Kleine Einführung in Java

Vera Röhr und Benjamin Blankertz

TECHNISCHE UNIVERSITÄT BERLIN

VORLESUNGSSKRIPT

Einführung in Java für AlgoDat

Vera Röhr und Benjamin Blankertz

[Fachgebiet Neurotechnologie, MAR 4.3]

Inhaltsverzeichnis

KAPITEL 1

Einführung

Dieses Dokument dient als Überblick über die objektorientierte Programmiersprache Java und ihre Konzepte. Dazu geben wir zuerst einen kleinen Einblick in den Aufbau der Sprache und die Syntax im Vergleich zu C, dann gehen wir ausführlich auf das Konzept der Objektorientierung, sowie Vererbung und Schnittstellen ein. Des Weiteren betrachten wir verschiedene Datenstrukturen in Java, Fehlerbehandlung und JUnit Tests. Um die Programmiersprache jedoch zu lernen, ist neben der Theorie vorallem auch Programmierpraxis notwendig. Sie finden viele Code-Beispiele in diesem Kapitel, lesen Sie diese gründlich und probieren Sie auch Abschnitte selbst aus. Weiterführendes Übungsmaterial finden Sie hier:

- Crashkurs auf ISIS
- hyperskill
- openHPI Objektorientierte Programmierung in Java

1.1 Annäherung an Java

Entstanden aus einem Software Projekt für Unterhaltungselektronik wurde Java als Programmiersprache für das Internet entwickelt. Inzwischen ist sie open source und eine der beliebtesten Programmiersprachen weltweit. Als Ziel hatten sich die Entwickler gesetzt, eine sichere und einfache objektorientierte Programmiersprache zu schreiben, die nicht in der Performance einbüßt und unabhängig vom Betriebssystem und der Plattform ausgeführt werden kann.
Einige dieser Ziele wurden durch das Konzept der *Java Virtual Machine* (JVM) umgesetzt. Die JVM bildet eine virtuelle Maschine für jedes aufgerufene Java Programm und simuliert einen Chip, für den Java ursprünglich entwickelt wurde.

Sie enthält einen Klassenlader, Speichermanagement und garbage collection, sowie eine execution engine. Java Bytecode kann also überall ausgeführt werden, wo eine JVM vorhanden ist und das ist sowohl im Browser als auch auf allen gängigen Betriebssystemen der Fall.
Dass Programme als Bytecode auf einer virtuellen Maschine interpretiert werden und nicht als Binärprogramm auf einem bestimmten Betriebssystem, führt dazu, dass auch schädlicher Code in einer Art kontrolliertem Container ausgeführt wird. Alle Programme werden zur Laufzeit überprüft (Bytecode verifier) und die Speicherverwaltung ist in der JVM bereits integriert. Zeiger, wie man sie aus C oder C++ kennt, werden generell nicht unterstützt, stattdessen nutzt Java Referenzen. Ungültige Zeiger könnten z. B. unkontrolliert Speicher beschreiben. Außerdem sind weitere Sicherheitsmaßnahmen in die Programmiersprache eingebaut wie die Möglichkeit, den Zugriff auf bestimmte Teile des Programms zu limitieren.
Ein Pfeiler der guten Performance von Java ist die *Just-in-time-Compilation*, welche während der Ausführung des Programmes den Byte Code in Maschinencode übersetzt. Damit kann die Laufzeit dynamisch optimiert werden, indem beispielsweise Variablen, die ihren Wert nicht mehr ändern, als Konstanten behandelt werden können.

1.1.1 Übersicht über die Unterschiede C – Java

Um das Erlernen von Java zu erleichtern und eine gewisse Vertrautheit in der Sprache zu erreichen, hat man sich stark an C++ (und C) orientiert. Deshalb ist die Syntax mitunter sehr ähnlich. Dennoch gibt es offensichtlich Unterschiede zwischen den Programmiersprachen, vor allem zu C. Neben den konzeptionellen Unterschieden zwischen einer objektorientierten (Java) und einer prozeduralen (C) Programmiersprache, gibt es in C keine JVM, keine automatische Speicherverwaltung etc. Im folgenden sind einige der wichtigsten Unterschiede aufgelistet:

	C	Java
Ausführung	Compiler erzeugt Maschinencode	Byte Code, muss interpretiert werden
Boolean	`int` mit 0 für *false* und $!=0$ für *true*	`boolean` mit Werten `true` und `false`
Array Deklaration	`int *x = malloc(N*sizeof(*x)`	`int[] x = new int[N];`
Array Größe	unbekannt für das Array	`x.length`
Zeichenketten	'\0' terminiertes `char` Array	Datentyp `String`
Datenstruktur definieren	`struct`	`class` - Klassen mit Methoden
Bibliotheken laden	`#include <stdio.h>`	`import java.io.File;`

Bibilotheksfunktionen nutzen	`#include "math.h"`
	`x = sqrt(3.14);`
	Funktionen sind global
		`x = Math.sqrt(3.14);`
		Funktionen haben *namespaces*

Speicher referenz.	Zeiger (*, &, +)	Referenzen
Speicher reservieren	`malloc`	`new`
Speicher freigeben	`free`	automatische Speicherbereinigung
Generischer Datentyp	`void *`	`Object`
Null	`NULL`	`null`
Variablen automatisch initialisiert	nicht garantiert	Instanzvariablen und Array Elemente initialisiert mit `0`, `null`, bzw. `false`
Variablen deklarieren	am Anfang eines Blocks	irgendwo, vor der Benutzung
Variablennamen Konvention	`mean_square_error`	`meanSquareError`
Dateinamen	`stack.c`, `stack.h`	`Stack.java` übereinstimmend mit Klassennamen
Bildschirmausgabe	`#include<stdio.h>` `printf("I am C\n");`	`System.out.println("I am Java\n");`
Kommentare	`/* ... */` oder vorangestelltes `//`	`/* ... */` oder vorangestelltes `//`

Weitere Informationen findet man unter `https://introcs.cs.princeton.edu/java/faq/c2java.html`

1.2 Programmieren in Java

Wie in C auch, wird in Java zunächst ein Quellcode geschrieben, dann wird dieser kompiliert und kann dann, wenn alles fehlerfrei war, ausgeführt werden.

In jedem dieser Schritte kann und wird es Fehler geben, die man eingebaut hat, inklusive des letzten Schrittes: dem (nur manchmal richtigen) Ergebnis. Es sind also in jedem Level Kontrollschleifen in Form von Debugging des Quellcodes zu durchlaufen (Siehe Abbildung 1.1), bevor der Bytecode fehlerfrei von der JVM ausgeführt werden kann und das richtige Ergebnis herauskommt.

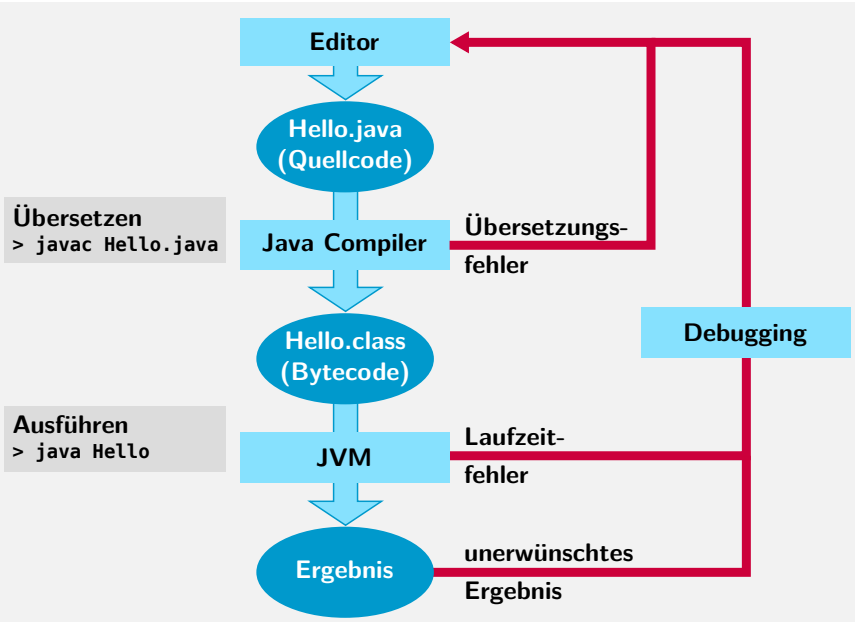


Abbildung 1.1 Ablauf des Programmierens in Java (ohne IDE): Zunächst wird der Quellcode in einem Editor geschrieben. Diese Datei muss die Endung `.java` haben, in diesem Beispiel heißt sie `Hello.java`. Sie wird durch die Eingabe des Befehls `javac Hello.java` in einer Konsole vom Java Compiler in Bytecode übersetzt. Die kompilierte Datei hat die Endung `.class`. Durch den Befehl `java Hello` wird der Bytecode von der *Java Virtual Machine* (JVM) ausgeführt.

Um das Wiederholen der Kontrollschleifen zu minimieren und zu vereinfachen, schnellen Zugriff auf häufig genutzte Tools, sowie automatische Formatierung und viele andere kleine und große Hilfen zu nutzen, wird Java fast immer innerhalb einer Integrierten Entwicklungsumgebung (*integrated development environment* IDE) programmiert. IDEs kombinieren typischerweise Editor, Compiler, Interpreter, Linker und Debugger in einem und bieten viele weitere Möglichkeiten, wie beispielsweise das Einbinden von automatischen Tests. In diesem Modul wird *Intellij IDEA (Community Edition)* benutzt. Hinweise zur Installation sind auf dem Blatt 0 oder im Java Crash Kurs zu finden.
Wenn wir uns nun noch einmal die Abbildung 1.1 anschauen, werden in einer IDE Kompiler- oder Übersetzungsfehler meist sofort als Fehler markiert, sodass

sie direkt beim Schreiben behoben werden können. Zum Kompilieren und Ausführen muss kein Terminal aufgemacht werden, beides ist bereits in der IDE integriert. Zum Finden der Ursache von Laufzeitfehlern oder einem falschen Ergebnis, kann man mit dem Debugger den Code Schritt für Schritt und Zwischenergebnis für Zwischenergebnis durchgehen. (siehe auch Abschnitt 4.1.1)
In den nächsten Abschnitten wiederholen wir kurz primitive Datentypen und Arrays, welche aus dem 1. Semester bekannt sein sollten, für Java.

1.2.1 Primitive Datentypen

Datentypen legen fest, um was für eine Art Variable es sich handelt, welche Operationen darauf ausgeführt werden können und wie die Repräsentation der Variablen im Speicher aussieht, d.h. welches Bit welche Bedeutung hat und wie viele es davon gibt. **Primitive Datentypen** (*primitive data types*) setzen sich nicht aus einfacheren Datentypen zusammen, sie sind die Grundbausteine der Datentypen. In Java gibt es die folgenden :

Datentyp	Inhalt	Wertebereich
`boolean`	1 Bit Wahrheitswert (Größe undefiniert)	`true`, `false`
`char`	16 Bit Unicode	Unicode Characters, z.B. a oder b oder c
`byte`	vorzeichenbehaftete ganze Zahl in 8 Bit	$-2^7 \cdots 2^7-1 = 127$
`short`	vorzeichenbehaftete ganze Zahl in 16 Bit	$-2^{15} \cdots 2^{15}-1 = 32767$
`int`	vorzeichenbehaftete ganze Zahl in 32 Bit	$-2^{31} \cdots 2^{31}-1 = 2147483647$
`long`	vorzeichenbehaftete ganze Zahl in 64 Bit	$-2^{63} \cdots 2^{63}-1 = 9223372036854775807$
`float`	Gleitkommazahl in 32 Bit	$\pm\, 2^{127} \approx 10^{38}$, 7 signifikante Stellen
`double`	Gleitkommazahl in 64 Bit	$\pm\, 2^{1023} \approx 10^{308}$, 15 signifikante Stellen

Wenn diesen Datentypen kein Wert zugewiesen wird, werden sie automatisch initialisiert und zwar auf `false` oder `0` bzw. den null-character für `char`. Am häufigsten werden `boolean`, `int` und `double` verwendet, sowie die Klasse (abstrakter Datentyp) `String`, welche Zeichenketten implementiert.
Mit Hilfe dieser Datentypen und Klassen, welche wir in Abschnitt 2.1 kennenlernen werden, können nun reale Größen und Zusammenhänge dargestellt werden. Damit diese Darstellung akkurat ist, sollten Datentypen mit Bedacht gewählt

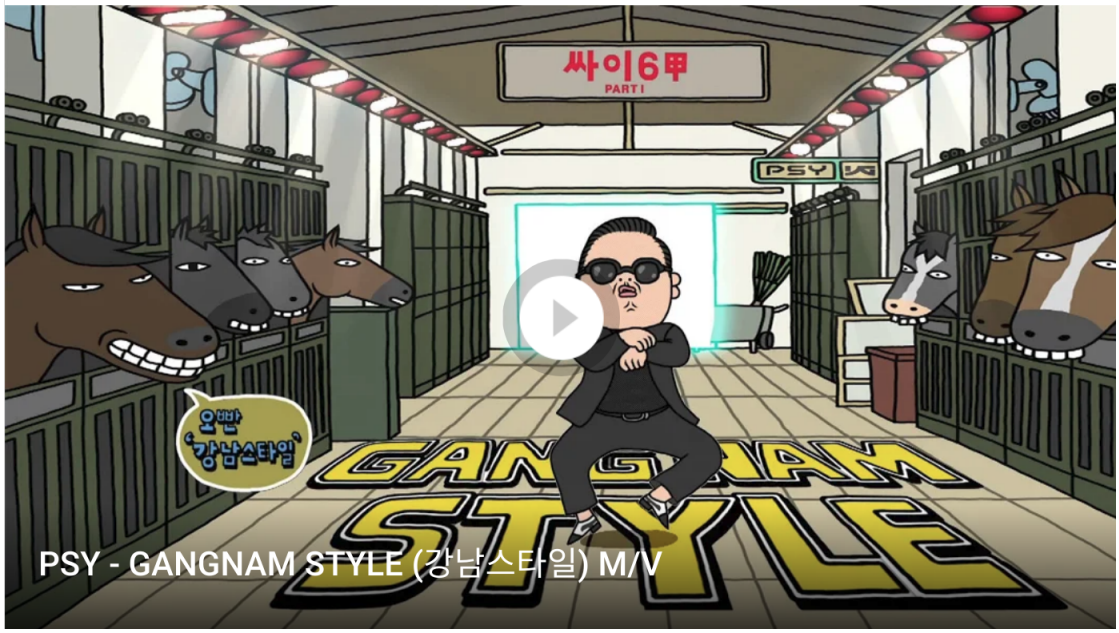
werden, was diese Anekdote von Youtube aus dem Jahr 2014 illustriert:

YouTube ▸ Öffentlich 01.12.2014

We never thought a video would be watched in numbers greater than a 32-bit integer (=2,147,483,647 views), but that was before we met PSY. "Gangnam Style" has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!

Hover over the counter in PSY's video to see a little math magic and stay tuned for bigger and bigger numbers on YouTube.

Übersetzen



1.2.2 Arrays

Arrays oder Felder (*array*) speichern eine bestimmte Anzahl an Objekten eines bestimmten Datentyps, wobei jedes Objekt eine Nummer bekommt, nämlich den Index, an dem es zu finden ist. Um in Java ein Array anzulegen, bedarf es dieser drei Schritte:

- Deklaration mit Angabe von Namen und Typ
- Speicher reservieren (Array anlegen)

- Elemente des Arrays mit Werten initialisieren

Als Code könnte das folgendermaßen aussehen:

```
double[] x;                  // Variable als Array deklarieren
x = new double[K];           // und erzeugen (Speicher reservieren)
for (int k = 0; k < K; k++)
  x[k] = 0.0;                // initialisieren
```

Was die Variable als Array kennzeichnet sind die eckigen Klammern hinter dem Datentyp. Um ein neues Objekt zu erzeugen, wird dann der new Operator aufgerufen und die Anzahl der Elemente des Arrays (hier K) wieder in den Klammern hinter dem Datentyp angegeben. Die Initialisierung auf 0.0 in der for-Schleife ist aber in diesem Fall nicht notwendig, da die automatische Initialisierung von Java die Werte bereits vorher auf 0.0 gesetzt hatte. Um ein Array der Länge K zu initialisieren, der mit Nullen gefüllt ist, brauchen wir also tatsächlich nur eine Zeile:

```
Datentyp[] x = new Datentyp [K];
```

oder für unser vorheriges Beispiel:

```
double[] x = new double[K];
```

Dabei ist zu beachten, dass die Indizierung von Arrays in Java (wie in vielen Programmiersprachen) mit `0` beginnt. Mit obiger Definition gibt es also die Elemente `x[0]` bis `x[K-1]`, ein Element `x[k]` existiert jedoch nicht.
Eine zweite Möglichkeit der Initialisierung, die sinnvoll ist, wenn die Elemente nicht Null sind und die Initialisierung über eine for-Schleife nicht sinnvoll ist, ist die Deklaration mit Initialisierung durch eine Werteliste. Dafür werden die Werte einfach in Listenform angegeben:

```
int[] fibo = { 0, 1, 1, 2, 3, 5, 8 };
```

Um auf eine bestimmte Stelle eines Arrays zuzugreifen, gibt man den spezifischen Index in eckigen Klammern an. Zum Beispiel: `fibo[5]`, wenn wir das mit folgendem Code ausführen würden, käme 5 heraus.

```
System.out.println(fibo[5])
```

Mehrdimensionale Arrays

In Java gibt es keine 'echten' mehrdimensionalen Arrays, stattdessen werden Arrays von Arrays (von Arrays ...) gebildet. Das hat viele wichtige Konsequenzen, z.B. beim Kopieren und Vergleichen, die später erläutert werden. Zunächst

schauen wir uns ein Beispiel für die Erzeugung eines 2-dimensionalen Arrays an:

```
// Deklaration 2-dim Array:
int[][] x;
// Erzeugen eines 2-dim Arrays
x = new int[4][3];
for (int i = 0; i < 4; i++) {
  for (int j = 0; j < 3; j++) {
    x[i][j] = i - j;
  }
}
```

Das funktioniert im Prinzip, wie bei eindimensionalen Arrays, nur dass wir jetzt zwei Indizes beachten müssen. Aber da wir eigentlich nur Arrays von Arrays bilden, müssen mehrdimensionale Arrays tatsächlich nicht `rechteckig` sein. D. h. jedes Array, das in dem ersten Array gespeichert ist, kann eine andere Länge haben. Das sieht dann z. B. so aus:

```
int[][] x;            // Variable als Array deklarieren
x = new int[5][];     // "erstes" Array deklarieren der Länge 5
for (int i = 0; i < x.length; i++) {
  x[i] = new int[i+1];// Arrays im Array mit unterschiedlichen Längen
  for (int j = 0; j < x[i].length; j++) {
    x[i][j] = i - j;
  }
}
```

Es besteht auch hier wieder die Möglichkeit der Initialisierung durch eine Werteliste, genau genommen sogar zwei Möglichkeiten:

```
int[][] square = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
int[][] triangle = new int[][] {{1}, {2, 3}, {4, 5, 6}};
```

Iterieren über Arrays

Um alle Elemente eines Arrays anzusprechen, kann man mit Hilfe einer for-Schleife über die Indizes iterieren. Dabei wird in jeder Iteration ein Element des Arrays angesprochen. Um die Länge eines Arrays zu bestimmen, kann `array.length` verwendet werden:

```
double[] folge = {12, -4, 5.6, 17};
double sum = 0.0;
for (int k = 0; k < folge.length; k++) {
  sum += folge[k];
}
```

Oder man iteriert direkt über das Array, ein Mechanismus, auf den wir an anderer Stelle genauer eingehen werden.

```
double[] folge = {12, -4, 5.6, 17};
double sum = 0.0;
for (double zahl : folge) {
  sum += zahl;  // zahl ist das gerade betrachtete Element von folge
}
```

In dem Codeabschnitt durchläuft die Variable `zahl` alle Elemente des Arrays `folge`. Direkte Iteration funktioniert auch über mehrdimensionale Arrays, nur dass die Elemente, über die iteriert wird, in der ersten Schleife auch noch Arrays sind:

```
int[][] triangle = {{1}, {2, 3}, {4, 5, 6}};
int sum = 0;
for (int[] row : triangle) { // die Elemente von triangle sind int-Arrays
  for (int element : row) { // die Elemente von row sind int
    sum += element;
  }
}
```

KAPITEL 2

Konzepte

2.1 Objektorientierte Programmierung (OOP)

Wir haben nun schon mehrfach erwähnt, dass Java eine objektorientierte Programmiersprache ist, aber noch nicht erklärt, was das eigentlich heißt. Das werden wir jetzt nachholen. Java umfasst neben den aus C bereits bekannten Variablen, Operatoren, Fallunterscheidungen, Schleifen und Funktionen, die in Java statische Methoden genannt werden, auch Objekte und Klassen. Letztere bilden die grundlegende Organisationseinheit der gesamten Sprache. Die Beziehung zwischen Objekten und Klassen betrachten wir im nächsten Abschnitt und beschränken uns für den Moment auf die namensgebenden Objekte. Objekte fassen Methoden und Daten in einer Einheit zusammen. Damit verwaltet jedes Objekt seine eigenen Daten und kontrolliert deren Manipulation, wobei sowohl der Zugriff auf die Daten, als auch auf die Methoden beschränkt sein kann. In prozeduralen Sprachen (wie C) können Unterprogramme dagegen auf den ganzen Speicherbereich zugreifen.

2.1.1 Das Konzept von Klassen und Objekten

Das Konzept von Klassen und Objekten basiert auf der Idee, dass wir mit unserem Code versuchen, die Wirklichkeit wiederzuspiegeln, nur in vereinfachter Form. Wir bauen **Modelle** und stellen damit reale Objekte dar. Diese Modelle sind unsere Klassen. Eine **Klasse** (*class*) definiert zusammengesetzte Datenstrukturen (wie `struct` in C) und die darauf zulässigen Operationen (objektbezogene **Methoden**).
Betrachten wir nun ein **Beispiel**: ein paar reale Spielsteine eines Brettspiels.
Sie haben viele unterschiedliche Eigenschaften, wie das Material, die Farbe, die

Position auf dem Spielbrett etc. Wenn wir jetzt ein Spiel programmieren würden, sind uns die meisten dieser Eigenschaften nicht wichtig, die Position auf dem Spielbrett könnte aber relevant sein. Letzteres wird dann ein Attribut der Klasse. Hierbei ist anzumerken, dass in der Klasse keine spezifische Position festgehalten wird, nur dass es eine gibt. Denn die Klasse wird der Bauplan werden, aus dem man viele verschiedene gesetzte Spielsteine instanziieren kann.

Realität	**reales Objekt** ▶ hat Eigenschaften (Attribute) ▶ man kann etwas damit machen (Operationen)
⇓ **Modellierung / Abstraktion** (Programmierung)	
Code	**Klasse** (fasst Objekte eines Typs zusammen) ▶ Definiert, welche Eigenschaften das Objekt besitzen kann (Attribute) ▶ Implementation der Operationen: Methoden
⇓ **Instanziierung** (beim Programmablauf)	
Speicher	**Objekt** (Exemplar, Instanz der Klasse) ▶ Besitzt konkrete Eigenschaften (Attribute haben Werte) ▶ Anwendung einer Operation: Aufruf der Instanzmethode für das konkrete Objekt.

Man kann Spielsteine in der Realität auf dem Brett bewegen und so kann man der entsprechenden Klasse Token eine Methode geben, die festlegt wie ein Stein auf dem Brett bewegt werden kann. Solange es aber kein spezifisches Objekt gibt, das bewegt wird, beschreibt die Methode nur die allgemeine Möglichkeit Spielsteine zu bewegen. Um die Methode ausführen zu können, muss ein Objekt instanziiert werden, zum Beispiel das Token `weißerKreisMitgrünemPunkt`. D. h. es muss dafür einen Platz im Speicher geben (der `new` Operator findet hier seine Anwendung). Außerdem bekommt `weißerKreisMitgrünemPunkt` eine bestimmte Position zugewiesen und kann nun auch auf dem Brett bewegt werden. **Objekte** sind also durch folgende Aspekte charakterisiert:

- **Identität:** Speicherbereich des Objektes
- **Zustand:** Wert des Datentyps = spezifische Position (Instanzvariablen)
- **Verhalten:** Definiert durch die Objektmethoden

Sie sind spezifische Exemplare ihrer Klasse.

2.1.2 Klassen in Java

Konstruktoren

Damit eine Klasse instanziiert werden kann, besitzt jede Klasse mindestens einen **Konstruktor**, eine Methode mit dem Klassennamen, bei der kein Ausgabeargument angegeben wird, auch nicht `void`. Der Konstruktor wird aufgerufen, wenn man mit **`new`** ein Exemplar oder eine Instanz (*instance*) der Klasse erzeugt. Dies ist dann ein **Objekt**. Es kann mehrere Konstruktoren geben, wenn sie unterschiedliche Signaturen (Sequenz der Argumenttypen) haben. Dabei rufen häufig komplexere Konstruktoren die einfacheren Konstruktoren auf: **Verkettung von Konstruktoren**. Mit `this(...)` kann ein anderer Konstruktor aufgerufen werden. Das muss aber immer die erste Codezeile im Konstruktor sein. Das Schlüsselwort **`this`** ist eine Referenz auf das Objekt, für das es aufgerufen wird.

Kategorien von Variablen in Klassen

In Klassen gibt es drei Kategorien von Variablen:

- **Klassenvariablen** sind Datenfelder, die als **`static`** definiert werden. Sie existieren nur ein einziges Mal pro Klasse und alle Objekte der Klasse können auf sie zugreifen.
- **Instanzvariablen** existieren unabhängig voneinander für jede Instanz der Klasse. Sie repräsentieren Eigenschaften bzw. den momentanen Zustand des konkreten Objektes.
- Lokale Variablen werden *innerhalb* von Methoden definiert und existieren nur für die Dauer des Methodenaufrufs.

Kategorien von Methoden in Klassen

In Klassen gibt es zwei Kategorien von Methoden:

- **Klassenmethoden** sind Methoden, die als **`static`** definiert werden. Sie beziehen sich auf kein konkretes Objekt der Klasse (können allerdings als ein Argument ein Objekt der eigenen Klasse nehmen). Daher können sie nur auf Klassenvariablen und nicht auf Instanzvariablen zugreifen.

  Klassenmethoden ruft man über den Klassennamen und den Punkt Operator auf, z. B. `Math.log(17)`.

- **Instanzmethoden** existieren unabhängig voneinander für jede Instanz der Klasse. Sie beziehen sich immer auf eine konkrete Instanz bzw. Objekt der Klasse und können auch auf private Eigenschaften dieses Objektes zugreifen.

  Instanzmethoden ruft man über den Variablennamen des Objektes und den Punkt Operator auf, z. B. `str.length()` für eine Variable `str` der Klasse `String`.

Beispiel Klasse 'Token' (Spielsteine)

Wir kehren nun zu unserem Spielsteinbeispiel zurück, um den Unterschied von Instanz- und Klassenvariablen zu verdeutlichen. Die meisten Variablen, die verwendet werden, sind Instanzvariablen. Klassenvariablen sind eher die Ausnahme.
Die Klasse `Token` hat die Position auf dem Spielbrett als Instanzvariablen in Form der Koordinaten `xPos` und `yPos`. Die Werte der Koordinaten können für jedes Objekt verschieden sein (und bei diesem Spiel müssen sie es sogar). Außerdem gibt es eine Klassenvariable `counter`, die zählt, wie viele Spielsteine es insgesamt gibt. Dieser Wert bezieht sich auf die Gesamtheit aller Objekte der Klasse. Daher macht die Variable als Instanzvariable keinen Sinn und wird über das Schlüsselwort `static` als Klassenvariable definiert.

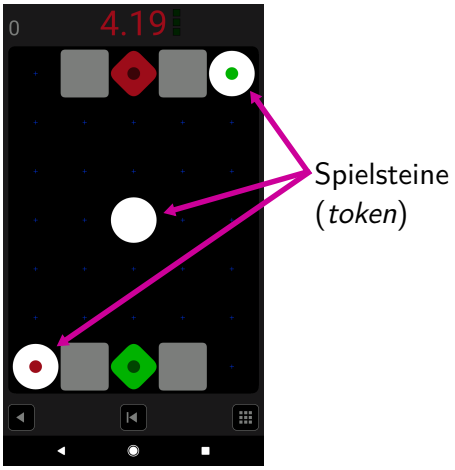


Abbildung 2.1 Jeder Spielstein hat eine eigene Position (Instanzvariablen `xPos`, `yPos`). Von der Klasse Token gibt es 3 Objekte (Klassenvariable `counter`).

Die Klassenvariable wird durch den 1. Konstruktor gesetzt. Um die Instanzvariablen zu setzen, muss der 2. Konstruktor aufgerufen werden, der dann den 1. Konstruktor aufruft, um den `counter` richtig zu setzen. Die Instanzmethode `moveTo()` setzt den Stein um.

```
class Token {
  static int counter;   // Klassenvariable
  int xPos, yPos;       // Instanzvariablen

  Token() {             // 1. Konstruktor
    counter++;
  }
  Token(int x, int y) { // 2. Konstruktor
    this();             // Verkettung
    xPos = x;
```

```
    yPos = y;
  }
                          // Methode
  void moveTo(int x, int y) {
    xPos = x;
    yPos = y;
  }
}
```

Hier fehlen noch die für Java typischen Zugriffsmodifizierer, siehe 2.1.2. In einer main-Methode könnte man folgenden Code ausführen, um die Funktionalität der Klasse zu überprüfen.

```
// Array deklarieren
Token[] spielstein = new Token[3];
// Konstruktor mit new aufrufen,
// um Objekte zu erstellen
spielstein[0] = new Token(0, 0);
spielstein[1] = new Token(2, 3);
spielstein[2] = new Token(4, 6);
// Methode aufrufen, um
// Objekt zu veraendern
spielstein[1].moveTo(0, 3);
// Zugriff auf Klassenvariable
System.out.println(Token.counter);
```

Die Klassenvariable gehört zur ganzen Klasse. Daher wird sie nicht mit einem Objekt, sondern dem Klassennamen aufgerufen.

Sichtbarkeitstypen von Variablen

Klassen sind durch die Vererbung in einer Hierarchie angeordnet: Oberklassen, Unterklassen (dazu mehr in 2.2) Die Sichtbarkeits- und Zugriffsrechte von Variablen in dieser Hierarchie werden bei der Deklaration durch die Zugriffsmodifizierer **`public`**, **`protected`** und **`private`** festgelegt. Ohne Modifizierer gilt das Zugriffsrecht '*package*'. Alle Klassen, die in einem gemeinsamen Verzeichnis liegen, bilden ein Paket (*package*). Dies sollten sinnvolle zusammengehörige Klassen sein.

Wer darf?	private	package	protected	public
Die Klasse selbst, innere Klassen	ja	ja	ja	ja
andere Klassen im selben Paket	nein	ja	ja	ja
Unterklassen in anderem Paket	nein	nein	ja	ja
Sonstige Klassen	nein	nein	nein	ja

- Zusätzlich kann der Modifizierer **`final`** verwendet werden, um Konstanten zu definieren.
- Konstanten können in einem static Block der Klasse initialisiert werden

Mit Zugriffmodifizierern und einer hinzugefügten Konstante sieht die Beispielklasse nun so aus:

```
public class Token {
  private static final Shapetype shape = CIRCLE;  // Konstante
  private static int counter;
  private int xPos, yPos;

  public Token() {
    counter++;
  }
  public Token(int x, int y) {
    this();
    xPos = x;
    yPos = y;
  }

  public void moveTo(int x, int y) {
    xPos= x;
    yPos= y;
  }
}
```

Bemerkung zur Sichtbarkeit von Instanzvariablen

Instanzvariablen sollten aus folgenden Gründen nicht **`public`** definiert werden:

- Kontrollierter Zugriff (mehr Sicherheit)
- Flexibilität: Implementation der Datenstruktur kann geändert werden, ohne dass der Client-Code geändert werden muss.

Der Zugriff auf die Instanzvariablen erfolgt dann über sogenannte *getter* und *setter* Methoden:

```
private int xPos;
  // ...
  public int getXPos() {
    return xPos;
  }
  public void setXPos(int x) {
    xPos= x;
  }
```

Auf primitive Datentypen wird diese Konvention nicht immer angewendet. Hier muss der Nutzen gegen den zusätzlichen Aufwand abgewogen werden. Der zeitliche Aufwand ist sehr gering, wenn man die entsprechende Unterstützung einer IDE nutzt (bei IDEA: Cursor in dem Variablennamen positionieren und Alt+Insert drücken, oder Rechtsklick, dann *Generate*, dann *Getter* bzw. *Setter* auswählen).

2.1.3 Objekte

Objekte sind Instanzen von Klassen. Sie werden per **`new`** durch den Konstruktor erzeugt. Bei einem Aufruf von `new` wird Speicher reserviert, Werte initialisiert durch den Konstruktor und eine Referenz auf das Objekt zurückgegeben. Wenn `new` mehrfach aufgerufen wird, werden mehrere Objekte derselben Klasse erzeugt, jedes mit einer eigenen Identität, also einem eigenen Platz im Speicher (siehe Beispiel Klasse auf S. 13). Der Zugriff auf Objekte geschieht über Referenzen. Daher werden die nicht-primitiven Datentypen auch Referenztypen genannt.

Zuweisungen von Referenzentypen

Eine Zuweisung mit einem Referenztyp erzeugt keine Kopie des Objektes, sondern nur eine neue Referenz auf das bestehende Objekt. Dies ist ein wesentlicher Unterschied von primitiven Datentypen und Referenztypen. Wenn wir uns also folgenden Codeausschnitt anschauen:

```
Token stein1;
Token stein2;
stein1= new Token(1, 1);
stein2= new Token(2, 2);
stein2= stein1;
stein2.moveTo(2,6);
System.out.println("Stein1: " + stein1);
System.out.println("Stein2: " + stein2);
```

ergibt sich diese Ausgabe im Terminal:

```
> javac Token.java
> java Token
Stein1: (2, 6)
Stein2: (2, 6)
```

Warum ist das so? Wir haben zuerst zwei unterschiedliche Objekte erzeugt, beide haben eine eigene Identität und eine eigene Referenz, siehe Abbildung 2.2 (a)–(c). Dann haben wir aber `stein2` die Referenz auf `stein1` zugewiesen (d). Das heißt, egal welche von beiden Steinen wir bewegen (e), es wird immer auch

die gleiche Referenz zugegriffen und damit auf das gleiche Objekt, welches nun hinter beiden Variablen steckt. Auf das zweite Objekt dagegen haben wir den Zugriff komplett verloren (Garbage (d-e)).

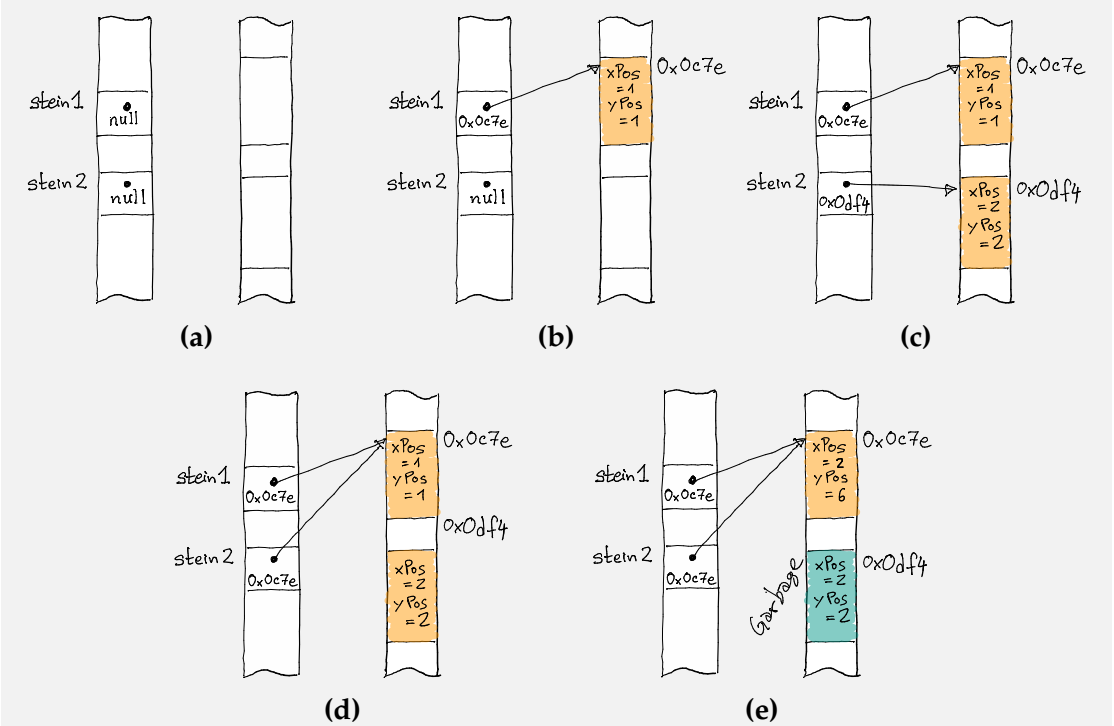


Abbildung 2.2 Funktionsweise von Referenztypen:(a) automatische Initialisierung mit `null`, **(b)** Initialisierung von `stein1`, **(c)** Initialisierung von `stein2`, **(d)** Zuweisung der Referenz, **(e)** Ausführung von `moveTo(2,6)`

Garbage Collection

Durch Zuweisungen von Referenztypen kann ein Objekt im Speicher "verloren" gehen, wenn keine Referenz auf das Objekt mehr existiert (siehe Seite 16). Solche Objekte werden als **Garbage** (Müll) bezeichnet, da auf sie nicht mehr zugegriffen werden kann. Bei Java läuft im Hintergrund eine **Speicherbereinigung** (*Garbage Collection*), die den Speicherplatz automatisch wieder freigibt.

Wie funktionierte die Ausgabe von Token?

Dies ist ein kleiner Vorgeschmack auf den Abschnitt 2.2. Klassen erben die Methoden ihrer Oberklasse (mehr darüber folgt), wobei jede Klasse eine Unterklasse von **`Object`** ist. Die `Object` Klasse implementiert die Methode `toString()` dadurch, dass Klassenname und Speicherplatz ausgegeben werden. Das war aber nicht die Ausgabe, die wir bekommen haben und zwar, weil wir die `toString()`-Methode *überschrieben* haben. D.h. wir haben eine eigene Implementierung der Methode in unserer Klasse, die aufgerufen wird, sobald wir unser Objekt ausgeben wollen. Die überschriebene Methode der Oberklasse ist aber immer noch über "**`super.`**" zugänglich. Wenn wir unsere `toString()`-Methode wieder mit dem Aufruf von `super.toString()` erweitern, hätten wir folgende Ausgabe:

```
public String toString() {     // Implementation von toString in Token Klasse
  String s = "(" + xPos + ", " + yPos + ")  " + super.toString();
  return s;
}
```

```
> javac Token.java
> java Token
Stein1: (2, 6) Token@12bb4df8
Stein2: (2, 6) Token@12bb4df8
```

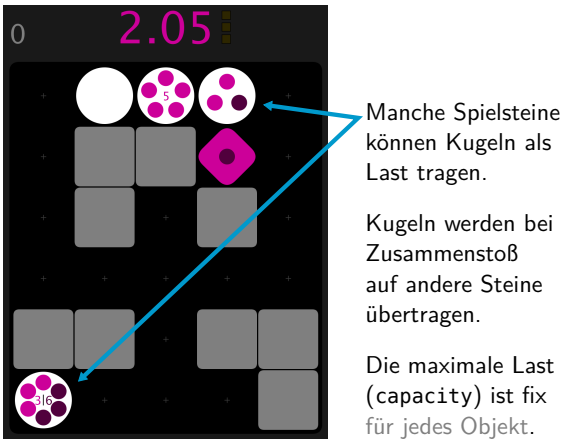
2.1.4 Syntaktische und Semantische Gleichheit von Objekten

Die Gleichheitsoperaton `x == y` von Java, prüft bei *Referenztypen*, ob `x` und `y` auf dieselbe Adresse im Speicher referenzieren. Dies nennt man auch **syntaktische Gleichheit**. Sind `x` und `y` unabhängig voneinander erzeugte Objekte (also mit unterschiedlichen Speicheradressen), so gilt `x != y`, selbst wenn alle Werte von `x` und `y` gleich sind. Um die **semantische Gleichheit** von Objekten zu prüfen, also ob `x` und `y` in allen relevanten Attributen gleich sind, gibt es die Methode **`equals()`**. Jede Klasse erbt eine `equals()`-Methode von `Object`, diese implementiert allerdings nur die syntaktische Gleichheit. Um eine semantische Gleichheit zu implementieren, muss `equals()` für eigene Klassen überschrieben werden.
IDEA kann eine `equals()`-Methode automatisch generieren. Das Überschreiben von `equals()` sollte mit dem entsprechenden Überschreiben von `hashCode()` einhergehen. Das wird zwar erst am Ende des Moduls besprochen, ist aber sehr relevant.

2.2 Klassenhierarchien und Vererbung (*inheritance*)

2.2.1 Vererbung

Mit dem Schlüsselwort **`extend`** in der Deklaration kann eine Klasse eine andere erweitern (*subclassing*). Die neue Klasse ist die **Unterklasse** oder abgeleitete Klasse, die andere wird **Oberklasse** oder Basisklasse genannt. Dabei werden alle sichtbaren Eigenschaften/ Methoden (`public` und `protected`) von der Oberklasse auf die Unterklasse übertragen, vererbt. Private (und `package` sichtbare) Eigenschaften werden nicht vererbt. Wie wir bereits gesehen haben, können geerbte Methoden, wie die `toString()`-Methode, überschrieben (*override*) werden. Die überschriebene Methode sollte dabei dieselbe Operation durchführen, nur spezialisiert für die Unterklasse. Den Vererbungsmechanismus von Instanzvariablen und -methoden an Unterklassen nennt man Implementierungsvererbung (*subclassing*). Ein anderer Vererbungsmechanismus folgt später. textbf[vorletzter Satz: Vererbungsmechanismus von Instanzvariablen und –methoden, was ist damit gemeint? Ggf. “Der hier beschriebene Vererbungsmechanismus für Instanzvariablen und -methoden an Unterklassen heißt...”, wenn das so korrekt ist.]
In der Unterklasse können weitere Variablen und Methoden definiert werden. Die Unterklasse ist also normalerweise ‘größer’ als die Oberklasse (mehr Daten und mehr Methoden). Sie spezialisiert die Oberklasse.
Wenn wir nun also unser Spiel erweitern wollen, mit Trägersteinen, welche Kugeln tragen und übertragen können, brauchen wir eine neue Klasse, die diese zusätzlichen Attribute und Methoden implementiert, aber trotzdem auch die bereits vorhandenen Attribute und Methoden nutzen kann.



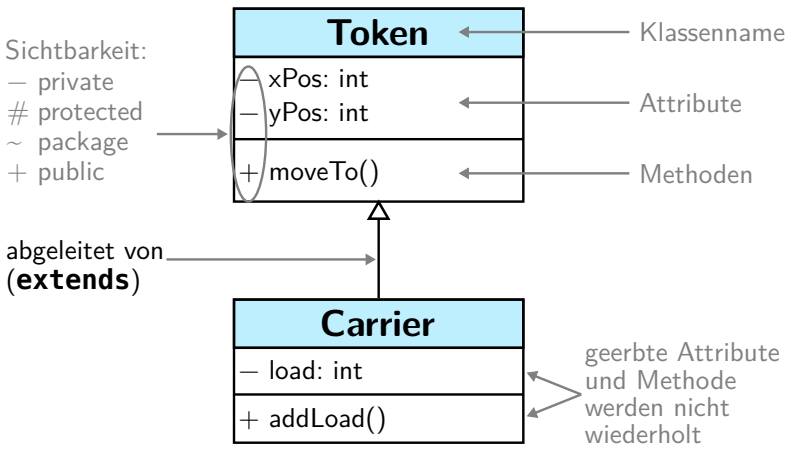
Hier bietet sich Vererbung an, damit die gemeinsamen Methoden, die in Token bereits implementiert sind, nicht neu implementiert werden müssen und wir damit Codedopplungen vermeiden. Für unser Beispiel sieht das dann so aus, wobei die Klasse Carrier von der Klasse Token erbt:

```java
public class Carrier extends Token {
  private int capacity;              // Instanzvar. zusätzlich zu den geerbten
  private int load;

  public Carrier(int capacity) {
    super();                         // Konstruktor der Oberklasse aufrufen
    this.capacity = capacity;
  }

  public void addLoad(int deltaLoad) {
    load += deltaLoad;
  }
}
```

```java
Carrier traeger = new Carrier(4);    // Träger mit Kapazität 4
traeger.moveTo(3, 4);                // geerbte Methode
traeger.addLoad(2);                  // eigene, neue Methode
```

2.2.2 Vereinheitlichte Modellierungssprache (*Unified Modeling Language*; UML)

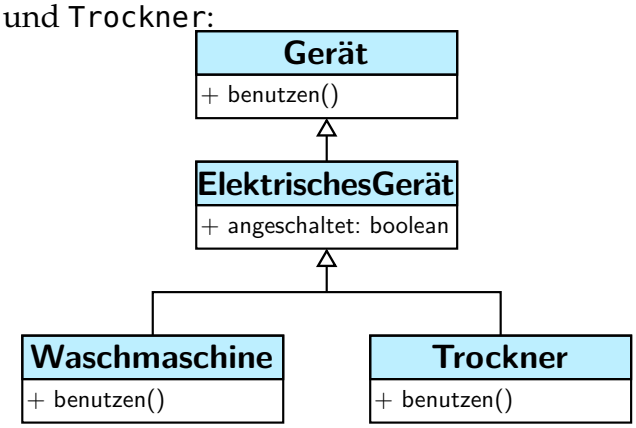
Klassenhierarchien können als Klassendiagramme in der **Vereinheitlichten Modellierungssprache** (*Unified Modeling Language*; UML) grafisch dargestellt werden. Dabei ist die Hierarchie von Oberklasse zur Unterklasse, von oben nach unten geordnet. Für unser Beispiel wäre das:



2.2.3 Vererbungshierarchie von Klassen

Eine Vererbungshierarchie kann viele verschiedene Ebenen haben und aus einer Klasse können beliebig viele Unterklassen abgeleitet werden. In diesem

Beispiel erbt die Klasse `Elektrogerät` von der Klasse `Gerät`, vererbt aber an `Waschmaschine` und `Trockner`:

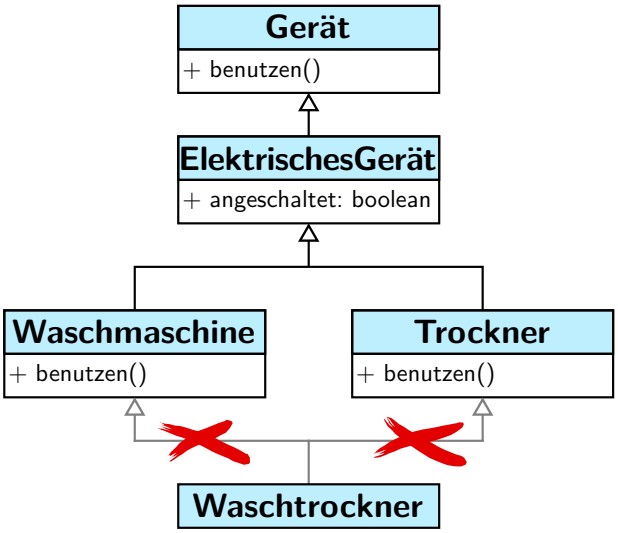


Wenn wir die genaue Implementierung der Methoden weglassen, könnte der dazugehörige Code so aussehen:

```
class Geraet {
  public void benutzen() {};
}

class ElektrischesGeraet extends Geraet {
  public boolean angeschaltet;
}

class Waschmaschine extends ElektrischesGeraet {
  public void benutzen() {[VERSION A: WASCHEN]};
}

class Trockner extends ElektrischesGeraet {
  public void benutzen() {[VERSION B: TROCKNEN]};
}

public class Haushalt {            // zum Testen
  public static void main(String[] args) {
    Waschmaschine w = new Waschmaschine();
    if (w.angeschaltet) {
      w.benutzen();
    }
  }
}
```

Allerdings würde das Programm mit oder ohne Implementierung nichts tun, denn die `Waschmaschine` ist ausgeschaltet. (`boolean` wird mit `false` initialisiert). Die Klassen könnten auch **`public`** deklariert werden. Dann müsste aller-

dings jede Klasse in einer eigenen Datei stehen, die denselben Namen wie die Klasse hat.
In Java gibt es *Einschränkungen* in den Kombinationsmöglichkeiten der Vererbung: Eine Klasse kann nur eine (direkte) Oberklasse besitzen. Das heißt folgendes Szenario (Mehrfachvererbung) ist nicht erlaubt:



Der Grund ist folgender: Würde eine Klasse von zwei Oberklassen abgeleitet, die eine Methode unterschiedlich implementieren, so wäre unklar welche Implementation vererbt wird. Welches `benutzen()` sollte `Waschtrockner` erben? Wir werden später Schnittstellen (*Interfaces*) kennenlernen, die einen Mechanismus für Mehrfachvererbung bieten, allerdings nur für Schnittstellenvorgaben, nicht für Implementierungen. Es gibt noch eine weitere Einschränkung: das Stichwort **`final`** in einer Klassendeklaration verbietet die Ableitung von Unterklassen. Auf diese Weise kann z. B. aus Sicherheitsgründen das Überschreiben von Methoden verhindert werden.

2.3 Polymorphismus

Polymorphismus (Vielgestaltigkeit) bezeichnet in der Biologie die Variation (individuelle Unterschiede) innerhalb einer Population. In der Programmierung ist es das Konzept, dass ein Bezeichner (Variable, Operator, Methode) Kontextabhängig unterschiedliche Datentypen annehmen kann. Durch die Möglichkeit der Vererbung und damit z.B. des Überschreibens von Methoden ist der Polymorphismus eines der Grundkonzepte von Java, auf welches wir auch bereits viele Male zurückgegriffen haben, ohne es zu benennen. Das betrifft insbesondere die erste Form des Polymorphismus, die wir uns jetzt anschauen: Im **Ad-hoc Polymorphismus** können Operatoren und Methoden mit unterschiedlichen

Signaturen überladen werden und abhängig von den Datentypen der Parameter unterschiedliches Verhalten haben.

- Überladene Operatoren: Der Operator + kann z.B. für unterschiedliche Datentypen angewendet werden.
- Überladene Methoden: Die Methode `Math.abs()` ist für unterschiedliche Eingabetypen (`int`, `long`, `float` und `double`) definiert.

Es gibt noch eine weitere Form des Ad-Hoc Polymorphismus, nämlich die Implizite Typumwandlung (*(ad-hoc) coertion polymorphism*). Dabei werden die Daten eines 'kleineren' Datentyps automatisch in einen 'größeren' Datentyp umgewandelt (*widening conversion*), wenn der Kontext es erfordert (z. B. `int` nach `double`). Die andere Richtung (*narrowing conversion*) geht in Java nur durch explizites Casting und zählt somit nicht zu Polymorphismus. Die zwei anderen großen Gruppen des Polymorphismus sind der parametrische Polymorphismus und der Subtyppolymorphismus. Im **parametrischen Polymorphismus** können Datentypen und Methoden Argumente variablen Typs haben. Das ist in Java mit Generics umgesetzt, welche in Abschnitt 2.4.2 besprochen werden. Im **Subtyp Polymorphismus** können Objekte den Typ ihrer Oberklasse annehmen. Eine Methode, die als Argument ein Objekt des Typs `T` erwartet, kann auch mit einem Objekt des Types `S` aufgerufen werden, wenn `S` eine Unterklasse von `T` ist:

```
class T { }             // Klasse T
class S extends T { }   // Subklasse S

class X {
  static void method(T t)
  {
    System.out.println("Hash of " +
           t + " is " + t.hashCode());
  }
}

class SubtypPolymorphismDemo {
    // ...
    T t = new T();
    S s = new S();
    // Aufruf nach Signatur für T Objekt:
    X.method(t);
    // Durch Polym. auch für S Objekt:
    X.method(s);
  }
}
```

Hier ist `X.method()` eine statische Methode. Das Prinzip gilt genauso, wenn es eine Objektmethode wäre.

2.4 Abstraktion

Im Prinzip der Abstraktion werden Konzept und Implementierung getrennt. Es wird gezeigt, was gemacht wird, ohne zu klären, wie es gemacht wird. Es wird keine Implementation vorgegeben, nur ein Umriss einer Funktion, ohne irgendwelche Details anzugeben. Man sagt auch, dass die Details versteckt werden.

2.4.1 Abstrakte Methoden und Klassen

Von einer Klasse, die als **`abstract`** deklariert wird, können keine Instanzen (Objekte) gebildet werden. Sie ist nur eine Modellierungsklasse für Unterklassen. Diese Klassen können neben normalen Methoden deren Implementierungen vererbt werden auch **abstrakte Methoden** besitzen: Bei abstrakten Methoden ist nur die Signatur vorgeben, ohne dass eine Implementation angegeben wird.

- Dies wird durch das Schlüsselwort **`abstract`** in der Methodendeklaration erreicht.
- Jede abgeleitete Klasse muss sich bei der Implementation an die vorgegebene Signatur halten.

Wenn eine abstrakte Klasse ausschließlich abstrakte Methoden hat, wird sie auch **rein** abstrakte Klasse gennant, andernfalls **partiell** abstrakte Klasse.
Bevor wir nun Abstrakte Datentypen betrachten, werden wir einmal Generics einführen.

2.4.2 Generics: Von einzelnen Spielsteinen zu einer Kollektion

Wir kehren wieder zu unserer Klasse Token und unserem Spielbeispiel zurück. Bisher haben wir über den Konstruktor einzelne Spielsteine erzeugt. Für das Spiel werden mehrere Spielsteine benötigt. Das könnte durch ein Array umgesetzt werden:

```
int nTokens = 3;
Token[] spielstein = new Token[nTokens];
for (int k = 0; k < nTokens; k++)
  spielstein[k] = new Token();
```

Störend bei dem Array ist, dass die Anzahl der Spielsteine festgelegt werden muss. Eine dynamische Veränderung (Löschen und Hinzufügen von Spielsteinen) ist nicht direkt möglich. Dies Problem ließe sich zwar auch mit Arrays lösen, aber aus Gründen, die im letzten Semester besprochen wurden, bevorzugen wir eine elegantere Lösung, z.B. mit einem Stapel basierend auf einer verketteten Liste. Im letzten Semester wurden Stapel/Listen von `int` Werten behandelt,

nun brauchen wir Stapel/Listen von Objekten, hier für die Klasse `Token`. Zur Erinnerung: Stapel funktionieren nach dem LIFO (last in first out) -Prinzip.

Damit generelle Datenstrukturen wie Stapel nicht für jeden Objekttyp neu programmiert werden müssen, gibt es in Java das Konzept der **generischen Typen** (*Generics*). Man kann Klassen definieren, bei denen der Typ einer Variablen selbst variabel ist, eine **Typvariable** oder formaler Typparameter (*formal type parameter*). Dazu schreibt man bei der Klassendeklaration die Typvariable in spitzen Klammern hinter den Klassennamen, z.B. `<E>`. Konvention ist dabei: einzelne Großbuchstaben, z.B. "T" für Typ (allgemein), "E" für Element, "K" für Schlüssel, "V" für Wert. Dann kann die Typvariable wie eine normale Typbezeichnung benutzt werden. Das heißt beispielsweise statt `Token`, einer spezifischen Typbezeichnung, wird `E`, eine Typvariable, benutzt. `E` ist hierbei nur ein Platzhalter, `E` ist keine Klasse und hat auch keinen Konstruktor, sie müssen mit einer spezifischen Typbezeichnung ersetzt und damit instanziiert werden.
Dabei gilt es zu beachten, dass Typparameter nur als Referenztypen instanziiert werden können. Daher gibt es für die primitiven Datentypen so genannte **Wrappertypen**, nämlich `Boolean`, `Integer`, `Short`, `Long`, `Double`, `Float`, `Byte`, `Character` für `boolean`, `int`, `short`, `long` usw. Das automatische *Casting* eines primitiven Typs auf den entsprechenden Referenztypen nennt man autoboxing, das Casting zurück unboxing

Nun schauen wir uns einmal die Implementierung eines Stapels mit und ohne Generics an. Zunächst **ohne Generics** (nur für `Token`):

```
public class TokenStack
{
  private Node head;

  private class Node {// innere Klasse
    Token item;
    Node next;
  }
  public void push(Token item) {
    Node tmp = head;
    head = new Node();
    head.item = item;
    head.next = tmp;
  }
  // ... other methods ...
}
```

Erzeugung eines Stapels für `Token` sieht dann so aus:

```
TokenStack tokens = new TokenStack();
```

Dies ist aber *nur* für die Klasse `Token` verwendbar. Wenn man **Generics** benutzt, ist die Klasse `Stack` allgemein verwendbar:

```
public class Stack<E>    // generics
{
  private Node head;

  private class Node {
    E item;             // E als Typ
    Node next;
  }
  public void push(E item) {
    Node tmp = head;
    head = new Node();
    head.item = item;
    head.next = tmp;
  }
  // ... other methods ...
}
```

'Allgemein verwendbar' heißt, dass man statt der Typvariablen `E` jeden beliebigen Referenztyp einfügen kann, also auch `Token`:

```
Stack<Token> tokens = new Stack<Token>();
```

2.4.3 Datenabstraktion

Ein Datentyp ist die Kombination aus einer Wertemenge und Operationen. Bei dem primitiven Datentyp `int` z. B. sind das die ganzen Zahlen von -2^{31} bis $2^{31}-1$ und die Rechenoperationen, Vergleichsoperationen etc. **Abstrakte Datentypen** (*abstract data type*; ADT) sind Datentypen und damit ebenfalls eine Kombination aus Wertemengen und Operationen. Aber der Zugriff auf (Teile der) Wertemenge erfolgt nur über festgelegte Operatoren. Diese Festlegung ist die Spezifikation, mit der wir uns im nächten Abschnitt befassen werden. Das führt dazu, dass die Nutzer- und Implementationssicht getrennt sind (man kann den Datentyp verwenden, ohne die Implementierung zu kennen). ADTs sind demnach durch ihre Semantik (Perspektive der Benutzerin; *'was?'*) gekennzeichnet. Daraus ergeben sich ein paar wichtige Vorteile:

- Kapselung (*encapsulation*): Nutzer- und Implementationssicht (was und wie) sind getrennt. Der client-code kann sich auf die ADT Beschreibung verlassen, und braucht keine Kenntnis über die Implementation.
- Modularität (*modularity*): Implementationen können verbessert werden, ohne dass der *client-code* angepasst werden muss.

- Geschütztheit (*integrity*): Nur diejenigen Daten und Methoden sind zugreifbar, die durch die Spezifikation dafür vorgesehen sind.

ADTs werden in Java durch das Klassenkonzept unterstützt, wie folgende kleine Code-Abschnitte zeigen, in denen jeweils die Entwickler- und die Anwenderperspektive aufgezeigt wird:

Entwickler (Implementation der ADT)

```
public class XYZ
{ ...
  public machWas()
  ...
}
```

Anwender (*client-code*, nutzt ADT)

```
public static void main()
{
  XYZ xyz = new XYZ();
  xyz.machWas();
}
```

Spezifikation von ADTs

ADTs realisieren eine Art Vertrag zwischen Nutzer (*client-code*) und Entwickler (Implementierung). Sie können auf unterschiedliche Weise spezifiziert werden.

- Beschreibung der **Schnittstelle für Anwendungsprogrammierung** (*Applications Programming Interface*; API)
- Implementierung von **Schnittstellen** (*interfaces*) in Java.

Die API dient zur Dokumentation, während das **`interface`** als Technik in der Programmierung benutzt wird. Die Varianten sind ansonsten gleichwertig. Beide sind hier einmal vorgestellt am Beispiel eines Stapels.

API eines Stapels (LIFO)

`public class Stack<E>`

	`Stack()`	Erzeugt leeren Stapel
`void`	`push(E item)`	Fügt ein Element hinzu.
`E`	`pop()`	Entfernt das letzte Element.
`boolean`	`isEmpty()`	Prüft, ob der Stapel leer ist.
`int`	`size()`	Gibt Anzahl der Elemente zurück.

```
interface Stack<E> {
  void push(E item);
  E pop();
  boolean isEmpty();
  int size();
}
```

2.4.4 Schnittstellen

Schnittstellen haben weder Datenfelder noch Konstruktoren. Sie können allerdings Konstanten definieren. Bei Methoden wird nur die Signatur definiert. Implementationen sind in Schnittstellen generell nicht möglich, es handelt sich also immer um abstrakte Methoden.
Eine Klasse „erbt“von einer Schnittstelle mit dem Schlüsselwort **`implements`**, was auch als Schnittstellenvererbung (*subtyping*) bezeichnet wird. In diesem Fall muss die Klasse alle Methoden der Schnittstelle Signatur-konform implementieren. Viele Klassen können dieselbe Schnittstelle implementieren. Aber im Unterschied zur Vererbung von Klassen und abstrakten Klassen (*subclassing*), kann eine Klasse mehrere Schnittstellen implementieren. Es können auch Schnittstellen von Schnittstellen abgeleitet werden, mit dem Schlüsselwort `extends`.

KAPITEL 3

Datenstrukturen

3.1 Die Schnittstellen `Iterator` und `Iterable`

Die Schnittstelle **`Iterable`** (im Paket `java.util`) besagt lediglich, dass es eine Methode `iterator()` gibt, die einen `Iterator` zurückgibt.

```
public interface Iterable<E>
{
  Iterator<E> iterator();
}
```

Was ein `Iterator` leisten muss, ist in der Schnittstelle **`Iterator`** festgelegt:

```
public interface Iterator<E>
{
  boolean hasNext();  // Returns true if the iteration has more elements.
  E next();           // Returns the next element in the iteration.
  void remove();      // Removes the last element returned by this iterator.
}
```

Das klingt zunächst etwas kompliziert, ist aber in der Anwendung sehr praktisch.
Wenn eine Klasse die Schnittstelle `Iterable` implementiert, kann man mit einer **`for`** Schleife einfach über die Elemente iterieren.
Lautet also die Deklaration unserer `Stack` Klasse (siehe 2.4.2, bzw. 3.1)

```
public class Stack<E> implements Iterable
```

dann ist dadurch festgelegt, dass wir folgendermaßen elegant und einfach über unsere Tokenliste iterieren können:

```
Stack<Token> tokens = new Stack<>();

// Erzeugung einiger Exemplare:
tokens.push(new Token(0,0));
tokens.push(new Token(2,3));
tokens.push(new Token(4,6));

// angenommen Token hat eine Methode 'render'
for (Token token : tokens) {
  token.render();
}
```

Nebenbemerkung: Arrays implementieren `Iterable`, siehe Abschnitt 1.2.2.
Mit dieser praktischen Erweiterung sieht also die API für einen Stapel so aus:

API eines Stapels (LIFO)

public class Stack<E> implements Iterable<E>

	`Stack()`	Erzeugt leeren Stapel
void	`push(E item)`	Fügt ein Element hinzu.
E	`pop()`	Entfernt das letzte Element.
boolean	`isEmpty()`	Prüft, ob der Stapel leer ist.
int	`size()`	Gibt Anzahl der Elemente zurück.

Da die Klasse die `Iterable` Schnittstelle erbt, brauchen die geerbten Methoden nicht explizit in der API erwähnt zu werden, sie müssen aber durchaus implementiert werden. Die Implementierung der Klasse Stapel wird hier mit der Implementierung der Methode `iterator()` und der Klasse `ListIterator` diesbezüglch erweitert:

Listing 3.1 Eine einfache Implementation eines Stapels. Überprüfungen fehlen, siehe Hinweis im Text.

```
// Iterator aus java.util importieren:
import java.util.Iterator;

public class Stack<E> implements Iterable<E> {
  private Node head;
  private int N;

  private class Node {
    E item;
    Node next;
  }
```

```
  public int size() {
    return N;
  }

  public boolean isEmpty() {
    return N == 0;
  }

  public void push(E item) {
    Node tmp = head;
    head = new Node();
    head.item = item;
    head.next = tmp;
    N++;
  }

  public E pop() {
    E item = head.item;
    head = head.next;
    N--;
    return item;
  }

  public Iterator<E> iterator() {
    return new ListIterator();
  }

  public class ListIterator implements Iterator<E> {
    private Node current = head;

    public boolean hasNext() { return current != null; }
    public void remove()     { }  // kein remove
    public E next() {
      E item = current.item;
      current = current.next;
      return item;
    }
  }
}
```

Bemerkung zu der Implementation

Es ist dennoch eine Minimalversion ohne essentielle Überprüfungen, z.B. am Anfang von `pop()` ob der Stapel leer ist. Außerdem wurde der Konstruktor weggelassen, da der default ausreicht (`head` wird mit `null` und `N` mit `0` initialisiert). Die Methode `remove()` des Iterators ist leer gelassen. Das ist erlaubt, `remove()` muss nur formal implementiert werden, aufgrund der Vorgabe durch das

Interface. Das Beispiel dient trotzdem nur der Illustrierung der Interfaces `Iterable` und `Iterator`. Eine saubere und vollständige Implementation von Stapeln findet man hier: https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/Stack.java.html

3.2 Collections in Java

Eine ***Collection*** ist eine Datenstruktur, die Speicherung von und Zugriff auf viele Objekte gleichen Typs erlaubt (Alternativen zu einfachen Arrays). Ein Beispiel sind die Stapel, die wir uns bereits angeschaut haben. In Java werden viele Varianten von *Collections* zur Verfügung gestellt. Für diese Veranstaltung sind **`LinkedList`** als Stack und Queue, **`PriorityQueue`** und in den letzten Kapiteln `HashMap` und `HashSet` wichtig. Alle Klassen sind dabei von dem Interface `Collection` abgeleitet und weiterhin eingeteilt in die Unter-Schnittstellen `List`, `Queue` und `Set`. Jede dieser Klassen stellt eine Vielzahl von Methoden zur Verfügung, mitunter auch Methoden, die in der Datenstruktur im klassischen Sinne nicht vorkommen würden. Dies macht die *Collections* sehr praktisch, birgt aber die folgende Gefahr: Einige Klassen bieten auch Methoden an, die in der Datenstruktur nicht effizient sind. Ein Beispiel hierfür ist der Stapel. Die typischen Methoden eines Stapels (`push()`, `pop()`, `peek()`, `isEmpty()`) haben eine konstante Laufzeit. Daher könnte man dies für alle Methoden eines `Stack` erwarten. Aber in der Java Collection `Stack` gibt es z.B. eine `contains()` Methode mit linearer Laufzeit. Ein weiteres Beispiel ist die Prioritätenwarteschlange. Dafür erwartet man eine konstante (`peek()`, `size()`) oder logarithmische Laufzeit (`add()`, `poll()`). In den Java Collections hat `PriorityQueue` aber auch Methoden `contains(Object)` und `remove(Object)` mit linearer Laufzeit. Daher werden in der Vorlesung 'kleinere' Varianten eingeführt, in denen die Funktionalität auf die eigentlichen und effizienten Methoden eingegrenzt ist (siehe auch `Queue` und `Bag` in Abschnitten 3.3 und auf Seite 34). Bei der Benutzung zusätzlicher Methoden in den Java Collections sollte immer auf deren Laufzeit geachtet werden (siehe auch Seite 34ff). In diesem Sinne werden im Folgenden noch Klassen für Warteschlangen und Multimengen eingeführt.

3.3 Warteschlangen (FIFO)

Wir stellen in diesem Abschnitt eine API einer Warteschlange vor. Warteschlangen sind abstrakte Datentypen, die nach dem *first in first out*- Prinzip funktionieren. Die folgende API und Implementierung einer Queue implementieren diese im engen Sinne einer Warteschlange als Klasse. In Java Collections ist `Queue` eine Schnittstelle (`interface`), welche zum Beispiel duch `LinkedList` implementiert wird.

API einer Warteschlange (FIFO)

`public class Queue<E> implements Iterable<E>`

	`Queue()`	Erzeugt leere Warteschlange
`void`	`enqueue(E item)`	Fügt ein Element hinzu.
`E`	`dequeue()`	Entfernt das erste Element.
`boolean`	`isEmpty()`	Prüft, ob die Warteschlange leer ist.
`int`	`size()`	Gibt Anzahl der Elemente zurück.

Da die Klasse die `Iterable` Schnittstelle erbt, brauchen die geerbten Methoden, wie bereits bei Stapeln beschrieben, nicht explizit in der API erwähnt zu werden.

Implementation einer Warteschlange

```
import java.util.Iterator;
public class Queue<E> implements Iterable<E>
{
  private Node head;
  private Node tail;
  private int N;

  private class Node
  { E item;
    Node next;
  }

  public int size()        { return N; }
  public boolean isEmpty() { return N == 0; }

  public E dequeue()
  { // Entfernt das Element vom Anfang der Schlange
    E item = head.item;
    head = head.next;
    if (--N == 0)          // Abfrage auf leer fehlt, siehe Bemerkung unten
      tail = null;
    return item;
  }

  public void enqueue(E item)
  { // Fügt Element an das Ende der Schlange
    Node wastail = tail;
    tail = new Node();
    tail.item = item;
    if (N++ == 0)
      head = tail;
    else
```

```
        wastail.next = tail;
    }

  // Methode iterator() und innere Klasse ListIterator wie beim Stapel
  }
```

Achtung: Es gelten die Hinweise wie bei der Stapel Implementation, siehe 3.1. Insbesondere fehlen essentielle Überprüfungen, z.B. am Anfang von `dequeue()`, ob die Schlange leer ist.

API und Implementation einer Multimenge

Wie Stapel und Warteschlangen, werden Multimengen auch benutzt um Objekte zu sammeln. Multimengen unterstützen aber explizit nicht das Entfernen von Elementen. Die Klasse beschränkt sich strikt auf das Sammeln und Iterieren über Elemente und ist sinnvoll zu nutzen, wenn man nicht mehr von einer Datenstruktur braucht, denn dann ist dies eine elegante Lösung.

API einer Multimenge

public class Bag<E> implements Iterable<E>

	`Bag()`	Erzeugt leere Multimenge
void	`add(E item)`	Fügt ein Element hinzu.
boolean	`isEmpty()`	Prüft, ob die Multimenge leer ist.
int	`size()`	Gibt Anzahl der Elemente zurück.

Der Zugriff auf Elemente erfolgt nur über den Iterator. Die Implementation kann exakt von `Stapel` übernommen werden, wobei `pull()` weggelassen und `push()` in `add()` umbenannt wird. Daher könnte natürlich auch ein `Stapel` an Stelle eines `Bag` benutzt werden.

3.4 Laufzeiten bei einfachen Java Datenstrukturen

Bei der Implementierung von Algorithmen ist bei der Auswahl der Datenstrukturen die Laufzeit der Operationen zu beachten. In diesem Kapitel werden die Laufzeiten für einige häufig benutzte Datenstrukturen / Methoden aufgelistet. Für andere Datenstrukturen oder Methoden ist die Java Dokumentation zu konsultieren.
Leider sind die Laufzeiten in der Java Dokumentation nicht immer angegeben. In diesem Fall sollte man in die Quellen schauen. Die weiter unten angegebenen Laufzeiten von Methoden der `LinkedList` sind der folgenden Seite entnommen: `http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/LinkedList.java`.

Laufzeiten von `Stack` und `Queue`

Laufzeiten für die in der Java Einführung angegebenen Implementationen von `Stack`, `Queue` und `Bag`:

Stack (*worst case*)	
`push()`	$O(1)$
`pop()`	$O(1)$

Queue (*worst case*)	
`enqueue()`	$O(1)$
`dequeue()`	$O(1)$

Bag (*worst case*)	
`add()`	$O(1)$

textbf[Diese Referenz ist in einem anderen Skript. Man kann es zwar anklicken, in meinem Programm führte es aber zu einer Seite innerhalb des Java-Skripts.] Der Zusatz *worst case* bei der Laufzeit ist insbesondere eine Abgrenzung zu einer *amortisierten Laufzeit*, siehe Seite 12 des AlgoDat Skriptes, bei der die Operationen manchmal eine längere Laufzeit haben, z. B. wenn nach einer gewissen Anzahl von Einfügungen ein Array vergrößert und der Inhalt kopiert werden muss.
Bei den Java Klassen `Stack` und `Queue` ist zu beachten, dass diese auch Methoden implementieren, die keine konstante sondern eine lineare Laufzeit besitzen, wie `remove()`.

Laufzeiten für ausgewählte Java *Collections*

Die `LinkedList` ist eine beidseitige Warteschlange (*double ended queue*, kurz deque). Daher unterstützt sie die Funktionalität einer Warteschlange und eines Stacks. Zu jedem `xxxFirst()` gibt es ein entsprechendes `xxxLast()` mit derselben Laufzeit. Weitere Methoden, die hier nicht aufgeführt sind, können in der Java Dokumentation recherchiert werden.

LinkedList (*worst case*)	
`addFirst(E e)`	$O(1)$
`contains(Object o)`	$O(N)$
`get(int index)`	$O(N)$
`indexOf(Object o)`	$O(N)$
`peekFirst()`	$O(1)$
`pollFirst()`	$O(1)$
`removeFirst()`	$O(1)$
`remove(int index)`	$O(N)$
`remove(Object o)`	$O(N)$
`set(int index, E e)`	$O(N)$

Altertnativ gibt es noch die Varianten `ArrayList` und `ArrayDeque`, die ähnliche Funktionalität mit Laufzeiten in derselben Wachstumsordnung zur Verfügung stellen. `ArrayList` bietet den Vorteil der direkten Indizierung und `ArrayDeque`

ist ansonsten die schnellste Variante innerhalb der jeweiligen Wachstumsordnung. Allerdings haben Einfügungen bei beiden *Array* Varianten nur *amortisiert* konstante Laufzeit, während `LinkedList` *worst case* konstante Laufzeit besitzt. `ArrayList` hat für Einfügung außer am Ende sogar nur lineare Laufzeit.

3.5 Die Schnittstellen `Comparable` und `Comparator`

Neben `Iterable` und `Iterator` gibt es zwei weitere wichtige Schnittstellen: **`Comparable`** und **`Comparator`**. Diese sind allerdings kein zusammengehöriges Paar, sondern zwei Varianten für unterschiedliche Fälle. Klassen sollten eine dieser Schnittstellen implementieren, wenn Methoden benutzt werden sollen, die auf einer Ordnung basieren, z.B. Sortieren. Die Schnittstelle `Comparable` befindet sich in dem Paket `java.lang` und `Comparator` in `java.util`. Die Schnittstelle `Comparable` sollte implementiert werden, wenn es nur eine sinnvolle Ordnung auf den Objekten der Klasse gibt (genannt 'natürliche Ordnung').

Die Schnittstelle Comparable

public interface Comparable<T>

`int`	`compareTo(T o)`	vergleicht dieses Objekt mit Objekt `o` bezüglich einer Ordnung

Wenn es alternative Möglichkeiten gibt, kann die Klasse mehrere Ordnungen über die `Comparator` Schnittstelle definieren. So kann z.B. eine Sortierfunktion mit unterschiedlichen Ordnungen aufgerufen werden.

Die Schnittstelle Comparator

public interface Comparator<T>

`int`	`compare(T o1, T o2)`	vergleicht die gegeben Objekte bezüglich einer Ordnung
	`...`	weitere Methoden, Implementation optional

Salopp gesprochen soll der Rückgabewert von `v.compareTo(w)` bzw. `compare(v, w)` eine Ordnung auf den Objekten von `T` definieren: Er ist

- negativ, falls `v` 'kleiner' als `w` ist,
- `0`, falls `v` gleich `w`, bzw. 'gleichwertig' zu `w` ist und
- positiv, falls `v` 'größer' als `w` ist.

Um dies genauer zu formulieren, gehen wir davon aus, dass sich die Rückgabewerte auf `-1`, `0` und `1` beschränken (sonst müsste man im Folgenden an einigen Stellen die Signum Funktion einfügen). Die Java Dokumentation fordert folgende Bedingungen (am Beispiel von `compare()`):

- `compare(v, w) == -compare(w, v)`

- Aus `compare(u, v) < 0` und `compare(v, w) < 0` folgt `compare(u, w) < 0`.
- Aus `compare(u, v) == 0` folgt `compare(u, w) == compare(v, w)` für alle `w`.

Diese Eigenschaften implizieren:

- `compare(v, w) = 0` definiert eine Äquivalenzrelation (wie $=$): reflexiv, symmetrisch und transitiv
- `compare(v, w) <= 0` definiert eine totale Ordnung (wie $\leq$): reflexiv, antisymmetrisch, transitiv und linear
- `compare(v, w) < 0` definiert eine strikte Ordnung (wie $<$): reflexiv und trichotom

Dabei beziehen sich die beiden Ordnungen auf die von obiger Äquivalenzrelation induzierten Äquivalenzklassen. Zusätzlich sollte die Äquivalenzrelation konsistent mit `equals()` sein, d. h. `v.compareTo(w) == 0` sollte denselben Wahrheitswert wie `v.equals(w)` haben. Die Java Dokumentation sieht vor, dass man gegen diese Regel verstoßen darf, wenn man dies klar in der Beschreibung vermerkt ("*Note: this class has a natural ordering that is inconsistent with equals.*" bzw. "*Note: this comparator imposes orderings that are inconsistent with equals.*").

Implementationsbeispiel Comparator

Wir betrachten nun folgende Beispielklasse `Person`:

```
import java.util.ArrayList;

public class Person {
  protected String name;
  protected int age;
  protected double height;

  public Person(String name, int age, double height) {
    this.name = name;
    this.age = age;
    this.height = height;
  }

  public String toString() {
    return "(" + name + ", " + age + "y, " + height + "cm)";
  }
}
```

Für diese Klasse gibt es mehrere Möglichkeiten, nach denen verglichen werden könnte, nämlich nach Namen, nach Alter und nach Größe. Dafür können wir

drei Klassen schreiben, die jeweils `Comparator` implementieren. Da sie den Vorgaben der Schnittstelle folgen, muss jeweils die Methode `compare` implementiert werden. Hierbei nutzen wir aus, dass die Klasse `Integer`, `String` und `Double` bereits eine `compare`-Methode implementiert haben.

```
import java.util.Comparator;
/* Die Comparator könnten auch als anonyme Klassen
im Aufruf implementiert werden.
 siehe Beispiele im git in Material/Code/Lecture03
*/
public class SortByAge implements Comparator<Person> {
  public int compare(Person person1, Person person2) {
    return Integer.compare(person1.age, person2.age);
  }
}

public class SortByName implements Comparator<Person> {
  public int compare(Person person1, Person person2) {
    return person1.name.compareTo(person2.name);
  }
}

public class SortByHeight implements Comparator<Person> {
  public int compare(Person person1, Person person2) {
    return Double.compare(person1.height, person2.height);
  }
}
```

Testen können wir die Funktionalität dann beispielsweise in der `main`-Methode der Klasse `Person`.

```
// main() Methode der Klasse 'Person'

  public static void main(String[] args) {
    ArrayList<Person> personen = new ArrayList<>();
    personen.add(new Person("Peter", 80, 175.8));
    personen.add(new Person("Paul",  81, 178.7));
    personen.add(new Person("Mary",  82, 177.2));

    personen.sort(new SortByAge());
    System.out.println("Sorted by Age:\n" + personen);

    personen.sort(new SortByName());
    System.out.println("Sorted by Name:\n" + personen);

    personen.sort(new SortByHeight());
    System.out.println("Sorted by Height:\n" + personen);

    personen.sort(new SortByHeight().reversed());
```

```
        System.out.println("Descending by Height:\n" + personen);
    }
```

Dann bekommen wir folgende Ausgabe:

```
Sorted by Age:
[(Peter, 80y, 175.8cm), (Paul, 81y, 178.7cm), (Mary, 82y, 177.2cm)]
Sorted by Name:
[(Mary, 82y, 177.2cm), (Paul, 81y, 178.7cm), (Peter, 80y, 175.8cm)]
Sorted by Height:
[(Peter, 80y, 175.8cm), (Mary, 82y, 177.2cm), (Paul, 81y, 178.7cm)]
Descending by Height:
[(Paul, 81y, 178.7cm), (Mary, 82y, 177.2cm), (Peter, 80y, 175.8cm)]
```

Implementationsbeispiel Comparable

Alternativ kann man sich auch auf eine Vergleichsmethode festlegen und die Klasse `Person` selbst als Implementation der Schnittstelle `Comparable` schreiben.

```
public class Person implements Comparable<Person> {
    protected String name;
    protected int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() {
      return "(" + name + ", " + age + "y)";
    }

    public int compareTo(Person other) {
      return Integer.compare(this.age, other.age);
    }

    public static void main(String[] args) {
        ArrayList<Person> personen = new ArrayList<>();
        personen.add(new Person("Mary", 82));
        personen.add(new Person("Peter",80));
        personen.add(new Person("Paul", 81));

        personen.sort(null);        // null: nutze compareTo()
        System.out.println("Sorted by Age:\n" + personen);
        Collections.sort(personen);     // Alternative
        System.out.println("Sorted by Age:\n" + personen);
        Collections.sort(personen, Collections.reverseOrder());
        System.out.println("Descending by Age:\n" + personen);
    }
```

```
}
```

KAPITEL 4

Gute Praxis

4.1 Umgang mit Fehlern und Ausnahmen

Wir werden uns nun mit Fehlern beschäftigen: Angefangen bei der Vermeidung von Fehlern über das Abfangen und Umgehen von Fehlern durch die Implementierung bis hin zum Finden von Fehlern durch Debugging.

4.1.1 Debugging

Bei der Korrektur von insbesondere logischen Fehlern ist ein Debugger eine immense Hilfestellung. Im Debugger kann das Programm dann Zeilenweise ausgeführt und Variableninhalte inspiziert werden.
Wir betrachten nun im speziellen den Debugger von *IDEA*. Falls ein Programm mit einer Exception (siehe Abschnitt 4.1.2) abbricht, kann einfach der Debugger gestartet werden, und er wird bei der verursachenden Zeile stehen bleiben. Läuft das Programm durch, liefert aber nicht das gewünschte Ergebnis, setzt man einen *Breakpoint* und startet dann den Debugger. In IDEA kann man dann den Programmablauf mit F8, F7, Shift+F8 und Alt+F9 steuern sowie mit weiteren Shortcuts oder Schaltknöpfen im Debugger Fenster. Im '*Variables*' Fenster des Debuggers kann der Inhalt komplexerer Variablen durch Klicken auf das Dreieck aufgeklappt werden. Durch Klicken auf '+' kann man arithmetische Ausdrücke als *watch* hinzufügen.

> Um das Debugging praktisch zu veranschaulichen und den Nutzen zu demonstrieren, haben wir zwei Videos erstellt, welche unter dem ISIS Link aufgerufen und heruntergeladen werden können.

Fehlervermeidung

Es gibt unterschiedliche Methoden, um Fehler und unerwartetes Verhalten zu vermeiden. Zum einen sind APIs und Dokumentation bei der Verwendung von vorhandenen Implementationen, wie abstrakten Datentypen, hilfreich bei der Vermeidung von Fehlern. Schnittstellen und abstrakte Methoden machen gewisse Vorgaben, die ebenfalls zur Vermeidung von Fehlern beitragen können. Das Testen von Code ist eine weitere Strategie zur Vermeidung von Fehlern, speziell *JUnit Tests* werden dafür häufig verwendet. Mit ihnen werden wir uns im Laufe dieses Abschnittes noch beschäftigen. Das als nächstes vorgestellte Modell *Design-by-Contract* ergänzt diese Verfahren.

Programmiermodel *Design-by-Contract*

Gemäß dem Programmiermodel *Design-by-Contract* (dt. Entwurf gemäß Vertrag) sieht vor, dass für jede Komponente einer Software präzise und nachvollziehbare Bedingungen formuliert werden, die diese Komponente erfüllen muss. Die Idee basiert auf einem Vertrag zwischen den einzelnen Softwarekomponenten, sodass wenn eine Methode aufgerufen wird, die aufrufende Methode genau weiß, was zu erwarten ist. Dabei wird für jede Methode definiert:

- Vorbedingungen (*preconditions*): Bedingungen, die der Client beim Aufruf einhalten muss.
- Nachbedingungen (*postconditions*): Bedingungen, die die Implementation bezüglich der Rückgabe der Methode zusichert.
- Nebeneffekte (*side effects*): Zustandsänderungen, die die Methode verursachen kann.

Wenn diese Bedingungen spezifisch formuliert wurden, ist die Zusammenarbeit der Komponenten klar definiert und damit weniger fehleranfällig.
Im Minimalfall kann das *Design by Contract* in der API definiert werden. Darüber hinaus sollten die Bedingungen über *exceptions* und *assertions* geprüft werden.

4.1.2 Ausnahmenbehandlung

Trotz aller Bemühungen, Fehler zu vermeiden, lassen sich viele Fehler erst zur Laufzeit feststellen.
In C zeigen Funktionen Fehler, die z.B. durch den Aufruf mit unzulässigen Parametern verursacht werden, durch einen speziellen Rückgabewert an, z.B. `-1`, `0` oder `NULL`. Damit liegt die Verantwortung, Fehler zu behandeln, ganz in der Verantwortung der Programmierenden.
In Java können Methoden bei Ausnahmen und Fehlern eine **`Exception`** auslösen, bzw. 'werfen' (**`throw`**). Die aufrufende Methode kann Exceptions mit **`try`** ...

catch Anweisungen abfangen und entsprechend reagieren, oder nach oben weiterleiten. Nur Exceptions, die nirgends abgefangen werden, führen zu einem Programmabbruch, dann aber mit entsprechender Fehlermeldung. In den Java Bibliotheken gibt es verschiedene Exceptions. Bei `catch` gibt man als Argument an, welche Exception Art(en) abgefangen werden soll(en). Man kann auch eigene Exceptions definieren, die von der Klasse `Exception` aus `java.lang` abgeleitet werden müssen.
Wir schauen uns nun erstmal ein Beispiel an, in dem keine Ausnahmen abgefangen werden. Das folgende Programm nimmt eine Eingabe in einem Dialogfenster entgegen und wandelt diese in eine `int` Zahl um.

```
public class readNumberNaiv
{
  public static void getInteger() {
   // Eingabedialog öffnen:
   String str = javax.swing.JOptionPane.showInputDialog("Zahl eingeben: ");
   int number = Integer.parseInt(str);
   System.out.println("Die Zahl " + number + " war lecker!");
  }

  public static void main(String[] args) {
    getInteger();
  }
}
```

Falls die Eingabe nicht in einen `int` umgewandelt werden kann, bricht die Methode `Integer.parseInt()` mit einer `NumberFormatException` ab. Während mit einem `try`... `catch`-Block das Programm nicht abbricht und stattdessen den Fehler abfängt und mit ihm umgeht:

```
public class readNumber
{
  public static void getInteger()
  {
    int number = 0;
    String str = "";
    while (true) {
      try { // Fehler im try-Block führen nicht zum Programmabbruch
                // sondern zur Ausführung des catch-Blocks
        str = javax.swing.JOptionPane.showInputDialog("Zahl eingeben: ");
        number = Integer.parseInt(str);
        break; // while Schleife verlassen
      } catch (NumberFormatException e) {// Fehlerbehandlung
        System.err.println("'" + str + "' schmeckt mir nicht.");
      }
    }
    System.out.println("Die Zahl " + number + " war lecker!");
```

```
    }

    public static void main(String[] args) { ... }   // wie zuvor
}
```

Wie bereits erwähnt können Exceptions auch von einer Methode geworfen werden, sodass ein Programm nicht vollständig durchgeführt wird, wenn bereits ein Fehler gefunden wurde. Dazu ergänzen wir unser Programm mit der Bedingung, dass eine Zahl in einem bestimmten Intervall einzugeben ist. Wenn diese Bedingung nicht erfüllt ist, werfen wir eine `InputMismatchException`.

```
public static void getInteger(int low, int high)
// Zahl zwischen low und high
{
 int number = 0;
 String str = "";
 while (true) {
    try {
      String msg = "Zahl eingeben ("  low + "-" + high + "): ";
      str = javax.swing.JOptionPane.showInputDialog(msg);
      number = Integer.parseInt(str);
      System.out.println("n = " + number);
      break;
    }
    catch (NumberFormatException e) {
      System.err.println("'" + str + "' schmeckt mir nicht.");
    }
 }
 if (number < low || number > high) {  // Exception auslösen
  throw new InputMismatchException("Zahl nicht im angegebenen Intervall!");
 }
 System.out.println("Die Zahl " + number + " war lecker!");
}
```

Mehr Details zu Ausnahmen gibt es unter dem diesem Link: http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_06_001.htm.

4.1.3 Assertionen

Mit einer **Assertion** (*assertion*) kann angezeigt werden, dass an der entsprechenden Stelle im Programmcode die angegebene Bedingung erfüllt sein muss. Damit können mit Assertionen frühzeitig Bedingungen abgefangen werden, die später zu Ausnahmen führen (würden) und eine aussagekräftige Reaktion des Programmes hervorrufen. Dadurch sind Assertationen dazu geeignet, wichtige Aspekte des *Design-by-Contract* umzusetzen (siehe Abschnitt 4.1.2), denn so lassen sich Vor- und Nachbedingungen im Code sicherstellen. Die von Assertionen ausgelösten Ausgaben werden immer angezeigt, d.h. sowohl bei positiven,

als auch bei negativem Ausgang, *wenn* man das Programm in dem Modus `-enableassertions` ausführt, siehe auch S. 46.
Als Anwendung einer Assertion gehen wir zurück zur unserem Beispielspiel. Darin soll die Methode `moveTo()` der `Token` Klasse sicherstellen, dass die Zielposition innerhalb des Spielfeldes liegt. Bisher kennt die `Token` Klasse allerdings die Größe des Spielfeldes gar nicht. Daher führen wir noch eine `Board` Klasse ein. Das Spielbrett (`Board`) enthält einen Stapel von Spielsteinen (`Token`) als Attribut.

```
public class Board
{
  public int sizeX, sizeY;      // wir machen es uns einfach: public!
  private Stack<Token> tokens;  // die Spielsteine auf dem Brett

  Board(int sizeX, int sizeY) {
    this.sizeX = sizeX;
    this.sizeY = sizeY;
    tokens = new Stack<>();
  }

  protected void addToken(Token token) {
    tokens.push(token);
  }

  public String toString() {
    return "Board of size " + sizeX + "x" + sizeY;
  }
}
```

Außerdem hat jeder `Token` das `Board` als Attribut. So hat jeder Spielstein Information über die Brettgröße und `moveTo()` kann *mit Assertionen* sicherstellen, dass der Stein nicht vom Brett gezogen wird.

```
public class Token {
  private Board board;
  private int xPos, yPos;

  Token(Board board, int x, int y)
  {
    this.board= board;
    xPos = x;
    yPos = y;
  }

  protected void moveTo(int x, int y)
  {
    assert x >= 0 && x < board.sizeX : "x-Wert außerhalb des Spielbrettes";
    assert y >= 0 && y < board.sizeY : "y-Wert außerhalb des Spielbrettes";
    xPos= x;
```

```
    yPos= y;
  }
}
```

Um das Beispiel ausführen zu können, benötigen wir eine `main()` Methode in der `Board` Klasse:

```
public static void main(String[] args)     // Methode in der Klasse Board
  {
    Board board = new Board(5, 7);
// Spielbrett der Größe 5x7 erzeugen
    Token token1 = new Token(board, 0, 0);   // Ein Spielstein erzeugen
    board.addToken(token1);                  //   und hinzufügen
    token1.moveTo(8, 8);
// Auf verbotene Position setzen
    System.out.println("Token 1: " + token1);
  }
```

Dabei ist zu beachten, dass in der Grundeinstellung Assertionen bei der Ausführung ausgeschaltet sind. Mit der Option `-ea` bzw. `-enableassertions` werden sie in der JVM eingeschaltet. In IDEA gibt man dies bei *VM Options* unter *Run | Edit Configurations ...* an.

```
> javac Board.java
> java -ea Board
Exception in thread "main" java.lang.AssertionError: x-Wert außerhalb des ...
        at Token.moveTo(Token.java:18)
        at Board.main(Board.java:28)
```

Man kann mit Assertionen auch eigene Annahmen über den Zustand in einer bestimmen Codezeile überprüfen. Schauen wir uns folgendes Beispiel an:

```
if (i % 3 == 0) {
  // ...
} else if (i % 3 == 1) {
  // ...
} else {
  assert i % 3 == 2 : i;
  // ...
}
```

textbf[“Sie stimmt aber für i<0 nicht” ist im Begriffsverzeichnis aufgeführt. Hier eventuell eigentlich emph statt iemph gedacht? Oder sollte etwas anderes ins Verzeichnis?] In dem Beispiel scheint die `assert` Bedingung sicher erfüllt zu sein. *Sie stimmt aber für `i<0` nicht*, da % der Divisionsrest ist und immer das Vorzeichen des Dividenden annimmt.
Besonders bei komplexeren Fallunterscheidungen (`if`.. `else` oder `switch`) kann ein `assert` hilfreich sein. Der Code im letzten `else`-Block hier könnte z.B. bei

Änderungen in den oberen `if`-Bedingungen auch ungültig werden.
Weitere Informationen, auch darüber wozu Assertionen nicht benutzt werden sollen, stehen hier: `https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html`.

4.1.4 Modultests (JUnit)

Modultests (*unit tests*) sind ein wichtiges Werkzeug der Softwareentwicklung. Sie werden benutzt, um einzelne Komponenten auf korrekte Funktionalität zu prüfen. Besonders wichtig sind sie in größeren Projekten, um sicherzustellen, dass korrekt implementierte Module auch bei Weiterentwicklungen und Überarbeitungen korrekt bleiben. Die nächste Teststufe auf hörerer Ebene heißt *Integrationstest* und wird hier nicht behandelt. Es gibt auch das Model der **testgetriebenen Entwicklung** (*test-driven development*). Dort werden die Tests zuerst geschrieben, also vor der zu testenden Methode. Dadurch wird der Anforderungsrahmen an die Implementation im Voraus durch die Tests gesteckt. Dann werden die Tests bei der Entwicklung automatisiert ausgeführt.
Für beide Modelle gilt: die Erfahrung aus der Softwareentwicklung zeigt, dass mit automatisch ausgeführten Tests die Fehlerraten deutlich sinken und der Entwicklungsprozess schneller ist. Der Qualitätssicherung durch Tests kann natürlich nur so gut sein, wie die Tests. Diese sind also mit Bedacht zu entwerfen.
Für Java sind **JUnit Tests** als Framework für Modultests verbreitet. Die Entwicklung von Modultests wird dabei von den meisten IDEs unterstützt.

> Wie JUnit Testing in IDEA funktioniert, können Sie sich in den Videos zu dieser Woche anschauen.

4.2 Ein Kommentar zu Kommentaren

Kommentare tragen nichts zum Ablauf eines Programmes bei. Sie sind dennoch extrem wichtig, denn sie bilden eine Dokumentation, nicht nur für andere, auch für einen selbst. Kommentieren sollte man möglichst direkt beim Programmieren. (Nur beim Programmieren in der Vorlesung oder für Videos dürfen die Kommentare weggelassen werden. bzw. werden mündlich gegeben.)

4.2.1 Standardisierte Javadoc Kommentare

Kommentare, die im **Javadoc** Format geschrieben werden, können automatisch in eine API im HTML Format übersetzt werden. Die Kommentare zur Beschreibung von Klassen und Methoden werden von `/**` und `*/` eingeklammert, also

mit doppeltem Stern in der Eröffnung. In diesen Doc Kommentaren können bestimmte Elemente durch Tags, die mit `@` beginnen gekennzeichnet werden, siehe:

Tag & Parameter	Usage
@code *literal*	Formatiert Text im Code Font
@author *name*	Name des Autors
@version *version*	Versionsnummer, höchstens eine pro Klasse/ Interface
@param *name description*	Beschreibt einen Parameter der Methode.
@return *description*	Beschreibt den Rückgabewert.
@link *reference*	Erzeugt einen Link auf eine Klasse, Interface oder Methode

Da JavaDoc Kommentare dann in HTML übersetzt werden, können HTML Befehle wie `<em>`, `</em>`, `<i>`, `</i>` und `<p>` verwendet werden. Weitere Informationen gibt es unter `https://docs.oracle.com/javase/8/docs/technotes/tools/unix/javadoc.html`.

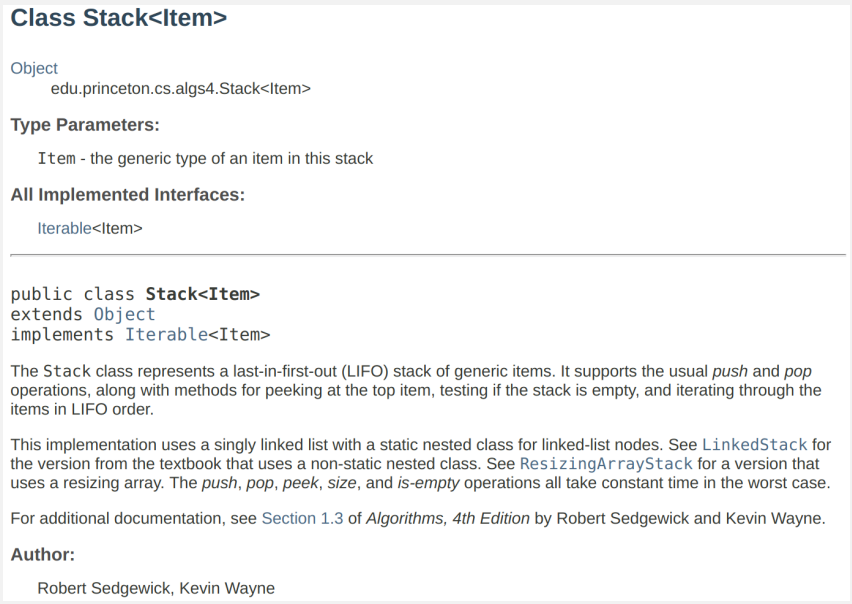
Wir schauen uns nun ein Beispiel an, welches Sie unter `https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/Stack.java.html` finden können. Wir betrachten eine Dokumentation einer Klasse Stack. Die Beschreibung der Klasse gibt an, was die Klasse implementiert, welche Datenformate benutzt werden und welche Parameter im Konstruktor gebraucht werden (`@param`). Sie steht vor der Deklaration der Klasse:

```
/**
 *  The {@code Stack} class represents a last-in-first-out (LIFO) stack of generic items.
 *  It supports the usual <em>push</em> and <em>pop</em> operations, along with methods
 *  for peeking at the top item, testing if the stack is empty, and iterating through
 *  the items in LIFO order.
 *  <p>
 *  This implementation uses a singly linked list with a static nested class for
 *  linked-list nodes. See {@link LinkedStack} for the version from the
 *  textbook that uses a non-static nested class.
 *  See {@link ResizingArrayStack} for a version that uses a resizing array.
 *  The <em>push</em>, <em>pop</em>, <em>peek</em>, <em>size</em>, and <em>is-empty</em>
 *  operations all take constant time in the worst case.
 *  <p>
 *  For additional documentation,
 *  see <a href="https://algs4.cs.princeton.edu/13stacks">Section 1.3</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 *
 *  @param <Item> the generic type of an item in this stack
 */
public class Stack<Item> implements Iterable<Item> {
// ...
```

Um Methoden zu dokumentieren, fügt man Kommentare vor den Methoden ein, die angeben, was die Methode macht, ob eine Exception geworfen wird, sowie welche Parameter die Methode nimmt und was der Rückgabewert ist.

```
// two exemplary methods of class Stack (taken from Sedgewick & Wayne as indicated below)
    /**
     * Adds the item to this stack.
     *
     * @param  item the item to add
     */
    public void push(Item item) {
        Node<Item> oldfirst = first;
        first = new Node<Item>();
        first.item = item;
        first.next = oldfirst;
        n++;
    }
    /**
     * Removes and returns the item most recently added to this stack.
     *
     * @return the item most recently added
     * @throws NoSuchElementException if this stack is empty
     */
    public Item pop() {
        if (isEmpty()) throw new NoSuchElementException("Stack underflow");
        Item item = first.item;        // save item to return
        first = first.next;            // delete first node
        n--;
        return item;                   // return the saved item
    }
```

Wenn man sich die generierte HTML Dokumentation dann anschaut, sieht sie so aus:



Class Stack<Item>

Object
edu.princeton.cs.algs4.Stack<Item>

Type Parameters:

`Item` - the generic type of an item in this stack

All Implemented Interfaces:

Iterable<Item>

```
public class Stack<Item>
extends Object
implements Iterable<Item>
```

The `Stack` class represents a last-in-first-out (LIFO) stack of generic items. It supports the usual *push* and *pop* operations, along with methods for peeking at the top item, testing if the stack is empty, and iterating through the items in LIFO order.

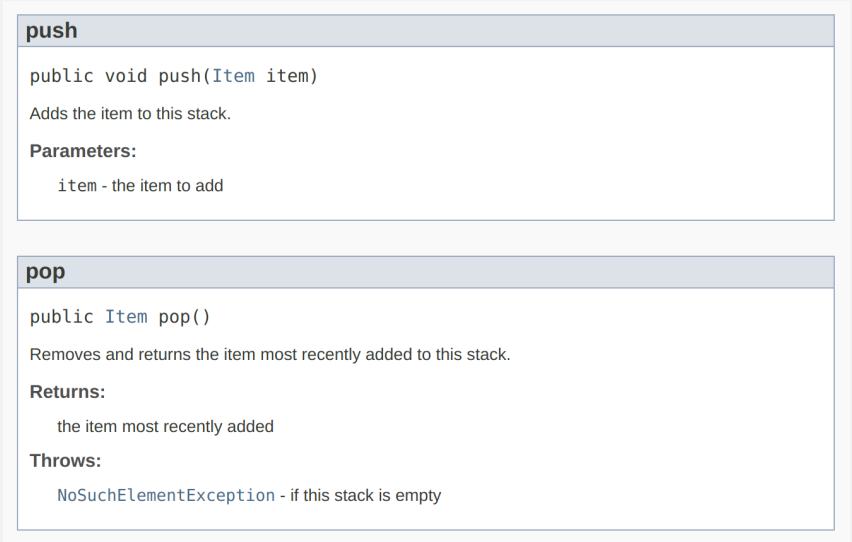
This implementation uses a singly linked list with a static nested class for linked-list nodes. See `LinkedStack` for the version from the textbook that uses a non-static nested class. See `ResizingArrayStack` for a version that uses a resizing array. The *push*, *pop*, *peek*, *size*, and *is-empty* operations all take constant time in the worst case.

For additional documentation, see Section 1.3 of *Algorithms, 4th Edition* by Robert Sedgewick and Kevin Wayne.

Author:

Robert Sedgewick, Kevin Wayne

Abbildung 4.1 Javadoc: Diese Java Documentation wurde automatisch aus den oben gezeigten Kommentaren in der Definition der Klasse `Stack` generiert.



push

```
public void push(Item item)
```

Adds the item to this stack.

Parameters:

`item` - the item to add

pop

```
public Item pop()
```

Removes and returns the item most recently added to this stack.

Returns:

the item most recently added

Throws:

`NoSuchElementException` - if this stack is empty

Abbildung 4.2 Javadoc: Diese Java Documentation für zwei Methoden wurde aus den oben gezeigten Kommentaren in dem Code der Methoden `push()` und `pop()` generiert.

Quellenverweise

Für die Einführung in Java wurden Informationen aus vielen Quellen geschöpft, insbesondere aus [Ullenboom, 2018], [Sedgewick and Wayne, 2014] und etwas aus [Vornberger, 2016]. Des Weiteren wurden die folgenden Webseiten verwendet:

- https://javapapers.com/core-java/java-history
- https://introcs.cs.princeton.edu/java/faq/c2java.html
- https://de.wikipedia.org/wiki/Klassendiagramm
- https://javapapers.com/core-java/java-polymorphism
- https://en.wikipedia.org/wiki/Abstract_data_type
- https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html
- https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html
- https://www.javatpoint.com/collections-in-java
- https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html
- https://en.wikipedia.org/wiki/Javadoc
- http://www.oracle.com/technetwork/java/javase/tech/index-137868.html
- https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/Stack.java.html
- https://openbook.rheinwerk-verlag.de/javainsel/08_001.html

Abbildungsverzeichnis

Listings

Index

Literaturverzeichnis

[Sedgewick and Wayne, 2014] Sedgewick, R. and Wayne, K. (2014). *Algorithmen: Algorithmen und Datenstrukturen (4. aktualisierte Auflage)*. Pearson Studium.

[Ullenboom, 2018] Ullenboom, C. (2018). *Java ist auch eine Insel*. Rheinwerk Computing, 13 edition.

[Vornberger, 2016] Vornberger, O. (2016). Algorithmen. skript zur vorlesung im ws 2016/2017. PDF und HTML unter http://www-lehre.inf.uos.de/~ainf/2016.