

Beispiele/Anleitung für VIM.

Grundlegendes

Nummern / Wiederholungen

Nahezu alle Befehle lassen sich mit vorheriger Nummerneingabe entsprechend oft wiederholen.

z.B.

l bewegt den Cursor im **Normalmodus** ein Zeichen nach rechts.

1000l bewegt den Cursor entsprechend 1000 Zeichen nach rechts.

In den Beispielen werde ich nützliche Verwendungen von Nummern für Wiederholungen mit aufführen.

Zeichen / Wörter Notationen

Manche Befehle benötigten Zeichen als Parameter, andere können Wörter annehmen. Folgende Auflistung zeigt die Varianten der Notationen, die in diesem Dokument verwendet werden.

Notation	Beschreibung	Beispiel
{t}	Text / Ein beliebiger Text	/ {t} → /Text
{n}	Number / Eine beliebige Zahl (egal wieviele Ziffern)	{n}l → 100l
{d}	Digit / Eine beliebige Ziffer (nur eine)	x{d} → x5
{c}	char / Ein beliebiges Zeichen	ct{c} → ct.
{L}	LETTER / Ein beliebiger Großbuchstabe	m{L} → mA
{l}	letter / Ein beliebiger Kleinbuchstabe	m{l} → ma
{Ll}	LETter / Ein beliebiger Buchstabe	m{Ll} → ma
{b}	Positiondefinition	c{b} → cf.
{r}	Register (a-z;A-Z;+;0-9;=;*)	"{r}yiw → "ayiw

Modis

VIM besitzt 4 primäre Modis:

- **Normalmodus**

In diesem Modus werden die meisten Befehle ausgeführt. Er dient hauptsächlich der

Cursorsteuerung.

VIM startet in diesem Modus und von jedem anderen Modus kann in der

Normalmodus mit **Esc** zurück gewechselt werden.

- **Einfügemodus** (INSERT)

In diesem Modus wird Text eingegeben. Es gibt wenige Tastenkombinationen für spezielle VIM Funktionen (z.B. Zugriff auf die Register). Ansonsten wird hier wirklich nur Text eingegeben.

Auch wenn es möglich ist der Cursor mit den Cursortasten zu bewegen (ggfs. sogar mit der Maus), sollte davon abgesehen werden.

Einfach Korrekturen am besten direkt mit **Backspace**, für größere wieder in den Normalmodus wechseln.

- **Visueller Modus** (VISUELL)

Dient um einen Bereich des Textes entweder zeichenweise, Zeilenweise oder Blockweise zu markieren und bietet Befehle zum kopieren, löschen, ändern, einfügen und einige andere.

- **Ex Modus**

Hier können komplexere Befehle mittels Namen oder Abkürzung eingegeben werden. Damit kann man grundsätzlich alles machen was VIM kann, jedoch wäre es nicht sehr effizient damit einfach nur den Cursor zu bewegen.

Dafür können hier Makros gestartet, Tastaturbelegung geändert, Einstellungen vorgenommen werden. Auch öffnen, speichern, schließen von Dateien oder neue Tabs, neue Fenster, neue Buffer werde hier erstellt.

Alle Befehle: [VIM Ex Befehle](#)

Es gibt noch die Suche ist, wobei dies eher kein Modus ist sondern einfach nur die Suche.

Ex-Modus

Man gelangt in den **Ex-Modus** mit **:**, Befehle vom **Ex-Modus** werde ich IMMER mit **:** vorangestellt darstellen.

Wichtige Befehle für den Ex:

Esc

Zurück in den **Normalmodus**

:w

Speichern von der aktuellen Datei (optional auch mit Dateiname z.B. **:w readme.txt**)

:q

Beenden von VIM. Wurde die Datei geändert, verweigert VIM das Beenden. Hier muss das Beenden mit **:q!** erzwungen werden.

:wq

Kombination aus Speichern und beenden, erzwingen muss man hier nie da ja immer vorher gespeichert wurde.

:qa

Alle Tabs, Fenster und Buffer schließen (erzwingen mit **:qa!**)

:wa

Alle Tabs speichern

:wqa

Alle Tabs speichern und schließen

Später noch mehr Anwendungsbeispiele und Befehle für den Ex-Modus

Navigation

VIM bietet einzigartige Möglichkeiten den Cursor an die gewünschte Stelle im Dokument zu bewegen. Die Maus ist komplett unnötig und wird nie benötigt bzw. sollte sogar nie verwendet werden.

Einfache Navigation

Num: Nummerneingabe für Wiederholungen möglich

Kürzel/Befehl	Beschreibung	Num
hjkl	Die wichtigste und nützlichste Bewegung des Cursors nach links (h), runter (j), hoch (k) und rechts (l).	ja
w, e	Springt zum nächsten Wort (an Anfang mit w ; ans Ende mit e)	ja
b	Springt zum vorherigen Wort (an Anfang)	ja
0, ^	Springt zum Zeilenanfang (ganz links mit 0 , erstes Zeichen von links mit ^)	nein
\$	Springt zum Zeilenende	nein
gg	Springt zum Dokumentenanfang	nein
G	Springt zum Dokumentenende	nein
H, M, L	Bewegt den Cursor nach oben (H igh), in die Mitte (M iddle) oder nach unten (L ow) Der Text selber wird nicht verändert	nein
%	Bewegt den Cursor zur passenden geöffneten bzw. geschlossenen Klammer	nein

Komplexere Navigation

Num: Nummerneingabe für Wiederholungen möglich

Kürzel/Befehl	Beschreibung	Num
f{c}, F{c}	Zeichensuche ZUM Zeichen - Vorwärts (f) und Rückwärts (F) z.B. f. springt zum nächsten .	Ja
t{c}, T{c}	Zeichensuche VOR Zeichen - Vorwärts (t) und Rückwärts (T) z.B. t. springt ein Zeichen vor dem nächsten .	Ja
;, ,	Zeichensuche wiederholen (;) bzw. zurück zum vorherigen gefundenen Zeichen (,)	Ja
/t}, ?{t}	Textsuchen im Dokument Vorwärts (/) und Rückwärts (?) z.B. /Wort	Nein
*, #	Text unter Cursor suchen (Vorwärts *) und Rückwärts (#)	nein
n, N	Testsuche wiederholen (n bzw. zurück zum vorherigen gefundenen Text (N)	Nein
[{,]}	Zur vorherigen {-Blockanfang ({}) bzw. zum nächsten {-Blockende (})	Ja
}, {	Zum nächsten Absatz (}) bzw. zum vorherigen Absatz ({)	Ja

Passende Ex-Befehle

:noh : Löscht die Markierung von gefundenem Text (für / und ?)

Scrollen und Seitenweise Navigation

Num: Nummerneingabe für Wiederholungen möglich

Kürzel/Befehl	Beschreibung	Num
Strg+E, Strg+Y	Eine Zeile hoch (E) bzw. runter (Y) scrollen	Ja
Strg+D, Strg+U	Eine halbe Seite nach unten (D) bzw. nach oben (U) scrollen Mit einer [[VIM Anleitung#ö / ä für halbe Seite hoch/runter	VIM Konfiguration]] kann auch einfach ö und ä verwendet werden.
Strg+F, Strg+B	Eine ganze Seite nach unten (F) bzw. nach oben (B) scrollen	Ja
zt, zz, zb	Bewegt die aktuelle Zeile nach oben (zt top), in die Mitte (zz) und nach unten (zb bottom) Kann auch mit vorangestellter Zahl (z.B. 10zz) eingeben werden. Dann befindet sich die angegebene Zeile (z.B. 10) oben/mitte/unten	nein
{n}G / :{n}	Springt zur Zeile {n}	nein

Positionsmarkierungen

Es gibt 2 Möglichkeiten Positionen zu markieren und 2 Möglichkeiten diese Positionen anzuspringen.

Lokale Positionsmarkierungen

`m{L}` : Markiert die aktuelle Position des Cursors

Globale Positionsmarkierungen

`m{L}` : Markiert die aktuelle Position des Cursor in Verbindung mit der geöffneten Datei.
Im Gegensatz zur lokalen Positionsmarkierung wird hier einfach nur ein Großbuchstabe eingegeben.

Zeile der Position anspringen

`'{L}` / `'{L}` : Der Cursor wird an den Zeilenanfang der markierten Position bewegt

Exakte Position anspringen

``{L}` / ``{L}` : Der Cursor wird an die exakte Position der Markierung bewegt (``` m sind Backticks!)

Werden Globale Positionsmarkierungen angesprungen wird die Datei, falls noch nicht geöffnet, automatisch geladen.

Text eingeben, ersetzen, löschen und einfügen

Neben der Cursor-Steuerung ist natürlich die Veränderung von Text eine grundlegende Eigenschaft von Texteditoren.

Grundlegende Eingabe

Num: Nummerneingabe für Wiederholungen möglich (hat meist erst den Effekt, wenn man in den Normal-Modus zurück wechselt)

Kürzel/Befehl	Beschreibung	Num
<code>i</code>	Einfügemodus an Cursorposition	Ja
<code>a</code>	Einfügemodus nach dem Cursor	Ja
<code>I</code>	Einfügemodus am Zeilenanfang	Ja
<code>A</code>	Einfügemodus am Zeilenende	Ja
<code>o</code>	Einfügemodus und Zeile unterhalb einfügen	Ja
<code>O</code>	Einfügemodus und Zeile oberhalb einfügen	Ja
<code>s</code>	Einfügemodus und ein Zeichen ersetzen	Ja
<code>{n}s</code>	Einfügemodus und <code>{n}</code> Zeichen ersetzen	Ja

Kürzel/Befehl	Beschreibung	Num
S	Einfügemodus und komplette Zeile ersetzen	Ja
C	Einfügemodus und restliche Zeile ab Cursorposition löschen	Ja
D	Löscht die restliche Zeile ab dem Cursor	Nein
J	Fügt die untere Zeile zur aktuellen Zeile hinzu	Ja
gJ	Wie J nur wir die Zeile nicht getrimmt (Leerzeichen bleiben)	
r{c}	Ein Zeichen ersetzen	Ja
{n}r{c}	{n} Zeichen ersetzen	Nein
R	Überschreibmodus an Cursorposition	Ja
X	Löscht das Zeichen unter dem Cursor (entspricht Windows Entf Taste)	Ja
X	Löscht das Zeichen links neben dem Cursor (entspricht Backspace)	Ja

Komplexere Texteingabe / -Ersetzung

Num: Nummerneingabe für Wiederholungen möglich (hat meist erst den Effekt, wenn man in den Normal-Modus zurück wechselt)

Kürzel/Befehl	Beschreibung	Num
c{b}	Change - Mit Positionsdefinition definierten Bereich ändern (Löschen und Einfügemodus)	Nein

z.B:

Kürzel/Befehl	Beschreibung
ciw	Change in Word - Aktuelles Wort ersetzen (löschen & Einfügemodus)
ct.	Change to . - Bis zum nächsten Punkt ersetzen
ci"	Change in " - Alle zwischen 2 Anführungszeichen ersetzen
ciC	Change in (- Alles zwischen ()-Klammern (geht auch mit { , [)

Weitere [Positionsdefinitionen](#)

Positiondefinition

Werden in Beispiel/Syntax bei **{b}** eingesetzt.

Manche Befehle erlaubt eine Positionsdefinition. Dies sind eigene Befehle die nur dafür benutzt werden können in Kombination mit anderen eine bestimmte Stelle oder Bereich zu definieren.

Diese können z.B. mit **d** (Delete), **v** (Markieren), **y** (Kopieren), **c** (Ändern) verwendet werden.

Am besten testen kann man diese mit **v**, der einen Bereich Zeichenweise markiert.

Es gibt folgende Positionsdefinitionen:

Kürzel/Befehl	Beschreibung	Beispiel
i	IN - Innerhalb der angegebenen Zeichen (s.u.) z.B. vi" markiert alles innerhalb 2 Anführungszeichen.	a " b " c
a	AUSSEN - Ausserhalb der angegeben Zeichen (s.u.), also inkl. dieser. z.B. va" markiert alles innerhalb 2 Anführungszeichen INKLUSIVE der Anführungszeichen	a "b" c
t	TO - Bis VOR dem angegeben Zeichen, diese Zeichen kann jedes Zeichen sein, z.B. vt. markiert alles von der aktuellen Cursorposition bis VOR dem Punkt (.).	a <Cursor. . c
T	TO - Rückwärts	a. Cursor> c
f	FIND - Bis ZUM angegebenen Zeichen, wie t nur inklusive dem angegeben Zeichen, z.B. vf. markiert alles von der aktuelle Cursorposition bis ZUM Punkt (.)	a <Cursor. . c
F	FIND - Rückwärts	

i und **a** erlaubt folgende Zeichen als Bereichsauswahl:

Zeichen	Beschreibung	Beispiel mit vi{c} bzw. va{c}
"	Innerhalb 2 Anführungszeichen	Text " markiert " Text Text "markiert" Text
q	Wie " wird aber nicht von allen Editoren unterstützt 1	Text " markiert " Text Text "markiert" Text
'	Innerhalb 2 einfachen Anführungszeichen	Text ' markiert ' Text Text 'markiert' Text
(Innerhalb von Klammern	Text (markiert) Text Text (markiert) Text
[Innerhalb von Eckklammern	Text [markiert] Text Text [markiert] Text

Zeichen	Beschreibung	Beispiel mit <code>vi{c}</code> bzw. <code>va{c}</code>
{	Innerhalb von geschweiften Klammern	Text { markiert } Text Text {markiert} Text
t	Innerhalb von HTML/XML Tags	Text <Tag> markiert </Tag> Test Text <Tag>markiert</Tag> Text
w	Wort - Das aktuelle Wort (<code>vaw</code> inkl. <code>einem</code> Leerzeichen)	Text markiert Text Text ==markiert ==Text

Visual Studio Code unterstützt diese Abkürzung. NeoVIM/VIM jedoch nicht. Mit folgenden Ex-Befehlen können diese nachträglich erstellt werden:

SHELL

```
noremap ciq ci"
noremap viq vi"
noremap yiq yi"
noremap diq di"
```

Natürlich können diese auch in die [VIM Konfiguration](#) eingetragen werden, damit diese immer zur Verfügung stehen.

Zusätzlich können für Positionsangaben alle Navigationsbefehle genutzt werden:

Beispiel	Beschreibung	Ergebnis
<code>v10l</code>	Es werden 10 Zeichen nach rechts markiert	Text markiert.. Text
<code>V5j</code>	Es werden 10 Zeilen markiert	
<code>v5w</code>	Es werden 10 Wörter markiert	Ich schreibe einen Text mit mehreren Wörtern.
<code>v/Wör</code>	Es wird vom Cursor bis zum gesuchten Text (<code>Wör</code>) markiert	Ich schreibe einen Text mit mehreren Wörtern.

Undo / Redo

Undo erfolgt im Normalmodus mit `u`, Redo mit `Strg+r`.

Natürlich kann dieses auch [in der Konfiguration remapped](#) werden.

Nun bietet auch VIM selbst beim Undo/Redo mehr Möglichkeiten als ein normaler Texteditor.

Bei einem Texteditor oder Textverarbeitung hat man keinen Einfluss darauf, was mit einem Undo-Schritt Rückgängig gemacht werden soll. Es kann ein Zeichen sein, ein Wort, eine Zeile, ein Absatz... Entweder wird zu wenig Rückgängig gemacht und man muss 100mal Strg+Z drücken. Oder Zu viel und man braucht Redos und muss dann trotzdem viel von Hand korrigieren.

In VIM ist ein Undo-Schritt das, was man im Insert-Modus gemacht hat. Verlässt man diesen **esc**) und betritt ihn wieder (**i** oder **c**,...) dann legt man so fest, was ein Undo-Schritt ist.

Daher ist es sinnvoll (aber auch schwer sich anzugewöhnen) an wichtigen Stellen kurz **<esc>i** zu tippen. Damit legt man eine neue Undo-Position fest.

Markieren dann löschen, ändern und kopieren

Beendet kann der Visuell-Modus ebenfalls mit der **Esc** Taste.

Die Markierung von Text (Wechsel in den Visuell-Modus) erfolgt mit folgenden Kürzel:

Kürzel	Beschreibung
v	Zeichenweise Markierung
V	Zeilenweise Markierung
Strg+V	Blockweise Markierung

Im Visuell-Modus sind die [Navigation](#) Kürzel/Befehle möglich um die Markierung zu erweitern. Es gibt jedoch weitere Kürzel, nur im Visuell-Modus funktionieren.

Kürzel	Beschreibung
	Cursor an Anfang / Ende der Markierung bewegen
o	Kann genutzt werden wenn der Anfang oder das Ende der Markierung verändert werden soll.

Die meisten Kürzel / Befehle haben im Visuell-Modus keine Kombinationen. Wie z.B. **c**, hier geht kein **c{b}** wie **ct**. sondern **c** löscht den markierten Text und wechselt in den Einfüge-Modus. Hier ein Überblick über die Kürzel/Befehle die anderes funktionieren als im Normal-Modus.

Kürzel	Beschreibung
c	Change - Löscht den markierten Text und wechselt in den Einfüge-Modus

Kürzel	Beschreibung
d	Delete - Löscht den markierten Text und wechselt in den Normal-Modus
y	Yank/Copy - Kopiert den markierten Text und wechselt in den Normal-Modus
v, V, Strg+V	Wechselt die Art (Zeichenweise, Zeilenweise, Blockweise) der aktuellen Markierung Erneutes drücken der zuvor gedrückten Taste beendet den Visuellen-Modus; eine andere Taste wechselt die Art
>, <	Rückt alle markierten Zeilen nach rechts (>) bzw. links (<) ein.
~	Wechselt alle markierten Buchstaben von Groß ↔ Kleinbuchstaben
u, U	Wechselt alle markierten Buchstaben in Kleinbuchstaben (u) oder Großbuchstaben (U)
r{c}	Ändert alle markierten Zeichen in das angegebene Zeichen
p	Ersetzt den markierten Bereich durch den kopierten Text

Würde man ein Wort z.B. mit **viw** markieren und dann **c** drücken, kommt das gleiche Ergebnis raus als würde man einfach **ciw** eingeben.

Ändern, Löschen und Kopieren

Erfolgt zum einen genauso wie man Text markieren kann. Nur mit anderen Kürzeln:

Markieren	Ändern	Löschen	Kopieren
v{b}	c{b}	d{b}	y{b}
Bei vorhandener Markierung	c	d	y

Somit ist **vi"c** (alles zwischen "-Zeichen markieren und ändern) das gleiche wie **ci"**.

Modus	Beschreibung
Ändern	Text wird gelöscht und es erfolgt der Wechsel in den Einfügemodus
Löschen	Text wird nur gelöscht, man bleibt im Normalmodus
Kopieren	Text wird nicht gelöscht (nur die Markierung) und man bleibt im Normalmodus

Mit **yy**, **dd** wird immer die komplette Zeile kopiert bzw. gelöscht. Ein **cc** macht kein Sinn, da es dafür ja **S** (komplette Zeile ändern) gibt.

Einfügen

Kopierter und gelöschter Text (egal ob mit **c** oder **d**) kann mit **p** bzw. **P** wieder eingefügt werden.

Um a) einen Text zu kopieren und b) durch einen anderen zu ersetzen, wird erst der Text kopiert (z.B. **yi "**) anschließend wird der Text der ersetzt werden soll markiert (z.B. **viw**) um dann mit **p** diesen zu ersetzen. ^3f25bc

Ich eine in schritte.

Register

Sind eine sehr nützliche Funktionalität von VIM. Grob kann man sagen, da es in VIM nicht **eine** Zwischenablage sondern **28!**

Denn Texte werden bei VIM nicht in die Zwischenablage des Betriebssystems kopiert, sondern in sogenannte Register.

Diese werden im Normalmodus mit "**{r}**" angesprochen und VOR den Befehlen, die etwas kopieren (**y**, **c**, **d**) oder einfügen (**p**, **P**) eingegeben.

Wenn **diw** ein Wort löscht & kopiert, wird es mit "**adiw**" ebenfalls gelöscht, jedoch in das Register **a** kopiert. Gleiches gilt für **y** (kopieren) und **c** (ändern).

Es gibt folgende Register:

Register Zeichen	Beschreibung
"	Unbenanntes Register
a-z	Zum wahlweises kopieren/einfügen
A-Z	Text wird im Register beim kopieren nicht ersetzt sondern erweitert
0	Nur kopierter Text (y)
1-9	Zuletzt kopierte Texte (Kopier-Verlauf)
+ / *	Betriebssystem Zwischenablage oder bei Linux ein Auswahl-Register der Zwischenablage
=	Ausdruckregister
_	Schwarzes Loch Register

Das Unbenannte Register (**"**) ist das Register, in dem Text kopiert wird, wenn kein Register angegeben ist (oder wenn Text einfach mit **p** oder **P** eingefügt wird.)

Der Unterschied zwischen Großbuchstaben und Kleinbuchstaben ist, das bei Kleinbuchstaben der vorherige Inhalt immer gelöscht wird (Kopiere "A" → Inhalt "A", Kopiere "B" → Inhalt "B", Kopiere "C" → Inhalt "C").

Bei Großbuchstaben wird das Register erweitert (Kopiere "A" → Inhalt "A", Kopiere "B" → Inhalt "AB", Kopiere "C" → Inhalt "ABC").

Beim Register **0** wird der Inhalt **nur** bei Verwendung von **y** (kopieren) verändert. Beim löschen (**d**) oder ändern (**c**) bleibt dieses Register unverändert.

Also kann man einen Text auch ohne vorherigem Markieren (siehe [Text durch anderen ersetzen](#)): Text kopieren (**yi"**), zum zu ersetzenden Text gehen und löschen (**di"**) und dann mit **"0p** das Register **0** einfügen.

Die Zwischenablage kann mit **+** angesprochen werden. Und zwar schreibend (mit **y**, **d** und **c**) als auch lesen für das Einfügen (**p** / **P**):

"+p fügt die Zwischenablage ein

"+yi" Kopiert den Text zwischen zwei " in die Zwischenablage.

Unter Linux gibt es noch das Auswahl-Register. Bei Interesse am besten danach googeln.

Mit dem Ausdruck-Register kann eine Rechnung eingegeben werden, dessen Ergebnis kann danach mit **p** eingefügt werden.

z.B. **"=5+10<enter>p** Berechnet 5+10 und fügt mit **p** dann **15** ein.

Das Schwarze Lock Register (**_**) sorgt dafür, dass der gelöschte Text nicht kopiert wird.

Nur sinnvoll bei **c** oder **d**, da man mit **y** nur dafür sorgt, dass doch nicht kopiert wird (dann könnte man es auch gleich sein lassen).

Register im Einfügemodus einfügen

Mit **Strg+R {r}** kann im Einfügemodus direkt ein Register eingefügt werden. z.B. statt mit **<esc>** in den Normalmodus zu wechseln um dann mit **"ap** das **a** Register einzufügen um dann mit **i** wieder in den Einfügemodus zu wechseln, kann einfach nur **<Strg+R>a** gedrückt werden.

Beim Ausdrucksregister wird mit **<Strg+R>=10+5** direkt das Ergebnis (15) eingefügt.

Auch kann man damit auf VIM-Variablen zugreifen **<Strg+R>z** fügt den Wert der Variable **z** ein.

Nützliche Verwendung von Registern

1. Erst alles kopieren, dann einfügen

Am nützlichsten sind die Register, wenn man erst alles was man braucht kopiert (z.B. in Register a, b, c, d...). Und dann an den benötigten Stellen einfügt.

2. Oft benötigtes jederzeit verfügbar

Quellcode oder Texte, die oft benötigt werden, in passende Register kopieren (z.B.

die AddUser-Methode in Register a, CopyUser in Register c usw.).
Dann kann man jederzeit deren Inhalte einfügen.

Was gibt es bei Register noch zu beachten?

Register werden für mehrere Dinge in VIM benötigt. Natürlich zum kopieren/einfügen von Texten.

Zusätzlich werden auch **Makros** in Register gespeichert. Man verliert damit auch in kopierten Text (oder mit kopieren das erstellte Makro).

Makros

Die Verwendung von Makros erlaubt komplexe Abläufe beliebig oft zu wiederholen.

Makros werden aufgezeichnet mit `q{l}` also `q` und ein Kleinbuchstabe.
Sie werden in den Register a-z gespeichert.

Nach der Aufnahme wird alles was man macht gespeichert.

Die Aufnahme beendet man mit erneutem `q`.

Beispiel: `qaiHello<esc>o`

Erstellt ein Makro auf Register `a`, das in den Einfügemodus wechselt, `Hello` schreibt und eine neue Zeile einfügt.

Zu beachten

Jegliche Makros sollten so aufgenommen werden, das die Position des Cursors beim Start des Makros **KEINE ROLLE** spielt. Ob am Anfang oder am Ende der Zeile, oder mitten drin.

So sollte man am Anfang des Makros mit `0` an den Zeilenanfang wechseln, dann spielt es für den restlichen Ablauf keine Rolle wo der Cursor war.

Makros ausführen

Das Ausführen von Makros ist umfangreicher als die Aufnahme.

Einfaches ausführen

Mit `@{l}` z.B. `@a`

Damit wird das Makro einmalig ausgeführt.

Nochmaliges ausführen

Mit @@ wird das zuletzt ausgeführte Makro nochmal ausgeführt.

Mehrmaliges ausführen

Mit {n}@{l} kann man das Makro auf Register l n mal ausführen z.B. 10@a

Unterschiedliche Ausführungsarten (Mehrfach / Je Zeile)

Ein Makro kann wie oben beschrieben x-Mal ausgeführt werden. Jedoch wird ein Makro beendet, wenn ein Fehler auftritt.

```
objecta.read();
objectb.calc();
objectc.write();
```

Mit @a@f.i.do<esc>j Wird ein Makro aufgenommen das

- An den Zeilenanfang geht (0)
- Ein . sucht (f.)
- In den Einfügemodus wechselt (i)
- .do schreibt
- In den Normalmodus wechselt <esc>
- Eine Zeile nach unten geht (j)

Nun kann das Makro 2 mal ausgeführt werden und der Text lautet dann:

```
objecta.do.read();
objectb.do.calc();
objectc.do.write();
```

Führt man das Makro 5 mal bei folgenden Text aus:

```
objecta.read();
// Verarbeiten
objectb.calc();
// Ausgeben
objectc.write();
```

Wird **nur** die erste Zeile korrekt ersetzt. Warum?

Das Makro wird das erste mal ausgeführt → funktioniert

Das Makro wird das zweite mal ausgeführt → es wird kein **.** gefunden und die Ausführung wird beendet.

Zeilenweises ausführen

Die Lösung ist das Zeilenweise ausführen. Hier ist nicht notwendig im Makro die Zeile zu verlassen (d.h. das **j** am Ende kann weg gelassen werden).

Nun die Zeilen markieren für den das Makro ausgeführt werden soll.

Nun kann man nicht einfach **@a** eingeben, weil dies nicht funktioniert, wenn etwas markiert ist.

Jedoch erlaubt der Ex-Modus einen Befehl für einen Bereich einzugeben:

:norm @a

norm : Führe für jede Zeile den Normal-Befehl (Kürzel) aus

@a : Ist der Kürzel der für jede Zeile ausgeführt werden soll.

Nun werden alle Zeilen iteriert und für jede Zeile das Makro **a** ausgeführt.

Wird ein **.** nicht gefunden, wird die Ausführung **nur** für diese Zeile beendet. Für die anderen Zeilen geht es normal weiter.

Das Ergebnis ist dann:

```
objecta.do.read();
.do// Verarbeiten
objectb.do.calc();
.do// Ausgeben
objectc.do.write();
```

WICHTIG Bei manchen VIM-Erweiterungen oder Programme die VIM unterstützen (z.B. Obsidian) führt ein nicht finden von **.** mit **f.** nicht zu einem Fehler und nicht zum Abbruch des Makros. Dafür bleibt der Cursor dort, wo er war (links, durch das **0**). Und dann kommen folgender Text heraus:

```
objecta.do.read();
.do// Verarbeiten
objectb.do.calc();
.do// Ausgeben
objectc.do.write();
```

Weitere Kürzel

Num: Nummerneingabe für Wiederholungen möglich

Kürzel/Befehl	Beschreibung	Num
<, >	Im Visuel-Modus (Markierung) markierte Zeilen nach links/rechts einrücken.	Ja
<<, >>	Die Aktuelle Zeile wird nach links/rechts eingerückt	Ja
<% , %>	Alle innerhalb der Klammer einfügen	Ja
<{b}, >{b}	Alle Zeilen innerhalb der Positiondefinition , s.u.	Ja
J / gJ	Aktuelle Zeile mit der darunter liegenden kombinieren (gJ entfernt keine unnötigen Leerzeichen am Anfang der nächsten Zeile)	Ja

Beispiel für **>i{** (Cursor irgendwo bei Zeile 1, 2 oder 3):

```
if (a)
{
// Zeile 1
// Zeile 2
// Zeile 3
}
```

Ergibt

```
if (a)
{
    // Zeile 1
    // Zeile 2
    // Zeile 3
}
```

Groß, Kleinschreibung umschalten g~, gu, gU

Oft muss man einfach nur ein oder mehrere Zeichen in Groß- oder Kleinbuchstaben umwandeln. Auch hierfür bietet VIM entsprechende Kürzel.

Num: Nummerneingabe für Wiederholungen möglich

Kürzel/Befehl	Beschreibung	Num
g~{b}	Wie Markierungen (v) oder c/d/y - Nur wird für die angegebenen Positiondefinition alle Zeichen in Groß-/Kleinbuchstaben gewechselt	
gu{b}	Wie oben, nur werden alle Zeichen in kleinbuchstaben umgewandelt	
gU{b}	Wie oben, nur werden alle Zeichen in GROßBUCHSTABEN umgewandelt	

Ist eine Markierung vorhanden, benötigt es nur **g~**, **gu** oder **gU**. z.B.

gu5l : 5 Zeichen nach rechts wechselt

gUiw: Aktuelles Wort in Großbuchstaben wechselt

V3jgu : Markiert die Aktuelle Zeile + 3 Zeilen nach unten und wechselt alles in Kleinbuchstaben.

vgU : Wechselt das aktuelle Zeichen (Markierung mit **v** ohne Bewegung ist nur ein Zeichen) in Großbuchstaben

guu: Wechselt alle Zeichen der aktuellen Zeile in Kleinbuchstaben

Natürlich kann ein Zeichen auch einfach mit **r{c}** ersetzt werden. z.B. ein kleines **r** in ein großes **R** wenn der Cursor auf dem kleinen **r** steht: **rR**

Einmaliger Normal-Modus

Um im Einfüge-Modus einmalig einen Normal-Befehl/Kürzel auszuführen kann **Strg+0** gedrückt werden. Nach diesem Befehl befindet man sich sofort wieder im Einfügemodus.

z.B. **Strg+0 dd** löscht einfach die aktuelle Zeile und man kann weiter schreiben.

Wobei manchmal sogar ein **Esc** schnell ist: **Esc S** wechselt in den Normalmodus, löscht die aktuelle Zeile und wechselt in den Einfügemodus.

Es gibt aber sicher Anwendungsfälle in denen **Strg+0** nützlich ist.

Suchen / Ersetzen

Das Suchen und Ersetzen in VIM erlaubt das einfache Suchen oder Ersetzen von Texten bis hin zu komplexen Regular Expression für die Textsuche.

Regular Expression (Reguläre Ausdrücke; nachfolgen RegEx genannt) sind Beschreibung für den Aufbau und Inhalt von Texten. Eine Art Prüfungs- und Suchmuster. Sie bieten extrem viele Möglichkeiten um Informationen aus Texten/Strings zu finden, heraus zu filtern oder zu ersetzen. Jede aktuell verwendete Sprache unterstützt RegEx, als auch Terminals wie PowerShell. Mein Tipp: Als Entwickler sollte man diese Technik definitiv, zumindest in den Grundlagen, beherrschen.

Die Suche wurde bereits erwähnt, der Vollständigkeitshalber beschreibe ich sie hier nochmal (dafür genauer).

Einfache Textsuche

Vorwärts mit `/<Text>` Rückwärts mit `?<Text>`. z.B. `/RegEx`
Je nach Modus kann auch ein RegEx eingegeben werden.

Die Suche startet bereits bei Eingabe des Suchtextes.

Modi und Einstellungen

Einstellungen werden im Ex-Modus vorgenommen und können in der [VIM Konfiguration](#) angegeben werden, damit sie dauerhaft gültig sind.

Aktiviert werden Einstellungen mit `:set <einstellungname>` z.B.: `set ignorecase`.
Deaktiviert werden sie, in dem man an `no` vor dem Einstellungsname setzt, z.B. `set noignorecase`

Folgende Einstellungen sind wichtig:

`ignorecase / noignorecase`

Ist `ignorecase` aktiviert, wird nicht zwischen Groß- und Kleinschreibung unterschieden.
`/regex` findet auch `RegEx`.

`smartcase / nosmartcase`

Bei aktiviertem `ignorecase` und `smartcase` wird automatisch die Groß- und Kleinschreibung berücksichtigt, wenn Großbuchstaben im Suchtext vorhanden sind (dann müssen die Vorkommen im Text dem Suchtext 100% übereinstimmen).

`magic / nomagic`

Dies steht im Zusammenhang mit RegEx. Bei Magic können Zeichen für RegEx direkt in der Suche verwendet werden: `/[C]` sucht eine `C`-Klammer

Bei **nomagic** sind Zeichen für RegEx einfach nur Suchtext. `/[C]` sucht `[C]`

Je nachdem ob man eher mit RegEx arbeitet ist es **magic** Modus sinnvoll. Kennt oder nutzt man RegEx nicht, macht der **nomagic** Modus sinn, da man dann einfacher nach bestimmten Zeichen suchen kann.

Modis während der Suche setzen

Ist z.B. **noignorecase** und **magic** aktiviert kann man einmalig in der Suche **ignorecase** und **nomagic** aktivieren.

Dies geschieht mit `\<Moduszeichen><Suchtext>`

Wobei der Modus irgendwo nach dem `/` oder `?` stehen darf (egal ob am Anfang, irgendwo im Text oder am Ende). Somit bringt dies alles das gleiche Ergebnis:

`abc\c`, `a\cbc`, `\cabc`

Beispiele sind für folgenden Text: **BeiSPIEL [(ABC)] abc [X]**

Moduszeichen	Beschreibung	Beispiele
<code>\C</code>	noignorecase Groß- und Kleinschreibung wird unterschieden	<code>/abc\C</code> findet nur das rechte <code>abc</code>
<code>\c</code>	ignorecase Groß- und Kleinschreibung wird ignoriert	<code>abc\c</code> findet <code>ABC</code> und <code>abc</code>
<code>\v</code>	very magic (keine Einstellung) RegEx-Format wird verwendet	<code>no magic</code> aktiv <code>/[X]</code> findet <code>[X]</code> <code>\v[X]</code> findet <code>X</code>
<code>\V</code>	very nomagic (keine Einstellung) Suchtext wird gesucht	<code>magic</code> aktiv <code>/[X]</code> findet <code>X</code> <code>\V[X]</code> findet <code>[X]</code>

Zusätzlich gibt es noch `\m` (**magic**) und `\M` (**nomagic**).

Um mittels RegEx Zeichen zu suchen müssen sie entweder in `[]` Klammern angegeben werden oder mittels Escape-Sequenz maskiert werden z.B. `[]` suchen mit `\[]`

So sucht `[X]` bei RegEx nur das `X`-Zeichen wobei `\[X]` das `[]` Zeichen maskiert und somit `[X]` sucht.

Meine Empfehlungen

Anfänger, keine RegEx Erfahrung/Verwendung:

```
set nomagic  
set ignorecase  
set nosmartcase
```

Verwendung	Beschreibung
Eingegebener Text	Wird direkt gesucht
Groß-/Kleinschreibung	ignoriert Groß- Kleinschreibung Nicht ignorieren mit \C im Suchtext
RegEx (1)	[C] Zeichen mit [C] maskieren
RegEx (2)	Suche mit \v... beginnen z.B. \v[X]

Fortgeschritten, RegEx wird nicht oder ab und zu verwendet:

```
set nomagic  
set ignorecase  
set smartcase
```

Verwendung	Beschreibung
Eingegebener Text	Wird direkt gesucht
Groß-/Kleinschreibung	ignoriert Groß- Kleinschreibung wenn Suchtext komplett klein geschrieben, andernfalls wird es unterschieden Nicht ignorieren mit \C im Suchtext Immer ignorieren mit \c im Suchtext
RegEx (1)	[C] Zeichen mit [C] maskieren
RegEx (2)	Suche mit \v... beginnen z.B. \v[X]

Profi, RegEx wird oft verwendet:

```
set magic  
set ignorecase  
set smartcase
```

Verwendung	Beschreibung
Eingegebener Text	Wird direkt gesucht
Groß-/Kleinschreibung	ignoriert Groß- Kleinschreibung wenn Suchtext komplett klein geschrieben, andernfalls wird es unterschieden

Verwendung	Beschreibung
	Nicht ignorieren mit <code>\C</code> im Suchtext Immer ignorieren mit <code>\c</code> im Suchtext
RegEx	Kann direkt verwendet werden (z.B. <code>/[X]</code>)
Kein RegEx	Mit <code>^V...</code> am Anfang der Suche

RegEx Suche

Hier eine Kleine Anleitung für RegEx. RegEx ist jedoch wesentlich vielfältiger, bei Interesse gibt es sehr viele Tutorials und Dokumentationen im Web.

Regular Expression versuchen immer Vorkommen der Suchmusters (Pattern) zu finden. Angenommen wir haben folgenden Text:

`Max Mustermann, Hauptstraße 17a, 11111 Irgendwo, 01234-556677,
MaxMus@gmail.com, Mobil 0171/11223344`

Suchen wir einfach nach `Max` wird es 2 mal gefunden (im Namen und in der E-Mail Adresse).

Ein Zeichen oder mehrere Mögliche Zeichen definieren

Mit `[<Zeichen>]` können wir EIN Vorkommen der angegebenen Zeichen suchen.

`[M][a][x]` sucht also nach einem M einen a und einem x... Also wieder nach `Max`.

`[M][au][xs]` jedoch nach einem M, einen a oder u und nach einem x oder s. Also wird `Max` und `Mus` gefunden.

Für `<Zeichen>` gibt es folgende Möglichkeiten:

Möglichkeit	Beschreibung	Beispiel
<code>[<Zeichenliste>]</code>	Eine Liste von Zeichen	<code>[ABC]</code>
<code>[<Bereiche>]</code>	Eine Bereich von Zeichen	<code>[0-9], [a-z], [A-Z]</code> Oder kombiniert <code>[0-9A-Za-z]</code>
<code>[^...]</code>	Ausschluss, d.h. die angegebenen Zeichen werden ausgeschlossen, alle anderen werden gesucht	<code>[^0-9]</code> (keine Ziffern)

Für bestimmte Zeichen, Zeichenkombinationen gibt es Abkürzungen:

Abkürzung	Beschreibung	Beispiel
.	Ein beliebiges Zeichen	.{5}
\s	White-Spaces (Space, Tab...)	\s{,3}
\S	Alles außer White-Spaces	\S{10}
\d	Ziffern (\rightarrow [0-9])	\d{5}
\D	Keine Ziffern (\rightarrow [^0-9])	\D{,1}
\w	Zeichen (Buchstaben, Ziffern und Unterstrich) \rightarrow [a-zA-Z0-9_]	\w{20}
\W	Keine Zeichen (Buchstaben, Ziffern und Unterstrich) \rightarrow [^a-zA-Z0-9_]	\w{20}
\xff	ASCI-Zeichen in Hex	\x20 (Space)
\x{ffff}	Unicode-Zeichen in Hex	\x{0020} (Space)

Extrem oft wird . und \d verwendet. z.B. ist \d{5} eine 5-Stellige PLZ.

Anzahl der Zeichen definieren

Mit {<zeichenanzahl>} kann festgelegt werden, wie oft angegebene Zeichen vorkommen dürfen.

Bei der PLZ 11111 reicht also [1]{5} oder kürzer 1{5} da die [...] Schreibweise nur sinn macht wenn man

- Sonderzeichen sucht
- mehrere Möglichkeiten/Zeichen definieren will

Für <zeichenanzahl> gibt es folgende Möglichkeiten:

Möglichkeit	Beschreibung	Beispiel
{n}	Exakt n mal	{5}
{n,}	Mindestens n mal (n bis unendlich)	{3,}
{,n}	Maximal n mal (0 bis n) Also ein muss nicht, kann aber maximal...	{,5}
{v,b}	Von v bis b mal	{3,5}

Auch hierfür gibt es Abkürzungen:

Abkürzung	Beschreibung	Beispiel
* , *?	Beliebig viele oder keine (0 oder mehr)	.* (beliebig viele Zeichen)
+ , +?	Beliebig viele aber min. 1 (1 oder mehr)	+{5}
? , ??	Kein oder eines (0 oder 1)	\d*-?\d* (z.B. für Telefonnr. 123-456 oder 123456)

Ohne ? nach der Abkürzung werden **SO VIELE ÜBEREINSTIMMUNGEN** wie möglich gesucht. z.B. x* sucht beliebig viele x

Mit ? aktiviert man die Menge mit **lazy**, es werden **SO WENIG ÜBEREINSTIMMUNGEN** wie möglich gesucht. z.B. x*? findet nur 1 x auch wenn dort mehrere stehen.

Anfang und Ende der Zeile definieren

^ definiert den Anfang der Zeile.

\$ definiert das Ende der Zeile.

Dies ist oft notwendig um ähnliche Vorkommen am Anfang oder Ende eindeutig finden zu können.

Gruppen

Für die VIM-Suche nicht notwendig, aber für Programmiersprachen sind Gruppen. Einfache Gruppen macht man mit (**Suchtext**)

Genannte Gruppen mit (?<Name>Suchtext)

Für was Gruppen? Damit kann man Inhalte von Suchen explizit trennen.

Für

```
Max Mustermann, Hauptstraße 17a, 11111 Irgendwo, 01234-556677,  
MaxMus@gmail.com, Mobil 0171/11223344
```

Und dem RegEx:

```
^(?<Vorname>.*), (?<Nachname>.*), (?<Strasse>.*), (?<Hausnummer>[0-9a-z]{1,10}), (?<PLZ>\d{5}), (?<Ort>.*), (?<Rufnummer>\d{3,10}[-/]\d{3,10}), (?<EMail>.*@.*[.].*), Mobil (?<Mobil>\d{3,10}[-/]\d{3,10})$
```

Wichtig: Ggf. müssen die  Zeichen bei der Rufnummer und Mobil-Nummer mit  maskiert werden. Dies ist von Sprache zu Sprache und Tool zu Tool anders.

Werden folgende Gruppen gefunden:

Gruppe	Wert
Vorname	Max
Nachname	Mustermann
Strasse	Hauptstraße
Hausnummer	17a
PLZ	11111
Ort	Irgendwo
Rufnummer	012345-556677
EMail	MaxMus@gmail.com
Mobil	0171/11223344

Sonstiges

Unter <https://regex101.com> steht ein kostenloser, Online RegEx-Editor zur Verfügung.

Hier das [Beispiel von oben](#) auf regex101.com. Rechts bei den Match Information sieht man die Gruppen und deren Werte.

Weitere Beispiele

```
-rw-r--r-- carstenschlegel staff          11.10.2023 17:22  
172812 chartmain.js
```

Benutzt Gruppen in Gruppen (Datum und Tag, Monat, Jahr; Zeit; Fullname)

```
(?<Attribute>[-rw]{10}) (?<Benutzer>.*)(?<Group>.*)(?<Datum>(?<Tag>\d{2})[.](?<Monat>\d{2})[.](?<Jahr>\d{4})) (?<Zeit>(?<Stunde>\d{2}):(?<Minute>\d{2})) [ ]*(?<Size>\d{1,10}) (?<Fullname>(?<Name>.*)[.](?<Extension>.*))
```

Suche wiederholen

Die letzte Suche kann einfach mit ; wiederholt werden (nächster Treffer). Mit , wird zum vorherigen Treffer gesprungen.

Die Suchrichtung (/ Vorwärts, ? rückwärts) hat auch Auswirkung auf ; und ,.

Hervorhebungen der Suche entfernen

Geht ganz einfach mit dem Ex-Befehl :noh für No-Highlighting.

Frühere Suchen wiederholen

Mit / und dann Cursor hoch kann einfach durch frühere Suchen gesprungen und mit Enter dieser erneut ausgeführt werden.

Suche abbrechen

Mit Esc wird die Suche abgebrochen. Auch wenn eine Position angezeigt wurde, wo das gesuchte gefunden wurde, befindet sich der Cursor wieder an der ursprünglichen Stelle. Sehr nützlich und ein Vorteil zu vielen anderen Texteditoren bei denen man die Cursorposition schon beim eingeben des ersten Suchzeichens verändert hat (z.B. VSC).

Nur herausfinden wie oft es gefunden wird

Hier hilft ein Trick:

Wir suchen den Text und weisen Vim an ihn durch nichts zu ersetzen, wird unterdrücken aber das ersetzen. Somit zeigt Vim nach der Ausführung nur an, wie oft es in wie vielen Zeilen gefunden wurde.

:%s/Suchtext//gn

Nur :%s///gn zeigt die Information für die letzte Suche an.

/Text

:%s///gn

5 match on 3 lines

Was das alles bedeutet wird in Suchen und Ersetzen beschrieben.

Cursor an das Ende eines Treffers verschieben

Soll z.B. Text gesucht werden, der Cursor aber immer auf dem t stehen, kann dies mit der Suchverschiebung erreicht werden: /Text/e

Achtung: Keine `\c` wie bei den Optionen `\cc` ... !

Suchen und Ersetzen

Zu den vielen Möglichkeiten und Varianten der Suche kommen nun noch einige Möglichkeiten für das Ersetzen.

Für das Ersetzen gibt es kein eigenes Kürzel wie `/` oder `?`. Diese sind wirklich NUR für die Suche verwendbar.

Um Text zu ersetzen benötigt man den Ex-Befehl `s` für `substitute`. Dieser hat folgenden Aufbau/Syntax:

`:[Bereich]s[ubstitute]{Muster}{Text}[/Flags]`

(`[...]` Bereich sind Optional)

Somit kann man einen `Bereich` angeben, man kann `s`, `sub` oder `substitute` schreiben, man kann `Flags` angeben, muss es aber nicht.

Der Bereich ist nicht explizit für das Ersetzen (`s`) sondern grundsätzlich für alle Ex-Befehle anwendbar und wird bei [Bereich für Ex-Befehle manuell festlegen](#) genauer beschrieben.

Tabs & Fenster

In VIM gibt es Tabs und Fenster. Die meisten Erweiterungen (z.B. für Visual Studio Code oder Eclipse) haben ein eigenes Tab/Fenster-System. Daher ist dieser Teil speziell für den echten VIM/NeoVIM Editor.

Tab-Kürzel/Befehle

Eine Datei wird in VIM `Buffer` genannt. Es kann beliebig viele Buffer geben, selbst bei nur einem Fenster/Tab. So öffnet `vim *.txt` alle `txt` Dateien in EINEM Fenster.

Kürzel/Befehl	Beschreibung
<code>:tabnew [{file}]</code>	Erstellt einen neuen Tab, optional mit Angabe der Datei
<code>gt / gT</code>	Nächster Tab bzw. vorheriger Tab
<code>{n}gt</code>	Zum Tab <code>n</code> wechseln
<code>:tabm {n}</code>	(<code>tabmove</code>) Aktuellen Tab an Position <code>n</code> verschieben
<code>tabc</code>	(<code>tabclose</code>) Aktuellen Tab schließen
<code>tabo</code>	(<code>tabonly</code>) Alle Tabs außer den Aktuellen schließen
<code>tabdo {Befehl}</code>	<code>Befehl</code> für alle Tabs ausführen

Alle Tabs mit `:wa` speichern bzw. `:qa` schließen (`:wqa` alle speichern & schließen).

Fenster (Teilung)/Buffer Kürzel/Befehle

Kürzel/Befehl	Beschreibung
<code>:new</code> / <code>:newv</code>	Neue Teilung horizontal oder vertikal
<code>:e {file}</code>	(<code>edit</code>) In einem Fenster/Tab eine Datei öffnen Öffnet nur die Datei, nicht einen Tab/Fenster!
<code>:sp {file}</code>	(<code>split</code>) Teilt das Fenster horizontal und öffnet die angegebene Datei.
<code>:vs {file}</code>	(<code>vsplit</code>) Teilt das Fenster vertikal und öffnet die angegebene Datei.
<code>:bn</code> / <code>:bp</code>	(<code>bnext/bprevious</code>) Wechselt zum nächsten / vorherigen Buffer
<code>:bd</code>	(<code>bdelete</code>) Löscht den aktuellen Buffer (Datei schließen)
<code>:b {n}</code>	(<code>buffer</code>) Wechselt zum Buffer <code>n</code>
<code>:b {filename}</code>	(<code>buffer</code>) Wechsel zur angegebenen Datei
<code>:ls</code> / <code>:buffer</code>	Listet die geöffneten Buffer auf
<code>:vert ba</code>	(<code>vertical ball</code>) Öffnet alle Buffer mit vertikaler Teilung
<code>:ba</code>	(<code>ball</code>) Öffnet alle Buffer mit horizontaler Teilung
<code>:tab ba</code>	(<code>tab ball</code>) Öffnet alle Buffer in Tabs
<code>Strg+w T</code>	Alle Fenster in Tabs verschieben
<code>Strg+w s</code>	(Split) Fenster horizontal teilen (gleicher Inhalt)
<code>Strg+w v</code>	(Vertical Split) Fenster vertikal teilen (gleicher Inhalt)
<code>Strg+w w hhkl</code>	Fenster wechseln (oder <code>Strg+w h/j/k/l</code> um das gewünschte Fenster zu selektieren) <code>Strg+w wl</code> wechselt also zum Fenster rechts neben dem aktuellen
<code>Strg+w w HJKL</code>	Fenster verschieben (anschließend H/J/K/L drücken um die Richtung zu bestimmen)
<code>Strg+w q</code>	Fenster schließen
<code>Strg+w x</code>	Fenster mit dem nächsten (darunter/darüber)
<code>Strg+w =</code>	Alle Fenster gleich groß

Ex-Befehle

Die für bestimmte Bereiche (z.B. Tabs, Fenster, suchen/ersetzen) benötigten Ex-Befehle wurde dort beschrieben.

Hier nun zusätzliche Ex-Befehle die nützlich sein können:

:AddLineNr

Fügt dem Dokument Zeilennummern am Anfang jeder Zeile hinzu

Diese stehen dann direkt im Text und können z.B. in andere Dokumente mit eingefügt werden.

:Tutor

Startet ein kleines Tutorial zu VIM

:buffers

Zeigt alle geladenen Buffers (Dateien) an.

Wechsel zu einem anderen Buffer mit `buf {n}` (bzw. `buffer {n}`) , wobei `n` die Nummer ganz links in der Buffer-Auflistung ist.

:center

Zentriert die aktuelle Zeile oder Auswahl (Visuel-Modus)

`:left` / `:right` richtet die Zeile(n) links/rechts aus

:fold

Faltet einen markierten Bereich

`:foldopen` Entfaltet ihn wieder

:help [{stichwort}]

Öffnet die Hilfe zum optional angegebenen Stichwort z.B. `help insert`

let {n}={v}

Weißt der Variable mit dem Name `n` den Wert `v` zu z.B. `let name="Max"`

Variablen können auch im Ausdrucksregister (`=`) verwendet werden.

`"=name<enter>p` fügt (`p`) den Inhalt der Variable `name` ein

Mit `echo {n}` kann der Wert einer Variable ausgegeben werden (z.B. `echo name`)

Zusätzlich können Werte addiert werden z.B.

`let z = 0 ... qa:let z = z + 1<enter>0"=z<enter>P<enter>`

Erstellt eine Variable `z` mit dem Wert 0.

Zeichnet ein Makro auf:

- `:let z = z + 1` Also `z` um 1 erhöhen
- `0` geht an Zeilenanfang
- `"=z` Ausdrucksregister, Wert von `z`
- `P` fügt den Wert von `z` ein.

Führt man dieses Makro für bestimmte Zeilen aus, würden diese mit 1, 2, 3, ... beschriftet werden.

norm {zeichen|Kürzel}

Erlaubt das Ausführen von Tasten und Kombinationen (Kürzel) im Normal-Modus.

`norm! {zeichen|Kürzel}` verwendet keine Mappings. Siehe [Funktionen](#)

Ex Befehle in der Konfigurationsdatei

Nahezu alle Ex Befehle können in der [Konfigurationsdatei](#) verwendet werden. Manche sind nützlich, manche eher unnötigt (z.B. `help` was beim starten von VIM immer erstmal die Hilfe öffnen würde).

Ex Befehle auflisten

Eine Auflistung aller Ex-Befehle oder die, die mit bestimmten Buchstaben anfangen kann man mit `:[{Text}]<Tab>` erreichen. z.B. `:a<Tab>` zeigt alle Ex-Befehle an, die mit `a` beginnen.

Bewegen in der Auflistung mit `<Tab>` bzw. `<Shift Tab>`, auswählen mit `<Enter>`.

set zum Konfigurieren

Genauso wie beim Befehlen, kann man die möglichen Parameter von `set` mit `set [{Text}]<Tab>` auflisten lassen. z.B. `set a<tab>` zeigt alle Parameter, die mit `a` beginnen.

Deaktiviert werden Einstellungen immer mit einem `no` vor dem Namen.

`set number` Aktiviert die Zeilennummern

`set nonumber` Deaktiviert die Zeilennummer

`set no<Tab>` zeigt alle Einstellungen an, die man deaktivieren könnte.

Die Einstellungen mit `set` werden neben dem [Mapping](#) am meisten in der [VIM Konfiguration](#) verwendet.

Ex-Befehle bei Markierungen

Bei markierten Zeilen sieht man nach dem drücken von `:` nicht nur das `:` Zeichen am unteren Rand von VIM. Sondern `: '<, '>`

Das `'<` steht für `Anfang der Markierung`, und das `'>` für das `Ende`.

Bereich für Ex-Befehle manuell festlegen

Man kann, wenn kein Bereich markiert ist, auch den Bereich selber festlegen:

`{nv},{nb} ...` : Bereich von `nv` bis `nb`

`:1,5 norm @a` Führt das Makro für Zeile 1 bis 5 aus.

`% ...` : Für das komplette Dokument

`:% norm @a` Führt das Makro für das komplette Dokument aus

`g{pattern} ...` : Globaler Bereich mit Prüfung

Das Pattern kann ein Regular-Expression sein oder einfach Text. Trifft das Pattern für eine Zeile zu (RegEx matched oder Text gefunden), wird für diese Zeile der Befehl (...) ausgeführt.

qa^2s# Nimmt ein Makro auf das an den Anfang der Zeile springt (^), dort 2 Zeichen ersetzt (2s) und # schreibt.

Der Text von oben mit folgendem Ex-Befehl: **g//norm @a** führt zu folgendem Ergebnis:

```
objecta.read(); # Verarbeiten objectb.calc(); # Ausgeben  
objectc.write();
```

Jede Zeile in der [//] (RegEx für //) gefunden wird, wird mittels Makro die // durch # ersetzt.

Wird nur eine einzige Zeile angegeben, ohne Ex-Befehl, wird zu dieser Zeile gesprungen (z.B. :10)

Mapping

Mapping ist a) ein Teil der [VIM Konfiguration](#) und b) nützlich bei der täglichen Arbeit mit VIM (Mapping geht in Erweiterungen für z.B. Visual Studio Code NUR in der Konfiguration!).

Die Parameter **:map <linker_teil> <rechter_teil>** des Mapping bestehen aus 2 Teilen:

- Linker Teil: Welche Taste oder Kombination wird gemappt
- Rechter Teil: Zu was wird gemappt (neuer Taste oder Kombination)

Getrennt werden die 2 Teile mit einem Leerzeichen.

Es gibt 2 Grundlegende Mapping-Varianten:

Recursives Mapping

Ein Mapping kann ein Kürzel enthalten das wiederum ein Mapping ist.

```
:map d c  
:map c b  
:map b a
```

Mapped d → c → b → a. D.h. egal ob man a, b, c oder d drückt, herauskommt immer a

Nicht Recursives Mapping

Ein Mapping hat als Ziel immer grundlegende Kürzel von VIM und kein Kürzel das per Mapping erstellt wurde.

```
:noremap d c  
:noremap c b  
:noremap b a
```

Hier wird **d** → **c**, und **c** auf **b** und **b** auf **a** gemappt. Aus dem **d** wird also ein **c** und dabei bleibt es. Ein **c** wird zu **b**. Mit **b** führt man **a** aus.
Hier ist nur **b** ein **a** und natürlich **a** selbst ist auch ein **a**.

Das noremap steht NICHT für Normal Remap sondern für None Recursive Map!

Modus für Mapping festlegen

Zusätzlich kann man mit einem Zeichen vor **map** bzw. **noremap** den Modus festlegen, für den das Mapping gilt.

Folgende Modi-Zeichen gibt es:

Zeichen	Modus
n	nur Normal-Modus
v	Visuell-Modus und Auswahl (Select)
o	Operation-Pending
x	Nur Visual-Modus
s	Nur Select-Modus
i	Einfüge-Modus
c	Kommandozeilen (Ex)
l	Einfüge, Kommandozeilen, RexEx-Suche und mehr

So kann man mit **:nmap diq di**" ein Mapping NUR für den Normal Modus erstellen.

Wenn möglich sollte IMMER ein Modus mit angegeben werden. Denn gerade **:map diq di**" funktioniert zwar, jedoch wird beim einfach **dd** auf ein dritten Zeichen gewartet, was die Ausführung von **dd** extrem verlangsamt.

Abhilfe hier schafft **:nmap diq di**" (gleiches gilt für **viq**, **ciq**, **yiq**)

Vorsicht bei recursivem Mapping

Wird z.B. `a` auf `aa` gemappt (`:map a aa`), so stellt dies im Normal-Modus kein Problem dar:

`a` wird zu `aa`

Das erste `a` wechselt einfach in den Einfüge Modus

Das zweite `a` wird eingegeben.

Anders sieht es bei Mappings aus, die NICHT in den Einfügemodus wechseln. z.B. `l` zu `ll` (`:map l ll`)

`l` wird zu `ll`

Ein `l` wird zu Cursor rechts

Ein anderen `l` wird zu `l`

Das ergibt eine Endlos-Schleife, und man springt mit einem `l` direkt zum Dokumentenende!

Schlimer noch beim Insert-Mapping von z.B. `:imap a aa`

Dies führt im Einfügemodus zum Absturz des Programms da aus einem `a` 2 `a` und daraus 4 `a` usw. werden. VIM reagiert nicht mehr und das Terminal muss beendet werden!

`:noremap a aa` funktioniert und fügt einfach nur 2 `a` anstelle von einem ein.

Damit kann man z.B. `ß` auf `ss` legen und `ä` auf `ae`

```
:inoremap ß ss
:inoremap ü ue
:inoremap ä ae
:inoremap ö oe
:inoremap Ü Ue
:inoremap Ä Ae
:inoremap Ö Oe
```

Bei der Eingabe werden die Umlaute und das `ß` automatisch in der alternativen Schreibweise eingegeben.

Sondertasten mappen

Nun können lesbare Zeichen direkt für das mappen verwendet werden. Aber was ist z.B. mit `Strg+v` oder `Esc`?

Dafür gibt es Schreibweisen um diese zu definieren.

Zusatztasten definieren

Schreibweise	Taste	Beispiel
<S-...>	Shift-Taste, z.B. Shift s	<S-s>
<C-...>	Control / Strg Taste, z.B. Strg y	<C-y>
<M-...>	Alt / Meta Taste, z.B. Alt F4	<M-F4>
<A-...>	Alt / Meta Taste, z.B. Alt F4	<A-F4>
<D-...>	Command / Super Key, z.B. Cmd A	<D-A>
Kombinationen	z.B. Shift+Strg a	<SC-a>
<k...>	Nummernblock, z.B. Num 5	<k5>
<F...>	Funktionstasten, z.B. F5	<F5>

Sonderzeichen definieren

Schreibweise	Taste
<BS>	Backspace
<Tab>	Tabulator
<CR>	Return/Enter
<Enter>	Return/Enter
<Return>	Return/Enter
<Esc>	Escape
<Space>	Space
<Up> / <Down>	Cursor hoch/runter
<Left> / <Right>	Cursor links/rechts
<Insert>	Einfügen (Einf)
	Delete
<Home>	Home/Pos1
<End>	End/Ende
<PageUp>	Bild hoch
<PageDown>	Bild runter
<lt>	< Taste
<bslash>	Backslash ()

Wichtig ist hier, das man die < Taste nicht einfach mappen kann, da diese ja für die Sonderzeichen-Definition verwendet wird.

map < h würde nicht gehen hier muss man
map <lt> h verwenden.

Sonstiges Besonderheiten

Leader

Mit `set mapleader="{c}"` z.B. `set mapleader="ü"` kann ein Leader-Zeichen definiert werden. Dieses Zeichen kann im Mapping mit `<Leader>` verwendet werden.

Dies wird hauptsächlich für Kombinationen verwenden und teilweise auch von Erweiterungen (um diese zu aktivieren).

```
set mapleader="ü"  
map <Leader>x i
```

Mappet also `üx` auf `i`

`<Leader>` kann auch mehrmals verwendet werden

```
map <Leader><Leader>x I
```

Mappet `üüx` auf `I`

Gar nichts

Will man Tasten oder Kombinationen in's **nichts** laufen lassen, kann man `<Nop>` für No Operation verwenden.

```
noremap <Up> <Nop>  
noremap <Down> <Nop>  
noremap <Left> <Nop>  
noremap <Right> <Nop>
```

Mappet die Cursor-Tasten zum nichts. D.h. die normalen Cursortasten funktionieren dann nicht mehr!

Zum lernen von VIM eine gute Methode, wenn die Cursor-Tasten nicht mehr funktionieren MUSS man `hjkl` verwenden

Macht man dies auch mit `inoremap <...> <Nop>` kann man auch im Einfügemodus keine Cursortasten mehr verwenden (was nicht so sinnvoll ist, da man dort vielleicht das eine oder andere mal die Cursor-Tasten drück um nicht 2 Modi-Wechsel vornehmen zu müssen).

Funktionen

Auch Funktionen können, wie das Mapping, in der [VIM Konfiguration](#) verwendet werden. Sie erlauben die Verwendung komplexerer Abläufe mittels einem selber erstellten Ex-Befehl:

Beim Mapping werden Tasten oder Kombinationen verändert. Funktionen erstellen einen Befehl der im Ex-Modus ausgeführt werden kann.

Aufbau von Funktionen

```
function Name(Parameter)
...
endfunction
```

Der Inhalt einer Funktion sind [Ex-Befehle](#)

Normal-Kürzel in einer Funktion ausführen

Hier wird der [normal](#)-Ex-Befehl verwendet (abgekürzt [norm](#))

```
function Test()
norm G
endfunction
```

[:call Test\(\)](#) springt also zum Dokumentenende.

Gerade bei Verwendung von Sonderzeichen ist es sinnvoller den [execute](#) ([exe](#)) Ex-Befehl zu verwenden.

```
nmap <C-a> ggVG
function Test()
exe "norm \<C-a>"
endfunction
```

Zuerst wird [Strg+A](#) auf [ggVG](#) ([gg](#) → Dokumenten Anfang; [V](#) Zeilen-Visuell-Modus, [G](#) Dokumenten Ende) gemappt.

[:call Test\(\)](#) markiert also das komplette Dokument.

Bei [exe](#) müssen [<](#) Zeichen mit dem Escape-Zeichen [\](#) angegeben werden.

Ex-Befehl erstellen

Nun ist `:call Test()` keine schöne Methode eine Funktion auszuführen. Besser wäre nur `Test`

Hierzu kann man mit `command` ein Ex-Befehl erstellt der einen anderen Ex-Befehl ausführt.

```
command Test call Test()
```

Der Ex-Befehl `Test` führt also `call Test()` aus. Theoretisch kann man hier auch kleine Kombinationen erstellen:

```
command Alles norm ggVG
```

`Alles` führt `norm ggVG` aus, was den kompletten Text markiert.

Um evtl. vorherige Definitionen vom Ex-Befehle zu überschreiben muss `command! ...` geschrieben werden. Es ist grundsätzlich sinnvoll **immer** `command! ...` zu verwenden um Fehler in der [VIM Konfiguration](#) durch mehrmaliges definieren eines Ex-Befehls zu vermeiden.

```
command! Test call Test()
```

Funktion neu definieren

Auch hier wird das `!` verwendet.

```
function! Test()  
    norm iGleicher Funktionsname, andere Funktion  
endfunction
```

(wichtig ist das `i` damit man vom `normal` Modus in den Einfügemodus kommt, der Rest ist nur Text der eingegeben wird).

Sonstiges

Funktionsnamen sind Case-Sensitiv. d.h.

```
function Test()  
    ...  
endfunction  
function test()  
    ...  
endfunction
```

sind 2 unterschiedliche Funktionen!

Auch beim Aufruf von `call Test()` bzw. `call test()` ist darauf zu achten!

Funktionen können im Ex-Modus direkt eingegeben werden. Der Ex-Modus wird dann beendet, wenn ein `endfunction` eingegeben wird. Dann steht die Funktion zur Verfügung.

Funktionen erlauben die Verwendung von Variablen mit `let varName=Wert`. Dies sind jedoch auch nur ganz normale Ex-Befehle.

Parameter werden in der Funktion mit `a:parameterName` angesprochen.

```
function Test(msg)  
    echo a:msg  
endfunction
```

#TODO cib → ci(, ciq → ci", ciB → ci{, ct (Tag), ciw → ci[

Cursor muss nicht in (), "", {}, [] sein, sondern das erste wird gesucht (Achtung bei {-Blöcken da man meisten in einem ist (Function, Klasse, Namespace,...))

Remap:

noremap cib ci{ (b wie Block)

noremap cic ci((c wie die Klammer)

noremap cia ci[(a wie Array)

Visual Studio Code und VIM

Für die meisten gut verwendbar für's Coding oder andere Arbeiten mit großen oder vielen Texten.

EasyMotion

Leader setzen in Settings (`⌘ ,`) auf `ü`

`üüs<char>` Sucht das Zeichen auf der ganzen Seite

`üuf<char>` Sucht das Zeichen AB Cursor

`üuf<char>` Sucht das Zeichen VOR Cursor

`üuh / üül` Zeigt alle Anfangsbuchstaben/CamelCase links/rechts vom Cursor an

`üuj / üük` Zeigt alle ersten Buchstaben der Zeilen unter/über dem Cursor an

`üuw / üüb` Zeigt alle Anfangsbuchstaben links/rechts vom Cursor an

`üü/<text>?` Sucht `text` und zeigt alle Anfangsbuchstaben nach dem Cursor an

`üü?<text>?` Suche `text` und zeigt alle Anfangsbuchstaben vor dem Cursor an

Sonstiges, Links, ...

Krasses Video zur Konfiguration von NeoVim: [# 0 to LSP : Neovim RC From Scratch](#)