Algorithmen und Datenstrukturen

Vorlesung #02 - Einführung in Java Teil 2
Laufzeitanalyse

Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

25 · Apr · 2023

Themen der heutigen Vorlesung

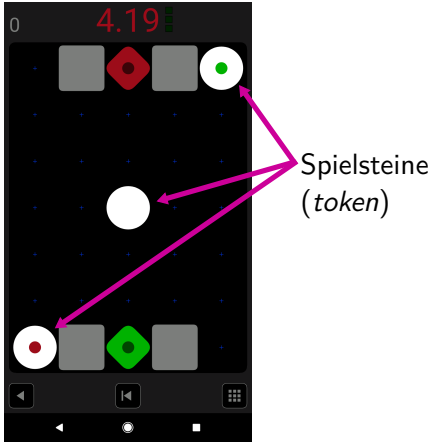
- Debugging mit der IDE
- Klassenhierarchie und Vererbung
- Generics
- Schnittstellen (API; `interface`) und Schnittstellenvererbung
- Die Schnittstellen `Iterator` und `Iterable`
- Wachstumsordnungen
- Empirische und Analytische Laufzeitanalyse
- Die Sache mit der Gleichheit (`equals()`)
- Java *Collections*

Debugging

- Bei der Korrektur von insbesondere logischen Fehlern ist ein Debugger eine immense Hilfestellung.
- Debugger springt bei Exception in die verursachenden Zeile
- oder hält an definierten *Breakpoints*.
- Programm kann zeilenweise ausgeführt und Variableninhalte inspiziert werden.
- Siehe Videos zu Debugging mit IDEA im ISIS Kurs

Beispiel Klasse 'Token' (Spielsteine)

```
1  class Token {
2    static int counter;    // Klassenvariable
3    int xPos, yPos;        // Instanzvariablen
4
5    Token() {              // 1. Konstruktor
6      counter++;
7    }
8    Token(int x, int y) { // 2. Konstruktor
9      this();              // Verkettung
10     xPos = x;
11     yPos = y;
12   }
13                          // Methode
14   void moveTo(int x, int y) {
15     xPos = x;
16     yPos = y;
17   }
18 }
```



Hier fehlen noch die für Java typischen Zugriffsmodifizierer, siehe unten.

Beispiel Klasse 'Token' (Spielsteine)

```java
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 unten.

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

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.

Update der Beispiel Klasse

```
1  public class Token {
2    private static final Shapetype shape = CIRCLE;  // Konstante
3    private static int counter;
4    private int xPos, yPos;
5
6    public Token() {
7      counter++;
8    }
9    public Token(int x, int y) {
10     this();
11     xPos = x;
12     yPos = y;
13   }
14
15   public void moveTo(int x, int y) {
16     xPos= x;
17     yPos= y;
18   }
19 }
```

Bemerkung zur Sichtbarkeit von Instanzvariablen

- Instanzvariablen sollten nicht **`public`** definiert werden:
  - Kontrollierter Zugriff
  - Flexibilität: Implementation der Datenstruktur kann geändert werden, ohne dass der Client-Code geändert werden muss.
- Aber wie kann dann auf `xPos`, `yPos` eines Spielsteins zugegriffen werden?
- Über *getter* und *setter* Methoden!

```
1  private int xPos;
2  // ...
3  public int getXPos() {
4    return xPos;
5  }
6  public void setXPos(int x) {
7    xPos= x;
8  }
```

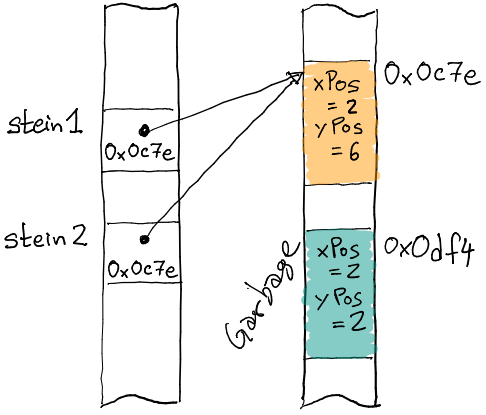
- Auf primitive Datentypen wird diese Konvention nicht immer angewendet.
  (Abwägung von Nutzen gegen zusätzlichen Aufwand)

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.

```
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);
```

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



Wie funktionierte die Ausgabe von Token?

- Jede Klasse ist eine Unterklasse von **`Object`**.
- Klassen erben die Methoden ihrer Oberklasse (mehr darüber folgt).
- Die Methoden können *überschrieben* werden. Dann ist die überschriebene Methode der Oberklasse immer noch über "**`super.`**" zugänglich.
- Die `Object` Klasse implementiert die Methode `toString()` dadurch, dass Klassenname und Speicherplatz ausgegeben werden.
- Die `toString` Methoden werden automatisch beim *casting* zu `String` verwendet.

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

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

Garbage Collection

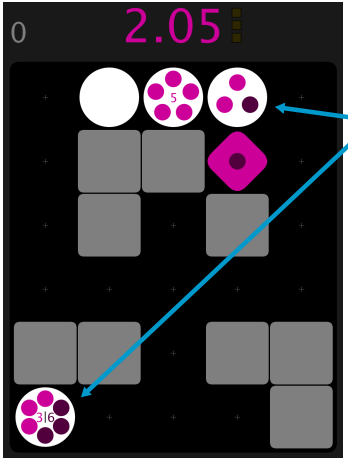
- Durch Zuweisungen von Referenztypen kann ein Objekt im Speicher "verloren" gehen, wenn keine Referenz auf das Objekt mehr existiert (siehe Seite **??**).
- Solche Objekte werden als **Garbage** (Müll) bezeichnet, da auf sie nicht mehr zugegriffen werden kann.
- Bei Java läuft im Hintergrund eine **Speicherbereinigung**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

Klassenhierarchien und Vererbung (*inheritance*)

- Mit **`extend`** kann eine Klasse eine andere erweitern (*subclassing*).
- Dies ergibt eine **Unterklasse**, die andere ist **Oberklasse**.
- Nur sichtbaren Eigenschaften/ Methoden (`public` und `protected`) werden **vererbt**.
- Die Unterklasse kann weitere Variablen und Methoden definieren.
- Geerbte Methoden können überschrieben (*override*) werden.
- 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.

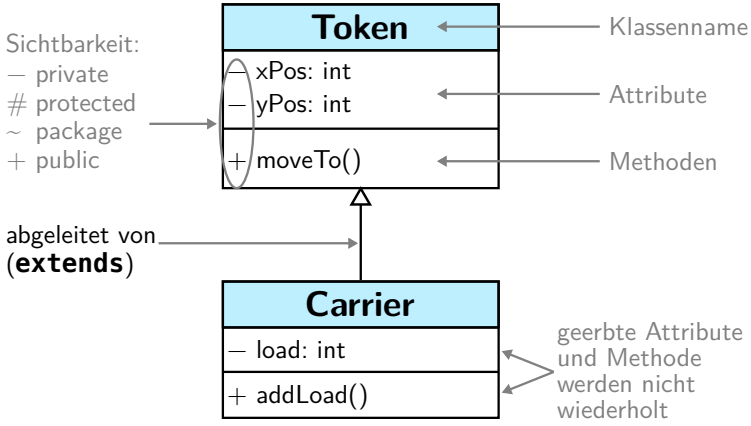
Beispiel zur Vererbung

```
1  public class Carrier extends Token {
2    private int capacity;              // Instanzvar. zusätzlich zu den geerbten
3    private int load;
4
5    public Carrier(int capacity) {
6      super();                         // Konstruktor der Oberklasse aufrufen
7      this.capacity = capacity;
8    }
9
10   public void addLoad(int deltaLoad) {
11     load += deltaLoad;
12   }
13 }
```

```
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)

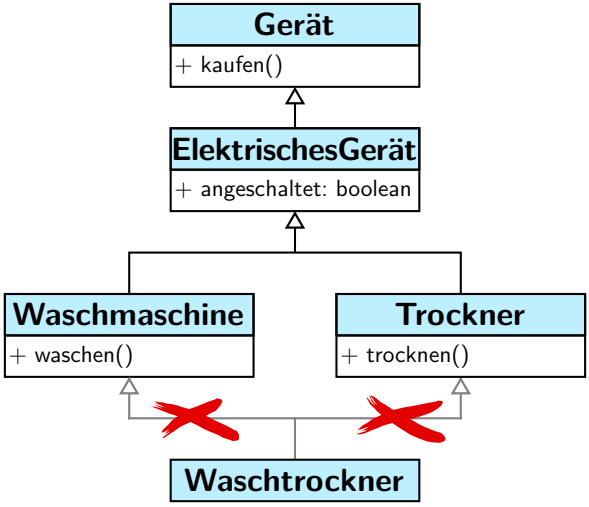
Klassenhierarchien können als Klassendiagramme in der **Vereinheitlichten Modellierungssprache** (*Unified Modeling Language*; UML) grafisch dargestellt werden:



Einschränkungen in der Vererbungshierarchie

- Von einer Klasse können beliebig viele Unterklassen abgeleitet werden.
- Aber eine Klasse kann nur eine (direkte) Oberklasse besitzen.
  - Bei mehreren Oberklassen könnte Vererbung Konflikte verursachen.
  - Mehrfachvererbung ist bei Schnittstellen möglich.
- Das Stichwort `final` in einer Klassendeklaration verbietet die Ableitung von Unterklassen.

Vererbungshierarchie von Klassen



Es kann keine Klasse `Waschtrockner` definiert werden, die `waschen()` von `Waschmaschine` und `trocknen()` von `Trockner` erbt, siehe Seite 15.

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).

Abstrakte Methoden und Klassen

- Von **abstrakten Klassen** können keine Instanzen gebildet werden. Sie sind nur eine Modellierungsklassen für Unterklassen.
- Schlüsselwort **`abstract`** in der Klassendeklaration
- Diese Klassen können neben normalen Methoden deren Implementierungen vererbt werden auch **abstrakte Methoden** besitzen:
- Abstrakten Methoden geben nur die Signatur vor; keine Implementierung
  - Schlüsselwort **`abstract`** in der Methodendeklaration
  - Abgeleitete Klassen müssen die vorgegebene Signatur einhalten.
- Wenn eine abstrakte Klasse ausschließlich abstrakte Methoden hat, wird sie auch **rein** abstrakte Klasse gennant, andernfalls **partiell** abstrakte Klasse.

Motivation von Generics: Von einzelnen Spielsteinen zu einer Kollektion

- Mehrere Spielsteine (`Token`) können in einem Array gespeichert werden:

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

- Oft besser: verketteten Liste, z. B. als Stapel
- IntroProg: Listen von `int`
- Wir brauchen Listen von *Objekten*, hier für die Klasse `Token`.

Generische Typen (*Generics*)

- Definiere Klassen, bei denen der Typ einer Variablen selbst variabel ist, eine **Typvariable** oder formaler Typparameter (*formal type parameter*)
- 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
- Eine Typvariable kann wie eine normale Typbezeichnung benutzt werden.
- Allerdings kann der Konstruktor des variablen Typs nicht explizit aufgerufen werden.

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

Autoboxing in Aktion

- Wie erzeugt man einen Stapel von `int` Werten?
- Nur Referenztypen können generische Typparameter instanziiern.
- Statt primitive Datentypen nimmt man **Wrappertypen**: `Boolean`, `Integer`, `Short`, `Long`, `Double`, `Float`, `Byte`, `Character`
- autoboxing: automatisches Casting von priviten Datentypen zu Wrappertypen; Rückweg: 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
```

Spezifikation von Abstrakten Datentypen (ADTs)

ADTs realisieren eine Art Vertrag zwischen Nutzerïn (*client-code*) und Entwicklerïn (Implementierung) durch:

- Beschreibung der **Schnittstelle für Anwendungsprogrammierung** (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();
}
```

API: Dokumentation; `interface`: Programmiertechnik

Schnittstellenvererbung

- Schnittstellen haben weder Datenfelder (nur Konstanten) noch Konstruktoren.
- Bei Methoden wird nur die Signatur definiert, keine Implementation, siehe abstrakte Klassen.
- Schnittstellenvererbung (*subtyping*) mit **`implements`**: Klasse muss alle Methoden der Schnittstelle Signatur-konform implementieren.
- Viele Klassen können dieselbe Schnittstelle implementieren.
- Eine Klasse kann mehrere Schnittstellen implementieren – im 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 21, bzw. Seite 28)

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

so kann man elegant und einfach über eine Tokenliste iterieren:

```
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 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.

Einschub: Wiederholung von Wachstumsordnungen

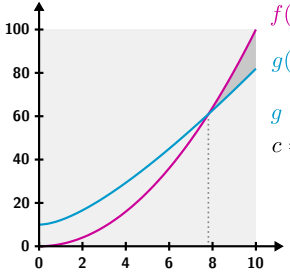
- Funktionsklassen von Wachstumsordnungen: $o(f)$, $O(f)$, $\Theta(f)$, $\Omega(f)$, $\omega(f)$
- Charakterisierung und Vergleich von Rechenzeit und Speicherbedarf verschiedener Algorithmen

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

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

Illustration von Wachstumsordnungen

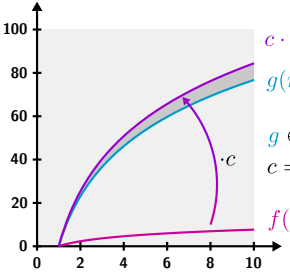
$$\boldsymbol{O}(f) = \{g \text{ Funktion} \mid \exists c > 0\ \exists n_0\ \forall n > n_0 \quad g(n) < c \cdot f(n)\}$$



$f(n) = n^2$

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

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



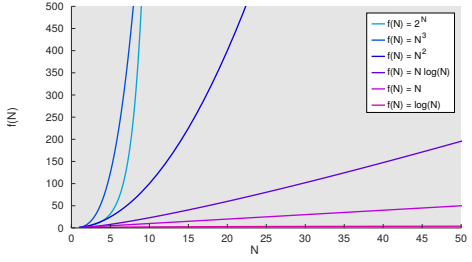
$c \cdot f(n) = 11\log(n)$

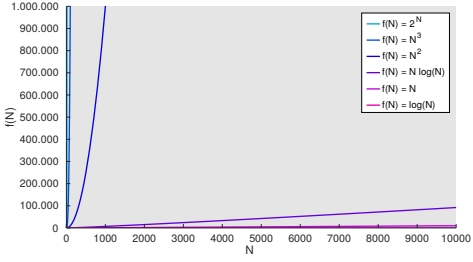
$g(n) = 2\log(n^5)$

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

$f(n) = log(n)$

Wachstumsordnungen



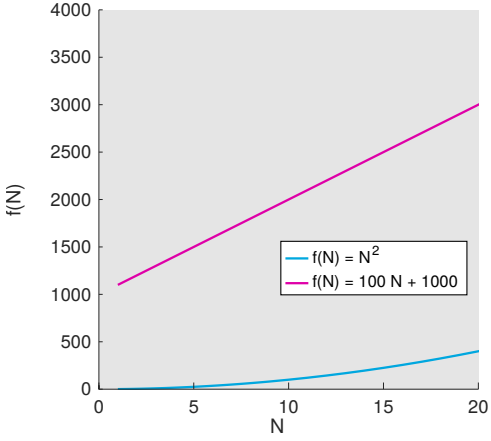


- Wertebereich für N: klein
- Wertebereich für N: groß
- Der Eindruck hängt auch stark vom gewählten Bereich auf der y-Achse ab.

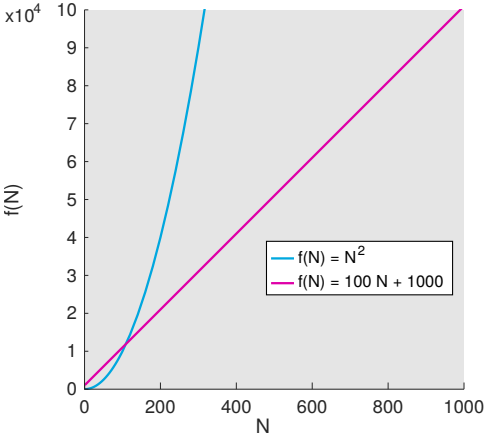
Der Einfluss von *Offset* und konstanten Faktoren

- Die Definition der Wachstumordnungen ignoriert konstante Faktoren und Offset (Verschiebung auf der y-Achse).
- Daher kann ein Vergleich, der nur auf Wachstumsordnungen beruht, irreführend sein. Aber:
- Wenn es um '**große** Eingaben' geht (z.B. um das Sortieren von > 10.000 Werten) ist tatsächlich fast nur die Wachstumsordnung relevant.
- Große Faktoren oder Offsets sind in der Rechenzeit von Algorithmen selten.
- Als Beispiel für den Einfluss vergleichen wir die Funktionen
  - $100N + 1000$ (kleinere Wachstumsordnung, großer Faktor und Offset) und
  - N^2 (größere 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:

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

- 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 $T \in \boldsymbol{\Theta}(N^2)$.
- Dies wird auch kurz $10N^2 + 5N + 27 \in \boldsymbol{\Theta}(N^2)$ geschrieben.

Wachstumsordnungen von Laufzeit und Speicherbedarf

- $\boldsymbol{O}(f)$, $\boldsymbol{\Theta}(f)$ und $\boldsymbol{\Omega}(f)$ werden benutzt, um Laufzeit und Speicherbedarf zu charaterisieren.
- Beträgt eine Laufzeit $T(N) = 15\,N^2 + 3\,N + 10$ Sekunden (bei Eingabegröße N), dann ist die **Laufzeit in $\boldsymbol{O}(N^2)$**, da $T(N) \in \boldsymbol{O}(N^2)$.
- Einheit (ob Millisekunden, Minuten oder Tage) irrelevant – wird durch konstanten Faktor c in der Definition von $\boldsymbol{O}(f)$ ausgeglichen
- Analog: Speicherbedarf als Wachstumordnung

Laufzeitanalyse

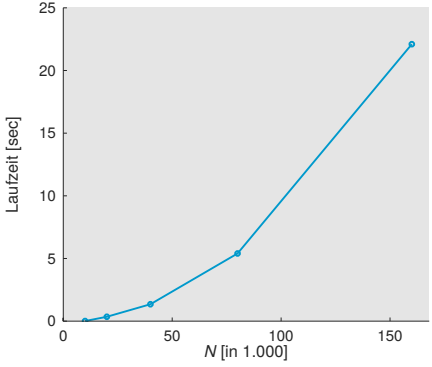
- Laufzeit in Abhängigkeit von der Eingabe, z.B. Länge N einer Liste, die sortiert wird
- Ggf. sind mehrere Eingabegrößen relevant, z.B. die Anzahl der Knoten V und Kanten E eines Graphen.
- Bei manchen Algorithmen spielt nicht nur die Anzahl der Eingabedaten sondern auch die konkreten Werte oder ihre Reihenfolge eine Rolle.
- Unterscheide Laufzeit im Durchschnitt und im *Worst Case*. Siehe auch die amortisierte Laufzeitanalyse, S. 52.
- Beispiel: Laufzeitanalyse der Methode `TwoSum.count(int[] a)` zunächst empirisch, dann analytisch.
- Sie zählt in dem `int` Array `a` die Paare `a[i]`, `a[j]`, deren Summe `0` ergibt.

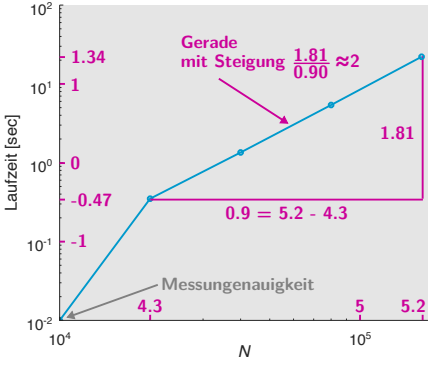
Empirische Laufzeitanalyse für TwoSum

```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 log-log 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 log-log 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. (TwoSum geht schneller: VL 11!)

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 54.

Syntaktische und Semantische Gleichheit von Objekten

- **Syntaktische Gleichheit**: `x == y`
- prüft bei Referenztypen, ob x und y auf dieselbe Adresse im Speicher referenzieren.
- 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.
- **Semantische Gleichheit**: `x.equals(y)`
- wird von `Object` als syntaktische Gleichheit vererbt und
- Sollte für eigene Klassen überschrieben werden (IDEA Automatismus).
- Dabei entsprechend `hashCode()` überschreiben, siehe VL 11.

Collections in Java

- Eine *Collection* ist eine Datenstruktur, die Speicherung von und Zugriff auf viele Objekte gleichen Typs erlaubt (Alternativen zu einfachen Arrays).
- In Java werden viele Varianten von ***Collections*** zur Verfügung gestellt.
- Für diese Veranstaltung sind **`LinkedList`** als Stack und Queue, **`PriorityQueue`** und in Vorlesung #11 `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.
- Man könnte dies für alle Methoden eines `Stack` erwarten. Aber `contains()` hat 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.
- Bei der Benutzung von Methoden in den Java Collections sollte immer auf deren Laufzeit geachtet werden (siehe auch Seite 54ff) und im Skript.

Rückschau

Nach dieser Vorlesung sollten Sie folgende Konzepte verinnerlicht haben:

- Debugging
- Referenztypen (Besonderheit bei Zuweisungen!)
- Vererbungsmechanismus von Java mit seinen Einschränkungen
- Überschreiben von Methoden bei der Vererbung
- rein und partiell abstrakte Klassen
- *Unified Modeling Language* (UML)
- Generische Typen
- Schnittstellenvererbung, *subclassing* vs. *subtyping*
- Schnittstellen `Iterable` und `Iterator`.
- Implementation: Stapel
- Wrappertypen, *autoboxing*, *unboxing*
- *Collections* in Java
- Empirische und Analytische Laufzeitanalyse

Anhang

Inhalt des Anhangs:

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.
- Dynamische Anpassung der Größe: z.B. verdoppeln, wenn Array voll
- Schrittweises Hinzufügen von Elementen hat im Fall der Array Vergrößerungen deutlich längere Laufzeiten (Kopieren der Daten).
- Die **amortisierten Laufzeitanalyse** mittelt über die unterschiedlichen Fälle.

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.
- Annahme oBdA: N ist Zweierpotenz
- Gesamtanzahl der Schleifendurchläufe beim schrittweisen Hinzufügen von N Elementen:

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

- Durchschnittlich $\frac{2N-1}{N}$, also knapp 2 Schleifendurchläufe pro `push()`
- 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 andere 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 Folie 28 angegebene Implementation von `Stack`

Stack (*worst case*)	
`push()`	$\boldsymbol{O}(1)$
`pop()`	$\boldsymbol{O}(1)$

Der Zusatz *worst case* bei der Laufzeit ist insbesondere eine Abgrenzung zu einer *amortisierten Laufzeit* (siehe Folie 52), 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.

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 einer Vielzahl von Tipps auf `tex.stackexchange.com` und anderen Internetseiten.

Index