The Code in Your Brain

XP Days 2024

andrena OBJECTS

10. Oktober 2024

Stefan Mandel

Stefan.Mandel@andrena.de

Ist das verständlich?

```
void transferMoney(Acccount from, Acccount to, int dollars) {
    from.dec(dollars);
    to.inc(dollars);
}
```

- Ja …

```
class Account {
    int priority;
    int balance;

    void dec(int delta) { priority -= delta; }

    void inc(int delta) { priority += delta; }

    void deposit(int dollars) { balance += dollars; }

    void withdraw(int dollars) { balance -= dollars; }
}
```

- WTF!

Welches Ergebnis hat die Funktion?

```
String combine(String s1, String s2) {
    var buffer = new StringBuilder(s1);
    buffer.append("-");
    buffer.append(s2);

    var string = buffer.toString();
    string.replace("-", " ");

    return string.toLowerCase();
}
```

- Was erwarten wir bei:

```
System.out.println(combine("Hello", "World"));
```

- Ausgabe:

```
Hello-World
```

- Warum?

Agenda

Verständnis (und Missverständnis)

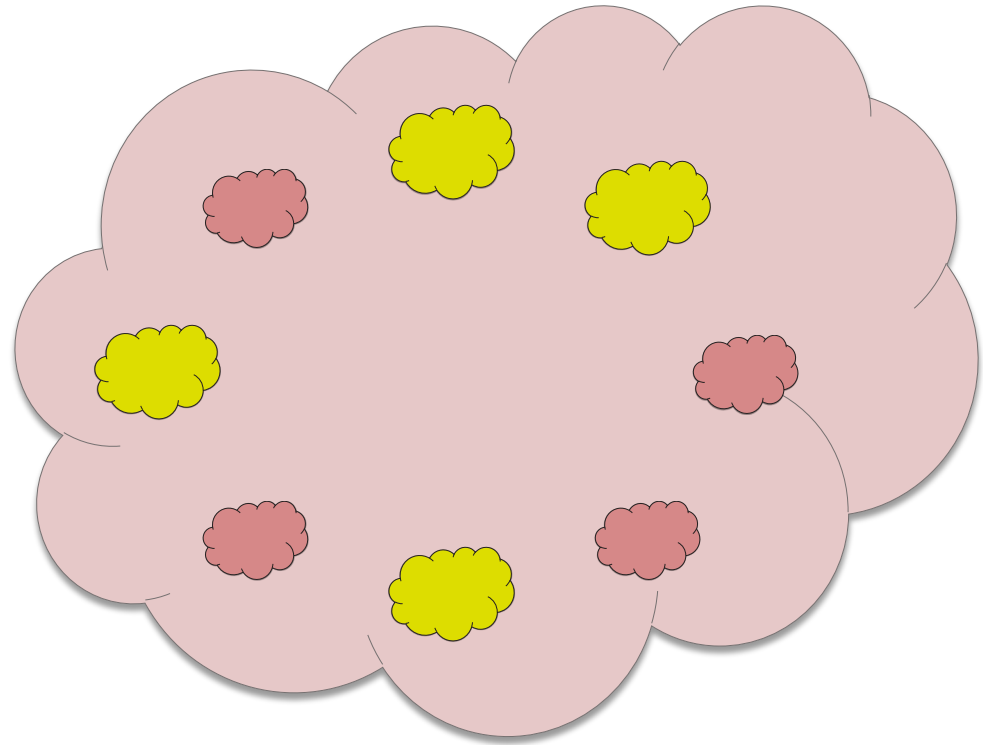
- enstehen im Gedächtnis
- das Gedächtnis im psychologischen Sinne umfasst die Prozesse
  - Erfassen von Information
  - Abrufen von Information
  - Kombinieren von Information
  - Speichern von Information

Folgende Aspekte werden wir beleuchten

- ein Gedächtnismodell aus der Psychologie (Miller, Baddeley, Cowan)
- ein psychologisches Modell aus der Verhaltensökonomomik (Kahneman)
- ein Modell von Wissen aus der Informatik (diverse)

Das Gedächtnis der Kognitionspsychologie

- Langzeitgedächtnis
  - speichert Chunks
  - potentiell unlimitiert
- Arbeitsgedächtnis
  - aktiviert Chunks
  - bis zu 4 ± 1 gleichzeitig



Chunks …

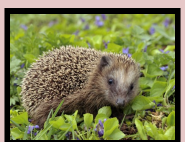
Igel

[1][2]

Informationen

(Bilder, Wörter)

sind objektiv



Igel

Chunks

(Vorstellungen, Ideen)

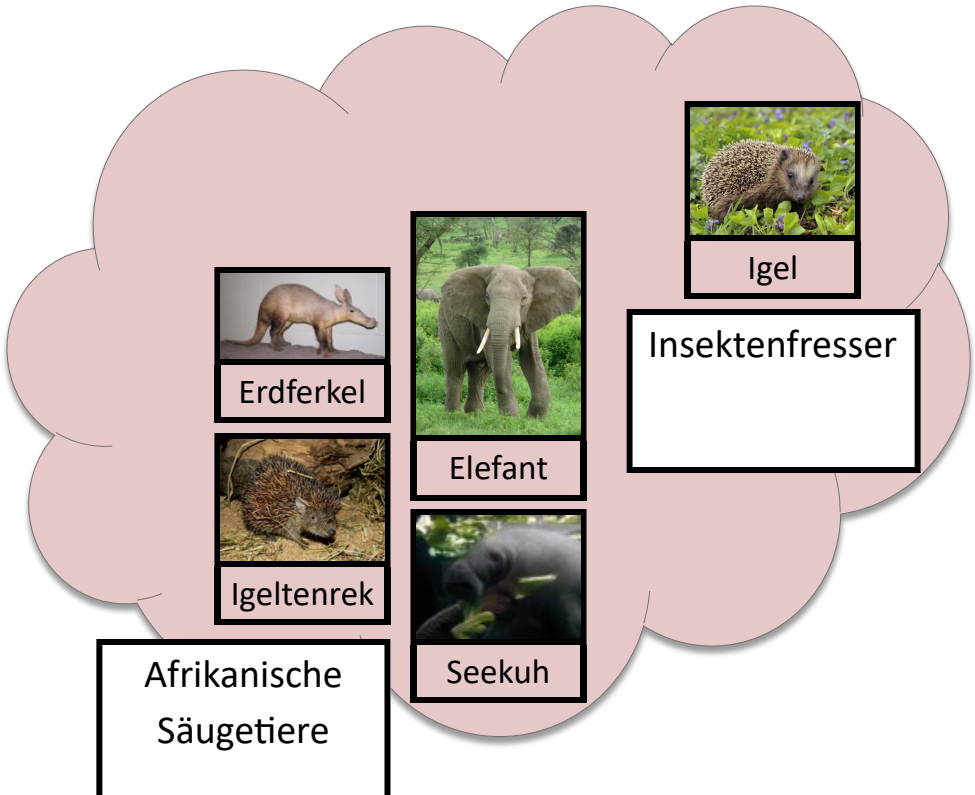
sind aber individuell

1. Kleiner Igeltenrek (Echinops telfairi), 2011 - Von Николай Усик / http://paradoxusik.livejournal.com/, CC BY-SA 3.0 https://creativecommons.org/licenses/by-sa/3.0, via Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Echinops_telfairi_Plzen_zoo_02.2011.jpg ↩
2. Braunbrustigel (Erinaceus europaeus), 2013 - Von © Michael Gäbler, CC BY-SA 3.0 https://creativecommons.org/licenses/by-sa/3.0, via Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Erinaceus_europaeus_(Linnaeus,_1758).jpg ↩

… im Kopf eines Experten

Igel

[1][2][3][4][5]



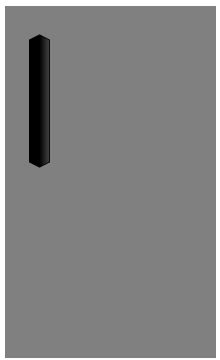
1. Kleiner Igeltenrek (Echinops telfairi), 2011 - Von Николай Усик / http://paradoxusik.livejournal.com/, CC BY-SA 3.0 https://creativecommons.org/licenses/by-sa/3.0, via Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Echinops_telfairi_Plzen_zoo_02.2011.jpg ↩
2. Braunbrustigel (Erinaceus europaeus), 2013 - Von © Michael Gäbler, CC BY-SA 3.0 https://creativecommons.org/licenses/by-sa/3.0, via Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Erinaceus_europaeus_(Linnaeus,_1758).jpg ↩
3. Manati (Trichechus manatus), 2005 - Von Chris Muenzer, CC BY 2.0 https://creativecommons.org/licenses/by/2.0, via Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Hpim0279.jpg ↩
4. Afrikanischer Elefant (Loxodonta africana), 2007 - Von The author is nickandmel2006 on flickr, CC BY-SA 2.0 https://creativecommons.org/licenses/by-sa/2.0, via Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Elephant_near_ndutu.jpg ↩
5. Erdferkel (Orycteropus afer) 2007 - Von I, Masur, CC BY-SA 3.0 http://creativecommons.org/licenses/by-sa/3.0/, via Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Orycteropus_afer_stuffed.jpg ↩

Eigenschaften von Chunks

- sind hochindividuell
- können andere Chunks enhalten, z.B.
  - Stacheln
  - Schnauze
  - Kurze Beine
- sind fehlertolerant
  - passen auch auf ähnliche Tiere
- große Chunks (Abstraktionen) werden vor kleineren (Details) wahrgenommen
  - Igel vor Stacheln

Ein Chunk

ist nicht nur die Summe seiner Unterchunks

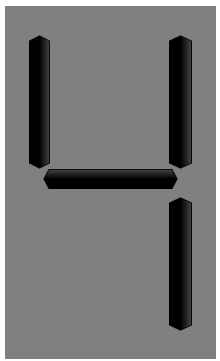


1 Balken

1 Chunk

Ein Chunk

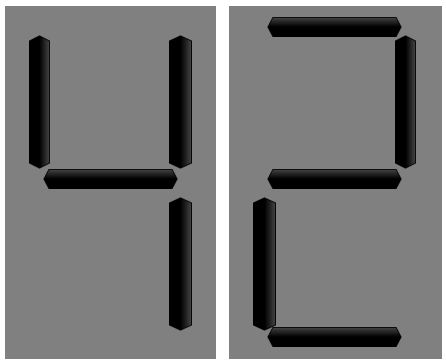
ist nicht nur die Summe seiner Unterchunks



Ziffer 4 1 Chunk(4 Balken)

Ein Chunk

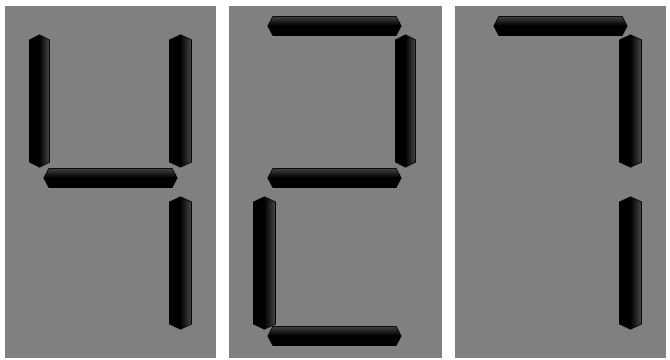
ist nicht nur die Summe seiner Unterchunks



Zahl 42 1 Chunk(9 Balken)

Ein Chunk

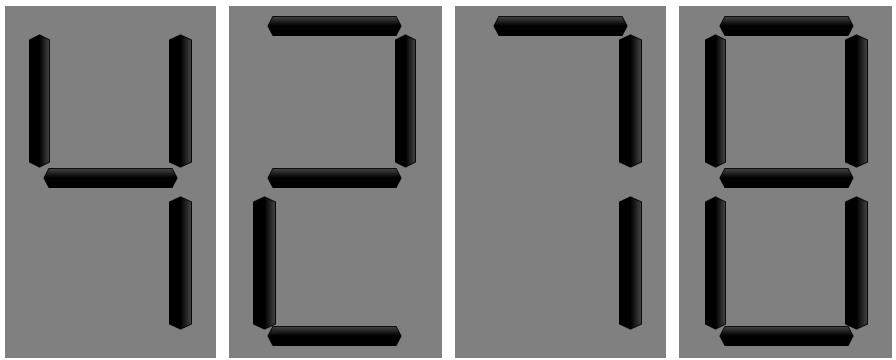
ist nicht nur die Summe seiner Unterchunks



Zahl 427 1 Chunk(12 Balken)

Ein Chunk

ist nicht nur die Summe seiner Unterchunks



Zahl 4278 1 Chunk(19 Balken)

Zu viele Chunks

Das Arbeitsgedächtnis

- kann 4 Chunks in einem Schritt verarbeiten
- was wenn es zu viele Chunks präsentiert bekommt?

Das Arbeitsgedächtnis

- fokussiert die 4 besten Chunks
- anhand erlernter Heuristiken
- passt meistens, aber nicht immer

Bekannte Heuristiken
(europäische Kultur)

- von oben nach unten
- von links nach rechts
- Großes/Fettes/Leuchtendes/Ungewöhnliches zuerst

Bekannte Heuristiken
(deutsche Sprache)

- Dativ-Objekt vor Akkusativ-Objekt

Das Gedächtnis als informatisches Modell

- Gängige Modelle sind
  - Regelbasiertes System
  - Neuronales Netz
- Beide Modelle sind deklarativ, d.h. Resultate sind abhängig
  - von Fakten/Eingabedaten
  - und Regeln/Vernetzungen
- Imperative Prozesse sind für dieses System schwer
  - und müssen simuliert werden
  - indem für jede Zwischenerkenntnis ein "Adhoc"-Chunk gespeichert wird
  - der potentiell kollidiert mit älteren Erkenntnissen
  - deswegen schwierig

Das Gedächtnis als Verhaltenmodell

Aus "Thinking, Fast and Slow" (Daniel Kahneman)

- Unser Verhalten wird gesteuert durch 2 Systeme
  - System 1: schnell, naiv, angenehm
  - System 2: langsam, sorgfältig, anstrengend
  - System 2 schaltet sich nur zu, wenn System 1 "Zweifel" hat
- das Buch demonstriert anschaulich
  - wie einfach es ist das schnelle System 1 auszutricksen
    - indem man alles was Zweifel erzeugt vermeidet
    - oder indem man die Zweifel durch Druck unterdrückt
  - wie wichtig es ist System 2 einzuschalten
    - … wenn man mit potentiell feindlichen Gegenspielern zu tun hat
- die Ergebnisse lassen sich aber auch uminterpretieren
  - wenn man es mit potentiell freundlichen Spielern zu tun hat (Team-Mitgliedern)
  - und es vermeidet andere System 1 auszutricksen

Chunking the Code

Code-Verständnis entsteht vor allem dann …

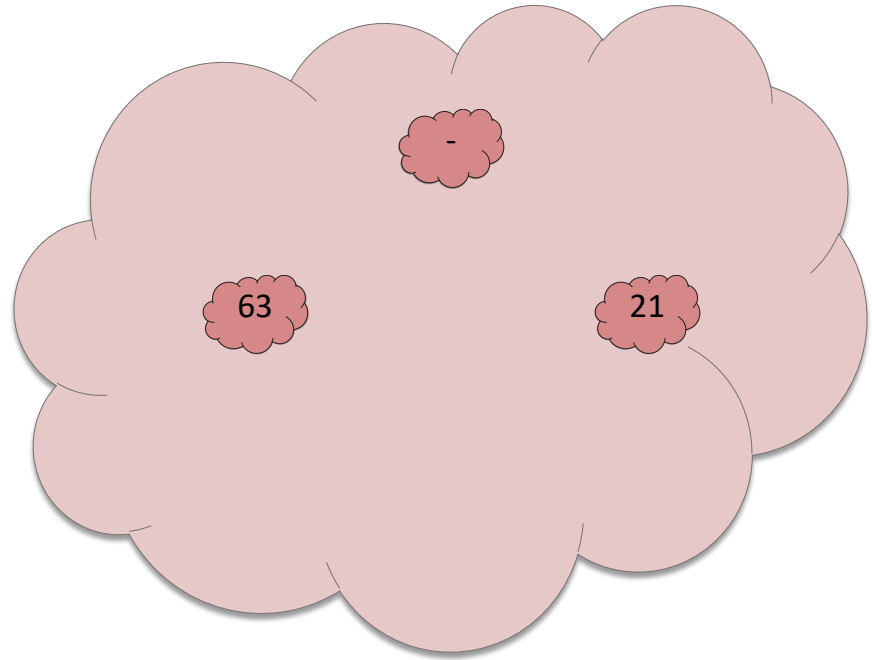
- wenn wir Chunks vorfinden, die wir kennen
- wenn wir 4 Chunks auf einmal vorfinden
- oder wenn die Heuristiken greifen und das Problem zerteilen können

Wir werden Beispiele sehen:

- Über mathematische Operationen
- Über Methodenaufrufe
- Über Mutability
- Über Duplikation
- Über Aliasing
- Über Konventionen

Binäre Operationen

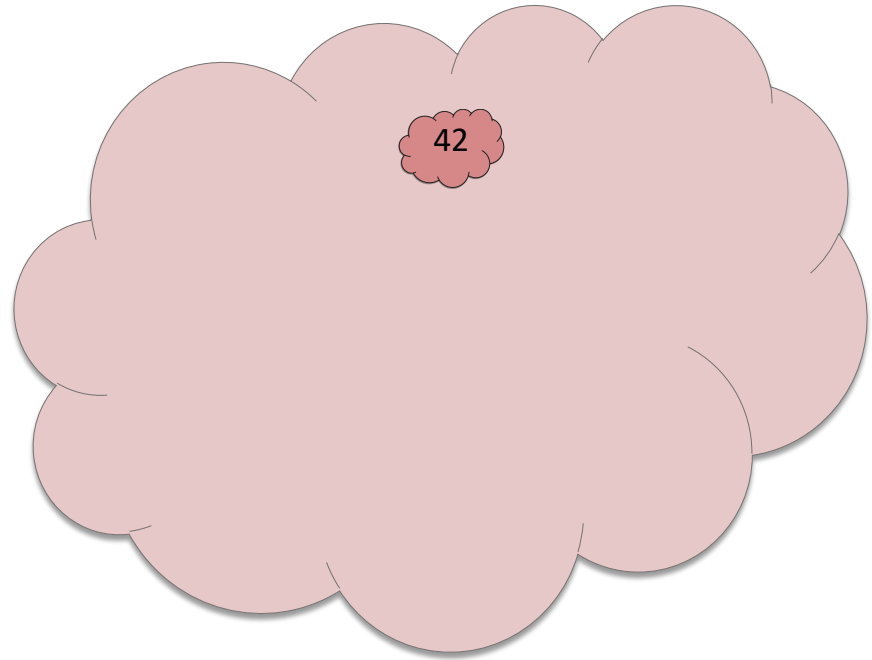
63 - 21



- 3 Chunks - verständlich

Binäre Operationen

42



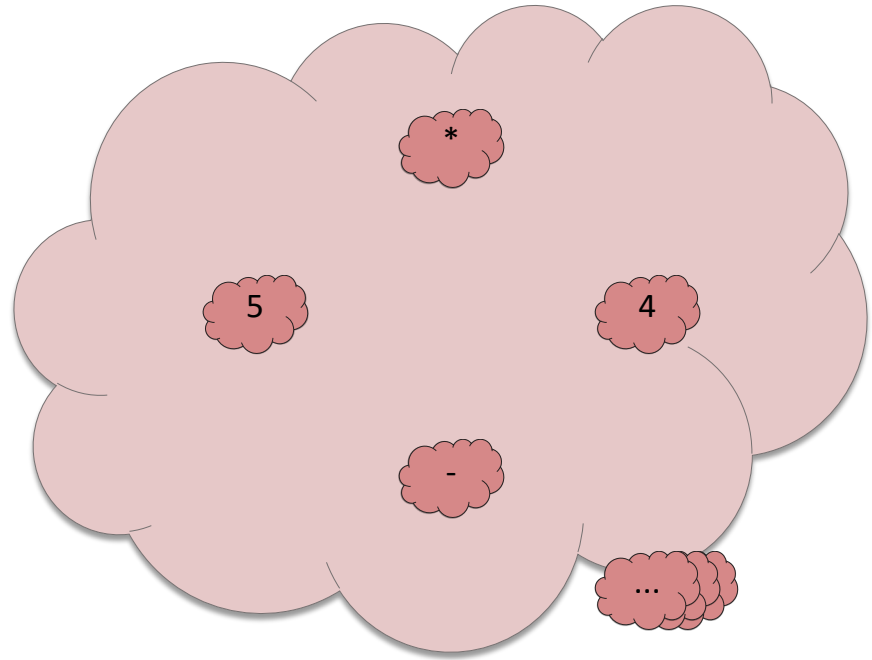
- 3 Chunks - verständlich

Hinweis

Elementare Operationen müssen ins Arbeitsgedächtnis passen

Komplexe Operationen

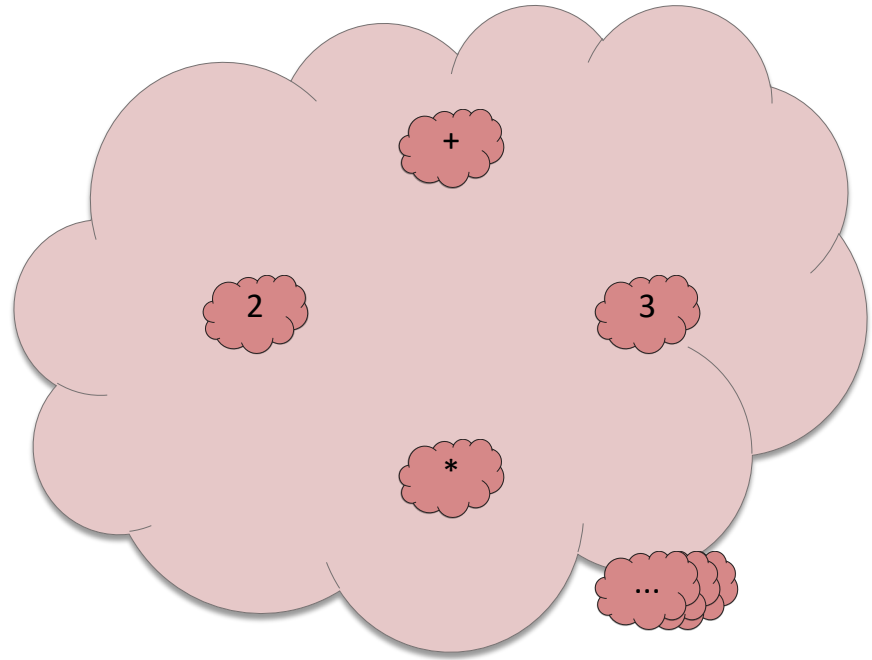
5 * 4 - 8 / 2



- mehr als 4 Chunks
- Heuristik "Von links nach rechts"

Komplexe Operationen

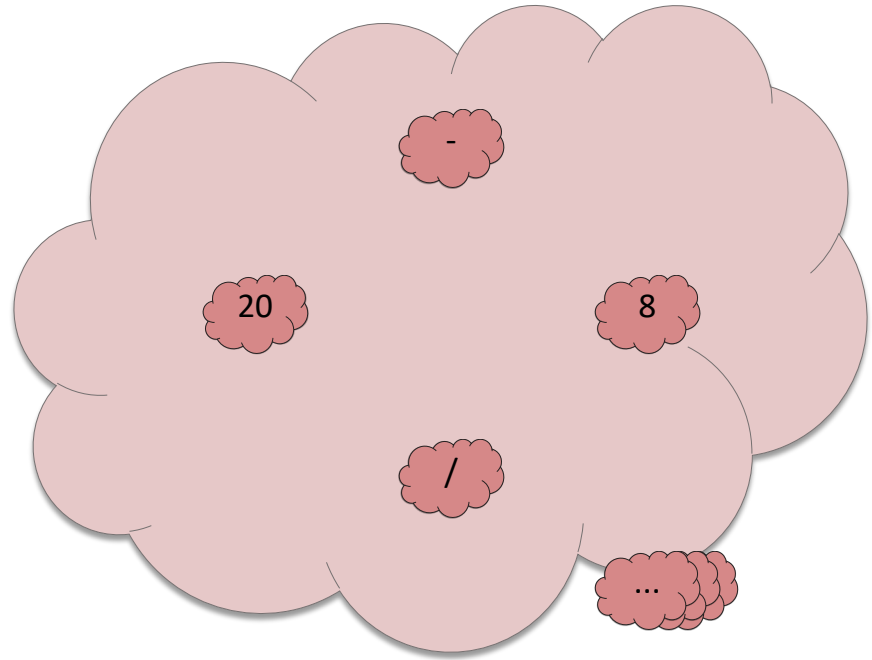
2 + 3 * 4 - 8 / 2



- mehr als 4 Chunks
- Heuristik "Von links nach rechts"

Komplexe Operationen

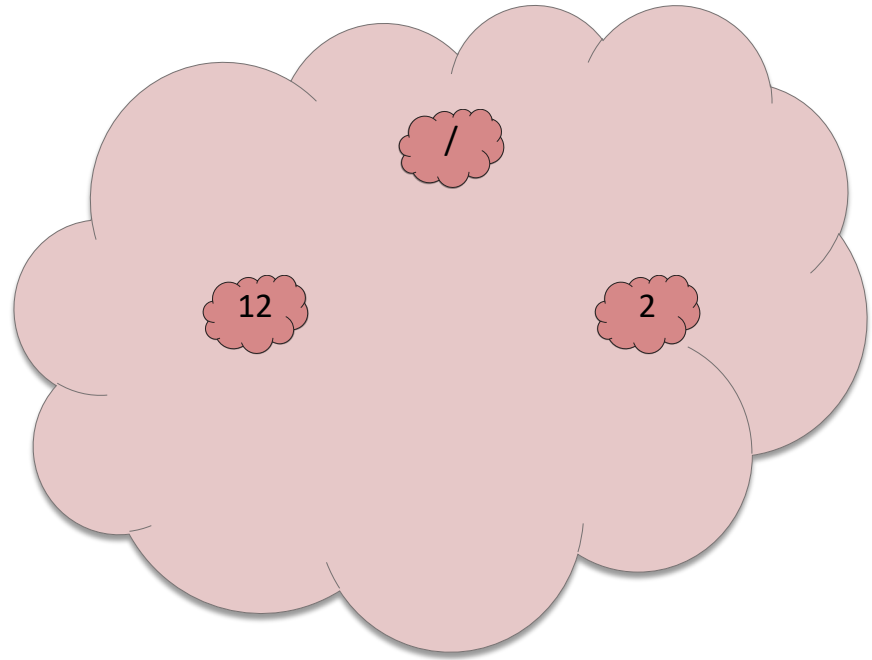
20 - 8 / 2



- mehr als 4 Chunks
- Heuristik "Von links nach rechts"

Komplexe Operationen

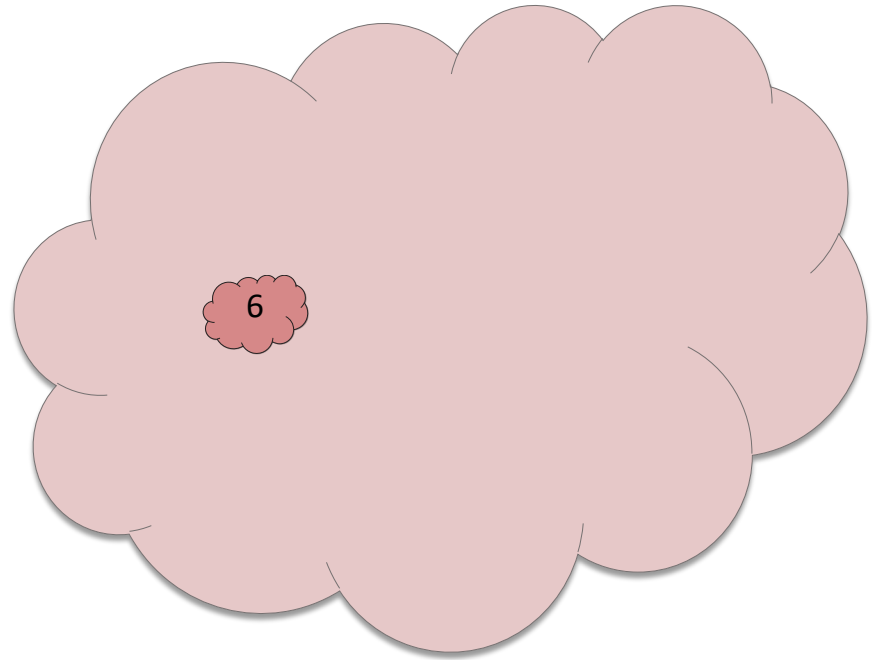
12 / 2



- mehr als 4 Chunks
- Heuristik "Von links nach rechts"

Komplexe Operationen

6



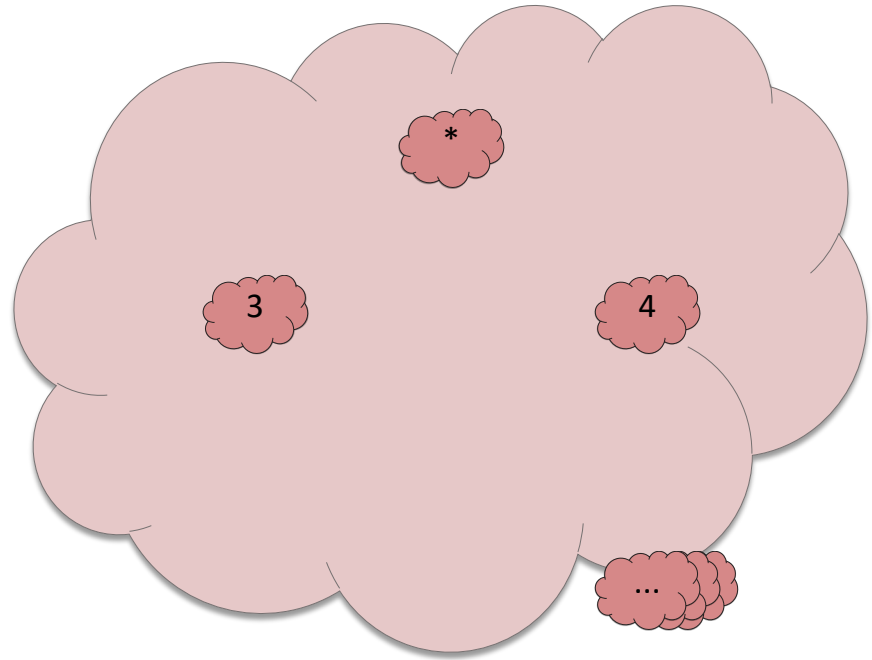
- mehr als 4 Chunks
- Heuristik "Von links nach rechts"
- Leider falsch

Komplexe Operationen

2

+ 3 * 4

- 8 / 2



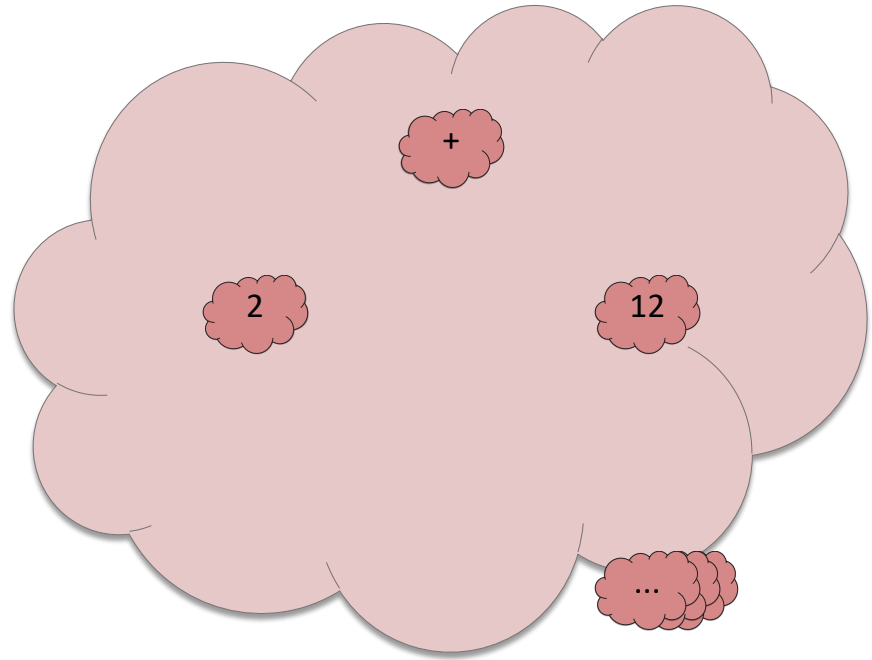
- mehr als 4 Chunks
- Heuristik "Von oben nach unten"

Komplexe Operationen

2

+ 12

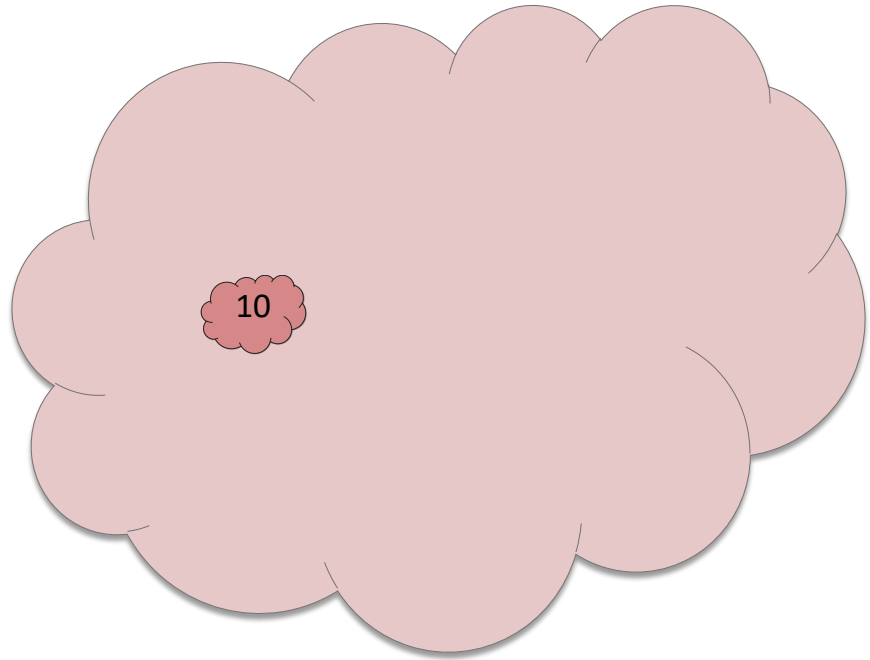
- 4



- mehr als 4 Chunks
- Heuristik "Von oben nach unten"

Komplexe Operationen

10



- mehr als 4 Chunks
- Heuristik "Von oben nach unten"
- Richtig

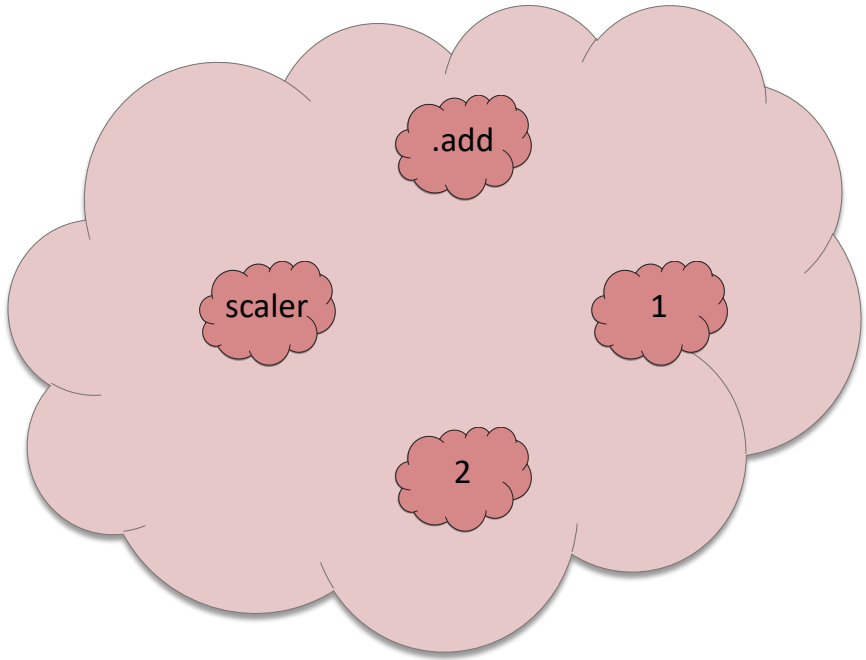
Mein Tipp

Nächster Schritt - nächste Zeile

Einfache Anweisungen

```
Scaler scaler = new Scaler();
int result = scaler.add(1, 2);
```

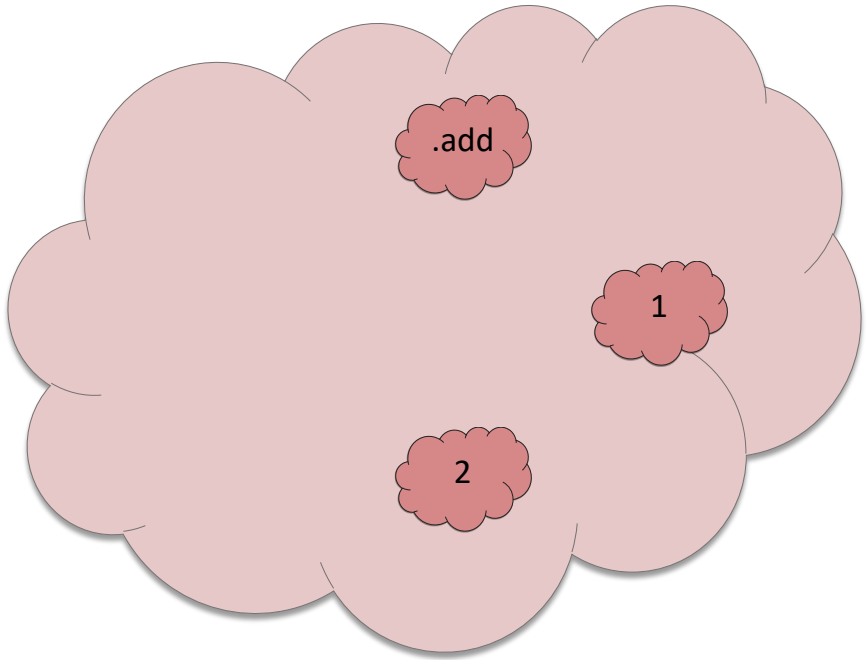
- genau 4 Chunks - verständlich



Einfache Anweisungen

```
Scaler scaler = new Scaler();
int result = scaler.add(1, 2);
```

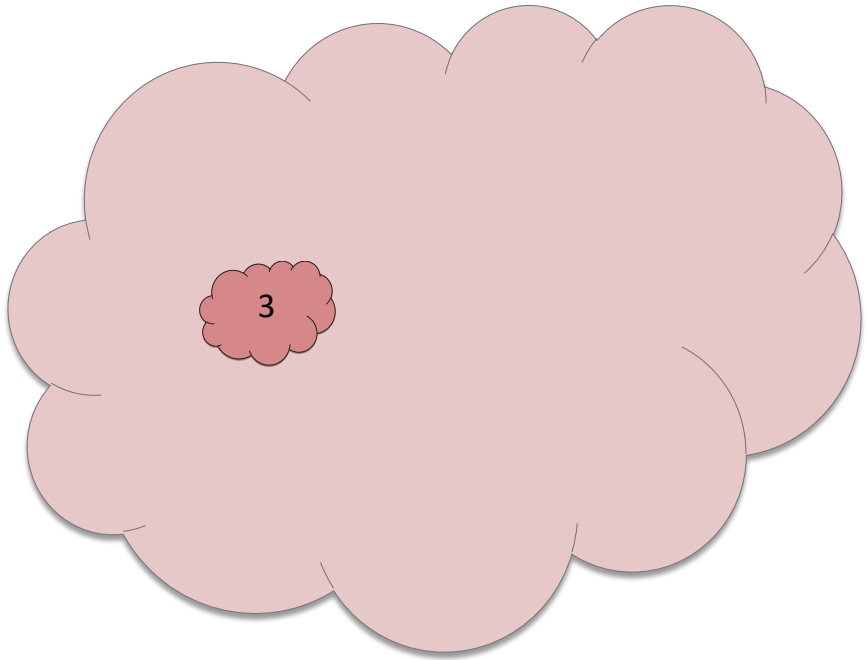
- genau 4 Chunks - verständlich
- bei puren Funktionen sogar 3 Chunks - verständlicher



Einfache Anweisungen

```
Scaler scaler = new Scaler();
int result = scaler.add(1, 2);
```

- genau 4 Chunks - verständlich
- bei puren Funktionen sogar 3 Chunks - verständlicher



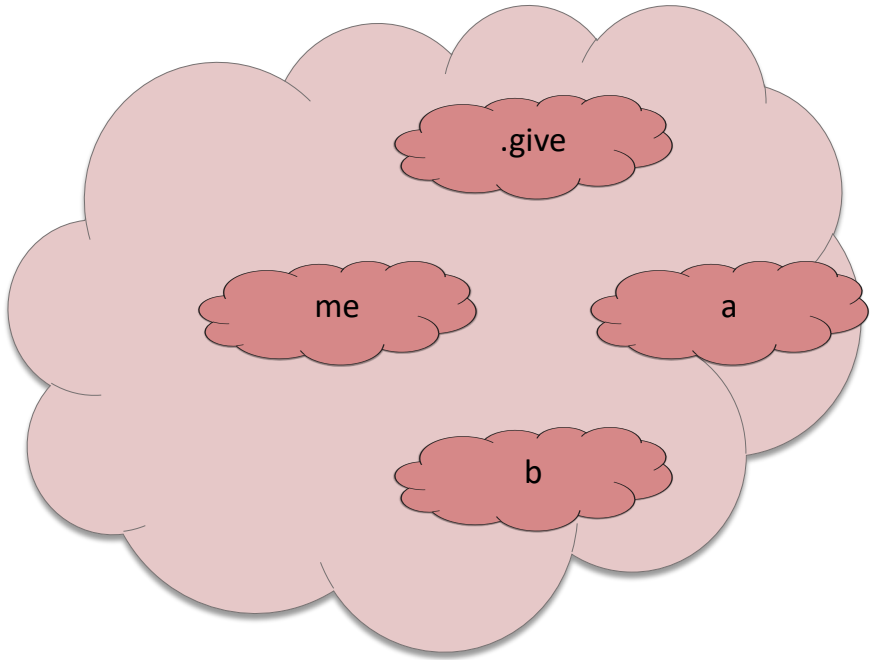
Clean Code

Höchsten 2-3 Argumente!

Einfache Anweisungen

```
Person me = new Person();
me.give(a, b);
```

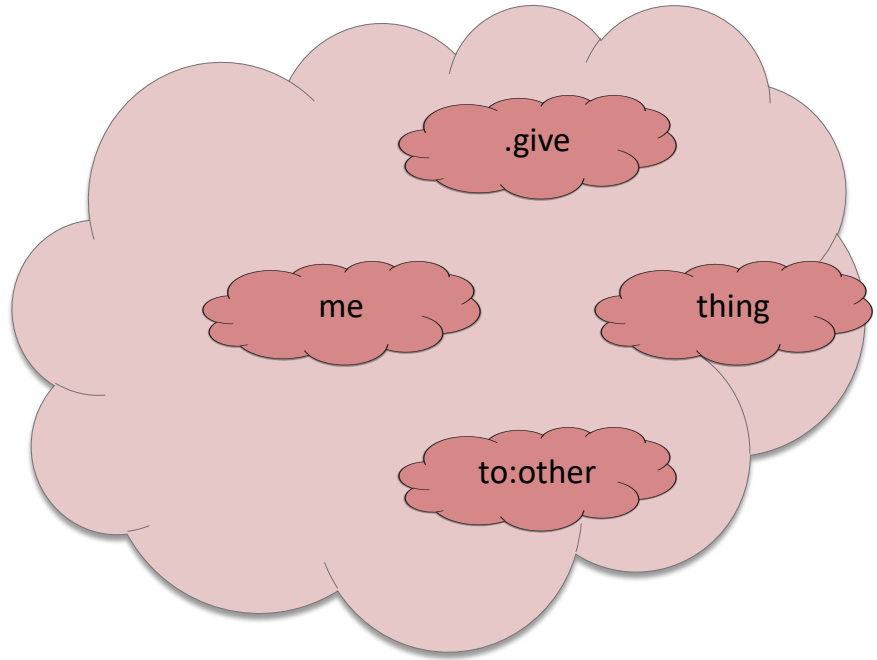
- 4 Chunks
  - aber nicht voll verständlich



Einfache Anweisungen

```
Person me = new Person();
me.give(thing, other);
```

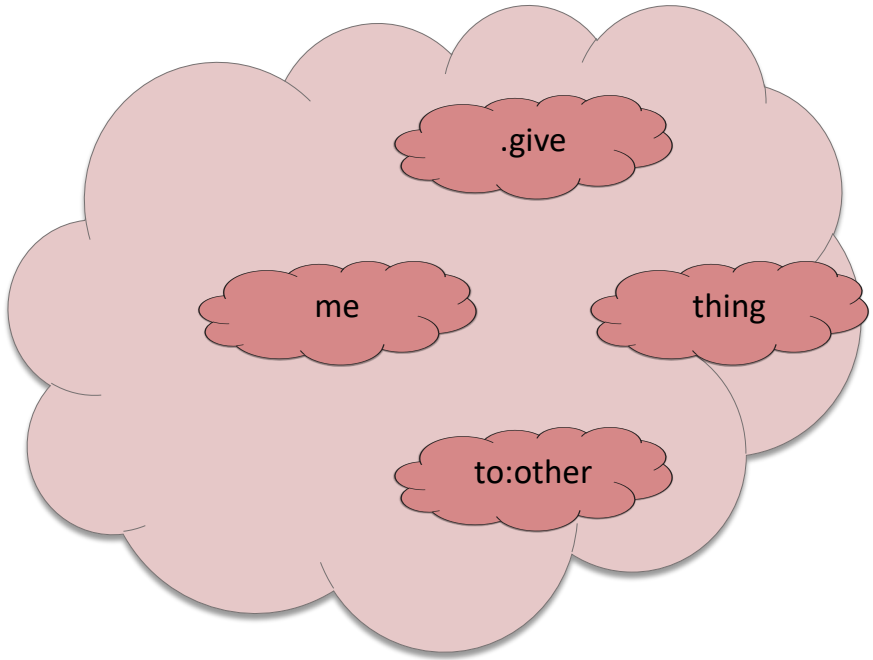
- 4 Chunks
  - aber nicht voll verständlich
- Heuristiken sagen uns
  - Deutsch:
    - `a` ist die andere Person (Dativ), `b` ist der Gegenstand (Akkusativ)
  - Englisch:
    - `a` ist der Gegenstand (Akkusativ), `b` ist die andere Person (Dativ, mit `to`)
- Klarer wird es mit Naming



Einfache Anweisungen

```
Person me = new Person();
me.give(thing).to(other);
```

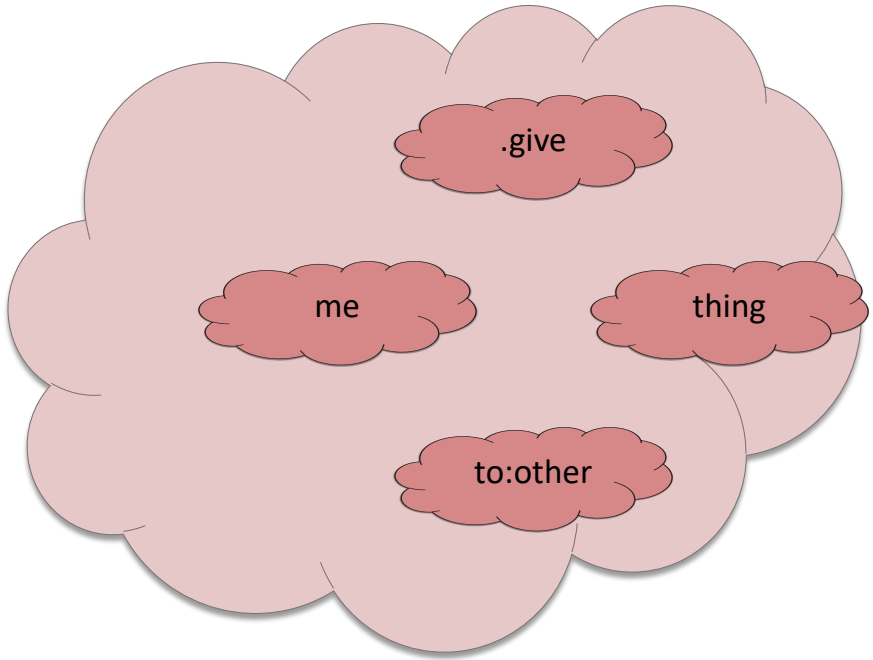
- 4 Chunks
  - aber nicht voll verständlich
- Heuristiken sagen uns
  - Deutsch:
    - `a` ist die andere Person (Dativ), `b` ist der Gegenstand (Akkusativ)
  - Englisch:
    - `a` ist der Gegenstand (Akkusativ), `b` ist die andere Person (Dativ, mit `to`)
- Klarer wird es mit Naming
- Oder Argumentrollen, z.B. in fluent APIs



Einfache Anweisungen

```
Person me = new Person();
me.give(a, b);
```

- 4 Chunks
  - aber nicht voll verständlich
- Heuristiken sagen uns
  - Deutsch:
    - `a` ist die andere Person (Dativ), `b` ist der Gegenstand (Akkusativ)
  - Englisch:
    - `a` ist der Gegenstand (Akkusativ), `b` ist die andere Person (Dativ, mit `to`)
- Klarer wird es mit Naming
- Oder Argumentrollen, z.B. in fluent APIs



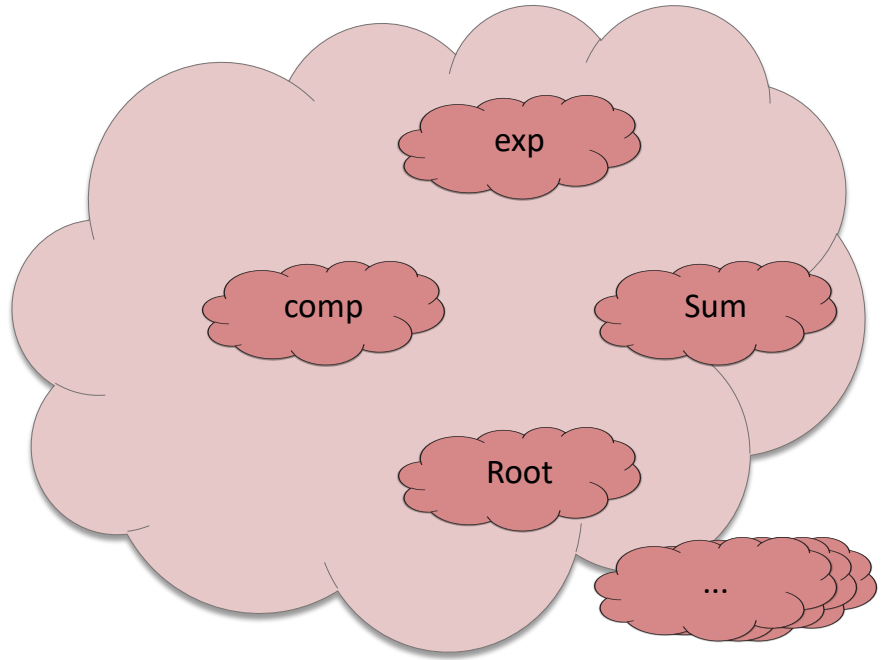
Mein Tipp

Rollen der Argumente klären

Komplexe Anweisungen

```
Computer comp = new Computer();
double result = comp.expSumAndRoot(3, 2, 4, 2, 2);
```

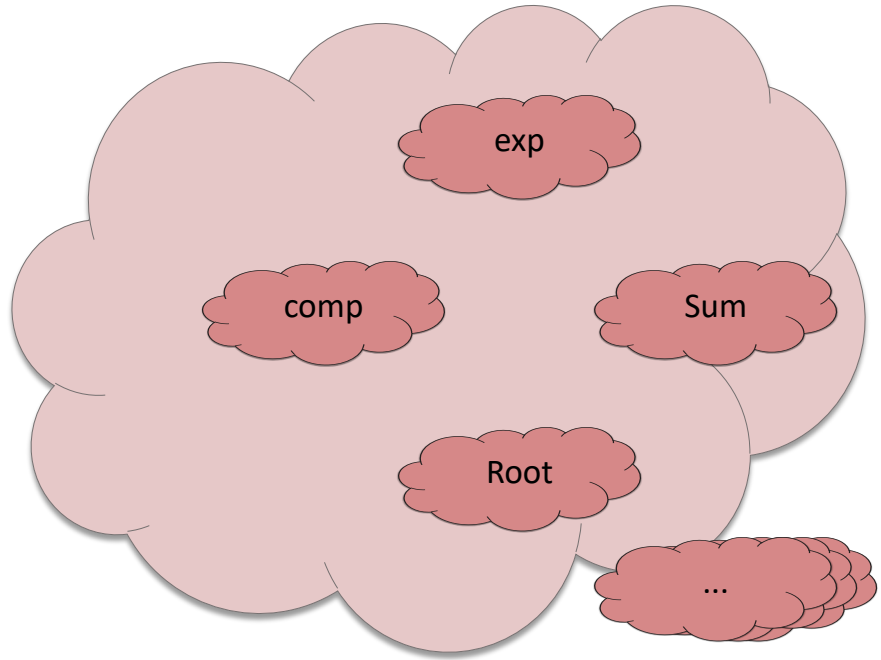
- mehr als 4 Chunks
  - `expSumAndRoot` ist ein "Wort"
  - aber im Gehirn 3 Chunks: `exp` , `sum` , `root`
  - eher unverständlich



Komplexe Anweisungen

```
Computer comp = new Computer();
double result = comp.expSumAndRoot(3, 2, 4, 2, 2);
```

- mehr als 4 Chunks
  - `expSumAndRoot` ist ein "Wort"
  - aber im Gehirn 3 Chunks: `exp` , `sum` , `root`
  - eher unverständlich



Komplexe Anweisungen

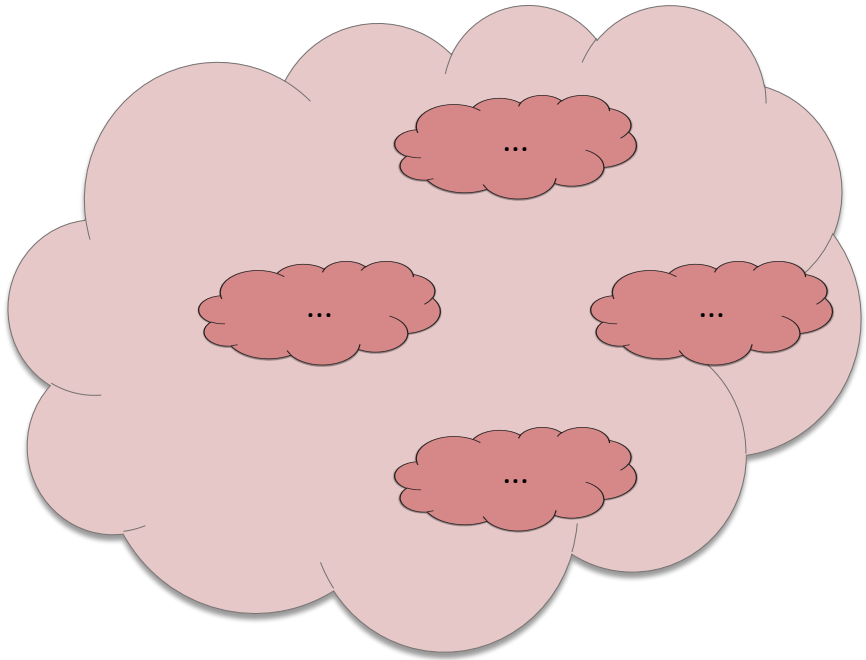
```
Computer comp = new Computer();

double exp1 = comp.exp(3, 2);
double exp2 = comp.exp(4, 2);

double sum = comp.sum(exp1, exp2);

double result = comp.root(sum, 2);
```

- Aufteilung in mehrere Schritte



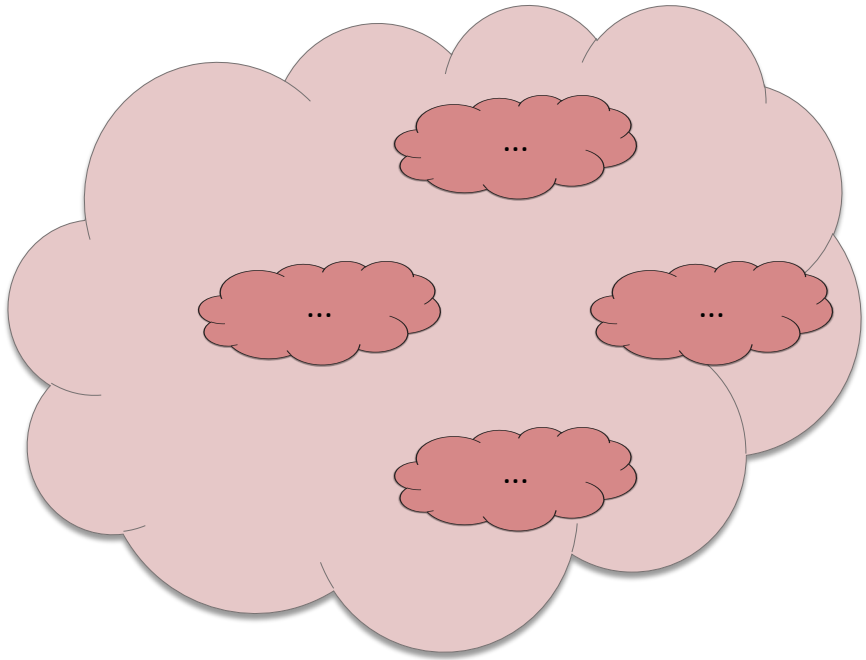
Komplexe Anweisungen

```java
Computer comp = new Computer();

double exp1 = 3 * 3; // comp.exp(3, 2)
double exp2 = 4 * 4; // comp.exp(4, 2)

double sum = exp1 + exp2; // comp.sum(exp1, exp2)

double result = Math.sqrt(sum); // comp.root(sum, 2)
```



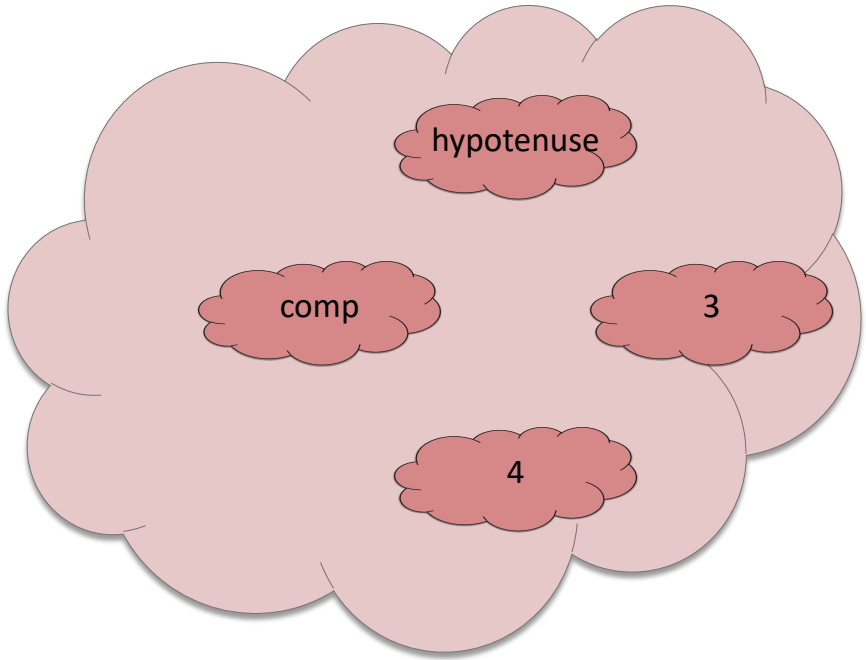
- Aufteilung in mehrere Schritte
- Ersetzung durch bekanntere Chunks
  - Rechenoperatoren statt Methoden

Komplexe Anweisungen

```
Computer comp = new Computer();

double result = comp.expSumAndRoot(3, 2, 4, 2, 2);
```

- Aufteilung in mehrere Schritte
- Ersetzung durch bekanntere Chunks
  - Rechenoperatoren statt Methoden
- Ersetzung durch eine bekannte Abstraktion
  - Hypotenuse statt Wurzel aus den Argumentquadraten



Mein Tipp

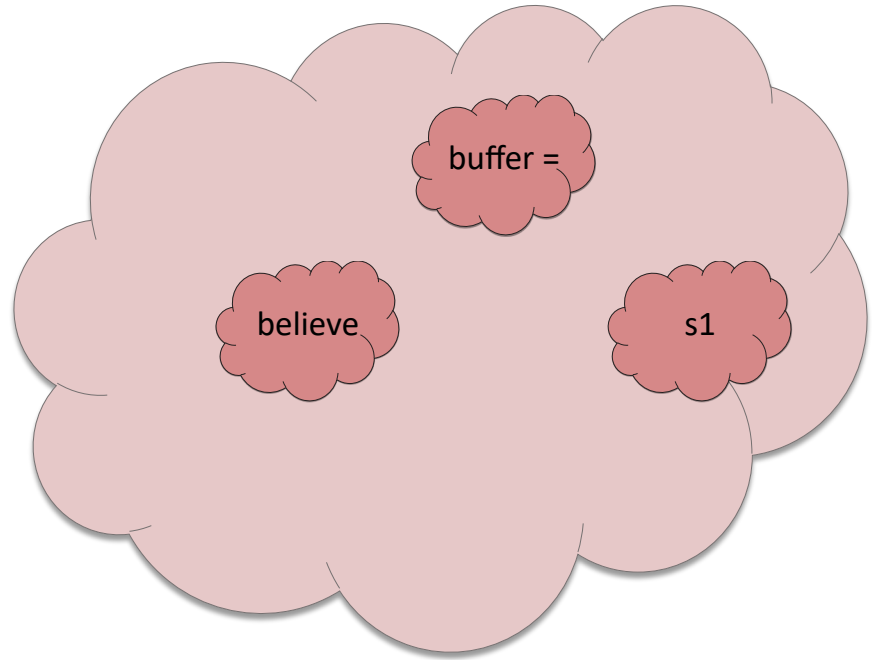
Komplexoperationen auftrennen oder abstrahieren

Mutability

```
var buffer = new StringBuilder(s1);
buffer.append("-");
buffer.append(s2);

var string = buffer.toString();
string.replace("-", " ");

return string.toLowerCase();
```



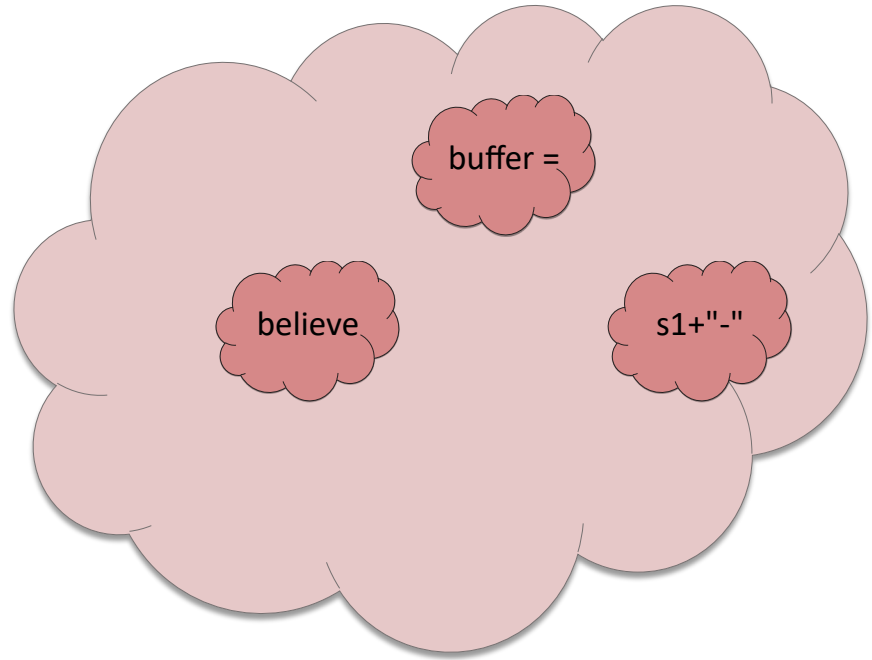
- bisher waren alle Schritte des Verständnisses nur lesend
- neu:
  - zu jeder Zeile den neuen Zustand speichern
  - und alten Zustand invalidieren
  - deutlich langsamer als Lesen

Mutability

```
var buffer = new StringBuilder(s1);
buffer.append("-");
buffer.append(s2);

var string = buffer.toString();
string.replace("-", " ");

return string.toLowerCase();
```



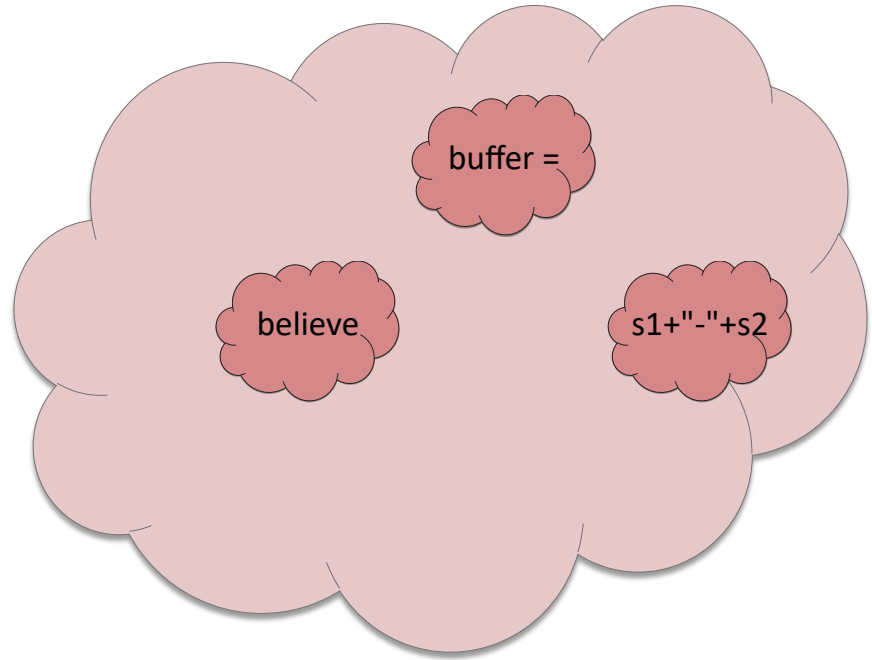
- bisher waren alle Schritte des Verständnisses nur lesend
- neu:
  - zu jeder Zeile den neuen Zustand speichern
  - und alten Zustand invalidieren
  - deutlich langsamer als Lesen

Mutability

```
var buffer = new StringBuilder(s1);
buffer.append("-");
buffer.append(s2);

var string = buffer.toString();
string.replace("-", " ");

return string.toLowerCase();
```



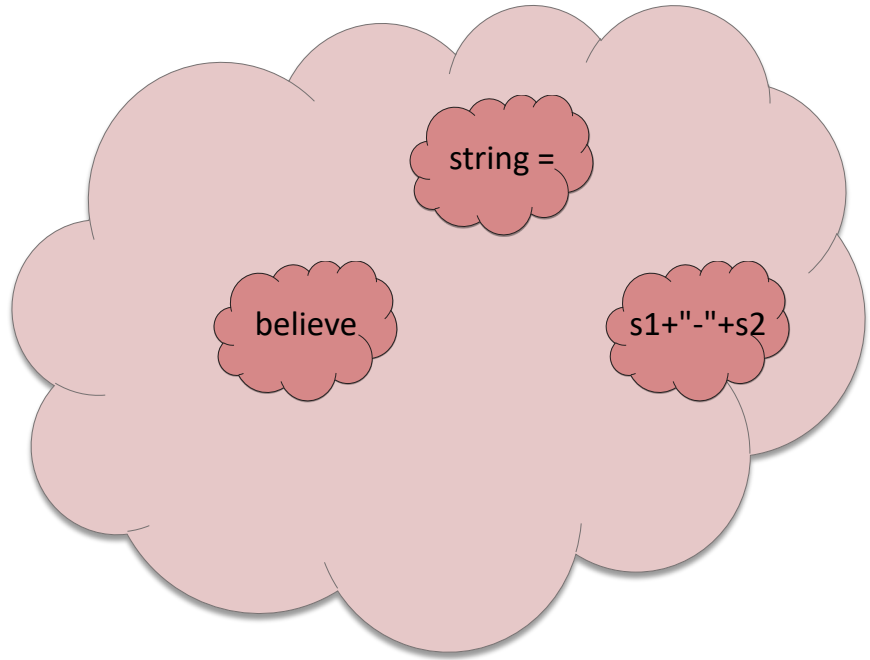
- bisher waren alle Schritte des Verständnisses nur lesend
- neu:
  - zu jeder Zeile den neuen Zustand speichern
  - und alten Zustand invalidieren
  - deutlich langsamer als Lesen

Mutability

```
var buffer = new StringBuilder(s1);
buffer.append("-");
buffer.append(s2);

var string = buffer.toString();
string.replace("-", " ");

return string.toLowerCase();
```



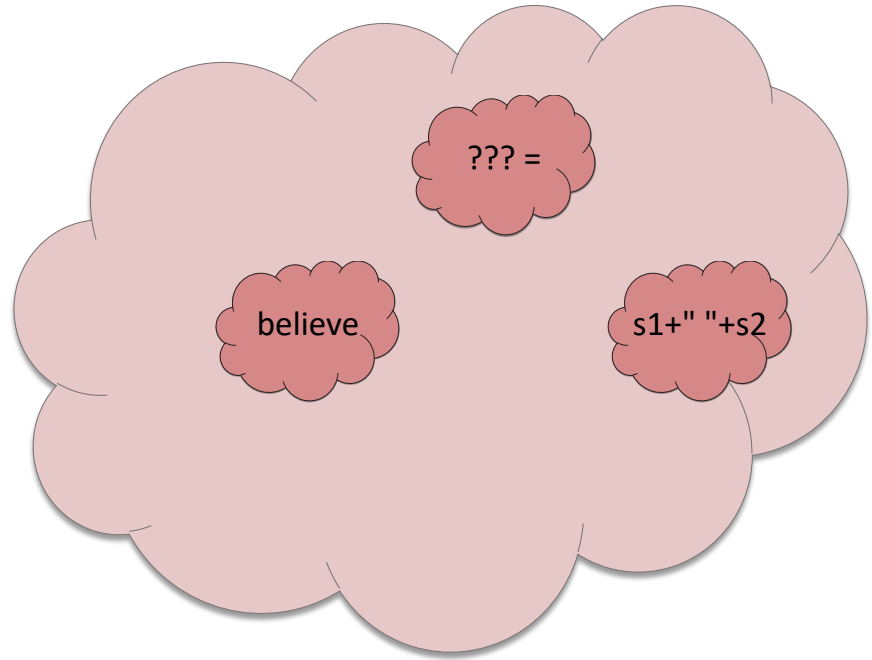
- bisher waren alle Schritte des Verständnisses nur lesend
- neu:
  - zu jeder Zeile den neuen Zustand speichern
  - und alten Zustand invalidieren
  - deutlich langsamer als Lesen

Mutability

```
var buffer = new StringBuilder(s1);
buffer.append("-");
buffer.append(s2);

var string = buffer.toString();
string.replace("-", " ");

return string.toLowerCase();
```



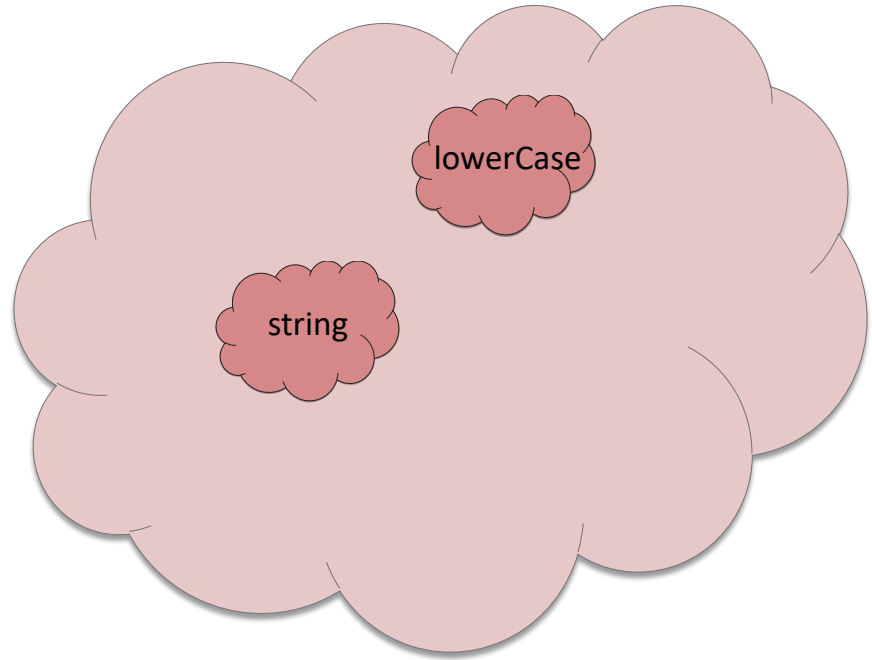
- bisher waren alle Schritte des Verständnisses nur lesend
- neu:
  - zu jeder Zeile den neuen Zustand speichern
  - und alten Zustand invalidieren
  - deutlich langsamer als Lesen

Mutability

```
var buffer = new StringBuilder(s1);
buffer.append("-");
buffer.append(s2);

var string = buffer.toString();
string.replace("-", " ");

return string.toLowerCase();
```



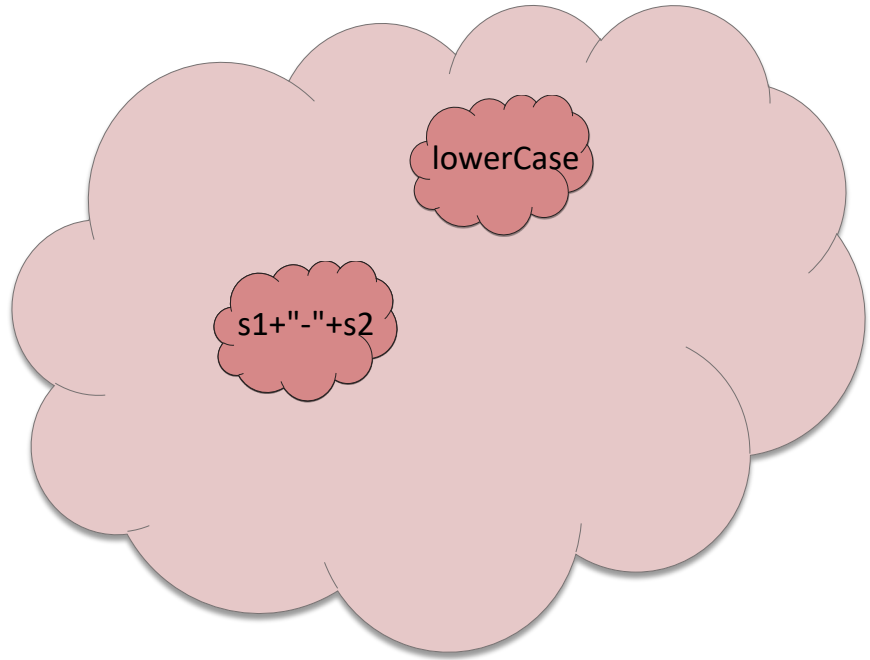
- bisher waren alle Schritte des Verständnisses nur lesend
- neu:
  - zu jeder Zeile den neuen Zustand speichern
  - und alten Zustand invalidieren
  - deutlich langsamer als Lesen

Mutability

```
var buffer = new StringBuilder(s1);
buffer.append("-");
buffer.append(s2);

var string = buffer.toString();
string.replace("-", " ");

return string.toLowerCase();
```



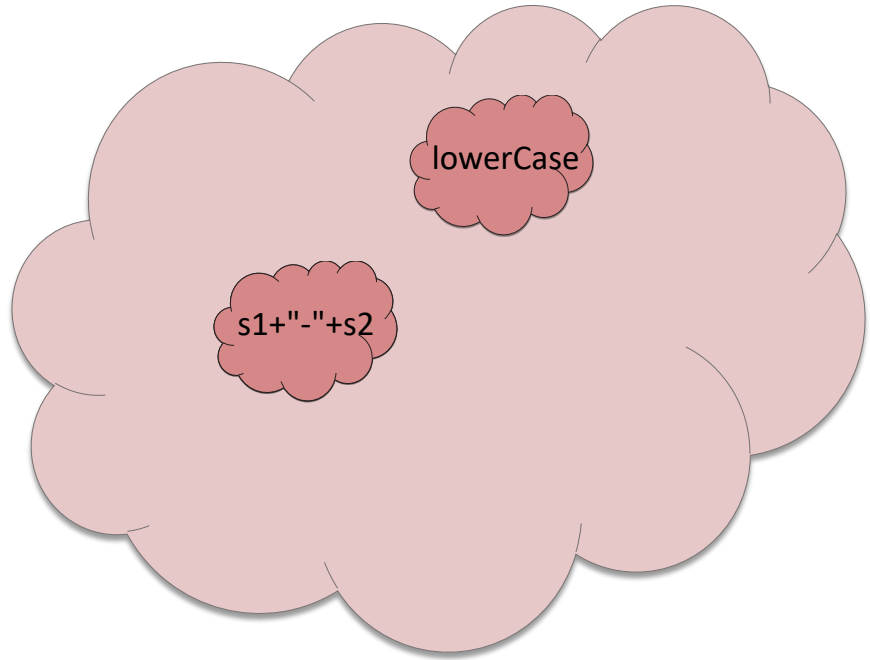
- bisher waren alle Schritte des Verständnisses nur lesend
- neu:
  - zu jeder Zeile den neuen Zustand speichern
  - und alten Zustand invalidieren
  - deutlich langsamer als Lesen

Mutability

```
var buffer = new StringBuilder(s1);
buffer.append("-");
buffer.append(s2);

var string = buffer.toString();
string.replace("-", " ");

return string.toLowerCase();
```



- bisher waren alle Schritte des Verständnisses nur lesend
- neu:
  - zu jeder Zeile den neuen Zustand speichern
  - und alten Zustand invalidieren
  - deutlich langsamer als Lesen

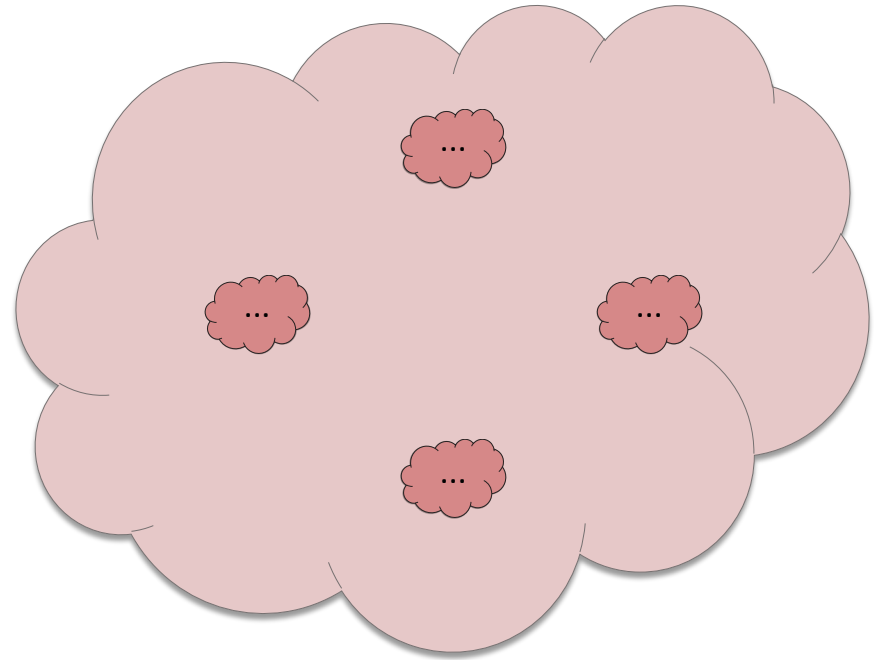
Clean Code

Command Query Separation

Aliasing

```
var a = new StringHolder("empty");
var b = a.clone();
var e = a;
var d = b;
var c = e;
var f = d;
var g = b;

g.set("Rust prevents this");
System.out.println(c);
```



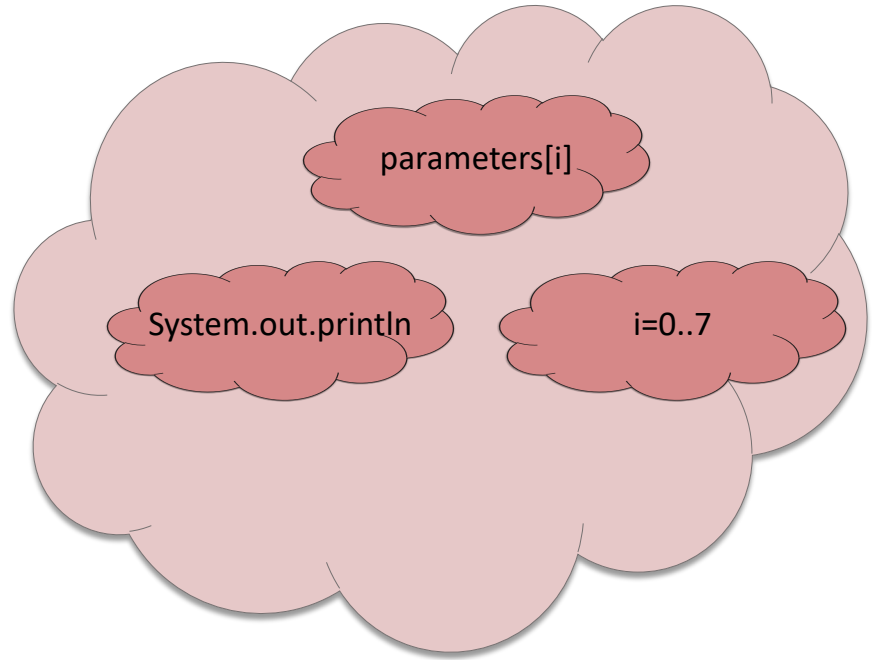
- mit jedem Alias werden Seiteneffekte komplexer
  - typischerweise entstehen Aliase durch Argumentübergaben
- Also entweder Immutable Shared Aliasing
  - jeder darf lesen, keiner schreiben
- oder Mutable Exclusive Aliasing
  - einer darf schreiben, keiner lesen

Mein Tipp

Keine Shared Mutable Aliase verwenden

Duplikation

```
System.out.println(parameters[0]);
System.out.println(parameters[1]);
System.out.println(parameters[2]);
System.out.println(parameters[3]);
System.out.println(parameters[4]);
System.out.println(parameters[5]);
System.out.println(parameters[6]);
System.out.println(parameters[7]);
```



- 8 Chunks? Unverständlich?
- nein, Chunks sind nicht Zeilen
- sondern Vorstellungen
- hier genügen 3 Chunks
- leider auch bei kleinen Fehlern

Hinweis

Chunks sind oft einfach zu erkennen, aber nicht immer

Code Konventionen

Typische Verstöße gegen Code-Konventionen

- unkonventionelle Blöcke/Scopes (z.B. if ohne {})
- unkonventionelle Identations
- Semikolons am Zeilenende

```
if (account1.balanceOk())
  account1.withdraw(transaction.value());
  account2.deposit(transaction.value());
```

Was ist das Problem?

- Konventionsgemäßen Code
  - liest das System 1 (schnell)
- Nicht konventionsgemäßer Code kann zu folgendem führen:
- System 1 liest über einen Fehler hinweg
  - und findet den Fehler nicht

Code Konventionen

Typische Verstöße gegen Code-Konventionen

- unkonventionelle Blöcke/Scopes (z.B. if ohne {})
- unkonventionelle Identations
- Semikolons am Zeilenende

```
if (!account1.balanceOk()) {return
  ;}
account1.withdraw(transaction.value());
account2.deposit(transaction.value());
```

Was ist das Problem?

- Konventionsgemäßen Code
  - liest das System 1 (schnell)
- Nicht konventionsgemäßer Code kann zu folgendem führen:
- System 1 liest über einen Fehler hinweg
  - und findet den Fehler nicht
- System 1 meldet Zweifel an, weil etwas unkonventionelles wahrgenommen wurde
  - und aktiviert System 2 (langsam)

Code Konventionen

Typische Verstöße gegen Code-Konventionen

- unkonventionelle Blöcke/Scopes (z.B. if ohne {})
- unkonventionelle Identations
- Semikolons am Zeilenende

```
if (!account1.balanceOk()) {return
  ;}
account1.withdraw(transaction.value());
account2.deposit(transaction.value());
```

Was ist das Problem?

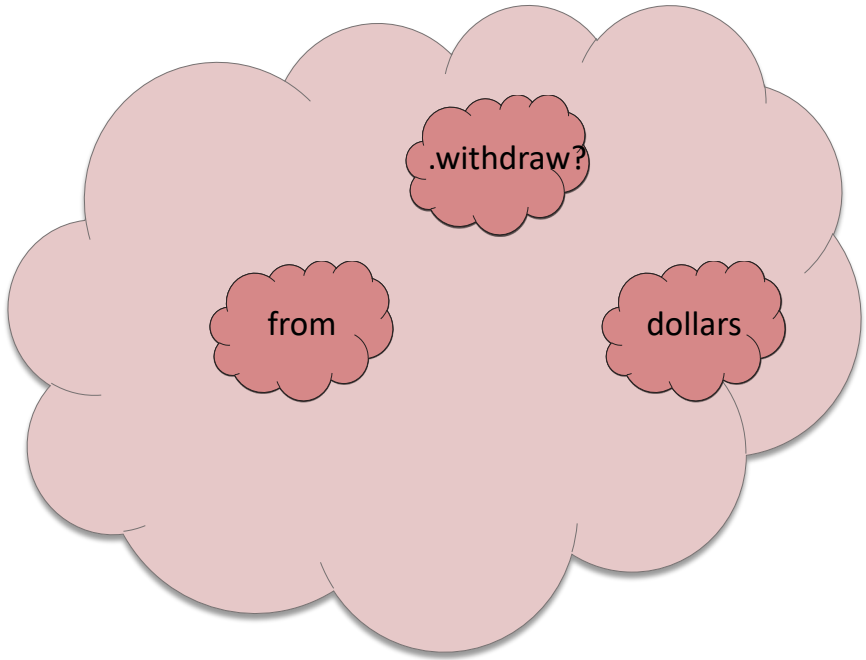
- Konventionsgemäßen Code
  - liest das System 1 (schnell)
- Nicht konventionsgemäßer Code kann zu folgendem führen:
- System 1 liest über einen Fehler hinweg
  - und findet den Fehler nicht
- System 1 meldet Zweifel an, we[illegible]nkonventionelles wahrgenommen wurde
  - und aktiviert System 2 (langsam)

Hinweis

Code mit Konventionen ist nicht nur ästhetischer, sondern auch verständlicher

Missverständnisse

```
void transferMoney(Acccount from, Acccount to, int dollars) {
    from.dec(dollars);
    to.inc(dollars);
}
```



- Das Problem aus der Systemsicht:
  - System 1 (schnell) liest den Code und hat keine Zweifel
  - deswegen wird System 2 nicht aktiviert zur Prüfung
- Aus der Gedächtnissicht sieht das so aus:
  - `inc` und `dec` haben verschiedene Bedeutungen (Chunks) in verschiedenen Kontexten
  - eine Bedeutung ist erhöhen/senken
  - Im Kontext `Account` macht das durchaus Sinn, es wird also genau der Chunk gewählt
- Das führt zu Fehlannahmen über die Funktionalität von `inc` und `dec`

Mein Tipp

Namen sollten im Kontext eindeutig sein

Konsequenzen - durch das Arbeitsgedächtnis

Unser Arbeitsgedächtnis

- arbeitet schrittweise
- höchstens 4 Chunks pro Schritt

Code

- muss so geschrieben werden, dass wir ihn schrittweise lesen können
- entweder in kleinen Paketen
- oder so, dass Heuristiken die richtigen kleinen Pakete erzeugen können

Konsequenzen - durch das Langzeitgedächtnis

Unser Langzeitgedächtnis

- arbeitet mit Chunks (Vorstellungen)
- Code-Konzepte sind uns verständlich, wenn bei uns der richtige Chunk angelegt ist
- Es ist also wichtig, dass Schreiber und Leser die gleichen Chunks begreifen

Das erreichen wir

- durch gemeinsamen Sprache
  - natürliche Sprache
  - Programmiersprache
  - Domänensprache
- den kritischen Blick der anderen
  - z.B. in Code Reviews, Pair Programming

Konsequenzen - durch den Informationsfluss

Information

- kann gelesen/abgeleitet werden (schnell)
- oder geschrieben/gemerkt werden (langsam)
  - insbesondere Seiteneffekte

Information

- kann schnell gelesen werden (System 1)
- oder kritisch gelesen werden (System 2)

Verständlicher Code

- verwendet wenige Seiteneffekten
- und ist so geschrieben, dass ein unkritischer Leser nichts Falsches versteht

Vielen Dank!

Theorie

- Cowan, Nelson (2001) The magical number 4 in short-term memory: A reconsideration of mental storage capacity
- Pinker, Steven (1998) How The Mind Works
- Kahneman, Daniel (2011) Thinking, Fast and Slow
- … zahlreiche Lehrbücher über kognitive Psychologie

Praktiken und Techniken:

- Clean Code (Robert C. Martin)
- Refactoring (Martin Fowler)
- The Art of Readable Code (Dustin Boswell)

Alternative Ansätze

- Programmer’s Brain (Felienne Hermans)