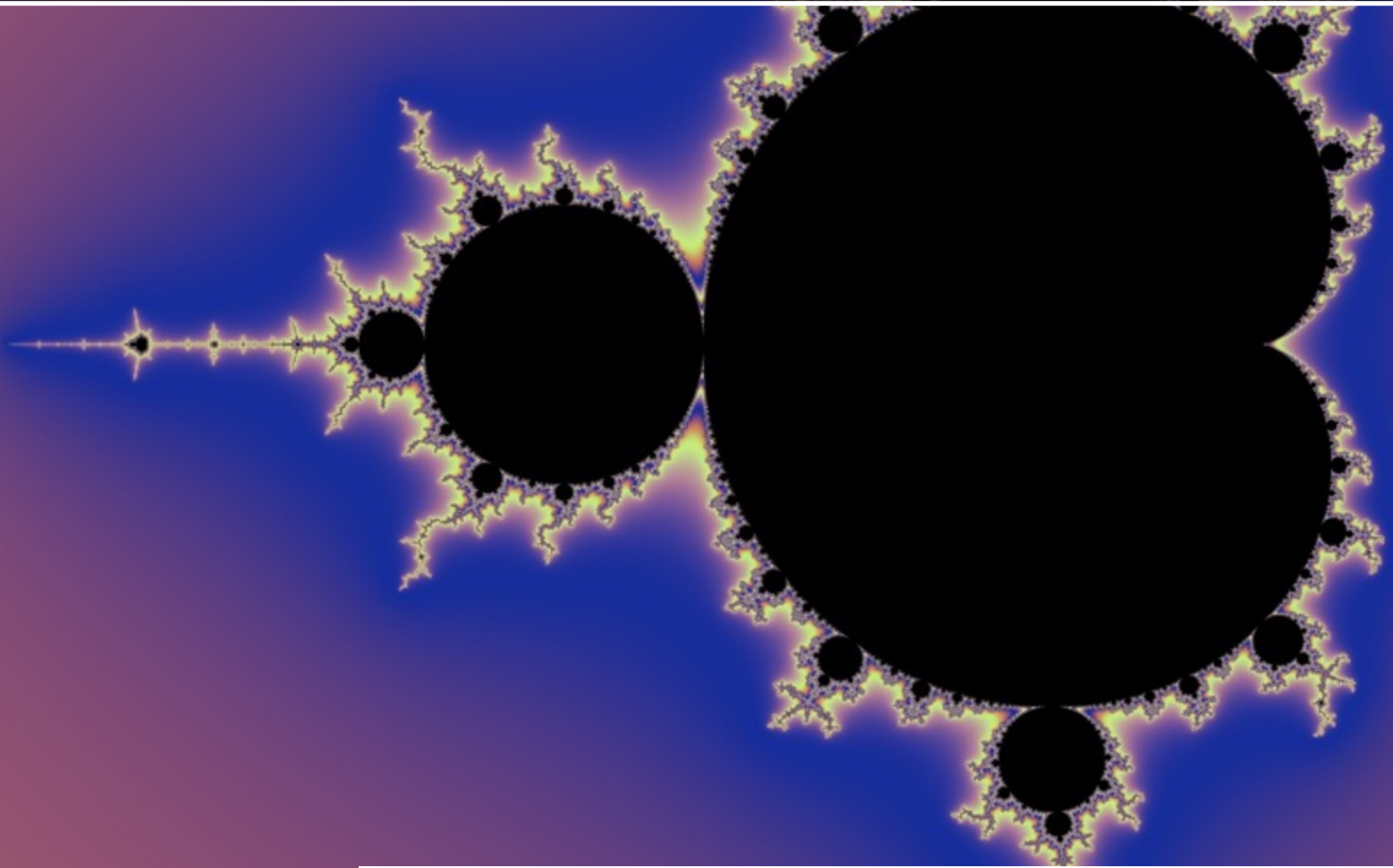


Software-Entwicklung 1

V08: Rekursion und Strings



Prof. Maalej & Team @maalejw



Status der 7. Übungswoche

Zeit	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
Vor mittag	Gruppe 1 Erfüllt: 64%	Gruppe 3 Erfüllt: 62%	Gruppe 5 Erfüllt: 61%	Gruppe 6 Erfüllt: 69%	Gruppe 8 Erfüllt: 55%
Nach mittag	Gruppe 2 Erfüllt: 74%	Gruppe 4 Erfüllt: 59%	Vorlesung	Gruppe 7 Erfüllt: 54%	

Überblick

1

Rekursion

2

Aufrufstack und Heap

3

Zeichenketten - Strings

4

Reguläre Ausdrücke


Beispiel: Fakultät mit Rekursion

- Fakultät $n!$ ist das Produkt aller natürlichen Zahlen von 1 bis n .
- $4! = 1 * 2 * 3 * 4 = 24$
- Definition:

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ n * (n - 1)! & \text{für } n > 0 \end{cases}$$

Rekursive Definition

Rekursiver Aufruf



```
public int fakultaet(int n)
{
    int result;
    if (n == 0)
    {
        result = 1;
    }
    else
    {
        result = n * fakultaet(n-1);
    }
    return result;
}
```

Kontrollfluss in der Rekursion

$n = 3$

```
public int fakultaet (int n)
{
    int result;
    if (n == 0) {
        result = 1;
    }
    else {
        result = n * fakultaet(n-1);
    }
    return result;
}
```

$n = 2$

```
public int fakultaet (int n)
{
    int result;
    if (n == 0) {
        result = 1;
    }
    else {
        result = n * fakultaet(n-1);
    }
    return result;
}
```

$n = 1$

```
public int fakultaet (int n)
{
    int result;
    if (n == 0) {
        result = 1;
    }
    else {
        result = n * fakultaet(n-1);
    }
    return result;
}
```

$n = 0$

```
public int fakultaet (int n)
{
    int result;
    if (n == 0) {
        result = 1;
    }
    else {
        result = n * fakultaet(n-1);
    }
    return result;
}
```

6

2

1

1

5

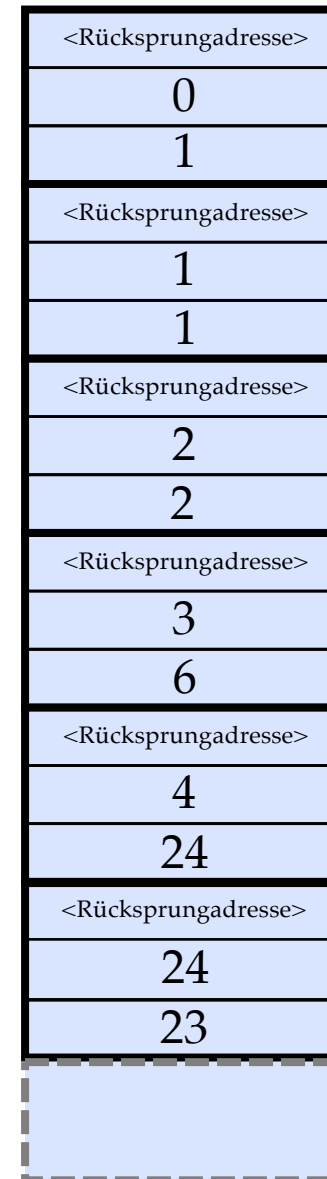
Aufrufstack

- **Aufrufstack** (engl.: call stack oder function stack)
- Speichert **zur Laufzeit** Informationen über die gerade aktiven Methoden in **Stackframes**
- Bei Methodenaufruf werden die **Rücksprung-adresse** und die **lokalen Variablen** in einem neuen Stackframe auf dem Stack gespeichert
- Bei Terminierung, wird der zugehörige Stackframe **vom Stack geräumt**
- In Java nicht zugänglich



Aufrufstack bei Rekursion

- **Rekursive Methodenaufrufe:**
 - Jeder rekursive Aufruf erzeugt einen Stackframe
 - Jeder Rekursionsstufe arbeitet auf ihren eigenen lokalen Variablen und gibt ein Ergebnis zurück
- Beispielausdruck: **23 + fakultaet(4)**
- Jeder Aufruf legt folgende Informationen auf dem Stack ab:
 - Platz für Ergebnis
 - Argument n
 - Rücksprungadresse in die rufende Methode



ein Stackframe
für **fakultaet**

Stackframe der
Klientenmethode,
die den Ausdruck
enthält

Lebensdauer lokaler Variablen

n = 3

```
public int fakultaet (int n)
{
    int result;
    if (n == 0) {
        result = 1;
    }
    else {
        result = n * fakultaet(n-1);
    }
    return result;
}
```

n = 2

```
public int fakultaet (int n)
{
    int result;
    if (n == 0) {
        result = 1;
    }
    else {
        result = n * fakultaet(n-1);
    }
    return result;
}
```

Diese lokale Variable lebt vier
Methodenausführungen lang...

n = 1

```
public int fakultaet (int n)
{
    int result;
    if (n == 0) {
        result = 1;
    }
    else {
        result = n * fakultaet(n-1);
    }
    return result;
}
```

n = 0

```
public int fakultaet (int n)
{
    int result;
    if (n == 0) {
        result = 1;
    }
    else {
        result = n * fakultaet(n-1);
    }
    return result;
}
```

...während diese nur für
eine Ausführung lebt.

Fakultät Berechnung iterativ

- Rekursive Programme haben **oft kein** gutes Speicher und Ablaufverhalten
- Jedes mal wird ein neues Segment auf dem Aufrufstack belegt
- Vergleichsweise hoher Aufwand
- Iterative Methode:

```
public int fakultaet (int n)
{
    int fakl = 1;
    for (int i = 1; i <= n; ++i)
    {
        fakl = i * fakl;
    }
    return fakl;
}
```



Rekursion allgemein

- Eine Methode *m* ruft während der Ausführung ihres Rumpfes sich selber erneut auf
- Dieser Prozess muss zwingend eine Abbruchbedingung haben
- Wir unterscheiden bei Rekursionen:
 - **Direkt**, wenn eine Methode *m* sich im Rumpf selbst ruft
 - **Indirekt**, wenn eine Methode *m1* eine andere Methode *m2* ruft, die aus ihrem Rumpf *m1* aufruft
- Grundgedanke der Rekursion
 - Die Methode löst einen **kleinen Teil des Problems** selbst
 - Der Rest wird in **kleinere Probleme zerlegt** und ruft sich selbst mit diesen kleineren Problemen auf

Grundstruktur der Rekursion

```
public <Ergebnistyp> loeseProblem ( <formale Parameter> )
{
    if ( <ProblemEinfachLösbar> ) ← Abbruchbedingung
    {
        return <EinfachesErgebnis>
    }
    else
    {
        <zerlegeProblem>
        <Ergebnis1> = loeseProblem ( <veränderteParameter> );
        <Ergebnis2> = loeseProblem ( <veränderteParameter> );
        ...
        return <ausgewerteteErgebnisse> ;
    }
}
```

rekursive Aufrufe

Fibonacci-Zahlen mit Rekursion

Definition:

- Erste Fibonacci-Zahl ist 0
- Zweite Fibonacci-Zahl ist 1
- n-te Fibonacci-Zahl ist Summe aus (n-1)ten und (n-2)ten Fibonacci-Zahl

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Rekursive Definition

$$\text{fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{für } n > 1 \end{cases}$$

```
public int fibonacci (int n)
{
    switch (n) {
        case 0: return 0;
        case 1: return 1;
        default: return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

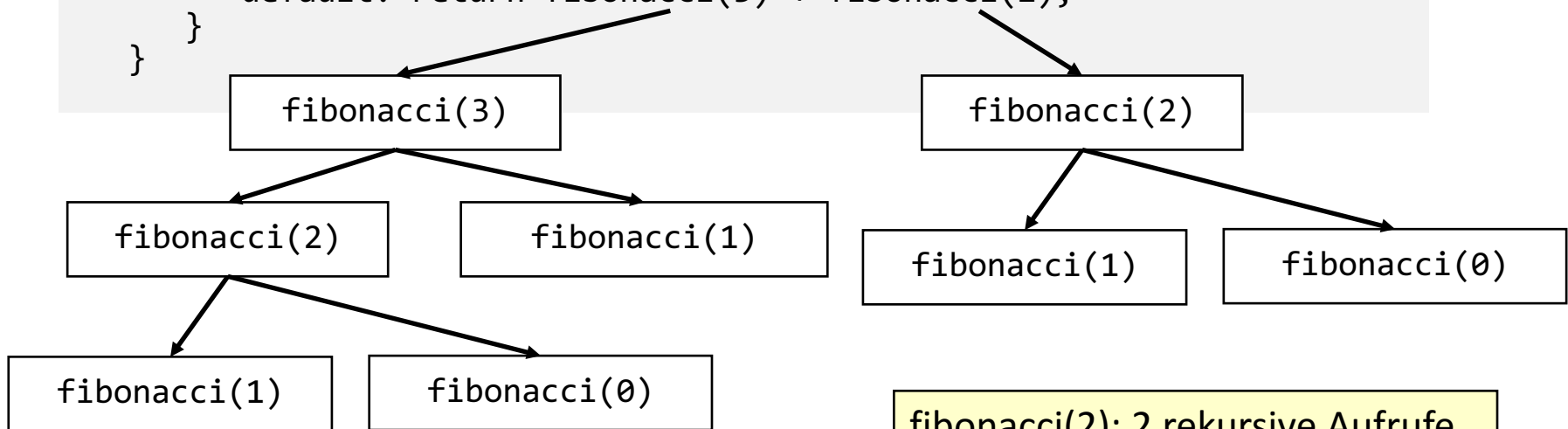


Wo ist das Problem?

Problem bei Fibonacci-Zahlen



```
public int fibonacci(4)
{
    switch(4) {
        case 0: return 0;
        case 1: return 1;
        default: return fibonacci(3) + fibonacci(2);
    }
}
```



fibonacci(2): 2 rekursive Aufrufe
fibonacci(3): 4 rekursive Aufrufe
fibonacci(4): 8 rekursive Aufrufe
fibonacci(5): 13 rekursive Aufrufe
...

Anwendungsbereiche von Rekursion

- Rekursion ist besonders in folgenden Fällen geeignet:
 - Auf **rekursiv definierten Strukturen** z.B. Baumstrukturen in der Informatik (Syntaxbäume, Entscheidungsbäume, Verzeichnisbäume...)
 - Viele sehr gute **Sortierverfahren** sind rekursiv definiert (z.B. Quicksort und Mergesort)
 - Viele Probleme auf **Graphen** lassen sich elegant rekursiv lösen

Rekursion: Stärken und Schwächen

- Rekursion kann für eine relativ **kleine Menge** von Problemen sehr **einfache, elegante** Lösungen produzieren
- Rekursion kann für eine etwas **größere Menge** von Problemen sehr **schwer zu verstehende** Lösungen produzieren
- Für die meisten Probleme führt die Benutzung von Rekursion zu **sehr komplizierten** Lösungen – in solchen Fällen sind simple Iterationen meist verständlicher
- Sollte sehr selektiv eingesetzt werden

Überblick

1

Rekursion

2

Aufrufstack und der Heap

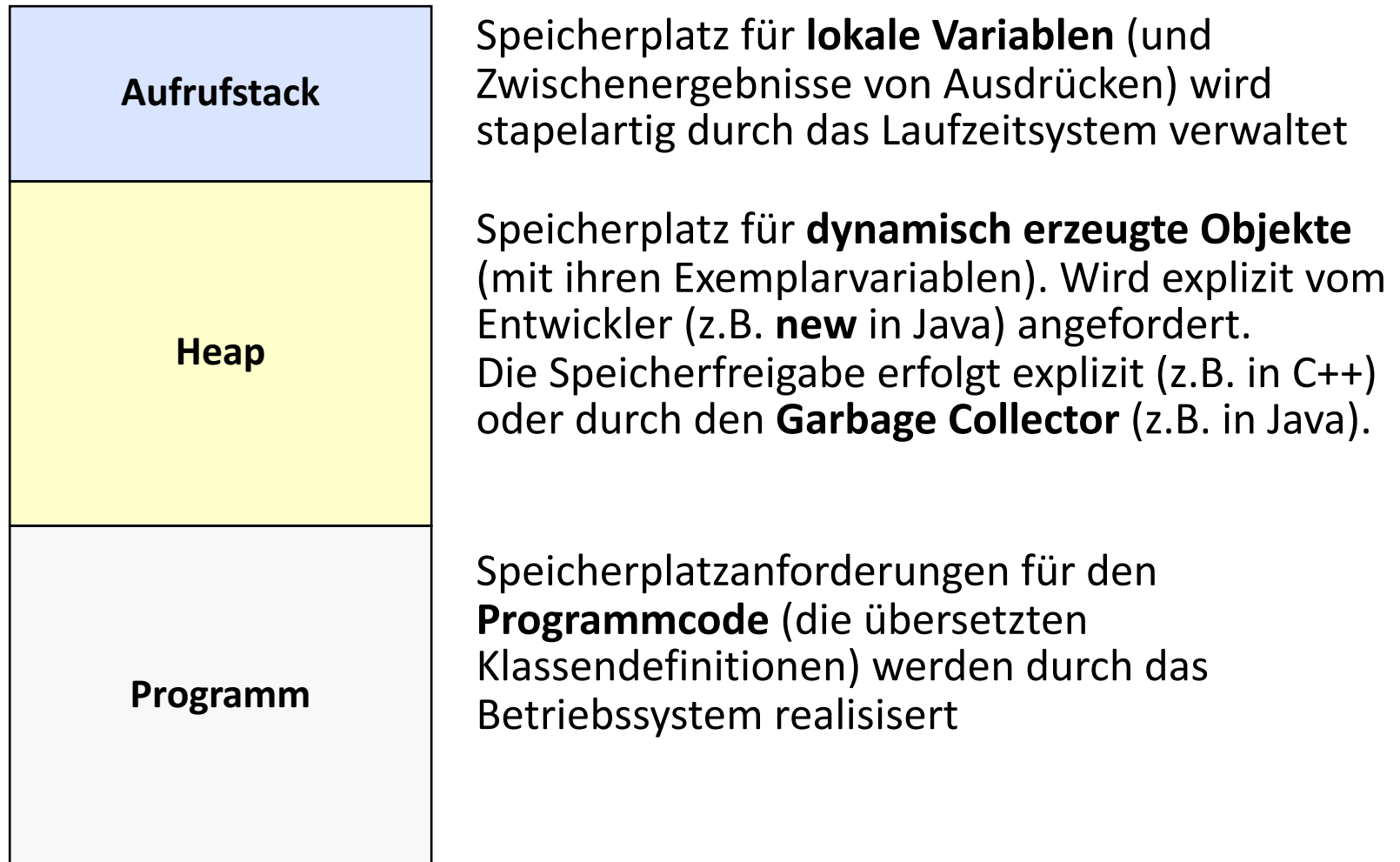
3

Zeichenketten - Strings

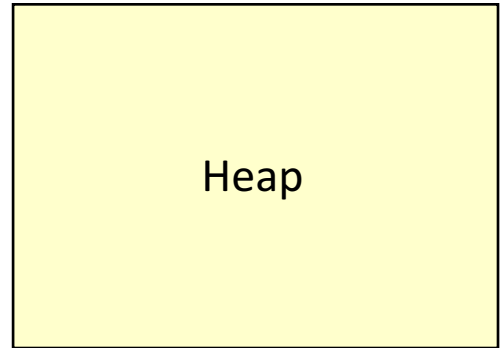
4

Reguläre Ausdrücke

Vereinfachtes Speichermodell



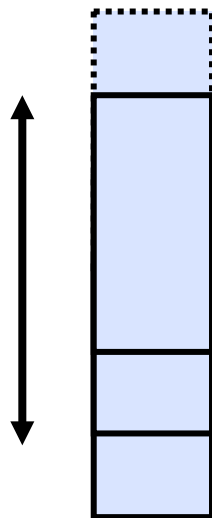
Heap



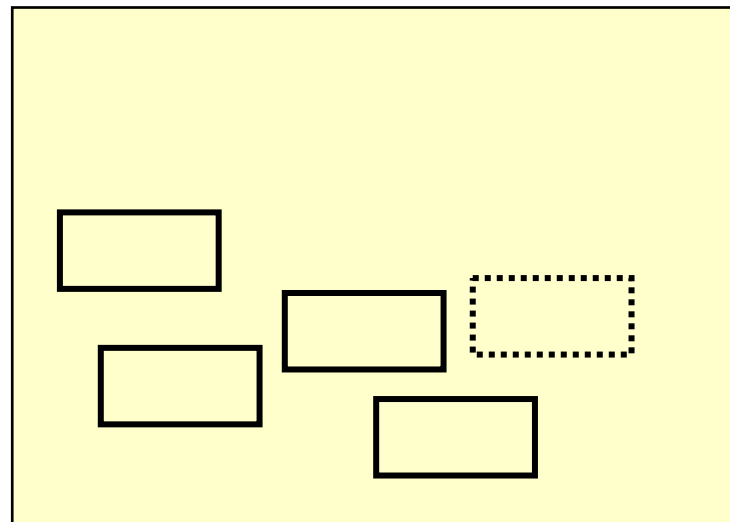
- **Dynamische Speicher**, auch **Heap**
(engl. für *Halde*, *Haufen*) ist ein Speicherbereich, aus dem zur Laufzeit eines Programmes zusammenhängende Speicherabschnitte angefordert und in **beliebiger Reihenfolge** wieder freigegeben werden können
- Die Freigabe kann sowohl manuell als auch mit Hilfe einer automatischen Speicherbereinigung (engl.: garbage collection) erfolgen
- Eine Speicheranforderung vom Heap wird auch **dynamische Speicheranforderung** genannt
- Kann eine Speicheranforderung wegen Speichermangel nicht erfüllt werden, kommt es zu einem Programmabbruch (in Java: **OutOfMemoryError**)

Heap und Aufrufstack

- Beim Aufrufstack werden angeforderte Speicherabschnitte **strikt in der umgekehrten Reihenfolge** wieder freigegeben werden, in der sie angefordert wurden
- Beim Aufrufstack spricht man deshalb auch von **automatischer Speichieranforderung** (weniger Laufzeitkosten als bei dynamischer Speichieranforderung)
- Bei Spezialfällen kann der für den Stack reservierte Speicher ausgehen - dann droht ein Programmabbruch wegen **Stapelüberlauf** (in Java: **StackOverflowError**).



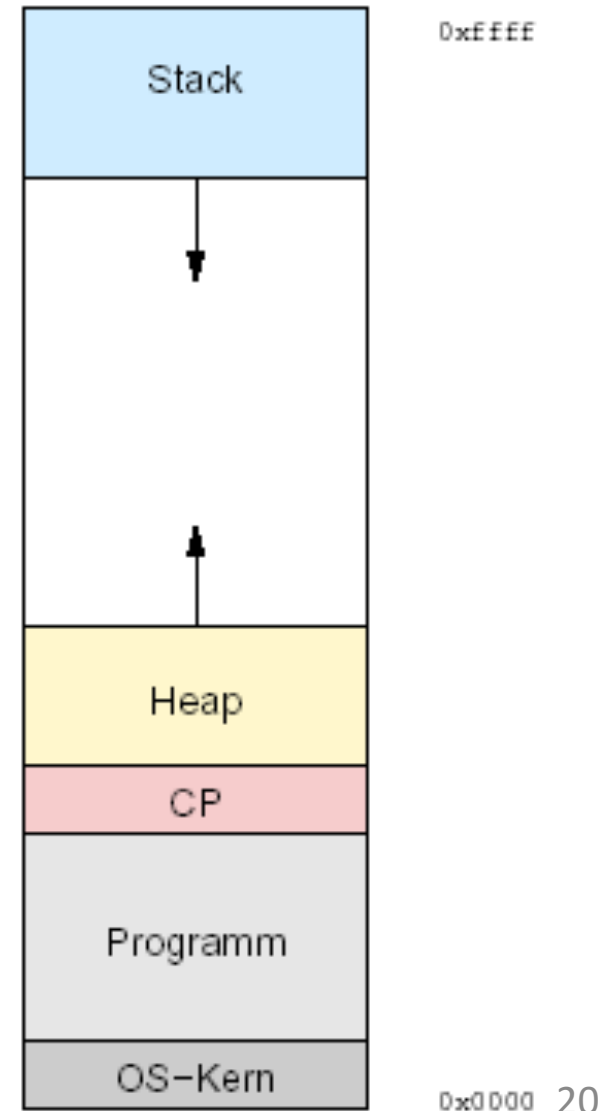
Aufrufstack



Heap

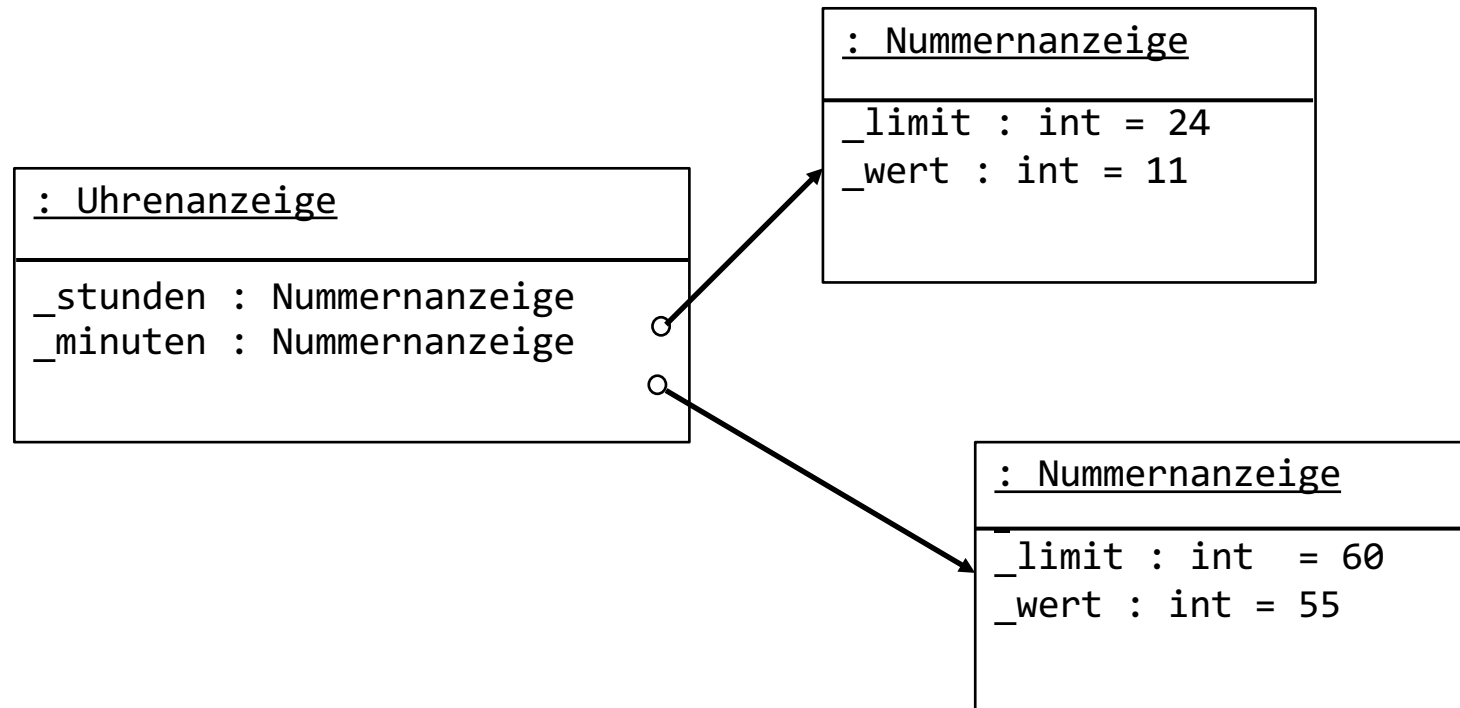
Speichereinteilung in einem Unix-System

- **Programm**
 - Programmtext mit allen Befehlen
 - Textsegment bleibt unverändert
- Constant Pool (CP)
 - Alle Konstanten und statischen Variablen des Programms
- **Heap**
 - Alle dynamisch zur Laufzeit des Programms erzeugten Variablen bzw. Objekte
- **Stack**
 - Parameterübergabe zwischen Funktionen
 - Speicherung der lokalen Variablen der einzelnen Funktionen



Objektdiagramme: Schnappschüsse vom Heap

- Ein Objektdiagramm in Java ist ein Schnappschuss vom **Heap** eines laufenden Programms
- Es zeigt einen Ausschnitt des Objektgeflechts zur Laufzeit in der **Virtual Machine**, um einen bestimmten Aspekt zu verdeutlichen



Garbage Collector in Java

Voraussetzungen:

- **Alle** Objekte eines Java-Programms liegen im Heap
- Auf dem **Aufrufstack** in den Speicherplätzen für die lokalen Variablen liegen entweder primitive Werte oder **Referenzen auf Objekte**
- Nur Objekte, die **vom Aufrufstack** aus **erreichbar** sind für die Programmausführung relevant
- Alle anderen Objekte im Heap sind „tote“ Objekte



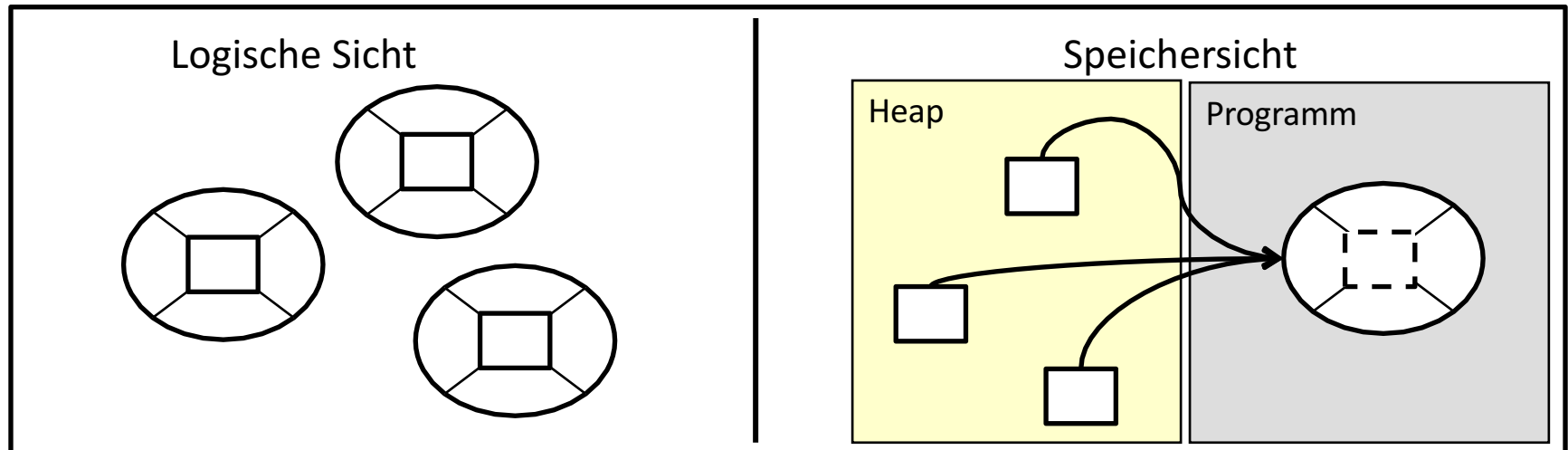
Vorgehen des Garbage Collectors

- Beobachtet regelmäßig die Referenzen auf dem Stack
- Von da aus wird transitiv das **gesamte Objektgeflecht** durchgeschaut und **markiert** erreichbare Objekte werden markiert
- Anschließend werden alle nicht markierten Objekte im Heap **gelöscht**
- Das Vorgehen aus **Markieren** und **Abräumen** heißt im Englischen **Mark and Sweep**

Methoden und Zustandsfelder zur Laufzeit

- Zur **Übersetzungszeit** gibt es jede **Methode** und **Exemplarvariable** nur einmal
- Sie sind statisch in den **Klassendefinitionen** beschrieben
- Zur **Laufzeit** gibt es für jedes Exemplar einer Klasse einen eigenen Satz Zustandsfelder und logisch auch einen Satz Methoden
- Dass ein Satz von Methoden (in der Klasse abgelegt) für alle Exemplare einer Klasse ausreicht, ist eine Optimierung

Drei Exemplare einer Klasse zur Laufzeit



Zusammenfassung

1

Rekursive Methodenaufrufe sind eine alternative Möglichkeit für Wiederholungen

2

Jede Wiederholung lässt sich **sowohl iterativ als auch rekursiv** formulieren, jeweils mit spezifischen Vor- und Nachteilen

3

Ziele wie **Verständlichkeit und Sicherheit** spielen bei der Wahl einer geeigneten Realisierung eine wichtige Rolle

4

„Hinter den Kulissen“ moderner Programmiersprachen sind der **Aufrufstack** und der **Heap** zentrale Strukturen für die Verwaltung von Variablen und Objekten

Überblick

1

Rekursion

2

Aufrufstack und der Heap

3

Zeichenketten - Strings

4

Reguläre Ausdrücke

Zeichenketten (Strings) in Programmiersprachen

- Zeichenkette ist eine **Folge** von einzelnen **Zeichen**.

0	1	2	3	4	5	6	7	8	9
4	.		A	d	v	e	n	t	?

- **Anzahl der Zeichen** in der Zeichenkette ist die **Länge**
 - Konzeptuell sind Zeichenketten in ihrer Länge unbegrenzt
 - In einigen Kontexten (z.B. Datenbanken) müssen Zeichenketten jedoch eine fest definierte Maximallänge haben.
- Zeichenketten sind notwendig in Anwendungen, in denen Texte (Prosa, Quelltexte, etc.) verarbeitet werden
- In objektorientierten Sprachen werden Zeichenketten als Objekte modelliert

Datentyp: **Zeichenkette**

Wertemenge: { Zeichenketten beliebiger Länge }

Operationen: Länge, Subzeichenkette, ...

String



- In Java werden Zeichenketten primär durch die Klasse String unterstützt
- Diese Klasse definiert, wie alle Klassen, einen Typ
- String ist in Java ein expliziter Bestandteil der Sprache, denn es gibt einige Spezialbehandlungen für diesen Typ:
 - String-Literale werden erkannt

```
String s = "Banane";
```

- Infix-Operator + kann auch auf Strings angewendet werden.
- Javadoc-Darstellung für die Klasse String:
<http://docs.oracle.com/javase/7/docs/api/java/lang/String.html>

Datentyp: **String**

Wertemenge: { String-Exemplare beliebiger Länge }

Operationen: **length, concat, substring, charAt, ...**

Escape-Sequenzen in String-Literalen

- Angenommen, wir wollen folgendes ausgeben:
 - Bitte einmal "Aaah" sagen!

```
System.out.println("Bitte einmal "Aaah" sagen!");
```

- **Problem:** Compiler sieht zwei String-Literale, getrennt von dem unbekannten Bezeichner Aaah
- **Lösung:** Escape-Sequenz für Anführungszeichen

```
System.out.println("Bitte einmal \"Aaah\" sagen!");
```



Gewünschtes Zeichen	Escape-Sequenz
Anführungszeichen	\"
Backslash	\\
Zeilenumbruch	\n

Strings sind unveränderlich



- Die Klasse String in Java definiert Objekte, die unveränderliche Zeichenketten sind:
 - Operationen auf Strings liefern Informationen über ein String-Objekt
 - **Sie verändern es niemals**
 - Der Infix-Operator + verkettet zwei Strings zu einem neuen String
- Strings sind damit sehr untypische Objekte in Java, denn sie haben einen **unveränderbaren** Zustand.



Typischer Fehler:

```
String s = „FckW“;  
s.toUpperCase(); // Das Ergebnis dieses Aufrufs verpufft
```

Gleichheit von Strings

- Da Strings Java Objekte sind, werden mit dem Operator == lediglich Referenzen verglichen
- Zwei String-Objekte können dieselbe Zeichenkette repräsentieren, sind aber dennoch verschiedene String-Exemplare
- Deshalb: **Strings** in Java immer mit der **equals-Methode** vergleichen!

```
"Banane" == "Banane"
```

```
"Banane" == new String("Banane")
```

Problem: Datentypen String und Zeichenkette sind nicht gleichzusetzen

Datentyp: **Zeichenkette**

Wertemenge: { Zeichenketten beliebiger Länge }

Operationen: Länge, Subzeichenkette, ...

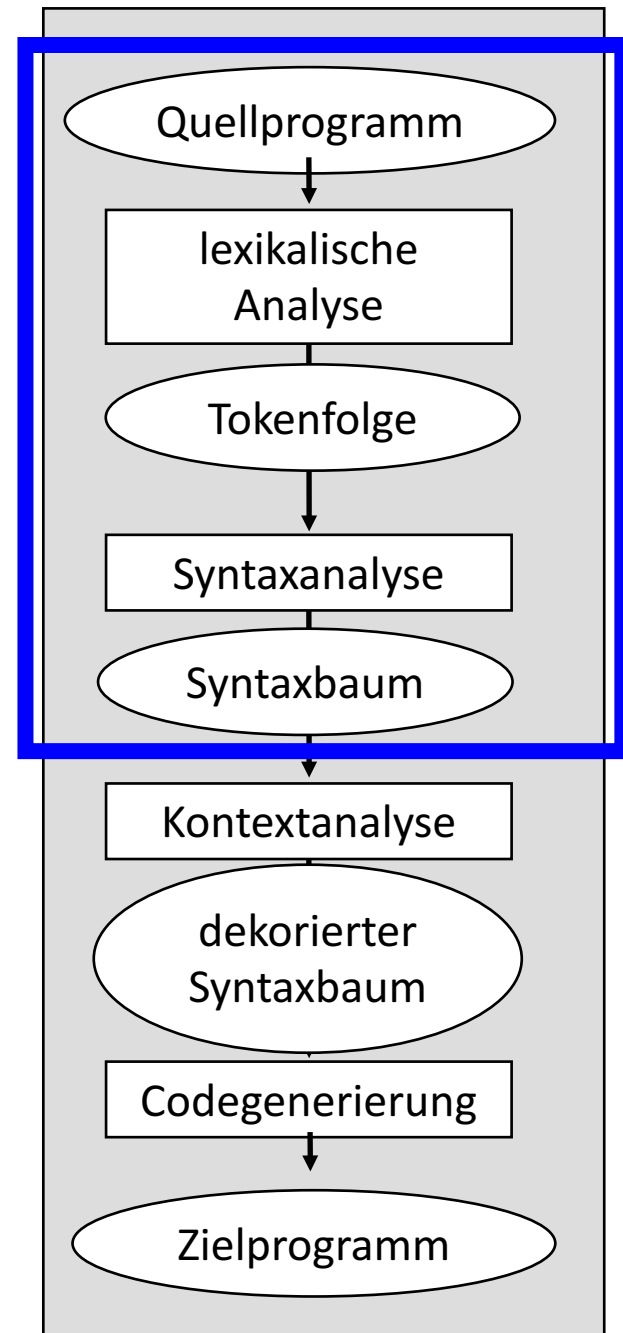
Datentyp: String

Wertemenge: { String-Exemplare beliebiger Länge }

Operationen: length, concat, substring, charAt, ...

Compiler

- Die Übersetzung von Programmen erfordert meist mehrere Schritte:
- Der für Menschen lesbare Quelltext wird durch einen Scanner in eine Folge von **Token** zerlegt (lexikalische Analyse)
- Ein Parser erzeugt aus der Tokenfolge einen Syntaxbaum (Syntaxanalyse)
- Dieser wird analysiert und ggf. dekoriert
- Daraus erzeugt der Codegenerator das ausführbare Maschinenprogramm (bzw. Zielprogramm) und optimiert ggf



Syntaktische Grundelemente

- **Token** sind die kleinsten syntaktischen Einheiten einer Sprache
- Bei Programmiersprachen:
 - Bezeichner, Literale, Operatoren, reservierte Wörter und Sonderzeichen wie Klammern und Semikolon

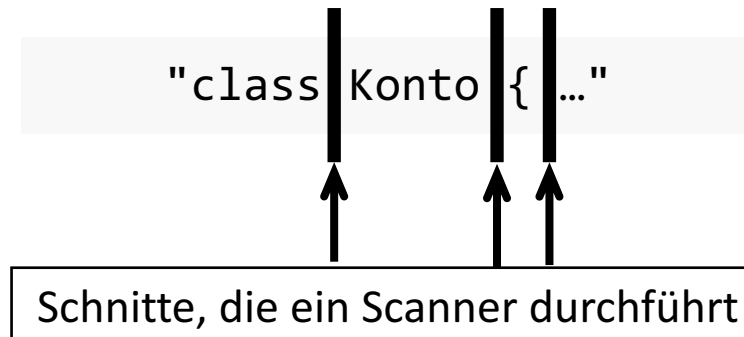
Gültige Token

nextItem	<=
3.1416	while

Ungültige Token

next-item	><
3,1416	{ }

- Lexikalische Analyse eines Compilers (Zerlegen eines Quelltextes in eine Folge von Tokens) lässt sich mit Hilfe von regulären Ausdrücken steuern



„Bezeichner“ ist ein Token



- Bezeichner werden verwendet, um Variablen, Methoden, Klassen etc. zu benennen
- Vereinfachte Definition:
 - *Ein Bezeichner besteht aus einem Buchstaben (ein Unterstrich wird auch als ein Buchstabe angesehen), gefolgt von beliebig vielen Buchstaben und Ziffern.*
- Ist eine Zeichenkette *s* (vom Typ String) ein gültiger Bezeichner?
- Abstrakt gefragt: Ist *s* ein Element der Menge aller gültigen Bezeichner?

```
s.matches(mengeGueeltigerBezeichner)
```

- Die Methode `matches` ist in der Klasse String definiert und erhält als Parameter einen **regulären Ausdruck** als String
- Ein regulärer Ausdruck beschreibt eine Menge von Zeichenketten
- **matches** gibt **true** genau dann, wenn die Zeichenkette *s* ein Element dieser Menge ist, ansonsten **false**

Überblick

1

Rekursion

2

Aufrufstack und der Heap

3

Zeichenketten - Strings

4

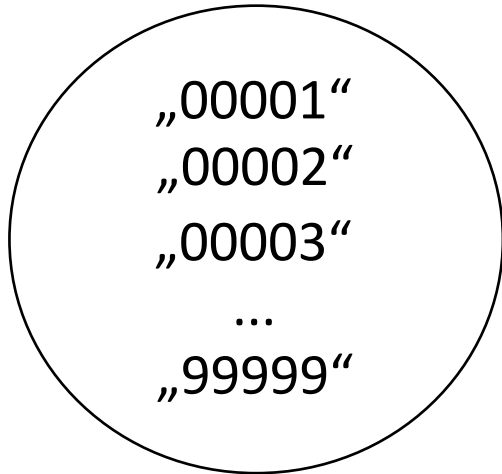
Reguläre Ausdrücke

Reguläre Ausdrücke



- Engl. *regular expression*
- Repräsentiert eine Menge von Zeichenketten

„[0-9]{5}“



“[a-z0-9_-]{3,16}“

Benutzernamen

- 3-16 Zeichen
- Buchstaben a-z
- Ziffern 0-9
- Sonstige Zeichen: _ und -

“([a-z0-9_\\.-]+)@([\\da-z\\.-]+)\\.([a-z\\.]{2,6})“

Alle syntaktisch gültigen E-Mailadressen

Bezeichner als regulärer Ausdruck

- Java-Bezeichners als regulären Ausdruck:

```
String mengeGueltigerBezeichner = "[a-zA-Z_][a-zA-Z_0-9]*";
```

- Wenn an einer bestimmten Stelle eines aus einer Menge einzelner Zeichen möglich sein soll, dann können diese Zeichen in **eckigen Klammern** angegeben werden:
 - h[oa]se beispielsweise definiert die reguläre Menge { hose, hase }.
- Zur weiteren Verkürzung erlaubt Java in den eckigen Klammern auch Bereichsangaben mit einem **Minuszeichen**. Beispielsweise:
 - se[1-3] definiert die reguläre Menge { se1, se2, se3 }
 - [a-z] definiert alle Kleinbuchstaben von a bis z

Bezeichner als regulärer Ausdruck

[a-zA-Z_][a-zA-Z_0-9]*

beliebig oft
wiederholt

ein Zeichen aus der Menge
der Buchstaben

ein Zeichen aus der Menge der
Buchstaben und Ziffern

```
String mengeGueltigerBezeichner = "[a-zA-Z_][a-zA-Z_0-9]*";
```

- Der * ist ein Postfix-Operator und besagt in einem regulären Ausdruck, dass sein Operand beliebig oft auftreten kann (auch gar nicht). In diesem Fall ist der Operand die zweite Menge, die Buchstaben und Ziffern definiert
- Java bietet zusätzlich die Postfix-Operatoren + (beliebige Wiederholung, mindestens einmal) und ? (entweder einmal oder gar nicht)
- Ein einzelner Punkt (.) in einem regulären Ausdruck steht für ein beliebiges Zeichen

Zeichenklassen in regulären Ausdrücken



Zeichenklasse	Beschreibung	Muster	Übereinstimmung
→ \w	Beliebiges Wortzeichen	\w	"l", "D", "a", "1", "3"
→ \W	Beliebiges Nicht-Wortzeichen	\W	" ", "."
→ \s	Leerraumzeichen	\w\s	"D ", "f "
→ \S	Nicht-Leerraumzeichen	\s\S	" _", " A", " q"
→ \d	Eine beliebige Dezimalziffer	\d	"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
→ \D	Nicht-Dezimalziffer	\D\D	" a", "e=", "D-", "lA", "VG"
→ .	Jedes beliebige Zeichen	.\w.	"#aG", "!f^", "aaa", ".f3", "{f}"

Quantifizierer in regulären Ausdrücken



Postfix-Operator	Beschreibung	Muster	Übereinstimmungen
→ *	Entspricht dem vorangehenden Element beliebig oft (auch kein Mal)	<code>\d*\.\d</code>	<code>".0"</code> , <code>"19.9"</code> , <code>"219.9"</code>
→ +	Entspricht dem vorangehenden Element mindestens einmal	<code>"be+"</code>	<code>„be“</code> , <code>"bee"</code> , <code>"beee"</code> , ...
→ ?	Entspricht dem vorangehenden Element nicht oder einmal	<code>"rai?n"</code>	<code>"ran"</code> , <code>"rain"</code>
→ {n}	Entspricht dem vorangehenden Element genau n-mal	<code>“,\d{3}”</code>	<code>“,043”</code> , <code>“,876”</code> , <code>“,543”</code> , ...
→ {n,m}	Entspricht dem vorangehenden Element mindestens n-, höchstens jedoch m-mal	<code>“\d{3,5}</code>	<code>"166"</code> , <code>"17668"</code> <code>"19302"</code>

Weitere Beispiele für reguläre Ausdrücke



```
String s = "ab";  
s.matches("ab") // => true  
s.matches("a")  // => false  
s.matches("aba") // => false
```

```
String re = "[ab][ab]";  
  
s.matches(re) // => true  
// (auch true für s = "aa", "ba" oder "bb")
```

```
re = "(ab)*"; // ab beliebig oft wiederholt  
  
s.matches(re); // => true  
// (auch true für s = "abab", "ababab", ...)
```

- Ausführliche Beschreibung:
<https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>

Zeichenketten und reguläre Ausdrücke



- Viele weitere Formen der Analyse von Zeichenketten
- Weitere Methoden mit regulären Ausdrücken neben matches sind:

```
String replaceFirst(String regex, String replacement)
```

- Liefert eine neue Zeichenkette als Kopie, in der das **erste Vorkommen** einer der Zeichenketten, die durch regex beschrieben sind, durch replacement **ersetzt** ist

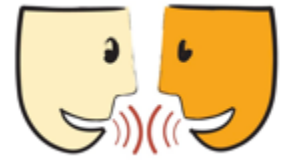
```
String replaceAll(String regex, String replacement)
```

- Liefert eine neue Zeichenkette als Kopie, in der **alle Vorkommen** von Zeichenketten, die durch regex beschrieben sind, durch replacement ersetzt sind

Ausführliche Beschreibung:

- <http://docs.oracle.com/javase/7/docs/api/java/lang/String.html>

Formale Sprachen



- Jede formale Sprache lässt sich verstehen als:
 - eine **Menge von Zeichenketten** eines **Alphabets**
- Durch **Grammatikregeln** wird definiert welche Zeichenketten des Alphabets, Wörter der Sprache (syntaktisch korrekt oder wohlgeformt)
- Eine Grammatik ist eine **Metasprache**, mit der eine andere Sprache beschrieben wird

Beispiel für ein **Alphabet**
(Vokabular, Zeichensatz):
Die Buchstaben von **a** bis **z**.

Beispiel für eine **formale Sprache**
über diesem Alphabet:
{a, aha, alter, aal, aabenra, aaarghh, ... }

informell:

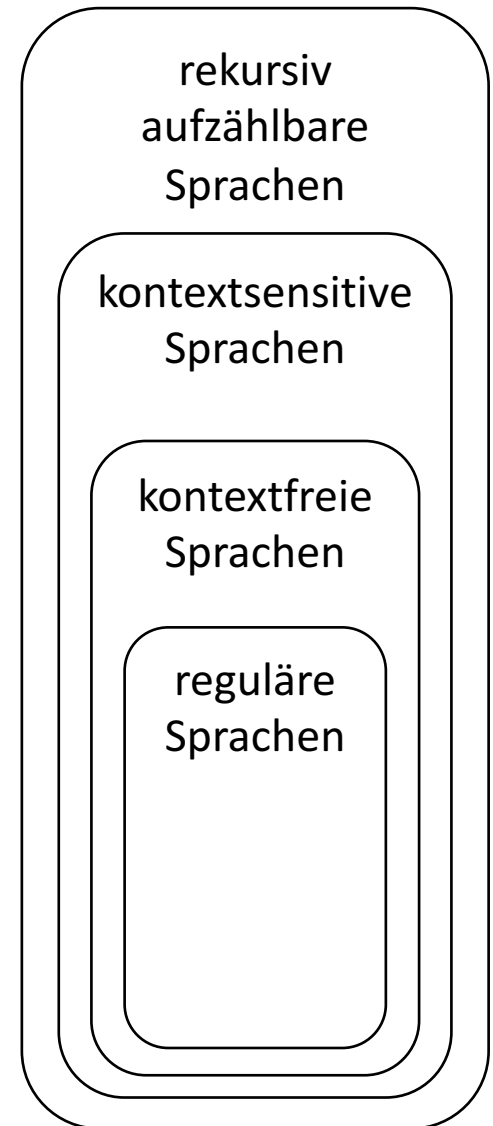
Menge aller Zeichenketten, die mit
mindestens einem a beginnen.

Wie können wir dies formaler fassen?

Grammatiken für Sprachen

- Der Linguist Noam Chomsky beschrieb Mitte der 50er Jahre sog. generative Grammatiken, um vier Klassen von Sprachen zu definieren:
 - reguläre, kontextfreie, kontextsensitive und rekursiv aufzählbare Sprachen
 - Reguläre Sprachen bilden die einfachste Klasse – jede höhere enthält die einfacheren
- Später zeigte sich:
 - Die Syntax von Programmiersprachen ist gut als kontextfreie Sprache beschreibbar
 - Die Token von Programmiersprachen können als **reguläre Sprachen** beschrieben werden

Mehr zur sog. Chomsky-Hierarchie sowie kontextsensitiven und rekursiv aufzählbaren Sprachen in FGI.



Reguläre Ausdrücke und reguläre Sprachen

- Mit regulären Ausdrücken (spezielle Form von Grammatik) können reguläre Sprachen/Mengen beschrieben werden.
- Reguläre Ausdrücke über einem Alphabet A und der durch sie beschriebenen regulären Mengen sind definiert als:
 - a mit $a \in A$ ist ein regulärer Ausdruck für die reguläre Menge $\{a\}$.
 - Sind p und q reguläre Ausdrücke für die regulären Mengen P und Q , dann ist:
 - $(p)^*$ ein regulärer Ausdruck, der die reguläre Menge P^* (Iteration, d.h. beliebig häufige Konkatenation mit sich selbst) bezeichnet,
 - $(p+q)$ ein regulärer Ausdruck, der die reguläre Menge $P \cup Q$ (Vereinigung) bezeichnet,
 - (pq) ein regulärer Ausdruck, der die reguläre Menge $P \bullet Q$ (Konkatenation) bezeichnet.
 - \emptyset ist ein regulärer Ausdruck, der die leere reguläre Menge bezeichnet.
 - ϵ ist ein regulärer Ausdruck, der die reguläre Menge $\{\epsilon\}$ bezeichnet, die nur aus dem leeren Wort ϵ besteht.

Beispiel eines regulären Ausdrucks

- Gegeben sei das Alphabet $\{a,b\}$ und die reguläre Menge $\{aa,ab,ba,bb\}$.
- Diese reguläre Menge wird beschrieben durch die regulären Ausdrücke:
 - $((((aa) + (ab)) + (ba)) + (bb))$ bzw.
 - $((a(a + b)) + (b(a + b)))$ bzw.
 - $((a + b)(a + b))$.

Mehr zu den theoretischen Grundlagen
regulärer Ausdrücke in FGI

Reguläre Ausdrücke in Java: fast wie in der Theorie

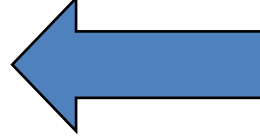
- Die pragmatisch in Programmiersprachen eingesetzte Syntax für reguläre Ausdrücke weicht von der Schreibweise, wie sie in der theoretischen Informatik Anwendung findet ab.
 - $(p)^*$ (lies: p beliebig oft) wird in Java genauso notiert
 - Ein $*$ als Postfix-Operator bedeutet also: der Operand beliebig häufig, auch gar nicht
 - $(p+q)$ (lies: p oder q) wird notiert als $p|q$
 - (pq) (lies: p gefolgt von q) wird notiert als pq
- Auch in Java können runde Klammern zum einfachen Gruppieren eingesetzt werden

Kontextfreie und reguläre Sprachen

Menge von
Zeichenketten



definiert,
beschreibt



kontextfreie
Grammatik G

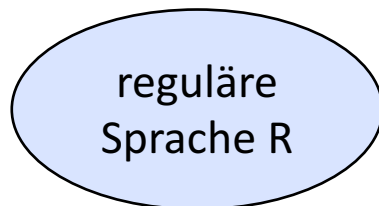
Java Quellcode

"class Konto { ..."

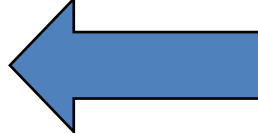
$\in L?$

Wo ist der
Unterschied?

Menge von
Zeichenketten



definiert,
beschreibt



reguläre
Grammatik G2

regulärer
Ausdruck

"_saldo"

$\in R?$

Der Unterschied liegt in der Mächtigkeit

- Kontextfreie Grammatiken sind beschreibungsmächtiger
- Mit ihnen lassen sich z.B. korrekt geklammerte Ausdrücke (Ausdrücke, die genau so viele schließende wie öffnende Klammern enthalten) beschreiben.
- Dies geht nicht mit regulären Ausdrücken!
- Ein falscher Versuch:



Expression:

`{ (} IntegerLiteral { InfixOp IntegerLiteral } { } }`

- Dieser falsche Versuch kann auch als regulärer Ausdruck formuliert werden.
- Die korrekte Lösung hingegen erfordert eine bestimmte Form der Rekursion, die für reguläre Ausdrücke nicht zugelassen ist:

- Expression:
`(Expression)`

...

Warum dann überhaupt
reguläre Ausdrücke?

Reguläre Ausdrücke sind effizient umsetzbar

- Die Syntax einer Programmiersprache ließe sich auch vollständig mit einer kontextfreien Grammatik beschreiben
- Reguläre Ausdrücke werden aus **Effizienzgründen** für die lexikalische Analyse verwendet
- Compiler erstellt automatisiert **Erkenner** für reguläre Ausdrücke
- Reguläre Ausdrücke werden deshalb beispielsweise von Suchmaschinen, Texteditoren und Programmiersprachen wie Java unterstützt

Zusammenfassung

1

In SE werden **Zeichenketten** oft verarbeitet.
Java definiert mit **String** einen Typ für **unveränderliche** Zeichenketten.

2

String bietet viele Operationen. Strings sollen ausschließlich mit **equals()** **verglichen** werden.

3

Programme, als Folgen von Zeichen aufgefasst, lassen sich in **elementare Bestandteile** zerlegen, die **Token** genannt werden.

4

Reguläre Ausdrücke sind ein mächtiges Beschreibungsmittel für Token, aber auch für andere Zwecke einsetzbar.