Variable Parameteranzahl (Variadics)

Wenn Sie nicht im Vorhinein wissen, wie viele Parameter an eine Funktion übergeben werden, können Sie den letzten Parameter der Parameterliste in der Form `name:typ...` als sogenannten »variadischen Parameter« definieren. Die drei dem Typ folgenden Punkte gelten dabei als Kennzeichnung für den variadischen Parameter.

Beim Aufruf können Sie anstelle dieses Parameters beliebig viele Daten im vorgesehenen Typ übergeben (auch gar keine). Innerhalb der Funktion sprechen Sie den Parameter einfach als Array an.

Die folgende Funktion ermittelt die kleinste der übergebenen Fließkommazahlen. Die Funktion gibt `nil` zurück, wenn die Parameterliste leer ist.

```
// ermittelt die kleinste der übergebenen Double-Zahlen
func min(data:Double...) -> Double? {
  if data.count==0 { return nil }
  var result = data[0]
  for d in data {
    if d<result { result=d }
  }
  return result
}

min()                    // nil
min(4.0, 1.0, 2.0, 3.0)  // 1.0 als Optional
```

6.3 Standardfunktionen

In Swift stehen rund 35 sogenannte Standardfunktionen zur Verfügung. Sie sind ein integrierter Bestandteil von Swift und können selbst dann genutzt werden, wenn Sie keine einzige externe Bibliothek mit `import` in Ihr Projekt einbeziehen. In diesem Abschnitt stelle ich Ihnen die wichtigsten Standardfunktionen (siehe Tabelle 6.1) beispielorientiert vor. Diese Funktionen kennenzulernen lohnt sich: Sie sind in der alltäglichen Swift-Programmierpraxis nahezu unverzichtbar.

Die Standardfunktionen werden vielfach auch als »globale Funktionen« bezeichnet, weil sie global in jedem Swift-Code zur Verfügung stehen. Das gilt aber natürlich auch für alle selbst definierten Funktionen.

Falls Sie erste Swift-Erfahrungen schon mit Version 1 gemacht haben, wird Ihnen sicher auffallen, dass es in Swift 2 wesentlich weniger Standardfunktionen gibt. Diese Funktionen wurden nicht einfach ersatzlos gestrichen, sondern stehen jetzt größtenteils als Methoden zur Verfügung. Möglich machte das eine grundlegende Syntax-

erweiterung in Swift 2: Mit Protokollerweiterungen (*Protocol Extensions*) können Protokolle nachträglich durch Defaultmethoden ergänzt werden (siehe Abschnitt 8.6).

Referenz aller Standardfunktionen

Eine Auflistung aller Standardfunktionen erhalten Sie, wenn Sie in Xcode bei gedrückter ⌘-Taste auf einen beliebigen Namen einer Standardfunktion klicken. Xcode zeigt Ihnen dann die Deklaration aller in Swift verfügbaren Datentypen, Protokolle, Klassen, Operatoren und eben auch Funktionen. Übersichtlicher und besser lesbar werden diese Informationen auf der folgenden Webseite präsentiert:

http://swiftdoc.org

Dort scrollen Sie zum Ende der Startseite bis zur Überschrift GLOBALS/FUNCTIONS.

Funktion	Bedeutung
`abs(x)`	Liefert den Absolutbetrag.
`max(a, b, c, ...)`	Liefert das größte Element.
`min(a, b, c, ...)`	Liefert das kleinste Element.
`print(s)`	Gibt `s` und einen Zeilenumbruch aus.
`readLine()`	Liest eine Zeichenkette von der Standardeingabe.
`sizeof(type)`	Ermittelt den Speicherbedarf eines Datentyps.
`swap(&var1,&var2)`	Vertauscht die Inhalte von `var1` und `var2`.
`zip(seq1, seq2)`	Bildet Tupel-Paare.

Tabelle 6.1 Die wichtigsten Standardfunktionen von Swift

min und max

`min` und `max` ermitteln den kleinsten bzw. größten Wert aus allen übergebenen Einzeldaten. Alle übergebenen Parameter müssen denselben Datentyp aufweisen.

```
min(1, 2, 3)  // 1
max(1, 2, 3)  // 3
```

`min` und `max` sind nicht für Arrays geeignet. Um die Extremwerte in einem Array oder einer anderen Sequenz zu ermitteln, setzen Sie die Methoden `min-` bzw. `maxElement` ein:

```
Array(1...10).minElement()  // 1
"Hello World".characters.maxElement()  // "r"
```

print und readline

Die vertraute Funktion `print` gibt den als Parameter übergebenen Ausdruck als Zeichenkette aus. Das funktioniert innerhalb von Xcode bzw. bei Projekten vom Typ Command Line Tool:

```
print(1, 2, 3)
// Ausgabe: 1 2 3

print("Hello", "World!")
// Ausgabe: Hello World!
```

Standardmäßig stellt `print` zwischen die Parameter ein Leerzeichen und beendet die Ausgabe mit einem Zeilenumbruch. Alternativ können Sie mit den optionalen Parametern `separator` und `terminator` andere Trenn- bzw. Abschlusszeichenketten einstellen:

```
print(1, 2, 3, separator:"")   // Ausgabe: 123
print("Hello ", terminator:"") // Hier kein Zeilenumbruch!
print("World!")                // Ausgabe: Hello World!
```

Eine leere Zeile geben Sie mit `print("")` aus. Vorsicht: `print()` ohne Parameter liefert in der aktuellen Swift-Version eine Fehlermeldung.

print im Playground

An sich funktioniert `print` natürlich auch im Playground. Ich bin aber in Xcode 7.0 und 7.1 über mehrere Fälle gestolpert (immer im Zusammenhang mit Closures), bei denen `print` keine Ausgaben lieferte. Derselbe Code funktionierte in einem gewöhnlichen Projekt einwandfrei. Hier liegt offensichtlich noch ein Bug vor.

`print(..., terminator:"")` lässt sich im Playground grundsätzlich nicht sinnvoll nutzen. Wenn Sie mehrere Ausgaben in einer Zeile zusammenfassen möchten, müssen Sie Ihren Code in einem gewöhnlichen Projekt testen.

`readLine` liest eine Zeile Text von der Standardeingabe. Damit eignet sich die Funktion zur Verarbeitung von Texteingaben in Terminal-Programmen. `readLine` ist gewissermaßen die Umkehrfunktion zur `print`-Funktion, die an die Standardausgabe schreibt.

swap

`swap` vertauscht den Inhalt von zwei Variablen oder Array-Elementen. Beachten Sie, dass Sie den Variablen das Zeichen `&` voranstellen müssen! Es ist ein Indikator für `inout`-Parameter.

Vorsicht

Wenn Sie an `swap` zweimal den gleichen Parameter übergeben, wird ein Fehler ausgelöst. Während `swap(&a, &a)` offensichtlich sinnlos ist, kann es durchaus zweckmäßig sein, zwei Array-Elemente mit `swap(&x[n], &x[m])` zu vertauschen. Dabei müssen Sie sicherstellen, dass `n` und `m` immer unterschiedliche Indizes sind!

zip

`zip` verknüpft die Elemente aus zwei Sequenzen paarweise zu Tupeln. Das Ergebnis ist eine Datenstruktur vom Typ `Zip2Sequence`. Wenn die beiden Sequenzen unterschiedlich viele Elemente aufweisen, dann bestimmt die kürzere Sequenz die Anzahl der Ergebnis-Tupel.

```
// Messzeiten und Messwerte in zwei Arrays
let time =  ["12:15", "12:30", "12:45", "13:00"]
let temp = [20.9, 20.8, 20.7, 20.9]
let combined = zip(time, temp)
```

Die resultierende Struktur können Sie unkompliziert in einer Schleife durchlaufen:

```
for (tm, tmp) in combined {
  print("Zeit \(tm)  -- Temperatur \(tmp)")
}
```

Mathematische Funktionen

Mit Ausnahme von `abs` fehlen in den Standardfunktionen mathematische Grundfunktionen, wie `sqrt`, `sin` oder `cos`. Dennoch können Sie auch diese Funktionen in jedem Programm verwenden. Woher kommen also diese Funktionen?

Die mathematischen Grundfunktionen sind in der Bibliothek `libSystem` definiert, die wiederum von den Bibliotheken Foundation, Cocoa und UIKit importiert wird. Beachten Sie, dass Sie an die meisten Funktionen Fließkommazahlen übergeben müssen. `sqrt(2)` funktioniert nicht, es muss `sqrt(2.0)` heißen. In der Darwin-Bibliothek sind auch einige Konstanten definiert, darunter die Kreisteilungszahl `M_PI` und die eulersche Zahl `M_E`.

```
sqrt(2.0)          // 1,4142135623731
cos(0.0)           // 1,0
pow(10.0, 3.0)     // 1000
M_PI               // 3,14159265358979
M_E                // 2.71828182845905
```

6.4 Standardmethoden und Standardeigenschaften

»Standardmethoden« und »Standardeigenschaften« gibt es in der offiziellen Dokumentation zu Swift nicht. Dennoch habe ich mir die Freiheit genommen, unter dieser Überschrift einige allgegenwärtige Methoden und Eigenschaften vorzustellen, die auf elementare Swift-Datentypen angewendet werden können. Dazu zählen z. B. `count`, `filter` oder `map` (siehe Tabelle 6.2).

Methode	Bedeutung
`pos.advancedBy(n)`	Ermittelt eine neue Position.
`data.contains(item)`	Testet, ob das Element in der Sequenz enthalten ist.
`data.count`	Ermittelt die Anzahl der Elemente.
`start.distanceTo(end)`	Ermittelt den Abstand zwischen Start- und Endindex.
`data.dropFirst()`	Liefert eine neue, um ein Element verkleinerte Sequenz.
`data.dropLast()`	Liefert eine neue, um ein Element verkleinerte Sequenz.
`data.filter(func)`	Liefert die Elemente, die der Funktion entsprechen.
`data.first`	Liefert das erste Element.
`data.flatMap(func)`	Wendet eine Funktion auf alle Elemente an.
`data.forEach(func)`	Wendet eine Funktion auf alle Elemente an.
`data.indexOf(item)`	Sucht das angegebene Element in der Sequenz.
`d.joinWithSeparator(s)`	Verbindet die Elemente einer String-Sequenz.
`data.last`	Liefert das letzte Element.
`data.map(func)`	Wendet eine Funktion auf alle Elemente an.
`data.maxElement()`	Liefert das größte Element.
`data.minElement()`	Liefert das kleinste Element.
`data.prefix(n)`	Liefert die ersten `n` Elemente.
`data.reduce(start, func)`	Wendet die Funktion paarweise auf die Elemente an.
`data.reverse()`	Liefert eine neue Sequenz in inverser Reihenfolge.

Tabelle 6.2 Wichtige Methoden und Eigenschaften für Arrays und Collections

Methode	Bedeutung
`data.sort(<)`	Liefert ein neues sortiertes Array.
`data.sortInPlace(<)`	Sortiert die vorhandenen Array-Elemente.
`data.split()`	Zerlegt die Sequenz in Array-Elemente.
`data.startsWith(seq)`	Testet, ob die Anfangselemente übereinstimmen.
`data.suffix(n)`	Liefert die letzten `n` Elemente.

Tabelle 6.2 Wichtige Methoden und Eigenschaften für Arrays und Collections (Forts.)

Von der globalen Funktion zur Methode

Viele Methoden bzw. Eigenschaften, die in diesem Abschnitt vorgestellt werden, waren in Swift 1.n als Funktionen implementiert. Eine der wesentlichsten Spracherweiterungen in Swift 2.0 bestand nun aber darin, Protokolle so wie Klassen erweiterbar zu machen (*Protocol Extensions*). Damit besteht nun die Möglichkeit, ein Protokoll, das von unterschiedlichen Typen implementiert wird, nachträglich um Methoden zu erweitern.

Genau von dieser Funktionalität machte Apple Gebrauch, um ehemals globale Funktionen wie `count` oder `map` beginnend mit Swift 2.0 als Methoden zu implementieren. Die meisten hier vorgestellten Methoden erweitern das Protokoll `CollectionType` und stehen somit für alle Datentypen zur Verfügung, die `CollectionType` implementieren. Dazu zählen unter anderem Arrays, Dictionaries und `String.CharacterView`, also der Datentyp, den Sie erhalten, wenn Sie die Eigenschaft `characters` auf eine `String`-Variable anwenden.

Da wir in diesem Buch die objektorientierte Programmierung bisher noch nicht richtig behandelt haben (siehe das folgende Kapitel!), folgt hier noch kurz eine Erklärung, worin sich Funktionen von Methoden und Eigenschaften unterscheiden: An Funktionen müssen alle zu verarbeitenden Daten als Parameter übergeben werden – z. B. `print(data)`. Eigenschaften und Methoden werden hingegen auf die Daten angewendet – also `data.count` bzw. `data.find(item)`.

count

Die Eigenschaft `count` ermittelt die Anzahl der Elemente von Arrays, Zeichenketten (mit `.characters`), Bereichen etc.:

```
let ar = [1, 2, 3, 4, 5, 6]
let s = "Hello World!"
```

```
let rng = 1...10
ar.count                // 6
s.characters.count      // 12
rng.count               // 10
```

first und last

Die Eigenschaften `first` und `last` liefern das erste bzw. letzte Element einer Sequenz als Optional. Gegebenenfalls müssen Sie durch ein nachgestelltes Ausrufezeichen das Unwrapping erzwingen. Wenn die Sequenz leer ist, liefern die Funktionen `nil` zurück. Die Sequenz bleibt in jedem Fall unverändert.

```
var ar = [1, 2, 3, 4]
var s = "Hello World!"
ar.first            // 1, Datentyp Int?
s.characters.last   // "!", Datentyp Character?
```

prefix und suffix

`prefix` und `suffix` liefern die ersten bzw. letzten *n* Elemente einer Sequenz. Wenn *n* größer als die Elementanzahl ist, liefern die Funktionen einfach alle Elemente. Ist die Sequenz bzw. das Array leer, ist konsequenterweise auch das Ergebnis leer. Der Rückgabedatentyp ist bei Arrays ein `ArraySlice` (also ein Teil-Array).

```
var ar = [1, 2, 3, 4]
let sub1 = ar.prefix(2)       // [1, 2], Datentyp ArraySlice<Int>
let sub2 = ar.suffix(2)       // [3, 4], Datentyp ArraySlice<Int>
var x = [Int]()
let sub3 = x.prefix(2)        // leer, weil auch x leer ist
```

dropFirst und dropLast

`dropFirst` und `dropLast` liefern eine neue Sequenz, die sich aus der um das erste oder letzte Element verminderten Ausgangsmenge ergibt. Die übergebenen Daten ändern sich nicht.

```
var ar = [1, 2, 3, 4]
dropFirst(ar)     // [2, 3, 4], Datentyp [Int]
dropLast(ar)      // [1, 2, 3], Datentyp [Int]

var s = "Hello World!"
String(dropFirst(s.characters))   // "ello World!"
String(dropLast(s.characters))    // "Hello World"
```

startsWith, contains und indexOf

Die Methode `startsWith` testet, ob eine Sequenz mit einer vorgegebenen Aufzählung von Elementen beginnt. Merkwürdigerweise gibt es keine entsprechende `endsWith`-Methode.

```
let ar = [1, 2, 3, 4, 5, 6]
ar.startsWith([1, 2])       // true
ar.startsWith([4, 5])       // false
```

`contains` überprüft, ob ein Wert in der Sequenz enthalten ist. Das Ergebnis lautet `true` oder `false`. Dabei kann an die Methode sowohl ein einzelnes Element übergeben werden als auch eine Funktion (eine Closure) mit einem Suchausdruck. Die beiden folgenden Beispielaufrufe ergeben beide `true`, weil das Array sowohl die Zahl 3 als auch gerade Zahlen enthält:

```
let ar = [1, 2, 3, 4, 5, 6]
ar.contains(3)                  // true
ar.contains( {$0 % 2 == 0} )    // true
```

`indexOf` durchsucht eine Sequenz nach dem ersten Auftreten eines Elements. Anstelle eines Suchwerts kann auch eine Funktion (eine Closure) angegeben werden, die `true` ergeben muss. Das Ergebnis ist ein Optional mit der Indexnummer zum Zugriff auf das Element bzw. `nil`, wenn die Suche ergebnislos bliebt. Beachten Sie, dass die Index-Nummerierung mit 0 beginnt – d. h., wenn ein Element an der ersten Position gefunden wird, lautet das Ergebnis 0.

```
let data = Array(1...10)
data.indexOf(7)                 // 6, Datentyp Index?
data.indexOf(12)                // nil
data.indexOf( {$0 % 2 == 0} )   // 1, Datentyp Index?
```

Es ist nicht möglich, die Startposition der Suche anzugeben. Daher eignet sich die Funktion nicht, um mehrere gleiche Elemente in der Sequenz zu finden. Eine Funktion, die die Sequenz beginnend mit dem letzten Element durchsucht, fehlt ebenfalls.

Zeichenketten durchsuchen

`indexOf` eignet sich auch für Zeichenketten, solange Sie nach einem einzigen Zeichen suchen (`s.characters.indexOf('x')`). Deutlich flexibler ist die in Abschnitt 3.3, »Zeichenketten«, bereits erwähnte Methode `rangeOfString` aus dem Foundation-Framework, die als Suchobjekt ganze Zeichenketten zulässt, einen Parameter für die Startposition aufweist und sogar rückwärts suchen kann.

advancedBy und distanceTo

Die in Abschnitt 3.3, »Zeichenketten«, schon vorgestellten Methoden `advancedBy` und `distanceTo` verarbeiten `Index`- bzw. `RandomAccessIndexType`-Elemente. Diese kommen in der Praxis am häufigsten bei der Bearbeitung von Zeichenketten vor. Auf den ersten Blick scheinen die Funktionen `distanceTo` und `advancedBy` einfach nur Varianten zu den Operatoren Minus und Plus zu sein:

```
2.distanceTo(7)  // Abstand von 2 nach 7:  5 (Datentyp Int)
7.distanceTo(2)  // Abstand von 7 nach 2: -5
2.advancedBy(5)   // von Position 2 um 5 Schritte weiter: 7
```

Bemerkenswert an den beiden Methoden ist der Umstand, dass diese nicht nur mit ganzen Zahlen rechnen können, sondern auch mit Zeigern (Indizes) auf Zeichenketten (Datentyp `Index`).

```
let s = "Hello World!"
let start = s.startIndex      // 0, Datentyp Index
let end = s.endIndex          // 12, Datentyp Index

// Position des ersten Leerzeichens ermitteln
let pos1 = s.characters.indexOf(" ") ?? end
pos1                          // 5, Datentyp Index
// pos1 - start               // Fehler, - kann mit
//                               Index nicht rechnen
start.distanceTo(pos1)         // 5, Datentyp Int

// zuerst ein Zeichen weiter, dann noch drei Zeichen
let pos2 = pos1.advancedBy(1)   // 6, Datentyp Index
let pos3 = pos2.advancedBy(3)   // 9, Datentyp Index
s[pos2..<pos3]                  // "Wor", Datentyp String
```

Fehlergefahr bei »advancedBy«

`advancedBy` löst einen Fehler aus, wenn Sie versuchen, damit eine Position vor dem Anfang bzw. hinter dem Ende einer Zeichenkette zu berechnen!

split und joinWithSeparator

`split` zerlegt eine Sequenz in kleinere Teile und liefert das Ergebnis als Array zurück. Die Trennung erfolgt an Positionen, bei denen ein Element eine als Funktion formulierte Bedingung erfüllt. `split` erwartet bis zu drei Parameter, wobei der erste und der zweite Parameter optional sind:

- Der erste Parameter, `maxSplit`, bestimmt die Maximalanzahl der resultierenden Array-Elemente.
- Der zweite Parameter, `allowEmptySlices`, gibt an, ob die Funktion bei einem mehrfachen Vorkommen des Trennzeichens leere Array-Elemente erzeugen soll. Standardmäßig ist das nicht der Fall.
- Im dritten Parameter, `isSeparator`, erwartet `split` die Trennfunktion. Wenn Sie die Funktion nicht als Closure hintanstellen, müssen Sie den Parameternamen angeben, also `data.split(isSeparator: {$0 == " "})`.

Wenn `split` auf ein Array angewendet wird, liefert es als Ergebnis ein Array von Arrays. Im folgenden Beispiel gelten 0-Elemente als Teiler:

```
let data = [1, 2, 0, 5, 6, 4, 0, 0, 3, 0, 2]
let splitted = data.split {$0==0}
// Datentyp [ArraySlice<Int>]
splitted  // [[1, 2], [5, 6, 4], [3], [2]]
```

Die zu `split` umgekehrte Funktion bietet die Methode `joinWithSeparator`. Der Datentyp des Ergebnisses lässt das Herz von Generics-Fans höher schlagen (siehe Abschnitt 8.2, »Generics«). Glücklicherweise ist eine Umwandlung in ein normales Array problemlos möglich:

```
let joined = splitted.joinWithSeparator([0])
// Datentyp JoinSequence<Array<ArraySlice<Int>>>
let newdata = Array(joined) // [1, 2, 0, 5, 6, 4, 0, 0, 3, 0, 2]
```

Im zweiten Beispiel wird ein Array von Zeichenketten zu einer langen Zeichenkette zusammengesetzt:

```
let s = ["e4", "e5", "c7", "c6"].joinWithSeparator(";")
// s = "e4;e5;c7;c6"
```

filter, map und reduce

`filter`, `map` und `reduce` sind gewissermaßen drei »klassische« Funktionen bzw. Methoden zur Listenverarbeitung. Im Zusammenhang mit Arrays habe ich Ihnen die drei Methoden bereits vorgestellt. Hier finden Sie nochmals zu jeder Funktion ein Beispiel.

`filter` liefert ein Array mit den Elementen der Ausgangsdaten, die eine Bedingung erfüllen. Die folgenden Zeilen filtern aus den ganzen Zahlen zwischen 1 und 100 diejenigen heraus, die größer als 10, kleiner als 50 und durch 3 teilbar sind. Die Bedingung ist als Closure formuliert (siehe Abschnitt 6.6).

```
func testNumber(x:Int) -> Bool {
  if x>10 && x<50 && (x % 3 == 0) {
    return true
  }
  return false
}

var data = Array(1...100)
let result = data1.filter( { testNumber($0) } )
result     // [12, 15, 18, ..., 48]
```

`map` wendet eine Funktion auf alle Elemente an und liefert die Ergebnisse als Array. Die im folgenden Beispiel definierte Funktion `f` erzeugt für jede als Parameter übergebene Integer-Zahl eine Zeichenkette aus ebenso vielen Sternen:

```
let data = [2, 5, 4]
func f(n:Int) -> String {
  var result=""
  for _ in 1...n { result += "*" }
  return result
}
data.map(f)  // ["**", "*****", "****"]
```

Zu `map` existiert die Variante `flatMap`. Sie eignet sich vor allem dann, wenn die `map`-Methode selbst Arrays verarbeitet. `map` liefert dann verschachtelte Arrays. `flatMap` bildet aus den Ergebniselementen hingegen ein einziges, eindimensionales Array:

```
var ar = [1, 2, 3]
let nested = ar.map( { Array(1...$0) } )
// nested = [[1], [1, 2], [1, 2, 3]]
let flat = ar.flatMap( { Array(1...$0) } )
// flat = [1, 1, 2, 1, 2, 3]
```

`reduce` wendet eine Funktion paarweise auf die übergebenen Daten an, zuerst auf den Startwert und das erste Element, dann jeweils auf das Ergebnis und das nachfolgende Element. Im folgenden Beispiel werden drei Ziffern mit binärem Und verknüpft, wobei der Startwert `0xffff` lautet. Somit ist das Ergebnis auf Zahlen bis 65.535 limitiert.

```
let data = [0xff, 0xf0, 0x10]
let result = data.reduce(0xffff, combine: { $0 & $1 })
String(result, radix:16)  // Ergebnis hexadezimal "10"
```

forEach

Die Methode `forEach` wendet eine Funktion oder Closure auf alle Elemente einer Sequenz an. `forEach` gibt kein Ergebnis zurück. Im Vergleich zu einer Schleife mit `for ... in sequenz` können Sie `forEach` weder durch `break` noch durch `return` abbrechen.

Die folgenden Zeilen zeigen drei Anwendungsbeispiele für `forEach`: Der erste `forEach`-Ausdruck bildet eine Summe über ein `Double`-Array, der zweite Ausdruck gibt die Zeichen einer Zeichenkette einfach einzeln aus. Beim dritten Beispiel wurde die Closure zur besseren Lesbarkeit hinter `forEach` platziert, was die Anzahl der offenen Klammerebenen verringert. `UnicodeScalar` bildet aus den übergebenen Zeichencodes die entsprechenden Zeichen, und `print` gibt diese ohne Zeilenumbruch aus.

```
let ar = [2.7, 3.9, 1.6]
var sum = 0.0
ar.forEach( { sum += $0} )
print(sum)  // Ausgabe: 8,2

let s="abc"
s.characters.forEach( { print($0) } )
// Ausgabe: "a", "b", "c"

(65...68).forEach()
  { print(UnicodeScalar($0), terminator: "") }
print("")
// Ausgabe "ABCD"
```

In der Praxis bietet sich der Einsatz von `forEach` vor allem in Kombination mit anderen Funktionen an, die selbst wiederum Arrays oder Sequenzen zurückgeben. Der folgende Ausdruck ist aus Platz- und Übersichtsgründen über mehrere Zeilen verteilt, es handelt sich aber um *einen* Ausdruck. Dabei verarbeitet `filter` die Elemente von `data` und gibt ein neues, verkleinertes Array zurück, das nur die durch 3 teilbaren Zahlen enthält. `map` quadriert diese Zahlen und liefert ein weiteres Array. Dieses wird von `forEach` verarbeitet.

```
let data = Array(1...20)
data.filter( {$0 % 3 == 0} )
  .map( {$0*$0} )
  .forEach( {print($0)} )
// Ausgabe: 9, 36, 81, 144, 225, 324
```

forEach versus map und flatMap

Sowohl `forEach` als auch `map` und `flatMap` wenden jeweils eine Funktion oder Closure auf die Elemente einer Aufzählung oder eines Arrays an. Worin besteht nun der Unterschied?

`forEach` ruft die angegebene Funktion für jedes Element der Sequenz auf, ignoriert aber eventuell zurückgegebene Ergebnisse. `map` erstellt aus den Ergebnissen hingegen ein Array. Wenn die Ausgangsdaten selbst schon Arrays sind, entsteht dabei ein verschachteltes Array. Diesen Sonderfall vermeidet `flatMap`: Es erstellt ein »flaches«, eindimensionales Array.

sort und reverse

Die Methoden `sort` und `reverse` habe ich in Abschnitt 4.1, »Arrays«, schon vorgestellt. Beide Methoden liefern *neue* Arrays, die im folgenden Beispiel jeweils wieder zu einer Zeichenkette zusammengesetzt werden:

```
let s = "Hello World!"
String(s.characters.reverse())  // "!dlroW olleH"
String(s.characters.sort(<))    // " !HWdellloor"
```

Wenn Sie die Elemente eines vorhandenen Arrays verändern möchten, müssen Sie anstelle von `sort` die Methode `sortInPlace` verwenden:

```
var ar = [7, 3, 12, 2]
ar.sortInPlace(<)
print(ar)                       // [2, 3, 7, 12]
```

6.5 Funktionale Programmierung

Swift wird vielfach als funktionelle Programmiersprache bezeichnet. Ob das auch zutrifft, darüber lässt sich trefflich streiten. Richtig ist, dass Swift Funktionen als Variablenwerte, Parameter und Rückgabewerte akzeptiert. Zusammen mit einem reichlichen Angebot von Standardfunktionen und den sehr universell einsetzbaren Aufzählungstypen (Arrays, Dictionaries) können viele Algorithmen im Sinne der funktionalen Programmierung realisiert werden.

Andererseits stellt Apple im E-Book »The Swift Programming Language« die prozeduralen und objektorientierten Sprachmerkmale von Swift eindeutig in den Vordergrund. Das trifft auch für die überwiegende Mehrheit aller Beispielprogramme zu, die im Internet zu finden sind, ganz egal ob Sie »offizielle« Seiten von Apple oder andere Wissensportale konsultieren. In diesem Buch bevorzuge ich ebenfalls einen eher »traditionellen« Programmierstil.