sichtbarkeit bei überschriebenen Methoden darf erhöht aber nicht eingeschränkt werden

Weitere Modifier

- **abstract**

- ◆ abstrakte Klassen sind i.d.R. noch nicht ausreichend spezialisiert und dienen nur als grobe Vorlage für Subklassen
- ◆ von abstrakten Klassen können keine Objekte erzeugt werden
- ◆ beinhaltet eine Klasse mind. eine abstrakte Methode, so muss die Klasse abstrakt werden
- ◆ abstrakte Methoden beinhalten nur den Methodenkopf, aber keine Implementierung im Methodenrumpf
- ◆ Subklassen von abstrakten Superklassen müssen alle abstrakten Methoden der Superklasse implementieren (konkrete Klasse) oder selbst abstrakt werden

- **final**

- ◆ Klassen, die mit dem Modifier final versehen sind, dürfen nicht weiter abgeleitet werden
- ◆ Methoden mit dem Modifier final dürfen nicht überschrieben werden
- ◆ final-Attribute sind Konstanten, deren Wert sich nicht verändern darf

Bedeutung von this und super

- mit dem Ausdruck super kann aus einer abgeleiteten Klasse mithilfe der Punktnotation auf Attribute und Methoden der Superklasse zugegriffen werden
- ein kaskadierender Aufruf super.super.x() ist nicht möglich
- this ist eine Referenzvariable, die beim Anlegen des Objekts automatisch erzeugt wird und auf das aktuelle Objekt zeigt
- über die this-Referenz können eigene Methoden, Instanz- und Klassenattribute angesprochen werden
- mithilfe von this(Übergabeparamter) oder super(Übergabeparameter) können Konstruktoren verkettet werden
- this() und super() müssen bei verketteten Konstruktoraufrufen immer die als erste Anweisung im Konstruktor stehen
- this() verweist auf Konstruktoren der aktuellen und super() auf Konstruktoren der Superklasse

Narrowing cast

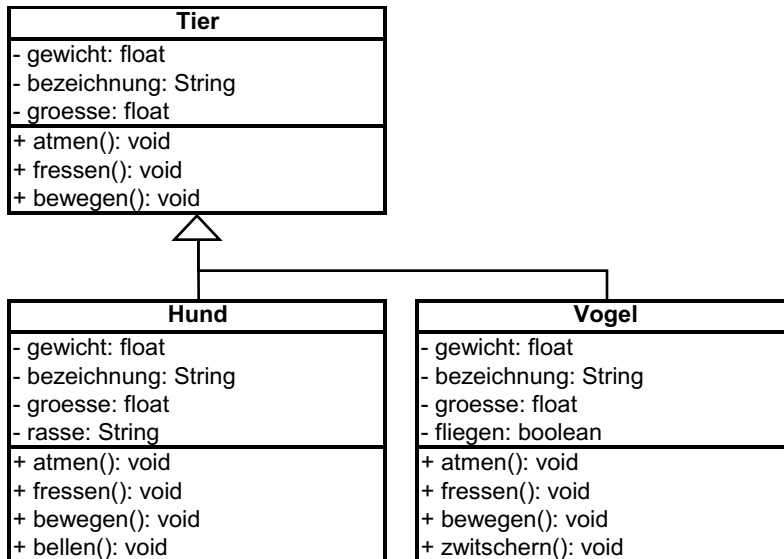
- Objekte von Subklassen können in Referenzvariablen gespeichert werden, die vom Typ ihrer Superklasse sind
- dabei wird nur die Sicht auf die Objekte beschränkt, d.h. es sind nur noch die Attribute und Methoden sichtbar, die in der Superklasse deklariert sind
- die subklassen-spezifischen Attribute und Methoden sind noch vorhanden, aber vorübergehend ausgeblendet
- der narrowing cast beschreibt also den Wechsel von einer Sicht mit mehr auf eine Sicht mit weniger Details bei einem Objekt
- der narrowing cast ist ein wesentliches Prinzip im Vererbungskonzept von Java
- der narrowing cast ist eine wesentliche Voraussetzung für die Polymorphie

Widening cast

- stellt die Umkehrung des narrowing cast dar
- Referenzen auf Objekte, die in einer Referenzvariable vom Typ der Superklasse gespeichert sind, sollen in einer Referenzvariable vom Typ der Subklasse gespeichert werden
- es handelt sich dabei um eine unsichere Konvertierung
 - ◆ es muss sichergestellt werden, dass es sich bei den Referenzen um Referenzen auf Objekte der Subklasse handelt
 - ◆ vor der Umwandlung muss der Typ der Referenz mit dem Operator `instanceof` überprüft werden
 - ➔ `referenzvariable instanceof Subklasse` muss `true` ergeben
 - ◆ bei der Durchführung muss ein expliziter Cast auf den Typ der Subklasse durchgeführt werden
`refSubklasse = (Subklasse)referenzdatentyp;`
- der widening cast wechselt von einer Sicht mit weniger auf eine Sicht mit mehr Details und blendet die subklassen-spezifischen Attribute und Methoden wieder ein

Polymorphie

- neben der Kapselung und der Vererbung ein weiteres Grundkonzept in der objektorientierten Programmierung
- bedeutet die Vielgestaltigkeit von Objekten
- basiert auf dem Konzept des dynamischen Bindens
- gleiche Methodenaufrufe rufen unterschiedliche Verhaltensweisen der Objekte hervor
- ein einfaches Beispiel



```
package prog1.demos.objekt;
class AutoTest {
    public static void main(String[] args) {

        Tier[] x = new Tier[2];

        x[0] = new Hund(); //Narrowing Cast
        x[1] = new Vogel(); //Narrowing Cast

        x[0].atmen();
        x[1].atmen();

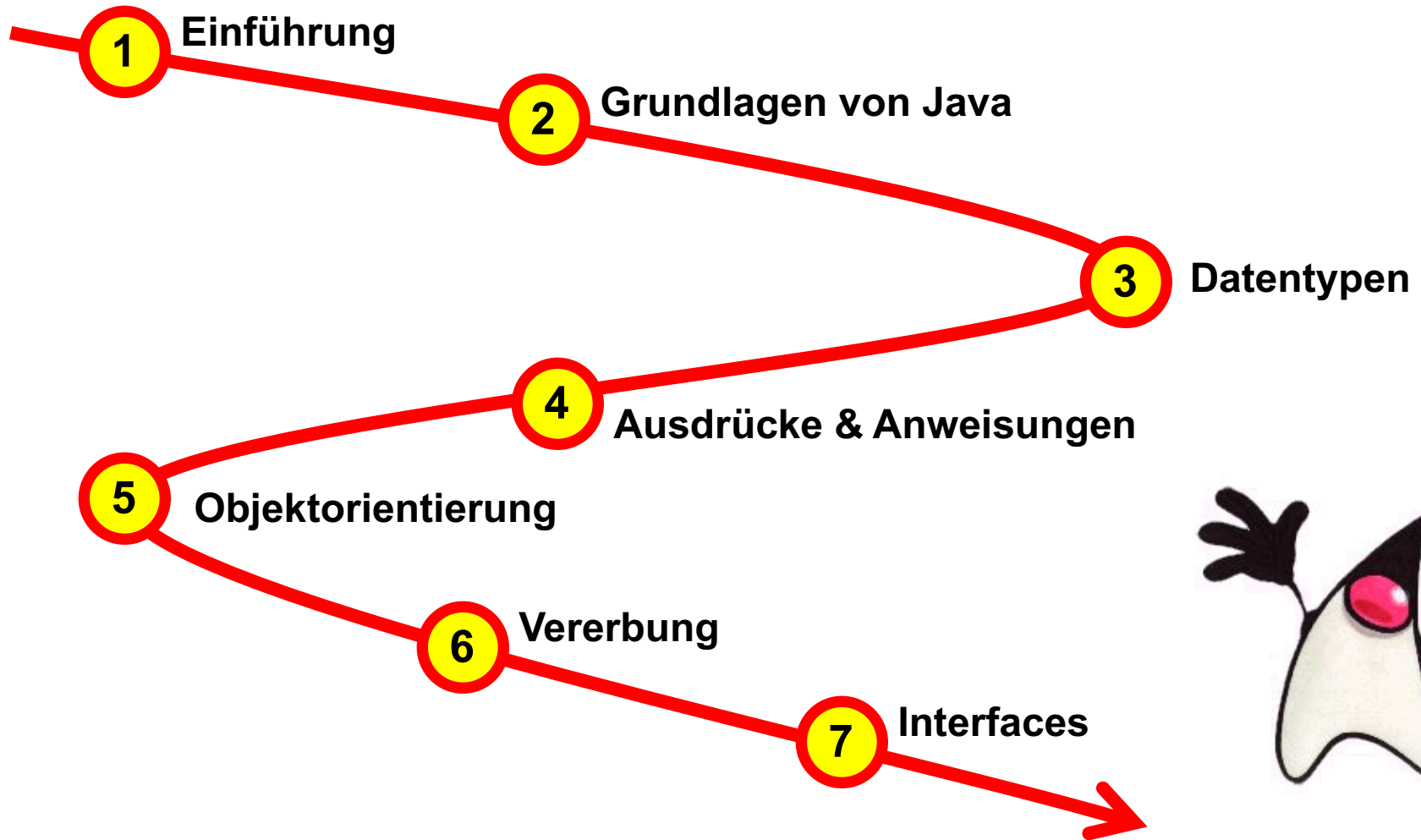
    }
}
```



Programmierung 1

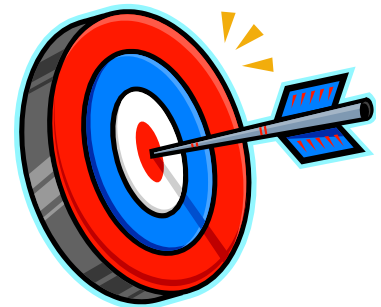
Kapitel 7 Interfaces

Themenüberblick



Lernziele

- Sie können die wesentlichen Eigenschaften von Interfaces beschreiben.
- Sie können Interfaces deklarieren und implementieren.
- Sie können Interfaces vererben.
- Sie können Interfaces in der UML darstellen.
- Sie kennen die besondere Bedeutung von Interfaces in Bezug auf Mehrfachvererbung.
- Sie können Polymorphie mit Hilfe von Interfaces realisieren.
- Sie können Objekte kopieren.



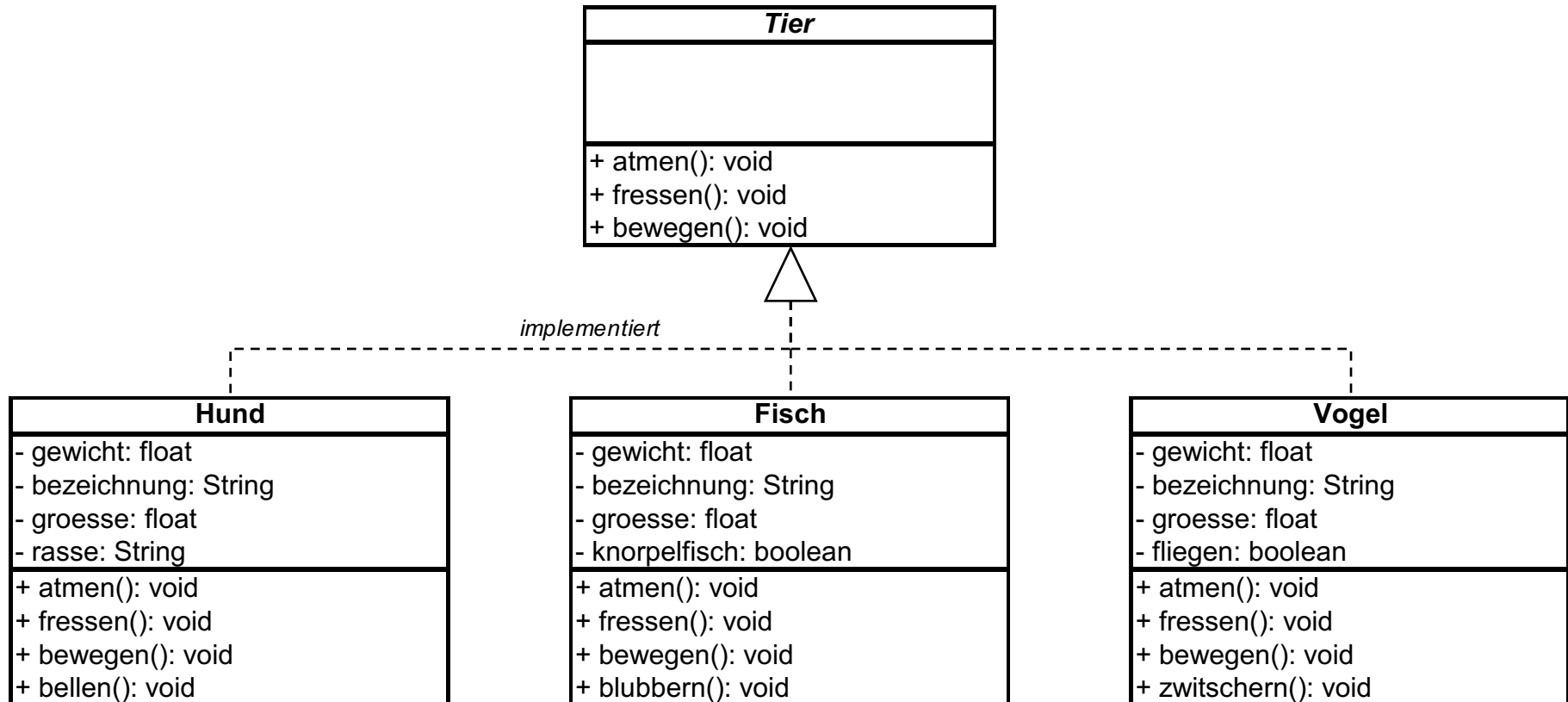
Eigenschaften von Interfaces

- Interfaces stellen eine besondere Form der Klassen dar
- sie definieren lediglich den Aufbau der Schnittstelle
 - ◆ Name von Methoden
 - ◆ Signaturen der Methoden
 - ◆ Attribute
- Interfaces beinhalten keine eigene Funktionalität
- die Funktionalität wird in Klassen implementiert, die das Interface nutzen wollen
- sie beinhalten ausschließlich abstrakte öffentliche Methoden
- alle Attribute sind als Konstanten festgelegt
- Deklaration von Interfaces mit dem Schlüsselwort *interface*
- Interfaces können von anderen Interfaces mit *extends* abgeleitet (vererbt) werden
- Interfaces können genau wie Klassen als Datentyp für Referenzvariablen genutzt werden

Implementierung von Interfaces

- um zu einem Interface die Funktionalität zur Verfügung zu stellen, muss eine Klasse das Interface implementieren
- dies erfolgt durch den Zusatz *implements* zur *class*-Anweisung
z.B. *class xyz implements abc { }*
- die Klasse muss alle Methoden des Interfaces implementieren oder als abstrakte Klasse deklariert werden
- Klassen können im Gegensatz zur Vererbung mehrere Interfaces implementieren
z.B. *class xyz implements abc, def { }*
- durch die Implementierung mehrerer Interfaces wird eine Art von Mehrfachvererbung realisiert
- Objekte einer Klasse sind zu allen Interface-Datentypen kompatibel, sobald die Klasse das Interface implementiert
z.B. alle Objekte der Klasse *xyz* könnten in Referenzvariablen vom Typ *abc* oder *def* gespeichert werden

Darstellung in der UML



Polymorphie über Interfaces

- analog der Polymorphie durch Vererbung
- Unterschied
 - ◆ Polymorphie durch Vererbung: Objekte einer Subklasse werden in Referenzvariablen vom Typ der Superklasse gespeichert
 - ◆ Polymorphie durch Interfaces: Objekte der implementierenden Klasse werden in Referenzvariablen vom Typ des implementierten Interfaces gespeichert

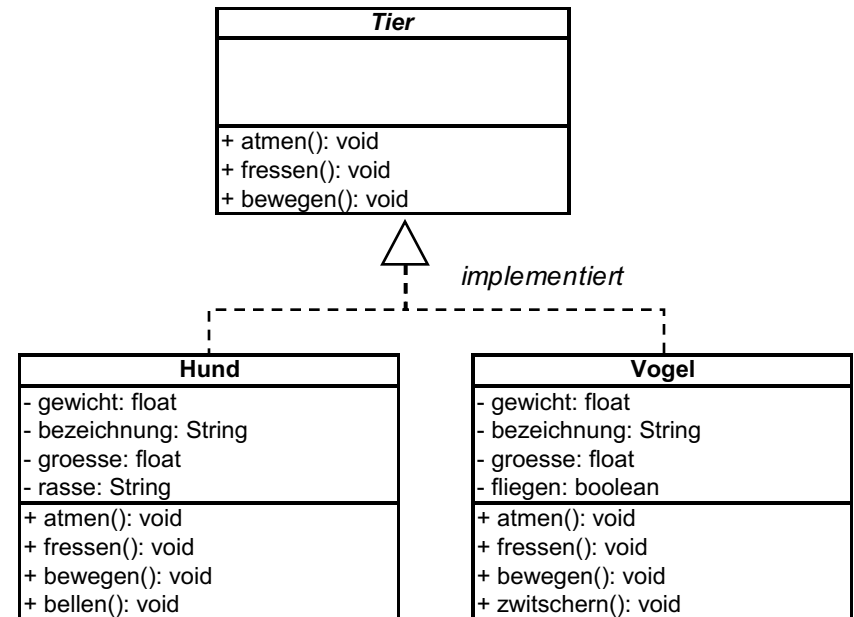
- Beispiel

```
package prog1.demos.objekt;
class AutoTest {
    public static void main(String[] args) {

        Tier[] x = new Tier[2];

        x[0] = new Hund(); //Narrowing Cast
        x[1] = new Vogel(); //Narrowing Cast

        x[0].atmen();
        x[1].atmen();
    }
}
```



Kopieren von Objekten

- es stehen zwei unterschiedliche Möglichkeiten zum Kopieren von Objekten zur Verfügung
- Möglichkeit 1
 - ◆ mit dem =-Operator
 - ◆ es wird lediglich die Referenz auf das Objekt kopiert
 - ◆ nach der Zuweisung verweisen beide Referenzvariablen auf dasselbe Objekt
- Möglichkeit 2
 - ◆ mit der clone()-Methode
 - ◆ es wird eine identische Kopie des Objektes erzeugt und unter einer anderen Referenz im Speicher hinterlegt
 - ◆ die Attributwerte der beiden Objekte sind gleich

Literaturverzeichnis

- BALZERT, HEIDE: Lehrbuch der Objektmodellierung - Analyse und Entwurf. Spektrum Akademischer Verlag, 2. Auflage, 2005. ISBN 3-8274-1162-9
- HÄUSLEIN, ANDREAS: Systemanalyse - Grundlagen, Techniken, Notierungen. VDE Verlag, 2004. ISBN 3-8007-2715-3
- HITZ, M., KAPPEL, G., KAPSAMMER, E. und RETSCHITZEGGER, W.: UML@Work - Objektorientierte Modellierung mit UML 2. dpunkt.verlag, 3., aktualisierte und überarbeitete Auflage, 2005. ISBN 3-89864-261-5
- HOLEY, T., WELTER, G. und WIEDEMANN, A.: Wirtschaftsinformatik. Kiehl Verlag, 2004. ISBN 3-470-52791-1
- RUPP, CHRIS / SOPHIST GROUP: Systemanalyse kompakt. Elsevier Spektrum Akademischer Verlag, 2004. ISBN 3-8274-1509-8
- STAHLKNECHT, P. und HASENKAMP, U.: Arbeitsbuch Wirtschaftsinformatik. Springer Verlag, 4. Auflage, 2006. ISBN 3-540-26361-6
- STAHLKNECHT, P. und HASENKAMP, U.: Einführung in die Wirtschaftsinformatik. Springer Verlag, 10. Auflage, 2002. ISBN 3-540-41986-1
- ZUSER, W., BIFFL, S., GRECHENING, T. und KÖHLE, M.: Software Engineering mit UML und dem Unified Process. Pearson Studium, 2001. ISBN 3-8273-7027-2