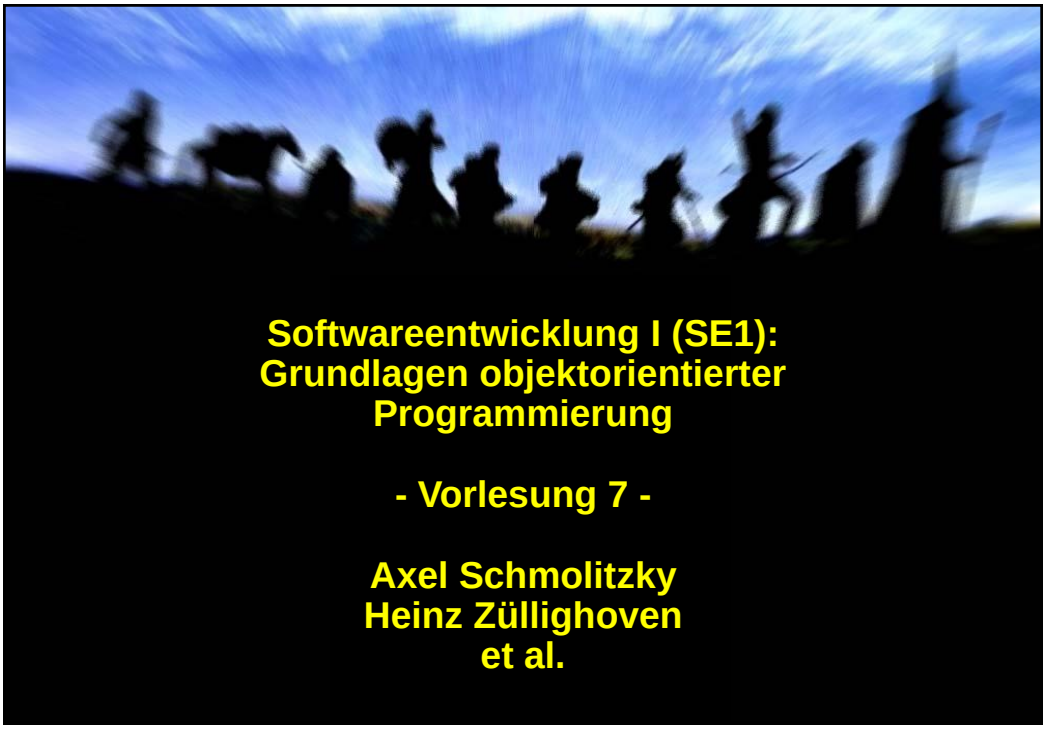


Softwareentwicklung I (SE1): Grundlagen objektorientierter Programmierung

- Vorlesung 7 -

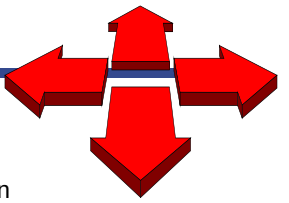
Axel Schmolitzky
Heinz Züllighoven
et al.

Ein Gedicht

- Vico von Bülow: Advent

wird fortgesetzt…

Rekursion



- Prozeduren/Methoden können sich in modernen Sprachen auch **selbst aufrufen** und damit **rekursiv** definiert sein.
- Rekursion ist neben den klassischen Schleifenkonstrukten eine zweite Möglichkeit, **Wiederholungen** zu programmieren.

Rekursion: ein erstes Beispiel

Ein häufiges Beispiel für die Verwendung einer rekursiven Programmierung ist die Berechnung der **Fakultät** einer Zahl. Die Fakultät ***n!*** ist das Produkt aller natürlichen Zahlen von ***1*** bis ***n***. ***4!*** beispielsweise ist ***1 * 2 * 3 * 4***, also ***24***.

Die mathematische Definition der Fakultät lautet:

rekursive Definition

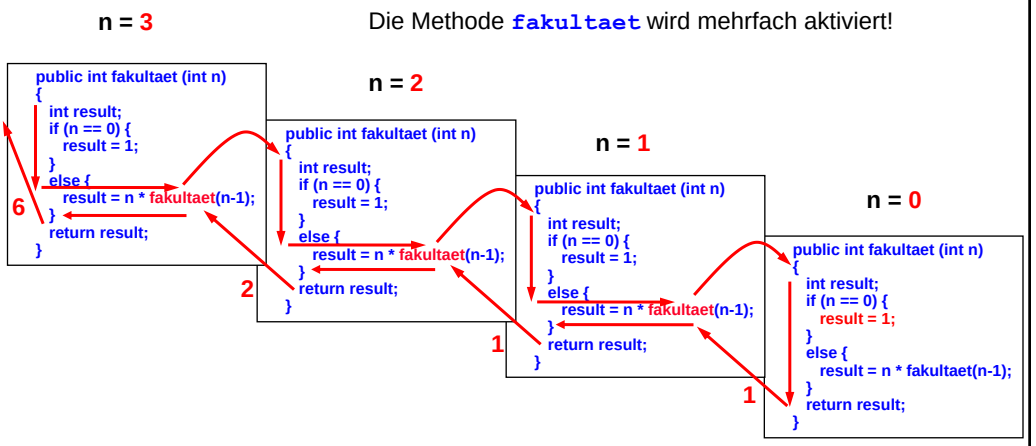
- Die Fakultät der Zahl ***0*** ist ***1***
- Die Fakultät einer natürlichen Zahl ***n***, mit ***n > 0***, ist ***n * (n – 1)!***

In Java lässt sich das so notieren:

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

rekursiver Aufruf

Rekursion: Der Kontrollfluss

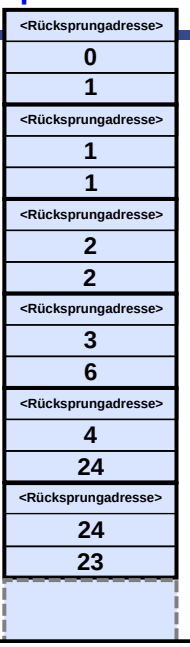


Der Aufrufstack

- Ein **Aufrufstack** (engl.: call stack oder function stack) ist eine Speicherstruktur, in der **zur Laufzeit** Informationen über die gerade aktiven Methoden gespeichert werden (in sogenannten **Stackframes**).
- Bei jedem neuen Methodenaufruf werden die **Rücksprung-adresse** und die **lokalen Variablen** (schließen die formalen Parameter mit ein) in einem neuen Stackframe auf dem Stack gespeichert. Wenn eine Methode terminiert, wird der zugehörige Stackframe wieder **vom Stack geräumt**.
- In höheren Programmiersprachen wie Java ist der Aufrufstack für die Programmierung zwar nicht zugänglich, Kenntnisse über seine Verwaltung erleichtern jedoch das Verständnis der Programmierung.

Rekursion: Der Aufrufstack für das Beispiel

- Da die lokalen Variablen auf dem Aufrufstack gespeichert werden, können **rekursive Methodenaufrufe** einfach realisiert werden:
  - Für jeden rekursiven Aufruf wird ein neuer Satz lokaler Variablen in einem Stackframe gespeichert.
  - So kann eine Methode auf jeder Rekursionsstufe auf ihren eigenen lokalen Variablen arbeiten und ihr Funktionsergebnis zurückgeben.
- Für den Beispielausdruck `23 + fakultaet(4)` würde jeder Aufruf folgende Informationen auf dem Stack ablegen:
  - Platz für Ergebnis
  - Argument n
  - Rücksprungadresse in die rufende Methode



ein Stackframe für **fakultaet**

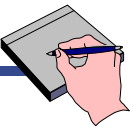
Stackframe der Klientenmethode, die den Ausdruck enthält

Rekursion: Das Beispiel als iteratives Programm

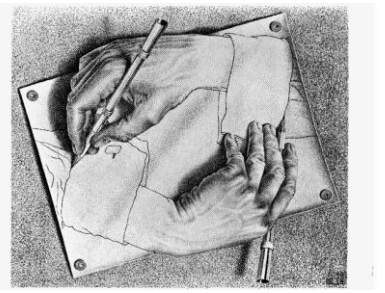
- Rekursive Programme haben in den meisten imperativen Programmiersprachen kein gutes Speicher- und Ablaufverhalten. Durch die wiederholten Methodenaufrufe wird immer wieder derselbe Programmcode bearbeitet und jedesmal ein neues Segment auf dem Aufrufstack belegt; ein vergleichsweise hoher Aufwand.
- Alternativ lässt sich die Fakultät in Java auch **iterativ** programmieren:

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

Rekursion allgemein



- Rekursion tritt auf, wenn eine Methode `m` während der Ausführung ihres Rumpfes erneut aufgerufen wird. Damit dieser Prozess nicht endlos läuft („nicht terminiert“), ist eine **Abbruchbedingung** zwingend notwendig.
- Wir unterscheiden:
  - Eine Rekursion ist **direkt**, wenn eine Methode `m` sich im Rumpf selbst ruft.
  - Eine Rekursion ist **indirekt**, wenn eine Methode `m1` eine andere Methode `m2` ruft, die aus ihrem Rumpf `m1` aufruft.
- Der Grundgedanke der Rekursion ist, dass die Methode einen ersten Teil eines Problems selbst löst, den Rest in kleinere Probleme zerlegt und sich selbst mit diesen kleineren Problemen aufruft.



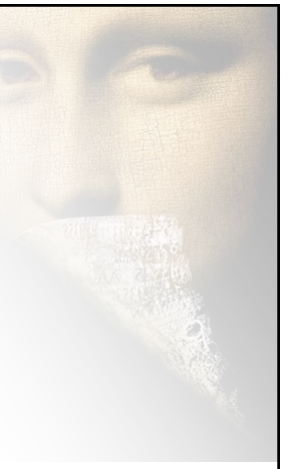
Rekursion: Grundstruktur

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

Abbruchbedingung

rekursive Aufrufe

Rekursion: DAS Gegenbeispiel



- Die **Fibonacci-Zahlen** sind sehr einfach definiert:
  - Die erste Fibonacci-Zahl ist 0.
  - Die zweite Fibonacci-Zahl ist 1.
  - Die n-te Fibonacci-Zahl ergibt sich aus der Summe der (n-1)ten und der (n-2)ten Fibonacci-Zahl.
- Die dritte Fibonacci-Zahl ist demnach 1, die vierte 2, die fünfte 3, die sechste 5 usw.
- Die rekursive Definition lässt sich unmittelbar rekursiv realisieren:

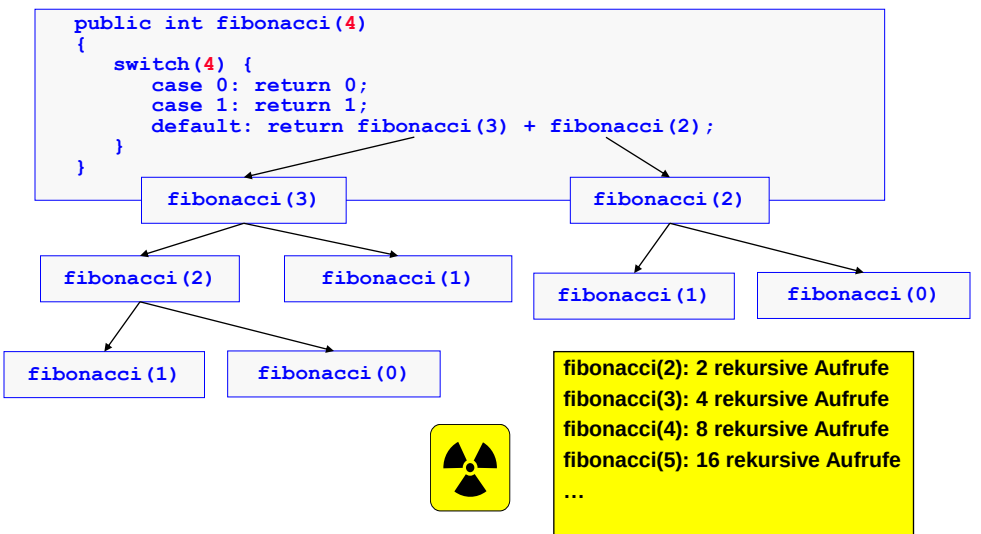
```
public int fibonacci (int n)
{
   switch (n) {
      case 1: return 0;
      case 2: return 1;
      default: return fibonacci(n-1) + fibonacci(n-2);
   }
}
```

Wo ist das Problem?

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 98

Rekursion: Wir beginnen etwas zu ahnen...

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



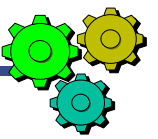
fibonacci(2): 2 rekursive Aufrufe
fibonacci(3): 4 rekursive Aufrufe
fibonacci(4): 8 rekursive Aufrufe
fibonacci(5): 16 rekursive Aufrufe
…

Rekursion: Elegante Anwendungen

- Rekursion ist besonders in folgenden Fällen geeignet:
  - Wenn die Struktur, die verarbeitet wird, selbst rekursiv definiert ist; darunter fallen zum Beispiel alle **Baumstrukturen** in der Informatik (Syntaxbäume, Entscheidungsbäume, Verzeichnisbäume, etc.).
  - Viele sehr gute **Sortierverfahren** sind rekursiv definiert, beispielsweise Quicksort und Mergesort.
  - Viele Probleme auf **Graphen** lassen sich elegant rekursiv lösen.
- Im Laufe Ihres Studiums werden Sie noch viele Anwendungsfälle von Rekursion kennen lernen!

Wir werden in SE1 noch einige gute Anwendungen von Rekursion betrachten.

Rekursion: Stärken und Schwächen



Steve McConnells Einschätzung zu Rekursion:

- Rekursion kann für eine relativ kleine Menge von Problemen **sehr einfache, elegante Lösungen** produzieren.
- Rekursion kann für eine etwas größere Menge von Problemen sehr einfache, elegante und **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. **Rekursion sollte sehr selektiv eingesetzt werden**.

Ergo: Es gibt Situationen, in denen Rekursion sich als gute Lösung anbietet. Es gibt mehr Situationen, in denen Rekursion sich als Lösung **verbietet**.

© Steve McConnell: *Code Complete 2*, Microsoft Press, 2004

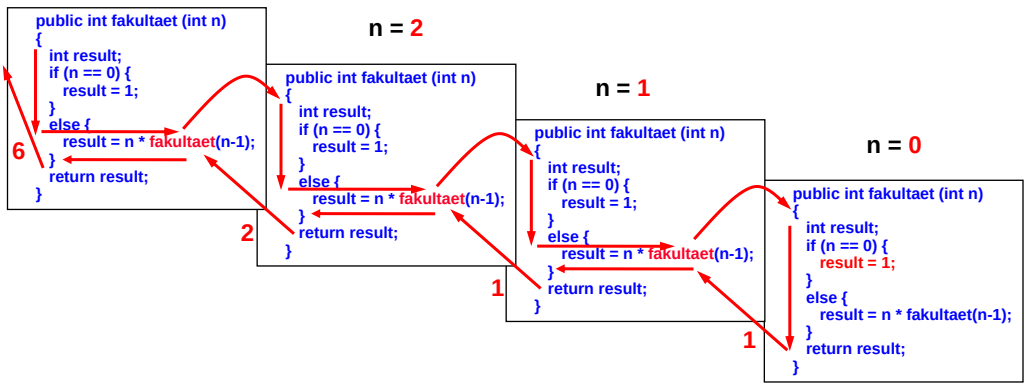
nachgereicht: Lebensdauer

- Die **Lebensdauer** (engl.: lifetime) einer Variablen oder eines Objektes ist eine *dynamische Eigenschaft*. Lebensdauer bezeichnet die Zeit, in der eine Variable (oder ein ggf. damit verbundenes Objekt) während der Laufzeit **existiert**. Während der Lebensdauer ist einer Variablen (oder einem Objekt) **Speicherplatz** zugewiesen.
- Sichtbarkeit und Lebensdauer können unabhängig voneinander sein, wie in folgender Situation:
  - Eine Exemplarvariable `x` ist statisch deklariert, ein entsprechendes Feld eines Objektes hält zur Laufzeit einen Wert.
  - In einer Methode ist eine gleichnamige lokale Variable `x` deklariert, die die Exemplarvariable verdeckt. Obwohl sie weiter im Speicher existiert, ist die Exemplarvariable während der Ausführung der Methode **nicht über den Namen `x` sichtbar**.
- Bei **Objekten** in Java ist die **Lebensdauer** davon abhängig, ob noch Referenzen auf sie existieren.

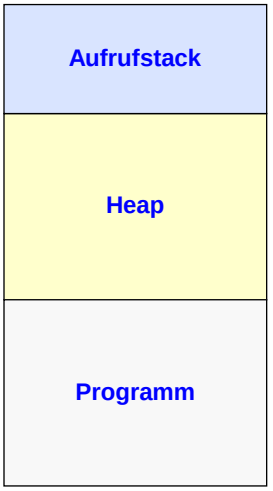
Rekursion: Lebensdauer lokaler Variablen

n = 3

Vier lokale Variablen `result` leben unterschiedlich lange!



Vereinfachtes Speichermodell von Sprachen mit dynamischen Objekten



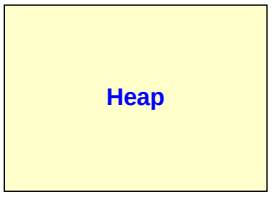
Der Speicherplatz für **lokale Variablen** (und Zwischenergebnisse von Ausdrücken) wird stapelartig durch das Laufzeitsystem verwaltet.

Der Speicherplatz für **dynamisch erzeugte Objekte** (mit ihren Exemplarvariablen) wird explizit vom Programmierer (z.B. `new` in Java) angefordert. Die Speicherfreigabe erfolgt explizit (z.B. in C++) oder durch den **Garbage Collector** (z.B. in Java).

Speicherplatzanforderungen für den **Programmcode** (die übersetzten Klassendefinitionen) werden durch das Betriebssystem befriedigt.

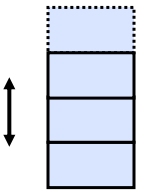
Der Heap

- Der **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`).

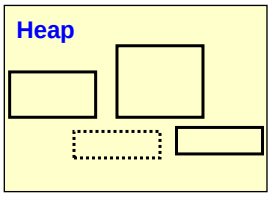


Heap und Aufrufstack

- Der Unterschied zwischen Aufrufstack und Heap besteht darin, dass beim Aufrufstack angeforderte Speicherabschnitte **strikt in der umgekehrten Reihenfolge** wieder freigegeben werden, in der sie angefordert wurden.
- Beim Aufrufstack spricht man deshalb auch von **automatischer Speicheranforderung**. Die Laufzeitkosten einer automatischen Speicheranforderung sind in der Regel deutlich geringer als die bei der dynamischen Speicheranforderung.
- Allerdings kann bei spezieller Nutzung durch sehr große oder sehr viele Anforderungen der für den Stack reservierte Speicher ausgehen - dann droht ein Programmabbruch wegen **Stapelüberlauf** (in Java: `StackOverflowError`).

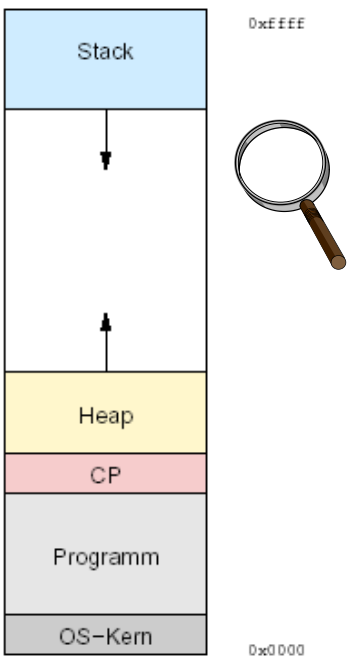


Aufrufstack



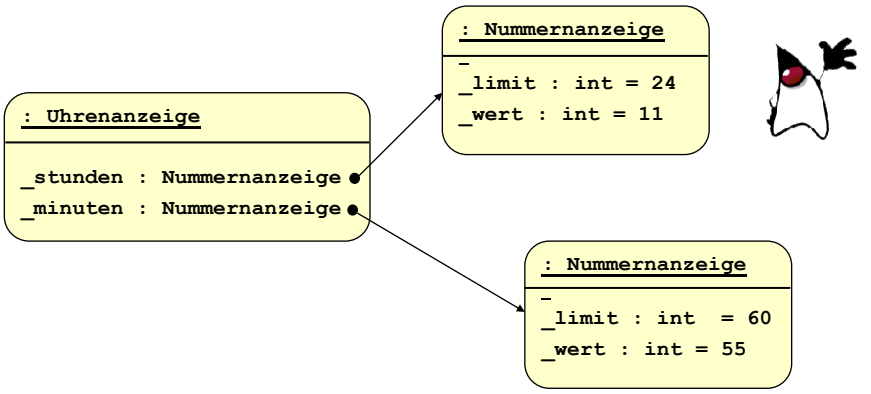
Beispiel: Speichereinteilung in einem Unix-System

- **Programm**:
  enthält den eigentlichen Programmtext mit allen Befehlen. Sofern keine selbstmodifizierende Programme zum Einsatz kommen, bleibt das Textsegment während des Programmablaufs unverändert.
- Constant Pool (CP):
  nimmt alle Konstanten und statischen Variablen des Programms auf.
- **Heap**:
  nimmt alle dynamisch zur Laufzeit des Programms erzeugten Variablen bzw. Objekte auf.
- **Stack**:
  wird für die Parameterübergabe zwischen Funktionen und für die Speicherung der lokalen Variablen der einzelnen Funktionen benutzt.



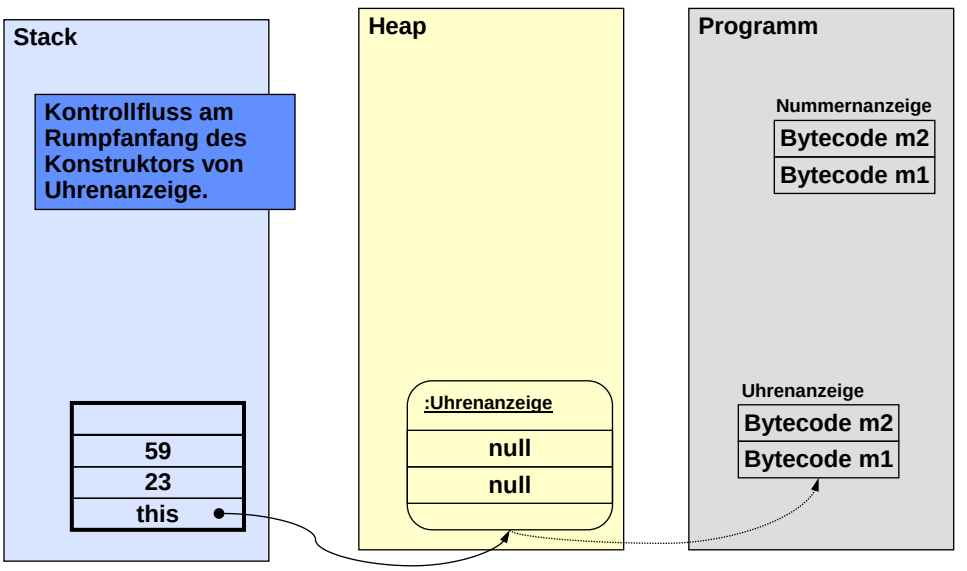
Java-Objektdiagramme: Schnappschüsse vom Heap

- Ein Objektdiagramm ist in Java immer 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.

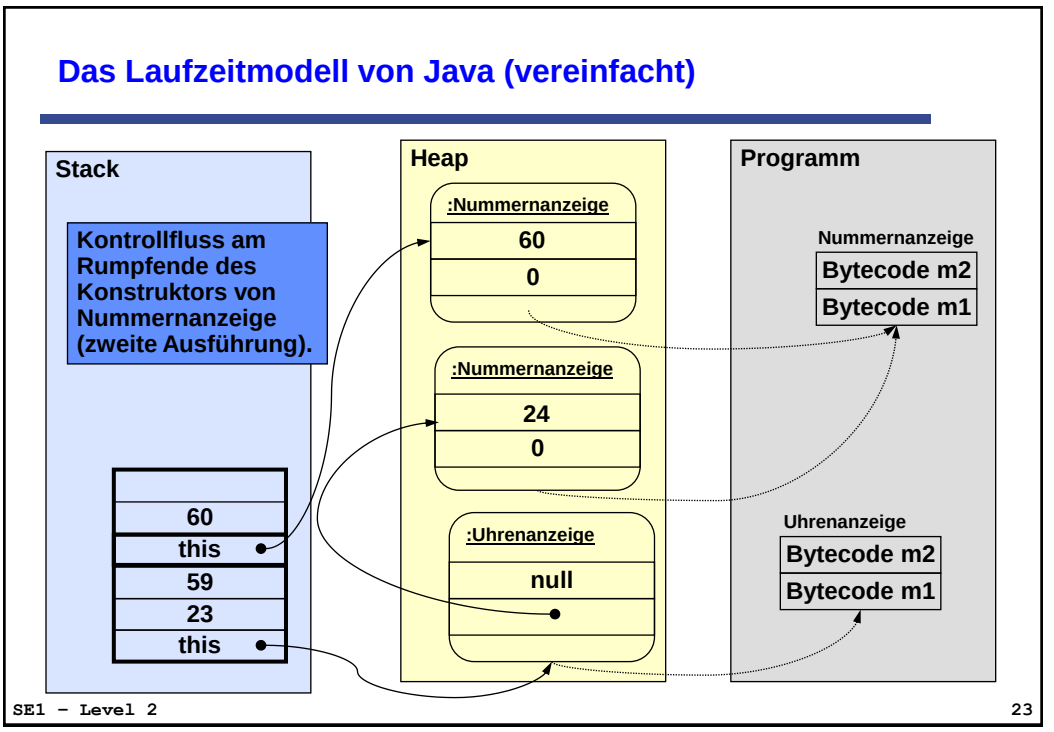


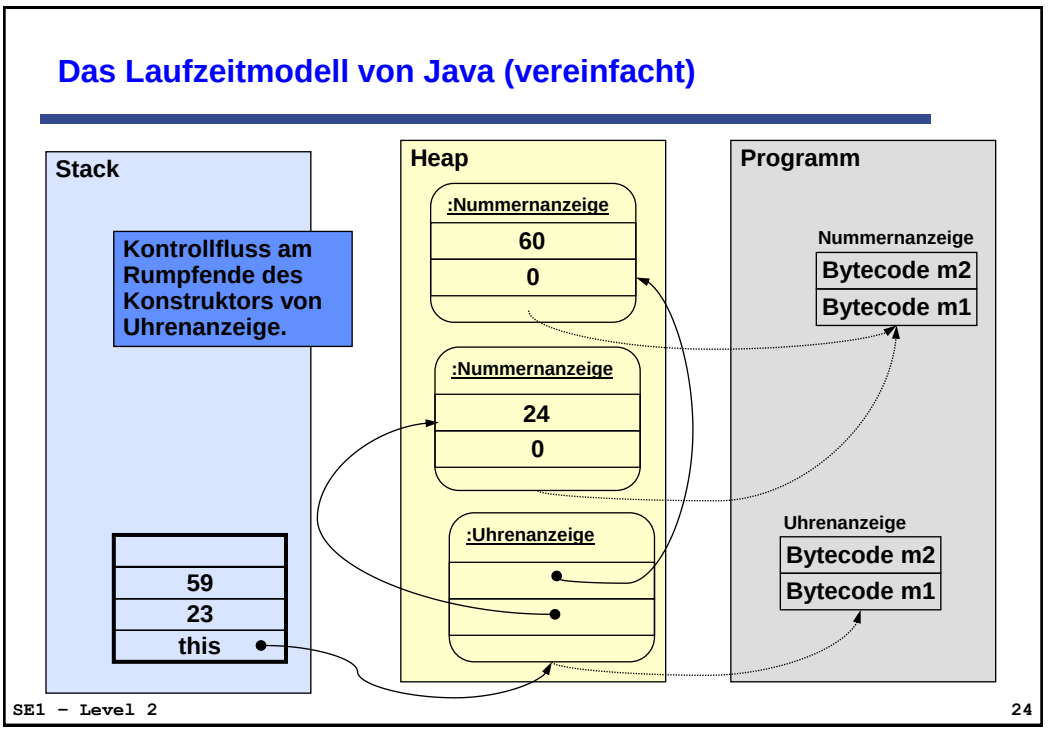
SE1 – Level 2 21

Das Laufzeitmodell von Java (vereinfacht)



SE1 – Level 2 22





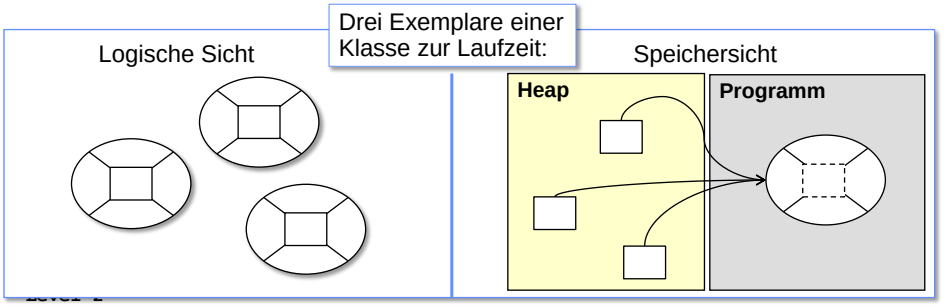
Der Garbage Collector in Java

- Mit unserem Wissen über Heap und Stack können wir nun erstmalig nachvollziehen, was der **Garbage Collector** von Java macht.
- Die Voraussetzungen sind:
  - **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 diejenigen Objekte, die **vom Aufrufstack** aus **erreichbar** sind, spielen für die Programmausführung eine Rolle. Alle anderen Objekte im Heap sind „tote“ Objekte.
- Daraus folgt das **Vorgehen** des Garbage Collectors:
  - Er verfolgt in regelmäßigen Abständen, ausgehend von den Referenzen auf dem Stack , transitiv das **gesamte Objektgeflecht** und **markiert** die erreichbaren Objekte. Anschließend werden alle nicht markierten Objekte im Heap **gelöscht**. Dieses Vorgehen aus **Markieren** und **Abräumen** heißt im Englischen **Mark and Sweep**.

SE1 – Level 2 25

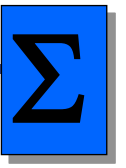
Methoden und Zustandsfelder

- Den Zusammenhang zwischen statischen und dynamischen Eigenschaften können wir anhand der Methoden und Felder noch einmal verdeutlichen:
  - zur **Übersetzungszeit** gibt es jede **Methode** nur **einmal**, ebenso wie die **Exemplarvariablen**. 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 lediglich eine Optimierung.



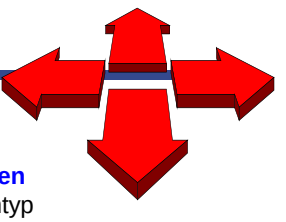
SE1 – Level 2 26

Zusammenfassung



- **Rekursive Methodenaufrufe** sind eine alternative Möglichkeit für Wiederholungen.
- Jede Wiederholung lässt sich **sowohl iterativ als auch rekursiv** formulieren, jeweils mit spezifischen Vor- und Nachteilen.
- Softwaretechnische Überlegungen wie **Verständlichkeit und Sicherheit** spielen bei der Wahl einer geeigneten Realisierung eine wichtige Rolle.
- „Hinter den Kulissen“ moderner Programmiersprachen sind der **Aufrufstack** und der **Heap** zentrale Strukturen für die Verwaltung von Variablen und Objekten.

Strings und Reguläre Ausdrücke



- Sehr häufig werden bei der Programmierung **Zeichenketten** verarbeitet. Java definiert mit dem Typ **String** einen Datentyp für **unveränderliche** Zeichenketten.
- Programme, als Folgen von Zeichen aufgefasst, lassen sich in elementare Bestandteile zerlegen, die **Token** genannt werden.
- **Reguläre Ausdrücke** sind ein mächtiges Beschreibungsmittel für Token, aber auch für andere Zwecke einsetzbar.

Zeichenketten in Programmiersprachen

- Moderne Programmiersprachen bieten Unterstützung für **Zeichenketten** (engl.: strings). Eine Zeichenkette ist eine Folge von einzelnen Zeichen.

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

- Die Anzahl der Zeichen in einer Zeichenkette wird auch als ihre **Länge** bezeichnet. Konzeptuell sind Zeichenketten in ihrer Länge **unbegrenzt**. In einigen Kontexten (z.B. Datenbanken) müssen Zeichenketten jedoch eine fest definierte Maximallänge haben.
- Eine Unterstützung für Zeichenketten ist in allen Anwendungen notwendig, in denen Texte (Prosa, Quelltexte, etc.) verarbeitet werden.
- In objektorientierten Sprachen werden Zeichenketten üblicherweise als Objekte modelliert.

Datentyp: **`Zeichenkette`**
Wertemenge: { Zeichenketten beliebiger Länge }
Operationen: **`Länge`**, **`Subzeichenkette`**, **`Zeichen an Position x`**, …

Zeichenketten in Java: Literale, Konkatenation

- In Java werden Zeichenketten primär durch die Klasse **`String`** unterstützt. Diese Klasse definiert, wie alle Klassen in Java, einen Typ.
- **`String`** ist in Java ein expliziter Bestandteil der Sprache, denn es gibt einige Spezialbehandlungen für diesen Typ:
  - **`String`-Literale** (Zeichenfolgen zwischen doppelten Anführungszeichen) werden vom Compiler speziell erkannt:

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

  - Der Infix-Operator **`+`** kann auch auf Strings angewendet werden; er konkateniert (verkettet) zwei Strings zu einem neuen String.
- Von der Klasse **`String`** gibt es eine javadoc-Darstellung, die alle Methoden beschreibt, die Klienten zur Verfügung stehen:
  - http://download.oracle.com/javase/1.5.0/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!
```

- Erster Versuch:

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

- Das Problem: Der Compiler sieht zwei String-Literale, getrennt von dem (ihm unbekannten) Bezeichner `Aaah`, da das zweite Anführungszeichen das erste String-Literal beendet.
- Wenn wir Anführungszeichen in einem String-Literal platzieren wollen, müssen wir eine so genannte **Escape-Sequenz** anwenden:

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

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

SE1 – Level 2 31

Strings in Java: Unveränderlich!

- Die Klasse `String` in Java definiert Objekte, die **unveränderliche Zeichenketten** sind:
  - Alle Operationen auf Strings liefern Informationen über ein `String`-Objekt (einzelne Zeichen, neue Zeichenketten), **verändern es aber niemals**.
  - Der Infix-Operator **+** verkettet zwei Strings zu **einem neuen** String.
- Strings sind damit sehr **untypische Objekte** in Java, denn sie haben keinen (veränderbaren) Zustand.

Typischer Fehler:

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

SE1 – Level 2 32

Gleichheit von Strings in Java

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

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

Das Problem: **!=**

Datentyp: **`Zeichenkette`**
Wertemenge: { Zeichenketten beliebiger Länge }
Operationen: **`Länge`**, **`Subzeichenkette`**, …

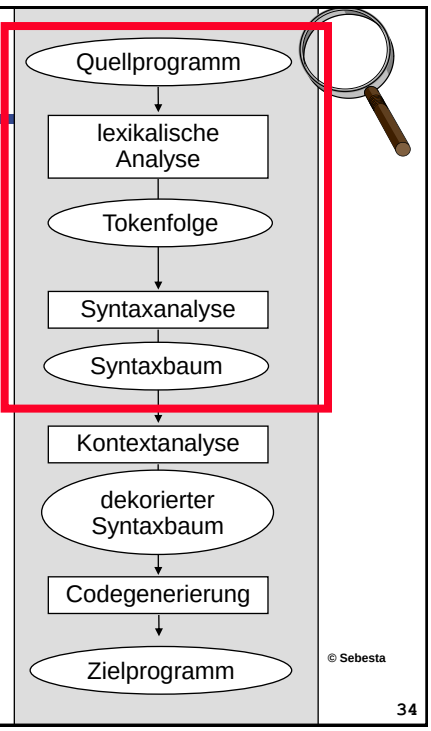
Datentyp: **`String`**
Wertemenge: { **`String`**-Exemplare beliebiger Länge }
Operationen: **`length`**, **`concat`**, **`substring`**, **`charAt`**, …

- Weil Strings in 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!**

SE1 – Level 2 33

Wie arbeitet eigentlich ein Compiler?

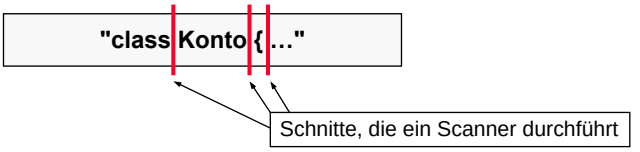
- Die Übersetzung von Programmen erfordert meist mehrere Schritte:
- Der für Menschen lesbare Quelltext wird durch einen **Scanner** in eine Folge von **Token** zerlegt (**lexikalische Analyse**).
- Ein **Parser** erzeugt aus der Tokenfolge einen Syntaxbaum (**Syntaxanalyse**).
- Dieser wird analysiert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



SE1 – Level 2 34

Syntaktische Grundelemente

- Die kleinsten (aus Sicht der Grammatik unteilbaren) syntaktischen Einheiten einer Sprache werden auch **Token** genannt. Dazu gehören bei Programmiersprachen:
  - Bezeichner, Literale, Operatoren, reservierte Wörter und Sonderzeichen wie Klammern und Semikolon.
  - In Java:
    - **gültige Token** z.B.: `nextItem 3.1416 <= while`
    - **keine Token** sind z.B.: `next-item 3,1416 ><`
- Die **lexikalische Analyse** eines Compilers (also das Zerlegen eines Quelltextes in eine Folge von Token durch einen Scanner) lässt sich mit Hilfe von **regulären Ausdrücken** steuern.



Ein erstes Beispiel: Das Token „Bezeichner“

- Ein sehr häufiges Token ist der **Bezeichner**. Bezeichner werden verwendet, um Variablen, Methoden, Klassen etc. zu benennen.
- Definition eines Bezeichners in **Java** (leicht vereinfacht):
  - Ein Bezeichner besteht aus einem Buchstaben (ein Unterstrich wird auch als ein Buchstabe angesehen), gefolgt von beliebig vielen Buchstaben und Ziffern.

- Wenn wir von einer Zeichenkette `s` (vom Typ `String`) feststellen wollen, ob sie ein gültiger Bezeichner ist, dann fragen wir etwas abstrakter, ob `s` ein Element der **Menge aller gültigen Bezeichner** ist. In Java können wir diese Frage beispielsweise so ausdrücken:

```
s.matches(mengeGueltigerBezeichner)
```

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

Bezeichner als regulärer Ausdruck

- Die Menge aller Zeichenketten, die durch einen regulären Ausdruck beschrieben wird, wird als **reguläre Menge** bezeichnet. Aber wie sieht ein solcher Ausdruck aus?
- Als erstes betrachten wir eine Möglichkeit, unsere vereinfachte Definition eines Java-Bezeichners als regulären Ausdruck zu beschreiben:

```
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:

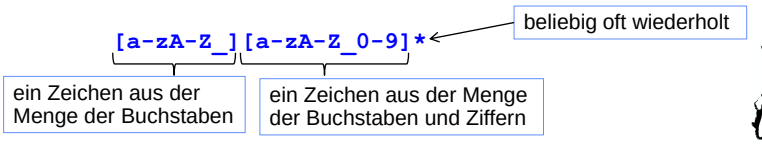
  `h[oa]se` beispielsweise definiert die reguläre Menge { **hose**, **hase** }.
- Zur weiteren Verkürzung erlaubt Java in den eckigen Klammern auch Bereichsangaben mit einem Minuszeichen. Beispielsweise:

  `se[1-3]` definiert die reguläre Menge { **se1**, **se2**, **se3** }.

  `[a-z]` definiert alle Kleinbuchstaben von a bis z.

Reguläre Ausdrücke in Java

- Somit haben wir unseren regulären Ausdruck schon fast verstanden:



- 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.
- Alternativ zum Postfix-Operator `*` (beliebige Wiederholung) bietet Java 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.

Weitere Beispiele für reguläre Ausdrücke in Java

- Weitere Beispiele:

```
String s = "ab";
s.matches("ab");
```
⇒ **true**, jeder String definiert sich selbst als Ausdruck

```
s.matches("a");
```
⇒ **false**, nur ein teilweiser „Match“

```
s.matches("aba");
```
⇒ **false**, auch knapp daneben

```
String re = "[ab][ab]";
s.matches(re);
```
⇒ **true** (gälte auch für `s = "aa"` oder `"ba"` oder `"bb"`)

```
re = "(ab)*";  // "ab" beliebig oft wiederholt; runde Klammern gruppieren
s.matches(re);
```
⇒ **true** (gälte auch für `s = ""`, `"abab"` oder `"abababab"` …)

```
re = "..";  // ein einzelner Punkt steht für ein beliebiges Zeichen
s.matches(re);
```
⇒ **true** (gälte auch für `s = "xy"` oder `"69"` …)

- Die ausführliche Beschreibung der Syntax regulärer Ausdrücke in Java findet sich unter http://docs.oracle.com/javase/1.5.0/docs/api/java/util/regex/Pattern.html#sum

Zeichenketten und reguläre Ausdrücke in Java

- Reguläre Ausdrücke sind nicht nur für Compiler nützlich, sondern können viele Formen der Analyse von Zeichenketten/Texten unterstützen.
- Java bietet die besagte Unterstützung für reguläre Ausdrücke, die mit Exemplaren der Klasse `String` arbeitet, seit der Version 1.4 an.
- Neben der Methode `matches` sind in der Klasse `String` weitere Methoden definiert, die mit regulären Ausdrücken arbeiten, u.a.:
  - `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;

http://download.oracle.com/javase/1.5.0/docs/api/java/lang/String.html

Formale Sprachen

- Jede **formale Sprache** lässt sich verstehen als:
  - eine Menge von **Zeichenketten** eines **Alphabets**

  Gleichbedeutend:
  - eine Menge von **Folgen von Symbolen** eines **Vokabulars** oder **Zeichensatzes**
- **Grammatikregeln** geben an, welche Zeichenketten des Alphabets **Wörter** der Sprache sind, d.h. **syntaktisch korrekt** oder **wohlgeformt** sind.
- Eine Grammatik ist somit eine **Metasprache**, mit der eine andere Sprache beschrieben wird.

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

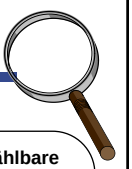
informell: Die Menge aller Zeichenketten, die mit mindestens einem **a** beginnen.

Wie können wir dies formaler fassen?

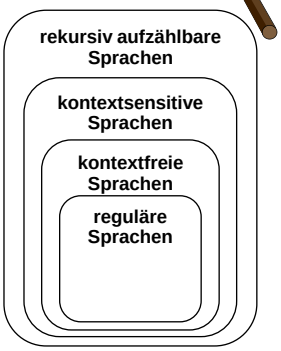
Beispiel für eine formale Sprache über diesem Alphabet:

{**a**, **aha**, **alter**, **aal**, **aabenra**, **aaarghh**, … }

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.
- Die uns bereits bekannte (**E**)**BNF** ist genau so beschreibungsmächtig wie kontextfreie Grammatiken.



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

Reguläre Ausdrücke und reguläre Sprachen

- Mit **regulären Ausdrücken** (also einer speziellen 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 ∈ 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∪Q** (Vereinigung) bezeichnet,
    - **(pq)** ein regulärer Ausdruck, der die reguläre Menge **P•Q** (Konkatenation) bezeichnet.
  - **Ø** ist ein regulärer Ausdruck, der die leere reguläre Menge bezeichnet.
  - ε ist ein regulärer Ausdruck, der die reguläre Menge {ε} 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)).

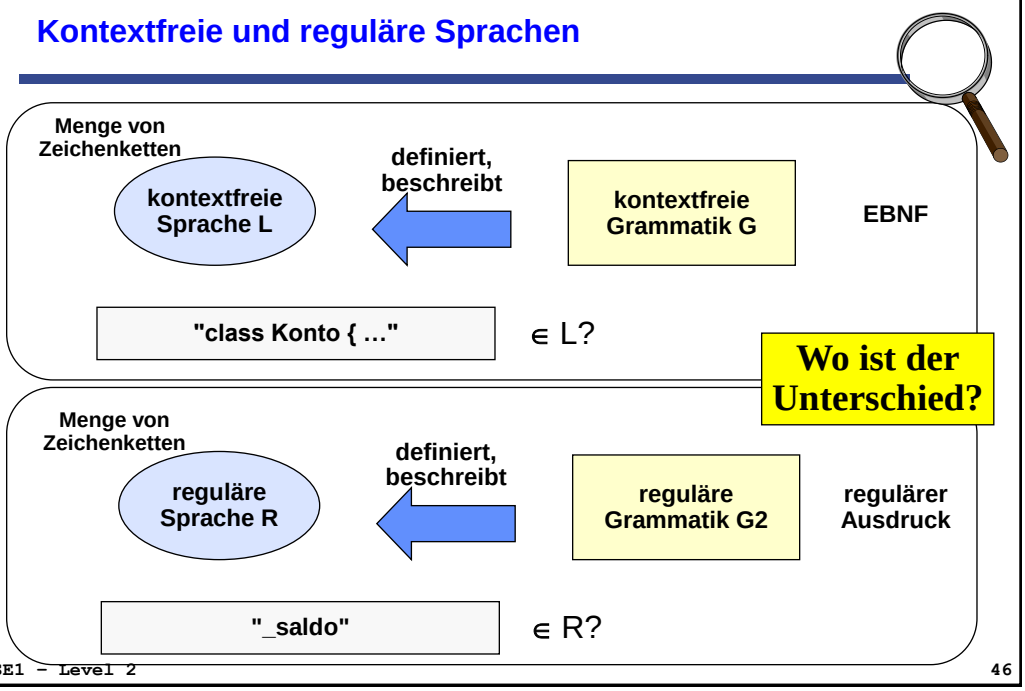
Mehr zu den theoretischen Grundlagen regulärer Ausdrücke in FGI

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, teilweise ab.
- Hier die grundlegenden theoretischen Konzepte übersetzt auf die **konkrete Syntax regulärer Ausdrücke** in Java:
  - **(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.

SE1 – Level 2 45

Kontextfreie und reguläre Sprachen



SE1 – Level 2 46

Der Unterschied liegt in der Mächtigkeit

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

  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:

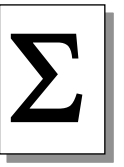
  Expression:
  (*Expression* **)**
  …

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 lediglich aus **Effizienzgründen** für die lexikalische Analyse verwendet.
- Sehr schnelle Erkenner für reguläre Ausdrücke lassen sich automatisiert erstellen; reguläre Ausdrücke werden deshalb beispielsweise von **Suchmaschinen**, **Texteditoren** und auch **Programmiersprachen** wie Java unterstützt.

Zusammenfassung



- Der in die Sprache integrierte Typ **`String`** in Java bietet eine mächtige Unterstützung für die Verarbeitung von Zeichenketten.
- Strings in Java sind Exemplare der Klasse **`String`** und somit Objekte, die über Referenzen zugegriffen werden; sie sind jedoch spezielle Objekte, denn sie sind **unveränderlich**.
- Strings sollten **ausschließlich mit equals verglichen** werden.
- Für die **Darstellung der Syntax** von Programmiersprachen (z.B. in Hand- und Lehrbüchern) haben wir bereits die **EBNF** und **Syntaxdiagramme** kennen gelernt. Sie sind gleichwertig mit kontextfreien Grammatiken.
- Die terminalen Symbole einer EBNF werden auch **Token** genannt. Sie werden häufig mit **regulären Ausdrücken** beschrieben.
- Java bietet seit Version 1.4 eine Unterstützung für reguläre Ausdrücke an, die auf dem Typ **`String`** basiert.

