Softwareentwicklung I (SE1): Grundlagen objektorientierter Programmierung

- Vorlesung 8 -

Axel Schmolitzky
Heinz Züllighoven
Guido Gryczan
et al.

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.

Vorspiel: Berechnung eines Funktionswertes

- **Berechnung eines Funktionswertes** folgt prinzipiell dem **Schema**:
  - Die formalen Argumente der Definition werden durch die aktuellen der Verwendung ersetzt,
  - die Bindungen der linken Seite der Definition werden auf die rechte Seite übertragen,
  - die Funktionen der rechten Seite werden ausgewertet.

Beispiel:

even (n) = (n div 2 * 2 = n) || Funktionsdefinition

even (8) || Verwendung von even

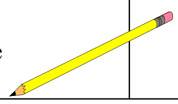
even (8) ⇒ (8 div 2 * 2 = 8) || Bindung der aktuellen Argumente

even (8) ⇒ (4 * 2 = 8) || Auswertung von div

even (8) ⇒ (8 = 8) || Auswertung von "*"

even (8) ⇒ (ja) || Auswertung von "="

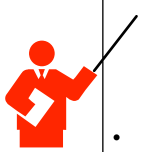
⇒ ja || Funktionswert von even (8)



" ||" bedeutet:

"hier beginnt ein Kommentar".

Rekursion



- Die **Grundidee**:
  - Löse ein Gesamtproblem durch die Aufspaltung in gleichartige einfachere Teilprobleme.
  - Definition eines Problems, einer Funktion oder eines Verfahrens durch sich selbst.

    [nach: Informatik-Duden]
- **Stellenwert**:
  - **Rekursion** ist neben **Sequenz** und **Fallunterscheidung** das wesentliche Strukturierungsmittel der Programmierung.
  - Konzeptionell führen Rekursionen oft zu **eleganten** Programmen (oder Entwürfen).
- **Anwendungsfälle**:
  - Die Lösung eines (Teil-)problems liegt in einer **rekursiven Formulierung** vor.
  - Später im Semester: Eine **rekursive Datenstruktur** (z.B. Liste oder Baum) muß **durchsucht** und nach bestimmten Kriterien **bearbeitet** werden.

Eine rekursive Formulierung der Addition

- Die **Addition** wird auf die **Nachfolgerbildung (succ)** zurückgeführt (und verwendet auch den Vorgänger **pred**):

  $a \text{ add } b = \text{succ} (a \text{ add } (\text{pred}(b)) \text{ für } \quad (a \in N, b \in N)$

  Merke: **N** bezeichnet die natürlichen Zahlen >= 0

- **Definition** von plus

  $\text{add}: N \times N \rightarrow N$

  $$\text{add}(a, b) = \begin{cases} a & \text{falls } b = 0 \\ 1 + (\text{add}(a, b - 1) & \text{sonst} \end{cases}$$

  Alternative Schreibweise: succ (add(a,pred(b)))

Eine rekursive Divisionsfunktion

- Die **Division** wird auf die **Subtraktion** zurückgeführt:
  $a \text{ div } b = 1 + (a - b) \text{ div } b \qquad \text{für} \quad (a \in N, b \in N^+, \ a \geq b)$
  Merke: N^+ bezeichnet die natürlichen Zahlen > 0
- **Definition** von divide
  $\text{divide} : N \times N^+ \rightarrow N$

$$\text{divide } (a, b) = \begin{cases} 0 & \text{falls } a < b \\ 1 + \text{divide } (a - b, b) & \text{sonst} \end{cases}$$

Beispiel: Auswertung der rekursiven Divisionsfunktion

Definition von divide: divide : $N \times N^+ \rightarrow N$

$$\text{divide}(a, b) = \begin{cases} 0 & \text{falls } a < b \\ 1 + \text{divide}(a - b, b) & \text{sonst} \end{cases}$$

Verwendung von divide:

$$\begin{aligned}
\text{divide}(7, 3) &\Rightarrow \begin{cases} 0 & \text{falls } 7 < 3 \\ 1 + \text{divide}(7 - 3, 3) & \text{sonst} \end{cases} \\
&\Rightarrow 1 + \text{divide}(4, 3) \\
&\Rightarrow 1 + \begin{cases} 0 & \text{falls } 4 < 3 \\ 1 + \text{divide}(4 - 3, 3) & \text{sonst} \end{cases} \\
&\Rightarrow 1 + (1 + \text{divide}(1, 3)) \\
&\Rightarrow 1 + (1 + \begin{cases} 0 & \text{falls } 1 < 3 \\ 1 + \text{divide}(1 - 3, 3) & \text{sonst} \end{cases}) \\
&\Rightarrow 1 + (1 + (0)) \\
&\Rightarrow 2
\end{aligned}$$

Rekursive Funktionsdefinitionen - Fakultät

- Eine *nicht-elementare* Funktionsdefinition:

  f: $N \rightarrow N$

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

- Die **Funktionsdefinition** nimmt auf der rechten Seite Bezug auf die **definierte Funktion** selbst.

Beispiel für schrittweise Auswertung

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

- **Berechne** den Wert fak (n) mit n = 0, 1, 2, …:
  - **Auswertungsschritte**:
    1. Prüfe, ob n = 0
    2. Wenn ja, dann ist das Ergebnis 1
    3. Sonst berechne als Zwischenergebnis fak (n - 1)
    4. Berechne das Ergebnis durch Multiplikation des Zwischenergebnisses mit n
- **Beachte**:

  Das Verfahren für fak (n) ist **rekursiv**, d.h. aus dem **Umfang seiner Beschreibung** können wir nicht unmittelbar auf den Aufwand einer bestimmten Auswertung schließen.
  - Beispiel: fak (0) ist sehr viel einfacher auszuwerten als fak (5)

© Löhr

Auswertung

f: N → N

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

- **Verwendung**:

$$f(2) \Rightarrow \begin{cases} 1 & \text{falls } 2 = 0 \\ 2 * f(2-1) & \text{falls } 2 > 0 \end{cases}$$
$$\Rightarrow 2 * f(1)$$
$$\Rightarrow 2 * \begin{cases} 1 & \text{falls } 1 = 0 \\ 1 * f(1-1) & \text{falls } 1 > 0 \end{cases}$$
$$\Rightarrow 2 * (1 * f(0))$$
$$\Rightarrow 2 * \begin{cases} 1 & \text{falls } 0 = 0 \\ 0 * f(0-1) & \text{falls } 0 > 0 \end{cases}$$
$$\Rightarrow 2 * (1 * (1))$$
$$\Rightarrow 2 * (1)$$
$$\Rightarrow 2$$

Zwischenergebnis

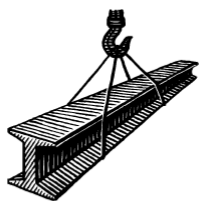
- Die **Berechnung eines Funktionswertes** folgt einem festen Schema:
  - Ersetzen der formalen durch die aktuellen Argumente,
  - Bindung gleicher Namen an die aktuellen Werte,
  - Auswertung der rechten Seite der Definition.
- **Rekursive Funktionsdefinitionen** beziehen sich auf der rechten Seite auf sich selbst.
- Das **Berechnungsschema** für Funktionen ist selbst rekursiv.
- **Komposition, Rekursion** und **Fallunterscheidung** sind wesentliche Strukturelemente der Programmierung.

Rekursion: Grundstruktur (in Java)

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

Abbruchbedingung

rekursive Aufrufe



Rekursion: Das Beispiel `fakultaet` in Java

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:

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

rekursive Definition

In Java lässt sich das so notieren:

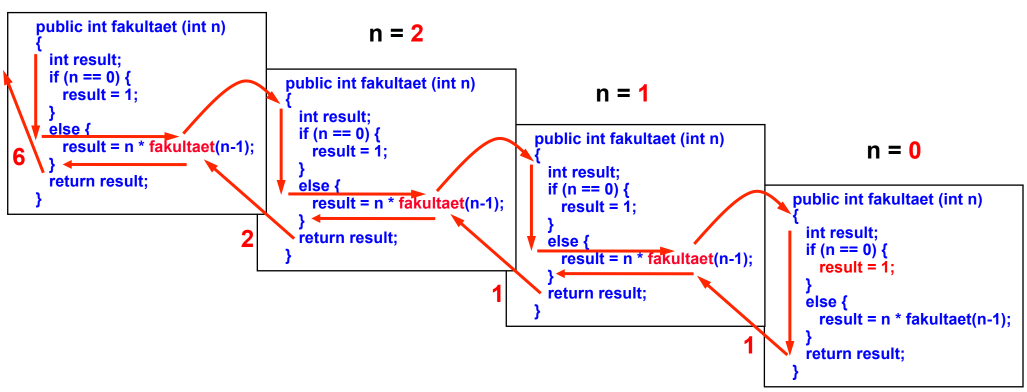
```
public int fakultaet(int n)
{
    int result;
    if (n == 0)
    {
        result = 1;
    }
    else
    {
        result = n * fakultaet(n-1);
    }
    return result;
}
```

rekursiver Aufruf

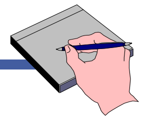
Fakultät: Der Kontrollfluss

n = 3

Die Methode `fakultaet` wird mehrfach aktiviert!



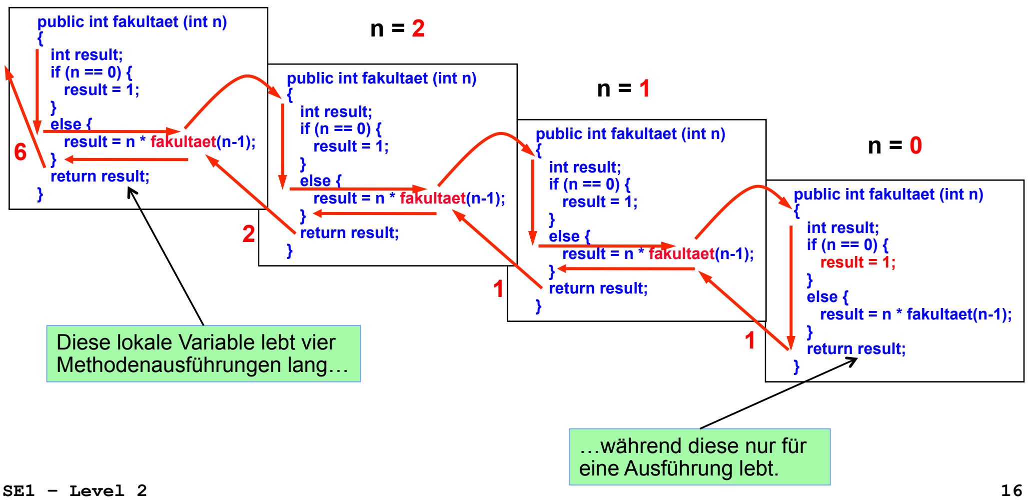
Wir erinnern uns: Lebensdauer



- 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.

Rekursion: Lebensdauer lokaler Variablen

n = 3 Vier lokale Variablen `result` leben unterschiedlich lange!



Der Aufruf-Stack
(vereinfachtes Modell)

- 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.

SE1 – Level 2 17

Rekursion: Der Aufrufstack für das Beispiel

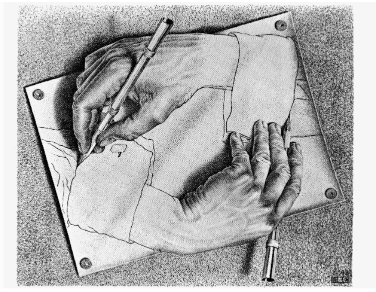
- Da die lokalen Variablen auf dem Aufrufstack gespeichert werden, können **rekursive Methodenaufrufe** einfach realisiert werden:
  - Für jeden rekursiven Aufruf wird ein neuer Satz lokaler Variablen in einem Stackframe gespeichert.
  - So kann eine Methode auf jeder Rekursionsstufe auf ihren eigenen 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

<Rücksprungadresse>	
0	
1	
<Rücksprungadresse>	
1	
1	
<Rücksprungadresse>	
2	
2	
<Rücksprungadresse>	
3	
6	
<Rücksprungadresse>	ein Stackframe für **fakultaet**
4	
24	
<Rücksprungadresse>	Stackframe der Klientenmethode, die den Ausdruck enthält
24	
23	

SE1 – Level 2 18

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: Elegante Implementation, effiziente Ausführung?

- 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:

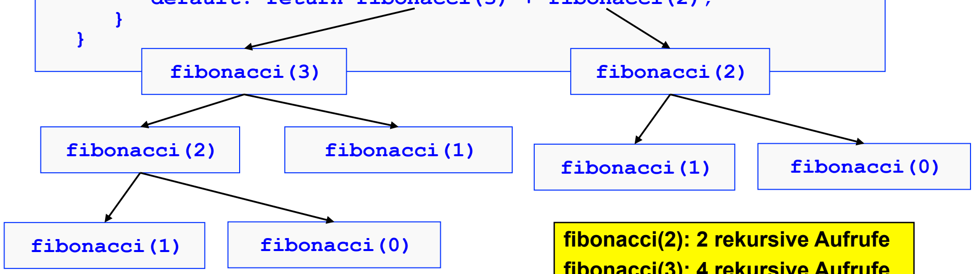
```
public int fibonacci (int n)
{
   switch (n) {
      case 1: return 0;
      case 2: return 1;
      default: return fibonacci(n-1) + fibonacci(n-2);
   }
}
```

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): 13 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!

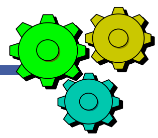
Wir werden in SE1 noch einige gute Anwendungen von Rekursion betrachten.

Rekursion: Fakultät als iteratives Programm

- Rekursive Programme hatten früher 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:
(oder ein optimierender Compiler verwenden, falls vorhanden)

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

Rekursion: Stärken und Schwächen



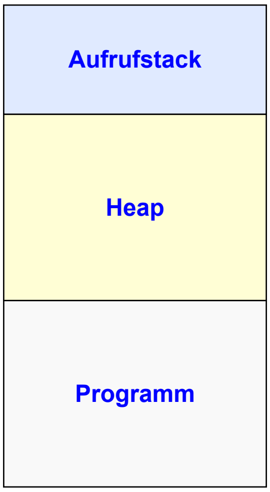
Steve McConnells Einschätzung zu Rekursion:

- 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

Vereinfachtes Speichermodell von Sprachen mit dynamischen Objekten



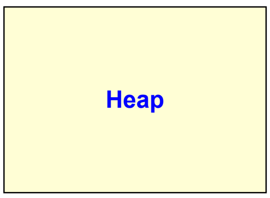
Der Speicherplatz für **lokale Variablen** (und Zwischenergebnisse von Ausdrücken) wird stapelartig durch das Laufzeitsystem verwaltet.

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.

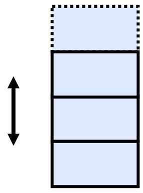
Der Heap

- 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`).

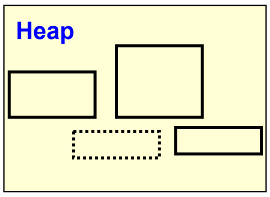


Heap und Aufrufstack

- 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`).

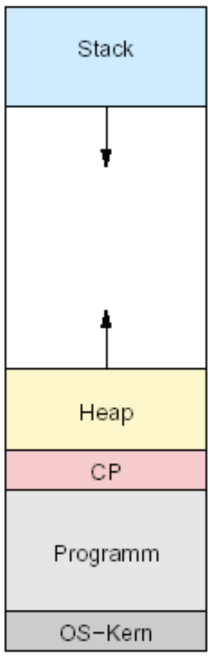


Aufrufstack



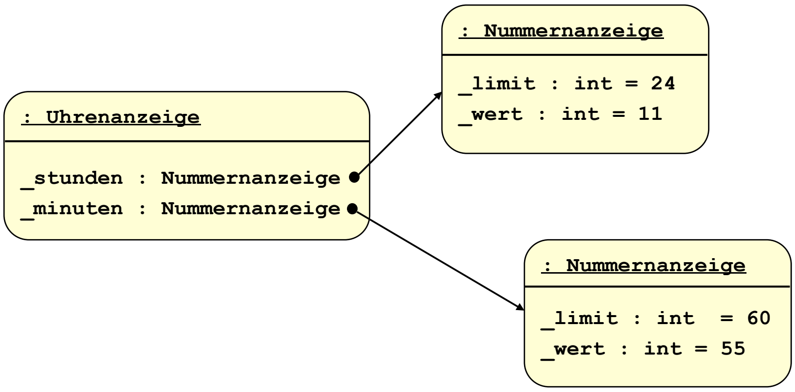
Beispiel: Speichereinteilung in einem Unix-System

- **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.



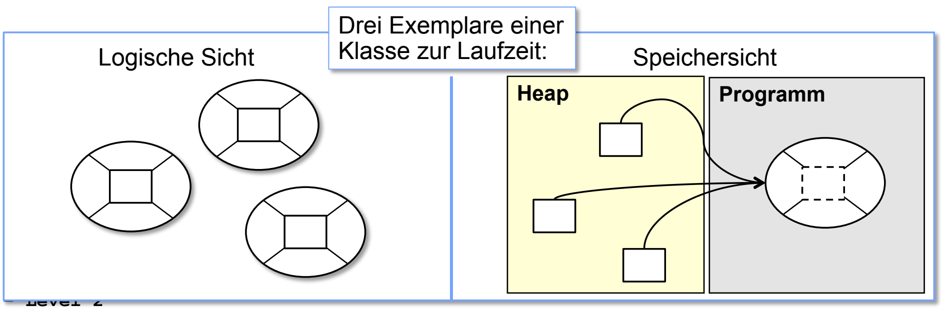
Java-Objektdiagramme: Schnappschüsse des Heap

- 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.

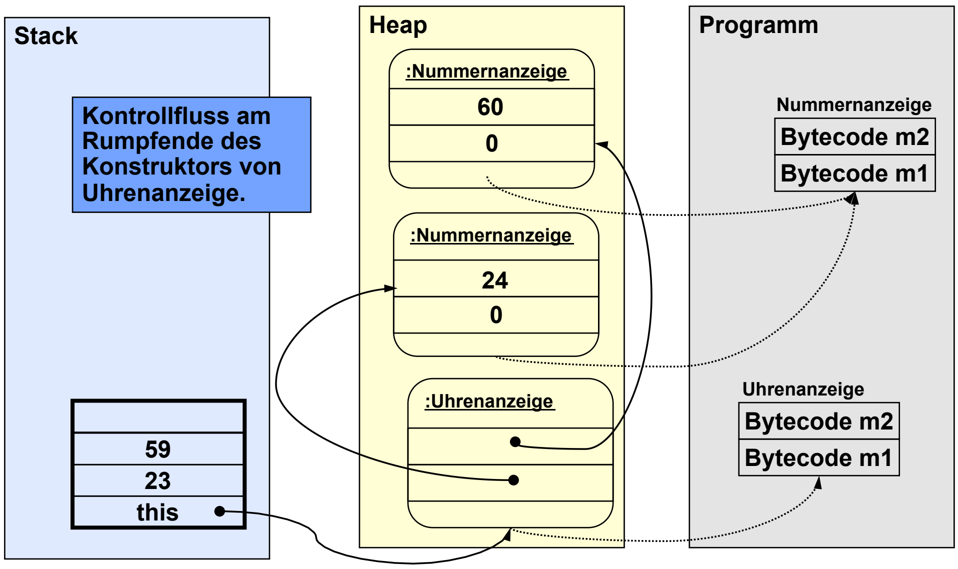


Methoden und Zustandsfelder

- 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.



Das Speichermodell zur Laufzeit am Beispiel



SE1 – Level 2 31

Der Garbage Collector in Java

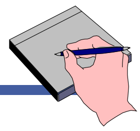
- Mit unserem Wissen über Heap und Stack können wir nun erstmalig nachvollziehen, was der **Garbage Collector** von Java macht.
- Die Voraussetzungen sind:
  - **Alle** Objekte eines Java-Programms liegen im Heap.
  - Auf dem **Aufrufstack** in den Speicherplätzen für die lokalen Variablen liegen entweder 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 32

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.

Zeichenketten in Programmiersprachen



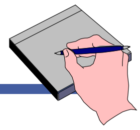
- Moderne Programmiersprachen bieten Unterstützung für **Zeichenketten** (engl.: strings). Eine Zeichenkette ist eine Folge von einzelnen Zeichen.

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

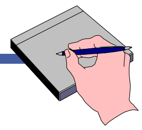


- In Java werden Zeichenketten primär durch die Klasse `String` unterstützt. Diese Klasse definiert, wie alle Klassen in Java, einen Typ.
- `String` ist in Java ein expliziter Bestandteil der Sprache, denn es gibt einige Spezialbehandlungen für diesen Typ:
  - **`String`-Literale** (Zeichenfolgen zwischen 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:
  - https://docs.oracle.com/javase/8/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`

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…
```


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!**

Axel Schmolitzky, Heinz Züllighoven, et al.