

Einführung in die Informatik

für ESE & MSTler

Einleitung

Organisatorisches, Motivation, Herangehensweise

Wolfram Burgard

Vorlesung

Zeit und Ort:

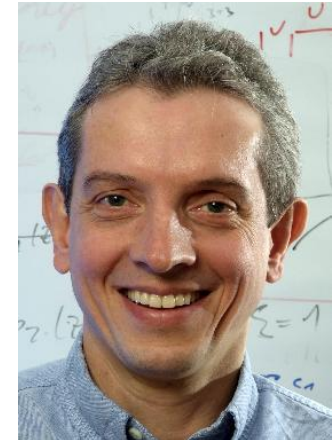
Mittwochs 14.00–16.00 Uhr
Gebäude 101 HS 00-026

Informationen zur Vorlesung, Aufzeichnungen, Übungszettel:

<http://ais.informatik.uni-freiburg.de/teaching/ss16/info/>

Dozent

- Prof. Dr. Wolfram Burgard
Gebäude 079, Raum 1010
Sprechstunden: n.V.
Email: burgard@informatik.uni-freiburg.de
Tel: 0761 203-8006/8026
<http://www.informatik.uni-freiburg.de/~burgard/>



Übungen

Organisation der Übungen:

- Thomas Darr
E-Mail: darr@informatik.uni-freiburg.de
<http://www.informatik.uni-freiburg.de/~darr>
- Andreas Kuhner
E-Mail: kuhnera@informatik.uni-freiburg.de
<http://www.informatik.uni-freiburg.de/~kuhnera>
- Alexander Schiotka
Email: schiotka@informatik.uni-freiburg.de
<http://www.informatik.uni-freiburg.de/~schiotka>



Übungsgruppen

- Eine zweistündige Übung pro Woche
- Übungsbeginn: 2. Semesterwoche
- Räume sind auf der Vorlesungshomepage angegeben
- Anmeldung über das Internet (Vorlesungsportal)

Gruppe	Tutor	Zeit	Raum
2	Chandran Goodchild	Donnerstag, 10:00 – 12:00	SR 00-031 Geb. 051
3	Nico Bühler	Donnerstag, 14:00 – 16:00	SR 00-031 Geb. 051
4	David Ruf	Donnerstag, 16:00 – 18:00	SR 00-006 Geb. 051

Von Studenten zu erbringende Leistungen

- Wir verlangen explizit keine Studienleistung während des Semesters
- Die aktive Teilnahme an den Übungen ist nicht verpflichtend, aber empfohlen
- **Benotete Klausur am 7.9.2016, 14-16 Uhr**
- Nachklausur oder zweite mündliche Prüfung

Übungszettel

- Ausgabe dienstags
- **Abgabe montags bis 23:59 Uhr** in der folgenden Woche
- Es gibt keine Bonuspunkte-Regelung (mehr)

Ziele dieser Vorlesung

Sie sollen in dieser Vorlesung Grundkenntnisse erlernen über

- Programmierung
- Modellierung
- Entwicklung
- Analyse
- Java
- ...

Was ist Informatik?

Informatik Duden:

Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Digitalrechnern (Computern).

Association of Computing Machinery (ACM):

Computer science is the systematic study of algorithms and data structures, specifically

1. their formal properties,
2. their mechanical and linguistic realizations, and
3. their applications.

Aspekte der Informatik?

- Technische Realisierung
- Effiziente Verfahren
- Theorie
- Programmiersprachen
- Techniken zur Programmentwicklung
- ...

Computer ...

- Was ist ein **Computer**?
- Kann man diesen Begriff **präzise definieren**?
- In welcher Form tauchen Computer im **täglichen Leben** auf?



?



?



?



?



?



?

Computer ...

Was ist ein Computer?

Informatik Duden: „(engl.: to compute = rechnen, berechnen; ursprünglich aus dem lat. computare = berechnen ...): *Universell einsetzbares Gerät zur automatischen Verarbeitung von Daten.*“

Im täglichen Leben: Maschinen, die für uns Werte berechnen (z.B. Steuern), die uns helfen, Briefe zu schreiben, die unsere Autos kontrollieren, mit deren Hilfe Daten analysiert werden ...

. . . und Programme

Was eigentlich ist ein Programm?

... und Programme

Was ist ein Programm?

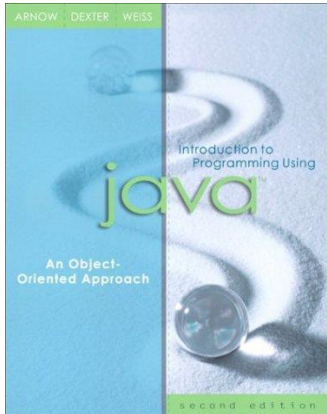
Zunächst: *Verarbeitungsvorschrift, die so präzise ist, dass sie von einem Computer ausgeführt werden kann.*

- Programme werden in speziellen Sprachen, so genannten *Programmiersprachen* formuliert.
- Der Inhalt eines Programms ist der *Code*.
- Computer führen Programme aus.

In diesem Kurs: Methodik der Programmierung am Beispiel von



Buch zur Vorlesung



Introduction to Programming Using Java: An Object-Oriented Approach, 2. Auflage, David Arnow, Scott Dexter, Gerald Weiss, ISBN 0-321-20006-3

Weitere Literatur auf der Vorlesungsseite oder unter

<http://ais.informatik.uni-freiburg.de/teaching/ss16/info/literature/>

und auf den Java Seiten von Oracle

<http://www.oracle.com/technetwork/java/javase/overview/index.html>

Java Tutorials Online

Java Tutorien auf den Java Seiten von Oracle

<http://docs.oracle.com/javase/tutorial/>



The Java™ Tutorials

[Download Ebooks](#)
[Download JDK](#)
[Search Java Tutorials](#)

The Java Tutorials are practical guides for programmers who want to use the Java programming language to create applications. They include hundreds of complete, working examples, and dozens of lessons. Groups of related lessons are organized into "trails".

The Java Tutorials primarily describe features in Java SE 8. For best results, [download JDK 8](#).

What's New

The Java Tutorials are continuously updated to keep up with changes to the Java Platform and to incorporate feedback from our readers.

This release of the tutorial corresponds to the JDK 8u40 release.

This release includes a new lesson in the Deployment trail that describes how to use the Java packaging tools to generate self-contained applications. Self-contained applications are Java applications that are bundled with the JRE that is needed to run. These applications are installed on a user's local drive and launched in the same way as native applications. See [Deploying Self-Contained Applications](#) for more information.

Trails Covering the Basics

These trails are available in book form as *The Java Tutorial, Fifth Edition*. To buy this book, refer to the box to the right.

- » [Getting Started](#) — An introduction to Java technology and lessons on installing Java development software and using it to create a simple program.
- » [Learning the Java Language](#) — Lessons describing the essential concepts and features of the Java Programming Language.
- » [Essential Java Classes](#) — Lessons on exceptions, basic input/output, concurrency, regular expressions, and the platform environment.
- » [Collections](#) — Lessons on using and extending the Java Collections Framework.
- » [Date-Time APIs](#) — How to use the `java.time` packages to write date and time code.
- » [Deployment](#) — How to package applications and applets using JAR files, and deploy them using Java Web Start and Java Plug-in.
- » [Preparation for Java Programming Language Certification](#) — List of available training and tutorial resources.

Creating Graphical User Interfaces

- » [Creating a GUI with Swing](#) — A comprehensive introduction to GUI creation on the Java platform.
- » [Creating a JavaFX GUI](#) — A collection of JavaFX tutorials.

Specialized Trails and Lessons

These trails and lessons are only available as web pages.

- » [Custom Networking](#) — An introduction to the Java platform's powerful networking features.
- » [The Extension Mechanism](#) — How to make custom APIs available to all applications running on the Java platform.



Not sure where to start?
See [Learning Paths](#)

Tutorial Contents

[Really Big Index](#)

Tutorial Resources

- » [View the Java Tutorials Online](#) (Last Updated 3/3/2015).
- » [The Java Tutorials' Blog](#) has news and updates about the Java SE tutorials.
- » [Download the latest Java Tutorials bundle](#).

In Book Form

- » [Download ebook files](#).
- » [The Java Tutorial, Sixth Edition](#). Amazon.com.

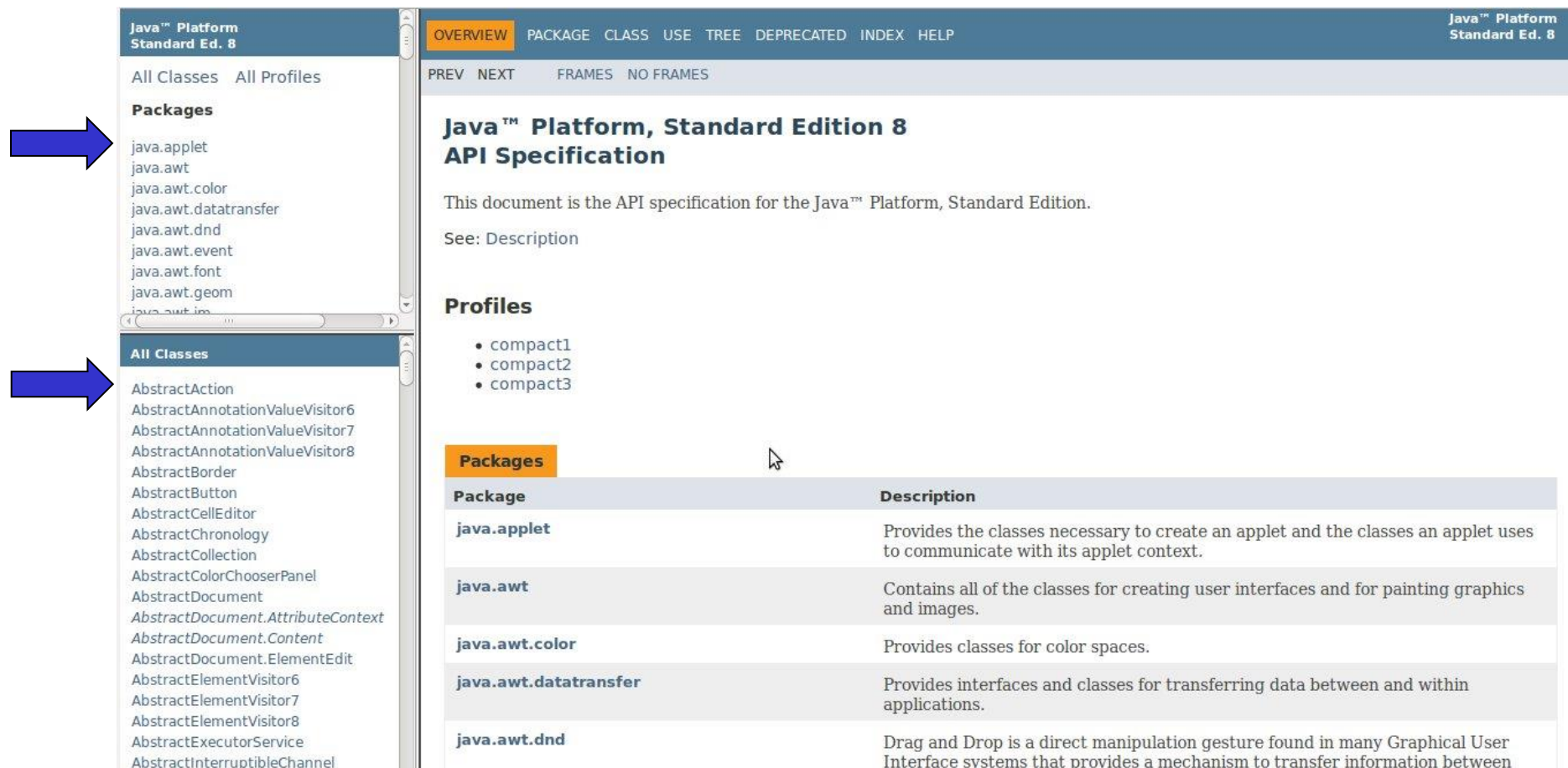
Other Resources

- » [Java SE Developer Guides](#)

Java API Dokumentation Online

Weitere Information finden Sie auf den Java Seiten von Oracle

<http://docs.oracle.com/javase/8/docs/api/>



Java™ Platform Standard Ed. 8

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV NEXT FRAMES NO FRAMES

Java™ Platform, Standard Edition 8 API Specification

This document is the API specification for the Java™ Platform, Standard Edition.

See: Description

Profiles

- compact1
- compact2
- compact3

Packages

Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between

Einführung in die Informatik

Jumping into Java

Programme, Modelle, Objekte, Klassen, Methoden

Wolfram Burgard

Java, eine moderne, objektorientierte Sprache

Beispielprogramm:

```
class Program1 {  
    public static void main(String[] arg) {  
        System.out.println("This is my first Java program");  
        System.out.println("but it won't be my last.");  
    }  
}
```

Ausgabe des Programms Program1:

```
This is my first Java program  
but it won't be my last.
```

Modelle

Modelle sind vereinfachte Darstellungen. Sie enthalten die relevanten Eigenschaften eines modellierten Objektes.

Beispiel Kundendienst:

- Stadtplan.
- Nadeln markieren die Standorte der Kundendienstmitarbeiter.
- Fähnchen markieren die Stellen, an denen Kunden warten.

Abstraktes, unvollständiges Modell, welches aber für die Zuordnung von Mitarbeitern zu Kunden gut geeignet sein kann.

Eigenschaften von Modellen

- **Elemente eines Modells repräsentieren andere, komplexe Dinge.**
Nadeln repräsentieren die Kundendienstmitarbeiter.
- **Die Elemente eines Modells haben ein bestimmtes, konsistentes Verhalten.**
Nadeln repräsentieren Positionen und können an andere Stellen bewegt werden.
- **Die Elemente eines Modells können entsprechend ihrem Verhalten zu verschiedenen Gruppen zusammengefasst werden.**
Mitarbeiter fahren von Kunde zu Kunde, Kunden kommen hinzu und verschwinden wieder.
- **Aktionen von außen stoßen das Verhalten eines Modellelements an.**
Nadeln werden in der Zentrale per Hand bewegt.

Objekte

In Java heißen die Elemente eines Modells **Objekte**.

Kundendienstmodell

Die 43 Kundendienstmitarbeiter werden durch 43 Nadeln repräsentiert.

Kunden werden durch Fähnchen markiert.

Wenn ein Kunde anruft, wird in der Zentrale ein Fähnchen in die Karte gesteckt.

Java-Modell

In Java würden die 43 Kundendienstmitarbeiter durch 43 Objekte modelliert.

In Java würden spezielle Kunden-Objekte verwendet.

In Java würde ein Kunden-Objekt erzeugt.

Verhalten

Alle Kundendienstmitarbeiter verhalten sich ähnlich: Sie fahren von Kunde zu Kunde und führen dort jeweils ihre Aufträge aus.

In Java haben alle Objekte für Kundendienstmitarbeiter das gleiche Verhalten.

Auch Kundenobjekte haben ein gleiches Verhalten: Nach Auftragseingang werden sie erzeugt. Sie werden gelöscht, sobald der Auftrag ausgeführt ist.

In Java wird das **Verhalten eines Objektes durch ein Programmstück definiert**, welches als **Klasse** bezeichnet wird.

Klassen und Instanzen

- **Kategorien von Elementen eines Modells heißen in Java Klassen.**
- Die **Definition einer Klasse** ist ein **Programmstück** (Code), welches spezifiziert, wie sich die **Objekte dieser Klasse verhalten**.
- Nachdem eine **Klasse definiert** wurde, **können Objekte** dieser Klasse **erzeugt werden**.
- **Jedes Objekt gehört zu genau einer Klasse und wird Instanz dieser Klasse genannt.**

Vordefinierte Klassen

- Glücklicherweise müssen Programmierer das Rad nicht ständig neu erfinden.
- Java beinhaltet eine große Anzahl **vordefinierter Klassen** und Objekte.
- Diese Klassen können verwendet werden, um auf einfache Weise **komplizierte Anwendungen** und z.B. **grafische Benutzeroberflächen** zu realisieren.

Ein einfaches Beispiel: Der Monitor



- Java Programme können Text auf den Monitor ausgeben
- In Java wird der Monitor durch ein Objekt modelliert
- Man kann an dieses Objekt **Nachrichten** senden
- Eine solche Nachricht enthält den auszugebenden Text

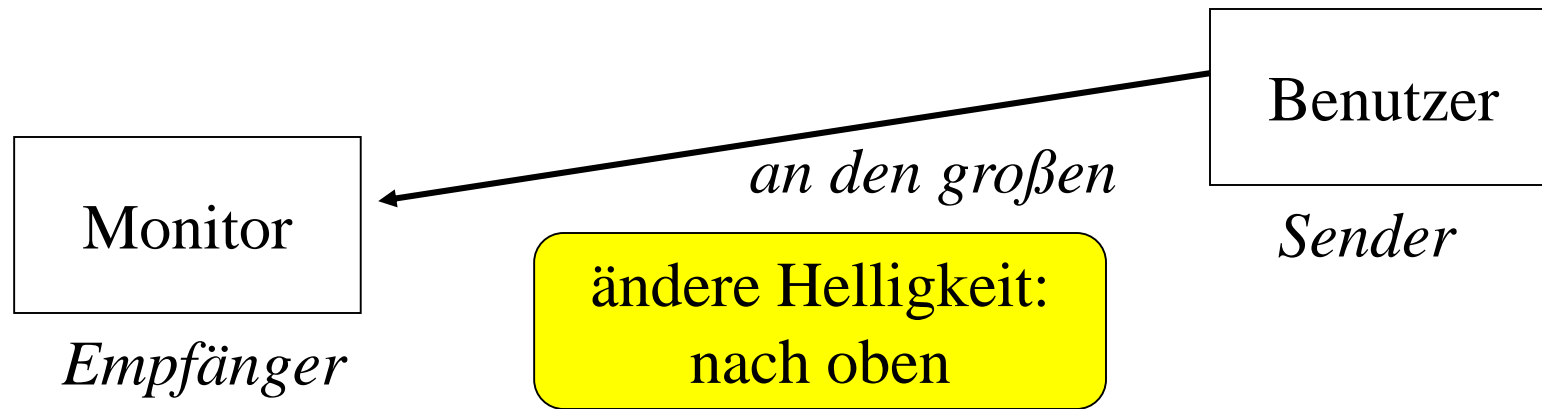
Ein einfaches Beispiel für eine Nachricht

Zwei Personen sitzen vor zwei unterschiedlich großen Bildschirmen. Eine Person sagt: „Würdest Du bitte die Helligkeit des großen erhöhen?“

Informationen in dieser Frage:

- des großen
- die Helligkeit soll verstellt werden
- nach oben

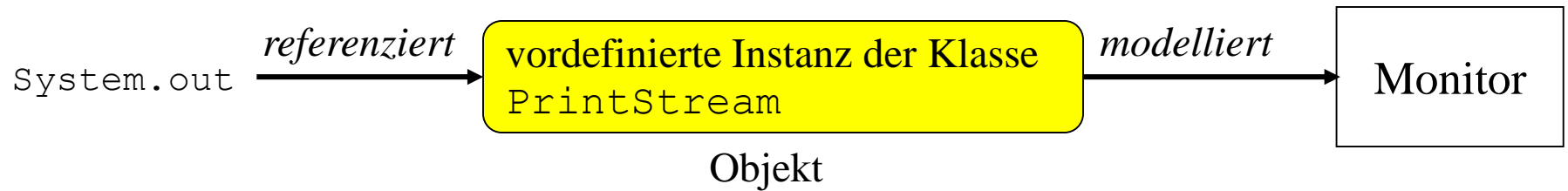
Die Aktion in der Java-Terminologie



- „Des großen“ ist eine **Referenz** , an die eine Nachricht (**Message**) geschickt wird.
- Die Nachricht spezifiziert ein **Verhalten** (Helligkeit verstellen)
- und enthält **weitere Details** (nach oben).

Repräsentation des Bildschirms in Java

- In Java wird der **Bildschirm** durch ein **vordefiniertes Objekt** repräsentiert.
- Dieses Objekt ist Instanz der Klasse **PrintStream**.
- Objekte der Klasse **PrintStream** erlauben das Ausgeben von **Zeichenketten**.



- Eine **Referenz in Java** ist eine Phrase, die einen **Bezug zu einem Objekt** herstellt. Wir sagen, dass sie das **Objekt referenziert**.
- **System.out** referenziert ein Objekt der Klasse **PrintStream**, welches den Monitor modelliert. **System.out** ist eine **Referenz** auf das Objekt.

Nachrichten/Messages in Java

- Um einen Text auf dem Bildschirm auszugeben, schickt man in Java eine **Nachricht** an das durch Objekt `System.out` referenzierte Objekt.
- Dieses Message spezifiziert ein **Verhalten**, welches die Klasse `PrintStream` zur Verfügung stellt.
- Hier ist diese Nachricht das Ausgeben einer Zeile, welche `println` genannt wird.
- Die **weiteren Details** sind dabei die Zeichen, die ausgegeben werden sollen, z.B.: „Welcome to Java!“.

Zusammenfassend:

- Referenz auf den Monitor: `System.out`
- Nachricht: `println`
- Notwendige Details: `("Welcome to Java!")`

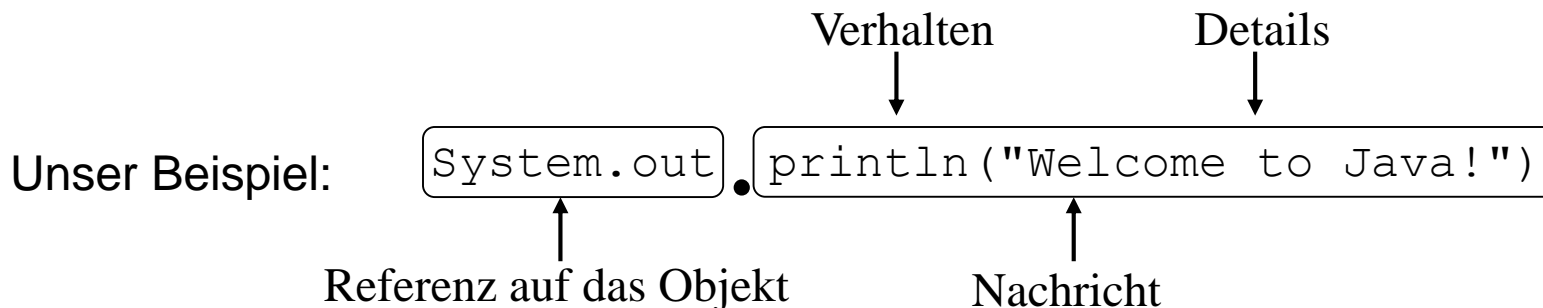
Verschicken einer Nachricht an das `System.out`-Objekt

In Java bestehen Nachrichten aus folgenden Komponenten:

- Der Name des Verhaltens (z.B. `println`)
- Die weiteren Informationen (z.B. `"Welcome to Java!"`), die in Klammern eingeschlossen wird.

In Java verwenden wir Phrasen, die aus folgenden Komponenten bestehen, um eine Nachricht an den entsprechenden **Empfänger** zu senden:

- einer Referenz auf das Empfängerobjekt (z.B. `System.out`) gefolgt von einem Punkt und
- der Nachricht, die wir verschicken wollen.



Java Statements

- Das Versenden einer Nachricht an ein Objekt ist eine vom Programmierer spezifizierte Aktion.
- Diese Aktion wird vom Computer ausgeführt, wenn das Programm läuft.
- In Java werden alle Aktionen durch **Statements** spezifiziert.

Um ein Statement zu erzeugen, welches unsere Ausgabe-Aktion durchführt, schreiben wir die entsprechende Nachricht hin und fügen ein Semikolon hinzu:

```
System.out.println("Welcome to Java!");
```


Ein Java-Programm

Ziel: Ausgabe der beiden Zeilen

```
This is my first Java program  
but it won't be my last.
```

Wir wissen:

- Es gibt ein vordefiniertes Objekt (referenziert durch `System.out`), dessen Verhalten auch das Ausgeben von Zeichen auf dem Bildschirm einschließt.
- Dieses Verhalten wird aktiviert, indem wir diesem Objekt eine Nachricht mit dem Namen `println` schicken.
- Die weiteren Details sind die **Argumente**, die das Objekt benötigt, um das Verhalten auszuführen (hier die Zeile, die ausgegeben werden soll).

Um zwei Zeilen auszugeben, schicken wir einfach zwei Nachrichten:

```
System.out.println("This is my first Java program");  
System.out.println("but it won't be my last.");
```

Das komplette Programm

Um ein **gültiges Programm** zu erhalten, müssen wir seinen **Namen** festlegen und zusätzlichen Text hinzufügen. Wir nennen Namen **Identifizier** oder **Bezeichner**.

Ein **Identifizier** ist eine **Folge von Buchstaben, Ziffern und Unterstrichen**. Das **erste Zeichen** muss ein **Buchstabe** sein.

Beispiele: `Program1`, `x7`, `xyp`, `xrz`

Dies ergibt das bereits bekannte Programm:

```
class Program1 {  
    public static void main(String[] arg) {  
        System.out.println("This is my first Java program");  
        System.out.println("but it won't be my last.");  
    }  
}
```

Einige Regeln (1)

Identifizier: Klassen und Nachrichten müssen einen Bezeichner haben. Dabei wird zwischen Groß- und Kleinschreibung unterschieden.

Keywords: **Keywords** oder **Schlüsselworte** sind spezielle Worte mit einer vordefinierten Bezeichnung, wie z.B. `static`, `class`, ...

Reihenfolge von Statements: Statements werden in der Reihenfolge, ausgeführt, in der sie im Programm stehen.

```
System.out.println("This is my first Java program");  
System.out.println("but it won't be my last.");
```

erzeugt eine andere Ausgabe als

```
System.out.println("but it won't be my last.");  
System.out.println("This is my first Java program");
```

Einige Regeln (2)

Formatierung: Halten Sie sich bei der Erstellung von Java-Programmen an folgende Regeln:

1. Jede Zeile enthält höchstens ein Statement.
2. Verwenden Sie die Space oder Tabulator-Taste, um Statements einzurücken.
3. Halten Sie sich an den Stil dieser Vorlesung (siehe Zettel).

Vorteile:

1. Programme sind leichter lesbar,
2. leichter zu verstehen und damit auch
3. leichter zu warten.

Einige Regeln (3)


Prinzipiell ist Java sehr flexibel.

Eine der Hauptregeln: **Bezeichner müssen durch ein Leerzeichen getrennt sein.**

`classProgram1` entspricht nicht `class Program1`

Zulässig wäre aber z.B.:

```
class Program1 { public static void  
main(String[] arg) { System.out.println("This is my first  
Java program"); System.out.println("but it won't be my  
last."); } }
```



Siehe: `examples/Program1jammed.java`

Einige Regeln (4)

Kommentare: Java erlaubt dem Programmierer, Kommentare in den Code einzufügen. Es gibt zwei Arten:

1. Eingeschlossene Kommentare, d.h. Text, der zwischen `/*` und `*/` eingeschlossen ist:

```
/*  
 * This program prints out several greetings  
 */
```

2. Zeilenkommentare: Alle Zeichen hinter einem `//` werden als Kommentare angesehen:

```
// This program prints out several greetings
```

Hinweise zur Verwendung von Kommentaren

1. Vor jeder Zeile, die mit dem Wort `class` beginnt, sollte ein Kommentar stehen, der die Klasse erklärt.
2. Kommentare sollten dazu dienen, Dinge zu erklären, die nicht unmittelbar aus dem Code abgelesen werden können.
3. Kommentare sollten nicht in der Mitte von einem Statement auftauchen.

Ein Programm mit Kommentaren

```
/*  
 * This program prints my first Java experience and my  
 * intent to continue.  
 */  
class Program1 {  
    public static void main(String[] arg) {  
        System.out.println("This is my first Java program");  
        System.out.println("but it won't be my last.");  
    }  
}
```


Verwendung von `PrintStream`-Objekten

Wenn wir die Nachricht

```
println("something to display")
```

an das durch `System.out` referenzierte `PrintStream`-Objekt senden, führt das zur Ausgabe des entsprechenden Textes auf dem Bildschirm.

Jedes weitere Zeichen danach wird am Beginn der nächsten Zeile ausgegeben.

Die Nachrichten `print` und `println`

Alternative:

```
print("something to display")
```

Bei Verwendung von `print` erscheint das nächste gedruckte Zeichen in der gleichen Zeile:

```
System.out.print("JA");
```

```
System.out.print("VA");
```

ergibt die Ausgabe

```
JAVA
```

Eine Variante der `println`-Nachricht ist `println()`. Die Nachricht

```
System.out.println();
```

bewirkt, dass nachfolgende Ausgaben in der nächsten Zeile beginnen.

Ein weiteres Programm

```
/*
 * This program illustrates the use of print vs. println.
 */
class Program1print {
    public static void main(String[] arg) {
        System.out.print("This is my first Java program");
        System.out.print(" ");
        System.out.println("but it won't be my last.");
    }
}
```

Vom Programm zur Ausführung

Um ein Programm auf einem Rechner ausführen zu können, müssen wir

1. das **Programm für den Computer zugänglich machen**,
2. das **Programm** in eine Form **übersetzen**, die der Rechner versteht, und
3. den Computer anweisen, das **Programm auszuführen**.

Schritt 1: Eingeben des Programms

Um ein Programm für den Rechner zugänglich zu machen, müssen wir eine **Programmdatei** erstellen, d.h. eine **Textdatei, die den Programmtext enthält**.

In Java muss der **Name der Datei** den Namen `X.java` haben, wobei `X` der Programmname oder Klassenname ist.

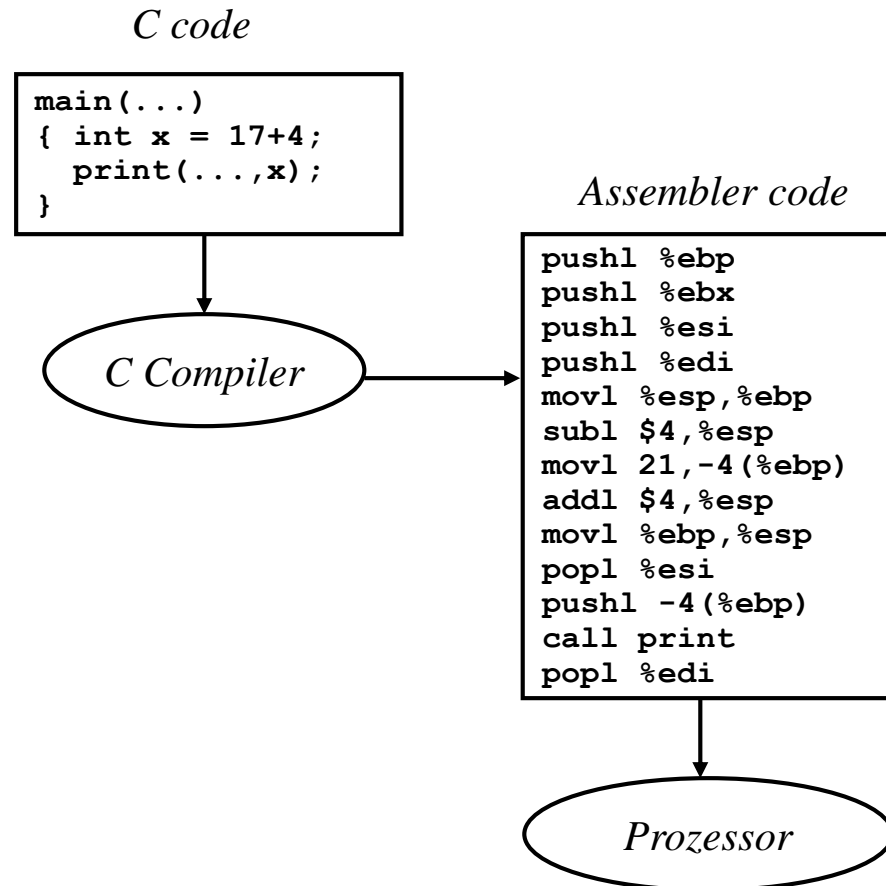
Es ist Konvention, dass Klassennamen groß geschrieben werden.

Schritt 2: Übersetzen des Programms

- Computer können Java-Programme nicht direkt ausführen.
- Computer sind nur in der Lage, Anweisungen einer primitiven und so genannten *Maschinsprache* auszuführen.
- Da wir eine abstrakte *Hochsprache* verwenden, müssen wir eine Möglichkeit finden, unsere Programme in Maschinen-Code zu überführen.
- Dafür gibt es verschiedene Modelle: Am weitesten verbreitet sind *Compiler* und *Interpreter*.

Das Compiler-Modell

Das Programm wird direkt in **Maschinencode** übersetzt und kann vom Rechner **unmittelbar ausgeführt** werden.



Das Interpreter-Modell

Prinzip:

- Das Programm wird von dem **Interpreter** direkt ausgeführt.
- Der Interpreter verwendet für jedes Statement bei der Ausführung die entsprechende Sequenz von Maschinenbefehlen.

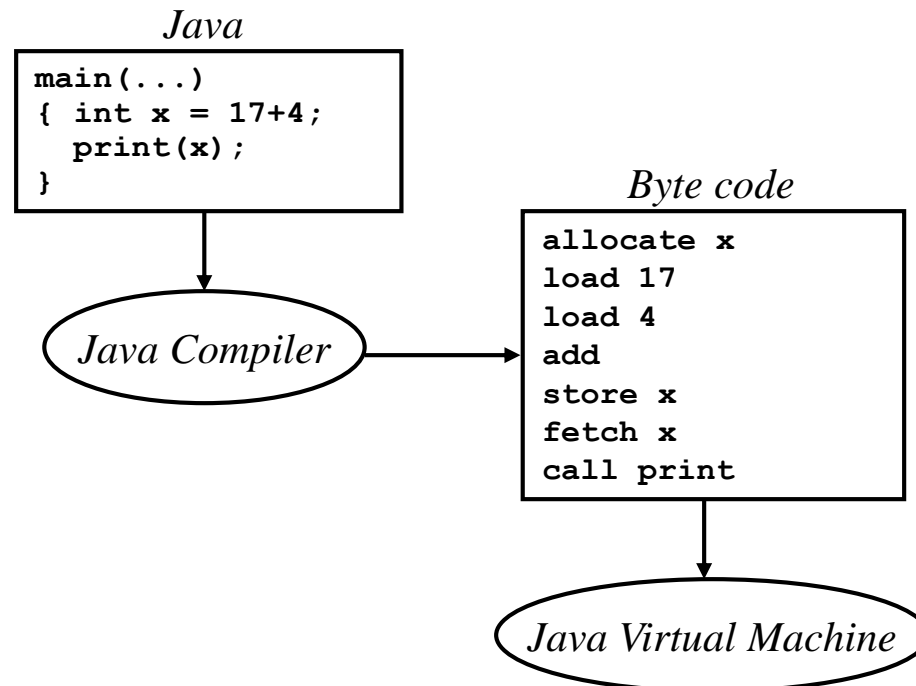
Vergleich zum Compiler:

1. Das Programm muss nicht erst übersetzt werden.
2. Die Interpretation ist langsamer als die Ausführung kompilierter Programme.

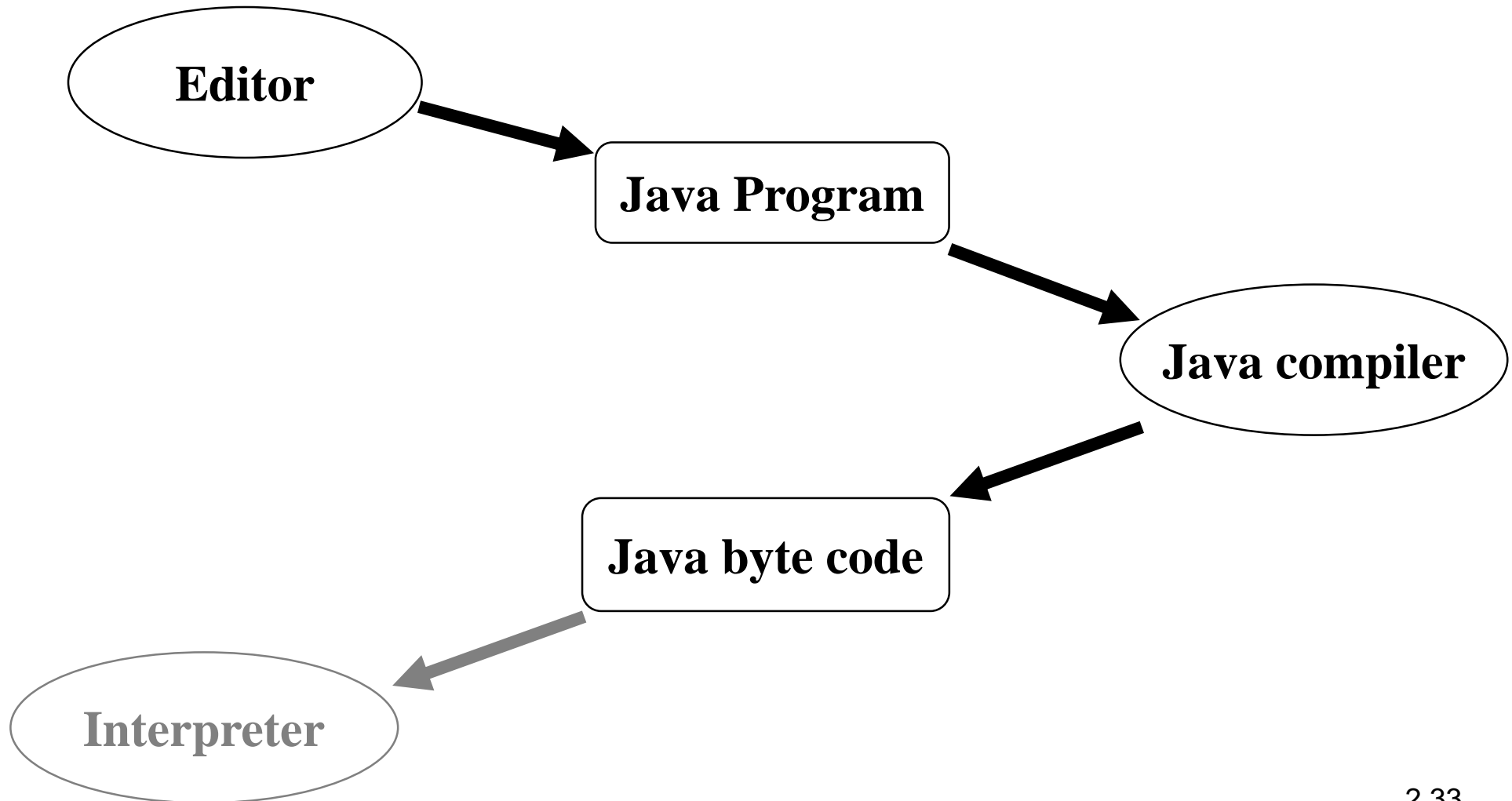
Das Java-Modell

Java verwendet einen Interpreter und einen Compiler:

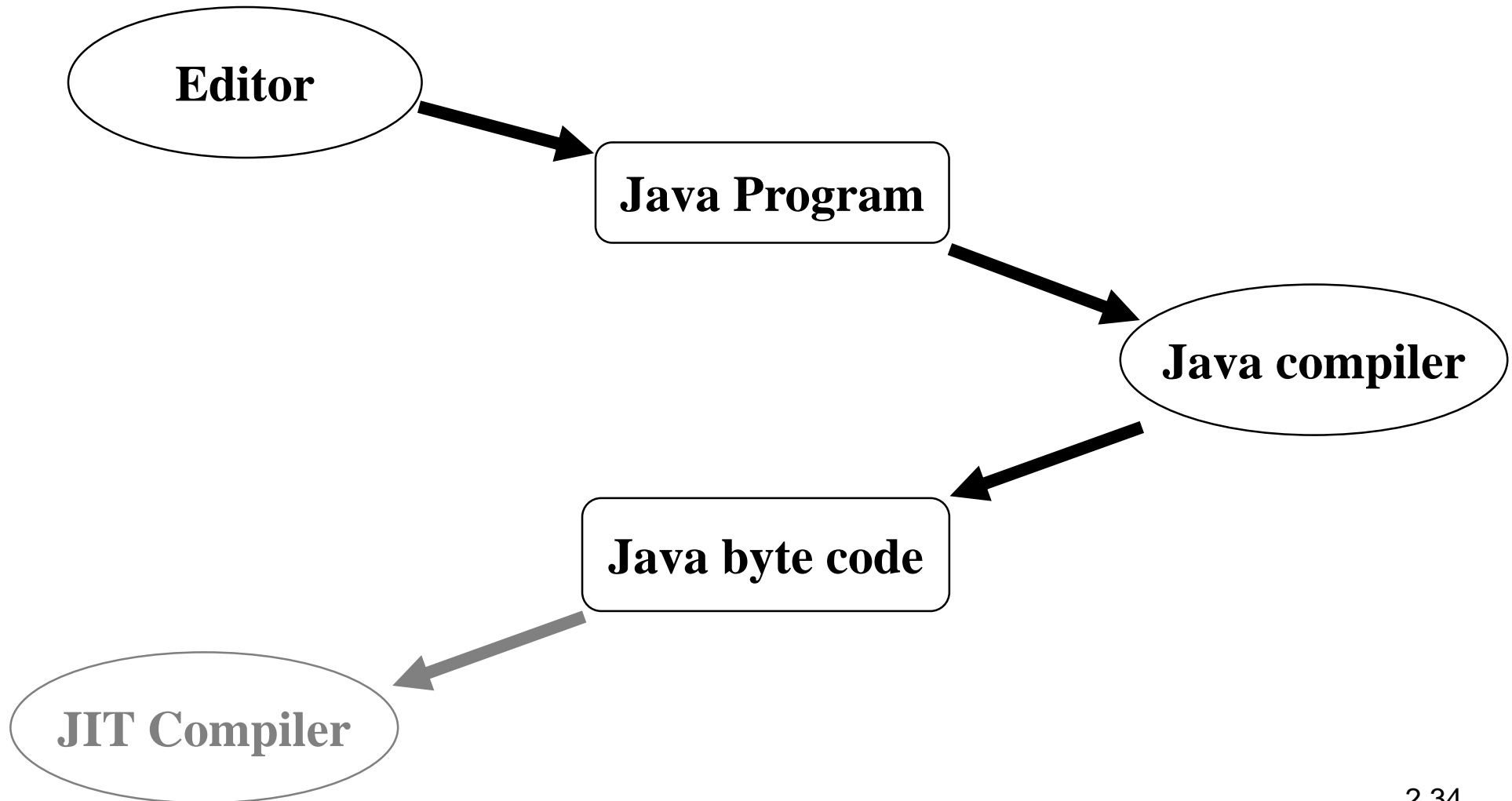
- Java **übersetzt** zunächst in eine **Byte Code** genannte Zwischensprache.
- Der Byte Code wird dann von der **Java Virtual Machine interpretiert**.



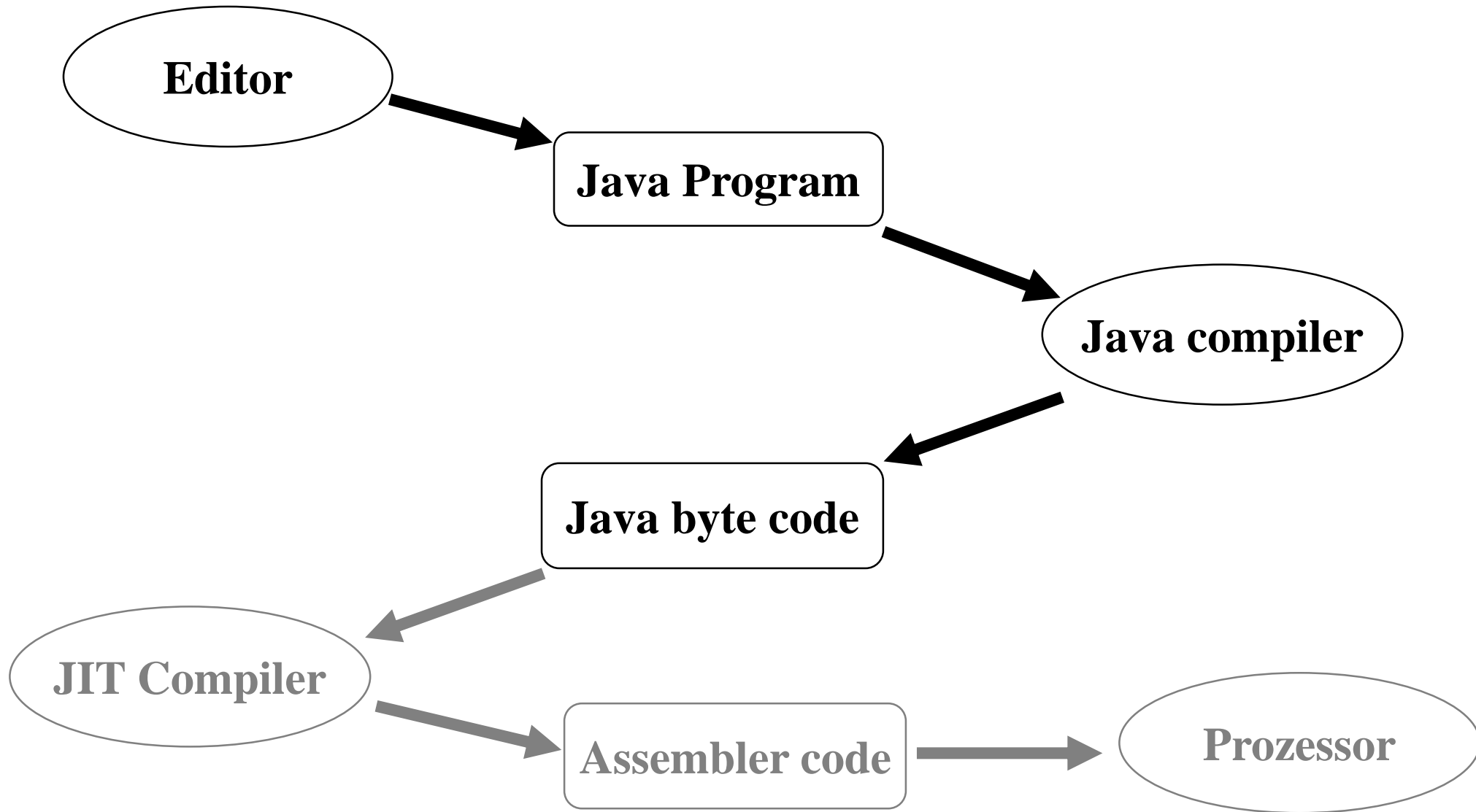
Erzeugung eines Ausführbaren Java-Programms



Erzeugung eines Ausführbaren Java-Programms



Erzeugung eines Ausführbaren Java-Programms



Programmierfehler

Wir unterscheiden grob zwei Arten von Fehlern:

1. **Compile-Zeit-Fehler** (Compile-Time Errors) werden bei der Übersetzung entdeckt und führen zum Abbruch des Übersetzungsprozesses.
2. **Laufzeitfehler** (Run-Time Errors) treten erst bei der Ausführung auf und liefern falsche Ergebnisse oder führen zum Absturz des Programms.

Laufzeitfehler sind **häufig schwieriger zu entdecken!**

Beispiel für einen Compile-Time Error

```
// This will lead to errors
class Program1 {
    public static void main(String[] arg) {
        Sistem.out.println("This is my first Java program");
        Sistem.out.println("but it won't be my last.");
    }
}
```

Siehe: examples/Program1sistem.java

Ausgabe des Compilers

```
javac Programlsistem.java
```

```
Programlsistem.java:4: error: package Sistem does not exist
    Sistem.out.println("This is my first Java program");
    ^
```

```
Programlsistem.java:5: error: package Sistem does not exist
    Sistem.out.println("but it won't be my last.");
    ^
```

```
2 errors
```

Beispiel für einen Run-Time Error

```
class Program1last {  
    public static void main(String[] arg) {  
        System.out.println("This is my first Java program");  
        System.out.println("but it will be my last.");  
    }  
}
```

Das Programm kann compiliert und ausgeführt werden und liefert die Ausgabe:

```
This is my first Java program  
but it will be my last.
```


Zusammenfassung (1)

- **Programme** sind Texte, die ein Computer ausführt, um eine bestimmte Aufgabe durchzuführen.
- **Java-Programme** können aufgefasst werden als Modelle, die Objekte und ihr Verhalten beschreiben.
- **Objekte** sind in Java die **grundlegenden Modellierungskomponenten**.
- **Objekte** mit gleichem Verhalten werden in **Kategorien**, bzw. **Klassen** zusammengefasst.
- Das **Verhalten** eines Objektes wird durch das **Senden einer Nachricht** an dieses Objekt angestoßen.

Zusammenfassung (2)

- Die **Verhalten** der Objekte werden durch **Programmstücke** beschrieben, die aus einzelnen **Statements** bestehen.
- Eine typische Form eines **Statements** wiederum ist das **Versenden einer Nachricht** an ein Objekt.
- Um einem Objekt eine Nachricht zu senden, verwendet man eine **Referenz** auf dieses Objekt und den entsprechenden **Methodennamen**.
- Java-Programme werden in **Java Byte Code** übersetzt.
- Dieser Byte Code wird dann von dem **Java-Interpreter** auf dem Rechner ausgeführt.
- Zur Effizienzsteigerung werden in der Praxis just-in-time (JIT) Compiler eingesetzt, die Java Byte Code in Assembler Code übersetzen.

Einführung in die Informatik

Objekte

Referenzen, Methoden, Klassen, Variablen, Objekte

Wolfram Burgard

Referenzen

- Eine **Referenz** in Java ist jede Phrase, die sich auf ein Objekt bezieht.
- Referenzen werden verwendet, um dem entsprechenden Objekt eine Nachricht zu schicken.
- Streng genommen ist `System.out` kein Objekt sondern nur eine Referenz.

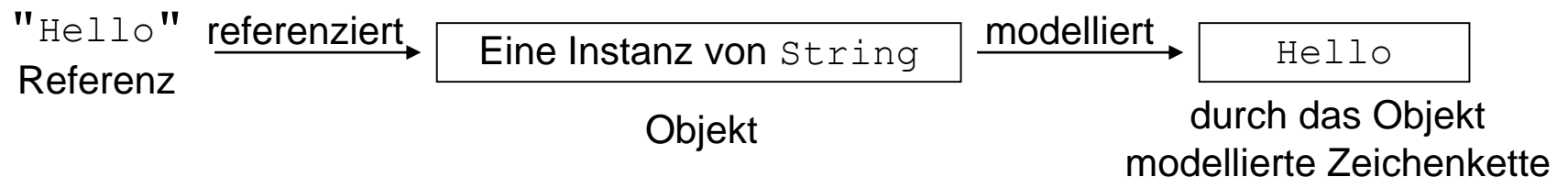
Bezeichnung: Das `System.out`-Objekt

Ausführen von Nachrichten

- Java-Statements werden in der Reihenfolge ausgeführt, in der sie im Programm stehen.
- Wenn eine **Nachricht** an ein Objekt (den Empfänger) geschickt wird, wird der Code des Senders unterbrochen, bis der Empfänger die Nachricht erhalten hat.
- Der Empfänger führt die durch die Nachricht spezifizierte Methode aus. Dies nennen wir „*Aufrufen einer Methode*“.
- Wenn der Empfänger die Ausführung seines Codes beendet hat, *kehrt die Ausführung zum Code des Senders zurück*.

Die String-Klasse

- `String` ist eine vordefinierte Klasse.
- Sie modelliert **Folgen von Zeichen (Characters)**.
- Zu den zulässigen Zeichen gehören Buchstaben, Ziffern, Interpunktionssymbole, Leerzeichen und andere, spezielle Symbole.
- In Java werden alle Folgen von Zeichen, die in Hochkommata eingeschlossen sind, als **Referenzen auf Zeichenketten** interpretiert.



Die String-Methode toUpperCase

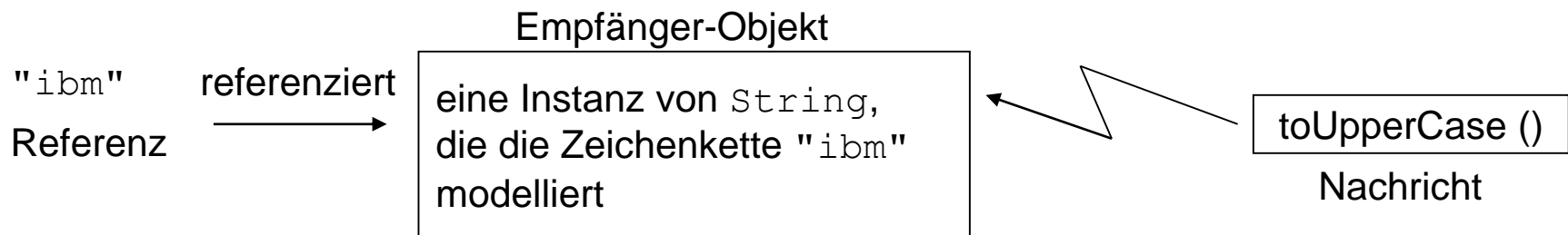
Eine Methode, die von der `String`-Klasse zur Verfügung gestellt wird, ist `toUpperCase`, welche keine Argumente hat.

Um eine **Nachricht** an die `String`-Klasse zu senden, verwenden wir die übliche Notation:

Referenz.Methodenname (Argumente)

Anwendungsbeispiel: `"ibm".toUpperCase ()`

Der Empfänger der `toUpperCase`-Nachricht ist das `String`-Objekt, welches durch `"ibm"` referenziert wird.



Effekte von String-Methoden

- Eine Methode der Klasse String ändert nie den Wert des Empfängers.
- Stattdessen liefert sie als Ergebnis eine Referenz auf ein neues Objekt, an welchem die entsprechenden Änderungen vorgenommen wurden.

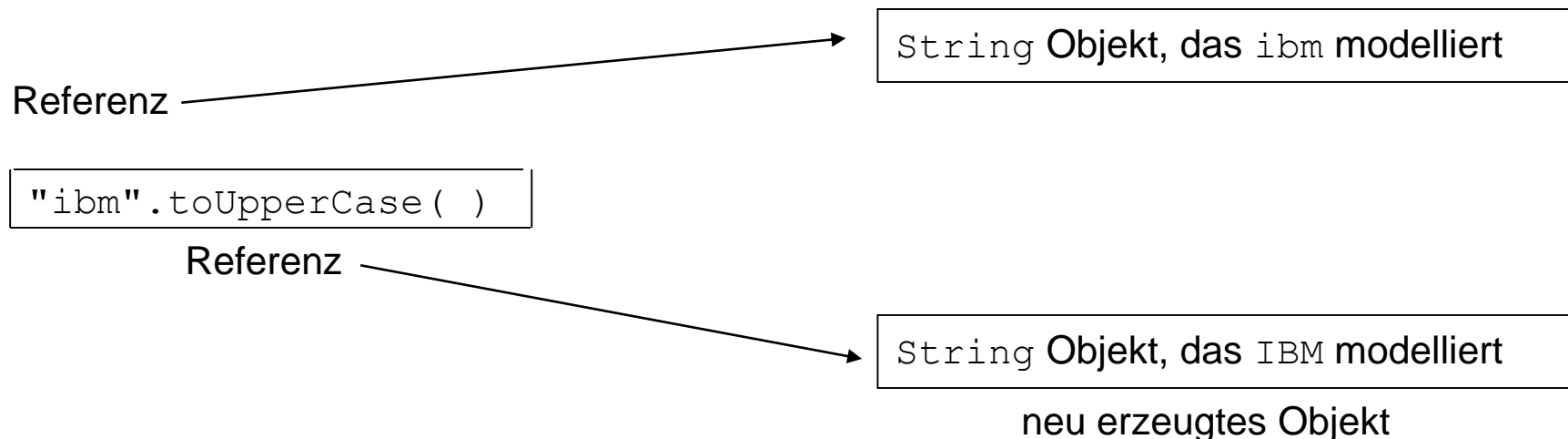
Beispiel:

- `"ibm".toUpperCase()` sendet nicht nur die `toUpperCase`-Nachricht an das `"ibm"`-Objekt.
- Der Ausdruck liefert auch eine Referenz auf ein neues `"IBM"`-Objekt.
- Wir sagen: „Der **Return-Wert** von `"ibm".toUpperCase()` ist eine **Referenz**.“

Beispiel

Da die `println`-Methode der Klasse `PrintStream` eine Referenz auf ein `String`-Objekt verlangt, können wir schreiben:

```
System.out.println("ibm".toUpperCase());
```



Methoden, Argumente und Return-Werte

Klasse	Methode	Return-Wert	Argumente
PrintStream	println	kein	kein
PrintStream	println	kein	Referenz auf ein <code>String</code> -Objekt
PrintStream	print	kein	Referenz auf ein <code>String</code> -Objekt
String	toUpperCase	Referenz auf ein <code>String</code> -Objekt	kein

Signatur einer Methode: Bezeichnung der Methode plus Beschreibung seiner Argumente

Prototyp einer Methode: Signatur + Beschreibung des Return-Wertes

Methoden ohne Return-Wert

- Viele Methoden liefern eine Referenz auf ein Objekt zurück.
- Das gilt insbesondere für Methoden, die ein Ergebnis liefern, wie z.B. die `String`-Methode `toUpperCase()`, die eine Referenz auf ein `String`-Objekt liefert.
- Aber welchen Typ sollen Methoden wie `println()` haben, die kein Ergebnis liefern?
- Wir sagen: „Methoden ohne Ergebnis haben den Typ `void`.“

Referenz-Variablen

- Eine **Variable** ist ein Bezeichner, dem ein Wert zugewiesen werden kann, wie z.B. sei $x=5$.
- Sie wird Variable genannt, weil sie zu verschiedenen Zeitpunkten verschiedene Werte annehmen kann.
- Eine **Referenz-Variable** ist eine Variable, deren Wert eine **Referenz** ist.
- Angenommen **line** ist eine **String-Referenz-Variable** auf ein `String`-Objekt, welches folgende Zeichenkette repräsentiert:
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
- Folgende Anweisungen geben zwei Zeilen von x-en aus:
`System.out.println(line);`
`System.out.println(line);`
- Möglich ist natürlich auch:
`System.out.println(line.toUpperCase());`

Deklaration von Referenz-Variablen und Wertzuweisungen

- Um in Java eine **Referenz-Variable** zu **deklarieren**, geben wir die Klasse und den Bezeichner an und schließen mit einem Semikolon ab:

```
String      greeting;  // Referenz auf einen Begrüßungs-String
PrintStream output;    // Referenz auf dasselbe PrintStream-
                        // Objekt wie System.out
```

- Um einer Variable einen Wert zu geben, verwenden wir eine so genannte **Wertzuweisung**:

Variable = Wert;

Beispiele:

```
output = System.out;
greeting = "Hello";
```

Wir können jetzt schreiben:

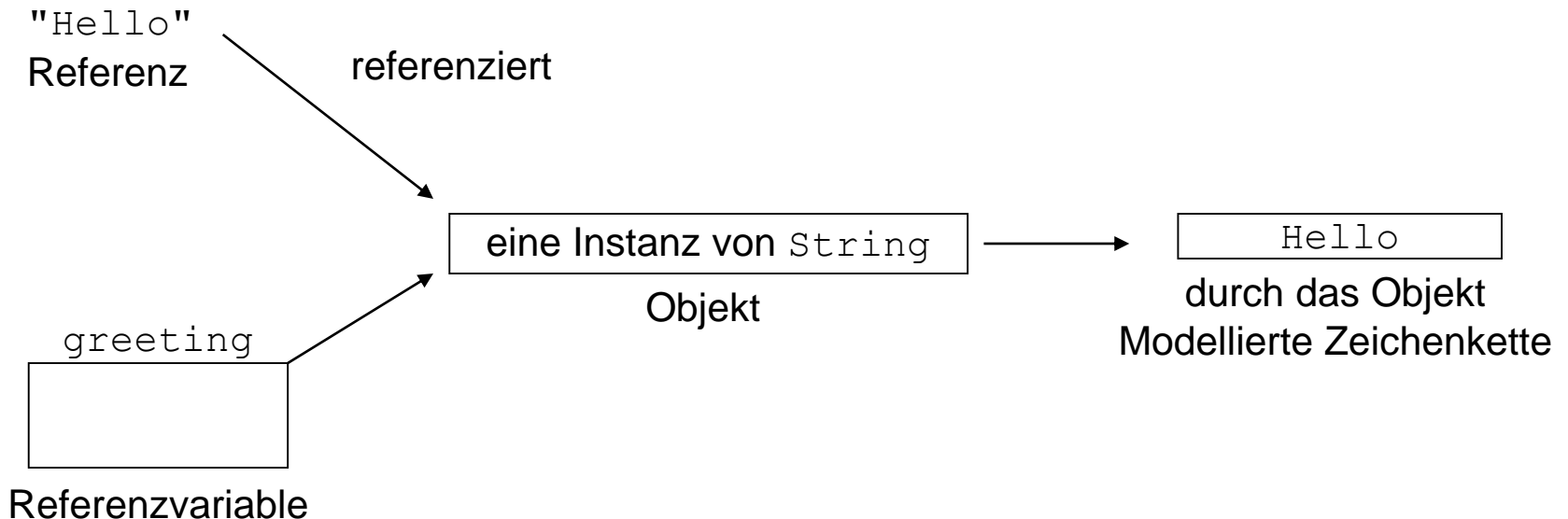
```
output.println(greeting.toUpperCase());
```

Effekt einer Wertzuweisung an eine Referenz-Variable

Nach

```
greeting = "Hello";
```

referenzieren "Hello" und `greeting` **dasselbe** Objekt.



Wertzuweisung versus Gleichheit

- Betrachte

```
t = "Springtime";  
t = "Wintertime";
```
- **Eine Wertzuweisung ordnet einer Variablen den Wert auf der rechten Seite des Statements zu.**
- **Der bisherige Wert der Variablen geht verloren.**
- Nach der ersten Zuweisung ist der Wert von `t` die Referenz auf das durch `"Springtime"` referenzierte `String`-Objekt.
- Nach der zweiten Zuweisung ist der Wert von `t` die Referenz auf das `"Wintertime"`-Objekt.
- Wir sagen: „**Eine Wertzuweisung ist *imperativ*.**“
- **Variablen enthalten immer nur den letzten, ihnen zugewiesenen Wert.**

Rollen von Variablen

- Je nachdem, wo Variablen auftreten, können sie
 1. Informationen speichern oder
 2. den in ihnen gespeicherten Wert repräsentieren.
- Beispiele:

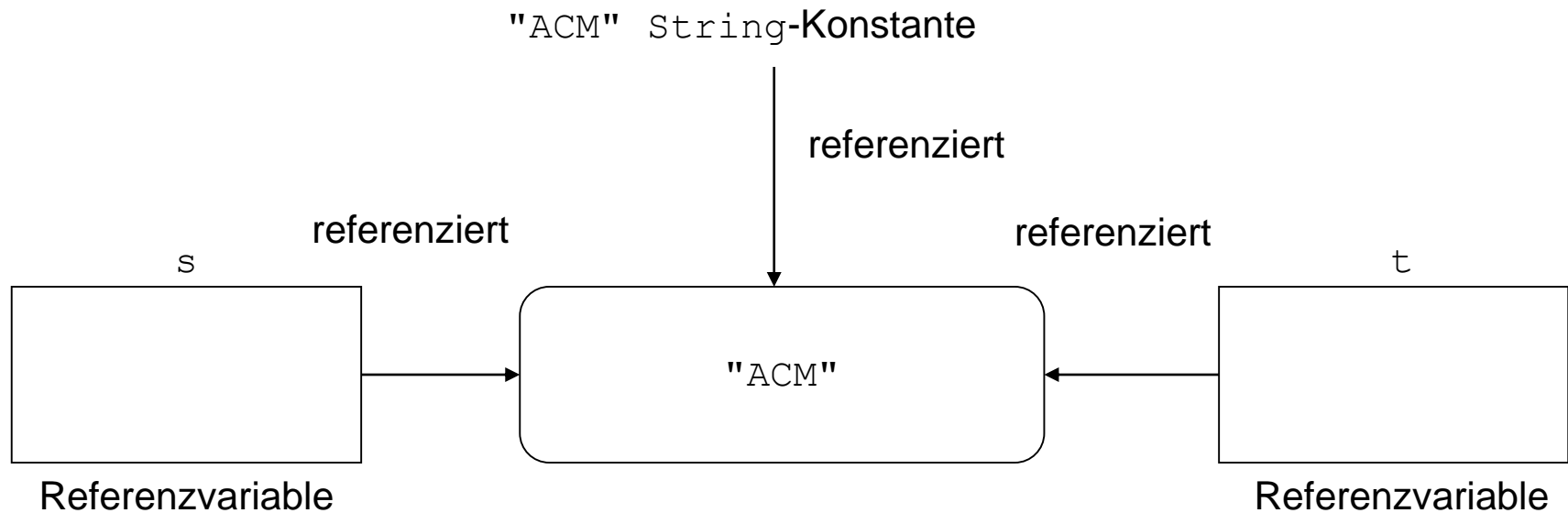
```
String s, t;  
s = "Springtime";  
t = s;
```
- Die erste Zuweisung speichert in `s` die Referenz auf das "Springtime"-Objekt (**Fall 1**).
- Die zweite Zuweisung bewirkt, dass `t` und `s` **dasselbe** Objekt referenzieren (**Fall 1 für `t` und Fall 2 für `s`**).

Mehrere Referenzen auf dasselbe Objekt

```
String s, t;
```

```
s = "ACM";
```

```
t = s;
```



Unabhängigkeit von Variablen

```
String s;  
String t;  
s = "Inventory";  
t = s;
```

- Nach der Anweisung `t = s` referenziert `t` dasselbe Objekt wie `s`.
- Wenn wir anschließend `s` einen neuen Wert zuweisen, z.B. mit `s = "payroll"`, ändert das den Wert für `t` nicht.
- Durch `t = s` wird lediglich der Wert von `s` in `t` kopiert.
- **Eine Wertzuweisung realisiert keine permanente Gleichheit.**
- **Variablen sind unabhängig voneinander**, d.h. ihre Werte können unabhängig voneinander geändert werden.

Reihenfolge von Statements (erneut) (1)

```
String greeting;  
String bigGreeting;  
greeting = "Hello, World";  
bigGreeting = greeting.toUpperCase();  
System.out.println(greeting);  
System.out.println(bigGreeting);
```

Ausgabe

```
Hello, World  
HELLO, WORLD
```

Reihenfolge von Statements (erneut) (2)

- Alternativ dazu hätten wir auch die folgende Reihenfolge verwenden können:

```
String greeting;  
greeting = "Hello, World";  
System.out.println(greeting);  
String bigGreeting;  
bigGreeting = greeting.toUpperCase();  
System.out.println(bigGreeting);
```

- **Deklarationen** können an jeder Stelle auftauchen, vorausgesetzt sie **gehen jedem anderen Vorkommen der deklarierten Variablen voraus.**
- Deklarationen können auch so genannte **Initialisierungen** enthalten:

```
String greeting = "Hello, World";
```

Methoden, Argumente und Return-Werte

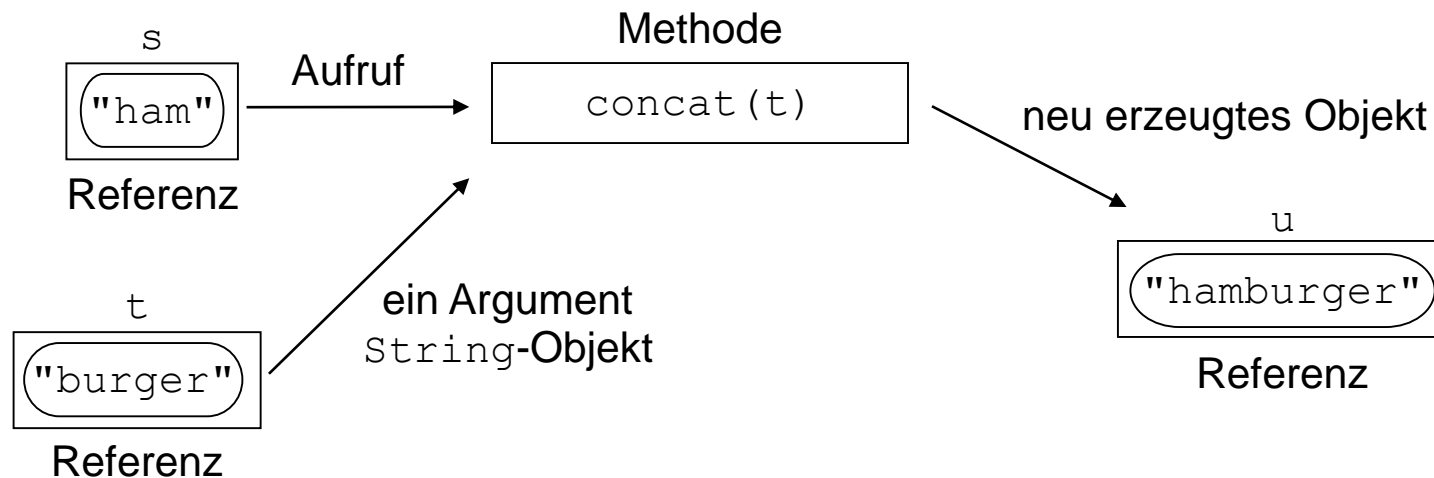
Methode	Return-Wert	Argumente
toUpperCase	String-Objekt-Referenz	keine
toLowerCase	String-Objekt-Referenz	keine
length	eine Zahl	keine
trim	String-Objekt-Referenz	keine
concat	String-Objekt-Referenz	String-Objekt-Referenz
substring	String-Objekt-Referenz	eine Zahl
substring	String-Objekt-Referenz	zwei Zahlen

Eigenschaften dieser Methoden

- `length` gibt die Anzahl der Zeichen Empfängerobjekt zurück.
`"Hello".length()` ist der Wert 5.
- `trim` liefert eine Referenz auf ein String-Objekt, welches sich durch das Argument dadurch unterscheidet, dass führende oder nachfolgende Leer- oder Tabulatorzeichen fehlen.
`" Hello ".trim()` ist eine Referenz auf `"Hello"`.
- `concat` liefert eine Referenz auf ein String-Objekt, welches sich durch das Anhängen des Argumentes an das Empfängerobjekt ergibt.
`"ham".concat("burger")` ist eine Referenz auf `"hamburger"`.

Wirkung der Methode concat

```
String s, t, u;  
s = "ham";  
t = "burger";  
u = s.concat(t);
```



Die Varianten der Methode `substring`

- `substring` erzeugt eine Referenz auf eine Teilsequenz der Zeichenkette des Empfängerobjekts.
- In Java startet die Nummerierung bei 0.
- Die Variante mit einem Argument gibt eine Referenz auf den Teilstring zurück, der an der durch den Wert des Argumentes spezifizierten Position beginnt.
- Die Version mit zwei Argumenten gibt eine Referenz auf den Teilstring, der an der durch den Wert des ersten Argumentes gegebenen Position beginnt und unmittelbar vor der durch das zweite Argument gegebenen Position endet.

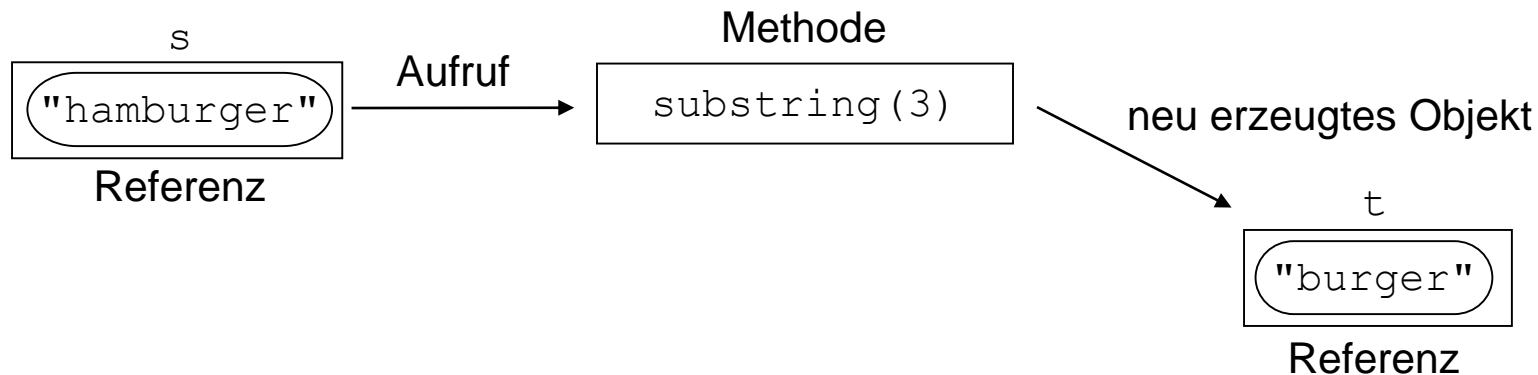
The diagram shows the string "hamburger" with indices 0 through 8 below each character. Two arrows point from the text "Vierter Buchstabe" (index 4, 'e') and "Erster Buchstabe" (index 1, 'a') to the characters 'b' at index 3 and 'u' at index 4, respectively. Below this, the text "Substring beginnend an Position 3" has an arrow pointing to a box containing the substring "burger", which starts at index 3 and ends at index 8.

Funktion der Methode `substring` mit einem Argument

```
String s, t;
```

```
s = "hamburger";
```

```
t = s.substring(3);
```



Funktion der Methode `substring` mit zwei Argumenten

```
String s, t;
```

```
s = "hamburger";
```

```
t = s.substring(3, 7);
```

`"hamburger".substring(3, 7)`

hamburger, von 3 bis 7

Substring endet hier

Substring beginnt hier

h a m (b) u r (g) e r

0 1 2 3 4 5 6 7 8

Beispiel: Finden des mittleren Zeichens einer Zeichenkette

```
String word = "antidisestablishmentarianism";
```

Zur Berechnung der mittleren Position des Wortes verwenden wir

```
word.length() / 2
```

Das mittlere Zeichen berechnen wir dann mit:

```
word.substring(word.length() / 2, word.length() / 2 + 1);
```

Das komplette Programm

```
class Program2 {  
    public static void main(String[] arg) {  
        String word = "antidisestablishmentarianism";  
        String middle;  
        middle = word.substring(word.length()/2,  
                                word.length()/2 + 1);  
        System.out.println(middle);  
    }  
}
```

Ausgabe des Programms:

s

Überladung/Overloading

- Die `String`-Klasse hat **zwei Methoden** mit Namen `substring`.
- Beide Methoden haben verschiedene **Signaturen**, denn sie unterscheiden sich in den Argumenten, die sie benötigen und bieten unterschiedliche Funktionalitäten.
- **Methoden mit gleichem Namen und unterschiedlichen Signaturen** heißen **überladen** bzw. **overloaded**.

Kaskadieren von Methodenaufrufen

```
String s1 = "ham", s2 = "bur", s3 = "ger", s4;
```

Um eine Referenz auf die Verkettung der drei durch `s1`, `s2`, und `s3` referenzierten `String`-Objekte zu erzeugen, müssten wir schreiben:

```
s4 = s1.concat(s2);  
s4 = s4.concat(s3);
```

Dies geht jedoch einfacher mit `s4 = s1.concat(s2).concat(s3);`

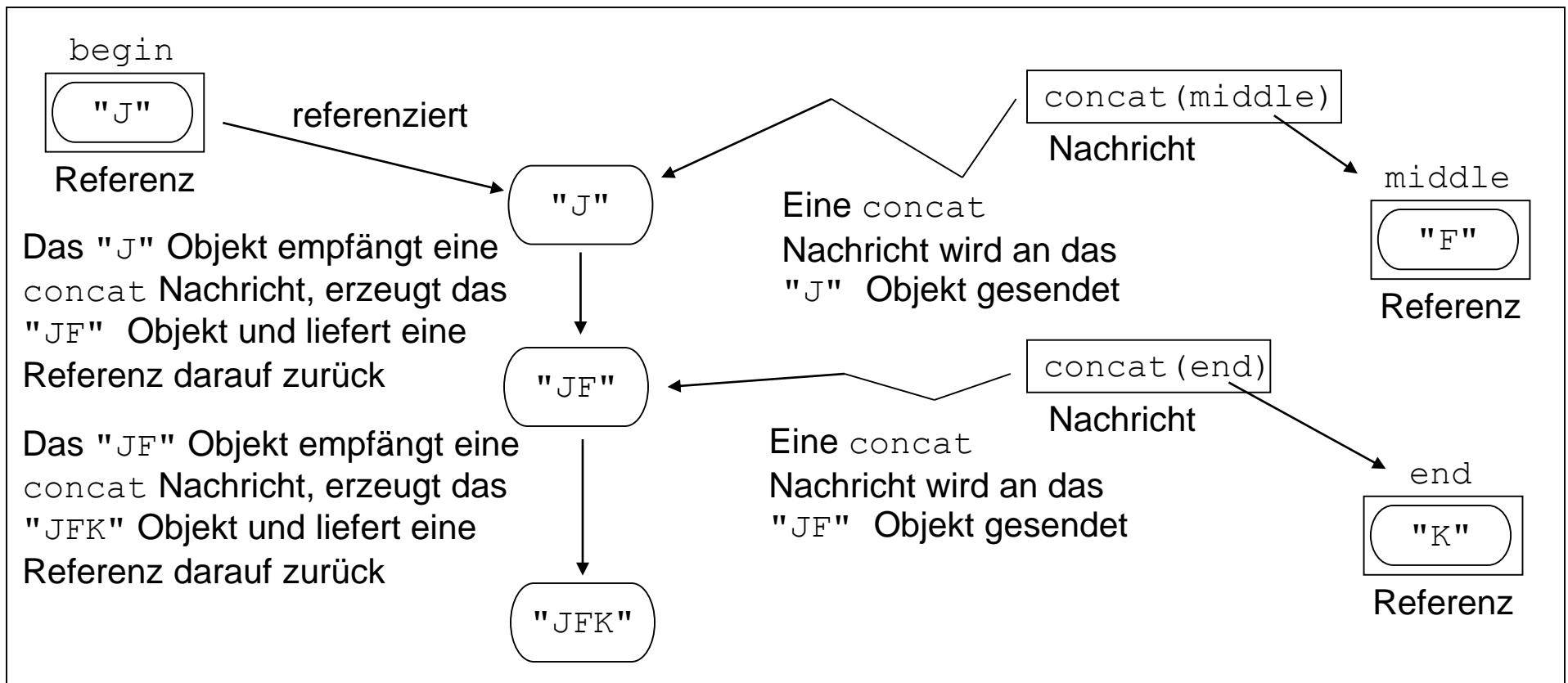
Funktionsweise:

1. Die Message `concat(s2)` wird an `s1` gesendet.
2. Die Message `concat(s3)` wird an das Ergebnisobjekt gesendet.
3. Die Referenz auf das dadurch erzeugte `String`-Objekt wird `s4` zugewiesen.

Funktion kaskadierter Nachrichten

```
String begin = "J", middle = "F", end = "K";
```

```
System.out.println(begin.concat(middle).concat(end));
```



Schachteln von Methodenaufrufen / Komposition

Alternativ zu Kaskaden wie in

```
s4 = s1.concat(s2).concat(s3);
```

können wir Methodenaufrufe auch **schachteln**, was einer **Komposition** entspricht:

```
s4 = s1.concat(s2.concat(s3));
```

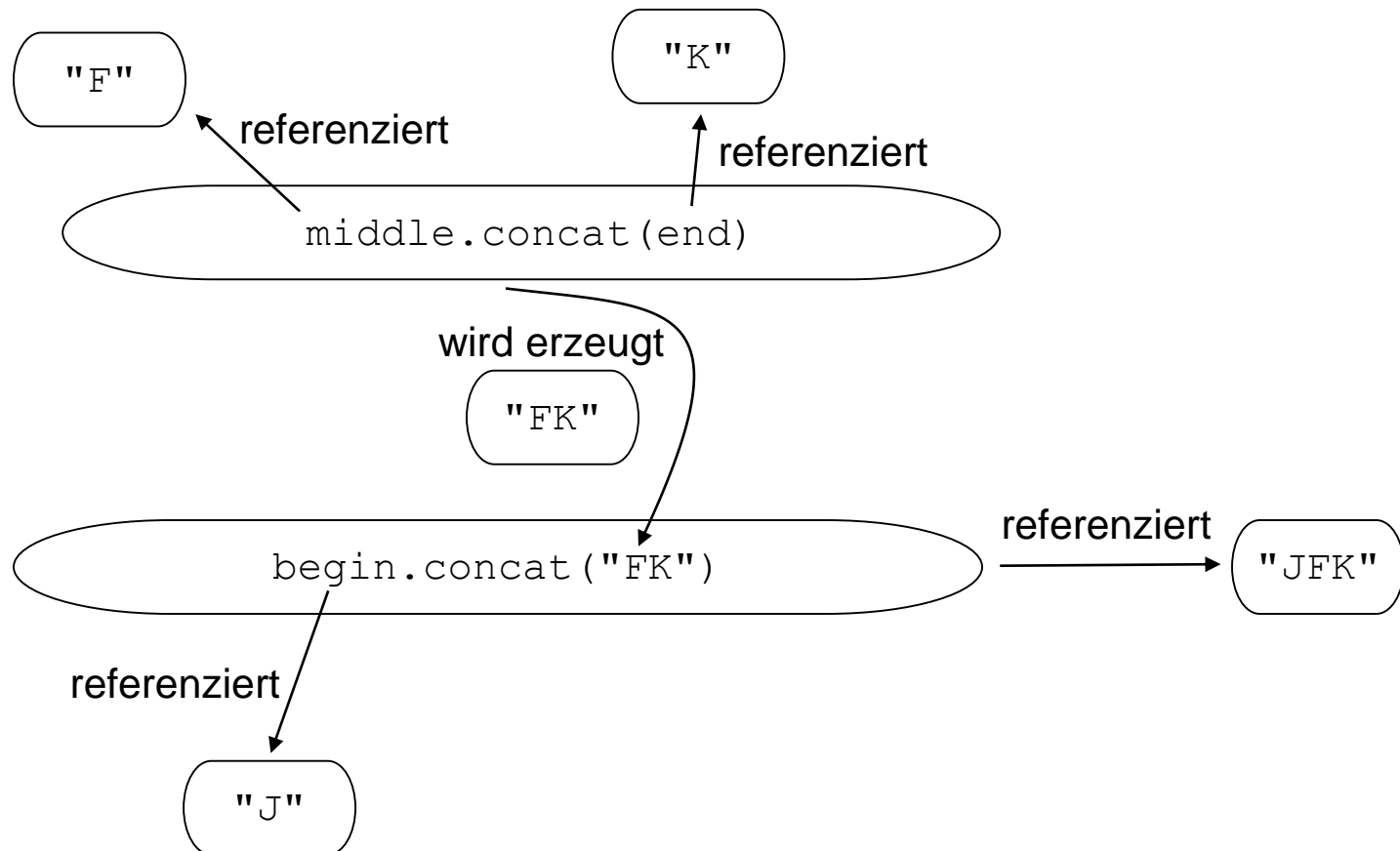
Auswertung des Ausdrucks auf der rechten Seite:

1. Die Message `concat(s3)` wird an `s2` gesendet.
2. An das `s1`-Objekt wird eine `concat`-Nachricht geschickt, die als Argument eine Referenz auf das in Schritt 1 erzeugte Ergebnisobjekt hat.
3. Die Referenz auf das dadurch erzeugte `String`-Objekt wird schließlich `s4` zugewiesen.

Wirkung der Komposition

```
String begin = "J", middle = "F", end = "K";
```

```
System.out.println(begin.concat(middle.concat(end)));
```



Weitere Eigenschaften von String-Objekten

- `String`-Objekte können nicht verändert werden. Alle Funktionen liefern als Return-Wert neue Objekte.
- Der leere String `""` hat die Länge 0, d.h. `"".length()` liefert 0.

Erzeugen von Objekten

- Jede Klasse hat wenigstens eine Methode zum Erzeugen von Objekten.
- Solche Methoden heißen **Konstruktoren**.
- Der Name eines Konstruktors stimmt stets mit dem der Klasse überein.
- Wie andere Methoden auch, können Konstruktoren Argumente haben.
- Da wir mit Konstruktoren neue Objekte erzeugen wollen, können wir sie an kein Objekt senden.
- In Java verwenden wir das **Schlüsselwort** `new`, um einen Konstruktor aufzurufen:

```
new String("hello world")
```

- Dies **erzeugt** ein `String`-Objekt und sendet ihm die Nachricht

```
String("hello world").
```

Die Operation new

- Das **Schlüsselwort** `new` bezeichnet eine **Operation**, die einen Wert zurückgibt.
- Der **Return-Wert** einer `new`-Operation ist die **Referenz** auf das neu erzeugte Objekt.
- Wir nennen `new` einen **Operator**.

Sichern neu erzeugter Objekte

- Der **new-Operator** liefert uns eine Referenz auf ein neues Objekt.
- Um dieses Objekt im Programm verwenden zu können, müssen wir die **Referenz in einer Referenzvariablen sichern**.

Beispiel:

```
String s, t, upper, lower;  
s = new String("Hello");  
t = new String(); // identisch mit ""  
  
upper = s.toUpperCase();  
lower = s.toLowerCase();  
  
System.out.println(s);
```

Zusammenfassung (1)

- Das **Verhalten von Objekten** wird durch **Methoden** spezifiziert.
- Die **Signatur** einer Methode besteht aus dem **Namen** der Methode sowie der **Anzahl und den Typen der Argumente**.
- Der **Prototyp** einer Methode ist die **Signatur zusammen mit dem Return-Wert**.
- Wird eine **Nachricht** an ein Objekt **gesendet**, wird der **Code des Aufrufers unterbrochen** bis die Methode ausgeführt worden ist.
- Einige Methode haben **Return-Werte**. Wenn eine Methode **keinen Return-Wert** hat, ist der Return-Typ **void**.
- **Variablen** können **Werte zugeordnet** werden.

Zusammenfassung (2)

- **Verschiedene Variablen** sind **unabhängig** voneinander.
- Jede Variable hat zu einem Zeitpunkt **nur einen Wert**.
- **Wertzuweisungen sind destruktiv**, d.h. sie löschen den vorhergehenden Wert der Variablen.
- **Referenzwerte**, die von Methoden zurückgegeben werden, **können Variablen zugewiesen werden**.
- **Return-Werte** können aber auch **Empfänger** neuer Nachrichten sein. Dies heißt **Kaskadierung**.
- **Return-Werte** können auch als **Argumente** verwendet werden. Dieser Prozess heißt **Komposition**.

Zusammenfassung (3)

- Neue Objekte einer Klasse können mit dem **new-Operator** erzeugt werden.
- Zusammen mit dem `new`-Operator verwenden wir den **Konstruktor**, der den gleichen Bezeichner hat wie die Klasse selbst.

Einführung in die Informatik

Files and Streams

Arbeiten mit Dateien und Streams

Wolfram Burgard

Dateien

- Bisher gingen alle Ausgaben nach **Standard Output**, d.h. auf den **Monitor**.
- Der Vorteil von **Dateien** ist die **Persistenz**, d.h. die **Information bleibt dauerhaft erhalten**.

Grundlegende Eigenschaften von Dateien:

Dateiname: Üblicherweise setzen sich Dateinamen aus Zeichenketten zusammen.

Inhalt (Daten): Die Daten können beliebige Informationen sein: Brief, Einkaufsliste, Adressen, ...

Grundlegende Datei-Operationen

- Erzeugen einer Datei
- In eine Datei schreiben.
- Aus einer Datei lesen.
- Eine Datei löschen.
- Den Dateinamen ändern.
- Die Datei überschreiben, d.h. nur den Inhalt verändern.

Die File-Klasse

- Java stellt eine vordefinierte Klasse `File` zur Verfügung.
- Der **Konstruktor** für `File` nimmt als Argument den **Dateinamen**.

Beispiel:

```
File f1, f2;  
f1 = new File("letterToJoanna");  
f2 = new File("letterToMatthew");
```

Hinweis:

Wenn ein File-Objekt erzeugt wird, bedeutet das nicht, dass gleichzeitig auch die Datei erzeugt wird.

Dateien Umbenennen und Löschen

- Existierende Dateien können in Java mit `renameTo` umbenannt werden.
- Mit der Methode `delete` können vorhandene Dateien gelöscht werden.

Prototypen:

Methode	Return-Wert	Argumente	Aktion
<code>delete</code>	<code>void</code>	keine	löscht die Datei
<code>renameTo</code>	<code>void</code>	<code>File</code> -Objekt-Referenz	nennt die Datei um

Ausgabe in Dateien

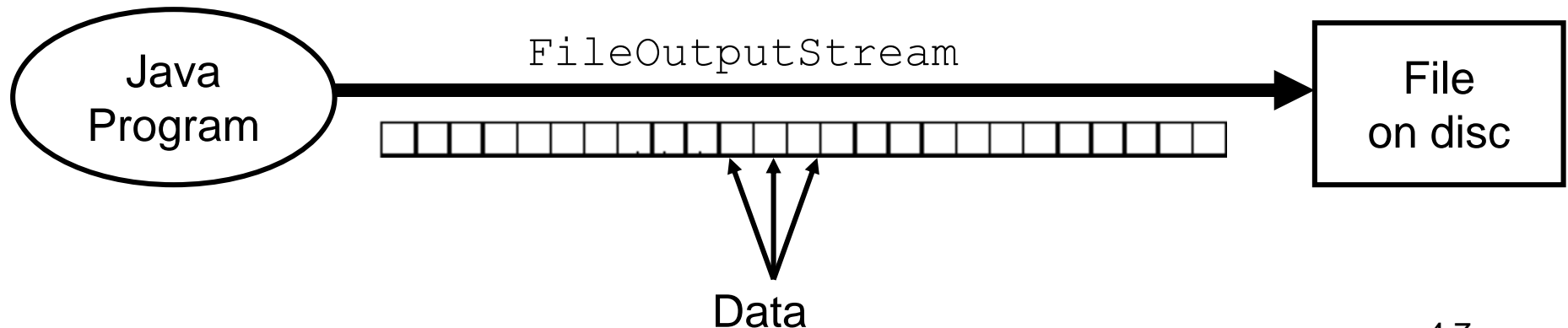
- In Java verwenden wir so genannte (Ausgabe-) Ströme bzw. (Output-) Streams, um Dateien mit Inhalt zu füllen.
- Die Klasse `FileOutputStream` stellt einen solchen Strom zur Verfügung.
- Der Konstruktor von `FileOutputStream` akzeptiert als Argument eine Referenz auf ein `File`-Objekt.
- Die Datei mit dem durch das Argument gegebenen Namen wird geöffnet.
- Ist die Datei nicht vorhanden, so wird sie erzeugt.
- Ist die Datei vorhanden, wird ihr Inhalt gelöscht.

Beispiel:

```
File f = new File("Americas.Most.Wanted");  
FileOutputStream fs = new FileOutputStream(f);
```

Wirkung von `FileOutputStream`

- `FileOutputStream` modelliert die Ausgabe als eine **Sequenz von kleinen, uninterpretierten Einheiten** bzw. **Bytes**.
- Sie stellt keine Möglichkeit zur Verfügung, die Daten zu gruppieren.
- Methoden wie `println` zum Ausgeben von Zeilen werden nicht zur Verfügung gestellt.



PrintStream-Objekte

- Um **Ausgaben auf dem Monitor** zu erzeugen, haben wir bisher die Nachrichten `println` oder `print` an das `System.out`-Objekt geschickt.
- Dabei ist `System.out` eine **Referenz auf eine Instanz der Klasse** `PrintStream`.
- Um in eine **Datei** zu **schreiben**, **erzeugen** wir ein `PrintStream-Objekt`, welches die **Datei repräsentiert**.
- **Danach wenden wir** dann die Methoden `println` oder `print` **an**.

Erzeugen von `PrintStream`-Objekten

Der **Konstruktor von `PrintStream`** akzeptiert eine Referenz auf einen `FileOutputStream`

```
File          diskFile = new File("data.out");
FileOutputStream diskFileStream =
                new FileOutputStream(diskFile);
PrintStream    target =
                new PrintStream(diskFileStream);

target.println("Hello Disk File");
```

Dieser Code erzeugt eine Datei `data.out` mit folgendem Inhalt

```
Hello Disk File
```

Eine evtl. existierende Datei mit gleichem Namen wird gelöscht und ihr Inhalt wird überschrieben.

Notwendige Schritte, um in eine Datei zu schreiben

1. Erzeugen eines `File`-Objektes
2. Erzeugen eines `FileOutputStream`-Objektes unter Verwendung des soeben erzeugten `File`-Objektes.
3. Erzeugen eines `PrintStream`-Objektes mithilfe der Referenz auf das `FileOutputStream`-Objekt.
4. Verwenden von `print` oder `println`, um Texte in die Datei auszugeben.

Kompakte Erzeugung von PrintStream-Objekten für Dateien

Die Konstruktion der `PrintStream`-Objekte kann auch ohne die `diskFileStream`-Variable durch **Schachteln von Aufrufen** erreicht werden:

```
import java.io.*;

class ProgramFileCompact {
    public static void main(String[] arg) throws IOException{
        String fileName = new String("data1.out");
        PrintStream target = new PrintStream(new
            FileOutputStream(new File(fileName)));
        target.print("Hello disk file ");
        target.println(fileName);
    }
}
```

Beispiel: Backup der Ausgabe in einer Datei

```
import java.io.*;
class Program1Backup {
    public static void main(String arg[]) throws IOException {
        PrintStream backup;
        FileOutputStream backupFileStream;
        File backupFile;

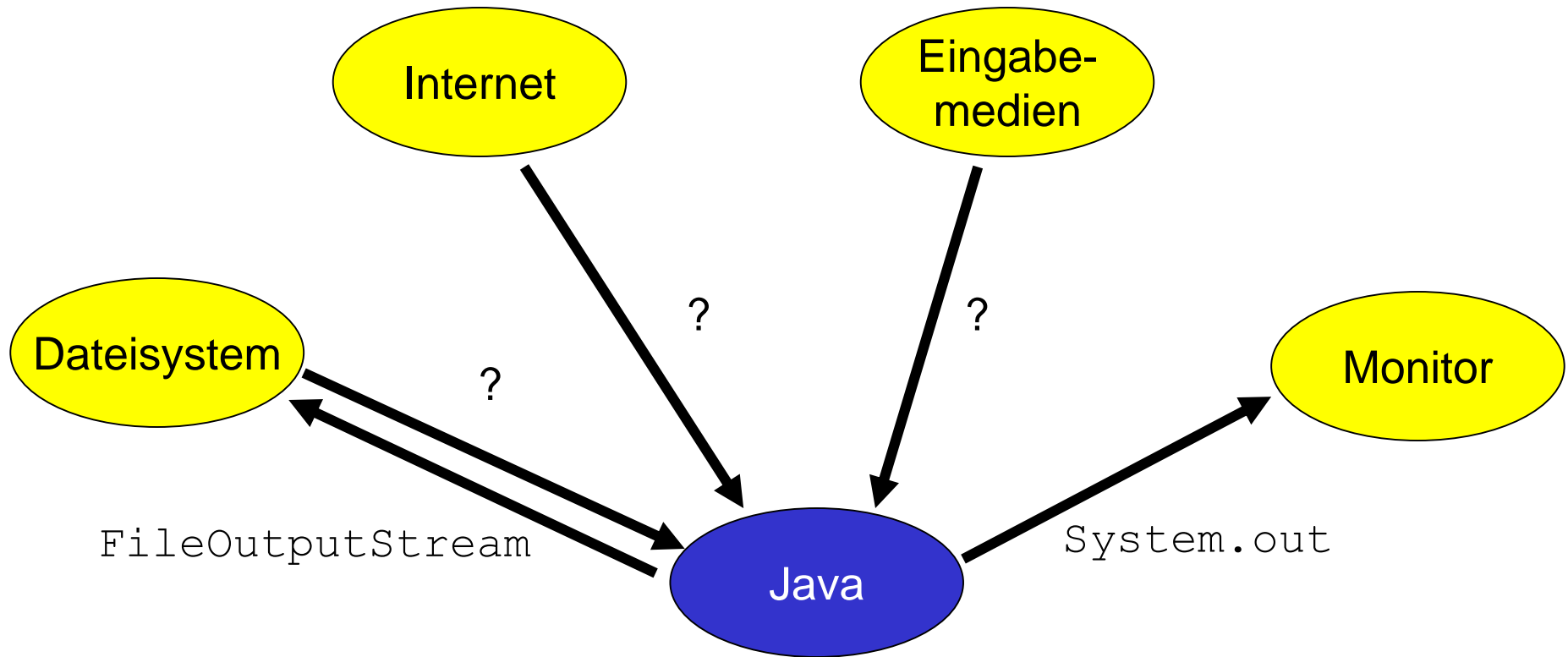
        backupFile = new File("backup");
        backupFileStream = new FileOutputStream(backupFile);
        backup = new PrintStream(backupFileStream);

        System.out.println("This is my first Java program");
        backup.println("This is my first Java program");
        System.out.println("... but it won't be my last.");
        backup.println("... but it won't be my last.");
    }
}
```

Mögliche Fehler

- Das **Erzeugen einer Datei** stellt eine **Interaktion mit externen Komponenten** dar (z.B. Betriebssystem, Hardware etc.)
- Dabei können **Fehler** auftreten, die **nicht durch das Programm selbst verschuldet** sind.
- Beispielsweise kann die **Festplatte voll** sein oder sie kann einen **Schreibfehler** haben. Weiter kann das **Verzeichnis**, in dem das Programm ausgeführt wird, **schreibgeschützt** sein.
- In solchen Fällen wird das einen Fehler produzieren.
- **Java erwartet, dass der Programmierer mögliche Fehler explizit erwähnt.**
- Dazu wird die Phrase `throws Exception` verwendet.

Mögliche Ein- und Ausgabequellen in Java



Eingabe: Ein typisches Verfahren

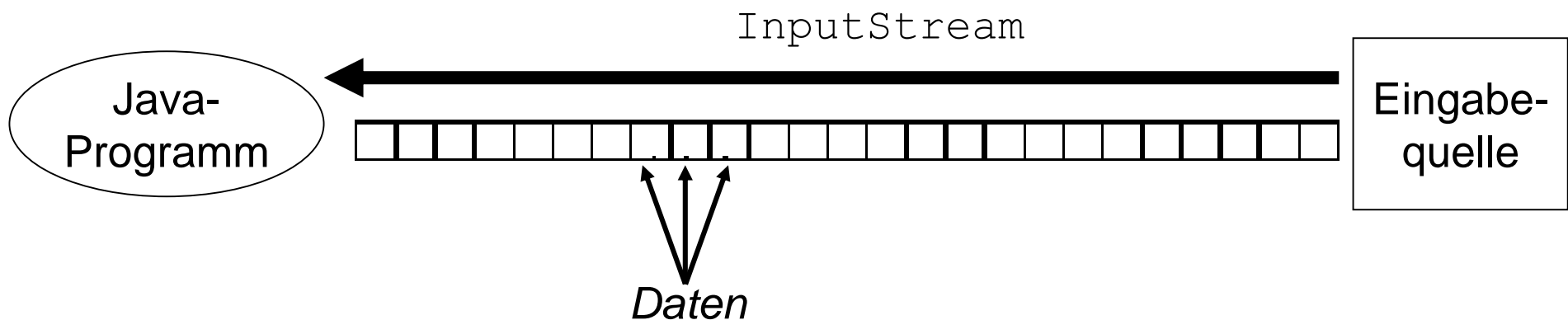
Um Eingaben von einem Eingabestrom verarbeiten zu können, müssen folgende Schritte durchgeführt werden.

1. Erzeugen Sie ein `InputStream`-Objekt, ein `FileInputStream`-Objekt oder verwenden Sie das `System.in`-Objekt.
2. Verwenden Sie dieses Eingabestrom-Objekt, um einen `InputStreamReader`-Objekt zu erzeugen.
3. Erzeugen Sie ein `BufferedReader`-Objekt mithilfe des `InputStreamReader`-Objektes.

Dabei wird `FileInputStream` für das **Einlesen aus Dateien**, `InputStream` für das **Einlesen aus dem Internet** und `System.in` für die **Eingabe von der Tastatur** verwendet.

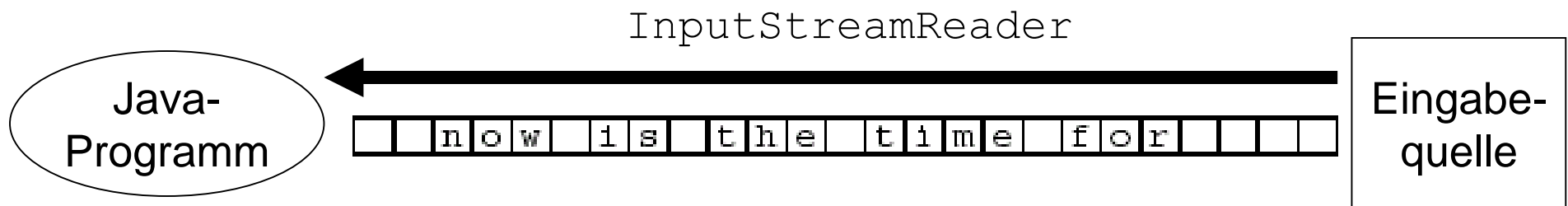
Wirkung eines InputStream-Objektes

`InputStream`-Objekte, `FileInputStream`-Objekte oder das `System.in`-Objekt modellieren die Eingabe als eine **kontinuierliche, zusammenhängende Sequenz kleiner Einheiten**, d.h. als eine **Folge von Bytes**:



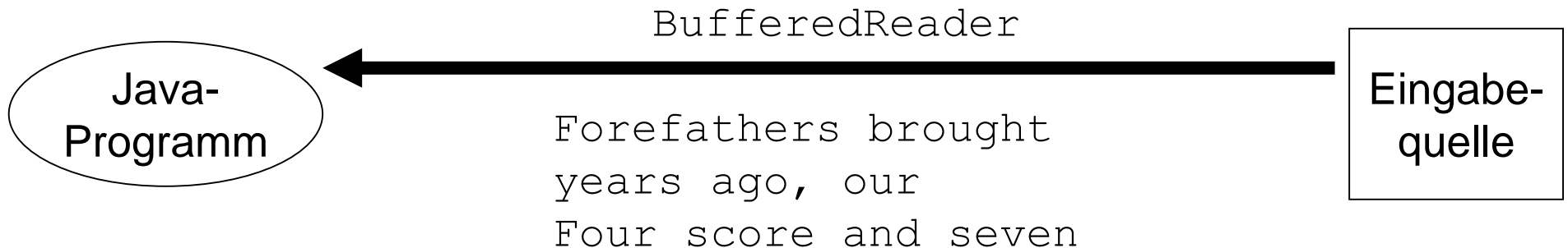
Wirkung eines InputStreamReader-Objektes

`InputStreamReader`-Objekte hingegen modellieren die Eingabe als eine **Folge von Zeichen**, sodass daraus **Zeichenketten** zusammengesetzt werden können:



BufferedReader

`BufferedReader`-Objekte schließlich modellieren die Eingabe als eine **Folge von Zeilen**, die einzeln durch `String`-**Objekte** repräsentiert werden können:



Eingabe vom Keyboard

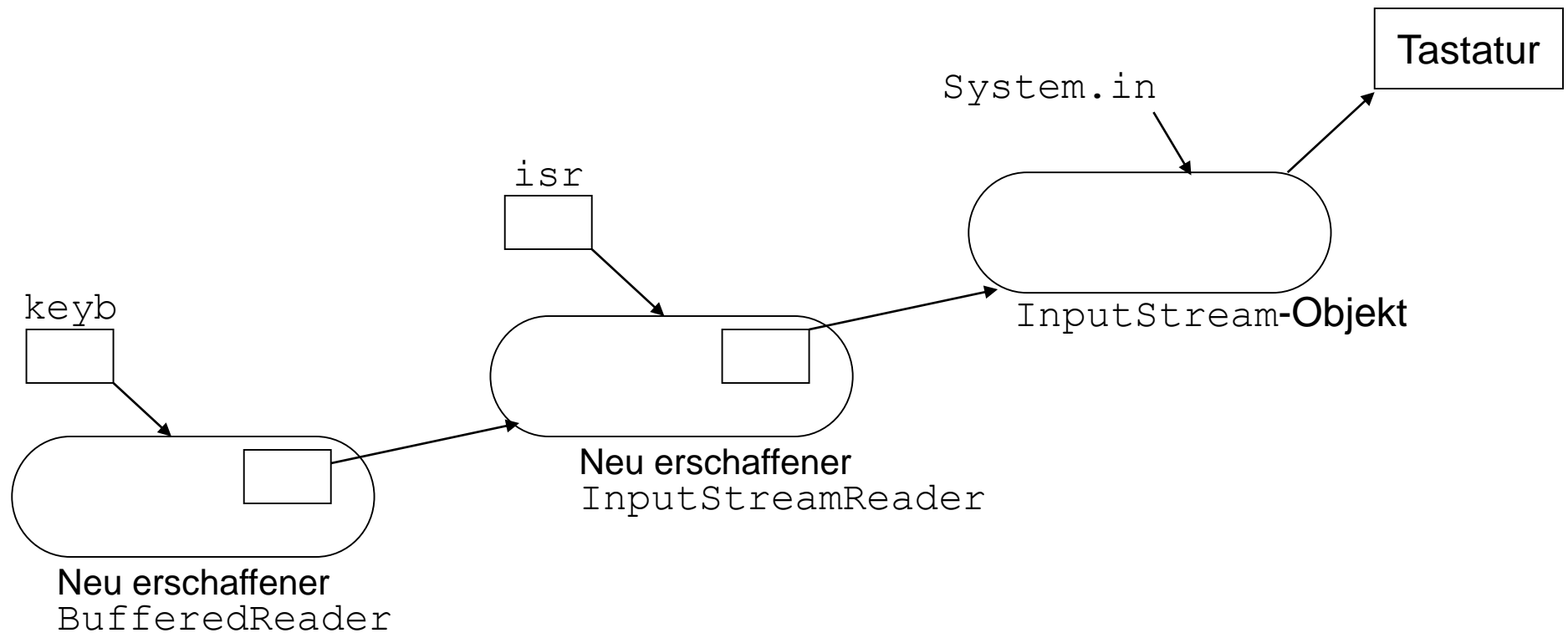
- Java stellt ein vordefiniertes `InputStream`-Objekt zur Verfügung, das die Eingabe von der Tastatur repräsentiert. `System.in` ist eine Referenz auf dieses Objekt.
- Allerdings kann man von `System.in` nicht direkt lesen.
- Vorgehen:

```
InputStreamReader isr;  
BufferedReader keyb;  
isr = new InputStreamReader(System.in)  
keyb = new BufferedReader(isr);
```

Das Einlesen geschieht dann mit:

```
keyb.readLine()
```

Schema für die Eingabe von der Tastatur mit Buffer



Beispiel: Einlesen einer Zeile von der Tastatur

Naives Verfahren zur Ausgabe des Plurals eines Wortes:

```
import java.io.*;
class Program4 {
    public static void main(String arg[]) throws IOException {
        InputStreamReader isr;
        BufferedReader keyboard;
        String inputLine;

        isr = new InputStreamReader(System.in);
        keyboard = new BufferedReader(isr);
        inputLine = keyboard.readLine();

        System.out.print(inputLine);
        System.out.println("s");
    }
}
```

Interaktive Programme

- Um den Benutzer auf eine notwendige Eingabe hinzuweisen, können wir einen so genannten **Prompt** ausgeben.
- `PrintStream` verwendet einen **Buffer**, um **Ausgabeaufträge zu sammeln**. Die Ausgabe erfolgt erst, wenn der Buffer voll oder das Programm beendet ist.
- Da dies eventuell erst nach der Eingabe sein kann, stellt die `PrintStream`-Klasse eine Methode **`flush`** zur Verfügung. Diese **erzwingt die Ausgabe des Buffers**.
- Vorgehen daher:

```
System.out.println(  
    "Type in a word to be pluralized, please ");  
System.out.flush();  
inputLine = keyboard.readLine();
```

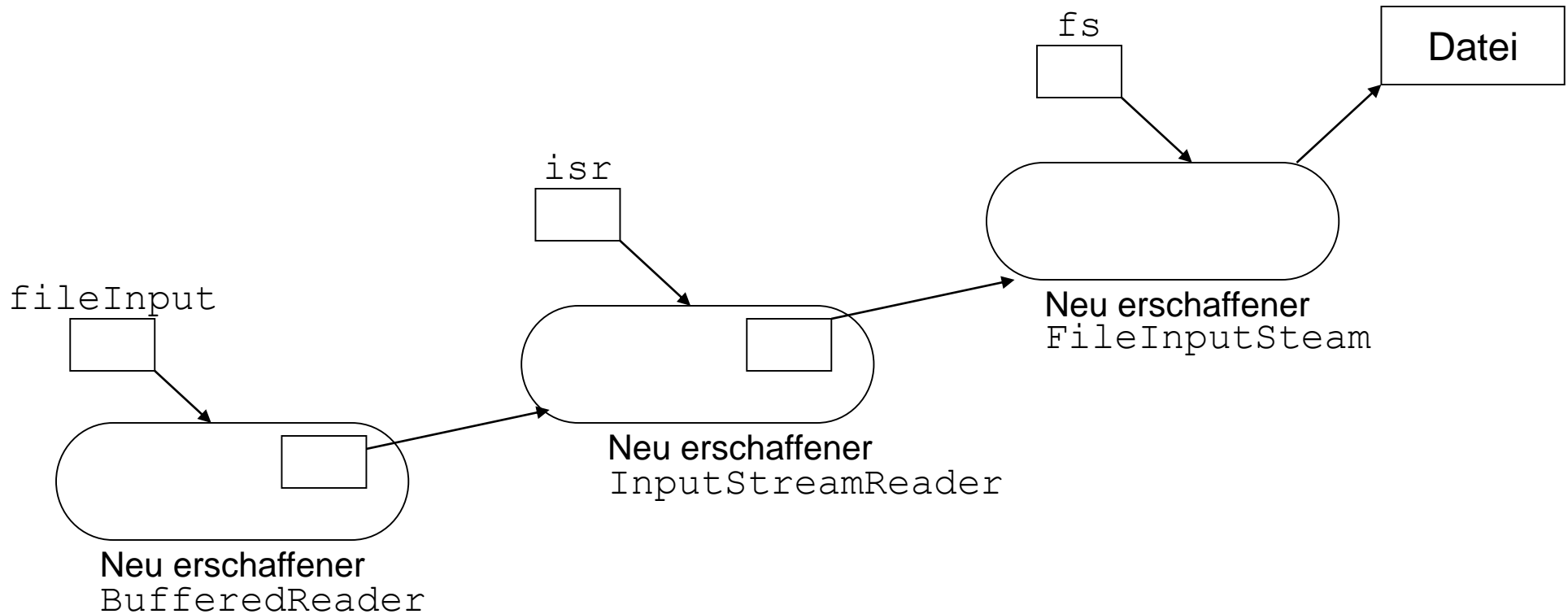
Input aus Dateien

Das **Lesen aus einer Datei** unterscheidet sich vom Lesen von der Tastatur nur dadurch, dass wir ein `FileInputStream`-Objekt und nicht das `System.in`-Objekt verwenden:

```
// Vom Dateinamen zum FileInputStream
File f = new File("Americas.Most.Wanted");
FileInputStream fs = new FileInputStream(f);

// Vom FileInputStream zum BufferedReader
InputStreamReader isr;
BufferedReader fileInput;
isr = new InputStreamReader(fs);
fileInput = new BufferedReader(isr);
```


Einlesen aus Dateien mit Buffer



Einlesen einer Zeile aus einer Datei

```
import java.io.*;
class Program5 {
    public static void main(String arg[]) throws IOException {
        String inputLine;
        // Vom Dateinamen zum FileInputStream
        File f = new File("Americas.Most.Wanted");
        FileInputStream fs = new FileInputStream(f);

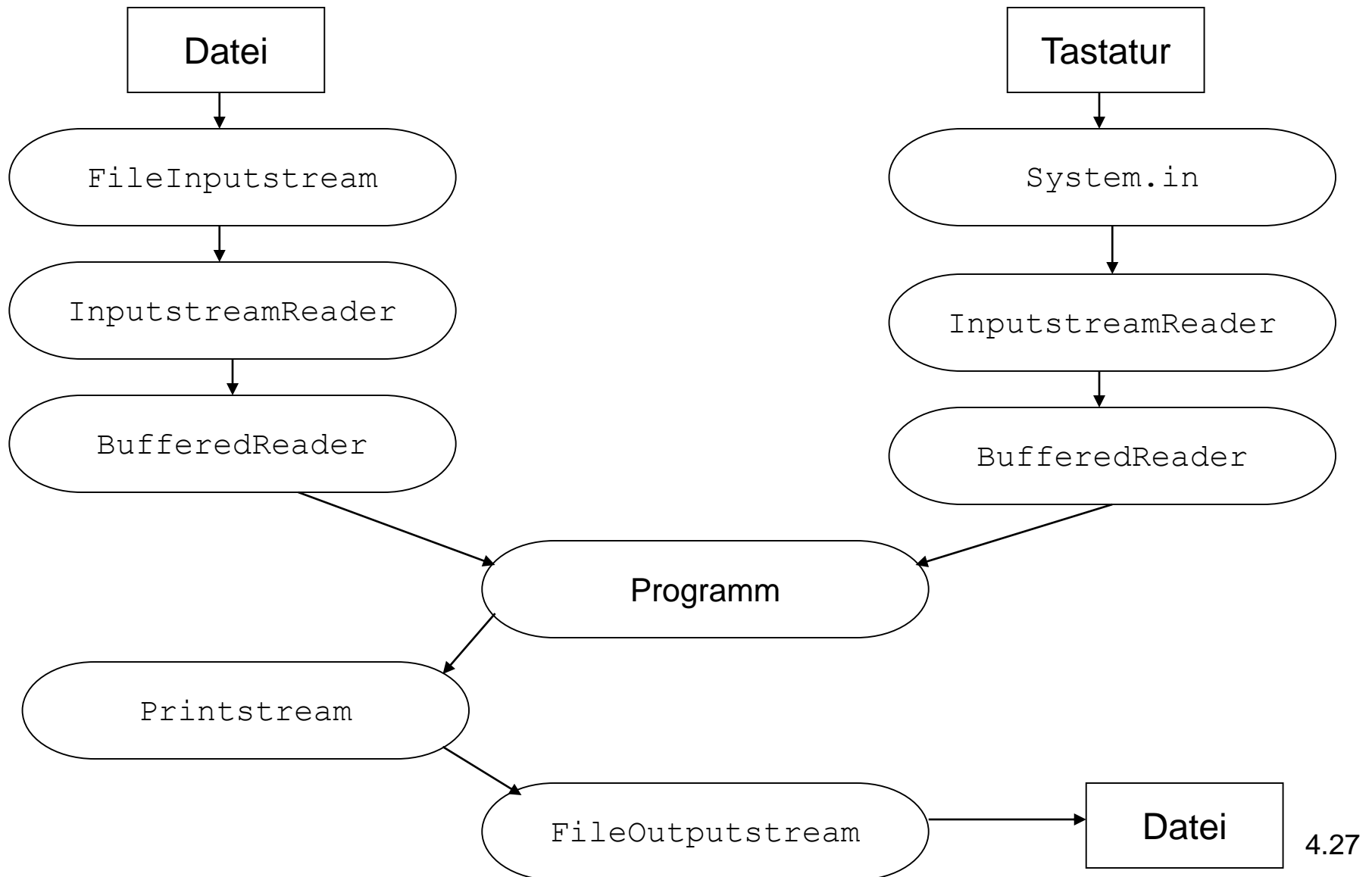
        // Vom FileInputStream zum BufferedReader
        InputStreamReader isr;
        BufferedReader fileInput;
        isr = new InputStreamReader(fs);
        fileInput = new BufferedReader(isr);

        inputLine = fileInput.readLine();
        System.out.println(inputLine);
    }
}
```

Gleichzeitige Verwendung mehrerer Streams: Kopieren einer Datei

1. Frage nach Quelldatei (und Zieldatei).
2. Lies Quelldatei.
3. Schreibe Zieldatei.

Schematische Darstellung



Daten aus dem Internet einlesen

Computer-Netzwerk: Gruppe von Computern, die untereinander direkt Informationen austauschen können (z.B. durch eine geeignete Verkabelung).

Internet: Gruppe von Computer-Netzwerken, die es Rechnern aus einem Netz erlaubt, mit Computern aus dem anderen Netz zu kommunizieren.

Internet-Adresse: Eindeutige Adresse, mit deren Hilfe jeder Rechner im Netz eindeutig identifiziert werden kann. Beispiele:

`www.informatik.uni-freiburg.de`

`www.uni-freiburg.de`

`www.whitehouse.gov`

Netzwerk-Ressource: Einheit von Informationen wie z.B. Texte, Bilder, Sounds etc.

URL: (Abk. für Universal Ressource Locator)
Eindeutige Adresse von Netzwerk-Ressourcen.

Komponenten einer URL

Bestandteil	Beispiel	Zweck
Protokoll	http	Legt die Software fest, die für den Zugriff auf die Daten benötigt wird
Internet-Adresse	www.yahoo.com	Identifiziert den Computer, auf dem die Ressource liegt
Dateiname	index.html	Definiert die Datei mit der Ressource

Zusammengesetzt werden diese Komponenten wie folgt:

`protocol://internet address/file name`

Beispiel:

`http://www.yahoo.com/index.html`

Netzwerk-Input

- Um Daten aus dem Netzwerk einzulesen, verwenden wir ein `InputStream`-Objekt.
- Die **Java-Klassenbibliothek** stellt eine Klasse `URL` zur Verfügung, um URLs zu modellieren.
- Die `URL`-Klasse stellt einen Konstruktor mit einem String-Argument zur Verfügung:

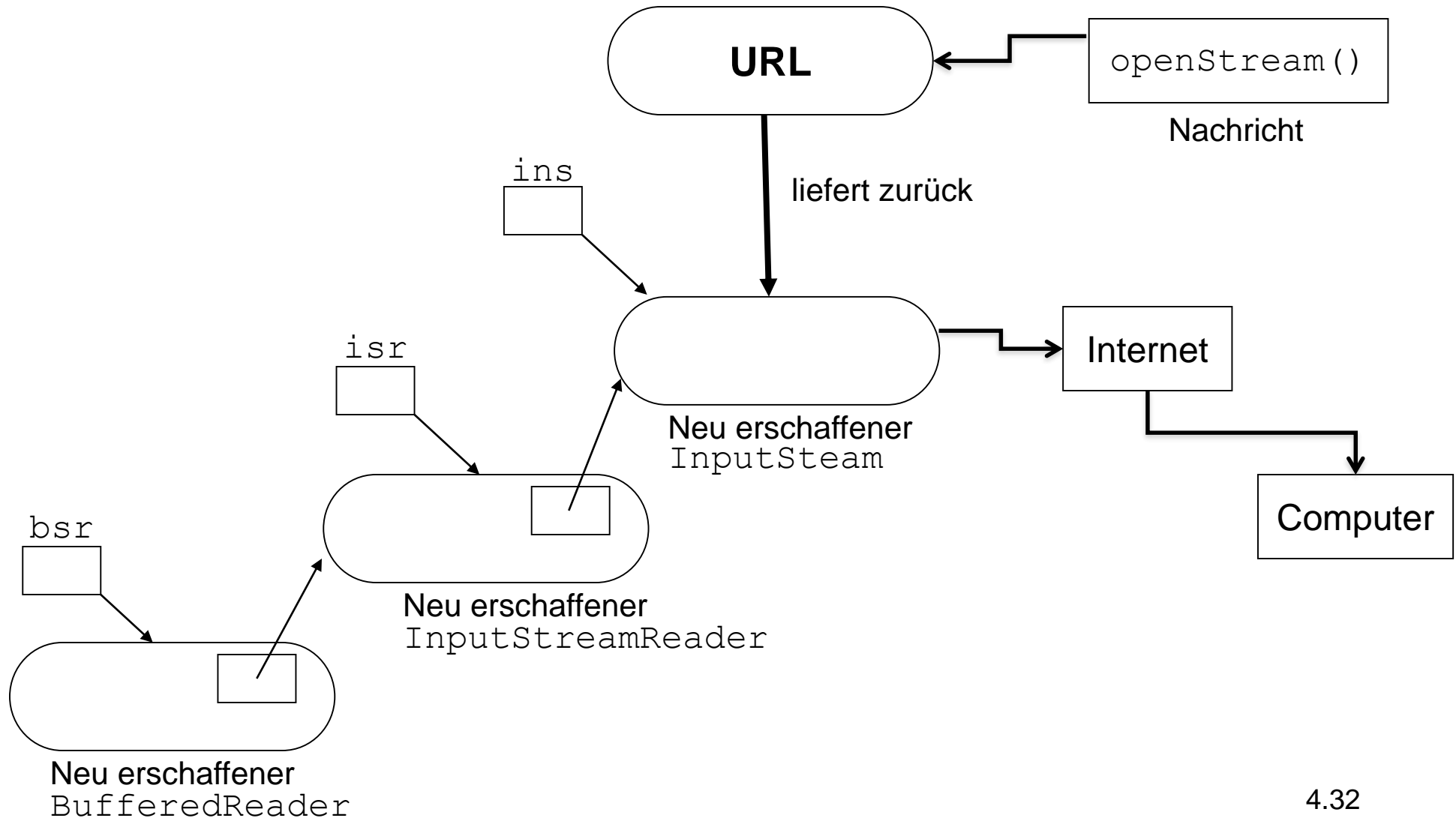
```
URL u = new URL("http://www.yahoo.com/");
```

- Weiterhin stellt sie eine Methode `openStream` bereit, die keine Argumente hat und ein `InputStream`-Objekt zurückgibt:

```
InputStream ins = u.openStream();
```
- Sobald wir einen `InputStream` haben, können wir wie üblich fortfahren:

```
InputStreamReader isr = new InputStreamReader(ins);  
BufferedReader remote = new BufferedReader(isr);  
... remote.readLine() ...
```

Einlesen aus dem Internet mit Buffer



Beispiel: Einlesen der ersten fünf Zeilen von www.informatik.uni-freiburg.de

```
import java.net.*;
import java.io.*;

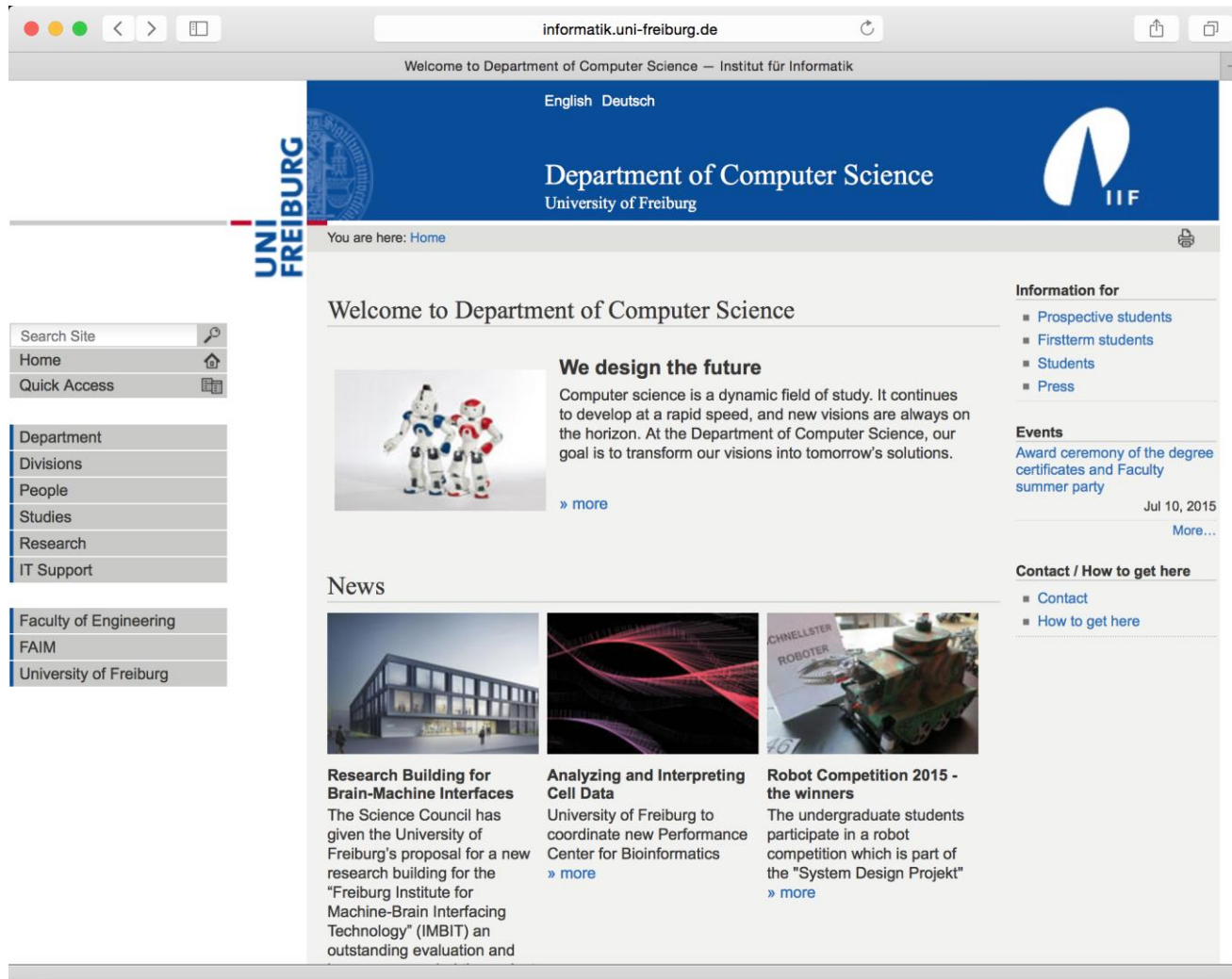
class WebPageRead {
    public static void main(String[] arg) throws Exception {
        URL u = new URL("http://www.informatik.uni-freiburg.de/");
        InputStream ins = u.openStream();
        InputStreamReader isr = new InputStreamReader(ins);
        BufferedReader webPage = new BufferedReader(isr);

        System.out.println(webPage.readLine());
        System.out.println(webPage.readLine());
        System.out.println(webPage.readLine());
        System.out.println(webPage.readLine());
        System.out.println(webPage.readLine());
    }
}
```

Ergebnis der Ausführung

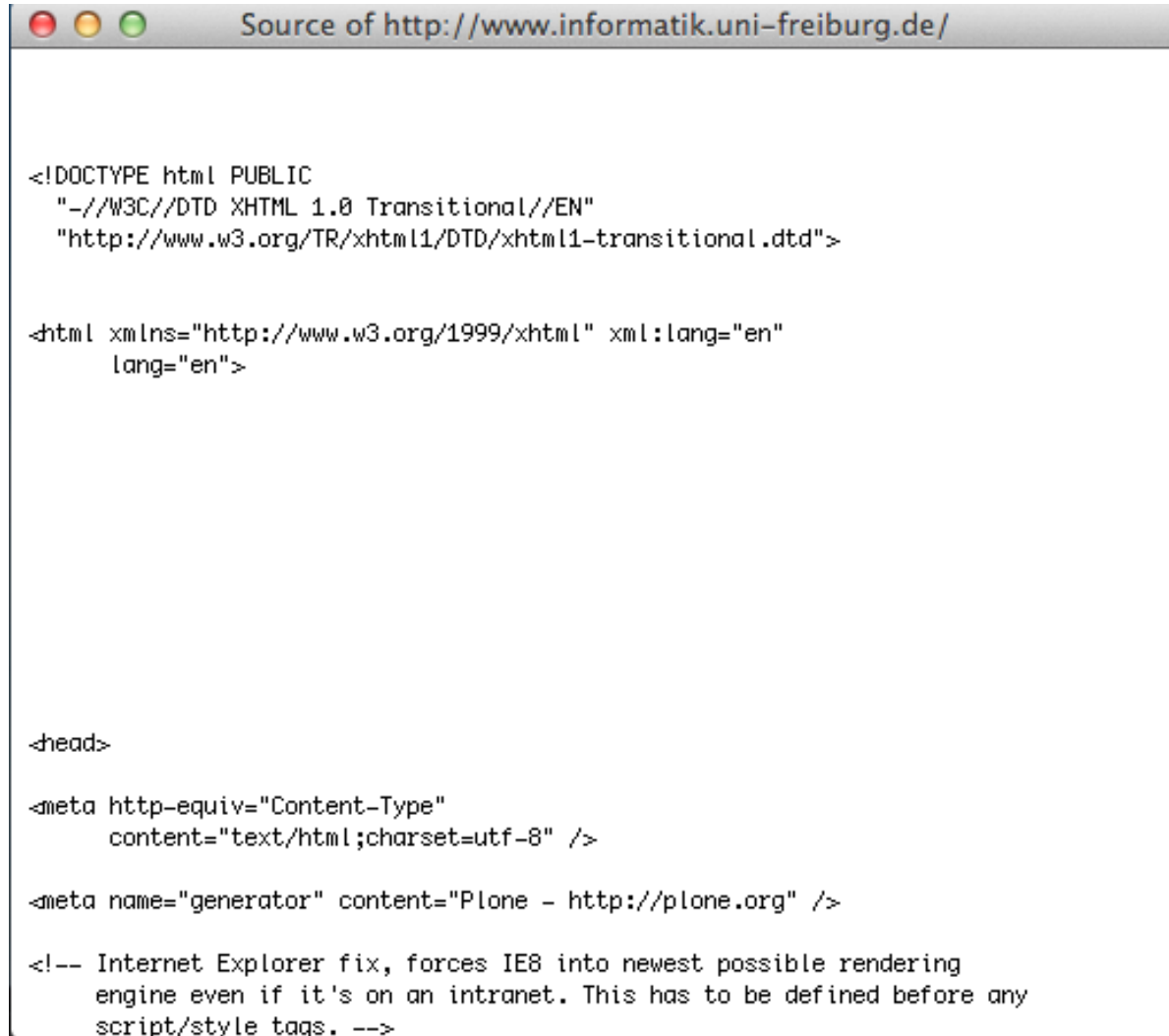
```
<!DOCTYPE html PUBLIC  
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

Die Titelseite der Informatik in Freiburg



Stand 29.4.2015

Der Quellcode der Titelseite



```
Source of http://www.informatik.uni-freiburg.de/

<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en">

<head>

<meta http-equiv="Content-Type"
  content="text/html; charset=utf-8" />

<meta name="generator" content="Plone - http://plone.org" />

<!-- Internet Explorer fix, forces IE8 into newest possible rendering
  engine even if it's on an intranet. This has to be defined before any
  script/style tags. -->
```

Zusammenfassung

- In Java lassen sich **komplexe Klassen** durch die **Kombination existierender Klassen** erreichen.
- Dabei verwenden wir **Objekte entsprechender Klassen** zur **Erzeugung von Objekte anderer Klassen**.
- Beispielsweise benutzt Java so genannte `Streams` um Daten zu lesen oder zu schreiben.
- Durch die Streams ist es möglich von der eigentlichen Datei zu abstrahieren. So kann man beispielsweise auch Daten aus dem Internet direkt lesen.
- Um beispielsweise Zeilen aus dem Internet einzulesen, benötigen wir ein `BufferedReader`-Objekt.
- Dies erfordert das Erzeugen eines `InputStreamReader`-Objektes
- Das `InputStreamReader`-Objekt hingegen benötigt ein entsprechendes `InputStream`-Objekt.

Einführung in die Informatik

Definition von Klassen

Wolfram Burgard

Motivation

- Auch wenn **Java** ein **große Zahl von vordefinierten Klassen und Methoden** zur Verfügung stellt, sind dies nur Grundfunktionen für eine Modellierung vollständiger Anwendungen.
- Um **Anwendungen** zu **realisieren**, kann man diese **vordefinierten Klassen nutzen**.
- Allerdings **erfordern** manche **Anwendungen Objekte und Methoden**, für die es **keine vordefinierten Klassen** gibt.
- **Java erlaubt** daher dem Programmierer, **eigene Klassen zu definieren**.

Klassendefinitionen:

Konstruktoren und Methoden

```
class Laugher1 {  
    public Laugher1() {  
    }  
    public void laugh() {  
        System.out.println("haha");  
    }  
}
```

- Durch diesen Code wird eine Klasse `Laugher1` definiert.
- Diese Klasse stellt eine einzige Methode `laugh` zur Verfügung

Anwendung der Klasse `Laugher1`

Auf der **Basis dieser Definition** können wir ein `Laugher1`-**Objekt deklarieren**:

```
Laugher1 x;
```

```
x = new Laugher1();
```

Dem durch `x` referenzierten **Objekt** können wir anschließend die **Message** `laugh` **schicken**:

```
x.laugh();
```

Aufbau einer Klassendefinition

- Das Textstück

```
class Laugher1 {
```

- läutet in unserem Beispiel die Definition der Klasse `Laugher1` ein.
- Die Klammern `{` und `}` werden **Begrenzer** oder **Delimiter** genannt, weil sie den Anfang und das Ende der Klassendefinition markieren.
- Zwischen diesen **Delimitern** befindet sich die Definition des **Konstruktors**

```
    public Laugher1() {  
    }
```

und einer **Methode**.

```
        public void laugh() {  
            System.out.println("haha");  
        }
```

Aufbau der Methodendefinition laugh

Die Definition der Methode besteht aus einem **Prototyp**

```
public void laugh()
```

und dem **Methodenrumpf**

```
{  
    System.out.println("haha");  
}
```

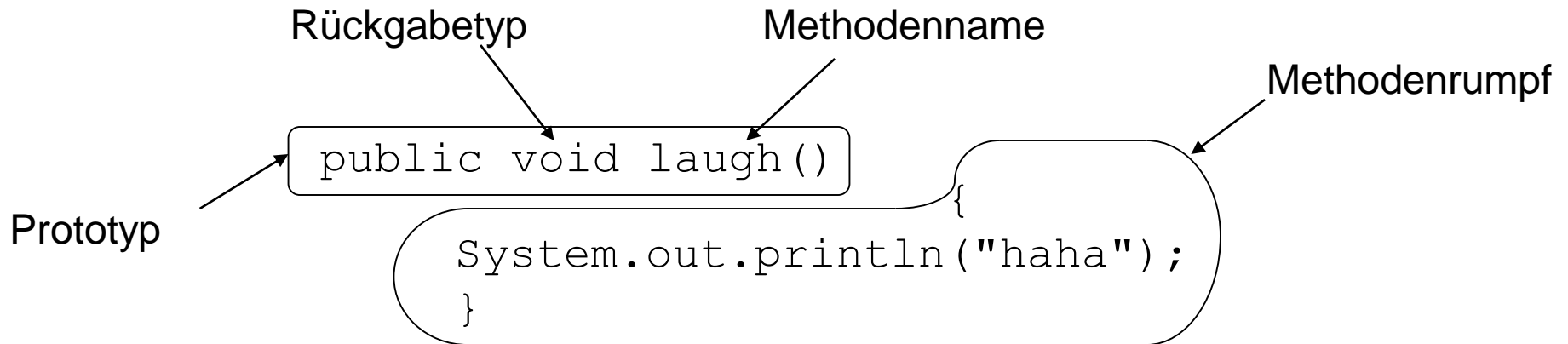
1. Der **Prototyp** der Methode beginnt mit dem **Schlüsselwort** `public`.
2. Danach folgt der **Typ des Return-Wertes**.
3. Dann wird der **Methodenname** angegeben.
4. Schließlich folgen **zwei Klammern** `()`, zwischen denen die Argumente aufgelistet werden.

Der Rumpf der Methode laugh

Der **Methodenrumpf** enthält die **Statements**, die ausgeführt werden, wenn die Methode aufgerufen wird.

Die Methode `laugh` druckt den Text "haha" auf den Monitor.

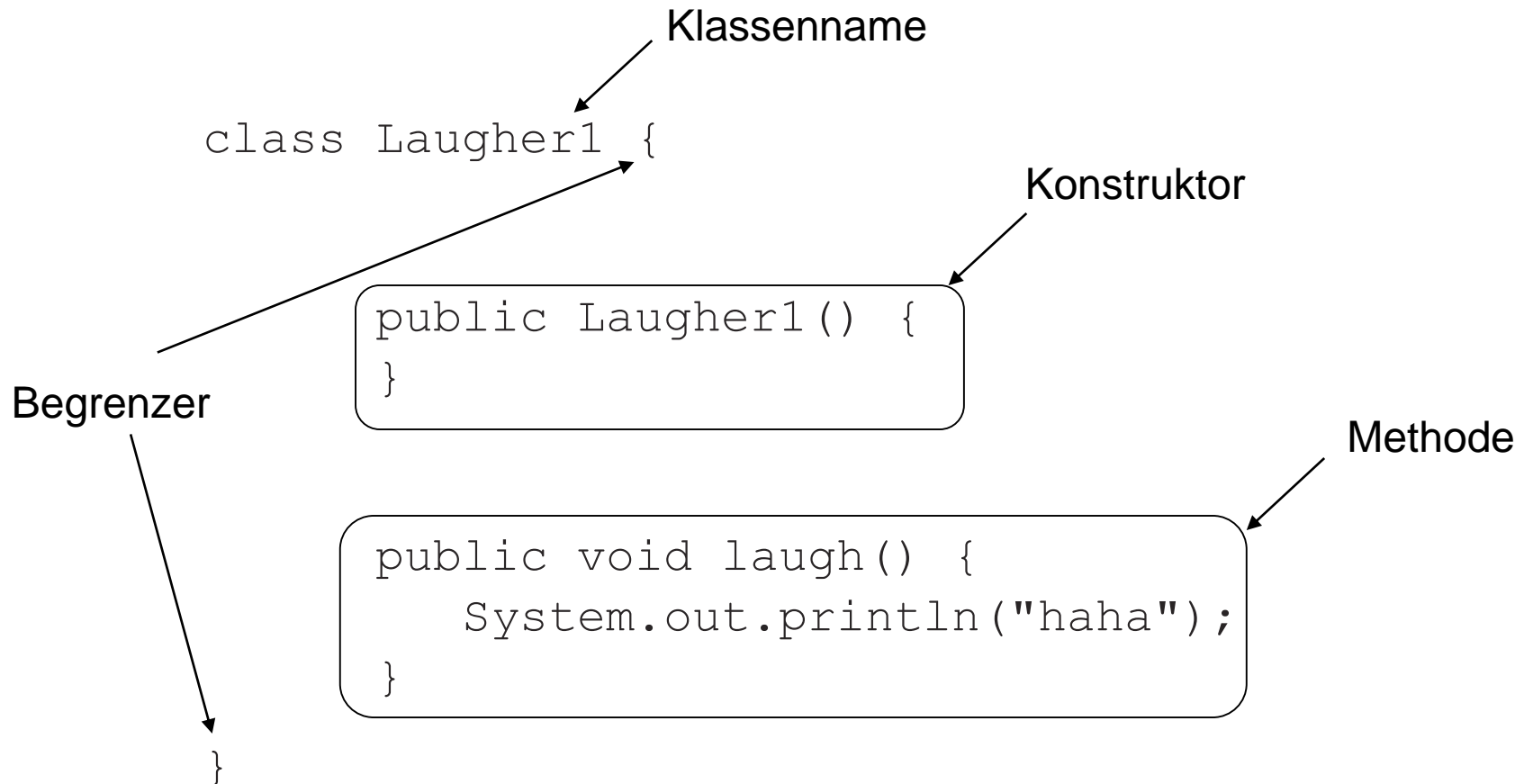
Zusammenfassend ergibt sich:



Der Konstruktor der Klasse `Laugher1`

- Die Form eines **Konstruktors** ist **identisch zu einer Methodendefinition**.
- Lediglich der **Return-Typ wird ausgelassen**.
- Konstruktoren werden immer mit dem Schlüsselwort `new` aufgerufen.
- Dieser Aufruf gibt eine **Referenz auf ein neu erzeugtes Objekt** zurück.
- Der Konstruktor `Laugher1` tut nichts.

Struktur der Klassendefinition `Laugher1`



Parameter

- In der Methode `laugh` wird der auszugebende Text vorgegeben.
- Wenn wir dem Sender einer Nachricht erlauben wollen, die Lachsilbe festzulegen, (z.B. ha oder he), müssen wir eine Methode mit **Argument** verwenden:

```
x.laugh("ha");
```

oder

```
x.laugh("yuk");
```

- **Parameter** sind **Variablen**, die im **Prototyp** einer Methode spezifiziert werden.

Definition einer Methode mit Argument

- Da unsere neue Version von `laugh` ein `String`-Argument hat, müssen wir den **Prototyp** wie folgt ändern:

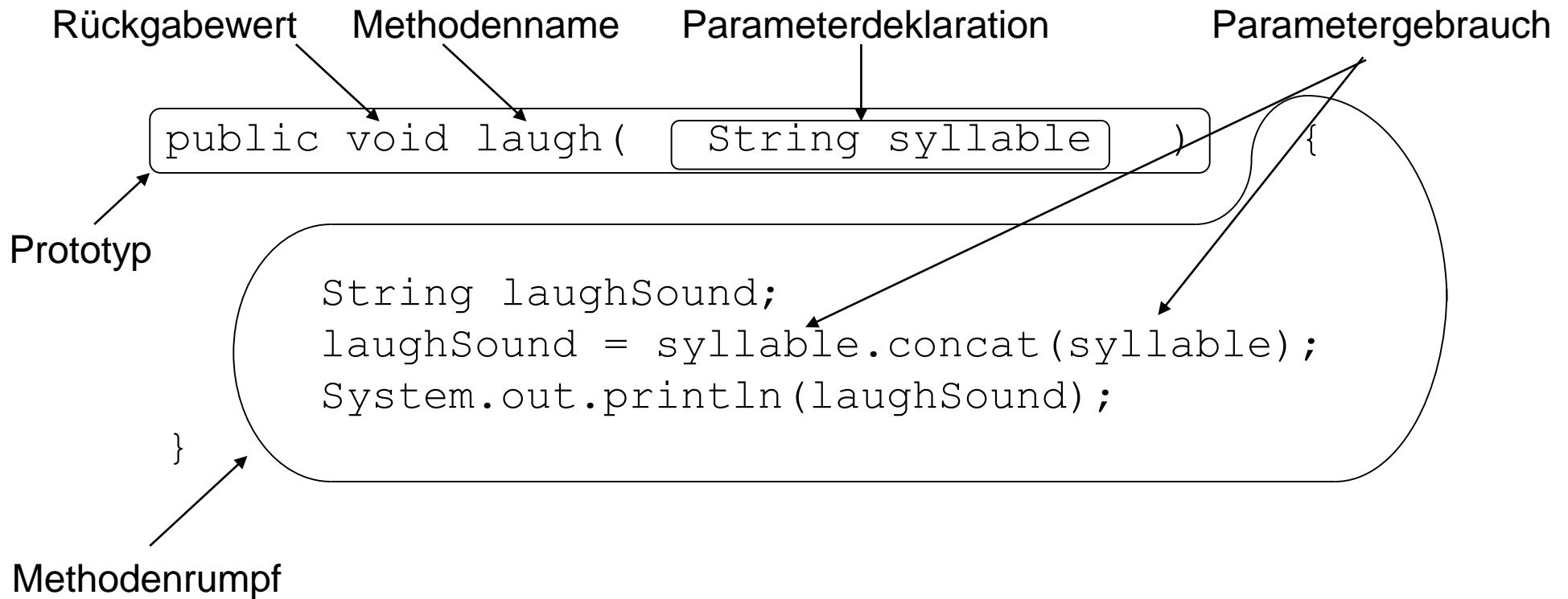
```
public void laugh(String syllable)
```

- Der Rumpf kann dann z.B. sein:

```
                                {  
    String laughSound;  
    laughSound = syllable.concat(syllable);  
    System.out.println(laughSound);  
}
```

- Wird diese Methode mit dem Argument `"ho"` aufgerufen, so gibt sie den Text `hoho` auf dem Bildschirm aus.

Struktur einer Methode mit Parametern



Eine erweiterte Laugher2-Klasse

```
class Laugher2 {  
    public Laugher2() {  
    }  
    public void laugh() {  
        System.out.println("haha");  
    }  
  
    public void laugh(String syllable) {  
        String laughSound;  
        laughSound = syllable.concat(syllable);  
        System.out.println(laughSound);  
    }  
}
```

Overloading

- Diese Definition von `Laugher2` stellt **zwei Methoden** mit dem **gleichen Namen** aber **unterschiedlichen Signaturen** zur Verfügung:

```
laugh()  
laugh(String syllable)
```

- In diesem Fall ist die Methode `laugh` **überladen** bzw. **overloaded**.
- Wenn wir `haha` ausgeben wollen, genügt der Aufruf

```
x.laugh();
```

- Um einen anderes Lachen (z.B. `yukyuk`) zu erzeugen, verwenden wir die zweite Methode:

```
x.laugh("yuk");
```

- Die **Methode ohne Parameter** repräsentiert das **Standardverhalten** und heißt daher **Default**.

Variante 3:

Veränderbare Standardsilbe

- Am Ende wollen wir auch die Möglichkeit haben, die Standardsilbe des `Laugher`-Objektes im **Konstruktor** anzugeben.
- Die gewünschte Anwendung ist:

```
Laugher3 x;  
x = new Laugher3("ho");  
x.laugh("heee");  
x.laugh();
```

- Um dies zu erreichen, erhält der Konstruktor jetzt ein `String`-Argument, so dass er folgende Signatur hat:

```
Laugher3(String s)
```

Instanzvariablen

- Wie können wir jetzt in der Methode `laugh()` auf dieses `String`-Objekt zugreifen?
- Die Lösung besteht darin, in der Klasse eine `String`-**Variable** zu definieren, die **außerhalb der Methoden der Klasse** steht.
- Eine solche Variable heißt **Instanzvariable**.
- Sie gehört zu dem **gesamten Objekt** und nicht zu einer einzelnen Methode.
- **Auf Instanzvariablen kann von jeder Methode aus zugegriffen werden.**
- **Instanzvariablen werden genauso deklariert wie andere Variablen.** In der Regel geht der Deklaration jedoch das **Schlüsselwort** `private` voraus.

Deklaration von und Zugriff auf Instanzvariablen

```
class Laugher3{  
    public Laugher3 ( String s) {  
        ...  
    }  
    public void laugh () {  
        ...  
    }  
    ...  
    private String defaultSyllable;  
}
```

Zu beachten :
defaultSyllable
kann in jedem
Rumpf einer Methode
benutzt werden

Instanzvariablen-Deklaration

Verwendung der Instanzvariable

- In unserem Beispiel ist die Aufgabe des Konstruktors, die mit dem Argument erhaltene Information in der Instanzvariablen `defaultSyllable` zu speichern:

```
public Laugher3(String s) {  
    defaultSyllable = s;  
}
```

- Anschließend kann die `laugh()`-Methode auf `defaultSyllable` zugreifen:

```
public void laugh() {  
    String laughSound;  
    laughSound =  
        defaultSyllable.concat(defaultSyllable);  
    System.out.println(laughSound);  
}
```

Die komplette Laugher3-Klasse

```
class Laugher3 {
    public Laugher3(String s) {
        defaultSyllable = s;
    }
    public void laugh() {
        String laughSound;
        laughSound = defaultSyllable.concat(defaultSyllable);
        System.out.println(laughSound);
    }
    public void laugh(String syllable) {
        String laughSound;
        laughSound = syllable.concat(syllable);
        System.out.println(laughSound);
    }
    private String defaultSyllable;
}
```


Verwendung einer Klassendefinition

1. Wir kompilieren `Laugher3.java`.
2. Wir schreiben ein Programm, das die `Laugher3`-Klasse benutzt:

```
class LaughALittle {  
    public static void main(String[] a) {  
        System.out.println("Live and laugh!!!");  
        Laugher3 x,y;  
        x = new Laugher3("yuk");  
        y = new Laugher3("harr");  
        x.laugh();  
        x.laugh("hee");  
        y.laugh();  
    }  
}
```

Der Klassenentwurfsprozess

Im vorangegangenen Beispiel haben wir mit einer einfachen Klasse begonnen und diese **schrittweise verfeinert**.

Für große Programmsysteme ist ein solcher Ansatz **nicht praktikabel**.

Stattdessen benötigt man ein **systematischeres Vorgehen**:

1. Festlegen des **Verhaltens** der Klasse.
2. Definition des **Interfaces** bzw. der **Schnittstellen** der Klasse, d.h. der Art der Verwendung. Dabei werden die **Prototypen** der Methoden festgelegt.
3. Entwicklung eines kleinen **Beispielprogramms**, das die Verwendung der Klasse demonstriert und gleichzeitig zum Testen verwendet werden kann.
4. Formulierung des **Skelettes** der Klasse, d.h. die **Standard-Klassendefinition** zusammen mit den **Prototypen**.

Festlegen des Verhaltens einer Klasse am Beispiel `InteractiveIO`

Für eine Klasse `InteractiveIO` wünschen wir das folgende Verhalten:

- Ausgeben einer Meldung auf dem Monitor (mit der Zusicherung, dass sie unmittelbar angezeigt wird).
- Von dem Benutzer einen `String` vom Keyboard einlesen.

Außerdem sollte der Programmierer die Möglichkeit haben, `InteractiveIO`-Objekte zu erzeugen, ohne `System.in` oder `System.out` verwenden zu müssen.

Interfaces und Prototypen (1)

Die **Schnittstelle** einer Klasse beschreibt die Art, in der die Objekte dieser Klasse verwendet werden können.

Für unsere `InteractiveIO`-Klasse wären dies:

- Deklaration einer `InteractiveIO`-Referenzvariablen:

```
InteractiveIO interIO;
```

- Erzeugen eines `InteractiveIO`-Objektes:

```
interIO = new InteractiveIO();
```

In diesem Beispiel benötigt der Konstruktor kein Argument.

Interfaces und Prototypen (2)

- Senden einer Nachricht zur Ausgabe eines `String`-Objektes an ein `InteractiveIO`-Objekt.

```
interIO.write("Please answer each question");
```

Resultierender Prototyp:

```
public void write(String s)
```

- Ausgeben eines Prompts auf dem Monitor und Einlesen eines `String`-Objektes von der Tastatur. Dabei soll eine Referenz auf den `String` zurückgegeben werden:

```
String s;  
s = interIO.promptAndRead("What is your first name? ");
```

Resultierender Prototyp:

```
public String promptAndRead(String s)
```

Ein Beispielprogramm, das InteractiveIO verwendet

Aufgaben des Beispielprogramms:

1. Demonstrieren, wie die neue Klasse verwendet wird.
2. Prüfen, ob die Prototypen so sinnvoll sind.

```
class TryInteractiveIO {  
    public static void main(String[] arg) throws Exception {  
        InteractiveIO interIO;  
        String line;  
  
        interIO = new InteractiveIO();  
        line = interIO.promptAndRead("Please type in a word: ");  
        interIO.write(line);  
    }  
}
```

Das Skelett von InteractiveIO

Gegenüber der kompletten Klassendefinition fehlt dem **Skelett** der Code, der die Methoden realisiert:

```
// Easy to use class for communicating with the user
class InteractiveIO {
    public InteractiveIO() {

        //Write s to the monitor
        public void write(String s) {

            //Write s to the monitor, read a string from the
            keyboard,
            //and return a reference to it.
            public String promptAndRead(String s) throws Exception {

                }
            }
        }
    }
}
```

Implementierung von InteractiveIO (1)

Die **Implementierung** einer Klasse besteht aus dem **Rumpf der Methoden** sowie den **Instanzvariablen**.

Dabei spielt die **Reihenfolge**, in der Methoden (weiter-) entwickelt werden, **keine Rolle**.

Der Konstruktor tut nichts:

```
public InteractiveIO() {  
}
```

Als nächstes definieren wir die Methode `write`:

```
public void write(String s) {  
    System.out.println(s);  
    System.out.flush();  
}
```


Implementierung von InteractiveIO (2)

Schließlich implementieren wir `promptAndRead`.

Um in einer Methode den **Return-Wert** zurückzugeben, verwenden wir das **Return-Statement**:

```
return Wert;
```

Dies ergibt:

```
public String promptAndRead(String s) throws Exception {  
    System.out.println(s);  
    System.out.flush();  
  
    BufferedReader br;  
    br = new BufferedReader(new InputStreamReader(System.in));  
  
    String line;  
    line = br.readLine();  
    return line;  
}
```

Die komplette Klasse InteractiveIO

```
import java.io.*;

class InteractiveIO {
    public InteractiveIO() {
    }
    public void write(String s) {
        System.out.println(s);
        System.out.flush();
    }
    public String promptAndRead(String s) throws Exception {
        System.out.println(s);
        System.out.flush();
        BufferedReader br;
        br = new BufferedReader(new InputStreamReader(System.in));
        String line;
        line = br.readLine();
        return line;
    }
}
```

Siehe: examples/InteractiveIO_1.java

Verbessern der Implementierung von InteractiveIO

- Häufig ist eine **erste Implementierung einer Klasse noch nicht optimal**.
- Nachteilhaft an unserer Implementierung ist, dass bei jedem Einlesen einer Zeile ein `BufferedReader` und ein `InputStreamReader` erzeugt wird.
- Es wäre viel günstiger, diese Objekte einmal zu erzeugen und anschließend wiederzuverwenden.
- Die entsprechenden Variablen können wir als **Instanzvariablen deklarieren** und die Erzeugung der Objekte können wir **in den Konstruktor verschieben**.

Prinzip der Verbesserung von InteractiveIO

```
class InteractiveIO{  
    public InteractiveIO() {  
        br = new BufferedReader(  
            new InputStreamReader(System.in));  
    }  
    ...  
    public String promptAndRead(String s) throws Exception {  
        System.out.println(s);  
        System.out.flush();  
  
        String line;  
        ...  
    }  
    private BufferedReader br;  
}
```

← Der Konstruktor

← Jetzt Instanzvariable

Weitere Vereinfachungen

1. Wir können die von `readLine` erzeugte Referenz auch direkt zurückgeben:

```
return br.readLine();
```

2. Beide Methoden `write` und `promptAndRead` geben etwas auf dem Monitor aus und verwenden `println` und `flush`. Dies kann in einer Methode zusammengefasst werden:

```
private void writeAndFlush(String s) {  
    System.out.println(s);  
    System.out.flush();  
}
```

Das Schlüsselwort `this`

Mit der Methode `writeAndFlush` können wir sowohl in `write` als auch in `promptAndRead` die entsprechende **Code-Fragmente** ersetzen.

Problem: Methoden werden aufgerufen, indem Nachrichten an Objekte gesendet werden. Aber an welches Objekt können wir aus der Methode `write` eine Nachricht `writeAndFlush` senden?

Antwort: Es ist **dasselbe Objekt**.

Java stellt das **Schlüsselwort `this`** zur Verfügung, damit eine **Methode das Objekt, zu dem sie gehört, referenzieren** kann:

```
this.writeAndFlush(s) ;
```

Die komplette, verbesserte Klasse InteractiveIO

```
import java.io.*;

class InteractiveIO {
    public InteractiveIO() {
        br = new BufferedReader(new InputStreamReader(System.in));
    }
    public void write(String s) {
        this.writeAndFlush(s);
    }
    public String promptAndRead(String s) throws Exception {
        this.writeAndFlush(s);
        return br.readLine();
    }
    private void writeAndFlush(String s) {
        System.out.println(s);
        System.out.flush();
    }
    private BufferedReader br;
}
```

Siehe: [examples/InteractiveIO_2.java](#)

Deklarationen und das `return`-Statement

Reihenfolge der Vereinbarungen: Die Reihenfolge von Variablen- und Methodendeklarationen in einer Klasse ist beliebig. Es ist jedoch eine gängige Konvention, erst die Methoden zu deklarieren und dann die Variablen.

Das `return`-Statement: Der Sender einer Nachricht kann nicht fortfahren, bis die entsprechende Methode beendet ist. In Java geschieht die Rückkehr zum Sender durch ein `return`-Statement oder, sofern die Methode den Typ `void` hat, am Ende der Methode. Allerdings können auch `void`-Methoden mit `return` beendet werden:

```
private void writeAndFlush(String s) {  
    System.out.println(s);  
    System.out.flush();  
    return;  
}
```


Zugriffskontrolle

- Eine **Klassendefinition** besteht aus **Methoden** und **Instanzvariablen**.
- Der Programmierer kann einen unterschiedlichen **Zugriff** auf **Methoden** oder **Variablen** gestatten, indem er die Schlüsselwörter `public` oder `private` verwendet.
- Als `public` deklarierte Methoden können **von außen aufgerufen** werden. Als `private` vereinbarte Methoden sind jedoch **nur innerhalb der Klasse bekannt**.
- Gleiches gilt für **Instanzvariablen**.

Variablen und ihre Lebensdauer

Wir haben **drei verschiedene Arten von Variablen** kennen gelernt:

1. als **Parameter im Kopf der Definition einer Methode**,
2. als **lokale Variable** definiert **innerhalb des Rumpfes einer Methode** und
3. als **Instanzvariablen in der Klassendefinition**.

Variablen als Parameter

Variablen, die **Parameter einer Methode** sind, werden **beim Aufruf der Methode automatisch erzeugt** und sind **innerhalb der Methode bekannt**. Ist die Methode beendet, kann auf sie nicht mehr zugegriffen werden. Ihre Lebenszeit entspricht der der Methode.

Parameter erhalten ihren **initialen Wert vom Sender** der Nachricht und die **Argumente des Aufrufs müssen exakt mit den Argumenten der Methode übereinstimmen**.

Für `void f(String s1, PrintStream p)` ist der Aufruf

```
f("hello", System.out)
```

zulässig. Die folgenden Aufrufe hingegen sind alle unzulässig:

```
f("hello")
```

```
f("hello", "goodbye")
```

```
f("hello", System.out, "bye")
```

Lokale Variablen

- **Lokale Variablen** die **in Methoden definiert** werden.
- Sie haben die gleiche Lebenszeit wie Parameter.
- Sie werden beim Aufruf einer Methode erzeugt und beim Verlassen der Methode gelöscht.
- Lokale Variablen müssen innerhalb der Methode initialisiert werden.
- Parameter und Variablen sind außerhalb der Methode, in der sie definiert werden, nicht sichtbar, d.h. auf sie kann von anderen Methoden nicht zugegriffen werden.
- Wird in verschiedenen Methoden derselbe Bezeichner für lokale Variablen verwendet, so handelt es sich um jeweils verschiedene lokale Variablen.

Beispiel

```
String getGenre() {  
    String s = "classic rock/".concat(getFormat());  
    return s;  
}
```

```
String getFormat() {  
    String s = "no commercials";  
    return s;  
}
```

Die Wertzuweisung an `s` in `getFormat` hat keinerlei Effekt auf die lokale Referenzvariable `s` in `getGenre`.

Rückgabewert von `getGenre` ist somit eine Referenz auf

```
"classic rock/no commercials"
```

Instanzvariablen

- **Instanzvariablen** haben dieselbe Gültigkeitsdauer wie das Objekt, zu dem sie gehören.
- Auf Instanzvariablen kann **von jeder Methode** eines Objektes aus **zugegriffen werden**.
- Der **Zugriff von außen** wird, ebenso wie bei den Methoden, durch die Schlüsselworte `public` und `private` geregelt.

Lebensdauer von Objekten

- **Objekte** können in den Methoden einer Klasse durch Verwendung von `new` oder durch den Aufruf anderer Methoden **neu erzeugt** werden.
- **Java löscht nicht referenzierte Objekte automatisch.**

```
public void m2() {  
    String s;  
    s = new String("Hello world!");  
    s = new String("Welcome to Java!");  
    ...  
}
```

- Nach der zweiten Wertzuweisung gibt es **keine Referenz** auf `"Hello world!"` mehr.
- **Konsequenz:** Objekte bleiben so lange erhalten, wie es eine Referenz auf sie gibt.

Das Schlüsselwort `this`

Mit dem Schlüsselwort `this` kann man innerhalb von Methoden einer Klasse das **Objekt selbst referenzieren**. Damit kann man

1. dem **Objekt selbst eine Nachricht schicken** oder
2. bei **Mehrdeutigkeiten** auf das Objekt selbst referenzieren.

```
class ... {  
    ...  
    public void m1() {  
        String s;  
        ...  
    }  
    ...  
    private String s;  
}
```

Innerhalb der Methode `m1` ist `s` eine **lokale Variable**. Hingegen ist `this.s` die **Instanzvariable**.

Der Konstruktor

- Der **Konstruktor** ist immer die erste Methode, die aufgerufen wird.
- Die Aufgabe eines Konstruktors ist daher, dafür zu sorgen, dass das entsprechende Objekt „sein Leben“ mit den **richtigen Werten beginnt**.
- Insbesondere soll der **Konstruktor** die **notwendigen Initialisierungen der Instanzvariablen vornehmen**.

Zusammenfassung (1)

- Eine **Klassendefinition** setzt sich zusammen aus der Formulierung der **Methoden** und der Deklaration der **Instanzvariablen**.
- Methoden und Instanzvariablen können als `public` oder `private` deklariert werden, um den **Zugriff** von außen festzulegen.
- Es gibt **drei Arten von Variablen**: **Instanzvariablen**, **lokale Variablen** und **Parameter**.
- **Lokale Variablen** sind Variablen, die **in Methoden deklariert** werden.

Zusammenfassung (2)

- **Parameter** werden **im Prototyp** einer Methode **definiert** und **beim Aufruf** durch die Wertübergabe **initialisiert**.
- **Instanzvariablen** werden **außerhalb der Methoden** aber **innerhalb der Klasse** definiert.
- **Instanzvariablen speichern Informationen, die über verschiedene Methodenaufrufe hinweg benötigt werden.**

Informatik I

Processing Numbers

Wolfram Burgard

Motivation

- Computer bzw. Rechenmaschinen wurden ursprünglich gebaut, um schnell und zuverlässig mit Zahlen zu rechnen.
- Erste Anwendungen von Rechenmaschinen und Computern waren die Berechnung von Zahlentabellen, Codes, Buchhaltungen, ...
- Auch heute spielen numerische Berechnungen immer noch eine bedeutende Rolle.
- Nicht nur in Buchhaltungen, graphischen Oberflächen (Kreise, Ellipsen, ...) sondern auch bei der Interpretation von Daten (z.B. Bildverarbeitung, Robotik, ...) wird üblicherweise mit Zahlen gerechnet.
- Auch Java bietet Möglichkeiten, Zahlen zu repräsentieren und arithmetische Berechnungen durchzuführen.

Primitive Datentypen

- Einer der grundlegende Datentypen von Computern sind Zahlen.
- Anstatt Klassen für die Manipulation von Zahlen zur Verfügung zu stellen, bietet Java einen **direkten Zugriff auf Zahlen**.
- Verschiedene Typen von Zahlen (ganze Zahlen, ...) werden in Java auch direkt, d.h. unter direkter Verwendung der zugrunde liegenden Hardware realisiert ohne den Umweg über Klassendefinitionen zu gehen.
- Dies hat den Vorteil, dass numerische Berechnungen besonders effizient ausgeführt werden können.

Operatoren versus Methoden

- Allerdings führt der **Verzicht auf Klassen für Zahlen** dazu, dass **Berechnungen nicht mithilfe von Nachrichten** ausgeführt werden, die an Objekte gesendet werden, sondern mithilfe so genannter **Operatoren**.
- Darüber hinaus ist

`x / (y + 1)`

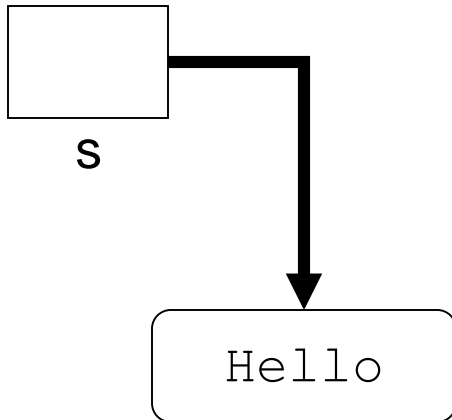
leichter lesbar als

`x.divide(y.add(1))`

Variablen versus Referenzvariablen

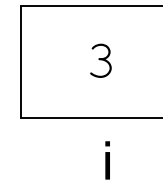
- **Referenzvariablen haben als Wert Referenzen** bzw. Bezüge auf Objekte.
- **Variablen hingegen enthalten Werte einfacher Datentypen** und werden nicht mit Objekten „assoziiert“.
- einer dieser Datentypen ist **int**, der ganze Zahlen repräsentiert.

```
String s = new String("Hello");
```



Objekt, das "Hello" repräsentiert

```
int i = 3;
```



Unterschiede zwischen Variablen und Referenzvariablen

	Referenzvariable	Einfache Variable
Definiert durch	Klassendefinition	Sprache
Wert erzeugt durch	new	System
Wert initialisiert durch	Konstruktor	System
Variable initialisiert durch	Zuweisung einer Referenz	Zuweisung eines Wertes
Variable enthält	Referenz auf ein Objekt	primitiven Wert
Verwendet zusammen mit	Methoden	Operatoren
Nachrichtenempfänger	ja	nein

Grundlegende Arithmetische Operatoren

Einige der Operatoren, die in Java im Zusammenhang mit ganzen Zahlen (`int`) benutzt werden können, sind.

+ Addition

Ergebnis ist die Summe der beiden Operanden: $5 + 3 = 8$

- Subtraktion

Ergebnis ist die Differenz der beiden Operanden $5 - 2 = 3$

*** Multiplikation**

Ergebnis ist das Produkt der beiden Operanden $5 * 3 = 15$

/ Division

Ganzzahlige Division ohne Rest: $5 / 3 = 1$

% Rest

Ergebnis ist der Rest bei ganzzahliger Division: $5 \% 3 = 2$

Operatoren, Operanden und Ausdrücke

- **Operatoren** korrespondieren zu Aktionen, die Werte berechnen.
- Die Objekte, auf die Operatoren angewandt werden, heißen **Operanden**.
- Operatoren zusammen mit ihren Operanden werden **Ausdrücke** genannt.
- In dem Ausdruck x / y sind x und y die Operanden und $/$ der Operator.
- Da **Ausdrücke** wiederum **Werte repräsentieren**, können Ausdrücke auch als Operanden verwendet werden.

Für die Integer-Variablen x , y und z lassen sich folgende **Ausdrücke** bilden:

$$x + y$$

$$z / x$$

$$(x - y) * (x + y)$$

Literale

- Innerhalb von **Ausdrücken** dürfen auch konkrete (Zahlen)-Werte verwendet werden.
- **Zahlenwerte, die von der Programmiersprache vorgegeben werden**, wie z.B. -1 oder 2 , heißen **Literale**.

Damit sind auch folgende Ausdrücke zulässig:

$$2 * x + y$$
$$75 / x$$
$$33 / 5 + y$$

Präzedenzregeln

- Sofern in einem Ausdruck mehr als ein Operator vorkommt, gibt es **Mehrdeutigkeiten**.
- Je nachdem, wie der Ausdruck ausgewertet wird, erhält man unterschiedliche Ergebnisse.
- Beispielsweise kann $4 * 3 + 2$ als 14 interpretiert werden, wenn man zunächst 4 mit 3 multipliziert und anschließend 2 addiert, oder als 20, wenn man zuerst 3 und 2 addiert und das Ergebnis mit 4 multipliziert.
- Java verwendet so genannte „**Präzedenzregeln**“, um solche Mehrdeutigkeiten aufzulösen.
- Dabei haben $*$, $/$ und $\%$ eine **höhere Präzedenz als** die **zweistelligen Operatoren** $+$ und $-$.
- Die **einstelligen Operatoren** $+$ und $-$ (**Vorzeichen**) wiederum haben höhere Präzedenz als $*$, $/$ und $\%$.

Präcedenzregeln und Klammern

Der Ausdruck

$$4 * 3 * -2 + 2 * -4$$

ist somit äquivalent zu

$$((4 * 3) * (-2)) + (2 * (-4))$$

Ebenso wie in der Mathematik kann man **runde Klammern** verwenden, um **Präcedenzregeln zu überschreiben**:

$$\begin{array}{l} (4 * 3) + 2 \\ 4 * (3 + 2) \end{array}$$

Wertzuweisungen und zusammengesetzte Wertzuweisungen

- Ausdrücke (wie die oben verwendeten) können auf der rechten Seite von Wertzuweisungen verwendet werden:

$$x = y + 4;$$
$$y = 2 * x + 5;$$

- Verschiedene Wertzuweisungen tauchen jedoch sehr häufig auf, wie z.B.

$$x = x + y;$$
$$y = 2 * y;$$

Für diese Form der Wertzuweisungen stellt Java **zusammengesetzte Wertzuweisungen** zur Verfügung:

$$x += y;$$

entspricht

$$x = x + y;$$
$$y *= 2;$$

entspricht

$$y = y * 2;$$

Inkrement und Dekrement

- Die häufigsten arithmetischen Operationen sind das Addieren und das Subtrahieren von 1.
- Auch hierfür stellt Java spezielle Operatoren zur Verfügung:

`x++; ++x;`

`y--; --y;`

- Die oberen zwei Operatoren heißt **Inkrement-Operatoren**. Die unteren zwei werden **Dekrement-Operatoren** genannt.
- Diese Statements stehen für eine Wertzuweisung, durch welche der Wert der entsprechenden Variable um eins erhöht bzw. erniedrigt wird.
- Dementsprechend dürfen die Argumente dieser Operatoren weder Literale noch zusammengesetzte Ausdrücke sein.

Methoden für Integers

- Die Menge der Operatoren ist auf die Grundrechenarten eingeschränkt.
- Häufig benötigt man jedoch **weitere Funktionen**.
- Da die eingebauten Datentypen wie `int` eingeführt wurden, um bei Berechnungen den Zusatzaufwand einer Objektorientierung zu vermeiden, stellt sich nun die **Frage, wie solche Funktionen realisiert werden können**.
- Da wir **keine Objekte** mehr haben, **denen wir eine Nachricht schicken können**, müssen wir uns entsprechende Alternativen suchen.

Methoden für Integers

- In Java besteht die Lösung darin, dass solche **Methoden in den jeweiligen Klassen realisiert werden**.
- Die entsprechenden **Nachrichten werden dann nicht an ein Objekt sondern an die entsprechende Klasse gesendet**.
- Beispielsweise werden auch für Integer-Objekte einige Methoden in den vordefinierten Klasse `Integer` und `Math` zur Verfügung gestellt.
- Eine dieser Methoden ist z.B. `Math.abs`:

```
int i = -2;
```

```
int j = Math.abs(i);
```

Das Schlüsselwort `static`

- **Methoden und Variablen**, die **nicht an Instanzen einer Klasse gebunden** sind, sollten das Schlüsselwort `static` tragen.
- **Statische Methoden/Variablen** „gehören“ somit **zur Klasse** und nicht zu Instanzen einer Klasse (den Objekten).
- **Statische Methoden** haben daher **keinen Zugriff** auf die **Instanzvariablen** der Klasse.
- Statische Methoden haben **keinen Zugriff** auf **nicht-statische Methoden** der Klasse.

Anwendung des Schlüsselworts `static`

- Beispiel einer statischen Methode:

```
class StaticTest {  
    StaticTest() {}  
    public static int sum(int a, int b) {  
        return a+b;  
    }  
}
```

- Da eine statische Methode nicht zu einem Objekt gehört, verwendet man den Klassennamen als Empfänger:

```
StaticTest.sum(1,2); // yields 1+2=3
```

- Eine typische Anwendung statischer Methoden und Variablen sind die mathematische Funktionen `Math.cos()`, `Math.sin()` sowie die Konstante `Math.PI`.

Auswertung von Ausdrücken

1. **Ausdrücke werden von links nach rechts unter Berücksichtigung der Präzedenzregeln und der Klammerung ausgewertet.**
2. Bei **Operatoren mit gleicher Präzedenz** wird **von links nach rechts** vorgegangen.
3. Dabei werden die **Variablen und Methodenaufrufe, sobald sie an die Reihe kommen, durch ihre jeweils aktuellen Werte ersetzt.**

Beispiele für die Ausdrucksauswertung

Gegeben:

```
int p = 2, q = 4, r = 4, w = 6, x = 2, y = 1;
```

Dies ergibt:

```
p * r % q + w / x - y  
2 * 4 % 4 + 6 / 2 - 1          ----> 2
```

```
p * x * x + w * x + -q  
2 * 2 * 2 + 6 * 2 + -4          ----> 16
```

```
(p + q * 2) + ((p - 2) * r - w)  
(2 + 4 * 2) + ((2 - 2) * 4 - 6)  ----> 4
```

Zuweisungen und Inkrementoperatoren in Ausdrücken

- Sowohl die Wertzuweisung `=` als auch die Inkrementoperatoren `++` und `--` stellen **Operatoren** dar.
- Sie dürfen daher auch in Wertzuweisungen vorkommen.
- Der Ausdruck `x = y` hat als Wert den Wert von `y`.
- Hat `x` den Wert 3, liefert `++x` als Ergebnis den Wert 4. Dabei wird `x` von 3 auf 4 erhöht.
- `x++` liefert in derselben Situation den Wert 3. Danach wird `x` um 1 erhöht.
- Zulässig sind daher

`x = y = z = 0; x = y = z++; x = z++ + --z;`

Empfehlung: Keine Zuweisungen und Operatoren mit Seiteneffekten in Ausdrücken verwenden!

Einlesen von Zahlen von der Tastatur

- Um Zahlen von der Tastatur einzulesen, benötigen wir entsprechende Methoden.
- In Java wird das durch die **Komposition von zwei Methoden** erreicht.
- Die erste liest ein **String-Objekt aus dem Eingabestrom**.
- Die zweite Methode **wandelt die Zeichen dieses String-Objektes in eine Zahl um**:

```
String s = br.readLine();  
int i = Integer.parseInt(s);
```

- Kompakter geht es mit:

```
int i = Integer.parseInt(br.readLine());
```


Mögliche Fehler

- Damit das Einlesen einer Zahl **erfolgreich** ist, muss sich die eingelesene Zeile tatsächlich auch in **eine Zahl umwandeln lassen**.
- Folgende Eingaben sind zulässig:

2

75

-1

- Bei folgenden Eingaben hingegen wird ein Fehler auftreten:

Hello

75 40

12o

Der Datentyp `long` für große ganze Zahlen

- Der Typ `int` modelliert ganze Zahlen in dem Bereich `[-2147483648, 2147483647]`
- Leider reicht dieser Wertebereich für viele Anwendungen nicht aus: Erdbevölkerung, Staatsschulden, Entfernungen im Weltall etc.
- Java stellt daher auch den Typ `long` mit dem Wertebereich `[-9223372036854775808, 9223372036854775807]` zur Verfügung, der für fast alle Anwendungen im Bereich Administration und Handel ausreicht.
- Für den Datentyp `long` gelten die gleichen Operatoren wie für `int`.
- **Long-Literale** werden durch ein abschließendes `L` gekennzeichnet.

```
long x = 2000L, y = 1000L;  
y *= x;  
x += 1500L;
```

Warum `int` und `long` und nicht nur `long`?

- Variablen vom Typ `int` benötigen nur vier Byte=32 Bit, während solche vom Typ `long` acht Byte = 64 Bit benötigen. Wenn also ein Programm sehr viele ganze Zahlen verwendet, verbraucht man bei Verwendung von `ints` nur die Hälfte an Speicherplatz.
- In der Praxis muss man die Anforderungen an die Genauigkeit sehr genau untersuchen und kann ggf. auf die speicherplatzsparenden `ints` zurückgreifen.

Gleichzeitige Verwendung mehrerer Typen: Casting

- Java erlaubt die Zuweisung eines Wertes vom Typ `int` an eine Variable vom Typ `long`.
- Dabei geht offensichtlich keine Information verloren.
- Umgekehrt ist das jedoch nicht der Fall, weil der zugewiesene Wert außerhalb des Bereichs von `int` liegen kann.
- Wenn man einer Variable vom Typ `int` einen Ausdruck vom Typ `long` zuordnen will und man sicher ist, dass keine **Bereichsüberschreitung** stattfinden kann, muss man eine spezielle Notation verwenden, die **Casting** genannt wird.
- Dabei stellt man dem Ausdruck den Typ, in den sein Wert **konvertiert** werden soll, in Klammern voraus.

Die folgenden Wertzuweisungen sind daher zulässig:

```
long x = 98;  
int i = (int) x;           // casting
```

Modellieren von Messdaten

- **Integer** sind Zahlen, die üblicherweise zum **Zählen** verwendet werden.
- **Integer** sollten daher immer dann verwendet werden, wenn der **Wertebereich einer Variablen in den ganzen Zahlen** liegt (Anzahl Studenten, die immatrikuliert sind, Anzahl der Kinder, ...).
- Insbesondere bei der Verarbeitung von **Messdaten** erhält man jedoch oft Werte, die **keine ganzen Zahlen** sind (540.3 Meter, 10.97 Sekunden, ...).
- Deshalb benötigt man in einer Programmiersprache auch **Werte mit Nachkommastellen**:

12.34

3.1415926

1.414

Fließkommazahlen

- In der Welt der Computer werden Messwerte üblicherweise durch **Fließkommazahlen** repräsentiert.
- Hierbei handelt es sich um Zahlen der Form

$$3.1479 \times 10^{15}$$

- Diese Zahl würde in Java repräsentiert durch

3.1479E15f

- Dabei sind sowohl der **Vorkommaanteil**, der **Nachkommaanteil** und der **Exponent** in der **Anzahl der Stellen begrenzt**.
- **Fließkommazahlen repräsentieren eine endliche Teilmenge der rationalen Zahlen.**

Die Datentypen `float` und `double`

- Java stellt mit `float` und `double` zwei **elementare Datentypen** mit unterschiedlicher Genauigkeit für die Repräsentation von **Fließkommazahlen** zur Verfügung.
- Der Typ `float` modelliert Fließkommazahlen mit **ungefähr siebenstelliger Genauigkeit**. Der **Absolutbetrag** der Werte kann entweder 0 sein oder im Bereich `[1.4E-45f, 3.4028235E38f]` liegen.
- Demgegenüber hat der Typ `double` eine **ungefähr fünfzehnstellige Genauigkeit**. Der **Absolutbetrag** der Werte kann entweder 0 sein oder im Bereich `[4.9E-324, 1.7976931348623157E308]` liegen.

Vergleich der Typen `float` und `int`

- Variablen vom Typ `float` benötigen ebenso wie Variablen vom Typ `int` lediglich 4 Byte = 32 Bit.
- Variablen vom Typ `float` können **größere Werte** repräsentieren als Variablen vom Typ `int`.
- Allerdings haben `floats` nur eine **beschränkte Genauigkeit**.

Beispiel:

```
float f1 = 1234567089f;  
System.out.println(f1);
```

liefert als Ausgabe

```
1.23456704E9.
```


Fließkommazahlen und Rundungsfehler

- **Fließkommazahlen** stellen nur eine **begrenzte Genauigkeit** zur Verfügung.
- Ein typisches Beispiel für mögliche **Rechenfehler** ist:

```
float x = 0.0644456f;  
float y = 0.032754f;  
float z = x * y;  
System.out.println(z);
```

- Ausgabe dieses Programmstücks ist 0.0021108512.
- **Korrekt wäre** 0.0021108511824.

Verwendung von `float` oder `double`

- Variablen vom Typ `float` und `double` werden ähnlich verwendet wie Variablen vom Typ `int`.
- Mit folgendem Programmstück wird die Fläche eines Kreises berechnen mit Radius `12.0` berechnet:

```
double area, radius;  
radius = 12.0;  
area = 3.14159*radius*radius;
```

Einlesen von Werten vom Typ `float` und `double`

- Das Einlesen von Werten für `double/float` ist so einfach wie für `int`.
- Java stellt ein **Klasse `Double/Float`** zur Verfügung, die es erlaubt, einen `double` Wert aus einem `String-Objekt` zu berechnen.

```
double d = Double.parseDouble(br.readLine());
```

Wann soll man `float` oder `double` verwenden?

- **Fließkommazahlen** werden in der Regel verwendet, wenn man **Zahlen mit Nachkommaanteil** benötigt.
- Die **Genauigkeit von `double` ist für viele Anwendungen hinreichend**.
- Allerdings gibt es **Anwendungen, für welche die Genauigkeit von `double` nicht ausreicht**.
- Ein typisches Beispiel ist das Lösen großer Gleichungssysteme.
- Probleme tauchen aber auch bei Berechnungen im Finanzbereich auf, bei denen Rundungsfehler bis zur zweiten Nachkommastelle ausgeschlossen werden müssen.

Gemischte Arithmetik

- Dieselben Gesetze, die für die **Konvertierung** zwischen `int` und `long` gelten, finden auch für `float` und `double` Anwendung.
- Allerdings kann man auch Integer-Variablen Werte von Fließkommazahlen zuordnen und umgekehrt.
- Nur wenn es mit **keinem Informationsverlust** verbunden ist, kann eine **Zuweisung direkt** erfolgen.
- Andernfalls muss man das **Casting** verwenden.

```
double x = 4.5;
int i = (int) x;
x = i;
```

- Dabei wird bei der **Konvertierung von Fließkommazahlen nach Integer-Zahlen stets der Nachkommaanteil abgeschnitten**

Zusammenfassung

- Java stellt verschiedene elementare Datentypen für das „**Verarbeiten von Zahlen**“ bereit.
- Die **Integer-Datentypen** repräsentieren ganze **Zahlen**.
- Die Datentypen `float` und `double` repräsentieren **Fließkommazahlen**.
- **Fließkommazahlen** sind eine Teilmenge der rationalen und reellen Zahlen.
- Die **Werte dieser Datentypen** werden durch **Literale** beschrieben.
- Für die Konvertierung zwischen Datentypen verwendet man das **Casting**.
- Berechnungen mit Daten vom Typ `double` und `float` können **Rundungsfehler** produzieren.
- Dadurch entstehen häufig **falsche Ausgaben und Ergebnisse**.
- **You have been warned!**

Einführung in die Informatik

Control Structures and Iterators

if, while, for und Iteratoren

Wolfram Burgard

Motivation

- Bisher bestanden die Rümpfe unserer Methoden aus einzelnen Statements, z.B. Wertzuweisungen oder Methodenaufrufen.
- Es gibt bisher keine Möglichkeit, Statements nur in Abhängigkeit bestimmter Umstände auszuführen.
- Durch **bedingte Anweisungen und Schleifen** können wir **flexiblere Methoden** schreiben und **deutlich mächtigere Modelle** entwickeln.

Das if-Statement

- Java stellt mit dem **if-Statement** eine Form der **bedingten Anweisung** zur Verfügung.
- Mit Hilfe des **if-Statements** können wir eine **Bedingung** testen und, je nach Ausgang des Tests, eine von zwei Anweisungen durchführen.

```
if (condition)
    statement1
else
    statement2
```

Das if-Statement

- Beispiel:

```
if (x == 2)
    result = 4;
else
    result = 5 * x;
```

- Zeile 1 enthält den Test, den wir ausführen.
- Zeile 2 enthält das Statement, das bei erfolgreichem Test ausgeführt wird.
- Zeile 3 enthält das Schlüsselwort `else` und läutet den Teil ein, der ausgeführt wird, wenn der Test fehlschlägt.
- Zeile 4 enthält das Statement, welches bei negativem Ausgang des Tests ausgeführt wird.

Mehrere Anweisungen in if-Statements

- In der Grundversion des **if-Statements** können nur einzelne Statements im **then-Teil** und **else-Teil** verwendet werden.
- Sollen **mehrere Statements** ausgeführt werden, muss man diese zu einem **Block zusammenfassen**, indem man sie in geschweifte Klammern ({ und }) einschließt.

```
if (x>y) {  
    System.out.print(x);  
    System.out.print(" is greater than ");  
    System.out.println(y);  
}  
else {  
    System.out.print(x);  
    System.out.print(" is not greater than ");  
    System.out.println(y);  
}
```

zusammen-
gesetzte
Statements

Multiple if-Statements

- Java erlaubt es auch das **else-Statement** wegzulassen, d.h. es wird kein Code ausgeführt, wenn die Bedingung falsch ist.
- Verschiedene **if-Statements** können auch **geschachtelt** werden

```
if (X > 2)
    if (X < 5)
        System.out.println("X ist größer als 2 und kleiner als 5");
    else
        System.out.println("X ist größer gleich 5");
```

- Des Weiteren sind **kaskadierte if-Statements** möglich

```
if (X > 2)
    System.out.println("X ist größer als 2");
else if (X < 0)
    System.out.println("X ist kleiner als 0");
else
    System.out.println("X ist größer gleich 0 und kleiner gleich 2");
```

Zu welchem `if` gehört ein `else`?

- Ein `else` gehört immer zu dem letzten `if`, für das noch ein `else` fehlt.
- Unser Beispiel entspricht daher:

```
if (X > 2) {  
    if (X < 5)  
        System.out.println("X ist größer als 2 und kleiner als 5");  
    else  
        System.out.println("X ist größer als 5");  
}
```

- Hierbei sollte die **Einrückung der Statements** die **Zuordnung der Statements widerspiegeln**.

Bedingungen in if-Statements

- Die **Bedingung** eines **if-Statements** muss ein **Ausdruck** sein, der entweder wahr oder falsch ist.
- Im Moment schränken wir uns auf Vergleiche zwischen Zahlwerten ein.
- Java stellt folgende **Operatoren für den Vergleich von Zahlen** zur Verfügung:

Operator	Bedeutung
<	kleiner
>	größer
==	gleich
<=	kleiner gleich
>=	größer gleich
!=	ungleich

Der Typ `boolean`

- Für **logischen Werte** **wahr** und **falsch** gibt es in Java einen primitiven Datentyp `boolean`
- Die **möglichen Werte** von Variablen dieses Typs sind `true` und `false`.
- Wie Integer-Variablen kann man auch Variablen vom Typ `boolean` vereinbaren.
- Diesen Variablen können **Werte logischer Ausdrücke** zugewiesen werden.

Anwendung vom Typ boolean

Typische Situation:

```
boolean hasOvertime;  
if (hours > 40)  
    hasOvertime = true;  
else  
    hasOvertime = false;  
...  
if (hasOvertime)    // same as: if (hasOvertime == true)  
    ...
```

Alternative:

```
boolean hasOvertime;  
hasOvertime = (hours > 40);  
...  
if (hasOvertime)  
    ...
```


Logische Operatoren und zusammengesetzte logische Ausdrücke


- Häufig besteht eine Bedingung aus **mehreren Teilbedingungen**, die gleichzeitig erfüllt sein müssen.
- Java erlaubt es, mehrere Tests mit Hilfe **logischer Operatoren** zu einem Test zusammenzusetzen:


```
hours > 40 && hours <= 60
```

- Der **&&-Operator** repräsentiert das logische **Und**.
- Der **||-Operator** realisiert das logische **Oder**.
- Der **!-Operator** realisiert die **Negation**.

Zusammengesetzte if-Anweisungen und Operatoren

- `if`-Anweisungen mit Operatoren können auch zerlegt werden in einzelne `if`-Anweisungen

<code>if (condition1)</code> <code>statement</code> <code>else if (condition2)</code> <code>statement</code>		<code>if (condition1 condition2)</code> <code>statement</code>
-----------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------	----------------------------------------------------------------------

<code>if (condition1)</code> <code>if (condition2)</code> <code>statement</code>		<code>if (condition1 && condition2)</code> <code>statement</code>
----------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------	------------------------------------------------------------------------------

Präzedenzregeln für logische Operatoren

- Der **!-Operator** hat die höchste Präzedenz von den logischen Operatoren. Zweithöchste Präzedenz hat der **&&-Operator**. Schließlich folgt der **||-Operator**.

- Der Ausdruck

```
if (this.hours < hours ||  
    this.hours == hours && this.minutes < minutes)
```

hat daher die gleiche Bedeutung wie

```
if (this.hours < hours ||  
    (this.hours == hours && this.minutes < minutes))
```

- **Durch Klammern werden (logische) Ausdrücke leichter lesbar!**

Wiederholungsanweisungen (Schleifen)

- Neben bedingten Anweisungen ist es in der Praxis häufig erforderlich, ein und dieselbe Anweisung oder Anweisungsfolge auf vielen Objekten zu wiederholen.
- Beispielsweise möchte man das Gehalt für mehrere tausend Mitarbeiter berechnen.
- In Java gibt es mit dem **while-Statement** eine weitere Möglichkeit die Programmausführung zu beeinflussen.
- Insbesondere lassen sich mit dem **while-Statement** Anweisungsfolgen beliebig oft wiederholen.

Das while-Statement

- Die allgemeine Form einer **while-Schleife** ist:

```
while (condition)
    body
```

- Dabei sind die **Bedingung** (`condition`) und der **Rumpf** (`body`) ebenso wie bei der `if`-Anweisung aufgebaut.
- Die **Bedingung** im **Schleifenkopf** ist ein logischer Ausdruck vom Typ `boolean`.
- Der **Rumpf** ist ein einfaches oder ein zusammengesetztes **Statement**.

Ausführung der `while`-Anweisung

1. Es wird **zunächst die Bedingung überprüft**.
2. Ist der **Wert des Ausdrucks `false`**, wird die **Schleife beendet**. Die Ausführung wird dann mit der nächsten Anweisung fortgesetzt, die unmittelbar auf den Rumpf folgt.
3. Wertet sich der **Ausdruck** hingegen zu **`true`** aus, so wird der **Rumpf der Schleife ausgeführt**.
4. Dieser Prozess wird **solange wiederholt, bis** in Schritt 2. der Fall eintritt, dass **sich der Ausdruck zu `false` auswertet**.

Nach Beendigung einer `while`-Schleife gilt somit immer die Negation ihrer Bedingung.

Beispiel: Einlesen aller Zeilen von `www.whitehouse.gov`

- Wir wollen ein Programm schreiben, das alle Zeilen einer Web-Seite einliest.
- Wie können wir feststellen, dass wir am **Ende der Datei** angekommen sind?
- Offensichtlich kann **am Ende einer Datei keine Zeile mehr eingelesen werden**.
- Um dies zu signalisieren liefert die **`readline`-Methode** einen speziellen Wert **`null`** zurück, der **repräsentiert**, dass eine **Referenz-Variable kein Objekt referenziert**.

Beispiel: Einlesen aller Zeilen von `www.whitehouse.gov`

```
import java.net.*;
import java.io.*;

class WHWWWLong {
    public static void main(String[] arg) throws Exception {
        URL u = new URL("http://www.whitehouse.gov/");
        BufferedReader whiteHouse = new BufferedReader(
            new InputStreamReader(u.openStream()));
        String line = whiteHouse.readLine(); // Read first object.
        while (line != null) {                // Something read?
            System.out.println(line);         // Process object.
            line = whiteHouse.readLine();     // Get next object.
        }
    }
}
```


Anwendung der `while`-Schleife zur Approximation

Viele Werte (Nullstellen, Extrema, ...) **lassen sich** (in Java) **nicht durch geschlossene Ausdrücke berechnen**, sondern müssen **durch geeignete Verfahren approximiert** werden.

Beispiel: Approximation von $\sqrt[3]{x}$

Ein beliebtes Verfahren ist die Folge
$$x_{n+1} = x_n - \frac{x_n^3 - x}{3x_n^2},$$

wobei $x_1 \neq 0$ ein beliebiger Startwert ist.

Mit $n \rightarrow \infty$ konvergiert^a x_n gegen $\sqrt[3]{x}$, d.h. $\lim_{n \rightarrow \infty} x_n = \sqrt[3]{x}$

^a Sofern kein $x_n = 0$

Muster einer Realisierung

- Zur näherungsweisen Berechnung verwenden wir eine `while`-Schleife.
- Dabei müssen wir **zwei Abbruchkriterien** berücksichtigen:
 1. Das **Ergebnis ist hinreichend genau**, d.h. x_{n+1} und x_n unterscheiden sich nur geringfügig.
 2. Um zu vermeiden, dass die Schleife nicht anhält, weil die gewünschte Genauigkeit nicht erreicht werden kann, muss man die **Anzahl von Schleifendurchläufen begrenzen**.
- Wir müssen also solange weiter rechnen wie folgendes gilt:

```
Math.abs((xnPlus1 - xn)) >= maxError && n < maxIterations
```

Das Programm zur Berechnung der Dritten Wurzel

```
import java.io.*;

class ProgramRoot {
    public static void main(String arg[]) throws Exception{
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));

        int n = 1, maxIterations = 1000;
        double maxError = 1e-6, xnPlus1, xn = 1, x;

        x = Double.valueOf(br.readLine()).doubleValue();
        xnPlus1 = xn - ( xn * xn * xn - x) / (3 * xn * xn);
        while (Math.abs((xnPlus1 - xn)) >= maxError && n < maxIterations){
            xn = xnPlus1;
            xnPlus1 = xn - ( xn * xn * xn - x) / (3 * xn * xn);
            System.out.println("n = " + n + ": " + xnPlus1);
            n = n+1;
        }
    }
}
```

Anwendung des Programms

Eingabe: -27

```
n = 1: -5.685155555555555
n = 2: -4.068560488977107
n = 3: -3.256075689936079
n = 4: -3.0196112473705674
n = 5: -3.0001270919925287
n = 6: -3.0000000005383821
n = 7: -3.0
Process ProgramRoot finished
```

Eingabe: 10^{90}

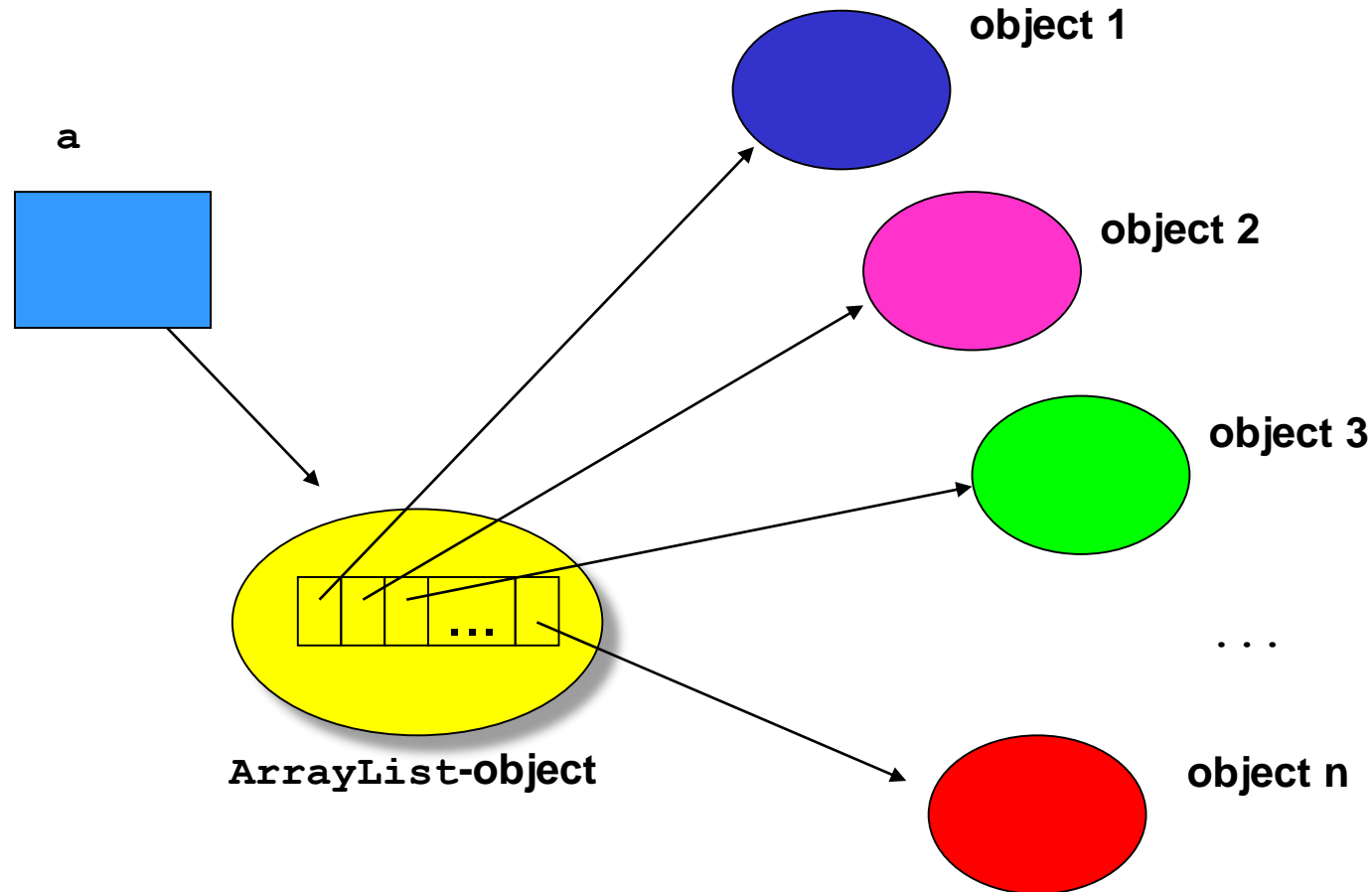
```
n = 1: 2.22222222222222218E89
n = 2: 1.481481481481481E89
n = 3: 9.876543209876541E88
n = 4: 6.584362139917694E88
...
n = 996: 9.999999999999999E29
n = 997: 1.0E30
n = 998: 9.999999999999999E29
n = 999: 1.0E30
Process ProgramRoot finished
```

Kollektionen mehrere Objekte:

Die Klasse ArrayList

- Mit `ArrayList` stellt Java eine Klasse zur Verfügung, die eine **Zusammenfassung von unter Umständen auch verschiedenen Objekten in einer Sequenz** erlaubt.
- **Grundoperationen für Kollektionen** von Objekten sind:
 - das **Erzeugen** einer Kollektion (mit dem Konstruktor),
 - das **Hinzufügen** von Objekten in die Kollektion,
 - das **Löschen** von Objekten aus der Kollektion, und
 - das **Verarbeiten** von Objekten in der Kollektion.

Kollektion von (eventuell unterschiedlichen) Objekten mit der Klasse ArrayList



Erzeugen eines ArrayList-Objektes

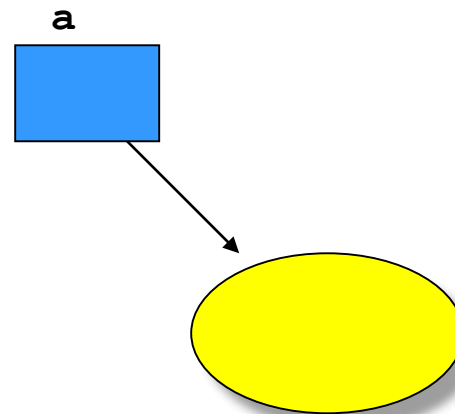
- Wie auch andere Klassen werden `ArrayList`-Objekte mit dem Konstruktor der `ArrayList`-Klasse erzeugt.
- Der Konstruktor von `ArrayList` hat keine Argumente, allerdings sollte **zusätzlich angegeben werden, welche Objekte die Kollektion speichern soll**. Dies wird mittels spitzer Klammern `<>` realisiert.

- Beispielsweise:

```
ArrayList<Integer> a1 = new ArrayList<Integer>();  
ArrayList<String>  a2 = new ArrayList<String>();
```

Erzeugen eines ArrayList-Objektes

- Heute betrachten wir `ArrayList`-Objekte, die `String`-Objekte speichern
- `ArrayList<String> a = new ArrayList<String>();`
- Wirkung des Konstruktors:



`ArrayList<String>`-object

Hinzufügen von Objekten zu einem ArrayList-Objekt

Um Objekte zu einem `ArrayList`-Objekt hinzuzufügen, verwenden wir die Methode `add`.

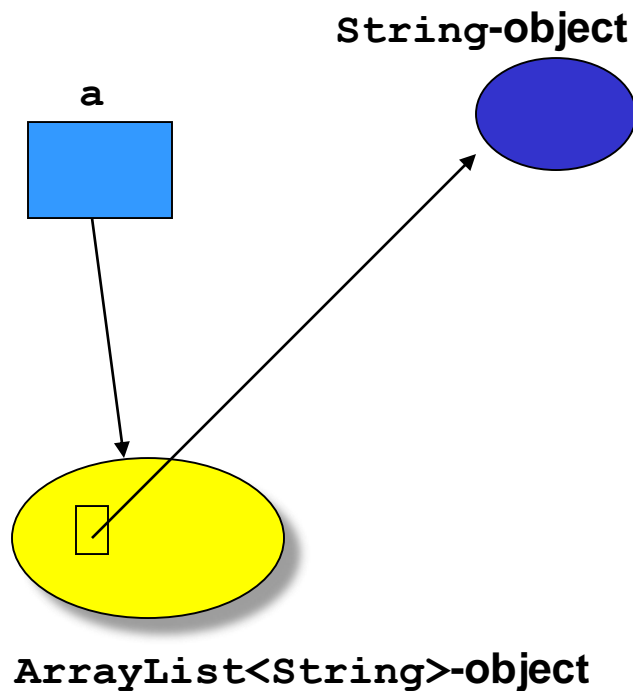
Dieser Methode geben wir als Argument das hinzuzufügende Objekt mit.

Das folgende Programm liest eine Sequenz von `String`-Objekten ein und fügt sie unserem `ArrayList`-Objekt hinzu:

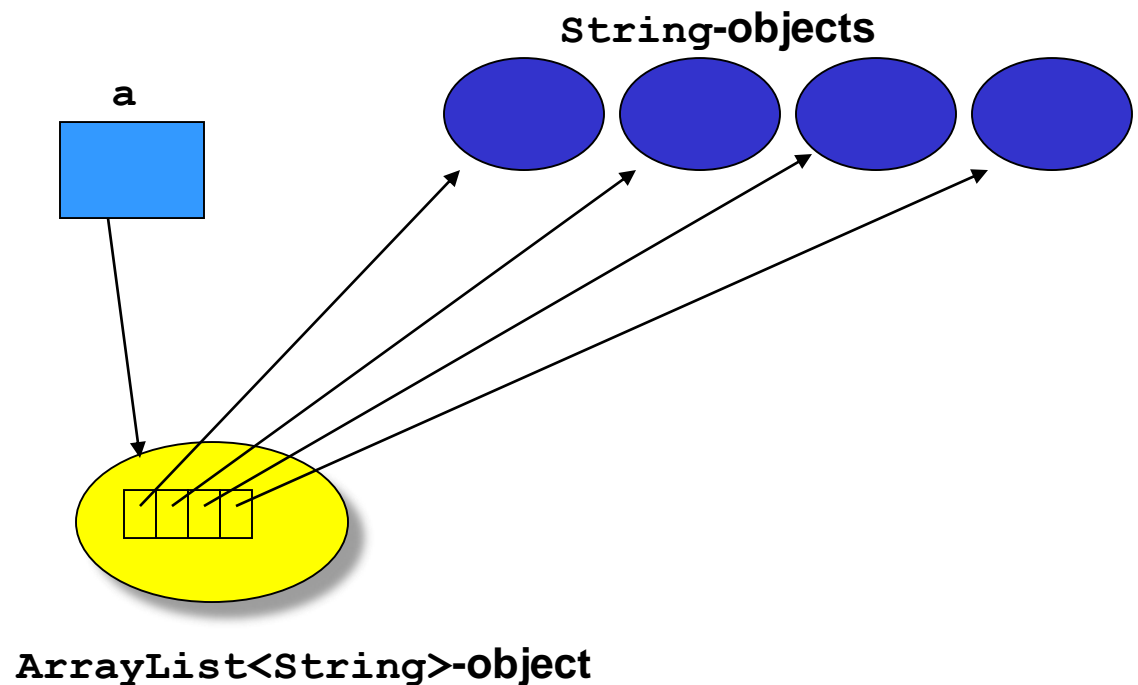
```
ArrayList<String> a = new ArrayList<String>();  
String s = br.readLine(); // Read first String object  
while (s != null){        // Something read?  
    a.add(s);              // Processing adds s to list  
    s = br.readLine();    // Read next String object  
}
```

Anwendung dieses Programmstücks

1 Aufruf von add



4 Aufrufe von add



Unser `ArrayList<String>`-Objekt enthält lediglich Objekte der Klasse `String`.

Durchlauf durch eine ArrayList

- Der Prozess des **Verarbeitens aller Objekte einer Kollektion** wird auch **Durchlauf** genannt.
- Ziel ist es, eine (von der Anwendung abhängige) Operation auf allen Objekten der Kollektion auszuführen.

- Dazu verwenden wir eine **while-Schleife** der Form:

```
while (es gibt noch Objekte, die zu besuchen sind)
    besuche das nächste Objekt
```

- Die **zentralen Aufgaben**, die wir dabei durchführen müssen, sind:
 - auf die **Objekte einer Kollektion zugreifen**,
 - **zum nächsten Element** einer Kollektion **übergangen** und
 - **testen, ob es noch weitere Objekte gibt**, die besucht werden müssen.

Wie kann man Durchläufe realisieren?

- Offensichtlich müssen diese **Funktionen von jeder Kollektionsklasse realisiert werden**.
- Daher sollten die entsprechenden Methoden möglichst so sein, dass sie **nicht von der verwendeten Kollektionsklasse** abhängen.
- Weiter ist es wünschenswert, dass **sich jede Kollektionsklasse an einen Standard** bei diesen Methoden **hält**.
- Auf diese Weise kann man sehr **leicht zu anderen Kollektionsklassen übergehen, ohne dass man das Programm ändern muss**, welches die Kollektionsklasse verwendet.

Iteratoren

Java bietet das **Interface Iterator** zur Realisierung von **Durchläufen durch ArrayList-Objekte** und andere Kollektionsklassen an.

Jede Kollektionsklasse stellt eine **Methode zur Erzeugung eines Iterator-Objektes** zur Verfügung.

Die Klasse **ArrayList** stellt eine Methode **iterator()** zur Verfügung. Diese liefert eine Referenz auf ein **Iterator**-Objekt. Ihr Prototyp ist:

```
Iterator<Object> iterator()    // Liefert einen Iterator für ein  
                               // ArrayList<Object>
```

Die entsprechende **Iterator** Klasse wiederum bietet die folgenden Methoden

```
boolean hasNext()             // True, falls es weitere Elemente gibt  
Object next()                 // Liefert das nächste Objekt  
void remove()                 // Entfernt das zuletzt betrachtete Element
```

Der Return-Type von next

- Im Prinzip muss die Methode `next()` **Referenzen auf Objekte beliebiger Klassen** liefern – je nach dem, welche Klasse im `ArrayList`-Objekt gespeichert wird.
- Um eine breite Anwendbarkeit realisieren zu können, müssen Klassen wie `ArrayList` oder `Iterator` diese **Flexibilität** haben.

Der Return-Type von `next()`

- Bei `ArrayList`en gibt man die zu speichernden Elemente an, z.B.
`ArrayList<Integer>` oder `ArrayList<String>`
- Die gleiche Technik wird auch bei `Iteratoren` verwendet, d.h.:
`Iterator<Integer>` oder `Iterator<String>`
- Somit weiß eine Methode wie `next()`, welchen Typ sie zurückgeben muss.
- **Castings** entfallen somit.

Durchlauf durch ein ArrayList-Objekt

Um einen Durchlauf durch unser `ArrayList<String>`-Objekt `list` zu realisieren, gehen wir nun wie folgt vor:

```
while (es gibt weitere Elemente) {  
    x = hole das nächste Element  
    verarbeite x  
}
```

Dies wird nun überführt zu

```
Iterator<String> e = a.iterator();  
while (e.hasNext()) {  
    String s = e.next();  
    System.out.print(s);  
}
```


Anwendung von ArrayList zur Modellierung von Mengen

- Auf der Basis solcher Kollektionsklassen wie `ArrayList` lassen sich nun andere Kollektionsklassen definieren.
- Im folgenden modellieren wir Mengen mit Hilfe der `ArrayList`-Klasse.
- Ziel ist die Implementierung einer eigenen Klasse `Set` einschließlich typischer Mengen-Operationen.

Festlegen des Verhaltens der Set-Klasse

In unserem Beispiel wollen wir die folgenden Mengenoperationen bzw. Methoden zur Verfügung stellen:

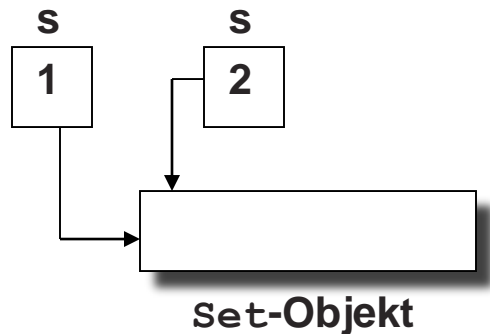
- `Set`-Konstruktor
- `contains` (Elementtest)
- `isEmpty` (Test auf die leere Menge)
- `add` (hinzufügen eines Elements)
- `copy` (Kopie einer Menge erzeugen)
- `size` (Anzahl der Elemente)
- `iterator` (Durchlauf durch eine Menge)
- `union` (Vereinigung)
- `intersection` (Durchschnitt) Alle Elemente ausgeben
- `toString` (Ausgabe der Elemente)

Notwendigkeit der `copy`-Operation

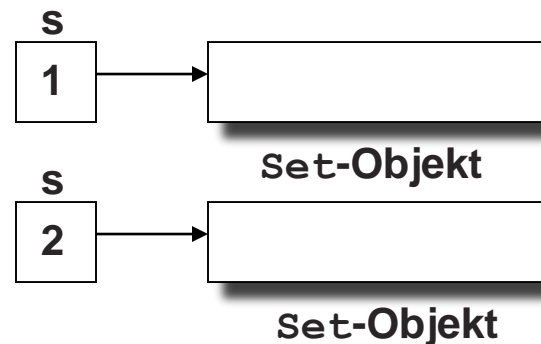
Der Effekt der Anweisung `s2 = s1 = new Set()` ist, dass es zwei Referenzen auf ein- und dasselbe `Set`-Objekt gibt:

Da Methoden wie `add` ein `Set`-Objekt verändern, benötigen wir eine Kopier-Operation um eine Menge zu speichern.

Nach der Anweisung `s2 = s1.copy()` gibt es zwei Referenzen auf zwei unterschiedliche Objekte mit gleichem Inhalt.



`s2 = s1 = new Set();`



`s2 = s1.copy();`

Festlegen der Schnittstellen

Prototypen der einzelnen Methoden:

```
public Set()  
public boolean isEmpty()  
public int size()  
public boolean contains(Object o)  
public void add(Object o)  
public Set copy()  
public Set union(Set s)  
public Set intersection(Set s)  
public Iterator<Object> iterator()  
public String toString()
```

Ein typisches Beispielprogramm

```
class UseSet {
    public static void main(String [] args) {
        Set s1 = new Set();
        s1.add("A");
        s1.add("B");
        s1.add("C");
        s1.add("A");
        System.out.println(s1);
        Set s2 = new Set();
        s2.add("B");
        s2.add("C");
        s2.add("D");
        s2.add("D");
        System.out.println(s2);
        System.out.println(s1.union(s2));
        System.out.println(s1.intersection(s2));
    }
}
```

Das Skelett der Set-Klasse

```
class Set {
    public Set() {... };
    public boolean isEmpty() {... };
    public int size() {... };
    public boolean contains(Object o) {... };
    public void add(Object o) {... };
    public Set copy() {... };
    public Set union(Set s) {... };
    public Set intersection(Set s) {... };
    public Iterator<Object> iterator() {... };
    public String toString() {... };
    ...

    private ArrayList<Object> theElements;
}
```

Implementierung der Methoden (1)

- Der **Konstruktor** ruft lediglich die entsprechende Methode der `ArrayList`-Klasse auf:

```
public Set() {  
    this.theElements = new ArrayList<Object>();  
}
```

- Die **Methoden** `size` und `empty` nutzen ebenfalls vordefinierte Methoden der Klasse `ArrayList`:

```
public boolean isEmpty() {  
    return this.theElements.isEmpty();  
}
```

```
public int size() {  
    return this.theElements.size();  
}
```

Implementierung der Methoden (2)

- Um alle **Elemente der Menge aufzuzählen**, müssen wir eine Methode `iterator` realisieren:

```
Iterator<Object> iterator() {  
    return this.theElements.iterator();  
}
```

- Die **copy-Methode** muss alle Elemente des `ArrayList`-Objektes durchlaufen und sie einem neuen `Set`-Objekt hinzufügen:

```
public Set copy() {  
    Set destSet = new Set();  
    Iterator<Object> e = this.iterator();  
    while (e.hasNext())  
        destSet.add(e.next());  
    return destSet;  
}
```


Implementierung der Methoden (3)

- Da Mengen jeden Wert höchstens einmal enthalten, müssen wir vor dem **Einfügen** prüfen, ob der entsprechende Wert bereits enthalten ist:

```
public void add(Object o) {  
    if (!this.contains(o))  
        this.theElements.add(o);  
}
```

Implementierung der Methoden (4)

- Um die **Vereinigung von zwei Mengen** zu berechnen, kopieren wir die erste Menge und fügen der Kopie alle noch nicht enthaltenen Elemente aus der zweiten Menge hinzu.

```
public Set union(Set s) {  
    Set unionSet = s.copy();  
    Iterator<Object> e = this.iterator();  
    while (e.hasNext())  
        unionSet.add(e.next());  
    return unionSet;  
}
```

Implementierung der Methoden (5)

- Um den **Durchschnitt** von zwei Mengen zu berechnen, starten wir mit der leeren Menge. Dann durchlaufen wir das Empfänger-Set und fügen alle Elemente zu der neuen Menge hinzu, sofern sie auch in dem zweiten Set-Objekt vorkommen.

```
public Set intersection(Set s) {  
    Set interSet = new Set();  
    Iterator<Object> e = this.iterator();  
    while (e.hasNext()) {  
        Object elem = e.next();  
        if (s.contains(elem))  
            interSet.add(elem);  
    }  
    return interSet;  
}
```

Implementierung der Methoden (6)

Um zu **testen, ob ein Objekt in einer Menge enthalten ist**, müssen wir einen Durchlauf realisieren. Dabei testen wir in jedem Schritt, ob das gegebene Objekt mit dem aktuellen Objekt in der Menge übereinstimmt:

- Hierbei ist zu beachten, dass der Gleichheitstest `==` lediglich testet, ob der Wert von zwei Variablen gleich ist, d.h. bei Referenzvariablen, ob sie **dasselbe** Objekt referenzieren (im Gegensatz zu „das gleiche“).
- Um beliebige Objekte einer Klasse miteinander vergleichen zu können, stellt die Klasse `Object` eine Methode `equals` zur Verfügung.
- Spezielle Klassen wie z.B. `Integer` oder `String` aber auch programmierte Klassen können ihre eigene `equals`-Methode bereitstellen.
- Im Folgenden gehen wir davon aus, dass eine solche Methode stets existiert.

Implementierung der Methoden (6)

- Daraus resultiert die folgende Implementierung der Methode `contains`:

```
public boolean contains(Object o) {  
    Iterator<Object> e = this.iterator();  
    while (e.hasNext()) {  
        Object elem = e.next();  
        if (elem.equals(o))  
            return true;  
    }  
    return false;  
}
```

Implementierung der Methoden (7)

Um die **Elemente auszugeben**, verwenden wir ebenfalls wieder einen Durchlauf. Dabei gehen wir erneut davon aus, dass die Klasse des referenzierten Objektes (wie die `Object`-Klasse) eine Methode `toString` bereitstellt.

- Prinzipiell gibt es hierfür verschiedene Alternativen.
- Eine offensichtliche Möglichkeit besteht darin, eine Methode `print(PrintStream ps)` zu implementieren.
- In Java gibt es aber eine elegantere Variante: Es genügt eine Methode `toString()` zu realisieren.
- Diese wird immer dann aufgerufen, wenn ein `Set`-Objekt als Empfänger-Objekt einer `print`-Methode ist.

Die Methode toString()

```
public String toString(){
    String s = "[";
    Iterator<Object> e = this.iterator();
    if (e.hasNext())
        s += e.next().toString();
    while (e.hasNext())
        s += ", " + e.next().toString();
    return s + "]";
}
```

Die komplette Klasse Set

```
import java.io.*;
import java.util.*;
class Set {
public Set() {
    this.theElements = new ArrayList<Object>();
}
public boolean isEmpty() {
    return this.theElements.isEmpty();
}
public int size() {
    return this.theElements.size();
}
Iterator<Object> iterator() {
    return this.theElements.iterator();
}
public boolean contains(Object o) {
    Iterator<Object> e = this.iterator();
    while (e.hasNext()) {
        Object elem = e.next();
        if (elem.equals(o))
            return true;
    }
    return false;
}
public void add(Object o) {
    if (!this.contains(o))
        this.theElements.add(o);
}
public Set copy() {
    Set destSet = new Set();
    Iterator<Object> e = this.iterator();
    while (e.hasNext())
```

```
        destSet.add(e.next());
    return destSet;
}
public Set union(Set s) {
    Set unionSet = s.copy();
    Iterator<Object> e = this.iterator();
    while (e.hasNext())
        unionSet.add(e.next());
    return unionSet;
}
public Set intersection(Set s) {
    Set interSet = new Set();
    Iterator<Object> e = this.iterator();
    while (e.hasNext()) {
        Object elem = e.next();
        if (s.contains(elem))
            interSet.add(elem);
    }
    return interSet;
}
void removeAllElements() {
    this.theElements.removeAllElements();
}
public String toString(){
    String s = "[";
    Iterator<Object> e = this.iterator();
    if (e.hasNext())
        s += e.next().toString();
    while (e.hasNext())
        s += ", " + e.next().toString();
    return s + "]";
}
private ArrayList<Object> theElements;
```

```
}
```


Unser Beispielprogramm (erneut)

```
class UseSet {  
    public static void main(String [] args) {  
        Set s1 = new Set();  
        s1.add("A");  
        s1.add("B");  
        s1.add("C");  
        s1.add("A");  
        System.out.println(s1);  
        Set s2 = new Set();  
        s2.add("B");  
        s2.add("C");  
        s2.add("D");  
        s2.add("D");  
        System.out.println(s2);  
        System.out.println(s1.union(s2));  
        System.out.println(s1.intersection(s2));  
    }  
}
```

Ausgabe des Beispielprogramms

```
java useSet
```

```
[A, B, C]
```

```
[B, C, D]
```

```
[B, C, D, A]
```

```
[B, C]
```

```
Process useSet finished
```

Eine generische Klasse GenericSet

```
import java.util.*;

class GenericSet <E> {
    public GenericSet() {
        this.theElements = new ArrayList<E>();
    }
    public boolean isEmpty() {
        return this.theElements.isEmpty();
    }
    public int size() {
        return this.theElements.size();
    }
    Iterator <E> iterator() {
        return this.theElements.iterator();
    }
    public boolean contains(E o) {
        Iterator<E> it = this.iterator();
        while (it.hasNext()) {
            Object elem = it.next();
            if (elem.equals(o))
                return true;
        }
        return false;
    }
    public void add(E o) {
        if (!this.contains(o))
            this.theElements.add(o);
    }
    public GenericSet <E> copy() {
        GenericSet <E> destSet = new GenericSet <E>
();
        Iterator<E> it = this.iterator();
        while (it.hasNext())
            destSet.add(it.next());
        return destSet;
    }
}
```

```
    public GenericSet <E> union(GenericSet <E> s) {
        GenericSet <E> unionSet = s.copy();
        Iterator<E> it = this.iterator();
        while (it.hasNext())
            unionSet.add(it.next());
        return unionSet;
    }
    public GenericSet <E> intersection(GenericSet <E> s) {
        GenericSet <E> interSet = new GenericSet <E>
();
        Iterator<E> it = this.iterator();
        while (it.hasNext()) {
            E elem = it.next();
            if (s.contains(elem))
                interSet.add(elem);
        }
        return interSet;
    }
    void removeAllElements() {
        this.theElements.removeAllElements();
    }
    public String toString(){
        String s = "[";
        Iterator<E> it = this.iterator();
        if (it.hasNext())
            s += it.next().toString();
        while (it.hasNext())
            s += ", " + it.next().toString();
        return s + "]";
    }
    private ArrayList<E> theElements;
}
```

Beispielprogramm für generische Sets

```
class UseGenericSet {  
    public static void main(String [] args) {  
        GenericSet <String> s1 = new GenericSet <String> ();  
        s1.add("A");  
        s1.add("B");  
        s1.add("C");  
        s1.add("A");  
        System.out.println(s1);  
        GenericSet <String> s2 = new GenericSet <String> ();  
        s2.add("B");  
        s2.add("C");  
        s2.add("D");  
        s2.add("D");  
        System.out.println(s2);  
        System.out.println(s1.union(s2));  
        System.out.println(s1.intersection(s2));  
    }  
}
```

Vorteile von generischen Klassen

- Mit Hilfe von generischen Klassen wird erreicht, dass Klassen auf einer Vielzahl von anderen Klassen operieren können, wobei aber gleichzeitig eine größere Sicherheit bereits zur Übersetzungszeit erreicht wird.
- Damit erhält man Flexibilität und kann gleichzeitig Programme sicherer machen, weil viele Überprüfungen bereits bei der Übersetzung gemacht werden können.
- Beispielsweise wird durch

```
GenericSet <String> s1 = new GenericSet <String> ();
```

sichergestellt, dass `s1` lediglich String-Objekte referenzieren kann.
- Wertzuweisungen können so zur Übersetzungszeit geprüft werden. In früheren Versionen von Java ging das lediglich zur Laufzeit durch die Casting-Operation.

Spezialisierung generischer Klassen

- Die Spezialisierung unserer generischen Klasse `GenericSet` kann beispielsweise mit Hilfe von Textueller Ersetzung durchgeführt werden.
- Ersetze an allen Stellen "`<E>`" durch "`-E`".
- Ersetze danach an allen Stellen das `E`, welches für eine beliebige Klasse steht durch die konkrete Klasse, beispielsweise `String`.
- Dadurch entsteht eine spezialisierte Klasse `GenericSet-String`, welche nicht generisch ist und keine Konstrukte generischer Klassen enthält.
- Diese könnte man prinzipiell mit einem alten Compiler übersetzen.
- Dazu muss allerdings auch der Iterator spezialisiert werden.

Die for-Schleife

- Speziell für Situationen, in denen die Anzahl der Durchläufe von Beginn an feststeht, stellt Java mit der `for`-Schleife eine Alternative zur `while`-Schleife zur Verfügung.
- Die allgemeine Form der `for`-Schleife ist:

```
for (Initialisierungsanweisung; Bedingung; Inkrementierung)  
    Rumpf
```

- Sie ist äquivalent zu

```
Initialisierungsanweisung  
while (Bedingung) {  
    Rumpf  
    Inkrementierung  
}
```

Anwendung: Potenzieren mit der `for`-Schleife

- Zur Formulierung des Verfahrens betrachten wir zunächst, wie wir die Berechnung von x^y per Hand durchführen würden:

$$x^y = \begin{cases} 1 & \text{falls } y = 0 \\ \underbrace{x * \dots * x}_{y \text{ mal}} & \text{sonst} \end{cases} = 1 * \underbrace{x * \dots * x}_{y \text{ mal}}$$

- Daraus ergibt sich ein informelles Verfahren:
 1. starte mit 1
 2. multipliziere sie mit x
 3. multipliziere das Ergebnis mit x
 4. führe Schritt 3) solange aus, bis y Multiplikationen durchgeführt wurden.

Potenzierung mit der for-Anweisung

- Bei der Potenzierung mussten wir genau y Multiplikationen durchführen.
- Die Anzahl durchgeführter Multiplikationen wird einfach in einer Variablen `count` gespeichert.

```
static int power(int x, int y){  
    int count, result = 1;  
  
    for (count = 0; count < y; count++)  
        result *= x;  
  
    return result;  
}
```

Komplexere for-Anweisungen

- Die Initialisierungs- und die Inkrementierungsanweisung können aus mehreren, durch Kommata getrennten Anweisungen bestehen.
- Betrachten wir die analoge `while`-Schleife, so werden die Initialisierungsanweisungen vor dem ersten Schleifendurchlauf ausgeführt.
- Auf der anderen Seite werden die Inkrementierungsanweisungen am Ende jedes Durchlaufs ausgeführt.
- Damit können wir auch folgende `for`-Anweisung zur Berechnung von x^y verwenden:

```
for (count = 0, result = 1; count < y; result*=x,  
    count++);
```
- Solche **kompakten Formen der for-Anweisung** sind **üblicherweise schwerer verständlich** und daher **für die Praxis nicht zu empfehlen**.

Zusammenfassung (1)

- **Bedingte Anweisungen** erlauben es, in Abhängigkeit von der Auswertung einer Bedingung im Programm **verschiedene Anweisungen durchzuführen**.
- Dadurch kann der Programmierer den **Kontrollfluss steuern** und in seinem Programm entsprechend **verzweigen**.
- Mit einem **if-Statement** kann man **zwei Fälle** unterscheiden.
- Durch Kaskardierung kann man **mehr als zwei Fälle** unterscheiden.

Zusammenfassung (2)

- Bedingungen sind **Boolesche Ausdrücke**, die zu `true` oder `false` ausgewertet werden..
- In Java gibt es dafür den primitiven Datentyp `boolean` mit den beiden Werten `true` und `false`.
- Einfache **Boolesche Ausdrücke** können mit den **Vergleichsoperatoren** `<`, `>`, `<=`, `>=`, `==`, und `!=`, die auf Zahltypen operieren, definiert werden.
- **Komplexere Boolesche Ausdrücke** werden mit den **logischen Operatoren** `&&`, `||` und `!` zusammengesetzt.

Zusammenfassung (3)

- Die **Wiederholung von Anweisungssequenzen** durch **Schleifen** oder **Loops** ist eines der **mächtigsten Programmierkonstrukte**.
- Mit Hilfe von Schleifen wie der **while-Schleife** können Sequenzen von **Anweisungen beliebig häufig wiederholt** werden.
- Die **for-Schleife** ist ein äquivalentes Konstrukt zur **while-Schleife**. Die **for-Schleife** eignet sich besonders, wenn die Anzahl der Iterationen im Vorhinein bekannt ist.

Zusammenfassung (4)

- **Kollektionen** sind Objekte, die es erlauben, **Objekte zusammenzufassen**.
- **ArrayList** ist eine solche **Kollektionsklasse**, mit der **Objekte beliebiger Klassen** zusammengefasst werden können.
- Die **einzelnen Objekte** eines `ArrayList`-Objektes können mit **Durchläufen** unter Verwendung eines Objektes der Klasse **Iterator** **prozessiert** werden.
- Mit Hilfe der Klasse `ArrayList` können wir dann **andere Kollektionsklassen definieren** (wie z.B. eine `Set`-Klasse).