PM1/PT Ruby: Objektsammlungen und Iteratoren

Collections, Container und weitere Datenstrukturen

Konzepte

- Objektsammlungen (Array)
- Iteratoren
- Intervalle (Range)
- Mehrdimensionale Sammlungen
- Zurückführen von Arraymethoden
  auf eine Basis von Methoden.

Motivation und Einführung

- In der Programmierung benötigen wir häufig Objektsammlungen. Beispiele:
  - PDAs speichern Notizen oder Verabredungen
  - Bibliotheken verwalten Informationen über Bücher und Zeitschriften
  - Universitäten halten Daten über Studenten

- Typische Eigenschaft solcher Sammlungen: die Anzahl der Elemente verändert sich über die Zeit. Sie haben keine feste Größe.

- Ein erster Versuch eine Klasse mit einer großen Menge von Instanz-Variablen zu definieren, die die Sammlung von Objekten aufnimmt, ist daher nicht zielführend, da wir hier die Anzahl der Elemente festlegen.

Motivation und Einführung

- In objektorientierten Programmiersprachen gibt es daher Klassen für die Modellierung von Objektsammlungen variabler Größe.

- Zwei Kategorien werden unterschieden:
  - **Listen oder Arrays**: Sammlungen variabler Größe mit einer definierten Ordnung für die Elemente
  - **Mengen**: Sammlung variabler Größe ohne eine definierte Ordnung für Elemente und ohne Dubletten.

- In Ruby ist die Klasse für die erste Kategorie ***Array***, die Klasse für die zweite Kategorie ***Set***.

- Daneben gibt es manchmal Spezialformen von Objektsammlungen: In Ruby ist eine solche Spezialform die Klasse ***Range***, die Intervalle repräsentiert.

Das Notizbuchbeispiel

- Wir wollen ein Notizbuch mit folgenden Eigenschaften entwerfen:
  - Notizen können gespeichert werden
  - Die Anzahl der Notizen ist nicht begrenzt
  - Einzelne Notizen können angezeigt werden
  - Die Anzahl der Notizen kann angezeigt werden.
- Wir werden das Notizbuch mit Hilfe der Klasse ***Array*** implementieren
  - ***Array*** kann beliebig viele Elemente aufnehmen.
  - ***Array*** ermöglicht es, auf Objekte unter Angabe der Position eines Objekts im Array zuzugreifen.
  - ***Array*** gibt Auskunft über die Anzahl der enthaltenen Elemente

Die Implementierung des Notizbuchs

- Bei genauer Betrachtung wird die Funktionalität des Notizbuch im Wesentlichen von einem Objekt der Klasse ***Array*** übernommen, das wir beim Erzeugen des Notizbuchs in der Instanvariable ***@notizen*** speichern.

- Die Klasse Array hat eine Vielzahl von Methoden, von denen wir einige in diesem Beispiel nutzen:
  - Für das Hinzufügen, Anhängen von Notizen in/an ein Notizbuch die Methode **<<** (append).
  - Für das Abfragen der Größe eines Notizbuches die Methode ***length().***
  - Beim Anzeigen eines Elementes mit einer eindeutigen Nummer, der Elementzugriff über die Position ***[nummer].***

```
class Notizbuch

  def initialize()
    @notizen = Array.new()
    # @notizen = [] Arrayliteral
  end

  def notiz_speichern(notiz)
    @notizen << notiz
  end

  def anzahl_notizen()
    return @notizen.length()
  end

  def notiz_zeigen(nummer)
     if (0 <= nummer &&
          nummer < anzahl_notizen())
         puts @notizen[nummer]
     end
  end
end
```

Exkurs: Arrayliterale

- Arrays können analog Strings durch **Arrayliterale** erzeugt werden.
- Ein Arrayliteral ist immer durch eckige Klammern begrenzt.
- ***[1,2,"morgen", Notizbuch.new()]***: Innerhalb der Klammern können beliebig viele durch Komma getrennte Elemente aufgezählt werden.
- ***[]*** ist das leere Array

Objektstrukturen mit Sammlungen

- Um die Arbeitsweise von Arrays besser zu verstehen, schauen wir uns nun an, wie das Notizbuch aussieht, wenn wir zwei Notizen eingefügt haben.
- Dazu schreiben wir uns ein kleines Szenario als Ruby-Script, das 2 Notizen in ein Notizbuch einfügt.
- Abbildung rechts zeigt dieses Script. Das Ergebnis schreiben wir als Objektdiagramm auf und schauen uns das Entstehen im Debugger an.

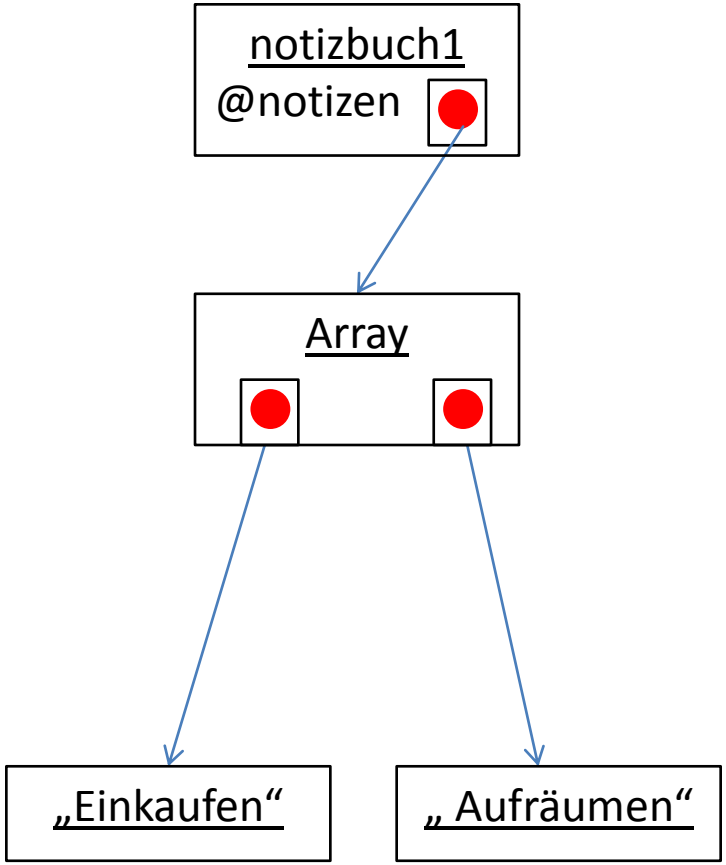
```
require 'Notizbuch'

notizbuch = Notizbuch.new()
notizbuch.notiz_speichern
    ("Einkaufen")
notizbuch.notiz_speichern
    ("Aufraeumen")
```

Objektstrukturen mit Sammlungen

- Die Abbildung rechts zeigt das Objekt-Diagramm für das Notizbuch nach Einfügen von zwei Notizen

- Mindestens 3 Eigenschaften von Arrays sollten Ihnen aufgefallen sein:
  - Das Array kann seine Größe beliebig erweitern. Wenn weitere Objekte eingefügt werden, wird Platz geschaffen.
  - Das Array merkt sich die Anzahl der eingefügten Elemente.
  - Die Reihenfolge, in der die Elemente eingefügt werden, wird beibehalten.

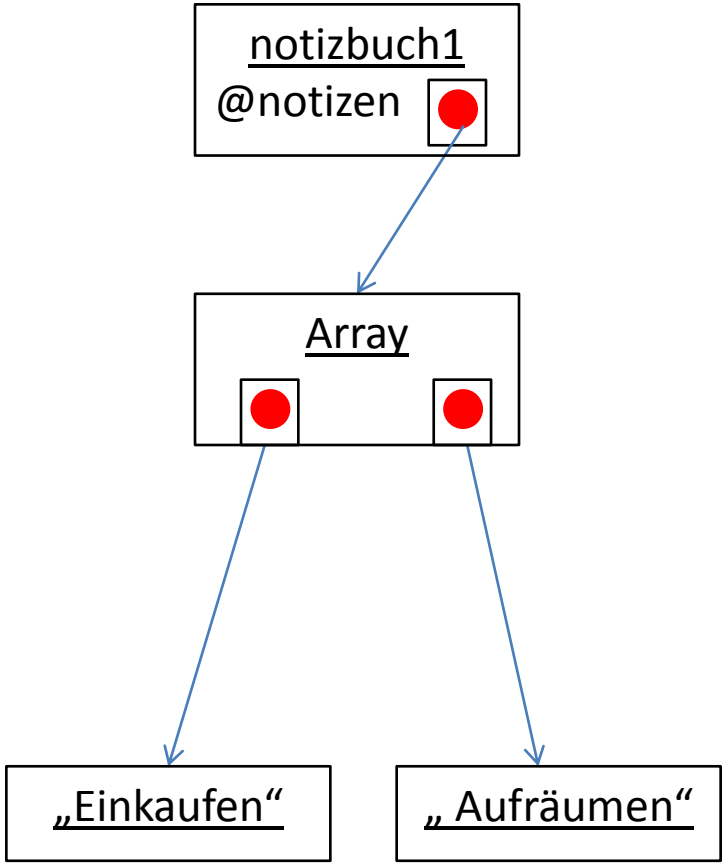


Objektstrukturen mit Sammlungen

- ***Array*** scheint die Funktionalität, die wir für unser Notizbuch brauchen vollständig anzubieten. Daher können wir die Klasse einfach nutzen.

- Ein weiterer Vorteil von der Nutzung von ***Array*** ist, dass ein Array einen Zähler für die enthaltenen Elemente verwaltet. Daher brauchen wir keine Instanz-Variable, die sich die Position der Elemente merkt.

- Das Konzept, das wir in dem einfachen Notizbuch anwenden, die Implementierung von Funktionalität durch Weiterreichen der Aufrufe an eine andere Klasse, wird auch **Delegation** genannt.



Nummerierung in Sammlungen

- Wir schreiben ein zweites Szenario, in dem wir zunächst 3 weitere Elemente in das Notizbuch einfügen und dann das erste, dritte und fünfte Element aus dem Notizbuch ausgeben lassen.

- Wenn wir das Szenario ablaufen lassen sehen wir, dass das erste Element im Notizbuch die Nummer 0 hat, das dritte die Nummer 2 und das letzte die Nummer 4.

- Die Position eines Elementes in einer Sammlung wird auch **Index** genannt. **Ein Array beginnt also immer mit dem Index 0 zu zählen.**

```
require 'Notizbuch'

notizbuch = Notizbuch.new()
notizbuch.notiz_speichern("Einkaufen
    ")
notizbuch.notiz_speichern("Aufräumen
    ")
notizbuch.notiz_speichern("Sport
    diese Woche")
notizbuch.notiz_speichern("Venedig
    Vortrag")
notizbuch.notiz_speichern("Eltern
    anrufen")

notizbuch.notiz_zeigen(0)
notizbuch.notiz_zeigen(2)
notizbuch.notiz_zeigen(4)
```

Nummerierung in Sammlungen

- Die Methode ***notiz_zeigen*** prüft vor der Ausgabe eines Elementes, ob der Index im Intervall ***[0,anzahl_notizen)*** liegt.
- Ohne diese Abfrage würde auf eine Position zugegriffen, auf der kein Element steht.
- Zugriffe über einen Index außerhalb des gültigen Bereich liefert in Ruby immer den Wert ***nil***.

```
def notiz_zeigen(nummer)
    if (0 <= nummer &&
        nummer < anzahl_notizen())
        puts @notizen[nummer]
    end
end
```

Exkurs: Intervallprüfungen mit *Range*

- Ruby enthält die Klasse ***Range***, die Intervalle modelliert.
- Wie für Strings und Arrays können Ranges über Literale erzeugt werden.
- Es gibt zwei Arten von Intervallen.
  - nach oben geschlossene Intervalle
  - nach oben offene Intervalle
- Ein nach oben geschlossenes Intervall mit den Grenzen ***min*** und ***max_inkl***: ***min..max_inkl***
- Ein nach oben offenes Intervall mit den Grenzen ***min*** und ***max_exkl***: ***min…max_exkl***
- Um zu prüfen, ob ein Element in einem Intervall liegt, benutzen wir eine der Methoden ***===, include?*** oder ***member?***

Methoden für Ranges

Kategorie	Methoden
Intervallgrenzen	begin, first, end, last
Gleichheit	==, eql?
Enthaltensein	===, include?, menber?
Konvertierung	to_s, to_a (erzeugt ein Array aus dem Range, in dem alle diskreten Werte zwischen Intervallstart und Intervallende aufsteigend aufgezählt sind.)
Iteratoren (gleich bei Arrays)	each, inject, select, reject, collect,
Iterator	step(n) Geht in der Schrittweite n über den Range und führt den übergebenen Block aus.

Intervallgrenzen - Methoden für Ranges

Methode	Beispiel
rng.first => obj rng.begin => obj Liefert den Intervallstartwert, auch dann wenn der Range nicht korrekt definiert wurde.	(1..5).begin #=> 1 (1...5).first #=> 1 (1..-5).begin #=> 1 (1...-5).first #=> 1
rng.end => obj rng.last => obj Liefert den bei der Definition angegebenen Intervallendwert, auch dann wenn der Range nicht korrekt definiert wurde.	('aaa'..'aaf').end #=> 'aaf' ('aaa'...'aaf').end #=> 'aaf' ('aaa'..'aaf').last #=> 'aaf' ('aaa'...'aaf').last #=> 'aaf'

Gleichheitsmethoden für Ranges

Methode	Beispiel
rng == obj => true or false rng == obj, wenn beide gleiche begin und end Werte haben (Test mit ==) und für beide exclude_end? gleich ist.	r1 = (1 .. 5) r2 = (1.0 .. 5.0) r3 = (1 … 5) r4 = (1 .. 5) r1 == r2 #=> true r1 == r4 #=> true r1 == r3 #=> false
rng.eql?(obj) => true or false rng.eql?(obj), wenn beide gleiche begin und end Werte haben (Test mit eql?) und für beide exclude_end? gleich ist.	r1.eql?(r2) #=> false r1.eql?(r3) #=> false r1.eql?(r4) #=> true

Enthaltensein-Methoden für Ranges

Methode	Beispiel
rng === obj => true or false rng.member?(val) => true or false rng.include?(val) => true or false Prüft, ob obj in rng enthalten ist. Verwendet dazu den Vergleichsoperator <=> von obj und den tatsächlichen Intervallgrenzen.	r1 = (1 .. 5) r2 = (1.0 .. 5.0) r3 = (1 … 5) r4 = (‘aaa‘ .. ‘aaf‘) r1 === 5 #=> true r1 === 0 #=> false r1 === 5.0 #=> true r2 === 5 #=> true r3 === 5 #=> false r4 === ‘aaa‘ #=> true r4 === ‘aac‘ #=> true r4 === ‘aba‘ #=> true r4 === ‘aa‘ #=> false r4 === ‘AAA‘ #=> false r4 === ‘AAA‘.downcase #=> false

Übungen

- **Ü5-1: Wie sieht das Literal für ein nach oben geschlossenes Intervall mit dem Grenzen -1 und 8 aus?**
- **Ü5-2: Wie sieht ein Literal für ein nach oben offenes Intervall mit den Grenzen 3 und 9 aus?**
- **Ü5-3: Wenn ein Array 10 Elemente enthält, welchen Wert liefert dann die Methode length()?**
- **Ü5-4: Wenn ein Array n Elemente enthält, wie müssen Sie dann auf das letzte Element zugreifen?**
- **Ü5-5: Schreiben Sie bitte die Methode notiz_zeigen unter Verwendung eines Ranges um!**

Elemente aus einer Sammlung entfernen

- Wir wollen nicht länger benötigte Notizen aus unserem Notizbuch entfernen.
- Die Methode ***notiz_entfernen(nummer)*** benutzt dazu die Methode ***delete_at*** der Klasse ***Array***.
- ***delete_at(k)*** entfernt das ***k+1*** Element eines Arrays

```
def notiz_entfernen(nummer)
    if    (0...anzahl_notizen()).include?(nummer)
      @notizen.delete_at(nummer)
    end
 end
```

Elemente aus einer Sammlung entfernen

- ***delete_at*** arbeitet destruktiv.
- Das Verhalten von ***delete_at*** überprüfen wir mit Hilfe des nebenstehenden Szenarios und des Debuggers und schauen uns das Notizbuch vor und nach dem Löschen eines Elementes an.

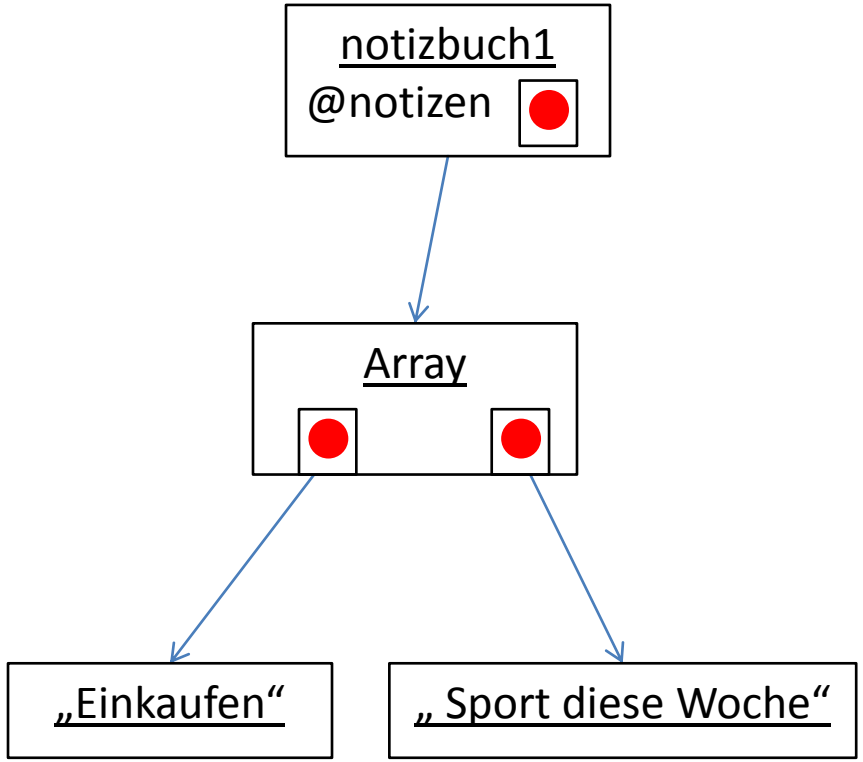
```
require 'Notizbuch'

notizbuch = Notizbuch.new()
notizbuch.notiz_speichern("Einkaufen
    ")
notizbuch.notiz_speichern("Aufräumen
    ")
notizbuch.notiz_speichern("Sport
    diese Woche")

puts("Aufräumen an den Freund
    delegiert")
notizbuch.notiz_entfernen(1)
notizbuch
```

Elemente aus einer Sammlung entfernen

- Wenn wir ein Element aus der Sammlung entfernen, dann verringert sich der Index aller nachfolgenden Elemente um 1.
- Entfernen wir im Szenario das 2‘te Element, dann stellt sich das Objekt-diagramm anschließend wie folgt dar.
- Wenn wir diese Indexverschiebung bei mehrfachem Löschen nicht geeignet berücksichtigen, dann „überspringen“ wir beim Löschen einzelne Elemente.



Elemente aus einer Sammlung entfernen

- Wir gehen wieder von dem Notizbuch mit den fünf Elementen aus dem zweiten Szenario aus.

- **Ü5-6: Wir wollen das zweite und dritte Element entfernen. Wie sieht die Sequenz von Methodenaufrufen auf dem Notizbuch aus?**

- **Ü5-7: Wir wollen das zweite, dritte und fünfte Element entfernen. Wie sieht die Sequenz von Methodenaufrufen auf dem Notizbuch dazu aus?**

```
require 'Notizbuch'

notizbuch = Notizbuch.new()
notizbuch.notiz_speichern("Einkaufen
    ")
notizbuch.notiz_speichern("Aufräumen
    ")
notizbuch.notiz_speichern("Sport
    diese Woche")
notizbuch.notiz_speichern("Venedig
    Vortrag")
notizbuch.notiz_speichern("Eltern
    anrufen")
```

Komplette Sammlung verarbeiten

- Um den kompletten Inhalt einer Sammlung vor und nach einem Methodenaufruf ausgeben zu können, benötigen wir eine Methode, die über alle Elemente des Arrays „läuft“ und diese nacheinander ausgibt.

- Dieses „über alle Elemente laufen“ heißt in Programmiersprachen über eine Sammlung zu **iterieren,** die Methode, die das Iterieren übernimmt, heißt **Iterator**.

- Der Iterator in Ruby ist die Methode ***each.***

- Jede Sammlung und fast jede Datenstruktur, für deren Elemente sich eine Ordnung definieren lässt, hat in Ruby einen Iterator.

Komplette Sammlung verarbeiten

- Wir schreiben eine Methode ***notizbuch_ausgeben,*** die den vollständigen Inhalt auf der Konsole ausgibt.
- Dazu iterieren wir mit der Methode ***each*** über das Array ***@notizen***, das die Notizen verwaltet.
- ***each*** holt nacheinander die Elemente aus dem Array ***@notizen*** und weist das jeweils aktuelle Element der Variable ***notiz*** zu. ***puts*** gibt dann den Inhalt der Variable ***notiz*** aus.

```
def notizen_ausgeben
      @notizen.each {|notiz|
        puts notiz
      }
end
```

Komplette Sammlung verarbeiten

- ***each*** holt nacheinander die Elemente aus dem Array ***@notizen*** und weist das jeweils aktuelle Element der Variable ***notiz*** zu. ***puts*** gibt dann den Inhalt der Variable ***notiz*** aus.
- Die geschweiften Klammern hinter dem ***each*** fassen einen Code**block** zusammen, der für jedes Element in dem Array ***@notizen*** ausgeführt wird.
- Die Variable ***notiz*** zwischen den senkrechten Strichen heißt **Blockvariable**.
- Ein Codeblock kann auch lokale Variablen enthalten. Daher **müssen** Blockvariablen immer von senkrechten Strichen umschlossen werden.

```
def notizen_ausgeben
      @notizen.each {|notiz|
        puts notiz
      }
 end
```

Komplette Sammlungen verarbeiten

- Wir können jetzt im letzten Szenario die den Inhaltes eines Notizbuches vor und nach dem Löschen mit der Methode ***notizen_ausgeben*** überprüfen.

```
require 'Notizbuch'

notizbuch = Notizbuch.new()
notizbuch.notiz_speichern("Einkaufen")
notizbuch.notiz_speichern("Aufraeumen")
notizbuch.notiz_speichern("Sport diese
    Woche")
notizbuch.notiz_speichern("Venedig
    Vortrag")
notizbuch.notiz_speichern("Eltern
    anrufen")


notizbuch.notizen_ausgeben
notizbuch.notiz_entfernen(1)
puts("Aufraeumen an den Freund
    delegiert")
notizbuch.notizen_ausgeben
```

Iterieren mit *for .. in ..*

- Eine alternative Schreibweise für einen Iterator über eine Sammlung ist das Konstrukt
  for <elem> in <sammlung>

- Dieses Konstrukt wird von Ruby in die ***each*** Schreibweise umgewandelt.

```
def notizen_ausgeben()
    for notiz in @notizen
      puts(notiz)
    end
 end
```

Komplette Sammlung mit Schleifen verarbeiten

- **Zählschleifen** mit **while,** die Alternative um eine komplette Sammlung zu verarbeiten, kennen wir aus der Darstellung von Kontrollstrukturen.
- Wenn wir eine Sammlung mit einer Schleife verarbeiten, dann benutzen wir einen Zähler für die Indizes der Elemente und zählen diesen solange hoch wie er kleiner der Länge der Sammlung ist.
- Im Unterschied zum Iterator müssen wir in dieser Lösung auf die Elemente über deren **Index** zugreifen.
- **Vorteile:** Wir kennen den Index und können ihn z.B. ausgeben.
- **Nachteile:** der Index muss mit 0 initialisiert werden. Der Index muss hochgezählt werden.

fehlerhaft: index nicht initialisiert und index nicht hochgezählt

```
def notizen_nummerieren()
  while(index < anzahl_notizen())
    puts("#{index}: #{@notizen[index]}")
  end
end
```

korrekt

```
def notizen_nummerieren()
  index = 0
  while(index < anzahl_notizen())
    puts("#{index}: #{@notizen[index]}")
    index += 1
  end
end
```

Iterator *each_index*

- Der Iterator ***each_index,*** der in der Klasse ***Array*** definiert ist, vereinigt die Vorteile eines Iterators, der über Inhalte läuft und einer Schleife, die den Index explizit verwaltet.

- Wenn wir bei einer Iteration über ein Array einen Index benötigen, dann sollte immer die Methode ***each_index*** verwendet werden.

- **Ü5-8: Schreiben Sie bitte die Methode notizen_nummerieren mit der Methode each_index!**

Übungen

- **Ü5-9: Schreiben Sie bitte eine Methode, die in unserem Notizbuch nach einer Notiz sucht, die ein spezielles Wort enthält, und dieses zurückgibt!**
- **Ü5-10: Schreiben Sie bitte eine Methode, die in unserem Notizbuch alle Notizen findet, die ein spezielles Wort enthalten und eine Sammlung dieser Notizen zurückgibt!**
- **Ü5-11: Schreiben Sie bitte eine Methode, die aus einem Array mit Zahlen alle Vorkommen von 3 löscht! Was ist dabei zu beachten?**
- **Ü5-12: Schreiben Sie bitte eine Methode, die aus unserem Notizbuch alle Notizen löscht, die ein spezielles Wort enthalten!**

- Für die Lösung von drei dieser Aufgaben brauchen wir u. A. die Methode ***include?*** von String.

Elemente in Sammlungen austauschen

- Sie haben sich verschrieben und möchten den Fehler korrigieren. Statt „Einkaufen“ steht „Eintaufen“ als Notiz im Notizbuch.
- Um diesen Fehler zu beseitigen, könnten wir die Eintrag „Eintaufen“ löschen und den korrekten Eintrag dem Notizbuch hinzufügen. Das allerdings würde die Reihenfolge verändern.
- Eigentlich wollen Sie den falschen Eintrag durch den richtigen ersetzen.
- Dazu verwenden wir die Methode ***[nummer]=wert*** der Klasse Array, die auch **Elementzuweisung** genannt wird.

```
def notiz_austauschen(nummer,neue_notiz)
    if (0...anzahl_notizen).include?(nummer)
      @notizen[nummer] = neue_notiz
    end
 end
```

Elemente in Sammlungen austauschen

```
require 'Notizbuch,
notizbuch = Notizbuch.new()
notizbuch.notiz_speichern("Eintaufen
    ")
notizbuch.notiz_speichern("Aufraeume
    n")
notizbuch.notiz_speichern("Sport
    diese Woche")
notizbuch.notiz_speichern("Venedig
    Vortrag")
notizbuch.notiz_speichern("Eltern
    anrufen")

notizbuch.notizen_ausgeben
notizbuch.notiz_austauschen(0,"Einka
    ufen")
puts("---- KORRIGIERT ----")
notizbuch.notizen_ausgeben
```

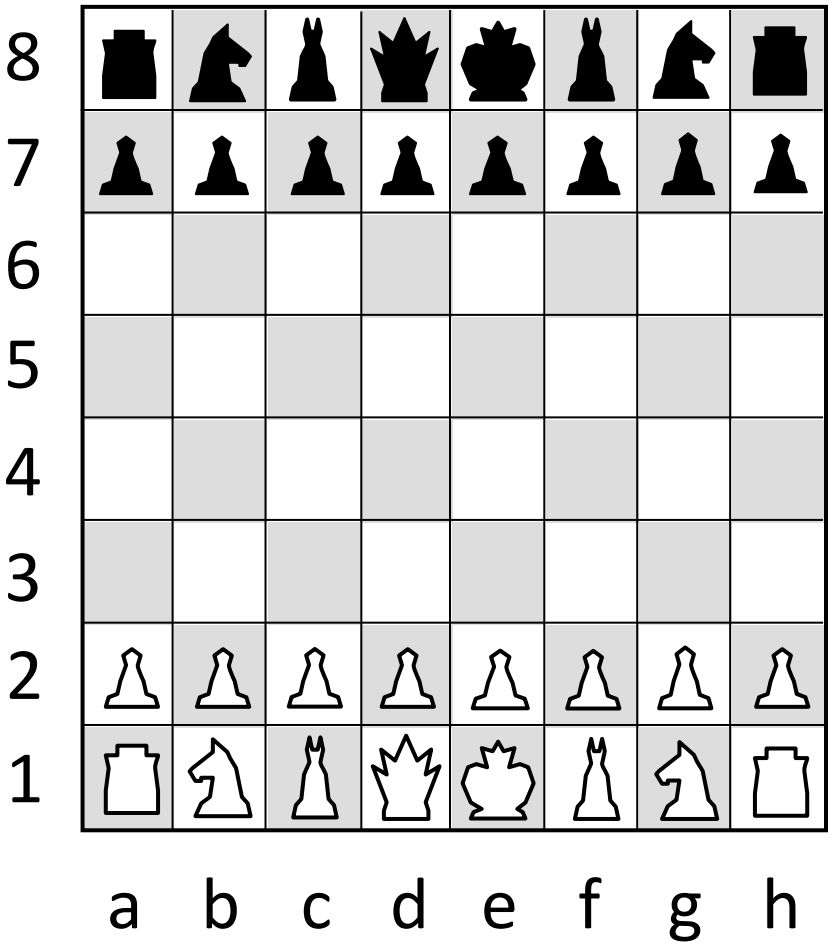
- Wir überprüfen die Korrektur in dem untenstehenden Script.

- **Ü5-13: Schreiben Sie bitte eine Methode, die eine Notiz gegen eine neue Notiz austauscht! In der Methode sollen die zu tauschende Notiz (nicht deren Index) und die neue Notiz übergeben werden.**

Exkurs zweidimensionale und dreidimensionale Arrays

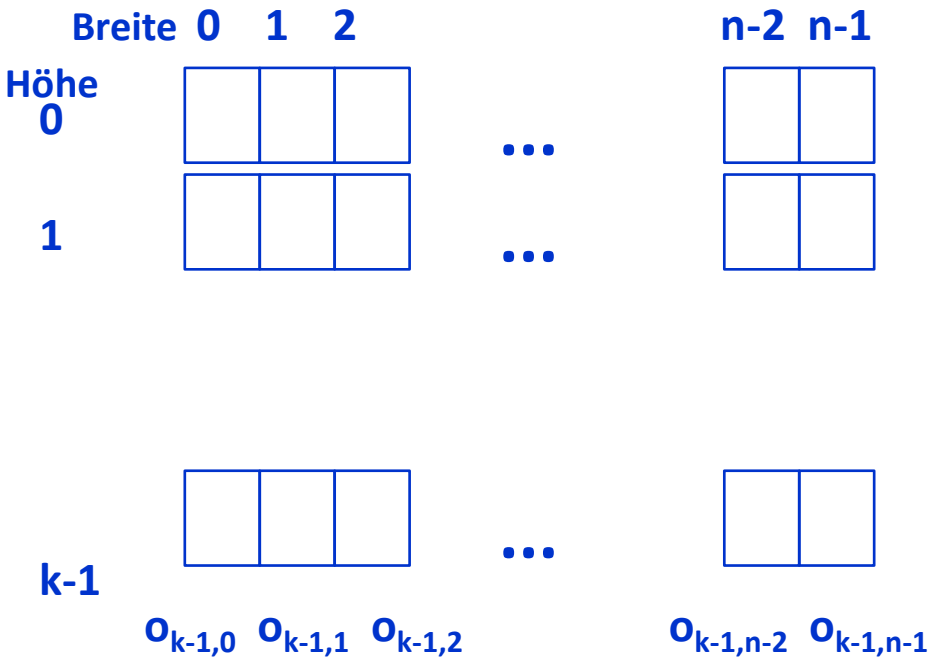
- Ein Schachbrett ist ein Array, auf dem die Position einer Figur von zwei Größen abhängt, der Position in der horizontalen Reihe (a-h) und der Position in der vertikalen Reihe (1-8)

- Arrays, in der die Position der Objekte durch zwei Größen eindeutig bestimmt ist, heißen 2-dimensionale Arrays.



2-dimensionale Arrays

- Abstrahieren wir von dem konkreten Beispiel, so erhalten wir als Darstellung für ein zweidimensionales Array:



- Dabei sind:
  - $o_{0,0}$-$o_{k-1,n-1}$ die Objekte in dem Array
  - (0,0) - (k-1,n-1) die Positionen der Objekte in dem Array
  - (k,n) die Längen der Dimensionen des Arrays

Das Schachbrett als 2-dimensionales Array in Ruby

- Erzeugen des Schachbretts

```
a_h = 8
eins_acht = 8
schachbrett = Array.new(eins_acht){Array.new(a_h)}
#Figuren
w_d = "weisse_dame"
w_k = "weisser_koenig"
# etc.
s_b1 = "schwarzer_bauer1"
s_b2 = "schwarzer_bauer2"
# etc
```

- Erzeugen der Ausgangsstellung: Gezieltes Setzen der Figuren auf ihre jeweilige Ausgangsposition.

```
a=0,b=1,c=2,d=3,e=4,f=5,g=6,h=7
#Ausgangsstellung nicht vollständig
schachbrett[7][d] = w_k
schachbrett[7][e] = w_d
schachbrett[1][a] = s_b1
schachbrett[1][b] = s_b2
```

Das Schachbrett als 2-dimensionales Array in Ruby

- Anstelle einzelner Figuren können wir auch ganze Reihen lesen oder schreiben.
- Eine Reihe entspricht einem eindimensionalen Array.

```
#Figuren
w_d = "weisse_dame"
w_k = "weisser_koenig"
w_t1 = "weisser_turm"
w_t2 = "weisser_turm"
w_l1 = "weisser_laufer"
#etc

# Lesen von Reihen
reihe_7 = schachbrett[7]
```

- Das Schreiben einer Reihe erledigen wir mit einem **Array-Literal.**

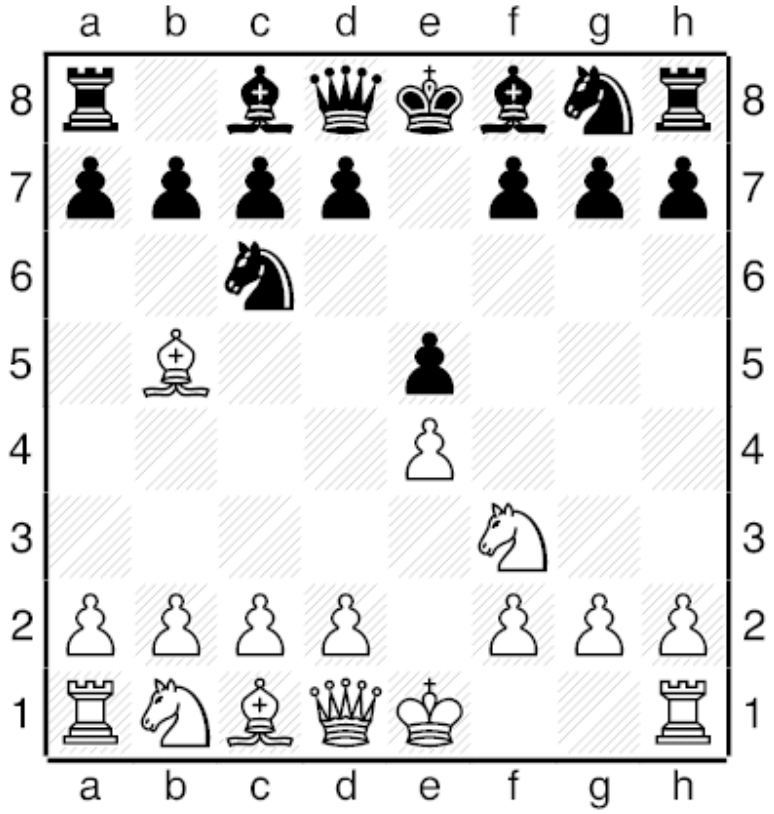
```
# Schreiben von Reihen
schachbrett[7] =
    [w_t1,w_s1,w_l1,w_k,w_d,w_l2,w_s2
    ,w_t2]
```

- **Ü5-14: Vervollständigen Sie bitte die Ausgangsstellung und spielen Sie die folgende Eröffnung!**

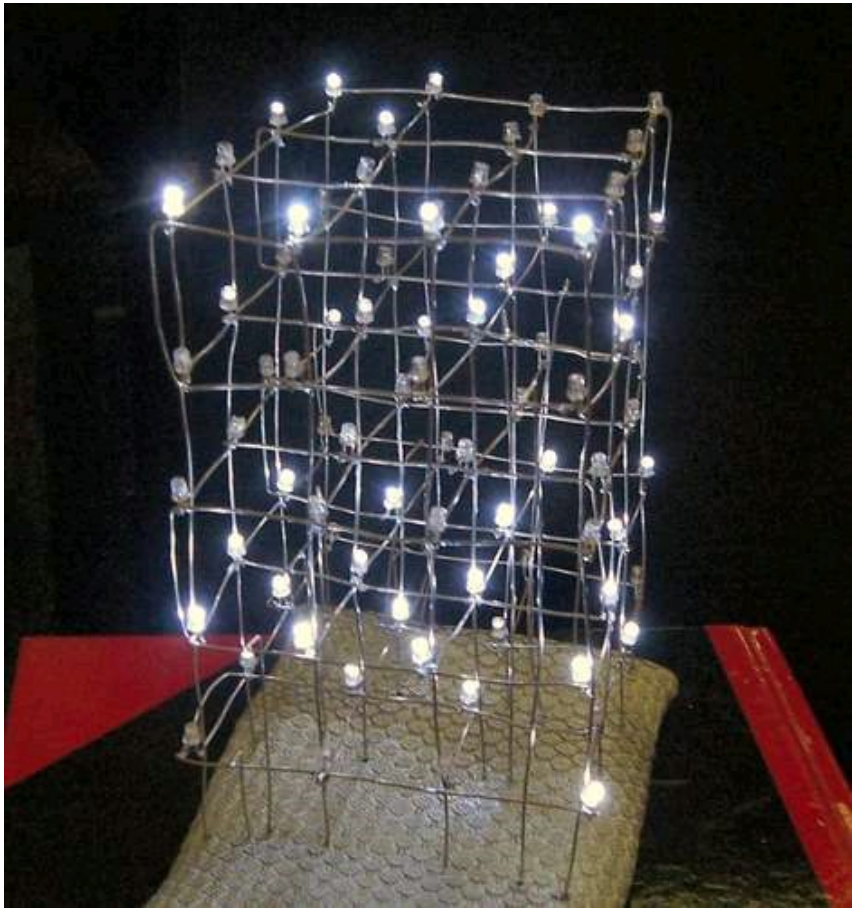
  w: e2-e4 s: e7-e5

  w: g1-f3 s: b8-c6

  w: f1-b5



Beispiel für ein 3-dimensionales Array



- In dem 3D LED Cube links sind die Positionen der einzelnen LEDs durch drei Größen (Breite, Höhe, Tiefe) eindeutig.
- Arrays, in denen die Positionen von Objekten durch drei Größen eindeutig sind, heißen auch 3-dimensionale Arrays (wer hätte das gedacht ☺)

Ein 3D LED Cube als 3-dimensionales Array in Ruby

- Einen leeren 3D Cube in den Dimension *(breite x hoehe x tiefe)* erzeugen.

```
# 3-dim (breite x hoehe x tiefe) Array
breite, hoehe, tiefe = 4
cube =
    Array.new(breite){Array.new(hoehe){Array
    .new(tiefe)}}
```

- LEDs erzeugen.

```
# LED's erzeugen insgesamt 64
led000 = "led111"
led001 = "led112"
led333 = "led444"
# ... etc
```

- Cube mit LEDs füllen.

```
# cube fuellen
cube[0][0][0] = led000
cube[0][0][1] = led001
cube[3][3][3] = led333
# ... etc
```

Ein 3D LED Cube als 3-dimensionales Array in Ruby

- Anstelle einzelner LEDs können wir auch Scheiben von LEDs aus dem Cube lesen oder in diesen schreiben.

```
# vertikale Scheiben des Cubes
# sind 2-dim Arrays
# Array.new(hoehe){
#       Array.new(tiefe)}
cube[2]
```

- Zum Schreiben verwenden wir dieses Mal ***2-dimensionale Array-Literale***

```
# vertikale Scheiben mit
# Array-Literalen setzen
cube[2] =
        [[led200,led201,led202,le
   d203],

   [led210,led211,led212,led213],

   [led220,led221,led222,led223],
         [led230,led231,led232,
   led233]]
```

Ein 3D LED Cube als 3-dimensionales Array in Ruby

- Anstelle ganzer Scheiben von LEDs können wir einzelne Reihen von LEDs aus dem Cube lesen oder in diesen schreiben.

```
# vertikale Reihen des Cubes
# sind 1-dim Arrays
# Array.new(tiefe)
cube[2][3]
```

- Zum Schreiben verwenden wir nun ***1-dimensionale Array-Literale***

```
# vertikale Reihen mit
# Array-Literalen setzen
cube[2][3] =
    [led230,led231,led232,led233]
```

Array Methoden in Ruby

Kategorie	Methoden
Erzeugen	Literal, Array.new
Elementzugriff	[], first, last, slice
Elementzuweisung	[]=,
Gleichheit	==, eql?, equal?, (hash), <=>,
Elementtest und -index	index, rindex, include?
Stack-, Queue-operationen ()	<< , push, pop, shift , unshift
Konkatenieren	+, concat
Löschen / Einfügen	clear, delete, insert, delete_at, slice!, delete_if,
Inhalt ersetzen	replace, fill
Umkehren	reverse, reverse!
Flach“klopfen“	flatten, flatten!
Abfragen	empty?, length, size

Array Methoden in Ruby

Kategorie	Methoden
Iteratoren	each, each_index, reverse_each select, collect(!), inject, map(!), reject(!), values_at

Array - Elementzugriff

Methode	Beispiele
array[index] → obj or nil **array[start, length] → an_array or nil** **array[range] → an_array or nil** **array.slice(index) → obj or nil** **array.slice(start, length) → an_array or nil** **array.slice(range) → an_array or nil** Liefert das Element auf Position *index* oder ein Teilarray von Position *start* mit *length* Elementen, oder eine Teilarray, dass durch *range* definiert ist. Bei negativen Indizes wird vom Ende des Arrays gezählt. (-1 ist der letzte Index). Liefert *nil*, wenn ein Index ausserhalb der Größe des Arrays liegt.	a = ["a", "b", "c", "d", "e"] a[2] + a[0] + a[1] #=> "cab" a[6] #=> nil a[1, 2] #=> ["b", "c"] a[1..3] #=> ["b", "c", "d"] a[4..7] #=> ["e"] a[6..10] #=> nil a[-3, 3] #=> ["c", "d", "e"] # Spezialfälle a[5] #=> nil a[5, 1] #=> [] a[5..10] #=> []

Array - Elementzugriff

Methode	Beispiele
array.first → obj or nil **array.first(n) → an_array** Liefert das erste Element oder die ersten n Elemente als Teilarray. **array.last → obj or nil** **array.last(n) → an_array or nil** Liefert das letzte Element oder die letzten n Elemente als Teilarray Wenn array leer ist liefert die erste Form nil, die zweite Form ein leeres Array	a = ["q", "r", "s", "t"] a.first #=> "q" a.first(1) #=> ["q"] a.first(3) #=> ["q", "r", "s"] a.first(10) #=> ["q", "r", "s", "t"] a = ["q", "r", "s", "t"] a.last #=> “t" a.last(1) #=> [“t"] a.last(3) #=> ["r", "s“,”t”] a.last(10) #=> ["q", "r", "s", "t"]

Array - Elementzuweisung

Methode	Beispiele
array[index] = obj → obj **array[start, length] = obj oder an_array oder nil** **→ obj oder an_array oder nil** **array[range] = obj oder an_array oder nil** Setzt das Element auf dem *index* auf *obj* oder ersetzt ein Teilarray, von Position *start* bis *start + length* in *array* oder ein Teilarray im Bereich *range* in *array.* Die Größe des Arrays wird dabei automatisch angepasst (verkleinert, vergrößert). Negative Indizes werden vom Ende gezählt. Wenn *length== 0,* wird ein Element eingefügt. Wenn nil in der zweiten und dritten Form genutzt wird, wird ein Element (Elemente) entfernt. *IndexError* wenn ein negativer Index über den Anfang des Arrays hinausgeht.	a = Array.new a[4] = "4" a #=> [nil, nil, nil, nil, "4"] a[0, 3] = ['a', 'b', 'c'] a #=> ["a", "b", "c", nil, "4"] a[1..2] = [1, 2] a #=> ["a", 1, 2, nil, "4"] a[0, 2] = "?" a #=> ["?", 2, nil, "4"] a[0..2] = "A" a #=> ["A", "4"] a[-1] = "Z" a #=> ["A", "Z"] a[1..-1] = nil a #=> ["A"]

Gleichheit von Arrays

Methode	Definition	Beispiel
ary1 == ary2	ary1.size() == ary2.size(), und es gilt dass ary1[i] == ary2[i] für alle i = 0 .. ary1.size-1	a1 = [1.0, 2, 3.0] a2 = [1, 2, 3] a1 == a2 #=> true
ary1.eql?(ary2)	ary1.size() == ary2.size(), und es gilt dass ary1[i] == ary2[i] für alle i = 0 .. ary1.size-1	a1.eql?(a2) #=> false
ary1.equal?(ary2)	ary1 identisch ary2, ary1, ary2 zeigen auf das selbe Objekt	a1.equal?(a2) #=> false a3 = a1 a3 == a1 #=> true
ary1 <=> ary2	0, wenn ary1== ary2 -1, wenn ary1 < ary2 1, wenn ary1 > ary2	a1 <=> a2 #=> 0 a3 = [1.0, 2.7, 3.0] a4 = [1.0, 2, 2.6] a1 <=> a3 #=> -1 a1 <=> a4 #> 1

Elementtest und –index für Arrays

Methode	Beispiel
array.index(obj) → int or nil Liefert den index des ersten Objektes in array, das == obj ist. Liefert nil wenn obj nicht enthalten ist.	a = ["a", "b", "c"] a.index("b") #=> 1 a.index("z") #=> nil
array.rindex(obj) → int or nil Liefert den Index des letzten Objektes in array, das == obj ist. Liefert nil wenn obj nicht enthalten ist.	a = ["a", "b", "b", "b", "c"] a.rindex("b") #=> 3 a.rindex("z") #=> nil
array.include?(obj) → true oder false true, wenn obj in array enthalten ist (Test mit ==), sonst false	a = ["a", "b", "c“,1,3.0] a.include?("b") #=> true a.include?("z") #=> false a.include?(3) #=> true

Arrays konkatenieren

Methode	Beispiel
array + other_array → an_array Liefert ein neues Array, das durch Anhängen von other_array an array entsteht **array - other_array → an_array** Differenz zweier Arrays. Liefert ein neues Array, das alle Elemente von array abzüglich denen in other_array enthält. Keine Mengendifferenz.	a1 =[1, 2, 3] a2 = [4, 5] a1 + a2 #=> [1, 2, 3, 4, 5] a1 #=> [1, 2, 3] [1, 1, 2, 2, 3, 3, 4, 5] - [1, 2, 4] #=> [3, 3, 5]
array.concat(other_array) → array Hängt other_array an array an. Verändert array.	a1.concat(a2) #=> [1, 2, 3, 4, 5] a1 #=> [1, 2, 3, 4, 5]

Stack-Queue Operationen für Array

Methode	Beispiel
array << obj → array **array.push(obj, ...) → array** Fügt obj / mehrere obj am Ende von array ein. Liefert das modifizierte array zurück. **array.pop → obj or nil** Entfernt und liefert das letzte Element aus array, modifiziert array. Liefert nil, wenn array leer ist .	a = ["a", "b", "c"] a << “d” << “e” #=> ["a", "b", "c", "d", "e"] a.push(“f", “g", “h") #=> ["a", "b", "c", "d", "e", "f“, “g”, “h”] a #=> ["a", "m“, “z”] a.pop a.pop.pop #=> “a” a #=> []
array.shift → obj or nil Entfernt und liefert das erste Element aus array, nil wenn array leer ist. **array.unshift(obj, ...) → array** Fügt Objekte am Anfang von array ein.	args = ["-m", "-q", "filename"] args.shift #=> "-m" args #=> ["-q", "filename"] a = ["b", "c", "d"] a.unshift("a") #=> ["a", "b", "c", "d"] a.unshift(1, 2) #=> [1, 2, "a", "b", "c", "d"]

Lösch / Einfüge Operationen für Array

Methode	Beispiel
array.clear → array Löscht alle Elemente aus array	a = ["a", "b", "b", "b", "c"] a.clear #=> []
array.delete(obj) → obj or nil Löscht Elemente aus array, die == *obj* sind. Liefert nil, wenn obj nicht enthalten sind.	a = ["a", "b", "b", "b", "c"] a.delete("b") #=> "b" a #=> ["a", "c"] a.delete("z") #=> nil
array.delete_at(index) → obj or nil Löscht und liefert das Element an Position *index.* Liefert nil, wenn der Index ausserhalb des arrays liegt.	a = ["ant", "bat", "cat“, "dog“] a.delete_at(2) #=> "cat" a #=> ["ant", "bat", "dog"] a.delete_at(99) #=> nil

Lösch / Einfüge Operationen für Array

Methode	Beispiel
array.slice!(index) → obj or nil **array.slice!(start, length)** **→ sub_array or nil** **array.slice!(range) → sub_array or nil** Löscht Elemente aus array, deren Bereich durch index, start,length oder range bestimmt wird. (Siehe Elementzugriff)	a = ["a", "b", "c"] a.slice!(1) #=> "b" a #=> ["a", "c"] a.slice!(-1) #=> "c" a #=> ["a"] a.slice!(100) #=> nil a #=> ["a"]
array.insert(index, obj...) → array Fügt die Werte vor dem Element mit dem gegebenen Index in array ein. Negative Indizes werden vom Ende des arrays gezählt.	a = ["a", "b", "c", "d"] a.insert(2, 99) #=> ["a", "b", 99, "c", "d"] a.insert(-2, 1, 2, 3) #=> ["a", "b", 99, "c", 1, 2, 3, "d"]

Nützliche Operationen für Array

Methode	Beispiel
array.reverse → an_array **array.reverse! → array**	["a", "b", "c"].reverse #=> ["c", "b", "a"] [1].reverse #=> [1]
array.flatten → an_array Erzeugt ein neues flaches Array. **array.flatten! → array or nil** Macht aus array ein flaches Array. Liefert nil wenn array bereits ein flaches Array ist.	s = [1, 2, 3] #=> [1, 2, 3] t = [4, 5, 6, [7, 8]] #=> [4, 5, 6, [7, 8]] a = [s, t, 9, 10] #=> [[1, 2, 3], [4, 5, 6, [7, 8]], 9, 10] a.flatten #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10 a = [1, 2, [3, [4, 5]]] a.flatten! #=> [1, 2, 3, 4, 5] a.flatten! #=> nil a #=> [1, 2, 3, 4, 5]
empty?, length, size	Prüft ob ein Array leer ist, Liefert die Länge eines Arrays.

Nützliche Operationen für Array

Methode	Beispiel
array.replace(other_array) → array Ersetzt den Inhalt von array durch other_array	a = ["a", "b", "c", "d"] b = [] b.replace(a) #=> ["a", "b", "c", "d"]
array.fill(obj) → array **array.fill(obj, start [, length]) → array** **array.fill(obj, range) → array** Ersetzt die Elemente in den spezifizierten Bereichen von array durch obj ein.	a.fill("x") #=> ["x", "x", "x", "x"] a.fill("z", 2, 2) #=> ["x", "x", "z", "z"] a.fill("y", 0..1) #=> ["y", "y", "z", "z"]

Methoden von Array selbst definieren

- Eine gute Art die Arbeitsweise der Methoden von Array kennenzulernen ist es, diese selbst mit den uns bis jetzt bekannten Methoden von Array zu implementieren.

- **Ü5-15: Implementieren Sie bitte die folgenden Methoden für Spezialfälle mit den Ihnen bisher bekannten „Bordmitteln“**
  - **reverse**
  - **push, pop**
  - **+, concat**
  - **fill**

reverse

- ***reverse*** dreht die Reihenfolge der Elemente in einem Array um. ***reverse*** arbeitet nicht destruktiv. Es gibt ein neues Array zurück.

```
def my_reverse_1(ary)
  # Länge von ary -1
  len = ary.length()-1
  yra = []
  ary.each_index{ |index|
    yra[index] = ary[len-index]
  }
  return yra
end

def my_reverse_2(ary)
  # Länge von ary -1
  len = ary.length()-1
  yra = []
  ary.each_index{ |index|
    yra << ary[len-index]
  }
  return yra
end
```

push/ pop

- ***push*** hängt ein Element hinten an ein Array an
- ***pop*** liefert das oberste (letzte) Element eines Arrays und entfernt es aus dem Array.

- Bsp.: ***[1,2,3,4].push(7)*** ergibt ***[1,2,3,4,7]***
  [1,2,3,4].pop() ergibt ***[1,2,3]***

+/ concat

- ***+/concat*** hängt ein Array hinten an ein anderes Array an.
- ***+*** arbeitet nicht destruktiv
- ***concat*** arbeitet destruktiv

- Bsp.: ***[1,2,3,4].concat([8,9])*** ergibt ***[1,2,3,4,8,9]***

fill

- ***fill*** befüllt ein Array mit Elementen. In der einfachsten Variante wird ein Argument übergeben.

- Bsp.: ***[1,2,3,4].fill("x")*** ergibt ***["x", "x", "x", "x"]***

Zusammenfassung

- **Sammlung**: Sammlungsobjekte sind Objekte, die eine beliebige Anzahl anderer Objekte enthalten können.
- **Iterator**: Ein Iterator ist eine Methode, die nacheinander die Elemente aus einer Sammlung holt und die Anweisungen im Iterator-Block auf diese Elemente anwendet. Dabei werden die einzelnen Elemente in jedem Schritt der Blockvariablen des Iterators zugewiesen.
- **Schleifen**: Schleifen sind Kontrollstrukturen, um einen Block von Anweisungen zu wiederholen. Schleifen benutzen Indizes, um auf Objekte einer Sammlung zuzugreifen.
- **Array**: Ein Array ist eine Sammlung von Objekten mit einer wohldefinierten Ordnung auf den enthaltenen Elementen. Die Ordnung der Objekte ist definiert über ihre Position im Array. Der Zugriff auf die Elemente erfolgt über ihren Positionsindex
- **Mehrdimensionale Arrays**: sind mehrdimensionale Sammlungen, wie Tabellen und Schachbretter, Würfel etc. Der Zugriff auf Elemente eines mehrdimensionalen Arrays erfolgt über einen kombinierten Index der verwendeten Dimensionen.
- **Intervalle**: sind Datenstrukturen, die einen endlichen Wertebereich definieren. Ruby kennt nach oben geschlossene und offene Intervalle.
- **Iteratoren für Array**: Array verfügt über eine Vielzahl von Iteratoren, die sich alle auf den Basisiterator ***each*** zurückführen lassen.

Quelle

David J. Barnes / Michael Kölling
ISBN: 978-3-8689-4907-0
672 Seiten
Erscheinungstermin: 1/2013
Ausstattung: 1 CD, 4-farbig
Sprache: Deutsch