The *Tiny Swift* Language Reference

About

This reference is based on *The Swift Programming Language (Swift 3)* published by *Apple Inc*. In its *Language Reference* part, the whole structure of the Swift programming language is introduced. My *Tiny Swift* is a lite version of the formal *Swift* programming language. It chooses some basic syntaxes of *Swift* and describes the grammar as follows. My job is not merely copying and pasting; rather, I spend lots of time deciding which part of the language should be taken and how to simplify the grammar. I hope this *Tiny Swift* can accomplish most of the jobs *Swift* can do.

Lexical

A meaningful token consists of an **identifier**, **keyword**, **literal**, or **operator**.

Whitespace and Comments

Whitespace is used in separating tokens or otherwise ignored.

$$whitespace \rightarrow \mathbf{SP} \mid \mathbf{\backslash r} \mid \mathbf{\backslash n} \mid \mathbf{\backslash t} \mid \mathbf{EOF}$$

Comments are treated as whitespace. Only single-line comment is allowed.

```
// this is a comment
```

A comment starts with **//** and end with **\r** or **\n**.

Identifiers

An *identifier* begins with an uppercase or lowercase letter A through Z or an underscore `_`. After the first character, digits are also allowed.

```
// these are identifiers
_aVar
tmp_0
i
```

$$\begin{aligned} identifier &\rightarrow identifier\text{-}head \quad identifier\text{-}character^{*} \\ identifier\text{-}head &\rightarrow [\mathbf{a\text{-}zA\text{-}Z\_}] \\ identifier\text{-}character &\rightarrow identifier\text{-}head \mid [\mathbf{0\text{-}9}] \end{aligned}$$

Keywords

The following keywords are reserved and can't be used as identifiers.

- **Keywords used in declarations:** *class*, *enum*, *func*, *let*, *static*, *struct*, *var*.

- **Keywords used in statements:** *break*, *case*, *continue*, *default*, *do*, *else*, *fallthrough*, *for*, *guard*, *if*, *in*, *return*, *repeat*, *switch*, *where*, *while*.
- **Keywords used in expressions and types:** *false*, *true*.
- **Keywords reserved in particular contexts:** *optional*.

Literals

A *literal* is the source code representation of a value of a type, such as a number or a string.

```
42              // Integer Literal
3.14159         // Floating-point literal
"Hello, world!" // String literal
true            // Boolean literal
```

Swift infers that the literal's type is one of the default literal types. The default types are:

- **Int** for integer literals
- **Double** for floating-point literals
- **String** for string literals
- **Bool** for boolean literals

For example, in the declaration `let str = "Hello, world"`, the default inferred type for *str* would be *String*.

$$\begin{aligned} \textit{literal} &\rightarrow \textit{numeric-literal} \mid \textit{string-literal} \mid \textit{boolean-literal} \\ \textit{numeric-literal} &\rightarrow \texttt{-}?\ \textit{integer-literal} \mid \texttt{-}?\ \textit{floating-point-literal} \\ \textit{boolean-literal} &\rightarrow \textbf{true} \mid \textbf{false} \end{aligned}$$

Next, we'll discuss these literals respectively.

Integer Literals

Integer literals represent integer values and are expressed in decimal. It contains the digits 0 through 9. Negative integer literals are expressed by appending a minus sign `-` to an interger literal. Integer literals can begin with leading zeros.

```
-001234567  // this is an integer literal
```

$$\textit{integer-literal} \rightarrow [\textbf{0-9}]^{+}$$

Floating-point Literals

Floating-point literals represent floating-point values and are expressed in decimal. It consist of a sequence of decimal digits followed by either a decimal fraction, a decimal exponent, or both. The decimal fraction consists of a decimal point `.` followed by a sequence of decimal digits. The exponent consists of an upper- or lowercase e prefix followed by a sequence of decimal digits that indicates what power of 10 the value preceding the e is multiplied by.

```
-12.34e+34  // represents -12.34*(10^34)
```

$$\begin{aligned} \textit{floating-point-literal} &\rightarrow \textit{integer-literal}\quad \textit{fraction}?\quad \textit{exponent}? \\ \textit{fraction} &\rightarrow \mathbf{.}\quad \textit{integer-literal} \\ \textit{exponent} &\rightarrow [\mathbf{eE}]\quad [\mathbf{+\text{-}}]?\quad \textit{integer-literal} \end{aligned}$$

String Literals

A *string literal* is a sequence of characters surrounded by double quotes("). The valid characters include: `a-z`, `A-Z`, `0-9`, `!@#%$^&*()`, `_-+=~`, `|{}[]:;<>,.?/`.

```
"das839(*#<>:}["    // a string literal
```

Operators

An *operator* is a special symbol or phrase that you use to check, change, or combine values. Operators include:

- **Assignment Operator:** `=`
- **Arithmetic Operator:** `+`, `-`, `*`, `/`
- **Remainder Operator:** `%`
- **Comparison Operator:** `==`, `!=`. `>`, `<`, `>=`, `<=`
- **Logical Operator:** `!`, `&&`, `||`

$$\begin{aligned} \textit{operator} &\rightarrow \textit{operator-character}^{+} \\ \textit{operator-character} &\rightarrow = \mid + \mid \text{-} \mid * \mid / \mid \% \mid ! \mid > \mid < \mid \& \mid \mid \\ \textit{prefix-operator} &\rightarrow \textit{operator} \\ \textit{binary-operator} &\rightarrow \textit{operator} \\ \textit{postfix-operator} &\rightarrow \textit{operator} \end{aligned}$$

Types

The *type* can be basic types or arrays and dictionaries of those basic types.

$$\textit{type} \rightarrow \textit{array-type} \mid \textit{dictionary-type} \mid \textit{type-identifier}$$

Type Identifier

A *type identifier* refers to a basic type.

$$\textit{type-identifier} \rightarrow \mathbf{Int} \mid \mathbf{Double} \mid \mathbf{Bool} \mid \mathbf{String}$$

Array Type

An *array type* represents an array of the same types.

```
[[String]]  // an array type
```

$$array\text{-}type \rightarrow [\ type\ ]$$

Dictionary Type

A *dictionary type* represents a dictionary composed of two types.

```
[Int:String]    // a dictionary type
```

$$dictionary\text{-}type \rightarrow [\ type\ :\ type\ ]$$

Type Inference

Swift uses type inference to let you omit the type or part of the type of many variables and expressions in your code. For example, instead of writing:

```
var a: Int = 6
```

you can write:

```
var a = 6
```

adn the compiler with infer that the variable a has the type *Int*.

Expressions

In Swift, there are four kinds of expressions:

- prefix expressions
  - **op** expr
- binary expressions
  - expr **op** expr
- primary expressions
  - the simplest kind of expression
  - "basic expression"
- postfix expressions
  - function call / member access

Evaluating an expression returns a value, causes a side effect, or both.

$$expression \rightarrow prefix\text{-}expression\quad binary\text{-}expression^{*}$$

Prefix Expressions

Prefix expressions combine an optional prefix operator with an expression. Prefix operators take one argument, the expression that follows them. Also, if it is a simple expression (with no operator), then the prefix operator might be none.

```
-(3+5)  // interpreted as OP(-) postfix-expression(3+5)
```

$$\textit{prefix-expression} \rightarrow \textit{prefix-operator}? \quad \textit{postfix-expression}$$

Binary Expressions

Binary expressions combine an infix binary operator with the expression that it takes as its left-hand and right-hand arguments:

`left-hand argument` `operator` `right-hand argument`

```
6 + 7   // a binary operation
a = 6   // another binary operation
```

$$\begin{aligned}\textit{binary-expression} &\rightarrow \textit{binary-operator} \quad \textit{prefix-expression} \\ &\quad | \ \textit{assignment-operator} \quad \textit{prefix-expression} \\ &\quad | \ \textit{conditional-operator} \quad \textit{prefix-expression} \\ \textit{assignment-operator} &\rightarrow = \\ \textit{conditional-operator} &\rightarrow ? \quad \textit{expression} \quad :\end{aligned}$$

Primary Expressions

Primary expressions are the most basic kind of expression. They can be used as expressions on their own, and they can be combined with other tokens to make prefix expressions, binary expressions, and postfix expressions.

$$\begin{aligned}\textit{primary-expression} &\rightarrow \textit{identifier} \\ &\quad | \ \textit{literal-expression} \\ &\quad | \ \textit{parenthesized-expression}\end{aligned}$$

Literal Expression

A *literal expression* consists of either an ordinary literal (such as a string or a number), an array literal, or a dictionary literal.

```
12345             // literal
[1,2,3,4]         // array literal
["id":16, "age":20] // dictionary literal
```

$$
\begin{aligned}
\textit{literal-expression} &\to \textit{literal} \\
&\mid \textit{array-literal} \\
&\mid \textit{dictionary-literal} \\
\\
\textit{array-literal} &\to [\quad \textit{array-literal-items}?\quad] \\
\textit{array-literal-items} &\to \textit{array-literal-item}\quad (\ ,\quad \textit{array-literal-item}\ )^* \\
\textit{array-literal-item} &\to \textit{expression} \\
\\
\textit{dictionary-literal} &\to [\quad \textit{dictionary-literal-items}\quad] \\
&\mid [\quad :\quad] \\
\textit{dictionary-literal-items} &\to \textit{dictionary-literal-item}\quad (\ ,\quad \textit{dictionary-literal-item}\ )^* \\
\textit{dictionary-literal-item} &\to \textit{expression}\quad :\quad \textit{expression}
\end{aligned}
$$

Parenthesized Expression

A parenthesized expression consists of a comma-separated list of expressions surrounded by parentheses. Each expression can have an optional identifier before it, separated by a colon `:`. Use parenthesized expressions to create tuples and to pass arguments to a function call.

```
(a:Int,b:Double)
(a,b)
```

$$
\begin{aligned}
\textit{parenthesized-expression} &\to (\quad \textit{expression-element-list}?\quad) \\
\textit{expression-element-list} &\to \textit{expression-element}\quad (\ ,\quad \textit{expression-element}\ )^* \\
\textit{expression-element} &\to \textit{expression}\quad \mid\quad \textit{identifier}\quad :\quad \textit{expression}
\end{aligned}
$$

Postfix Expressions

Postfix expressions are formed by applying a postfix operator or other postfix syntax to an expression. Syntactically, every primary expression is also a postfix expression.

$$
\begin{aligned}
\textit{postfix-expression} &\to \textit{primary-expression} \\
&\mid \textit{postfix-expression}\quad \textit{postfix-operator} \\
&\mid \textit{function-call-expression} \\
&\mid \textit{explicit-member-expression} \\
&\mid \textit{subscript-expression}
\end{aligned}
$$

Function Call Expression

A *function call expression* consists of a function name followed by a comma-separated list of the function's arguments in parentheses. The *function name* can be any expression whose value is of a function type. If the function definition includes names for its parameters, the function call must include names before its argument values separated by a colon `:`. So a function call can have the following forms:

`function name` (`argument value 1` , `argument value 2`)

`function name` (`argument name 1` : `argument value 1` , `argument name 2` : `argument value 2`)

```
f(1,2)
f(a:1, b:2)
```

$$function\text{-}call\text{-}expression \rightarrow postfix\text{-}expression \quad parenthesized\text{-}expression$$

Explicit Member Expression

An *explicit member expression* allows access to the members of a named type. It consists of a period `.` between the item and the identifier of its member.

`expression` . `member name`

```
array.length
```

$$explicit\text{-}member\text{-}expression \rightarrow postfix\text{-}expression \quad . \quad identifier$$

Subscribe Expression

A *subscript expression* provides subscript access using the getter and setter of the corresponding subscript declaration.

```
a[1]
```

$$subscript\text{-}expression \rightarrow postfix\text{-}expression \quad [\quad experssion \quad]$$

Statements

In *TinySwift*, there are two kinds of statements: simple statements, and control flow statements.

- **simple statement**
  - expression
  - declaration
- **control flow statement**
  - loop statements
  - branch statements
  - control transfer statements

A semicolon `:` can optionally appear after any statement and is used to separate multiple statements if they appear on the same line.

$$
\begin{aligned}
\textit{statement} \rightarrow\ & \textit{expression}\ \ :? \\
\mid\ & \textit{declaration}\ \ :? \\
\mid\ & \textit{loop-statement}\ \ :? \\
\mid\ & \textit{branch-statement}\ \ :? \\
\mid\ & \textit{control-transfer-statement}\ \ :?
\end{aligned}
$$

Loop Statements

Loop statements allow a block of code to be executed repeatedly, depending on the conditions specified in the loop. *TinySwift* has three loop statements

- **for-in** statement
- **while** statement
- **repeat-while** statement.

$$
\begin{aligned}
\textit{loop-statement} \rightarrow\ & \textit{for-in-statement} \\
\mid\ & \textit{while-statement} \\
\mid\ & \textit{repeat-while-statement}
\end{aligned}
$$

For-In Statement

A `for-in` statement allows a block of code to be executed once for each item in a collection (or any type) that conforms to the Sequence protocol.

```
for a in b {
  // a is a member of b
  // do sth
}
```

$$
\textit{for-in-statement} \rightarrow \textbf{for}\ \ \textit{pattern}\ \ \textbf{in}\ \ \textit{expression}\ \ \textit{where-clause}?\ \ \textit{code-block}
$$

While Statement

A `while` statement allows a block of code to be executed repeatedly, as long as a condition remains true.

```
while (a < b) {
  // do sth
  a += 1
}
```

$$
\begin{aligned}
\textit{while-statement} &\rightarrow \textbf{while}\ \ \textit{condition}\ \ \textit{code-block} \\
\textit{condition} &\rightarrow \textit{expression}
\end{aligned}
$$

Repeat-While Statement

A `repeat-while` statement allows a block of code to be executed one or more times, as long as a condition remains true.

```
repeat {
  // do sth
  a += 1
} while (a < b)
```

$$\textit{repeat-while-statement} \to \textbf{repeat} \quad \textit{code-block} \quad \textbf{while} \quad \textit{condition}$$

Branch Statements

Branch statements allow the program to execute certain parts of code depending on the value of one or more conditions. The values of the conditions specified in a branch statement control how the program branches and, therefore, what block of code is executed. Swift has three branch statements:

- **if** statement
- **guard** statement
- **switch** statement

$$\begin{aligned}\textit{branch-statement} \to\ & \textit{if-statement} \\ \mid\ & \textit{guard-statement} \\ \mid\ & \textit{switch-statement}\end{aligned}$$

If Statement

An `if` statement is used for executing code based on the evaluation of one or more conditions.

```
if (a < b) {
  // do sth
} else if (a == b) {
  // sth else
} else {
  // others
}
```

$$\begin{aligned}\textit{if-statement} &\to \textbf{if} \quad \textit{condition} \quad \textit{code-block} \quad \textit{else-clause}? \\ \textit{else-clause} &\to \textbf{else} \quad \textit{code-block} \\ &\mid \textbf{else} \quad \textit{if-statement}\end{aligned}$$

Guard Statement

A `guard` statement is used to transfer program control out of a scope if one or more conditions aren't met.

```
guard a < b else {
  // sth wrong!
}
// do sth
```

$$\textit{guard-statement} \to \textbf{guard} \quad \textit{condition} \quad \textbf{else} \quad \textit{code-block}$$

Switch Statement

A `switch` statement allows certain blocks of code to be executed depending on the value of a control expression.

```
switch a {
case 'a':
    // do sth
case 'b' where c < d:
    // do sth
    fallthrough
default:
    // do sth
}
```

$$\begin{aligned}
\textit{switch-statement} &\to \textbf{switch} \quad \textit{expression} \quad \{ \quad \textit{switch-case}^{*} \quad \} \\
\textit{switch-case} &\to \textit{case-label} \quad \textit{statement}^{+} \\
&\quad \mid \textit{default-label} \quad \textit{statement}^{+} \\
\textit{case-label} &\to \textbf{case} \quad \textit{case-item} \quad : \\
\textit{case-item} &\to \textit{pattern} \quad \textit{where-clause}? \\
\textit{default-label} &\to \textbf{default} \quad : \\
\textit{where-clause} &\to \textbf{where} \quad \textit{expression}
\end{aligned}$$

Control Transfer Statements

Control transfer statements can change the order in which code in your program is executed by unconditionally transferring program control from one piece of code to another. Swift has five control transfer statements

- **break** statement
- **continue** statement
- **fallthrough** statement
- **return** statement

$$\begin{aligned}
\textit{control-transfer-statement} &\to \textbf{break} \\
&\quad \mid \textbf{continue} \\
&\quad \mid \textbf{fallthrough} \\
&\quad \mid \textbf{return} \quad \textit{expression}?
\end{aligned}$$

Declarations

A *declaration* introduces a new name or construct into your program. For example, you use declarations to introduce functions and methods, variables and constants.

$$
\begin{aligned}
\textit{declaration} \rightarrow\ & \textit{constant-declaration} \\
| \ & \textit{variable-declaration} \\
| \ & \textit{typealias-declaration} \\
| \ & \textit{function-declaration} \\
| \ & \textit{enum-declaration} \\
| \ & \textit{struct-declaration} \\
| \ & \textit{class-declaration}
\end{aligned}
$$

Top-Level Code

The top-level code in a Swift source file consists of zero or more statements, declarations, and expressions.

$$\textit{top-level-declaration} \rightarrow \textit{statement}^{*}$$

Code Blocks

A *code block* is used by a variety of declarations and control structures to group statements together.

```
{
  // do sth
}
```

$$\textit{code-block} \rightarrow \{\quad \textit{statement}^{*} \quad \}$$

Constant Declaration

A *constant declaration* introduces a constant named value into your program.

let `constant name` **:** `type` **=** `expression`

```
let a: Int = 6
let b = 5, c:String = "abc"
```

$$
\begin{aligned}
\textit{constant-declaration} &\rightarrow \textbf{let} \quad \textit{pattern-initializer-list} \\
\textit{pattern-initializer-list} &\rightarrow \textit{pattern-initializer} \quad (\ , \quad \textit{pattern-initializer}\ )^{*} \\
\textit{pattern-initializer} &\rightarrow \textit{pattern} \quad \textit{initializer}\,? \\
\textit{initializer} &\rightarrow = \quad \textit{expression}
\end{aligned}
$$

Variable Declaration

A *variable declaration* introduces a variable named value into your program and is declared using the var keyword.

var `constant name` **:** `type` **=** `expression`

```
var a: Int = 6
var b = 5, c:String = "abc"
```

$$\begin{aligned} \textit{variable-declaration} &\rightarrow \textbf{var} \quad \textit{pattern-initializer-list} \\ \textit{pattern-initializer-list} &\rightarrow \textit{pattern-initializer} \quad (\ , \quad \textit{pattern-initializer}\ )^{*} \\ \textit{pattern-initializer} &\rightarrow \textit{pattern} \quad \textit{initializer}\,? \\ \textit{initializer} &\rightarrow = \quad \textit{expression} \end{aligned}$$

Typealias Declaration

A *type alias* declaration introduces a named alias of an existing type into your program. Type alias declarations are declared using the **typealias** keyword and have the following form:

typealias `name` **=** `existing type`

```
typealias A = Int
var a: A = 5
```

$$\begin{aligned} \textit{typealias-declaration} &\rightarrow \textbf{typealias} \quad \textit{typealias-name} \quad \textit{typealias-assignment} \\ \textit{typealias-name} &\rightarrow \textit{identifier} \\ \textit{typealias-assignment} &\rightarrow = \quad \textit{type} \end{aligned}$$

Function Declaration

A *function declaration* introduces a function or method into your program.

func `function name` **(** `parameters` **) ->** `return type` **{**

`statements`

}

```
func f(a:Int) -> Double {
  // do sth
  return a + 3.14
}
```

$$\begin{aligned}
\textit{function-declaration} &\to \textit{function-head} \quad \textit{function-name} \quad \textit{function-signature} \quad \textit{functio} \\
\\
\textit{function-head} &\to \textbf{func} \\
\textit{function-name} &\to \textit{identifier} \\
\textit{function-signature} &\to \textit{parameter-clause} \quad \textit{function-result}? \\
\textit{function-result} &\to \texttt{->} \quad \textit{type} \\
\textit{function-body} &\to \textit{code-block} \\
\\
\textit{parameter-clause} &\to \texttt{(} \quad \texttt{)} \\
&\quad \mid \texttt{(} \quad \textit{parameter-list} \quad \texttt{)} \\
\textit{parameter-list} &\to \textit{parameter} \quad (\texttt{,} \quad \textit{parameter})^* \\
\textit{parameter} &\to \textit{external-parameter-name}? \quad \textit{local-parameter-name} \quad \textit{type-anno} \\
\textit{external-parameter-name} &\to \textit{identifier} \\
\textit{internal-parameter-name} &\to \textit{identifier}
\end{aligned}$$

Enumeration Declaration

An *enumeration declaration* introduces a named enumeration type into your program. The body of an enumeration declared using either form contains zero or more values—called *enumeration cases*.

```
enum A {
  case a,b,c
  case d,e,f
}
```

$$\begin{aligned}
\textit{enum-declaration} &\to \textbf{enum} \quad \textit{enum-name} \quad \texttt{\{} \quad \textit{enum-member}^+ \quad \texttt{\}} \\
\textit{enum-member} &\to \textbf{case} \quad \textit{enum-case-list} \\
\textit{enum-case-list} &\to \textit{enum-case} \quad (\texttt{,} \quad \textit{enum-case})^* \\
\textit{enum-case} &\to \textit{identifier} \\
\textit{enum-name} &\to \textit{identifier}
\end{aligned}$$

Struct Declaration

A *structure declaration* introduces a named structure type into your program. The body of a structure contains zero or more *declarations*.

```
struct a {
  b: Int
  c: Double = 3.14
}
```

$$\begin{aligned}
\textit{struct-declaration} &\to \textbf{struct} \quad \textit{struct-name} \quad \textit{struct-body} \\
\textit{struct-name} &\to \textit{identifier} \\
\textit{struct-body} &\to \texttt{\{} \quad \textit{declaration}^* \quad \texttt{\}}
\end{aligned}$$

Class Declaration

A *class* declaration introduces a named class type into your program. The body of a class contains zero or more *declarations*.

```
class a {
  b: Int = 2
  c: Double = 3.14
}
```

$$\begin{aligned} \textit{class-declaration} &\rightarrow \textbf{class} \quad \textit{class-name} \quad \textit{class-body} \\ \textit{class-name} &\rightarrow \textit{identifier} \\ \textit{class-body} &\rightarrow \{ \quad \textit{declaration}^* \quad \} \end{aligned}$$

Reference

1. The Swift Programming Language (Swift 3.0.1)