Subscripts

Classes, structures, and enumerations can define *subscripts*, which are shortcuts for accessing the member elements of a collection, list, or sequence. **You use subscripts to set and retrieve values by index without needing separate methods for setting and retrieval**. For example, you access elements in an `Array` instance as `someArray[index]` and elements in a `Dictionary` instance as `someDictionary[key]`.

您可以为单个类型定义多个*subscript*，然后根据传递给下标的索引值的类型选择要使用的适当*subscript overload*。*subscript*不限于单个维度，您可以定义具有多个输入参数的*subscript*以满足您的自定义类型的需求。

Subscript Syntax

You write subscript definitions with the `subscript` keyword, and specify one or more input parameters and a return type, in the same way as instance methods. Unlike instance methods, subscripts can be read-write or read-only. This behavior is communicated by a getter and setter in the same way as for computed properties:

```
subscript(index: Int) -> Int {
    get {
        // Return an appropriate subscript value here.
    }
    set(newValue) {
        // Perform a suitable setting action here.
    }
}
```

The type of `newValue` is the same as the return value of the subscript. As with computed properties, you can choose not to specify the setter's `(newValue)` parameter. A default parameter called `newValue` is provided to your setter if you do not provide one yourself.

As with read-only computed properties, you can simplify the declaration of a read-only subscript by removing the `get` keyword and its braces:

```
subscript(index: Int) -> Int {
    // Return an appropriate subscript value here.
}
```

Subscript Options

Subscripts can take any number of input parameters, and these input parameters can be of any type. Subscripts can also return a value of any type.

Like functions, subscripts can take a varying number of parameters and provide default values for their parameters, as discussed in Variadic Parameters and Default Parameter Values. However, unlike functions, subscripts can't use in-out parameters.

一个类或结构可以根据需要提供尽可能多的*subscript*实现，并且将根据使用*subscript*时在*subscript*方括号中包含的一个或多个值的类型来推断要使用的适当的*subscript*。多个下标的定义称为*subscript overloading*。

While it is most common for a subscript to take a single parameter, you can also define a subscript with multiple parameters if it is appropriate for your type. The following example defines a `Matrix` structure, which represents a two-dimensional matrix of `Double` values. The `Matrix` structure's subscript takes two integer parameters:

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(repeating: 0.0, count: rows * columns)
    }
    func indexIsValid(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValid(row: row, column: column), "Index out of
range")
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValid(row: row, column: column), "Index out of
range")
            grid[(row * columns) + column] = newValue
        }
    }
}
```

Values in the matrix can be set by passing row and column values into the subscript, separated by a comma:

```
matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
```

Type Subscripts

Instance subscripts, as described above, are subscripts that you call on an instance of a particular type. You can also define subscripts that are called on the type itself. This kind of subscript is called a *type subscript*. You indicate a type subscript by writing the `static` keyword before the `subscript` keyword. Classes can use the `class` keyword instead, to allow subclasses to override the superclass's implementation of that subscript. The example below shows how you define and call a type subscript:

```
enum Planet: Int {
    case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune
    static subscript(n: Int) -> Planet {
        return Planet(rawValue: n)!
    }
}
let mars = Planet[4]
print(mars)
```