

The Swift Language Handbook

An Introduction
and Guidebook for
Pascal Programmers
Second Edition



Henry Lowe

All Rights Reserved: This book was published by the author Henry Lowe under Ascriva Press. No part of this book may be reproduced in any form by any means without the express permission of the author. This includes reprints, excerpts, photocopying, recording, or any future means of reproducing text. If you would like to do any of the above, please seek permission first by contacting the author at hlowe@ascriva.com

Published in the United States by Ascriva Health Informatics LLC, Stanford, California.

© 2022 - 2026 by Ascriva Health Informatics LLC.

Table of Contents

Preface	1
The Pascal Programming Language.....	1
The Swift Programming Language.....	1
Origins of This Book.....	2
The Swift Playgrounds App	4
Code Examples.....	4
Swift's Print() Statement.....	5
References.....	7
1 Introduction	8
1.1 The Form of a Program.....	8
1.2 Declaration	10
1.2.1 Case Sensitivity	10
1.2.2 Constants	11
1.2.3 Variables.....	13
1.2.4 Constant and Variable Names.....	14
2 Operators	16
2.1 Assignment.....	16
2.2 Equality	16
2.3 Non-Equality	17
2.4 Arithmetic Operators	18
2.5 Logical Operators	20
2.6 Range Operators	22
3 Data Types	27
3.1 Numbers	27
3.1.1 Integer	27
3.1.2 Floating Point Number.....	28
3.1.3 Numeric Type Conversion	29
3.1.4 Rounding.....	30
3.1.5 Truncation.....	31
3.1.6 Generating Random Numbers.....	32
3.1.7 Numeric Functions	33
3.1.8 Converting Numbers to Strings	34

3.2 Booleans	35
3.2 Booleans	35
3.3 Strings and Characters.....	36
3.3.1 Character Declaration.....	37
3.3.2 String Declaration	37
3.3.3 String Literals	38
3.3.4 String Length	40
3.3.5 String Character Indices	41
3.3.6 String Concatenation	47
3.3.7 String Comparison.....	49
3.3.8 String Interpolation	50
3.3.9 String Insertion and Removal.....	51
3.3.10 String Search.....	54
3.3.11 String Replace	55
3.3.12 String Split	56
3.3.13 Substrings.....	57
3.4 Arrays	58
3.4.1 Declaration	59
3.4.2 Initialization	59
3.4.3 Element Operations	61
3.4.4 Searching an Array	64
3.4.5 Sorting an Array	70
3.5 Sets.....	71
3.5.1 Declaration.....	72
3.5.2 Set Operations	73
3.5.2 Set Operations	73
3.5.4 Sorting a Set.....	76
3.6 Dictionaries	76
3.6.1 Declaration.....	76
3.6.2 Adding and Deleting Entries	78
3.6.3 Optionals.....	79
3.6.4 Iteration.....	81
3.6.5 Sorting Entries.....	83
3.6.6 Searching Dictionaries	83
3.7 Records and Structures.....	86
3.7.1 Declaration	87

3.7.2 Property Values	89
3.7.3 Instance Methods	91
3.8 Enumeration	93
3.8.1 Declaration	94
3.9 Classes and Why They are Not Discussed Here	101
3.10 Type Aliases	102
4 Control Flow	104
4.1 If Statement.....	104
4.2 If Then Else Statement	105
4.3 Guard Statement.....	106
4.4 Switch Statement	108
4.5 Repeat Loops	111
4.6 Repeat While Loops	114
4.7 Control Transfer Statements.....	116
4.7.1 Break Statement.....	116
4.7.2 Continue Statement	117
4.7.3 Fallthrough Statement	118
5 Functions and Procedures	119
5.1 Function Definition	122
5.2 Calling Swift Functions	122
5.3 Argument Labels	123
5.4 Return Values.....	124
5.5 Function Parameters	125
5.5.1 Variadic Parameters.....	127
5.6 Emulating Pascal Procedures	128
5.7 Tuples.....	131
5.8 Mutating Functions	134
5.9 Generic Functions	137
5.10 Defer	138
6 File Management.....	140
6.1 URL.....	141
6.1.1 Declaration.....	142
6.1.2 Methods	144

6.3 FileManager	146
6.3.1 Declaration	147
6.3.2 Methods	147
6.3.3 Useful Functions	149
6.4 Data	152
6.4.1 Declaration	153
6.4.2 Iteration	154
6.4.3 Reading and Writing Data Files	154
6.4.4 Splitting	155
6.4.5 When to use Data versus String.....	156
7 Error Handling.....	157
7.1 Throwing Functions and the Throw Statement.....	158
7.2 Do-Catch Statement.....	160
7.3 Using Optionals	161
7.4 Returning Failure as a Value: Tuples and Result.....	162
7.5 Async throws (Swift Concurrency).....	163
8 Extensions	165
8.1 Syntax	165
8.2 Examples	165
8.3 Limitations	168
9 Modules.....	170
9.1 Namespaces.....	170
9.2 Import Keyword.....	170
10 Access Control.....	172
10.1 Access Control Levels	172
10.2 Defining Access Levels.....	173
11 Protocols	175
11.1 Defining a Protocol	175
11.2 The Static Keyword in Protocols	177
11.3 Protocols as Types	179
12 Closures Revisited	180

12.1 The Basic Idea	180
12.2 Closures as Function Parameters.....	181
12.3 Trailing Closures (Swift's favorite syntax).....	181
12.4 Common Closure-based operations	182
12.5 Capturing values from the surrounding scope	183
12.6 Escaping closures (executed later)	184
12.7 Closures and `self` (strong reference cycles).....	185
12.8 A note for Swift 6: concurrency and @Sendable.....	186
Index.....	187
About the Author.....	203

Preface

The Pascal Programming Language

The Pascal programming language was designed by [Nicklaus Wirth](#) at the Federal Institute of Technology in Zürich, Switzerland, in 1970. Pascal is descended from the Algol programming language, which was first described in 1958. By the 1980's Pascal had become the standard programming language used to teach college-level computer science. It evolved to become an object-oriented language (Object Pascal) and was used in the development of the original Apple Macintosh operating system. In time, Pascal was eclipsed by other languages, particularly C, C++, Objective-C and Java. Pascal lives on today as [Delphi Pascal](#) and the [Free Pascal](#) Compiler (FPC).

The Swift Programming Language

Swift is a compiled general-purpose object-oriented programming language developed by Apple Inc. and the open-source community. Development of Swift began at Apple in 2010 by [Chris Lattner](#), with the eventual collaboration of many other programmers. Swift was designed to replace Apple's earlier programming language Objective-C, which lacked many modern language features.

First released in 2014, Swift continues to evolve rapidly, while its core language concepts remain stable. As a language it offers strong typing, safety, modern compiler features and semantic expressiveness without abandoning rigor.

Origins of This Book

I began programming in Pascal in 1982, using Apple UCSD Pascal for the Apple II family of computers, and was fascinated by Pascal's support for structured programming - which significantly enhanced the design, implementation, and maintenance of code. The ability to create rich user-defined data types in Pascal was also a powerful feature of the language. At the time, Pascal impressed me as an elegant, compact, intuitive, and extensible language. It was also a "safe" language because of strong type checking and sparse use of pointers.

I discovered Randy Clarke and Stephen Koehler's wonderful book: The UCSD Pascal Handbook - A Reference and Guidebook for Programmers (Prentice-Hall 1982). Using this well-designed book, one could quickly review UCSD Pascal's features, language and syntax - plus code examples.

As I approached learning the Swift programming language, I looked for an equivalent short reference guide. Not finding one that met my needs, I decided to create this book, which is designed to help those familiar with Pascal to begin exploring Swift. Where possible, I try to demonstrate how these two elegant languages intersect, hopefully creating a conceptual bridge between them.

This book is not intended to be a comprehensive reference guide to Swift. Nor does it cover all aspects of the Swift language. Instead, it attempts to map the core concepts of Standard Pascal to the equivalent language concepts in Swift to hopefully facilitate that “Aha!” moment when one gets a firm conceptual foothold on a new programming paradigm.

This second edition (March 2026) corrects typographical errors in the first edition, updates many topics to reflect the evolution of the Swift language and adds a new chapter on Closures.

Henry Lowe
Stanford 2026

The Swift Playgrounds App

While many developers use Xcode for Swift development, Swift Playgrounds provides a lightweight and interactive environment well suited for experimenting with the examples in this book. Playgrounds provides an interactive Read-Eval-Print-Loop (REPL) environment into which you type Swift code and get instant feedback. Launch the Playgrounds app and select ‘New Blank Playground’ from the Mac File Menu or tap ‘+ Playground’ in the iPad version of the app. This gives you a new empty playground into which you can enter Swift code. Entering ‘import Foundation’ as the first line of the playground ensures that Swift’s Foundation framework is available when you use the examples in this book.

Code Examples

Throughout this book you will find illustrative code examples.

Pascal code examples are printed as plain text, e.g.:

```
Program Hello;  
    Var Greeting:String;  
    Begin  
        Greeting:='Hello world';  
        Writeln(Greeting);  
    End.
```

Swift code examples are displayed within a colored text box to distinguish them visually from Pascal examples and surrounding prose:

```
let greeting = "Hello world"  
print(greeting)
```

Swift's Print() Statement

Swift's `print()` statement is used in the code examples throughout this book. The `print()` statement sends output to the console. If you are using the Playgrounds app to experiment with Swift, the `print()` statement is a useful way to display constants, variables and data in the console pane of a playground window. For example, enter `'print("Hello world!")'` and the text `'Hello world'` is displayed in the playground's console pane.

When using Swift's `print()` statement all non-string variables are converted to strings automatically.

The full definition of `print()` is:

```
print(items, separator, terminator)
```

The ‘items’ parameter contains what we wish to display either as a literal, constant, variable or expression.

The optional ‘separator’ parameter defines the character used to separate multiple items within `print()`. The default separator is a single space character.

The optional ‘terminator’ parameter defines the character used to terminate the display output. The default terminator is newline character “`\n`”. Alternatives include: the tab character “`\t`” and the space character.

We can use these optional `print()` parameters as follows:

```
let word1 = "hello"  
let word2 = "world"  
print(word1,word2,separator:"-",terminator:"\t")  
// displays hello-world
```

References

The definitive Swift reference text is Apple's 'The Swift Programming Language', which is available for free from the Apple Book Store via Apple's Books app. I highly recommend using this book as your standard Swift reference.

Online Swift documentation is available on the Apple Developer Site (some content requires an account):

<https://developer.apple.com/documentation/swift/>

and on the Swift Open Source site at:

<https://www.swift.org>

1 Introduction

Pascal and Swift share many core conceptual features.

1.1 The Form of a Program

A Pascal program has the general structure shown below:

```
Program HelloWorld;  
  Var (* Declare a variable *)  
      Greeting: String;  
Begin  
  Greeting:='Hello World';  
  Writeln(Greeting);  
End.
```

A Swift program may be written as:

```
var greeting: String //Declare a variable  
greeting = "Hello World"  
print(greeting)
```

Let's start with some obvious differences in the example:

Unlike Pascal, Swift has no construct to indicate the start of a program. A Swift program begins with the first line of code, in this case: `var greetings: String`.

Pascal requires that statements end with a semicolon. This is not a requirement in Swift. However, a semicolon can be used to separate multiple statements written in the same line of Swift source code. While Swift does not require a semicolon after each statement, you can use one if you wish.

A major difference between Pascal and Swift is Swift's (and many other modern languages) use of 'curly braces' i.e. `{ }` to define blocks of code.

A code block defines a scope (the set of all variable-name bindings visible to the compiler within a part of a program) that determines the lifetime of variables declared within the block. Variables declared within a block are not valid when control exits the block. Curly braces are also used to define the beginning and end of logical programming structures such as loops. Pascal uses `Begin` and `End` statements to define blocks of code. More on the use of curly braces in Swift later.

A single line comment in Swift begins with `//` and multi-line comments begin with `/*` and end with `*/`. In Pascal source code comments can be enclosed in 'curly brackets' or parentheses e.g. `{Comment}` (`* Comment *`).

1.2 Declaration

All variables in a Pascal program have a specific type. Some types are built into the language, while others are user-defined. Swift follows a similar model. However, Swift's compiler can often infer types automatically, reducing the need for explicit type declarations in many cases.

1.2.1 Case Sensitivity

Pascal is a case-insensitive language. You can use all uppercase, all lower case or a mix of cases when writing Pascal code. Identifiers, constants, and variables are also case-insensitive. One consequence of this is that in Pascal the variable 'MyVariable' and 'myVariable' reference the same data.

Swift is a case-sensitive language: 'MyVariable' and 'myVariable' are two completely different variables.

One common problem that Pascal programmers encounter when starting to use Swift is forgetting to use the correct case when entering identifiers. Most Swift keywords and

standard functions are written in lowercase. For example, use 'print' not 'Print' or 'PRINT':

```
print("Hello World") // is fine
Print("Hello World") // fails - uppercase 'P'
```

Swift's built-in data types are written with the initial character in uppercase e.g.: String, Int, Double, Bool, Array etc.

1.2.2 Constants

Constants are declared once in a program and their value cannot be changed during program execution. Constants can be declared anywhere in a program, but they must be declared before they are referenced. In Pascal constants are declared using the Const keyword:

```
Const
Dozen = 12;
Pi = 3.14159;
ErrorMessage = 'An error occurred';
```

Swift does not use the Const keyword but instead declares constants using the 'let' keyword e.g.

```
let Dozen: Int = 12
let Pi: Double = 3.14159
let errorMessage: String = "An error occurred"
```

As in Pascal, Swift does not require that you define a constant's type, but you can if you wish. This is referred to as 'Type Annotation'

In Swift multiple constants can be declared on a single line separated by commas e.g:

```
let minValue: Int = 1, maxValue: Int = 99
```

```
let minValue = 1, maxValue = 31
```

OR

As in Pascal, Swift also allows the use of expressions in constants:

```
let square2 = 2 * 2
```

In Swift, for performance reasons, it is recommended that you use constants to hold data that will not change during program execution.

1.2.3 Variables

In Swift (as in Pascal) a variable's value can be changed during program execution. Variables must be declared before they are referenced. Swift variables are declared as:

```
var userName: String
var startValue: Int = 100
var mean, median, mode: Int
```

Note that Swift's variable declarations do not require ending the declaration with a semicolon, as is the case in Pascal, though you can add a semicolon if you so desire.

In Swift the following variable declarations are legal:

```
var userName: String
var userName: String;
```

In Swift it is usually not necessary (or required) to declare the variable type (Type Annotation), as the Swift compiler

can usually infer the type from the declaration (Type Inference). The following variable declarations are legal:

```
var greetingMessage = "Hello World"  
var startValue = 10
```

Once a variable or constant has been declared to hold values of a specific type, you cannot redeclare that constant or variable later in the program to hold values of a different type. This strong typing is enforced at compile time.

1.2.4 Constant and Variable Names

Swift constant and variable names can contain almost any character, including Unicode characters e.g.:

```
let 🐸 = "Frog"
```

While Unicode identifiers are supported, they are used sparingly in practice and primarily for illustrative purposes.

Variable names cannot start with a number. Variable names cannot contain whitespace characters, math symbols, arrows, private-use Unicode scalar characters or line and box-drawing characters.

It is recommended (for readability purposes) to use a ‘Camel case’ naming convention when naming constants and variables in Swift. Camel case uses a lowercase letter for the first character of the first ‘word’ in a variable name followed by a capital letter for each subsequent ‘word’ e.g:

```
var myVariableName: String
```

Pascal programmers often use a similar convention, called ‘Pascal Case’ - in which the first character of a constant or variable name made of compound words is upper case - as opposed to lower case in Swift.

Var: MyFileName: String; - Pascal Case

var myFileName: String - Camel Case in Swift

Constant and variable names in Swift can be of any length and all characters are significant. As Swift is a case-sensitive language the following are considered separate variables:

```
var MYVARIABLENAME: String  
var myvariablename: String  
var myVariableName: String
```

2 Operators

Pascal and Swift support a similar set of operators, though with different syntax.

2.1 Assignment

Pascal uses the ‘:=’ operator for assignment. For example:

```
Var Int1: Integer;  
Int1:=42;
```

Swift uses the ‘=’ operator for assignment. For example:

```
var int1:Int  
int1 = 42
```

2.2 Equality

The equality operator in Pascal is ‘=’. For example:

If int1 = int2 then ...

In Swift the equality operator is '==', e.g.:

```
var int1,int2:Int
int1 = 1
int2 = 1
if int1 == int2 {
    let equal = true
}
```

2.3 Non-Equality

In Pascal we test for non-equality using the '<>' operator:

If int1 <> int2 then ...

In Swift we test for non-equality using the '!=' operator:

```
var int1,int2:Int
int1 = 1
int2 = 1
if int1 != int2 {
    let equal = false
}
```

In both Pascal and Swift to test if one variable is less than another we use the '<' operator, and use the '>' operator to test if one variable is greater than another:

```
var int1,int2:Int
int1 = 1
int2 = 1
if int1 > int2 {
    let greaterThan = false
}
```

The Greater Than Or Equals To operator '>=' and the Less Than Or Equals To operator '<=' are also the same in both Pascal and Swift.

2.4 Arithmetic Operators

Pascal and Swift use the same operators for addition '+', subtraction '-', division '/' and multiplication '*':

```
var int1,int2,int3:Int
int1 = 1
int2 = 1
int3 = (int1 + int2) * (int2 / int1) - int1
```

It is worth noting that, as in Pascal, division behavior in Swift depends on operand types. Integer division truncates the result, while floating-point division preserves fractional values.

The Remainder operator returns the remainder when one number is divided by another. In Pascal this is called ‘Mod’, which is short for Modulo:

```
Int3:=int2 Mod Int1;
```

Swift uses the Remainder operator ‘%’ rather than “modulo”, though the behavior is similar for positive integers:

```
var int1,int2,int3:Int  
int3 = int2 % int1
```

In both Pascal and Swift one can change the sign of a number by prefixing a numeric value with the ‘-’ operator (the Unary Minus Operator). This operator toggles the sign of a number:

```
var int1:Int  
int1 = 1  
int1 = -int1 // int1 = -1  
int1 = -int1 // int1 = 1
```

The Unary Plus Operator ‘+’ does not change the sign of the number and its inclusion in both Pascal and Swift appears to be for symmetry.

2.5 Logical Operators

Both Pascal and Swift support the Logical Operators AND, OR and NOT:

The AND operator is 'AND' in Pascal and '&&' in Swift.

In Pascal:

```
Var Bool1,Bool2:Boolean;  
Bool1:=True;  
Bool2:=False;  
If Bool1 AND Bool2 then  
    Writeln('True')  
Else  
    Writeln('False');
```

In Swift:

```
var bool1:Bool = true  
var bool2:Bool = false  
  
if bool1 && bool2 {  
    print ("True")  
} else {  
    print ("False")  
}
```

The OR operator is 'OR' in Pascal and '||' in Swift.

In Pascal:

```
Var Bool1,Bool2:Boolean;  
Bool1:=True;  
Bool2:=False;  
If Bool1 OR Bool2 then  
    Writeln('True')  
Else  
    Writeln('False');
```

In Swift:

```
if bool1 || bool2 {  
    print ("True")  
} else {  
    print ("False")  
}
```

As in many modern languages, Swift's logical AND (&&) and OR (||) operators use short-circuit evaluation. Short-circuit evaluation means that a logical expression is evaluated from left to right, and evaluation stops as soon as the final result is known.

The NOT operator is 'NOT' in Pascal and '!' in Swift.

In Swift Logical NOT is a prefix operator (i.e., appears just before the value it operates on, without whitespace) that toggles a Boolean value.

```
let bool1 = false
let bool2 = !bool1 // bool2 = true
```

In Pascal NOT is used in a similar fashion:

```
Bool1:=False;
Bool2:=Not Bool1; {Bool2 is True}
```

2.6 Range Operators

Swift includes several range operators that define a range of values. The closest equivalent construct in Pascal are subranges. In Pascal we can define a type, constant or variable as a subrange of any scalar type. For example:

```
digits = 0..9; (* Subrange of Integer *)
Grade = "A,B,C,D,E,F" (* User-defined type *)
pass = 'A'..'C'; (* Subrange of Grade *)
```

Pascal may also use subranges when defining arrays:

```
myArray[1..99] of String;
```

Swift's Closed Range Operator ('x...y') defines a range extending from the value of x to the value of y. For example, 2...7 is a range of integers from 2 to 7. The value of x must not be greater than the value of y. Note the three

periods used by Swift as opposed to the two periods used in Pascal.

Swift's Half-Open Range Operator ($x..<y$) defines a range starting with the value of x but not including the value of y . For example, $2..<7$ is a range of integers starting with 2 and ending with 6. This type of range operator is useful when working with zero-based lists, such as Swift arrays. The value of x must not be greater than the value of y . Note the use of two periods in this type of range.

Swift's One-Sided Range operator defines a range that starts with a value and extends as far as possible beyond that start value. For example, if a Swift array contains 100 members, e.g. `Array[0..99]` then the One-Sided Range `[9...]` will include the array members `Array[9]` to `Array[99]`. We can also use a form of the One-Sided Range in which the left side of the range is undefined, e.g. `[...9]` would reference array members 0 to 9.

Ranges are often used in Swift for-in loops. In Pascal we might define a for-loop as:

```
for index:= 1 to 9
    writeln(index);
```

In Swift we can iterate in a similar fashion using ranges:

```
var index1, index2, index3: Int
var myArray: [Int] = [1,2,3,4,5]
for index1 in 1...9 {
    print(index1) }
for index2 in 1..<10 {
    print(index2) }
for index3 in myArray[1...] {
    print(myArray[index3]) }
```

Because Swift ranges are a type, there are range properties and methods that we can use:

```
let myRange = 0...9
print(myRange.contains(5)) //true
print(myRange.lowerBound) // 0
print(myRange.upperBound) // 9
print(myRange.isEmpty) //false
```

Swift also supports comparing ranges:

```
let myRange = 0...9
let anotherRange = 1...8
print(myRange == anotherRange) //false
print(myRange != anotherRange) //true
```

And checking if ranges overlap:

```
print(myRange.overlaps(anotherRange)) //true
```

There are a number of Swift operators that have no clear equivalent in Pascal. Two that are used frequently are Ternary Conditional and Compound Assignment.

Swift supports a Ternary Conditional operator that provides a compact way to choose between two values based on a Boolean condition. It has the form `condition ? value-If-True : value-If-False`. If the condition evaluates to true, the expression returns the first value; otherwise, it returns the second. While Pascal does not have an exact equivalent, the ternary operator is best thought of as a concise, expression-based form of an `if ... then ... else` statement.

```
let temperature = 30
let description = temperature > 25 ? "Warm" : "Cool"
print(description) // displays "Warm"
```

Swift supports compound assignment operators that combine an arithmetic operation with assignment. Instead of writing a variable on both sides of an assignment, the operation and assignment are expressed together using operators such as `+=`, `-=`, `*=`, `/=`, and `%=`. These operators update the variable by applying the operation to its current value and assigning the result back to the same variable.

```
Var total = 10
Total += 5
print(total) // displays 15
```

Swift includes a nil-coalescing operator (??) that is used with optional values (explained later). It returns the value wrapped in an optional if that value is present; otherwise, it returns a specified default value. This allows a program to supply a safe fallback when a value may be missing, without requiring an explicit if statement to test for nil.

```
let userName: String? = nil
let displayName = username ?? "Guest"
print(displayName) // displays "Guest"
```

3 Data Types

Pascal and Swift share many core data types. In this chapter we will discuss the following data types:

- Integer
- Floating Point Number
- Double Precision Number
- Boolean
- Character
- String
- Array
- Set
- Dictionary
- Structure
- Enumeration

3.1 Numbers

3.1.1 Integer

Integers are numbers that contain no fractional component and may be signed or unsigned e.g., -99, +99, 99. Swift supports signed integers in 8, 16, 32, and 64-bit sizes named:

Int8 (-128 to +127)

Int16 (-2^{15} to $2^{15}-1$)

Int32 (-2^{31} to $2^{31}-1$)

Int64 (-2^{63} to $2^{63}-1$)

Index

@

@Sendable, 186

A

Absolute value of a number, 33

Access Control, 172

Fileprivate, 173

Internal, 173

Open, 172

Private, 173

Public, 172

Algol programming language, 1

And, 20

Array, 22, 23, 58, 68, 76, 77

Accessing all elements, 63

Appending new elements, 62

Changing elements, 63

Counting elements, 61

Declaration, 59

Determining if empty, 62

Element operations, 61

Filtering, 69

Index, 61

Initialization, 59

Inserting elements, 62

Obtaining element index and value, 63

Removing elements, 62

Retrieving elements, 63

- Searching, 64
- Shuffle, 71
- Zero-indexed, 59

Assignment, 16

Associative Array, 76

async throws, 163

B

Bool type in Swift, 35

Booleans, 35

Break statement, 116

Break to label, 117

C

Case sensitivity, 10

catch, 160

Ceil(), 30

Chaining of methods, 145

Change the sign of a number. See Operators, Unary Minus

Character, useful methods, 37

Characters, 37

Closures, 180

Closures as Function Parameters, 181

Code block, definition, 9

Comments, in source code, 9

ComparisonResult, 50, 66

concurrency, 163, 180, 186

Concurrency, 163, 186

Constant and variables, naming, 14

Constants, 11

- Continue statement, 117
- Control Flow, 104
- Control transfer statements, 116
- Converting Numbers to Strings, 34
- Curly braces { }, 9

D

Data

- Declaration, 153
- Iteration, 154
- Reading and writing data files, 154
- Splitting, 154

Data type, 152

Data types, 10

Data Types

- Double, 28
- Float, 28
- Floating point numbers, 28

Defer statement, 138

Delphi Pascal, 1

Dictionary

- Adding entries, 78
- Case sensitivity, 83
- Changing entries, 78
- Converting to array, 82
- Counting key-value pairs, 83
- Creating from arrays, 77
- Declaration, 76
- Definition, 76
- Display values, 80
- Empty, 77

- Extensions, 84
- Iteration, 82
- Removing keys, 81
- Removing key-value pair, 81
- retrieving key list, 82
- Retrieving value list, 82
- Searching, 83
- Sorting keys and values, 83
- Subscript syntax, 78
- Subscripts, 78
- Updating entries, 78

Do-Catch Statement, 160

Double, 28

E

Emulating Pascal procedures, 128

eNum, 50

Enum

- AllCases, 95
- Associated values, 96
- Case, 94
- CaseIterable, 95
- Count cases, 95
- Creating new constants or variables from, 95
- Declaration, 94
- Dot notation, 96
- List case labels, 95
- Raw values, 96
- Using collection properties with, 99

Enumerations, 96, 157, See eNum

Equality, 16

Error Handling, 157

- Error handling with optionals, 161
- Error Protocol, 157
- Escaping closures, 184
- Exponential function, 34
- Extension Keyword, 166
- Extensions, 165

F

- Fallthrough statement, 118
- File Management, 140
- Filemanager
 - Creating a new directory, 149
 - Creating a new file, 151
 - Declaration, 147
 - Description, 146
 - Methods and properties, 147
 - Obtaining path to directory, 148
 - Obtaining path to Documents directory, 147
 - Reading a file, 152
 - User domain mask, 148
 - Writing to a file, 151
- FileManager, 146
- Fileprivate access level, 173
- Float, 28
- Floating point numbers, 28
- Floor(), 30
- Free Pascal Compiler, 1
- Func. See Function
- Function
 - Ampersand '&', 129
 - Argument labels, 123

- Calling, 122
- Calling without parameters, 125
- Defer, 138
- Definition, 122
- Generic, 137
- Inout keyword, 128
- Mutating, 134
- Parameter with multiple values, 127
- Parameters, 125
- Return keyword, 124
- Return values, 124
- Throwing, 158
- Type parameter, 137
- Using default parameter values, 126
- Variadic parameters, 127

Functions and procedures, 119

G

- Generic functions, 137
- Generics, 59
- Guard Statement, 106

H

- Hash Table, 76
- Hash value, 77
- Hashable, 77

I

- If Statement, 104
- If Then Else Statement, 105
- Import Keyword, 170

Integer, 27

Internal access level, 173

J

JSON, 155

K

Key-value pair, 81

L

Lattner, Chris, 1

Let, 12

Logarithm, 34

Logical Operators, 20

LowerCamelCase, 89

M

Mod. See Operators, Remainder

Modules, 170

Modulo. See Operators, Remainder

Mutating methods, 93

N

Namespace, 170

Non-equality, 17

Not, 21

Numbers, 27

- Absolute value, 33

- Convert to String, 35

- Exponential function, 34

- Random, 32
- Rounding, 30
- Rounding down, 30
- Rounding rules, 31
- Rounding up. See Ceil()
- Rounding, decimal places, 31
- Square, 34
- Square root, 33
- Truncation, 31

O

- Objective-C, 1
- Open access level, 172
- Operators
 - Addition, 18
 - And, 20
 - Arithmetic, 18
 - Assignment, 16
 - Division, 18
 - Equality, 16
 - Logical, 20
 - Multiplication, 18
 - Non-equality, 17
 - Not, 21
 - Or, 20
 - Range, 22
 - Remainder, 19
 - Subtraction, 18
 - Unary Minus, 19
 - Unary Plus, 19
- Optionals, 44, 45, 79, 80, 81, 133
- Or, 20

P

parameters, 128

Pascal

- Access control, 172
- Apple UCSD Pascal, 2
- Arrays, 58
- Begin .. End, 105
- Boolean, 35
- Break, 116
- Case, 108
- Case sensitivity, 10
- Char, 36
- Code blocks, 105
- Code examples, 4
- Comments, source code, 9
- Concat, 40
- Constants, 11
- Control Flow, 104
- Copy, 40
- Data types, 27
- Declaration, 10
- Delete, 51
- Delphi, 1
- Down to, 112
- Emulating procedures in Swift, 128
- File, 140
- File management, 140
- For, 111
- Free Pascal Compiler, 1
- Functions, 119
- Goto, 116
- Halt, 116

- History, 1
- If, 104
- If then Else, 105
- Insert, 51
- Length, 40
- Numerics, 27
- Operators, 16
- Parameters, 120
- Pos, 40
- Procedures, 119
- Program structure, 8
- Random function, 32
- Real data type, 28
- Records, 86
- Repeat loops, 111
- Repeat Until, 115
- Repeat While, 114
- Return, 119
- Round function, 30
- Semicolon, ending statements with, 9
- Sets, 71
- Str function, 34
- Strings, 36
- Subranges, 22
- Trunc function, 31
- UCSD Pascal Handbook, 2
- Units, 171
- Value parameters, 120

Pascal case, 15

Pascal's Copy(Source,Index,Size) function, 43

Power function, numeric, 34

Print() statement, 5

Private access level, 173

Protocols as Types, 179
Public access level, 172

R

Random boolean, 33

Random numbers, 32

Range

 Closed ('x...y'), 22

 Comparing, 24

 Contains, 24

 Half-open (x..<y), 23

 IsEmpty, 24

 Lowerbound, 24

 Methods, 24

 One-sided ('x...') or ('...y'), 23

 Overlap, 24

 Upperbound, 24

 Use in iteration, 111

 Used in loops, 23

Read Eval Print Loop (REPL), 4

Records and Structures, 86

Reference types, 134

Remainder, 19

Repeat Loops, 111

Repeat while loops, 114

Rounding numbers, 30

S

Sandboxing, 144

Scope, definition, 9

Self, 134, 166

Semicolon, ending statements with, 9

Set

Asymmetric difference, 74

Declaration, 72

Definition, 72

Intersection, 74

Iteration, 74

Membership tests, 75

Operations, 73

Sorting, 76

Subtract, 74

Union, 74

Sets, 71

Square root of a number, 33

Static Keyword, 177

Stride, 113

string, 37

String Character Indices, 41

String Comparison, 49

String Concatenation, 47

String Insertion and Removal, 51

String Interpolation, 50

String Length, 40

String Literals, 38

String Replace, 55

String Search, 54

String Split, 56

Strings

Changing case, 54

Comparison, 49

Concatenation, 47

- Embedded special characters, 39
- Extended special characters, 203
- Extending Swift Strings, 41
- Indices, 42
- Initialization, 38
- Initialization from a file, 38
- Inserting characters, 51
- Length, 40
- Literals, 38
- Multiline literals, 39
- NSString.CompareOptions, 56
- Removing characters, 52
- Replacing characters, 55
- Searching, 54
- Splitting, 56
- Substrings, 57
- Testing if empty, 39

Strings and characters, 36

StringStrings

- Interpolation, 50

Struct

- Computed properties, 90
- Create new instance, 89
- Declaration, 87
- Default property values, 89
- Defining new, 88
- Embedded functions. See Instance methods
- Getter and setter methods, 90
- Getting and setting property values, 90
- Initializing properties, 89
- Instance methods, 91
- Properties, 89
- Self reference, 92

Subscript syntax, 63, 78, 79, 80

Substrings, 57

Swift

- Brief history, 1

- Case sensitivity, 10

- Code examples, 4

- Let keyword, 12

- Online documentation, 7

- Playgrounds app, 4

- Print() statement, 5

- Program structure, 8

- Swift.org*, 7

- The Swift Programming Language Reference, 7

Swift Playgrounds App, 4

Switch default case, 108

Switch Statement, 108

T

Throw Statement, 158

Throws keyword, 158

Trailing Closures, 181

Truncating Numbers, 32

try, 158

Try Statement, 161

Try!, 159, 160

Try?, 161

Tuple, 130, 131, 132, 133, 162, 163

Tuple Definition, 131

Tuples, 131, 133

Tuples in error handling, 162

Type annotation, 12

Type Conversion

Numbers, 29

Type inference, 14

U

Unicode, 14, 36, 37, 41

Universal Remote Locator. See URL

Unwrapping optionals, 80

UpperCamelCase, 88

URL

Accessing components, 146

As file reference, 142

Components, 141

Declaration, 142

Definition, 141

Determining if file is reachable, 146

Methods, 144

Obtaining file extension from, 146

Obtaining file name from, 146

Referencing relative to Home directory, 145

Standardized, 145

V

Value types, 93, 134

Variables, 13

Variadic parameter, 48, 127

Void, 130

W

Wirth, Nicklaus, 1

About the Author

Henry Lowe MD, FACMI is an Emeritus Professor at Stanford University and a Board-Certified Internal Medicine physician. From 2002 until 2013 Dr. Lowe was Chief Information Officer and Senior Associate Dean for Information Resources and Technology at Stanford University School of Medicine. He was also the founding Director of Stanford's Center for Clinical Informatics. He has extensive experience in clinical informatics research and development. Dr. Lowe is an elected Fellow of the American College of Medical Informatics (ACMI) and the Founder of Ascriva Health Informatics, a company based in Silicon Valley, California.