Chapter 18
Bonus Chapter: Swift

Swift is a general-purpose programming language created by Apple Inc. for developing applications for Apple platforms. First released in 2014, it builds on the core of Apple's earlier Objective-C programming language, adding several modern features and, most importantly, removing low-level C concepts like the ubiquitous pointer constructs. In fact, it was promoted as "Objective-C without the C".

Although it currently plays a rather subordinate role in scientific computing, great potential is often attributed to it. In this chapter, we will outline the possibilities.

Swift is a compiled language. Compilers and required program libraries are available for Ubuntu Linux and partly for Windows. However, by far the easiest approach at present is to use the Xcode environment on the macOS operating system. In this chapter, we will follow this path.

The site `swift.org` is highly recommended as a starting point. You can find documentation and tutorials there, most notably The Swift Programming Language Book. The book is also available as an ePub version for free from the Apple Book Store. If you prefer a PDF version, you can find it on `appsdissected.com`. Users are expressly encouraged to participate in the development of the language.

Programming Environment

The most convenient way to experiment with Swift is to use an interactive Read-Eval-Print Loop (REPL), similar to the one in Julia. In Xcode you open a new Playground on which programs can be tested immediately. Working programs can then be compiled into stand-alone programs that can be run directly on the computer. All programs in this chapter have been tested on Xcode 14.3 running Swift version 5.8.

On non-macOS systems, the Swift extension for Visual Studio Code is often recommended. Again, details can be found on the `swift.org` website.

18.1 Basics

We start with the classical "Hello World" greeting.

Example 18.1 The program consists of a single line:

```
print("Hello World!")  // my first Swift program
```

In Swift, strings are enclosed in (double) quotes. Swift doesn't require you to put a semicolon after every statement in your code, although you can if you want. However, semicolons are required when you want to put multiple separate statements on one line. A comment block is closed in a pair '`/* */`'. Unlike C, comment blocks can also be nested.

Arithmetical Expressions

Swift provides integer and floating point data types represented by default in machine precision as `Int` and `Double`, which is typically 64 bits. We will use double-precision floating-point numbers throughout, but will continue to refer to them as floats for convenience.

Swift knows the basic arithmetic operators `+`, `-`, `*`, `/`, as well as '`%`' for the remainder in integer division. As in C, the result of a mixed expression consisting of integers and floats is again a float. The '`/`' operation applied to arguments of type `Int` denotes integer division. Binary operators either require spaces around both sides or neither side: `1 + 2` or `1+2` are correct, while `1+ 2` is not.

Boolean Operators, Comparisons

The Boolean constants are denoted by `true` and `false`, the Boolean operators as in C: '`!`' for negation, '`&&`' for and, and '`||`' for or. Comparisons and equality are denoted as in C.

Variables and Assignments

As in C, variables are statically typed:

```
var a: Int; var b: Double
```

Variables cannot be used before they have been assigned. Assignment and declaration can be combined. In this case the type annotation can be omitted as it is clear from the context:

```
var a: Int = 1  // is equivalent to
var a = 1
```

Swift has a special '`let`' notation for variables that remain constant after initialization:

```
let pi = 3.14
```

Arithmetic operations between variables (or constants) of different types are not allowed. In addition to the operators defined above, Swift provides composite assignment operators that combine an assignment with another operation. An example is the assignment operator for addition '`+=`':

```
var a = 1
a += 2  // a is now equal to 3
```

Output

As in Julia, *string interpolation* inserts the value of an expression '*expr*' into a string and replaces the placeholder '`\(`*expr*`)`'. Example:

```
let a = 40
print("The answer is \(a+2)")  // 42
```

By default, a print statement is terminated with a newline character '`\n`'. This behavior can be changed, for example to insert only a space after the output. Example:

```
print(4, terminator:" "); print(2)  // 4 2
```

Control Structures: Branching, Loops

Swift provides a variety of control flow statements. These include while loops, if statements, and for loops.

We first demonstrate the use of if statements and while loops using the Collatz problem discussed in Examples 3.3 and 6.2 in the Python and C chapters.

Example 18.2 (Collatz Problem) Here is a Swift version:

```
1 var n = 100
2 while n > 1 {
3   if n % 2 == 0 { n = n/2 }
4   else { n = 3*n + 1 }
5   print(n) }
```

As in C, statement blocks are enclosed in curly braces. In contrast to C, this also applies to single statements as in lines 3 and 4.

The if-else statement in lines 3 and 4 can again be replaced by a shorter form using a ternary operator as in C:

```
n = (n % 2 == 0) ? n/2 : 3*n + 1
```

We come to the for loop. Unlike C, Swift's for loop is a pure counter loop that works like in Python and Julia:

Example 18.3 We demonstrate the for loop using our Rice and Chessboard problem, already discussed in Example 2.4 in the Foundations chapter:

```
1 var grains: UInt = 0
2 var fieldval: UInt = 0
3 for fieldno in 1 ... 64 {
4   fieldval = (fieldno == 1) ? 1 : 2*fieldval
5   grains += fieldval }
6 print(grains)  // 18446744073709551615
```

In lines 1 and 2 we declare and initialize the larger integer types needed to store the grain numbers. In line 3, the counter variable `fieldno` is implicitly declared as an integer.

Interlude: Ranges

In line 3 above, we define a closed range `1...64`. It runs from 1 to 64, including the upper bound 64. In contrast, the half-open range consructor `..<` excludes the upper bound.

A countdown is also often useful in for loops:

```
let range = 1 ..<  5
for i in range.reversed() { print(i, terminator:" ") }  // 4 3 2 1
```

The range operators are not limited to integer ranges. They can also be used for floating point numbers: '`0.0 ... 1.0`' denotes the closed interval $[0,1]$, and '`0.0 ..< 1.0`' the half-open interval $[0,1)$.

Remark A finer range control can be achieved with

```
for x in stride(from: 5, through: 1, by: -2) {
  print(x, terminator:" ") }; print()  // 5 3 1
for y in stride(from: 5, to: 1, by: -2) {
  print(y, terminator:" ") }           // 5 3
```

However, we will not need this in the following.

We demonstrate the use of range operators for both integer and float ranges:

Example 18.4 Here is a Swift implementation of our Monte Carlo method for computing the number π, introduced in Sect. 4.9 in the SciPy chapter:

```
let samples = 100_000
var hits = 0
for _ in 1...samples {
  let x = Double.random(in: 0..<1)
  let y = Double.random(in: 0..<1)
  let d = x*x + y*y
  if d <= 1 { hits += 1 } }
print(4.0 * Double(hits) / Double(samples))  // 3.14844
```

Note the different meaning of the underscore in lines 1 and 3. In line 1 it is only for better readability. In line 3 it means that no index variable is needed inside the loop.

In lines 4 and 5, the `Double.random(in:)` method is used to generate random numbers in the interval $[0, 1)$. The dot notation suggests that `Double` is a class type. Actually, it is a struct type. We discuss both in a later section.

In line 8, we need to explicitly convert `hits` and `samples` to `Double` values.

As a final note on control structures, we mention that also in Swift, loops can be exited with `break` and `continue`. A `break` statement terminates the loop it is contained in, the `continue` statement is used to skip the rest of the code inside a loop for the current iteration only.

18.2 Functions

The core Swift language does not provide any numeric functions beyond the basic operators. However the Foundation framework contains a collection corresponding to the C Math library. Example:

```
import Foundation  // contains  among others numeric functions
print(sin(Double.pi / 2))  // 1.0
```

Note that the number `pi` is a member of the `struct Double`.

Remark A new comprehensive open source project for the Swift ecosystem, Swift Numerics, is under development, see for example `swift.org/blog/numerics` and `github.com/apple/swift-numerics`.

In the following, we will discuss how to create user defined functions.

Example 18.5 (Factorial) We illustrate the basic function concepts using a standard factorial function:

```
1 func fac(n: Int) -> Int {
2   var res = 1
3   for i in 1...n { res *= i }
4   return res }
5 print(fac(n: 5))  // 120
```

In line 1, the keyword `func` begins the definition of a function named `fac` with one integer argument `n`. The result type, also an integer, is indicated by the arrow symbol '`->`'.

Notice again in lines 2 through 4 that statement blocks must be enclosed in curly braces, even if they consist of a single statement.

In line 4 the accumulated value is returned in `res`.

Note that in line 5 the function is called in the form `fac(n: 5)`, where the *argument name* `n` is mentioned explicitly. In more complex definitions this can be useful to make the role of arguments easier to understand. If, as in our simple case here, we don't need this help, we can turn it off and just replace line 1 with

```
func fac(_ n: Int) -> Int  // call with fac(5)
```

The underscore means that the argument name is not mentioned. We can then call the function in the form

```
print(fac(5))  // 120
```

As a second example, we give a recursive variant of the factorial function:

Example 18.6 (Recursive Factorial)

```
func fac(_ n: Int) -> Int { return (n == 1 ? 1 : n*fac(n-1)) }
print(fac(5))  // 120
```

The output of a function is not limited to a single value. For example, it can be a so-called *tuple*, i.e. a comma-separated list of values enclosed in parentheses.

Example 18.7 Here is a function that swaps the values of two integer arguments:

```
1 func swap(_ x: Int, _ y: Int) -> (Int, Int) { return (y, x) }
2 let (a, b) = swap(1, 2)
3 print(a, b)  // 2 1
```

In line 1, the return value is specified as a tuple of `Int` values. In line 2, the result is assigned to the tuple `(a, b)`, where again the data types of `a` and `b` are determined by the context.

For later use we mention that the values in a tuple can be of any type, and do not need to be of same type.

Note that in Swift the arguments are passed by value. If we want a function that modifies its arguments, we must explicitly specify the variables as modifiable with the keyword `inout`:

Example 18.8 Here is a modifying version of the `swap` function:

```
1 func swap_mod(_ a: inout Int, _ b: inout Int) {
2   let tmp = a; a = b; b = tmp }
```

Note that `swap_mod` has no return value.

We test the function:

```
3 var a = 1, b = 2
4 swap_mod(&a, &b)  // access to inout variables requires &
5 print(a, b)       // 2 1
```

Note the ampersand in line 4. It reminds us of its use as an address operator in C. And indeed, the adresses of `a` and `b` are simply swapped internally here as well.

Functions as Arguments and Return Values

As is common in various languages, a function can take another function as an argument:

Example 18.9 The following function `ddx` computes an approximation to the derivative of a function f at a point x:

```
1 func ddx(_ f: (Double) -> Double, _ x: Double) -> Double {
2   let h = 1e-14
3   return (f(x+h) - f(x)) / h }
```

In line 1, the argument `f` is declared to be of type `(Double)->Double`, i.e. as a function $f\colon \mathbb{R} \to \mathbb{R}$.

We test the `ddx` operator:

```
func testFun(_ x: Double) -> Double { return x*x }
print(ddx(testFun, 0.5))  // 0.9992007221626409
```

Recall that in Python and Julia a function can in turn return a function as an output value. With a little caution, this is also possible in Swift:

Example 18.10 Continuing the program from the last example, we write a function that returns the approximate derivative function f' of an input function f.

To improve readability, we introduce an abbreviated representation of the function type `(Double)->Double`:

```
4 typealias DtoD = (Double) -> Double
```

We then define the derivative operator:

```
5 func ddx(_ f: @escaping DtoD) -> DtoD {
6   func f_prime(_ x: Double) -> Double { return ddx(f, x) }
7   return f_prime }
```

There are two observations here that may need some explanation. One is the use of the `@escaping` keyword in line 5. We return to this in the following remark. The other is that we use the same name for both functions declared in lines 1 and 5. In fact, Swift uses a mechanism to distinguish the two functions based on the number and types of arguments, similar to Julia's multiple dispatch method.

We test the derivative operator for our `testFun` above:

```
let testFun_prime = ddx(testFun)
print(testFun_prime(0.5))  // 0.9992007221626409
```

Remark 18.11 It remains to explain the `@escaping` keyword. In Swift, functions are instances of a reference type and can be automatically deallocated when they are no longer referenced. Now, the argument function `f` in line 5 is also referenced in the returned result `f_prime`. The annotation `@escaping` states that the reference to `f` must be kept even after the return of the enclosing function `ddx`.

Operators

Recall that on several occasions we have used operator overloading to provide an infix notation for mathematical operators. In Swift, we can even define new operators this way. As an example, we define a power operator to evaluate expressions of the form `x**n` for floating point numbers `x` and integers `n`. This is useful, for example, when evaluating polynomials:

```
1 infix operator **
2 func **(lhs: Double, rhs: Int) -> Double {
3   var res = 1.0
4   for _ in 1...rhs { res *= lhs }
5   return res }
6 print(4.2 ** 3)  // 74.08800000000001
```

In line 1, we declare ‘`**`’ in general to denote infix operators. Then we specify how it is applied to a pair of arguments consisting of types `Double` and `Int`.

Closures

Closures provide an alternative, convenient way to define functions, similar to the lambdas in e.g. Python.

Example 18.12 We demonstrate the idea with a function to compute the power a^b for natural numbers a and b:

```
1 let pow = { (a: Int, b: Int) -> Int in
2   var res = 1
3   for _ in 1...b { res *= a }
4   return res }
5 print(pow(2, 4))  // 16
```

A closure consists of a type declaration for arguments and return value, followed by the keyword ‘`in`’ as in line 1, followed by the function body in lines 2 through 4. The closure is enclosed in curly braces.

If the data types involved are obvious from the context, the type annotation can be omitted. In the present case we can replace line 1 with

```
let pow = { (a, b) in
```

A closure may omit names for its parameters. Its parameters are then implicitly named `$` followed by their position: `$0`, `$1`, `$2` and so on. In this case, the keyword `in` must also be omitted:

```
let pow = {var res = 1; for _ in 1...$1 {res *= $0}; return res}
```

For a closure consisting of only a single expression, the value of that expression is returned. The content of this expression is also taken into account when performing type inference for the surrounding expression:

```
print({$0 + $1}(1, 2))    // 3
print({$0 + $1}(1, 2.0))  // 3.0
```

18.3 Arrays

Also in Swift there are several data types that collect elements of other types. In the following we will focus on arrays, by far the most important collection type.

The easiest way to create arrays is to enclose a comma-separated list of values in square brackets:

```
1 var arr: [Int] = [3, 1, 4]  // or, equivalent:
2 var arr = [3, 1, 4]
```

In line 1, the type annotation `[Int]` stands for "array of integers". It can also be omitted if the element type is clear from the context.

Contiguous integer arrays can also be created from ranges:

```
var contArr = Array(3...5)     // [3, 4 ,5]
```

Arrays with constant entries can be created with a '`repeating: count:`' initializer:

```
Array(repeating: 0, count: 3)  // [0, 0, 0]
```

Empty arrays require a type annotation:

```
var emptyIntArray: [Int] = []
```

Array elements are accessed using subscript syntax, as in C or Python, with indices starting at 0.

As in Python, you can access various properties and methods using dot notation. For example, the number of elements in an array `arr` can be retrieved with the `count` property:

```
var arr = [3, 1, 4, 1]
arr.count  // 4
```

To add individual elements to the end of an array, use the `append` method:

```
arr.append(5)      // [3, 1, 4, 1, 5]
```

To remove elements from an array, use the `remove(at:)` method:

```
arr.remove(at: 2)  // 4
print(arr)         // [3, 1, 1, 5]
```

Note that the element at index 2 is removed and the following elements are shifted down one index position to fill the gap.

Arrays are Value Types

Unlike Python and Julia, arrays in Swift are value types, as can be seen from the following:

```
let a = [1, 2]; var b = a; b[1] = 3
print(a)  // [1, 2]
```

Remark For the interested reader: This is not the whole truth, however. To illustrate, let's follow the memory addresses of the arrays in the computation above:

```
1 let a = [1, 2]; a.withUnsafeBufferPointer { print ($0) }
2   // UnsafeBufferPointer(start: 0x0000600000410ec0, count: 2)
3 var b = a;      b.withUnsafeBufferPointer { print ($0) }
4   // UnsafeBufferPointer(start: 0x0000600000410ec0, count: 2)
5 b[1] = 3;       b.withUnsafeBufferPointer { print ($0) }
6   // UnsafeBufferPointer(start: 0x0000600001f1b820, count: 2)
```

Note that `b` in line 3 gets the same memory address as `a` in line 1. Only when `b` in line 5 is changed is it saved to a new location.

This resource management technique is called *copy-on-write*.

As a slightly larger example for the use of arrays we consider:

Example 18.13 (Sieve of Eratosthenes) Here is a Swift implementation of the Sieve of Eratosthenes from Example 3.4 in the Python chapter:

```
1 let n = 100
2 var L = Array(2...n)
3 var P: [Int] = []
4 while L != [] {
5   let p = L[0]
6   P.append(p)
7   for i in (0 ..< L.count).reversed() {
8     if L[i] % p == 0 { L.remove(at: i) } } }
9 print(P)
```

The only thing to note is that in the loop, in lines 7 and 8, we remove the multiples of `p` in reverse order, starting at the end. The reason for this is that the remove operation shortens the list, so if we counted from bottom up, we would run out of range.

Subarrays

As in Python, when you want to retrieve a section of an array, you use a one-sided or two-sided range operator:

```
let arr = [3, 1, 4, 1, 5]
arr[1...3]  // [1, 4, 1]
arr[...2]   // [3, 1, 4]
arr[3...]   // [1, 5]
```

These slices refer to the orginal `arr` array but can be assigned to new arrays, e.g. with

```
let subArr = arr[1...3]
```

Array Concatenation

You can concatenate arrays using the `+` operator to create a new array:

```
let a = [3, 1, 4]; let b = [1, 5]
let c = a + b  // [3, 1, 4, 1, 5]
```

Other Useful Array Methods

Swift provides several other useful built-in array methods. Here are some examples.

Example 18.14 (The `reversed` and `reverse` Methods) Similar to ranges, you can reverse the order of an array by using the `reversed` and `reverse` methods:

```
1 var a = [1, 2, 3]
2 let b = Array(a.reversed())
3 print(b)  // [3, 2, 1]
4 print(a)  // [1, 2, 3]
5 a.reverse()
6 print(a)  // [3, 2, 1]
```

In line 2 we use `reversed` to create a reversed view of `a`, which can then be assigned to a new array using the `Array` constructor. The array `a` itself is not changed.
In line 5, `a` is modified in place using the `reverse` method.

Example 18.15 (The `map` Method) The `map` method takes each value of an array, applies a function to it, and finally places the resulting values in a new array:

```
1 let a = [1, 2, 3, 4]
2 let b = a.map(fac)  // fac defined in Example 18.6
3 print(b)            // [1, 2, 6, 24]
```

In line 2, the factorial function is mapped to all elements in the array `a`.

Very often the `map` method is used with closures:

```
4 let c = a.map( { $0 * $0 } )  // [1, 4, 9, 16]
```

In this case the enclosing parentheses can be omitted, and the closure can be written directly after the method name:

```
5 let c = a.map { $0 * $0 }  // equivalent to line 4
```

Note that this abbreviated form can also be used in general with array methods that take functions as arguments.

Example 18.16 In Example 18.13 we used a for-loop construction to remove all muliples of a given number from an array. Instead of a loop, we can use the `filter` method. Here we remove all even numbers from the array `a`:

```
1 var a = Array(1...6)
2 a = a.filter { $0 % 2 != 0 }  // [1, 3, 5]
```

In line 2, again note that we simply append the closure to be applied by the `filter` method instead of writing it as an argument in parentheses.

Using the `filter` method, the for loop in lines 7 and 8 in Example 18.13 can be replaced by

```
L = L.filter { $0 % p != 0 }
```

Multiple methods can also be chained:

Example 18.17 In the Python chapter we discussed list comprehension. By combining the `filter` and `map` methods, we can emulate such behavior in Swift:

```
let a = [3, 1, 4, 3]
let b = a.filter{$0 < 4}.map{$0 * $0}  // [9, 1, 9]
```

An extremely versatile method is `reduce`. It is related to the MPI reduce and reduction methods used in parallel programming.

Example 18.18 Here is an example of using `reduce` to add the first 100 natural numbers:

```
1 let a = Array(1...100)
2 let sumNums = a.reduce(0, {(sum, n) in sum + n})  // 5050
```

What happens is that `sum` is initialized with the first argument `0` in `reduce`. Then the expression `sum + n` is successively evaluated for each `n` in `a` and the updated value assigned to `sum`.

In the present case, the closure used can easily be inlcuded in the argument pair for `reduce(,)`. Again, as for closures in general, there is a shorter notation: we can write `reduce(0)` and then append the closure `{...}` outside the argument list.

We get an even shorter notation if we replace the names `sum` and `n` with their positional arguments. Instead of line 2 we can then write equivalently:

```
2 let sumNums = a.reduce(0) { $0 + $1 }  // 5050
```

Example 18.19 (Horner's Rule) We show how to use the `reduce` method to evaluate polynomials $p(x) = \sum_{i=0}^{n} a_i x^i$ based on Horner's rule

$$(*) \qquad p(x) = (\dots(a_n x + a_{n-1})x + \cdots)x + a_0.$$

As an example, we consider the polynomial $p(x) := 1 + 2x + 3x^2$ and compute $p(4) = 57$:

```
1 let a = [1, 2, 3]; let x = 4
```

We first give a solution with a for loop. We compute $(*)$ from the inside out, so that the loop runs in the reverse order of `a`, and arrive at

```
var res = 0
for c in a.reversed() { res = x*res + c }
```

Observing the order of the positional arguments in Example 18.18 above, this immediately translates to

```
let res_red = a.reversed().reduce(0) { x*$0 + $1 }
```

In both cases we get the correct answer 57.

We conclude this section with an example that shows that arrays can also be the output of a function.

Example 18.20 (Quicksort) Here is a Swift implementation of the Quicksort algorithm. For background we refer to the Python chapter:

```
func qsort(_ lst: [Int]) -> [Int] {
  if lst == [] { return [] }
  let p = lst[0]
  let sml = lst[1...].filter { $0 < p }
  let grt = lst[1...].filter { $0 >= p }
  return qsort(sml) + [p] + qsort(grt) }
```

We test it:

```
let lst = [4, 5, 7, 3, 8, 3]
print(qsort(lst))  // [3, 3, 4, 5, 7, 8]
```

18.4 Optionals

The C language provides a special pointer value `NULL` to indicate that a pointer does not point to a valid object. This can be used as a protection against dereferencing an invalid pointer, which would cause the program to crash.

In Swift, this idea is extended to non-pointer data types. A variable can be specified as *optional* by appending a question mark to the data type. This means that it either has a special value `nil`, which means no value, or it contains a valid value which can then be accessed through a mechanism called *unwrapping*.

Example 18.21 With

```
var a: Int?
```

the variable `a` is declared to be of type optional `Int`. It is automatically initialized to have no value:

```
print(a)   // nil
```

Here `nil` is the abbreviation for "nihil", the Latin word for "nothing".

The variable can also be assigned a valid `Int` number:

```
a = 42
```

This number is then stored in the form of an optional:

```
4 print(a)   // Optional(42)
```

which can be accessed through unwrapping, where an exclamation mark is appended to the variable:

```
5 print(a!)  // 42
```

Returning to the pointer analogy, unwrapping an optional `a!` corresponds to dereferencing a pointer `*ptr`. As with pointers, this kind of unwrapping mechanism should only be used when you are sure that the optional will have a value:

```
6 var b: Int?; print(b!)  // program crashes
```

In line 6, the variable `b` is `nil`; the program crashes if you try to unwrap the non-existent value.

Example 18.22 (`remove` Function) In Example 18.13, we mentioned that Swift, unlike Python, does not provide a method to remove an *entry* from an array. The deeper reason for this is that it is not clear what should happen if the entry does not occur in the array. This is where the optional type can help:

```
1 func remove(_ a: [Int], _ x: Int) -> [Int] {
2   var b = a
3   let i = b.firstIndex(of: x)
4   if i != nil { b.remove(at: i!) }
5   return b }
```

The method `firstIndex(of:)` in line 3 returns by definition the index of the first occurrence of `x` if it occurs in `a`, and `nil` otherwise. If `i` in line 4 contains a valid index value, the value will be unwrapped and the entry at that index position is removed. Otherwise the array remains untouched.

We test the function:

```
6 var a = [3, 1, 4, 1]
7 print(remove(a, 1))  // [3, 4, 1]
8 print(remove(a, 2))  // [3, 1, 4, 1]
```

In line 7, the first occurrence of 1 is removed. In line 8, nothing happens because the element 2 does not occur in `a`.

Remark Note that the `remove` function can be used to replace the loop in lines 7 and 8 in Example 18.13 with

```
for x in L { if x % p == 0 { L = remove(L, x) } }
```

Conditional statements like the one in line 4 in Example 18.22 above occur so frequently that Swift provides an equivalent shorthand notation that uses the '`if let`' construction. For example, line 4 above can be replaced with

```
4 if let i = i { b.remove(at: i) }
```

The assignment to `i` is only performed if `i` is a valid index value. Unwrapping `i` in the `remove(at:)` method is then no longer necessary.

An even shorter equivalent formulation is as follows:

```
4 if let i { b.remove(at: i) }
```

18.5 Classes and Strucutures

Classes in Swift are very similar to classes in Python and C++. Structs go beyond those used in C and Julia because they can also encapsulate application methods. In fact, classes and structs in Swift differ only slightly syntactically.

One of the most important differences is that structs are always copied when passed, while classes are passed by reference. Another difference is that one class can inherit the properties of another through subclassing. This is not possible with structures.

Example 18.23 As a first example, we set up a structure `Frac` to handle fractions, similar to Examples 3.19, 7.5, and 8.19.

In its minimal form `Frac` provides the data type:

```
1 struct Frac { var num, den: Int }
```

A `Frac` instance can be initialized and accessed like this:

```
2 let a = Frac(num: 3, den: 4)
3 print(a.num, a.den)  // 3 4
```

In line 2, we use a built-in default initializer that requires the names of the members `num` and `den` in the call. We can override this requirement with a self-defined initializer:

```
4 extension Frac {
5   init(_ num: Int, _ den: Int){ self.num = num; self.den = den } }
```

Note the `extension` keyword in line 4. It causes line 5 to be appended to the `Frac` definition as if it were written after the member declaration in line 1. This is a very handy way to add extra components in a structure.

As in Python, the keyword `self` is used to distinguish the property names from the argument names in the initializer. The arguments to the initializer are passed just like in a function call when you create an instance of the struct.

Then we can instantiate a new fraction either as in line 2, but now also in the form

```
6 let a = Frac(3, 4)
```

As mentioned earlier, unlike other languages, we can also insert methods into structs, either in the original definition or as an extension like in line 4.

Here is a method for printing fractions:

```
7   func show() { print("\(self.num)/\(self.den)") }
```

We then get

```
8 a.show()  // 3/4
```

Note that we have used string interpolation here, as explained in Sect. 18.1, to format the output.

We include a method for multiplying fractions in the structure definition:

```
9    static func mul(_ a: Frac, _ b: Frac) -> Frac {
10     return Frac(a.num * b.num, a.den * b.den) }
```

Here the keyword `static` means that `mul` is a *type* method, not an instance method. It is called like this:

```
11 let b = Frac.mul(a, a)
12 b.show()  // 9/16
```

Note the use of the type prefix `Frac`.

We can turn the `mul` method into an infix-operator by simply changing the name `mul` to '*'. Then we can write

```
13 let b = a * a
14 b.show()  // 9/16
```

Note that the `Frac` prefix is not used in line 14.

Now that we have seen an example of a structure definition, let's show an example of a class. But this is easy. Syntactically, we just need to change the keyword `struct` to `class` in the example above. For the sake of convenience, we'll give the entire definition:

Example 18.24 A class for the representation of fractions:

```
1 class Frac {
2   var num, den: Int
3   init(_ num: Int, _ den: Int){ self.num = num; self.den = den }
4   func show() { print("\(self.num)/\(self.den)") }
5   static func *(_ a: Frac, _ b: Frac) -> Frac {
6            return Frac(a.num * b.num, a.den * b.den) } }
```

Just note that the initializer in line 3 cannot omitted. Classes do not have a built-in default initializer like structs.

In general, it is more common in Swift to use structures than classes. In fact, many of the internal data types, such as numbers, but also arrays, are implemented with structures.

One example where we need classes is in modeling *inheritance*:

Example 18.25 We define a class `Pol` for representing polynomials, similar to Python and C++, for which we then introduce a subclass `Par` for polynomials of degree 2:

```
1 class Pol {
2   let coeff: [Double]
3   init(_ coeff: [Double]) { self.coeff = coeff }
4   func eval(_ x: Double) -> Double {
5     return coeff.reversed().reduce(0) {x * $0 + $1} } }
```

In line 2, we specify that a polynomial is represented by an array of type `Double`. In line 3, the special function `init` loads the external parameter array into the local `coeff` array. In lines 4 and 5, we define a method for evaluating a polynomial represented by such an array. Here we use Horner's rule discussed in Example 18.19.

We test the construction so far:

```
let p = Pol([1, 2, 3])
print(p.eval(4))  // 57.0
```

We include a method for adding polynomials.

```
6  extension Pol {
7    static func +(_ ls: Pol, _ rs: Pol) -> Pol {
8      let (hdg, ldg) =
9        (ls.coeff.count < rs.coeff.count) ? (rs, ls) : (ls, rs)
10     var sumCoeff = hdg.coeff
11     for i in 0 ..< ldg.coeff.count { sumCoeff[i] += ldg.coeff[i] }
12     return Pol(sumCoeff) } }
```

In line 7, the keyword `static` associates the addition method with the class type itself rather than with an instance of the class. In lines 8 through 10 we choose the higher degree polynomial argument and in line 11 we add the coefficients of the lower degree polynomial.

The adition method is called like this:

```
let q = Pol([4, 5])
let r = p + q     // p defined as above
print(r.coeff)    // [5.0, 7.0, 3.0]
print(r.eval(4))  // 81.0
```

Example 18.26 (Subclassing) As in Python and C++, we demonstrate inheritance using a subclass `Par` of `Pol` to represent parabolas:

```
13 class Par: Pol {
14   override init(_ coeff: [Double]) {
15     if coeff.count != 3 { fatalError("No parabola") }
16     super.init(coeff) }
17   override func eval(_ x: Double) -> Double {
18     return coeff[0] + coeff[1]*x + coeff[2]*x*x }
19   static func +(ls: Par, rs: Par) -> Par {
20     var sumCoeff: Vec = [0, 0, 0]
21     for i in 0...2 { sumCoeff[i] += ls.coeff[i] + rs.coeff[i] }
22     return Par(sumCoeff) } }  // end subclass Par
```

In lines 14 and 17, we override the method definitions in the superclass. In Swift, this requires the explicit inclusion of the keyword `override`.

The `init` function simply rejects polynomials that are not of degree 2, and otherwise passes the argument to the `init` function in `Pol`.

The `eval` method is just a simplified variant of the superclass method.

The subclass methods can be called like this:

```
23 let p = Par([1, 2, 3]); let q = Par([3, 2, 1])
24 print((p + q).eval(4))  // 84.0
```

Notice that the distinction between the addition operators in `Pol` and `Par` is not visible in the expression in line 24. What happens is that, as with Julia's multiple dispatch method, the type of the arguments determines which one is used.

To make it clear which method is being used, you can, for example, include `print("used: parabola operator")` in the method body.

Linked Lists

We have already dicussed several times how linked lists can be implemented. Here we present a Swift implementation based on classes and structs.

Example 18.27 We start with the data type to represent nodes:

```
1  class Node {
2    var val: Int
3    var next: Node?  // nil-initialized
4    init(_ val: Int) { self.val = val } }
```

Here we need to use classes. The reason for this is that structs cannot refer to instances recursively, as required in line 3.

The initializer in line 4 does not have to assign a value to the '`next`' variable, since it is automatically initialized with `nil`.

The construction of the linked list itself can be based on a class of such nodes:

```
5  class LinkedList {
6    var head, tail: Node?
```

again automatically `nil`-initialized by default.

The following method is used to link a new node at the end of the list:

```
7    func addNode(_ val: Int) {
8      let newNode = Node(val)
9      if tail == nil { head = newNode }
10     else { tail!.next = newNode }
11     tail = newNode }
```

In line 10, we can safely unwrap the value in `tail`, since the test in line 9 only takes us to this line if `tail` is not `nil`.

Finally, we add a method to display the contents of the list:

```
12   func show() {
13     var node = head
14     while node != nil {
15       print(node!.val, terminator: " ")
16       node = node!.next } }
17 } // end linked list def
```

Again, note that unwrapping the nodes in lines 15 and 16 is safe because of the test in the loop header in line 14.

We test the construction:

```
var list = LinkedList()
for i in 1...10 { list.addNode(i) }
list.show()  // 1 2 3 4 5 6 7 8 9 10
```

Remark We can base the construction of the linked list on a struct instead of a class. To do this, we change the type specifier in line 5 from `class` to `struct`. But note that structs are immutable by default. We therefore need to explicitly declare the `addNode` function as *mutating*, i.e. we replace the function header in line 7 with

```
7 mutating func addNode(_ val: Int)
```

18.6 Linear Algebra

Swift does not yet provide user-friendly tools for high-level linear algebra. There are some promising proposals, such as the LANumerics package available on the site `github.com/phlegmaticprogrammer`, but they are all still in the early stages of development.

What Swift does offer, however, is a fine-tuned implementation of the highly efficient low-level packages BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage), which were originally developed for the Fortran language. For background, see `en.wikipedia.org/wiki/LAPACK` and `netlib.org/lapack`.

In this section, we take the opportunity to discuss these packages, which also form the basis for numerical linear algebra in several other languages, including MATLAB and Scientific Python.

The required components are included in the Accelerate package. Apparently due to an ongoing reorganization of synchronization between Xcode and the Accelerate package, however, there doesn't seem to be an obvious way to adjust the required settings to a playground at this time.

Instead, we create a new Swift Xcode project of the form Comand Line Tool for macOS. In the start window we select Build Settings and then search for Preprocessor Macros, and click on the rightmost <Multiple values> item. In the panel that appears, we enter the two lines ACCELERATE_NEW_LAPACK=1 and ACCELERATE_LAPACK_ILP64=1.

In the following we assume the Accelerate package is loaded with

```
import Accelerate
```

Vectors

We define *vectors* as arrays of `Double`. For better readability we declare:

```
typealias Vec = [Double]
```

We use BLAS routines to compute basic vector operations, more specifically the `cblas` variant with a C-like interface.

Multiplication by a scalar a is computed using the BLAS function `dscal`, where `scal` stands for "scaled" or "scalar" and "d" for double precision. The prefix `cblas` indicates that the BLAS implementation in Accelerate is based on the C adapted version:

```
1 func *(_ a: Double, _ x: Vec) -> Vec {
2   let n = x.count
3   var res = x
4   cblas_dscal(n, a, &res, 1)
5   return res }
```

What the `dscal` routine computes in line 4 is essentially `x *= a`. Therefore, we copy `x` into an `inout` variable `res`, which is specifed as a mutable function argument by the preceding ampersand.

The last argument 1 means that the vector entries are processed with an increment of 1: `x[0]`, `x[1]`, `x[2]`, etc., i.e. all elements. An increment of, say, 2 would mean that only `x[0]`, `x[2]`, `x[4]` are considered. In this chapter we have no use for this variant.

Example:

```
let x: Vec = [1, 2, 3]
print(2 * x)  // [2.0, 4.0, 6.0]
```

Vector addition is computed with

```
1 func +(_ x: Vec, _ y: Vec) -> Vec {
2   let n = x.count
3   var res = y
4   cblas_daxpy(n, 1, x, 1, &res, 1)
5   return res }
```

The routine name '`daxpy`' in line 4 stands for "compute $ax + y$ in double precision", in this case for $a = 1$, as indicated by the first 1 entry. Here we need to copy the y value into an `inout` variable `res` to get `res += x`. The remaining two 1 entries specify that we assume the `x` and `y` elements to be stored contiguously.

Example:

```
let y: Vec = [3, 2, 1])  // x as above
print(x + y)             // [4.0, 4.0, 4.0]
```

Remark Note that the addition operator actually overwrites the standard meaning in arrays, namely concatenation, see e.g. Example 18.20. If you want a cleaner construction, you can instead use, for example, an operator `.+.` with

```
infix operator .+.
```

and then change `+` to `.+.` in the definition above. Then:

```
print(x + y)    // [1.0, 2.0, 3.0, 3.0, 2.0, 1.0]
print(x .+. y)  // [4.0, 4.0, 4.0]
```

Vector subtraction $x - y$ can also be computed with `daxpy` in the form `x += (-1)*y`:

```
func -(_ x: Vec, _ y: Vec) -> Vec {
  let n = x.count
  var res = x
  cblas_daxpy(n, -1, y, 1, &res, 1)
  return res }
```

As a final example, we show how to compute the *dot product* between two vectors:

```
func *(_ x: Vec, _ y: Vec) -> Double {
  let n = x.count
  return cblas_ddot(n, x, 1, y, 1) }
```

Again, the 1 arguments mean that the vectors are assumed to be stored contiguously.

Matrices

We define a matrix row by row as an array of `Double` arrays:

```
typealias Mat = [[Double]]  // equivalent = [Vec]
```

Example:

```
let A: Mat = [[1, 2, 3], [4, 5, 6]]
```

With matrices, it is often convenient to print them in the standard 2D form. To do this, we introduce the following `show` method, extending the `Mat` structure:

```
extension Mat {
  func show(_ n: Int, _ m: Int) {
    for i in 0 ..< self.count {
      for j in 0 ..< self[0].count {
        print(String(format: "%\(n+m+3).\(m)f",
                              self[i][j]), terminator:" ") }
      print() } } }
```

Note that we specify the number of integer and decimal digits (in that order) independently. Example:

```
A.show(1, 2)  // with A as above
  1.00   2.00   3.00
  4.00   5.00   6.00
```

The Functions `flatten` and `reshape`

To process matrices with Blas and Lapack routines we need a representation in flat form as linear arrays:

```
func flatten(_ A: Mat) -> (Vec, Int, Int) {
  let rows = A.count; let cols = A[0].count
  var data: Vec = []
  for j in 0 ..< cols { for i in 0 ..< rows {
    data.append(A[i][j]) } }
  return (data, rows, cols) }
```

Note that in lines 1 and 6 the return value is specified as a tuple, similar to Example 18.7, here as a 3-tuple. This also ilustrates that the values in a tuple can be of different types, here `Vec` and `Int`.

Also note that we choose a linear representation in *column-major* form. The reason for this is that the Lapack routines to be introduced later require this form.

Example:

```
let (data, rows, cols) = flatten(A)  // A as above
print(data, rows, cols)  // [1.0, 4.0, 2.0, 5.0, 3.0, 6.0] 2 3
```

We also need the inverse:

```
1 func reshape(_ data: Vec, _ rows: Int, _ cols: Int) -> Mat  {
2   var A: Mat = []
3   for i in 0 ..< rows {
4     var w: Vec = []
5     for j in 0 ..< cols { w.append(data[i + rows * j]) }
6     A.append(w) }
7   return A }
```

In line 5, we reconstruct the individual rows, and append them row by row to the matrix `A` in line 6.

Example:

```
let B = reshape(data, rows, cols)  // values from example above
print(B)  // [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
```

Matrix-Vector Multiplication

Using `flatten` and `reshape`, we define a *matrix-vector multiplication* operator with the Blas routine `dgemv`. Here "`mv`" stands for "matrix-vector", "`ge`" for "general":

```
1 func *(_ A: Mat, x: Vec) -> Vec {
2   let (data, rows, cols) = flatten(A)
3   var res = Vec(repeatElement(0, count: rows))
4   cblas_dgemv( CblasColMajor, CblasNoTrans,
5     rows, cols, 1.0, data, rows, x, 1, 0.0, &res, 1 )
6   return res }
```

Note that the first argument in line 4 specifically states that we use column-major ordering to ensure compatiblility with subsequent Lapack routines. The second argument specifies that we do not require transposition of the input matrix.

In its general form, `dgemv` returns a vector of the form `y = a*A*x + b*y`. In line 5, we first specify the dimensions of the input matrix `A` and then set the values `a = 1.0` and `b = 0.0`.

The elements of `A` are stored in column-major order in the linear array `data`. The following second entry of '`rows`' specifies the number of elements in each column and in particular that the values are stored contiguously.

Example:

```
let x: Vec = [3, 2, 1]
print(A * x)  // [10.0, 28.0]
```

Matrix-Matrix Multiplication

As a final example of the basic linear algebra operations, we show how to perform *matrix-matrix multiplication* $AB = C$ with the routine `cblas_dgemm`:

```
1 func *(A: Mat, B: Mat) -> Mat {
2   let (dataA, rowsA, _) = flatten(A)
3   let (dataB, rowsB, colsB) = flatten(B)
4   var dataC = Vec(repeating: 0.0, count: rowsA * colsB)
5   cblas_dgemm( CblasColMajor, CblasNoTrans, CblasNoTrans,
6     rowsA, colsB, rowsB,
7     1.0, dataA, rowsA, dataB, rowsB, 1.0, &dataC, rowsA )
8   return reshape(dataC, rowsA, colsB) }
```

The underscore in the tuple in line 2 means that the third entry is not used.

In line 4, we initialize an array `dataC` with the appropriate length to zero to store the product matrix.

In line 6, we specify the dimensions of the argument matrices `A` and `B`. Note that the number of rows in `B` is equal to the number of columns in `A`.

In line 7, the product matrix `C` is stored in column-major form in the linear array `dataC`. The last argument `rowsA` is equal to the number of elements in each column of `C`.

Example:

```
let B: Mat = [[1,4], [2,5], [3,6]]  // Then with A as above:
print(A * B)  // [[14.0, 32.0], [32.0, 77.0]]
```

The LAPACK Package

Swift's Accelerate framework also provides the LAPACK package, which builds on BLAS and offers routines for solving systems of linear equations, eigenvalue problems, and related matrix factorizations, including LU, Cholesky, and QR decompositions.

While the `cblas` package used above has been adapted to some extent to C and Swift, the LAPACK interface still clearly shows its origins in Fortran. Remember that function arguments in Fortran are always passed by reference. This is also true for Swift Accelerate, so all function arguments must be specified as mutable with an ampersand prefix.

Also, we have no choice here whether to store matrices in row-major or column-major order. Fortran requires column major.

To demonstrate the general principle we develop a program to invert matrices.

Example 18.28 (Matrix Inversion) Technically, the inversion of a matrix A is performed in two steps in Lapack. First, we use use the routine `degtrf` to compute the LU factorization $A = LU$, followed by a call to `dgetri` that then computes U^{-1}, and solves the matrix equation $XL = U^{-1}$. For an explanation of `degtrf` and Lapack's general naming convention, see Example 17.14 in the Fortran chapter.

We start with the LU decomposition:

```
1 func inv(_ A: Mat) -> Mat {
2   var (data, rows, cols) = flatten(A)
3   var info = 0; var ipiv = [Int](repeatElement(0, count: rows))
4   dgetrf_(&rows, &cols, &data, &rows, &ipiv, &info)
```

In line 3, recall from the Fortran version that the vector `ipiv` stores the row permutations required in the decomposition.

In line 4, note that in Accelerate all Lapack routines are written with an underscore. The second entry of '`rows`' again specifies the number of elements in each column of the matrix representation in `data`.

We come to the actual inversion:

```
5   var lwork = rows; var work = Vec(repeatElement(0, count: rows))
6   dgetri_(&rows, &data, &rows, &ipiv, &work, &lwork, &info)
7   return reshape(data, rows, cols) }
```

The LU decompostion supplied by `dgetrf` is passed to `dgetri` via the array `data` in line 6, and then modified in place so that when `dgetri` returns, it contains the inverse of `A` in linear representation. Note that the "tri" part of the routine name indicates that `dgetri` works with triangular matrices.

In line 7, the linear array is reshaped to a matrix representation.

The parameters `work` and `lworks` can be used for technical optimization.

Example:

```
let A: Mat = [[1, 2, 3], [1, 1, 1], [3, 3, 1]]
inv(A).show(1, 2)  // Out:
  -1.00   3.50  -0.50
   1.00  -4.00   1.00
   0.00   1.50  -0.50
```

Application Example: The Gauss-Seidel Method

In the exercises for the scientific Python chapter, we introduced the iterative *Gauss-Seidel method* for solving a system of linear equations $Ax = b$. Here, the matrix A is decomposed into a triangular component L including the diagonal and a strictly upper triangular component U such that $A = L + U$.

Starting from an initial value $x^{(0)}$, the method works by iteratively computing the following sequence:

$$(*) \qquad x^{(k+1)} := L^{-1}(b - Ux^{(k)}).$$

Under the condition that the main diagonal consists only of elements $a_{ii} \neq 0$, the sequence $(*)$ converges to the solution.

As an example, we consider the linear equation system

$$\begin{pmatrix} 4 & 3 & 0 \\ 3 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 24 \\ 30 \\ -24 \end{pmatrix}.$$

We show how to find the solution $x = (3, 4, -5)^T$ in Swift, starting from the initial approximation $x^{(0)} = (3, 3, 3)^T$.

First we partition the matrix `A` into the sum `A = L + U` of lower and upper triangular parts:

```
let A: Mat = [[4, 3, 0], [3, 4, -1], [0, -1, 4]]
var L = A; var U = A
for i in 0..<3 { for j in 0..<3 {
  if  i < j { L[i][j] = 0 }
  else { U[i][j] = 0 } } }
```

We compute the inverse of the lower part `L` using the function `inv` from Example 18.27 above:

```
let Linv = inv(L)
Linv.show(1, 3)
  0.250   0.000   0.000
 -0.188   0.250   0.000
 -0.047   0.062   0.250
```

Starting with

```
var x: Vec = [3, 3, 3]
var b: Vec = [24, 30, -24]
```

we follow the approximation for 10 iteration steps:

```
for _ in 1...10 { x = Linv * (b - (U * x)) }
print(x)       // [2.97, 4.02, -4.99]    (rounded)
print(A * x)  // [23.96, 30.00, -24.0]  (rounded)
```

Remark Note that above we used a combination of two general methods for computing the inverse L^{-1}. Now, in our case L is a *triangular* matrix, and for such matrices Lapack provides a single simplified routine `dtrtri`:

```
1 var (data, rows, _) = flatten(L)
2 var info = 0
3 dtrtri_("l", "n", &rows, &data, &rows, &info)
```

The underscore in the tuple in line 1 again means that the third entry is not used. The first argument `"l"` in line 2 indicates that the input matrix is lower triangular, the annotation `"n"` that it is non-unit triangular, i.e. the diagonal is not constant 1.

The result is the same as before:

```
let LtriInv = reshape(data, 3, 3)
LtriInv.show(1, 3)
```

with the output:

```
 0.250   0.000   0.000
-0.188   0.250   0.000
-0.047   0.062   0.250
```

Linear Equations: Direct Solution

We use the LAPACK routine `dgesv` to define a function for the direct solution of matrix equations:

```
1 func solve(_ A: Mat, _ b: Vec) -> Vec {
2   var (data, rows, _) = flatten(A)
3   var nrhs = 1; var ipiv = [Int](repeatElement(0, count: rows))
4   var bx = b; var info = 0
5   dgesv_(&rows, &nrhs, &data, &rows, &ipiv, &bx, &rows, &info)
6   return bx }
```

Note that function `dgesv` is generally intended for matrix equations of the form $AX = B$. The variable `nrhs` in line 2 represents the number of columns on the right-hand side B. In the present case, we consider only an equation with a single vector `b` and therefore set `nrhs = 1`.

`dgesv` now works as follows: First, the LU decomposition of `A` is computed and stored in the vector `data`, just as for `dgetref` in Example 18.28.

In line 4, the value of the right hand side `b` of the equation is stored in the `inout` variable `bx`. The solution `x` is then computed using the LU decomposition in `data` and the result stored in `bx`, overwriting the initial value of `b`.

We test the `solve` function for the same matrix `A` and vector `b` as in the Gauss-Seidel example:

```
let A: Mat = [[4, 3, 0], [3, 4, -1], [0, -1, 4]]
let b: Vec = [24, 30, -24]
print(solve(A, b))  // [3.0, 4.0, -4.999999999999999]
```