Algorithmen und Datenstrukturen

Vorlesung #02 - Einführung in Java Teil 2

Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

17 · Apr · 2019

Themen der heutigen Vorlesung

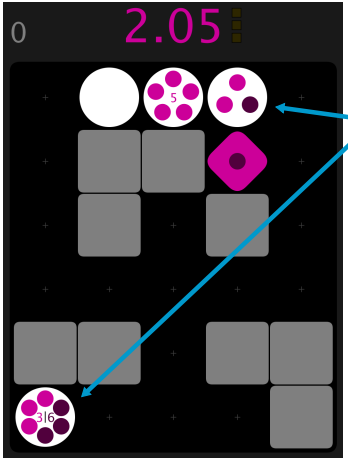
- Klassenhierarchie und Vererbung
- Generics
- Schnittstellen für Anwendungsprogrammierung (API)
- Schnittstellen (`interface`) und Schnittstellenvererbung
- Die Schnittstellen `Iterator`, `Iterable` und `Comparable`, `Comparator`
- Java *Collections*
- Debugging mit der IDE
- Die Sache mit der Gleichheit (`equals()`)
- Wachstumsordnungen
- Empirische und Analytische Laufzeitanalyse
- Polymorphismus

Klassenhierarchien und Vererbung (*inheritance*)

- 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.
- Alle sichtbaren Eigenschaften/ Methoden (`public` und `protected`) werden von der Oberklasse auf die Unterklasse übertragen, **vererbt**.
- Private (und `package` sichtbare) Eigenschaften werden nicht vererbt.
- In der Unterklasse können weitere Variablen und Methoden definiert werden.
- Geerbte Methoden können überschrieben (*override*) werden. Die überschriebene Methode sollte 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.
- Achtung: die Unterklasse ist also normalerweise 'größer' als die Oberklasse (mehr Daten und mehr Methoden).

Anlass zur Vererbung

- In dem Beispielspiel gibt es unterschiedliche Spielsteine.
- Die Trägersteine benötigen zusätzliche Attribute und Methoden, um die *Last* zu speichern und zu verändern.
- Daher benötigen sie eine eigene Klasse.
- Hier bietet sich Vererbung an, damit die gemeinsamen Methoden nicht neu implementiert werden müssen.



Manche Spielsteine können Kugeln als Last tragen.

Kugeln werden bei Zusammenstoß auf andere Steine übertragen.

Die maximale Last (`capacity`) ist fix für jedes Objekt.

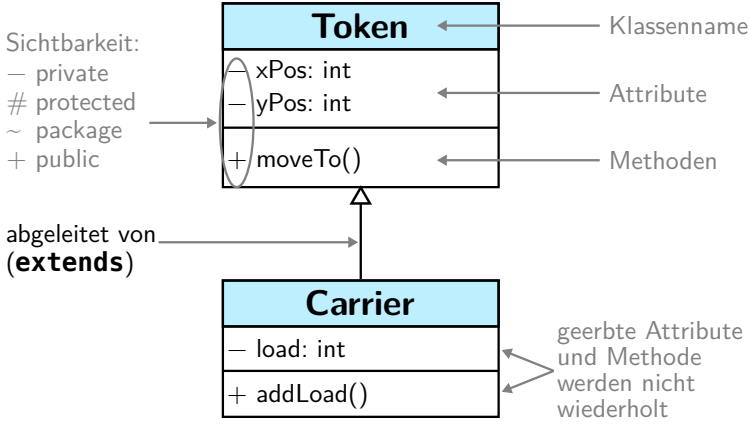
Beispiel zur Vererbung

```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
```

Vereinheitlichte Modellierungssprache (*Unified Modeling Language*; UML)

Klassenhierarchien können als Klassendiagramme in der **Vereinheitlichten Modellierungssprache** (*Unified Modeling Language*; UML) grafisch dargestellt werden:



Einschränkungen in der Vererbungshierarchie

- Von einer Klasse können beliebig viele Unterklassen abgeleitet werden.
- Aber eine Klasse kann nur eine (direkte) Oberklasse besitzen.
  - Ein Grund ist folgender: Würde eine Klasse von zwei Oberklassen abgeleitet, die eine Methode unterschiedlich implementieren, so wäre unklar welche Implementation vererbt wird.
  - Wir werden später Schnittstellen (*Interfaces*) kennenlernen, die einen Mechanismus für Mehrfachvererbung bieten, allerdings nur für Schnittstellenvorgaben, nicht für Implementierungen.
- Das Stichwort **`final`** in einer Klassendeklaration verbietet die Ableitung von Unterklassen.

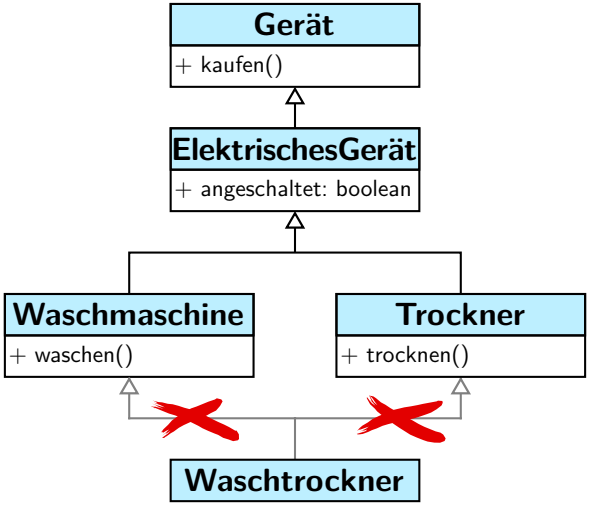
Abstrakte Methoden und Klassen

- Von einer Klasse, die als **`abstract`** deklariert wird, können keine Instanzen 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.

Wiederholung: Klassenmethoden (Statische Methoden)

- Methoden tauchen in Java meist als Objektmethode auf.
- D.h. die Methode wird in einer Klasse definiert und bezieht sich auf ein Objekt der Klasse (Aufruf durch das Objekt mit Punkt-Operator: `traeger.addLoad(2)`).
- Als Alternative gibt es **Klassenmethoden** (oder statische Methoden), die durch den Modifizierer **`static`** definiert werden.
- Klassenmethoden haben keinen Bezug zu einem konkreten Objekt (Aufruf durch den Klassennamen mit Punkt-Operator: `Math.sqrt(2)`).
- Klassenmethoden samt Implementierung kann es auch in abstrakten Klassen geben, von denen keine Objekte gebildet werden können.

Vererbungshierarchie von Klassen



Es kann keine Klasse `Waschtrockner` definiert werden, die `waschen()` von `Waschmaschine` und `trocknen()` von `Trockner` erbt, siehe Seite 6.

Vererbungshierarchie von Klassen – Code soweit möglich

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

class ElektrischesGeraet extends Geraet {
  public boolean angeschaltet;
}

class Waschmaschine extends ElektrischesGeraet {
  public void waschen() {};
}

class Trockner extends ElektrischesGeraet {
  public void trocknen() {};
}

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

- Die Klassen könnten auch **`public`** deklariert werden. Dann müsste allerdings jede Klasse in einer eigenen Datei stehen, die denselben Namen wie die Klasse hat.
- Auch abgesehen davon, dass der Methodenrumpf von `waschen()` leer ist, würde das Programm nichts tun. Die `Waschmaschine` ist ausgeschaltet (`boolean` wird mit `false` initialisiert).

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 durch ihre Semantik (Perspektive der Benutzerin; *'was?'*) gekennzeichnet, insbesondere durch Wertemenge, Operationen und Verhalten.
- Abstraktion zeigt, was gemacht werden kann und versteckt, wie es gemacht wird.
- Im Gegensatz dazu sind primitive Datentypen an ihre konkrete Repräsentation gebunden (Sicht der Implementation; *'wie?'*), für `int` z.B. als 32-Bit Zweierkomplement mit Überlaufproblematik.
- ADTs werden in Java durch das Klassenkonzept unterstützt.

[Sedgewick & Wayne S. 81; https://en.wikipedia.org/wiki/Abstract_data_type]

Datenabstraktion

Entwickler (Implementation der ADT)

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

Anwender (*client-code*, nutzt ADT)

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

- Wichtige Vorteile der Datenabstraktion:
  - Kapselung (*encapsulation*): Nutzer- und Implementationssicht (was/wie) sind getrennt. Der client-code kann sich auf die ADT Beschreibung verlassen, und braucht keine Kenntnis über die Implementation.
  - Modularität (*modularity*): Implementation können verbessert werden, ohne dass der *client-code* angepasst werden muss.
  - Geschütztheit (*integrity*): Nur diejenigen Daten und Methoden sind zugreifbar, die durch die Spezifikation dazu vorgesehen sind.
- Bevor wir Arten der Spezifikation von ADT besprechen, werden generische Typen eingeführt.

Motivation von Generics: Von einzelnen Spielsteinen zu einer Kollektion

- 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`.

Generische Typen (*Generics*)

- 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. Konvention: 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.
- Allerdings kann der Konstruktor des variablen Typs nicht explizit aufgerufen werden.
- Typparameter können nur als Referenztypen instanziiert werden. 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 automatisch *Casting* eines primitiven Typs auf den entsprechenden Referenztypen nennt man autoboxing.

Beispiel für generische Typen

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:

```
TokenStack toks = new TokenStack();
```

Mit Generics (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 ...
}
```

Erzeugung eine Stapels für Token:

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

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.

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();
}
```

Die API dient zur Dokumentation, während das **`interface`** als Technik in der Programmierung benutzt wird. Die Varianten sind ansonsten gleichwertig.

Schnittstellenvererbung

- Schnittstellen haben weder Datenfelder noch Konstruktoren. Sie können allerdings Konstanten definieren.
- Bei Methoden wird nur die Signatur definiert. Implementationen sind in Schnittstellen nicht möglich, siehe abstrakte Klassen.
- Eine Klasse erbt eine Schnittstelle mit dem Schlüsselword **`implements`**: Schnittstellenvererbung (*subtyping*). In diesem Fall muss die Klasse alle Methoden der Schnittstelle Signatur-konform implementieren.
- Viele Klassen können dieselbe Schnittstelle implementieren.
- Eine Klasse kann mehrere Schnittstellen implementieren. Dies ist ein wichtiger Unterschied zum *subclassing*.
- Es können auch Schnittstellen von Schnittstellen abgeleitet werden, mit dem Schlüsselwort `extends`.

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.

Anwendung von `Iterable` Objekten

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 Seite 15, bzw. Seite 21)

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

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

```
Stack<Token> toks = 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 Vorlesung #01.

Update der Stapel API

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.

Update/ Vervollständigung der Implementation eines Stapels

```
// 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;
  }

// Fortsetzung naechste Seite
```

Update/ Vervollständigung der Implementation eines Stapels

```
// Fortsetzung der Stack Klasse

  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;
    }
  }
}
```

Bemerkungen zu der Stapel Implementation

- Achtung: Diese Implementation ist eine Minimalversion ohne essentielle Überprüfungen, z.B. am Anfang von `pop()` ob der Stapel leer ist.
- Es wurde auch der Konstruktor weggelassen, da der default ausreicht (`head` wird mit `null` und `N` mit `0` initialisiert).
- Die Methode `remove()` eines Iterators muss zwar formal implementiert werden (Vorgabe durch das Interface), die Implementation darf aber leer sein.
- Der unten angegebene Link bietet eine saubere Vollimplementation.

Collections in Java

- In Java werden viele Varianten von ***Collections*** zur Verfügung gestellt.
- Eine *Collection* ist eine Datenstruktur, die Speicherung von und Zugriff auf viele Objekte gleichen Typs erlaubt (Alternativen zu einfachen Arrays).
- Für diese Veranstaltung sind **`LinkedList`** als Stack und Queue, **`PriorityQueue`** und in Vorlesung #12 `HashMap` und `HashSet` wichtig.
- Alle Klassen sind 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.
- Dies macht die *Collections* sehr praktisch, birgt aber die folgende Gefahr:
- Einige Klassen bieten auch Methoden an, die in der Datenstruktur nicht effizient sind.

Collections in Java: Achtung mit Laufzeit

- 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.
- Bei einem Prioritätenwarteschlange 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` im Anhang).
- Bei der Benutzung zusätzlicher Methoden in den Java Collections sollte immer auf deren Laufzeit geachtet werden (siehe auch Seite 60ff).

Debugging

- Bei der Korrektur von insbesondere logischen Fehlern ist ein Debugger eine immense Hilfestellung.
- Falls ein Programm mit einer Exception 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.
- Im Debugger kann das Programm dann Zeilenweise ausgeführt und Variableninhalte inspiziert werden.
- 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 Variable durch Klicken auf das Dreieck aufgeklappt werden.
- Durch Klicken auf '+' kann man arithmetische Ausdrücke als *watch* hinzufügen.
- Demo: Debugging in IDEA

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, gibt es die Methode **`equals()`**, die jede Klasse von `Object` erbt. Vererbt wird allerdings nur die syntaktische Gleichheit.
- Um eine semantische Gleichheit zu implementieren, muss `equals()` für eigene Klassen überschrieben werden. Dies kann von IDEA automatisch generiert werden.
- Das Überschreiben von `equals()` sollte mit dem entsprechenden Überschreiben von `hashCode()` einhergehen. Dies wird allerdings erst in Vorlesung #12 besprochen und relevant.

Einschub: Wiederholung von Wachstumsordnungen

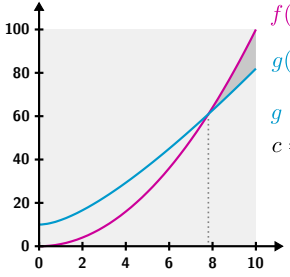
- Im letzten Semester wurden Funktionsklassen von Wachstumsordnungen eingeführt: $\boldsymbol{o}(f)$, $\boldsymbol{O}(f)$, $\boldsymbol{\Theta}(f)$, $\boldsymbol{\Omega}(f)$, $\boldsymbol{\omega}(f)$ um Rechenzeit und Speicherbedarf zu quantifizieren.
- Diese werden auch in dieser Vorlesung benutzt, um Rechenzeit und Speicherbedarf verschiedener Algorithmen zu charakterisieren und vergleichen.
- Die Wichtigste für uns ist

$$\boldsymbol{O}(f) = \{g \text{ Funktion} \mid \exists c > 0\ \exists n_0\ \forall n > n_0\ \ g(n) < c \cdot f(n)\}$$

- und die wichtigsten Wachstumsordnungen sind
  - $\boldsymbol{O}(1)$ konstant
  - $\boldsymbol{O}(\log N)$ logarithmisch
  - $\boldsymbol{O}(N)$ linear
  - $\boldsymbol{O}(N \log N)$ 'leicht überlinear'
  - $\boldsymbol{O}(N^2)$ quadratisch
  - $\boldsymbol{O}(N^3)$ kubisch
  - $\boldsymbol{O}(2^N)$ exponentiell zur Basis 2

Illustration von Wachstumsordnungen

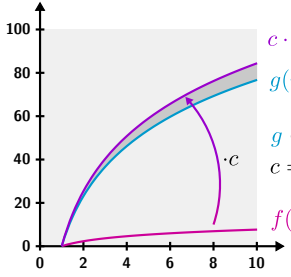
$$\boldsymbol{O}(f) = \{g \text{ Funktion} \mid \exists c > 0\ \exists n_0\ \forall n > n_0 \;\; g(n) < c \cdot f(n)\}$$



$f(n) = n^2$

$g(n) = 3n\log(n+1) + 10$

$g \in \boldsymbol{O}(f)$:
$c = 1 \quad n_0 = 8$



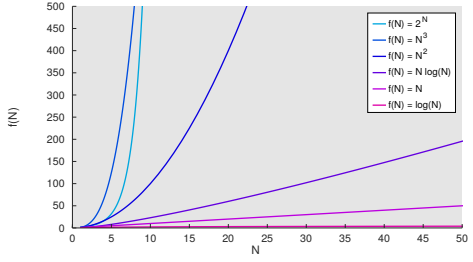
$c \cdot f(n) = 11\log(n)$

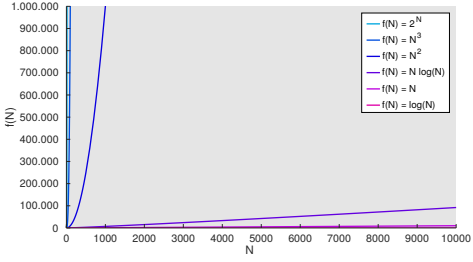
$g(n) = 2\log(n^5)$

$g \in \boldsymbol{O}(f)$:
$c = 11, \quad n_0 = 2$

$f(n) = log(n)$

Wachstumsordnungen



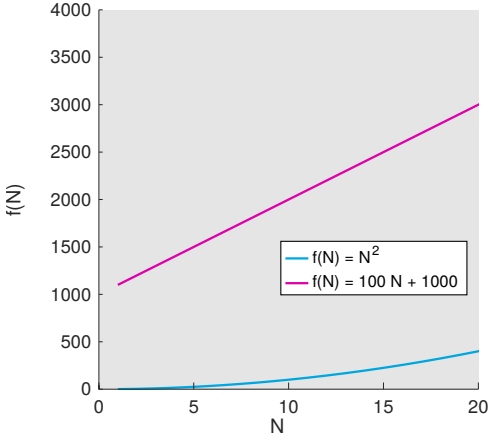


- Wertebereich für N: klein
- Wertebereich für N: groß
- Der Eindruck hängt auch stark vom gewählten Bereich auf der y-Achse ab.

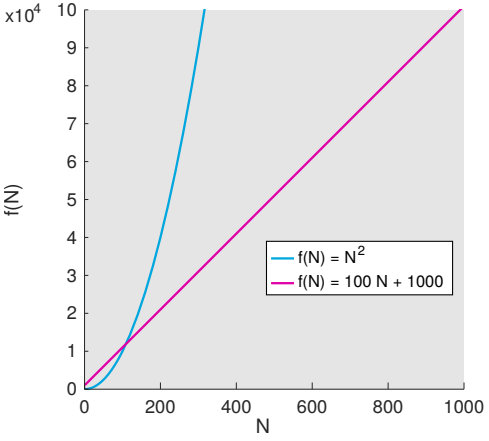
Der Einfluss von *Offset* und konstanten Faktoren

- Die Definition der Wachstumordnungen ignoriert konstante Faktoren und Offset (Verschiebung auf der y-Achse).
- Dies bedeutet, dass eine Beurteilung, die nur auf Wachstumordnungen beruht, immer kritisch hinterfragt werden muss.
- Praktisch ist tatsächlich fast nur die Wachstumsordnung relevant, wenn es um '**große** Eingaben' geht (z.B. um das Sortieren von > 10.000 Werten).
- Sehr große Offsets oder konstante Faktoren sind in der Rechenzeit von Algorithmen äußerst selten.
- Als Beispiel für den Einfluss vergleichen wir die Funktionen
  - $100N + 1000$ (kleinere Wachstumsordnung, großer Faktor und Offset) und
  - N^2 (große Wachstumsordnung, kleiner Faktor und Offset).

Der Einfluss von *Offset* und konstanten Faktoren



- Im kleinen Wertebereich haben Offset und konstanter Faktor einen starken Einfluß.



- Bei größeren N zählt im Wesentlichen die Wachstumsordnung.

Weitere Klassen von Wachstumsordnungen

- Das Gegenstück zu der Klasse $\boldsymbol{O}(f)$ die $\leq$ entspricht, ist die Entsprechung von $\geq$ bzgl. Wachstumsordnungen:

$$\boldsymbol{\Omega}(f) = \{g \text{ Funktion} \mid \exists c > 0\ \exists n_0\ \forall n > n_0 \quad g(n) > c \cdot f(n)\}$$

- Funktionen g, die sowohl in $\boldsymbol{O}(f)$ als auch in $\boldsymbol{\Omega}(f)$ sind, haben dieselbe Wachstumsordnung wie f.
- Für die Gleichheit von Wachstumsordnungen wird die Klasse $\boldsymbol{\Theta}(f)$ definiert:

$$\boldsymbol{\Theta}(f) = \boldsymbol{O}(f) \cap \boldsymbol{\Omega}(f)$$

- Für die Funktion $T(N) = 10N^2 + 5N + 27$ gilt also $T \in \boldsymbol{\Theta}(N^2)$.
- Die wird auch kurz $10N^2 + 5N + 27 \in \boldsymbol{\Theta}(N^2)$ geschrieben.

Wachstumsordnungen von Laufzeit und Speicherbedarf

- Die Wachstumsordnungen $\boldsymbol{O}(f)$, $\boldsymbol{\Theta}(f)$ und $\boldsymbol{\Omega}(f)$ werden benutzt, um die Laufzeit und den Speicherbedarf von Programmen und Algorithmen zu charaterisieren.
- Lässt sich z. B. die Laufzeit eines Programmes bei einer Eingabegröße N durch $T(N) = 15\,N^2 + 3\,N + 10$ Sekunden abschätzen, dann spricht man von einer **Laufzeit in $\boldsymbol{O}(N^2)$**, da $T(N) \in \boldsymbol{O}(N^2)$.
- Dabei spielt die Einheit der Zeitmessung keine Rolle (ob Millisekunden, Minuten oder Tage), da dies durch den konstanten Faktor c in der Definition von $\boldsymbol{O}(f)$ ausgeglichen wird.
- Analog kann man den Speicherbedarf als Wachstumordnung ausdrücken.

Laufzeitanalyse

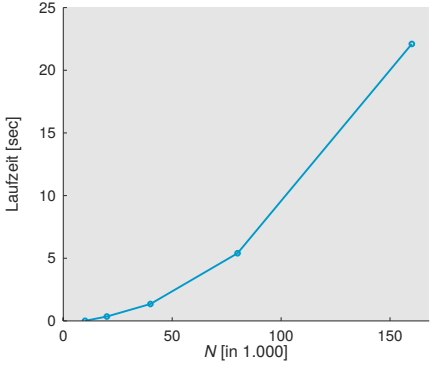
- Für eine Laufzeitanalyse stellt man zunächst die Abhängigkeit von der Eingabe fest, z.B. die Länge N einer Liste, die sortiert werden soll.
- Es können auch mehrere Eingabegrößen relevant sein, z.B. die Anzahl der Knoten V und die Anzahl der Kanten E eines Graphes.
- Bei manchen Algorithmen spielt nicht nur die Anzahl der Eingabedaten sondern auch die konkreten Werte eine Rolle. Manche Sortieralgorithmen laufen z.B. schneller, wenn die Daten schon halbwegs vorsortiert sind.
- Dies führt zu einer Unterscheidung von Laufzeit im Durchschnitt und im *Worst Case*. Siehe in diesem Zusammenhang auch die amortisierte Laufzeitanalyse auf S. 58.
- Wir führen als Beispiel eine Laufzeitanalyse der Methode `TwoSumCount(int[] a)` durch zunächst empirisch, dann analytisch.
- Diese Funktion zählt in dem `int` Array `a` die Paare `a[i]`, `a[j]`, deren Summe `0` ergibt.

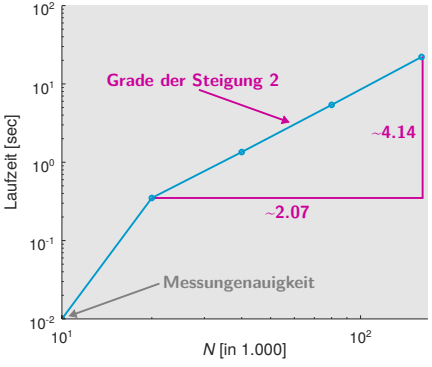
Empirische Laufzeitanalyse für TwoSum

```java
public class TwoSum
{
  public static int count(int[] a)   // statische Funktion, kein Objekt notwendig
  { int N = a.length;
    int counter = 0;
    for (int i = 0; i < N; i++)
      for (int j = i+1; j < N; j++)
        if (a[i] + a[j] == 0)
          counter++;
    return counter;
  }

  public static void main(String[] args)
  { int N = Integer.parseInt(args[0]);
    int[] a = new int[N];
    for (int i = 0; i < N ; i++)
      a[i]= -10000 + (int)(20000*Math.random());

    long start = System.currentTimeMillis();
    int counter = count(a);          // Aufruf innerhalb der Klasse ohne 'TwoSum.'
    long stop = System.currentTimeMillis();
    System.out.println("Count: " + counter + " in " + (stop-start)/1000.0 + "s");
  }
}
```

Empirische Laufzeitanalyse für TwoSum

- Für Felder der Länge N = 10k, 20k, 40k, 80k und 160k wurden folgende Laufzeiten gemessen: 0.01 s, 0.34 s, 1.35 s, 5.40 s, 22.11 s.
- Der genaue funktionale Zusammenhang zwischen Eingabegröße und Laufzeit ist aus den Datenpunkten nicht ersichtlich (ob quadratisch, kubisch, ...).
- Trick: Verwende logarithmische Skala auf beiden Achsen.
- Die Steigung 2 im *loglog* Plot zeigt einen quadratischen Zusammenhang an.





Mathematische Erklärung für den loglog Trick

- Die Steigung einer Funktion $f(N)$ von der Stelle N_1 zu N_2 ist:

$$\frac{f(N_2) - f(N_1)}{N_2 - N_1}$$

- Mit logarthmischer Skala auf beiden Achsen ergibt sich

$$\frac{\log(f(N_2)) - \log(f(N_1))}{\log(N_2) - \log(N_1)}$$

- Für eine Funktion $f(N) = N^k$ errechnet sich also die Steigung im loglog Plot zu

$$\frac{\log(f(N_2)) - \log(f(N_1))}{\log(N_2) - \log(N_1)} = \frac{\log(N_2^k) - \log(N_1^k)}{\log(N_2) - \log(N_1)} = \frac{k\log(N_2) - k\log(N_1)}{\log(N_2) - \log(N_1)}$$

$$= k\,\frac{\log(N_2) - \log(N_1)}{\log(N_2) - \log(N_1)} = k$$

Analytische Laufzeitanalyse für TwoSum

```
1  public static int TwoSumCount(int[] a)
2  {                                    // Häufigkeit der Ausführung jeder Code Zeile
3    int N = a.length;                  // 1
4    int counter = 0;                   // 1
5
6    for (int i = 0; i < N; i++)        // N
7      for (int j = i+1; j < N; j++)    // N(N − 1)/2
8        if (a[i] + a[j] == 0)          // N(N − 1)/2
9          counter++;                   // 0 bis N(N − 1)/2, abhängig von den Daten
10   return counter;                    // 1
11 }
```

- Um die Rechenzeit zu bestimmen, müssen diese Häufigkeiten mit der benötigten Ausführungsdauer jeder Codezeile multipliziert und dann aufsummiert werden.
- Wir interessieren uns hier nur für Wachstumsordnung. Daher können wir die unterschiedlichen Ausführungsdauern der Zeile ignorieren und zählen nur die Zeilen.
- In Zeile 9 zählen wir den *worst case*. Somit erhalten wir
  $1 + 1 + N + 3 \cdot N(N-1)/2 + 1 = \frac{3}{2}N^2 - \frac{1}{2}N + 3,$
  also Wachstumsordnung N^2.

Fazit für eine Laufzeitanalyse in Wachstumsordnungen

- Für eine Laufzeitanalyse in Wachstumsordnungen spielen konstante Faktoren keine Rolle.
- Daher kommt es hier nur darauf an, wie oft die innerste Schleife durchlaufen wird.
- Dagegen spielt die Anzahl der Befehle in einer Schleife oder anderswo keine Rolle, und ebenso wenig, wie aufwändig die einzelnen Befehle sind.
- Letzteres gilt natürlich nur, wenn die Ausführungsdauer der Befehle nicht von der Eingabe abhängt.
- Bei der Benutzung von Bibliotheksfunktionen z.B. aus den Java Collections muss die Laufzeit in der Dokumentation recherchiert und entsprechend berücksichtigt werden, siehe auch Seite 60.

Erinnerung Autoboxing

- Wie erzeugt man einen Stapel von `int` Werten?
- Erinnerung: Generische Typparameter können nur mit Referenztypen instanziiert werden. Für primitive Datentypen müssen die Wrappertypen verwendet werden.
- Das automatische Casting von priviten Datentypen zu Wrappertypen heißt *autoboxing*, das Casting zurück unboxing.

```
Stack<Integer> intStack = new Stack<>();

int e = 17;
intStack.push(e);      // autoboxing von int to Integer
intStack.push(-3);
e = intStack.pop();    // unboxing von Integer to int
```

Die Wrappertypen haben, u. a. durch das *autoboxing* einige verwunderliche Eigenschaften. Es sind Referenztypen, aber sie verhalten sich zum Teil anders.

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) Kontext-abhängig unterschiedliche Datentypen annehmen kann.
- Es gibt verschiedene Arten des Polymorphismus.

Ad-Hoc Polymorphismus

- **Ad-hoc Polymorphismus:** Operatoren und Methoden können 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.
  - Implizite Typumwandlung (*coertion polymorphism*): Daten eines 'kleineren' Datentyps werden 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).

Universeller Polymorphismus

- **Parametrischer Polymorphismus**: Datentypen und Methoden können Argumente variablen Typs haben: Generics.
- **Subtyp Polymorphismus**: Objekte können 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.

Die Schnittstellen `Comparable` und `Comparator`

- Neben `Iterable` und `Iterator` gibt es ein weiteres Paar wichtiger 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 Schnittstellen `Comparable` und `Comparator`

Die Schnittstelle Comparable

public interface Comparable<T>

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

- Die Schnittstelle `Comparable` sollte implementiert werden, wenn es nur eine sinnvolle Ordnung auf den Objekten der Klasse gibt (genannt 'natürliche Ordnung').

Die Schnittstelle Comparator

public interface Comparator<T>

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

- 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 Schnittstellen `Comparable` und `Comparator`

- In beiden Varianten sollen `v.compareTo(w)` bzw. `compare(v, w)` Werte `-1`, `0`, oder `1` zurückliefern, und zwar
  - `-1` für v<w
  - `0` für v=w und
  - `1` für v>w.
- wobei die rechte Seite eine sinnvolle Ordnung für die Elemente der Klassen darstellt.

Damit diese Relation eine sinnvolle Ordnung induziert, muss Folgendes erfüllt sein:

- Sie muss für alle Paare von Objekten definiert sein (total)
- Für alle v gilt v=v, d.h. `v.compareTo(v)==0` (reflexiv)
- Wenn v<w ist, dann auch w>v; wenn v=w dann auch w=v (anti/symmetrisch)
- Aus u<v und v<w folgt u<w (transitiv)

Implementationsbeispiel Comparator 1/3

```java
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)";
  }

  // main() Methode folgt
}
```

Implementationsbeispiel Comparator 2/3

```
import java.util.Comparator;
// Die Comparator könnten auch als anonyme Klassen in 'Person' integriert werden.

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);
  }
}
```

Implementationsbeispiel Comparator 3/3

```
// 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);
  }
```

Implementationsbeispiel Comparable

```
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);
        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);
    }
}
```

Rückschau

Nach dieser Vorlesung sollten Sie folgende Konzepte verinnerlicht haben:

- Vererbungsmechanismus von Java mit seinen Einschränkungen
- Überschreiben von Methoden bei der Vererbung
- rein und partiell abstrakte Klassen
- *Unified Modeling Language* (UML)
- Datenabstraktion (ADT) und Application Programmming Interface (API)
- Generische Typen
- Schnittstellenvererbung, subclassing vs. subtyping
- Schnittstellen `Iterable` und `Iterator` sowie `Comparable` und `Comparator`
- Implementationen: Multimenge, Stapel, Warteschlange
- Wrappertypen, *autoboxing*, *unboxing*
- Polymorphismus
- *Collections* in Java
- Empirische und Analytische Laufzeitanalyse

Anhang

Inhalt des Anhangs:

API für eine Warteschlange (FIFO)

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 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;
  }

// Fortsetzung naechste Seite
```

Implementation einer Warteschlange (2)

```
// Fortsetzung der Queue Klasse

  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 Seite 23. Insbesondere fehlen essentielle Überprüfungen, z.B. am Anfang von `dequeue()` ob die Schlange leer ist.

API und Implementation einer Multimenge

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 immer ein `Stapel` an Stelle einer `Bag` benutzt werden.
- Der einzige Sinn einer separaten Klasse `Bag` ist es explizit zu machen, dass keine Funktionalität zum Entfernen von Elementen gebraucht wird.

Amortisierte Laufzeitanalyse

- Bei manchen Algorithmen kann die Laufzeit von Fall zu Fall stark schwanken.
- Beispiel: Implementation eines Stapels mit einem Array an Stelle einer verketteten Liste.
- Die Größe des Arrays muss dynamisch angepasst werden, z.B. kann die Größe verdoppelt werden, wenn ein Element bei voll belegtem Array hinzugefügt werden soll.
- Dann ergeben sich beim schrittweisen Hinzufügen von Elementen, jeweils in den Fällen der Array Vergrößerungen deutlich längere Laufzeiten (Daten müssen von dem alten in das neue Array kopiert werden).
- Bei der **amortisierten Laufzeitanalyse** wird die Laufzeit über die unterschiedlichen Fälle gemittelt.

Amortisierte Laufzeitanalyse (Beispiel)

- Die Schleife zum Kopieren der Daten aus dem alten in das neue Array wird N mal durchlaufen, wenn das Array von Größe N auf $2N$ erweitert wird.
- Nehmen wir N als Zweierpotenz an, so ist beim schrittweisen Hinzufügen von N Elementen die Gesamtanzahl der Schleifendurchläufe

$$1 + 2 + 4 + 8 + \cdots + N = 2N - 1$$

- Es ergibt sich also eine durchschnittliche Anzahl von $\frac{2N-1}{N}$, also knapp 2 Schleifendurchläufen pro `push()` Operation. Die amortisierten Kosten sind also in $\boldsymbol{O}(1)$.

Laufzeit bei verschiedenen Datenstrukturen

- Bei der Implementierung von Algorithmen ist bei der Auswahl der Datenstrukturen die Laufzeit der Operationen zu beachten.
- Im folgenden werden die Laufzeiten für einige Datenstrukturen aufgelistet.
- Für fehlende Datenstrukturen oder Methoden ist die Java Dokumentation zu konsultieren.
- Leider sind die Laufzeiten oft in der Dokumentation gar nicht angegeben. In diesem Fall sollte man in die Quellen schauen, z. B. `http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/LinkedList.java`.

Laufzeiten

Laufzeiten für die auf Seiten 21, 55 und 57 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()`	$O(1)$

Der Zusatz *worst case* bei der Laufzeit ist insbesondere eine Abgrenzung zu einer *amortisierten Laufzeit* (siehe Seite 58), 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.

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. Außerdem besitzt sie noch weitere Methoden, die hier nicht aufgeführt sind.

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.

Laufzeiten für ausgewählte Java *Collections*

Bei der `PriorityQueue` ist zu beachten, dass ein Ändern der Priorität durch Entfernen (`remove(Object o)`) und wieder Einfügen (`add(E e)`) realisiert werden muss, was in einer linearen Laufzeit resultiert. In der nächsten Vorlesung wird die `IndexPriorityQueue` eingeführt, die dies in logarithmischer Zeit erlaubt, sofern Indizes für die Elemente verfügbar sind.

PriorityQueue (*worst case*)	
`add(E e)`	$O(\log N)$
`contains(Object o)`	$O(N)$
`peek()`	$O(1)$
`poll()`	$O(\log N)$
`remove()`	$O(\log N)$
`remove(Object o)`	$O(N)$

Literatur

- Sedgewick R & Wayne K, **Introduction to Programming in Java: An Interdisciplinary Approach.** 2. Auflage, Addison-Wesley Professional, 2017. Onlinefassung: `https://introcs.cs.princeton.edu/java`
- Ullenboom C, **Java ist auch eine Insel.** 13. Auflage, Rheinwerk Computing, 2018. Onlinefassung: `http://openbook.rheinwerk-verlag.de/javainsel`

Danksagung. Die Folien wurden mit LaTeX erstellt unter Verwendung vieler Pakete, u.a. beamer, listings, lstbackground, pgffor und colortbl sowie eine Vielzahl von Tipps auf `tex.stackexchange.com` und anderen Internetseiten.

Index