iOS 程式設計

Learn the Essentials of Swift

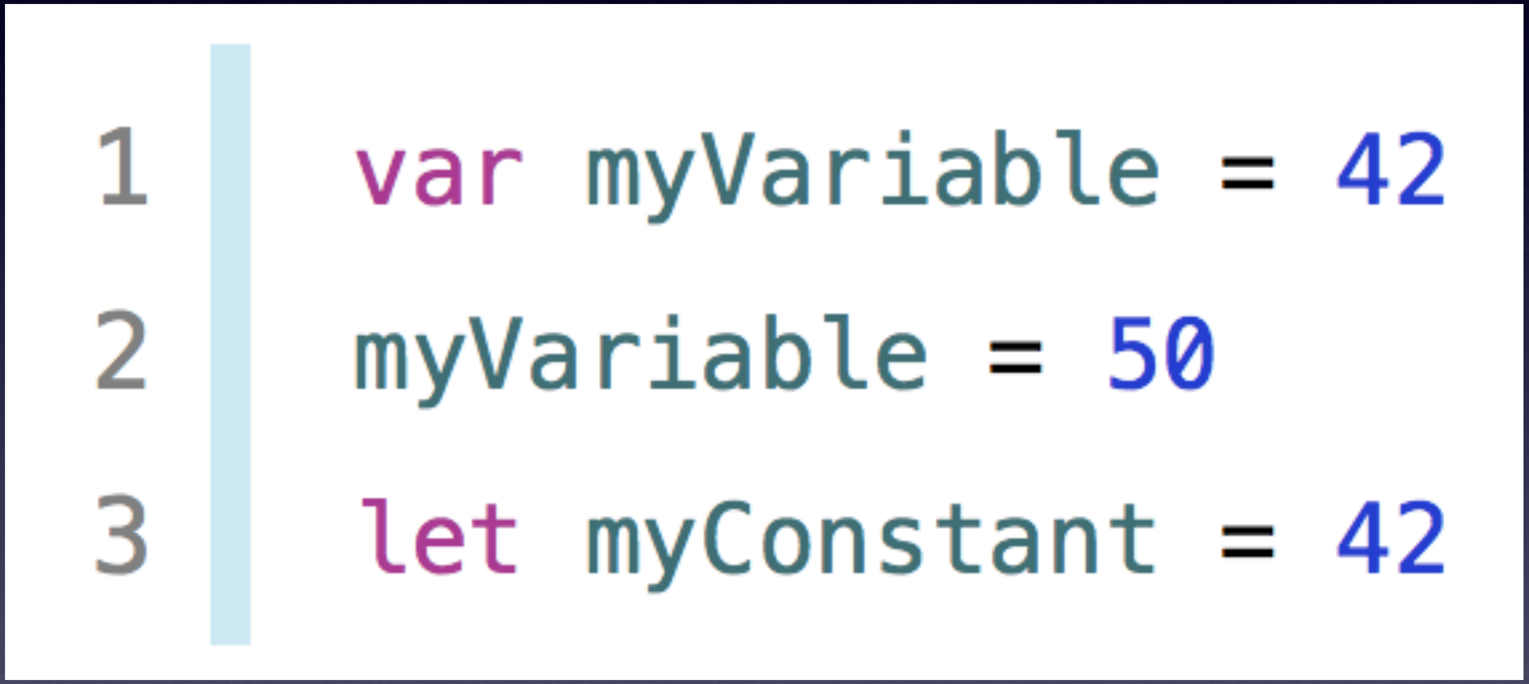
Brian Wang 王昱景

brian.wang.frontline@gmail.com

Basic Types

- A constant is a value that stays the same after it’s declared the first time, while a variable is a value that can change.
- A constant is referred to as immutable, meaning that it can’t be changed, and a variable is mutable.
- Use **let** to make a constant and **var** to make a variable.

- 說變就變的變數（Variable）
- var + 變數名稱 + ＝ + 儲存的資料
- 始終如一的常數（Constant）
- let + 常數名稱 + ＝ + 儲存的資料



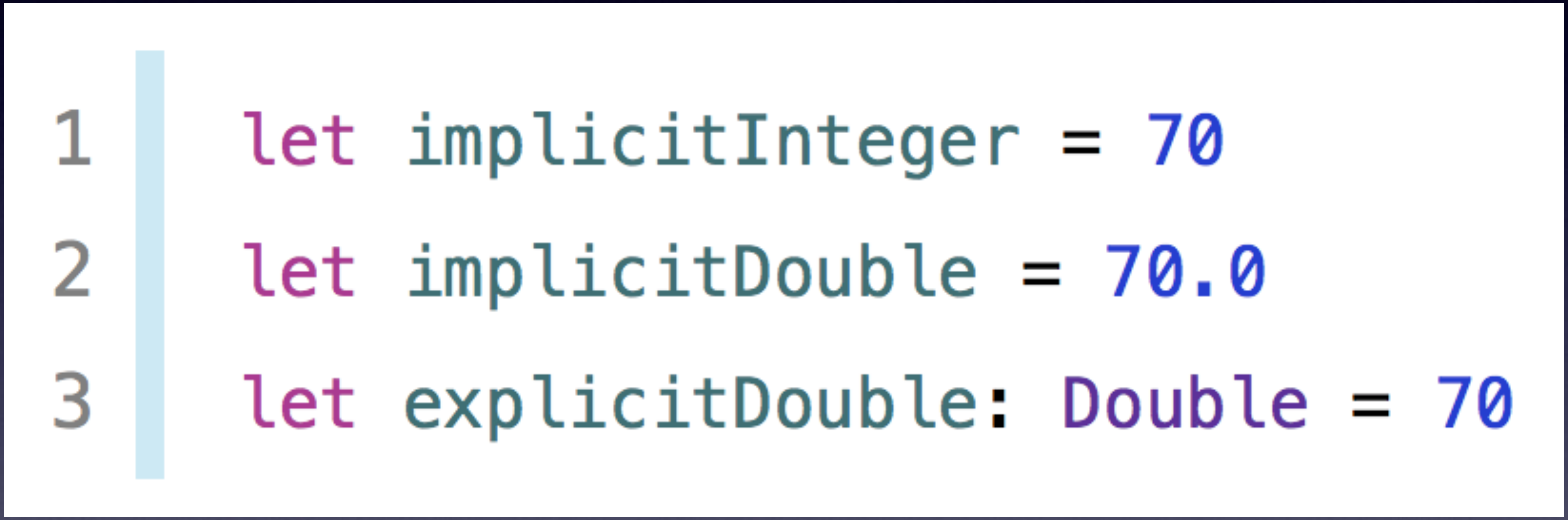
```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

- Every constant and variable in Swift has a type, but you don’t always have to write the type explicitly.
- Providing a value when you create a constant or variable lets the compiler infer its type.
- This is called type inference.

- Once a constant or variable has a type, that type can’t be changed.
- If the initial value doesn’t provide enough information (or if there is no initial value), specify the type by writing it after the variable, separated by a colon.

- 如柯南般的型別自動推理（Type Inference）
- Swift 的變數一旦曾經儲存某個型別的資料，它這輩子就注定只能儲存那個型別
- var + 變數名稱 + ： + 型別 + ＝ + 儲存資料
- 在 Swift 裡型別的字首習慣大寫

- 一定要初始化的變數和常數
- 變數使用前一定要先初始化
- 常數在宣告時就需要指定初始值



```swift
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

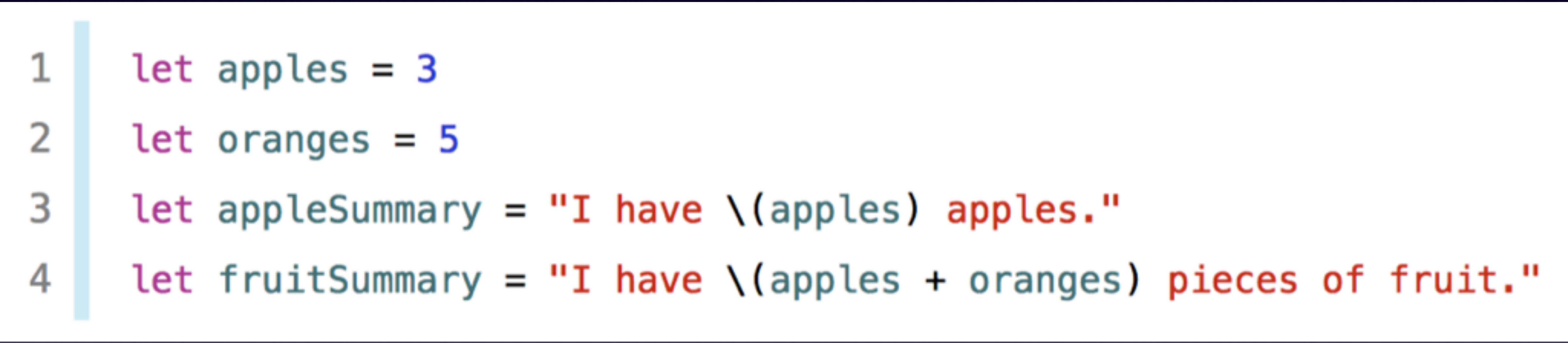
- Int：整數
- Float 和 Double：帶有小數點的浮點數
- Bool：真假值（true 為真，false 為假）
- String：字串

- Values are never implicitly converted to another type.
- If you need to convert a value to a different type, explicitly make an instance of the desired type.

- 字串置換（string interpolation）
- \ + （ + 變數名或常數名 + ）
- 字串相加

```
let label = "The width is "
let width = 94
let widthLabel = label + String(width)
```

- Write the value in parentheses, and write a backslash (\) before the parentheses.
- This is known as string interpolation.

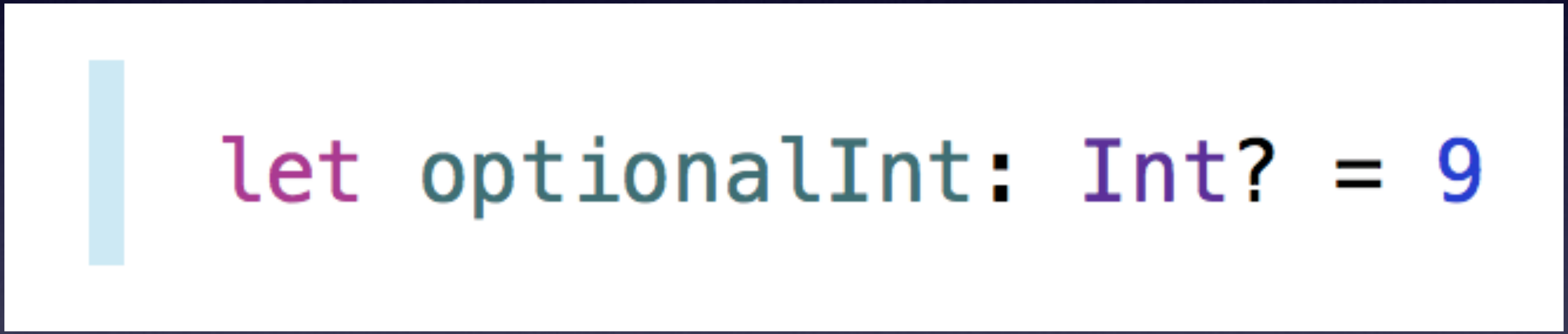


```swift
let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

- Use optionals to work with values that might be missing.
- An optional value either contains a value or contains nil (no value) to indicate that a value is missing.
- Write a question mark (?) after the type of a value to mark the value as optional.

- 可有可無的 optional
- 在 Swift 程式裡，所有我們宣告的變數或常數，不管型別是什麼，都可以幫它添加 optional 的特性，使其成為一個可以無值，也可以有值的特別變數或特別常數

- 一個 optional 的變數或常數，可以被設定為 nil，表示它處於無值的狀態
- 在宣告變數或常數時，於型別的後頭接上問號，即可表示它是一個 optional



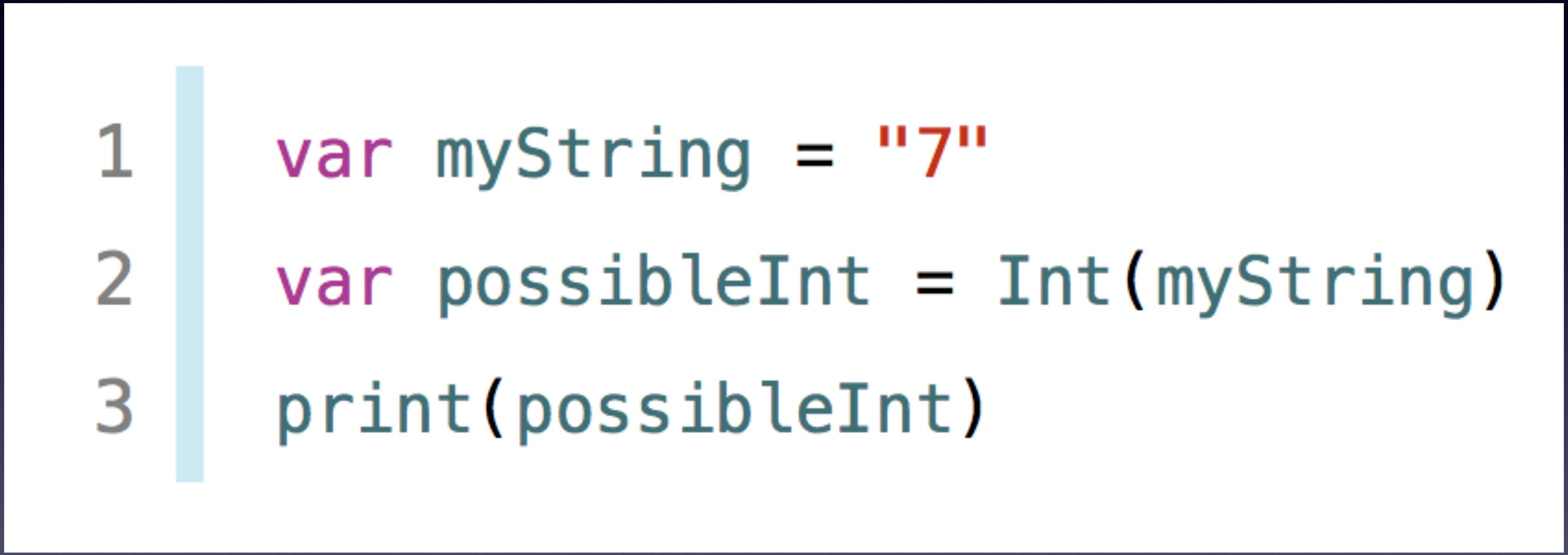
```swift
let optionalInt: Int? = 9
```

- To get the underlying type from an optional, you unwrap it.
- You'll learn unwrapping optionals later, but the most straightforward way to do it involves the force unwrap operator (!).
- Only use the unwrap operator if you're sure the underlying value isn't nil.

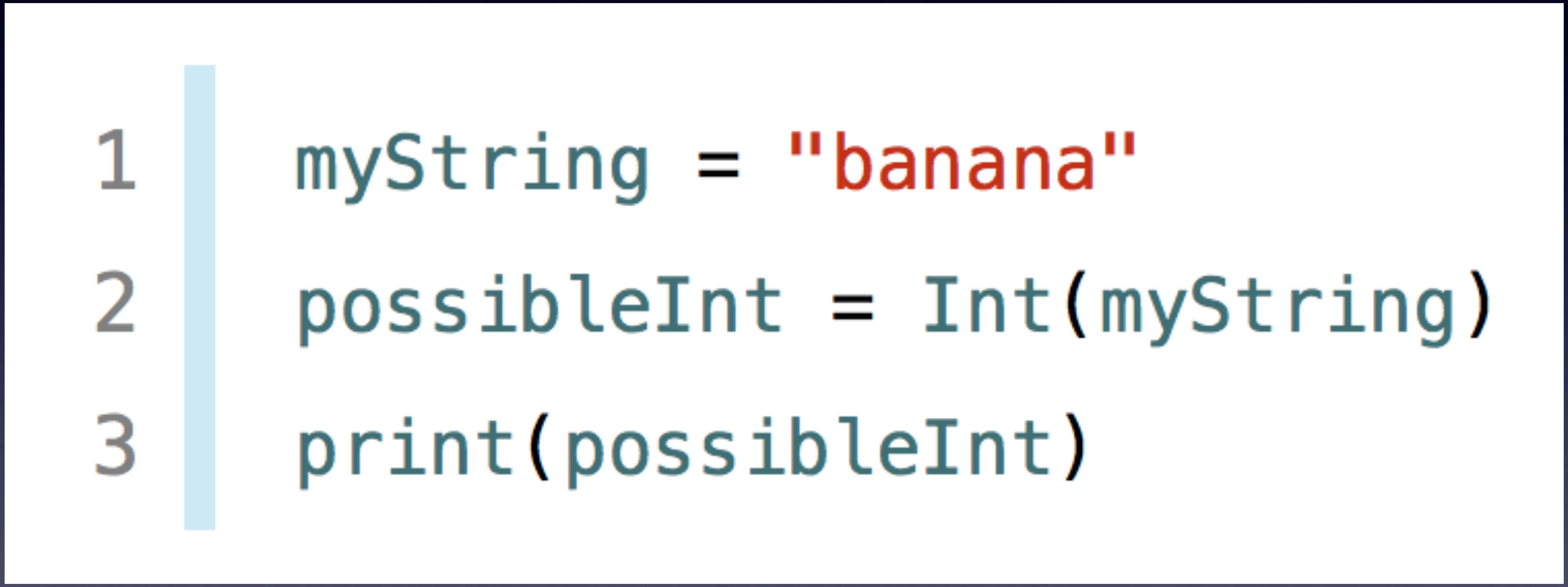
- optional 變數儲存的內容其實是被包裝起來，我們需要解開包裝，才能存取到裡頭儲存的內容
- 利用驚嘆號解開包裝取值的方法稱為 force-unwrap，也就是強迫解開包裝

```swift
let actualInt: Int = optionalInt!
```

- Optionals are pervasive in Swift, and are very useful for many situations where a value may or may not be present.
- They're especially useful for attempted type conversions.



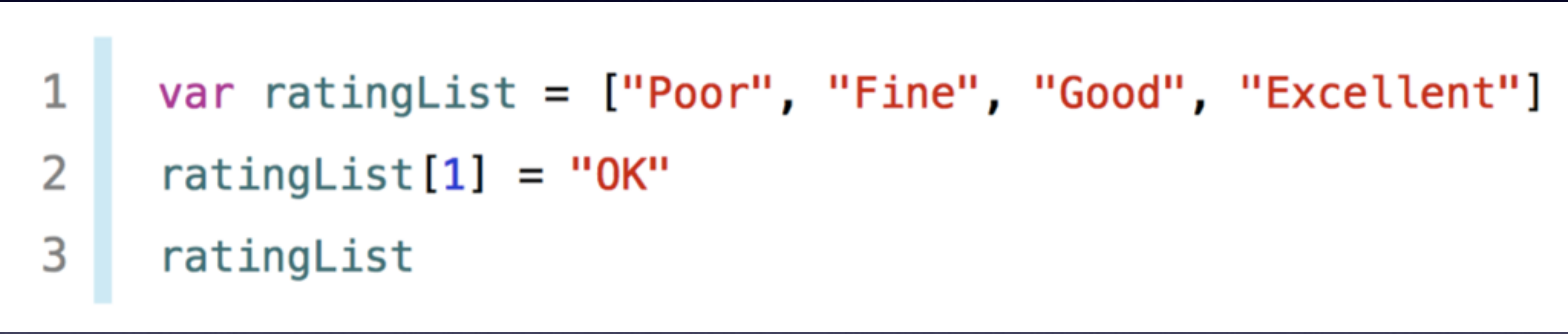
```swift
var myString = "7"
var possibleInt = Int(myString)
print(possibleInt)
```



```
myString = "banana"
possibleInt = Int(myString)
print(possibleInt)
```

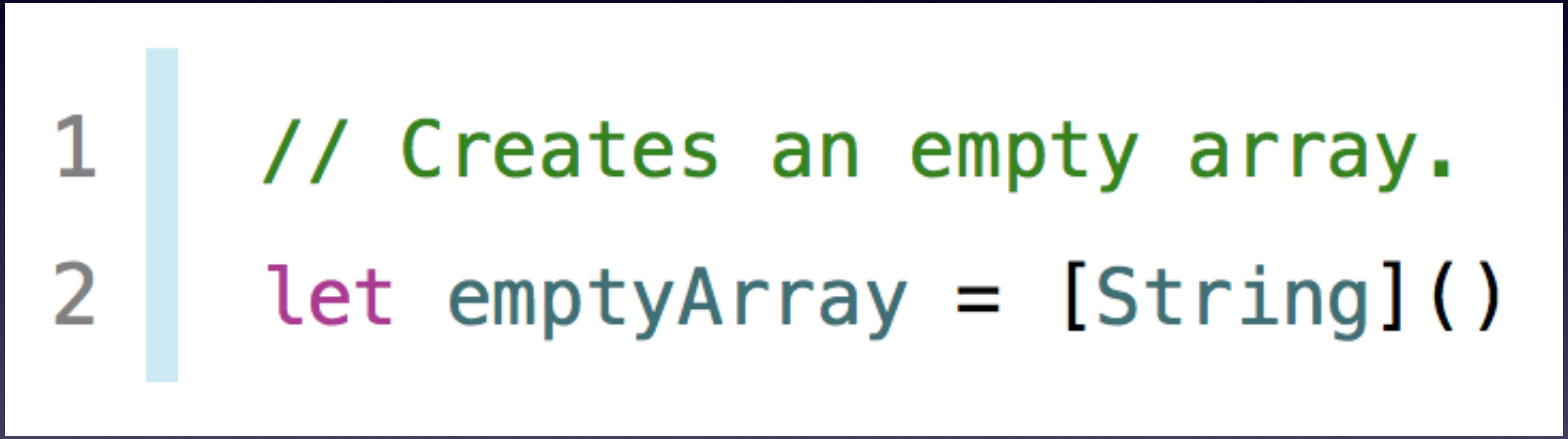
- An array is a data type that keeps track of an ordered collection of items.
- Create arrays using brackets ([]), and access their elements by writing the index in brackets.
- Arrays start at index 0.

- 領取號碼牌乖乖排隊的 array
- 將資料塞入 array 的方法很簡單，以 [ ] 為邊界，於其中我們可以任意地填入多個資料，彼此以逗號分隔
- 傳入編號時需特別小心，不可傳入不存在的編號，否則會造成程式 crash
- 由於 array 裡成員的編號依序遞增，從 0 起跳



```
var ratingList = ["Poor", "Fine", "Good", "Excellent"]
ratingList[1] = "OK"
ratingList
```

- To create an empty array, use the initializer syntax.
- You’ll learn more about initializers in a little while.



```swift
// Creates an empty array.
let emptyArray = [String]()
```

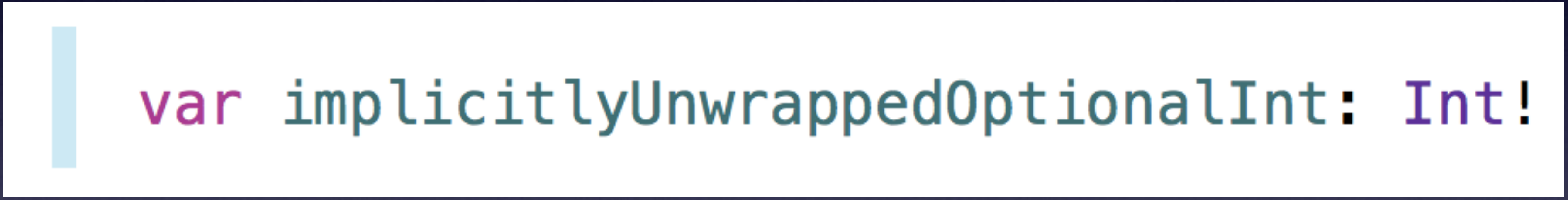
- You’ll notice that the code above has a comment.

- A comment is a piece of text in a source code file that doesn’t get compiled as part of the program but provides context or useful information about individual pieces of code.

- A single-line comment appears after two slashes (//) and a multiline comment appears between a set of slashes and asterisks (/* … */).

- An implicitly unwrapped optional is an optional that can also be used like a nonoptional value, without the need to unwrap the optional value each time it’s accessed.
- This is because an implicitly unwrapped optional is assumed to always have a value after that value is initially set, although the value can change.
- Implicitly unwrapped optional types are indicated with an exclamation mark (!) instead of a question mark (?).

- 自動取值的 Implicitly Unwrapped Optionals
- 只要在宣告時，將原來的問號換成取值的驚嘆號即可



```swift
var implicitlyUnwrappedOptionalInt: Int!
```

Control Flow

- Swift has two types of control **flow statements**. Conditional statements, like **if** and **switch**, check whether a condition is true—that is, if its value evaluates to the Boolean true—before executing a piece of code. **Loops**, like **for-in** and **while**, execute the same piece of code multiple times.

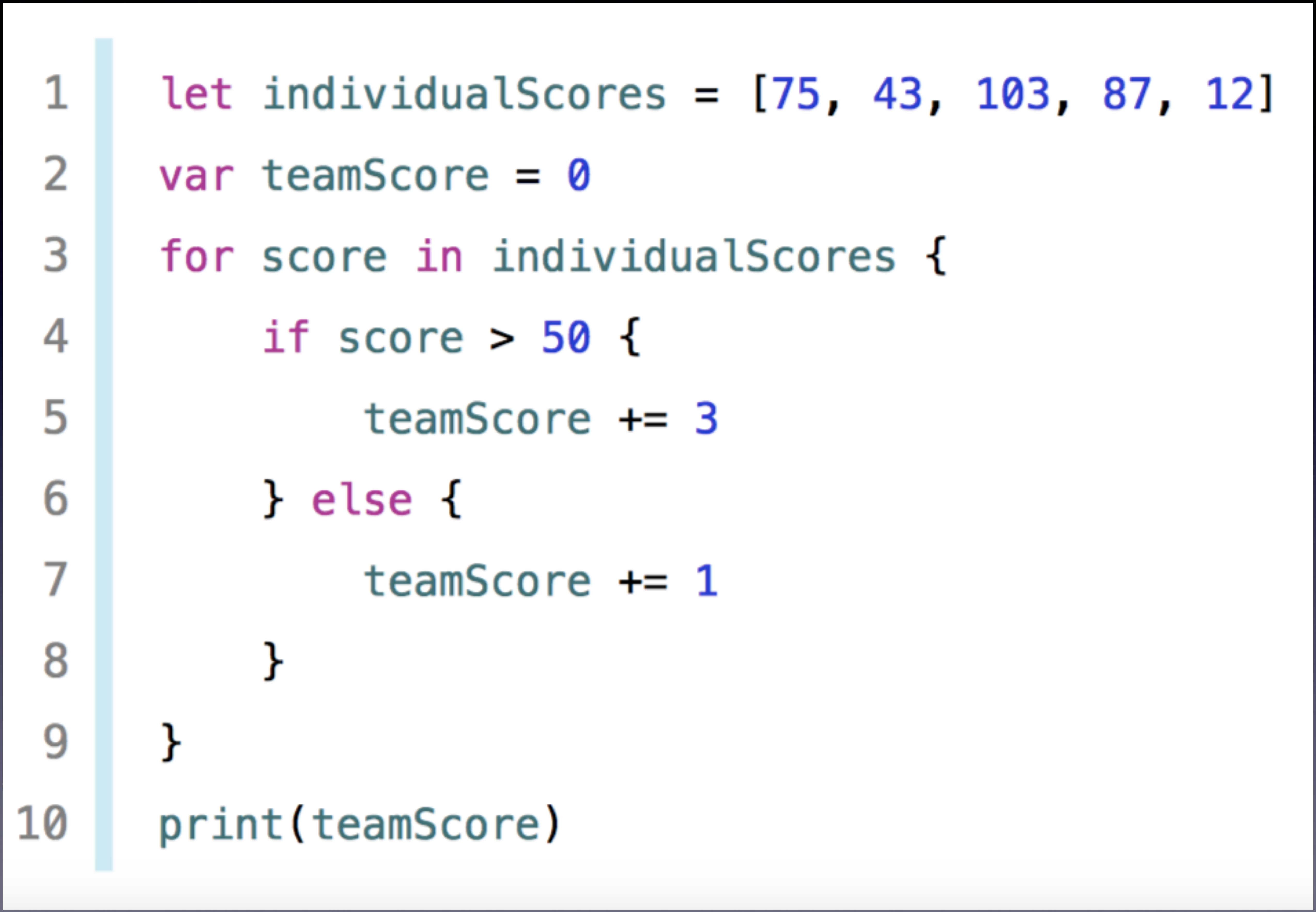
- An if statement checks whether a certain condition is true, and if it is, the if statement evaluates the code inside the statement.
- You can add an **else** clause to an if statement to define more complex behavior.
- An else clause can be used to chain if statements together, or it can stand on its own, in which case the else clause is executed if none of the chained if statements evaluate to true.

- 控制流程的四大天王：if else、switch、while 和 for
- 如果有如果的 if else
- 甩掉無用的（ ）
- 只能以 Bool 型別判斷條件是否成立
- 在 Swift 的 if 裡，判斷條件是否成立一定要以 true / false 為準

- 在處理比較複雜的 if else 判斷時，我們常結合 && 和 ||
- 只有 Bool 型別才能做 && 和 || 運算
- 一定要添加 { }
- == 比較是否相等，例如比較字串是否每個字都一樣
- === 比較是否為同一個東西

```swift
let number = 23
if number < 10 {
    print("The number is small")
} else if number > 100 {
    print("The number is pretty big")
} else {
    print("The number is between 10 and 100")
}
```

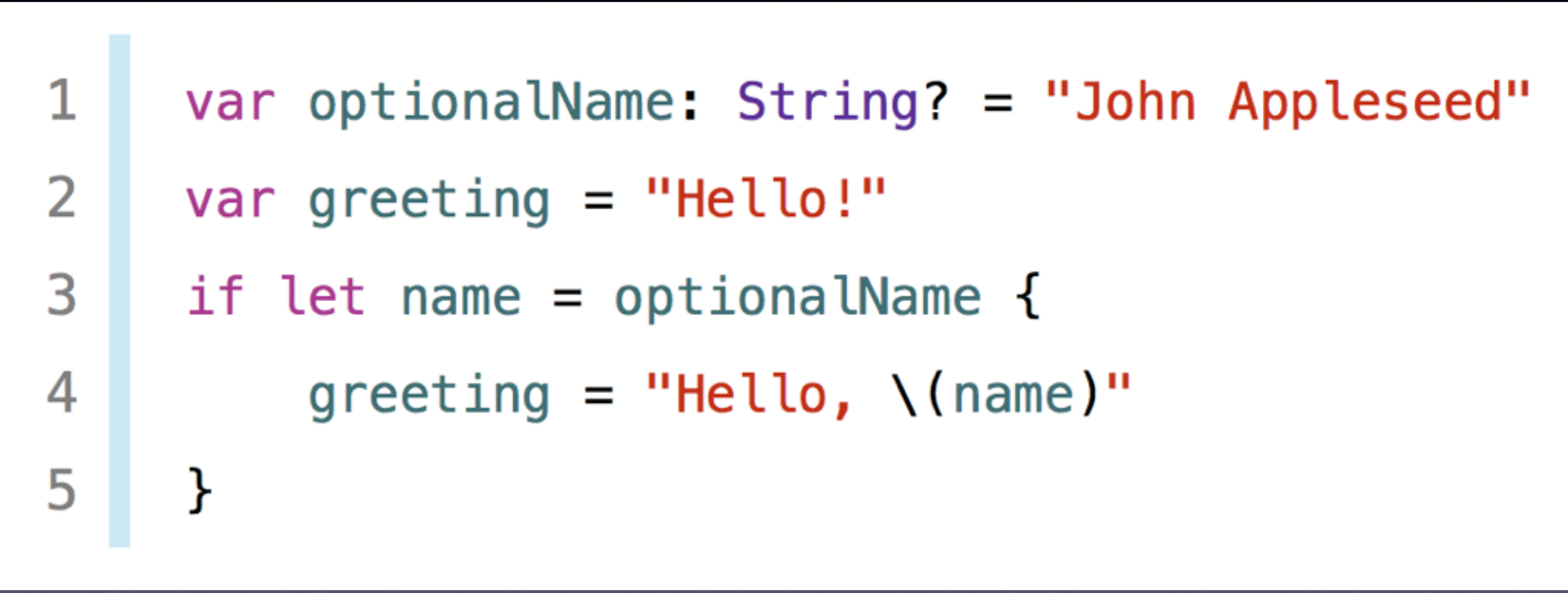
- Statements can be nested to create complex, interesting behavior in a program.



```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
print(teamScore)
```

- Use optional binding in an if statement to check whether an optional contains a value.

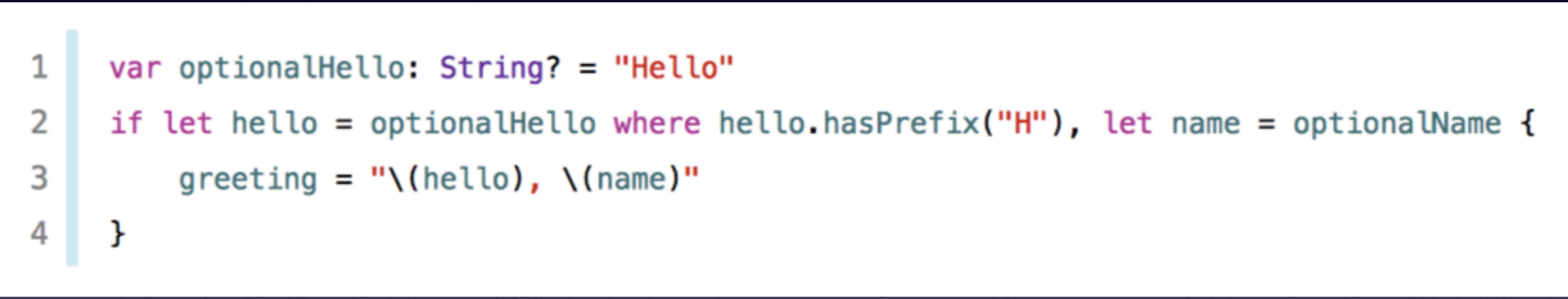
- 判斷並取值的 optional binding
- 判斷 optional 是否有值
- 若是 optional 有值即可放心地利用驚嘆號取值
- optional binding 的語法如下：
    - if ＋ let ＋ 常數 ＋ ＝ ＋ optional 變數或常數
    - if ＋ var ＋ 變數 ＋ ＝ ＋ optional 變數或常數



```swift
var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

- If the optional value is nil, the conditional is false, and the code in braces is skipped.
- Otherwise, the optional value is unwrapped and assigned to the constant after let, which makes the unwrapped value available inside the block of code.

- You can use a single if statement to bind multiple values.
- A where clause can be added to a case to further scope the conditional statement.

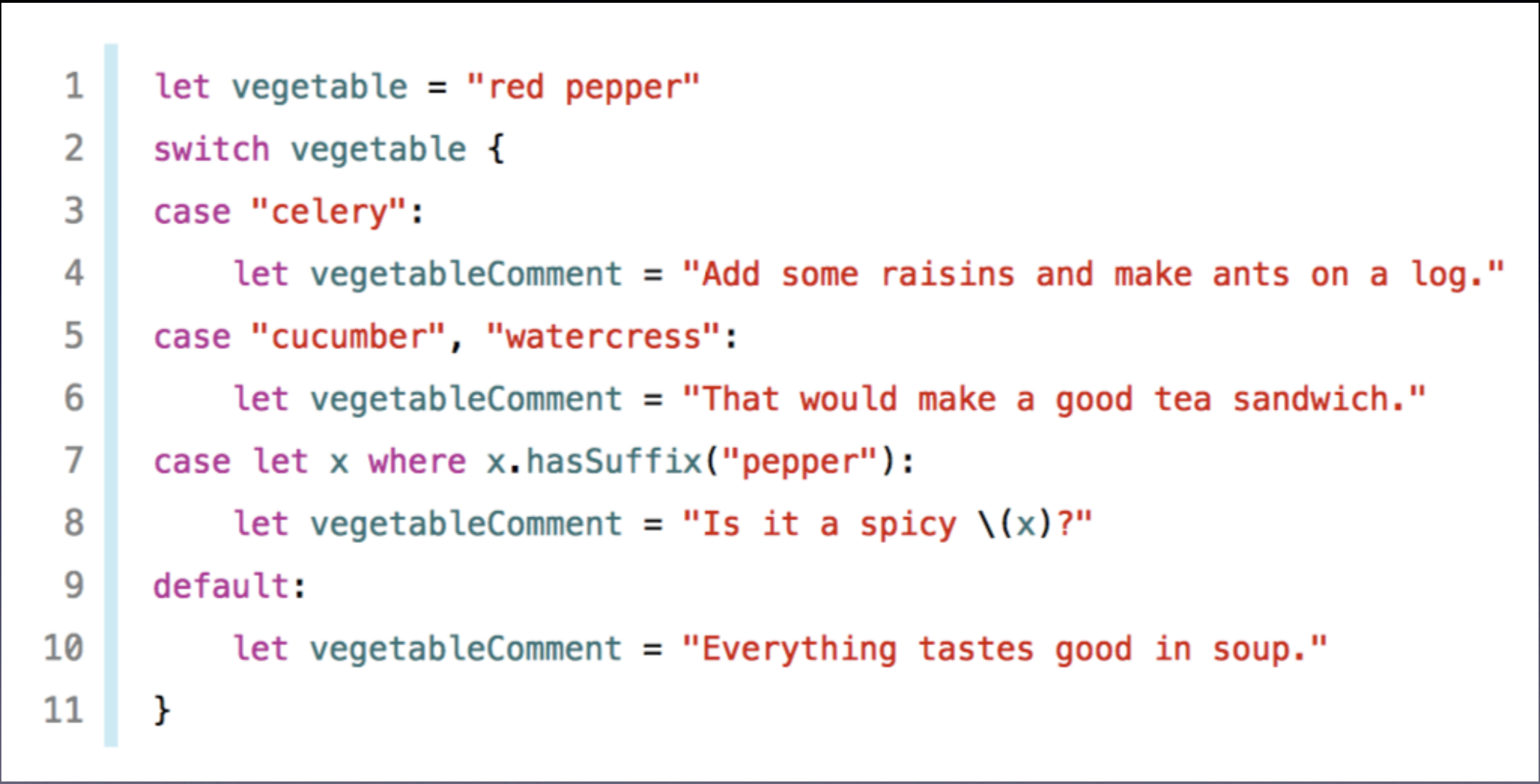


```swift
var optionalHello: String? = "Hello"
if let hello = optionalHello where hello.hasPrefix("H"), let name = optionalName {
    greeting = "\(hello), \(name)"
}
```

- Switches in Swift are quite powerful.
- A switch statement supports any kind of data and a wide variety of comparison operations—it isn’t limited to integers and tests for equality.

- 全面進化的 switch
- （）又被拋棄了
- 一定要包含所有的 case
- 自動自發地跳出 switch，不需要 break 幫忙
- 當 case 下我們沒有要做任何事，沒有任何程式碼時，至少要輸入 break

- 繼續執行下個 case 的 fallthrough
- 一個 case 比對多個條件
- 可以比對任何型別



```swift
let vegetable = "red pepper"
switch vegetable {
case "celery":
    let vegetableComment = "Add some raisins and make ants on a log."
case "cucumber", "watercress":
    let vegetableComment = "That would make a good tea sandwich."
case let x where x.hasSuffix("pepper"):
    let vegetableComment = "Is it a spicy \(x)?"
default:
    let vegetableComment = "Everything tastes good in soup."
}
```

- Notice how let can be used in a pattern to assign the value that matched that part of a pattern to a constant.
- Just like in an if statement, a where clause can be added to a case to further scope the conditional statement.
- However, unlike in an if statement, a switch case that has multiple conditions separated by commas executes when any of the conditions are met.

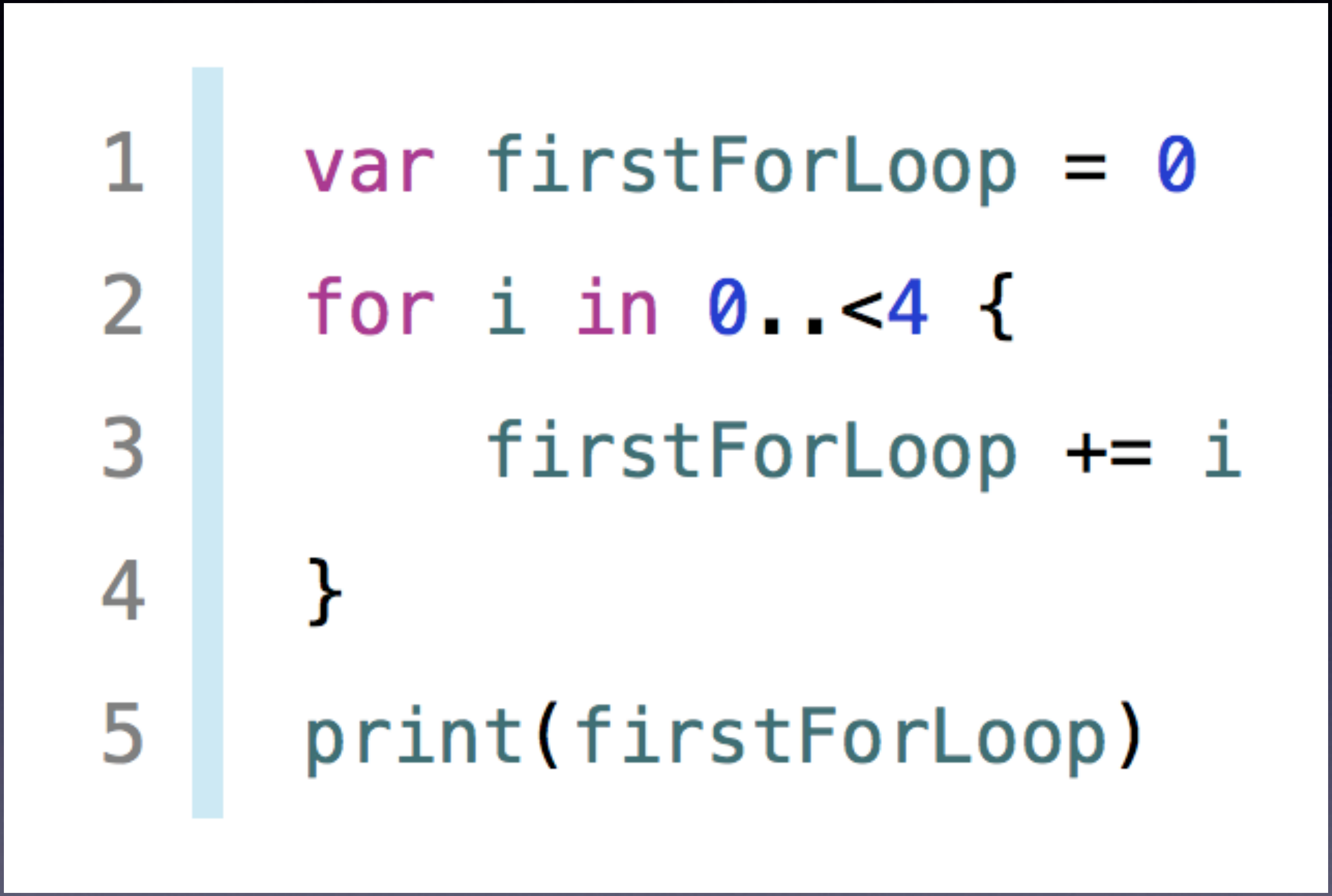
- After executing the code inside the switch case that matched, the program exits from the switch statement.
- Execution doesn’t continue to the next case, so you don’t need to explicitly break out of the switch statement at the end of each case’s code.

- Switch statements must be exhaustive.
- A default case is required, unless it’s clear from the context that every possible case is satisfied, such as when the switch statement is switching on an enumeration.
- This requirement ensures that one of the switch cases always executes.

- You can keep an index in a loop by using a Range.
- Use the half-open range operator (..<) to make a range of indexes.

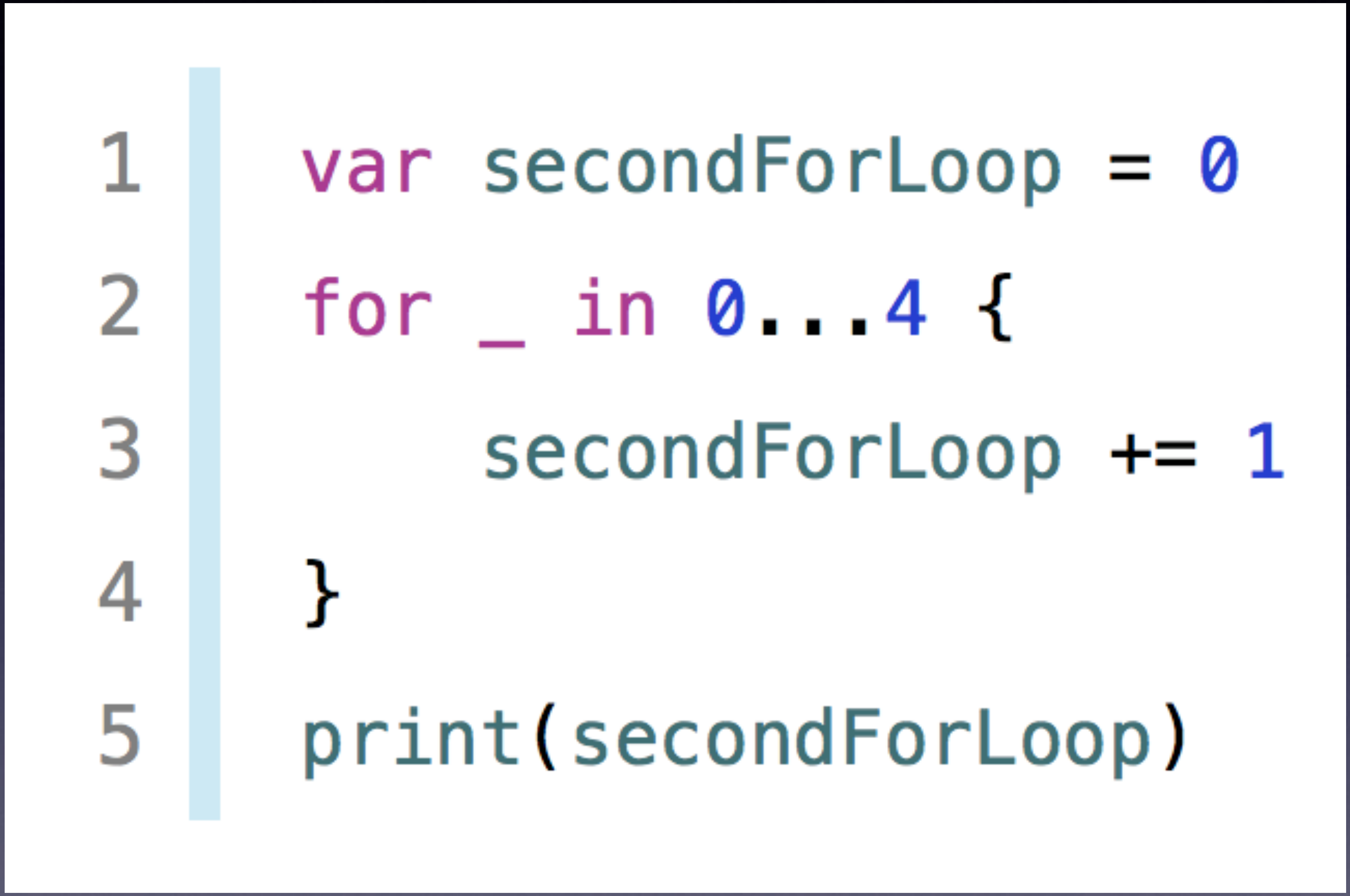
- for 迴圈大顯神通
- 甩掉（ ）的 for
- 留戀（）的 for

- 不只三點式的 range
- Half-Open Range Operator：...<（含頭去尾，只包含開頭不包含尾巴，1...<3：包含 1, 2)
- Closed Range Operator：...（頭尾包含，包含開頭，也包含尾巴，1...3：包含 1, 2, 3)
- for + 常數名 + in + 某種可取出一個個資料的集合



```swift
var firstForLoop = 0
for i in 0..<4 {
    firstForLoop += i
}
print(firstForLoop)
```

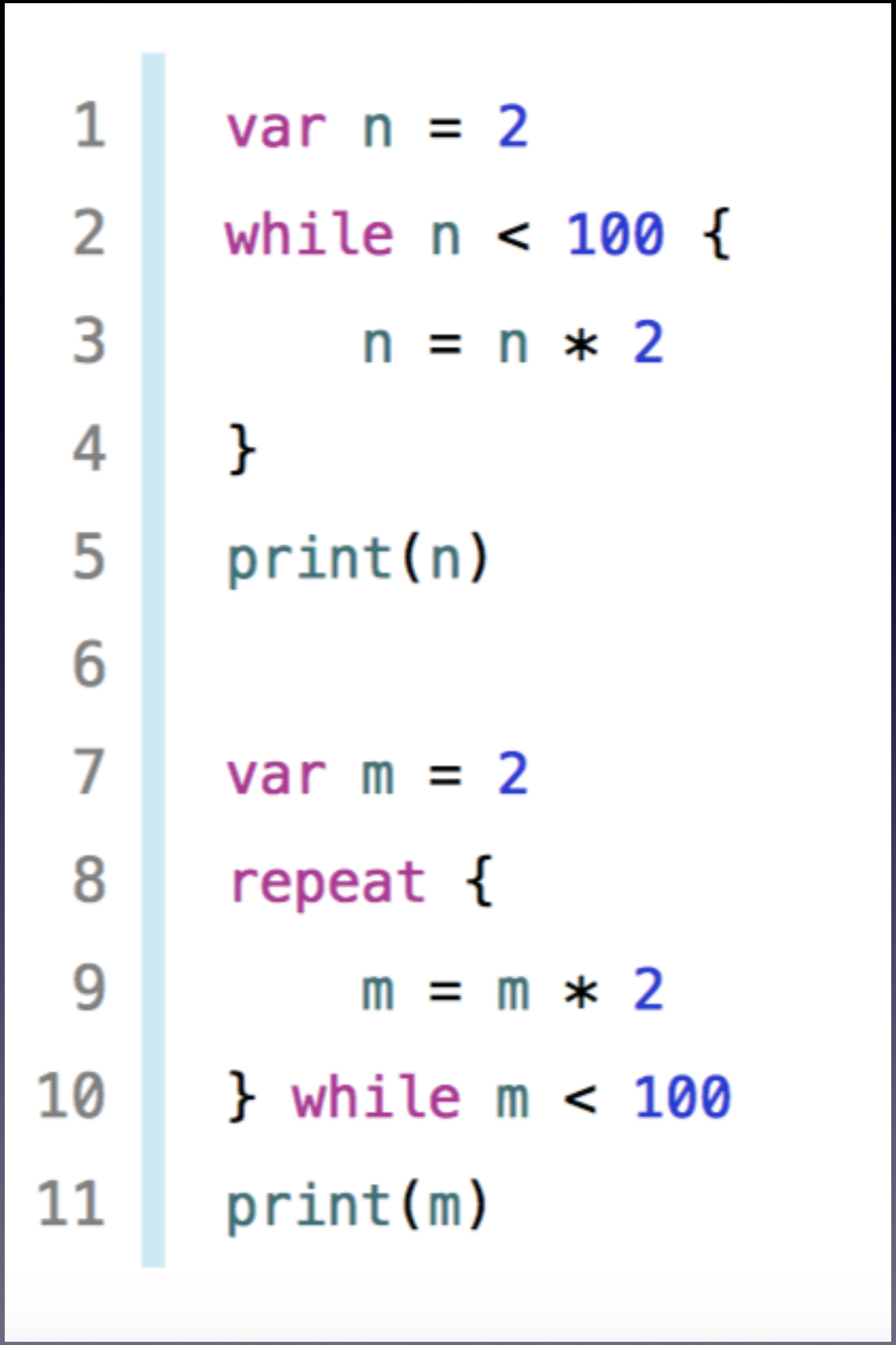
- The half-open range operator (..<) doesn’t include the upper number, so this range goes from 0 to 3 for a total of four loop iterations.
- Use the closed range operator (...) to make a range that includes both values.



```swift
var secondForLoop = 0
for _ in 0...4 {
    secondForLoop += 1
}
print(secondForLoop)
```

- Use **while** to repeat a block of code until a condition changes.
- The condition of a loop can be at the end instead, ensuring that the loop is run at least once.

- 少不了的 while 迴圈



```
var n = 2
while n < 100 {
    n = n * 2
}
print(n)

var m = 2
repeat {
    m = m * 2
} while m < 100
print(m)
```

Functions and Methods

- A **function** is a reusable, named piece of code that can be referred to from many places in a program.
- Use **func** to declare a function.
- A function declaration can include zero or more parameters, written as **name: Type**, which are additional pieces of information that must be passed into the function when it's called.

- Optionally, a function can have a return type, written after the **->**, which indicates what the function returns as its result.
- A function’s implementation goes inside of a pair of curly braces (**{}**).

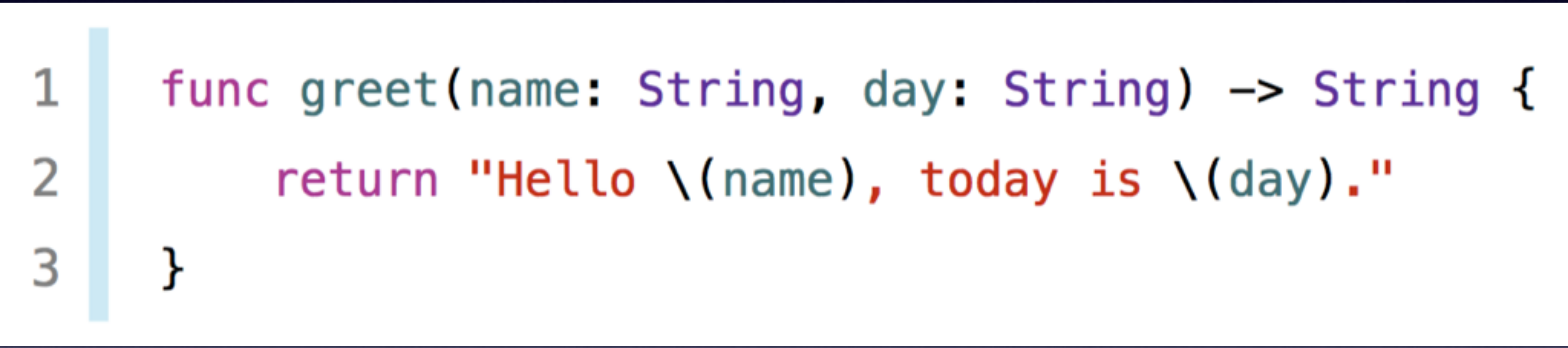
- 程式碼居住的溫暖的家 - function

- 最簡純的無參數 function：
  func ＋ 名稱 ＋ （ ） ＋ { 程式碼區塊 }

- 更多彈性，接受參數的 function：
  func ＋ 名稱 ＋ （參數名稱 ＋ ： ＋參數型別） ＋
  { 程式碼區塊 }

- 不同參數間只要以**逗號**隔開

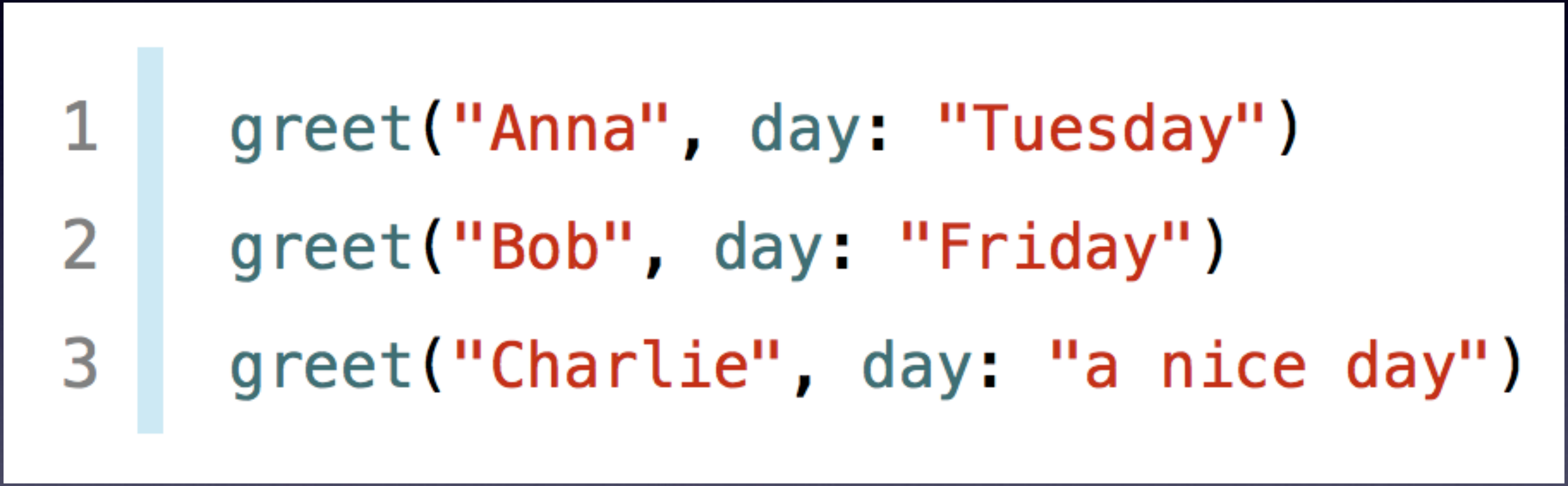
- function 也有回傳值：
  func ＋ 名稱 ＋ （參數1, 參數2... ） ＋-> ＋ 回傳型別＋ { 程式碼區塊 }



```swift
func greet(name: String, day: String) -> String {
    return "Hello \(name), today is \(day)."
}
```

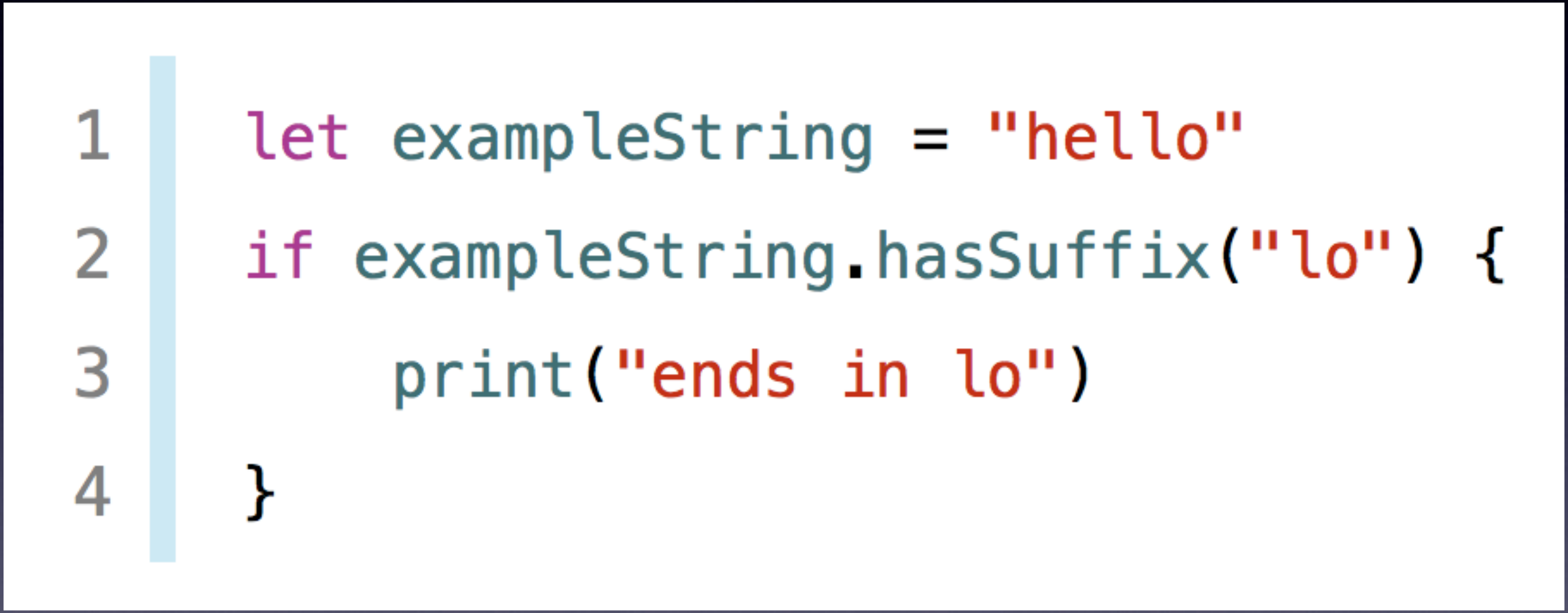
- Call a function by following its name with a list of arguments (the values you pass in to satisfy a function’s parameters) in parentheses.
- When you call a function, you pass in the first argument value without writing its name, and every subsequent value with its name.

- 呼叫 function：function 名稱 ＋ （ ）



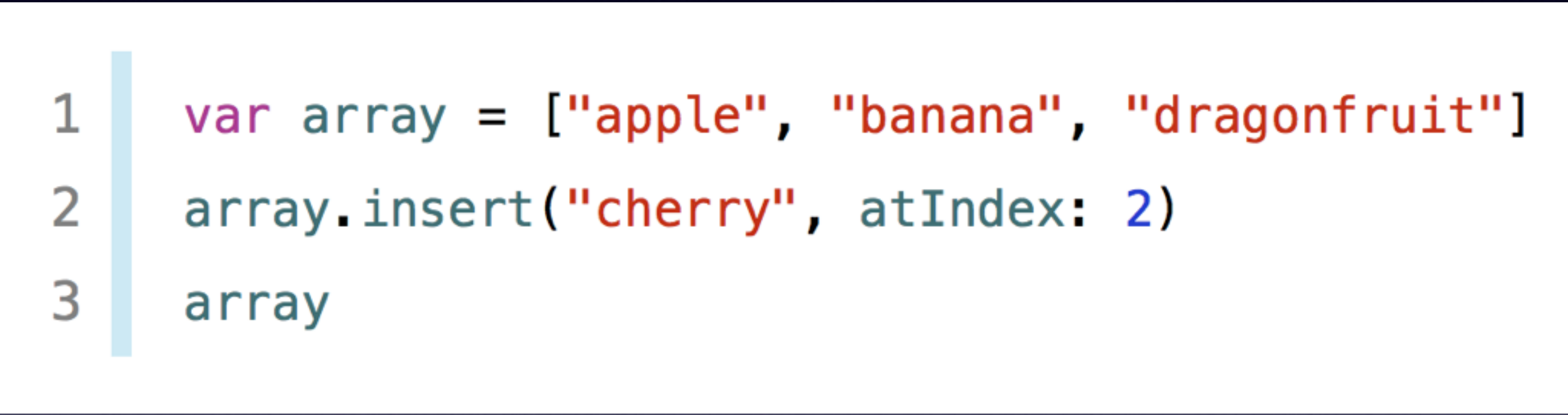
```
greet("Anna", day: "Tuesday")
greet("Bob", day: "Friday")
greet("Charlie", day: "a nice day")
```

- Functions that are defined within a specific type are called methods.
- Methods are explicitly tied to the type they’re defined in, and can only be called on that type.



```swift
let exampleString = "hello"
if exampleString.hasSuffix("lo") {
    print("ends in lo")
}
```

- As you see, you call a method using the dot syntax.
- When you call a method, you pass in the first argument value without writing its name, and every subsequent value with its name.



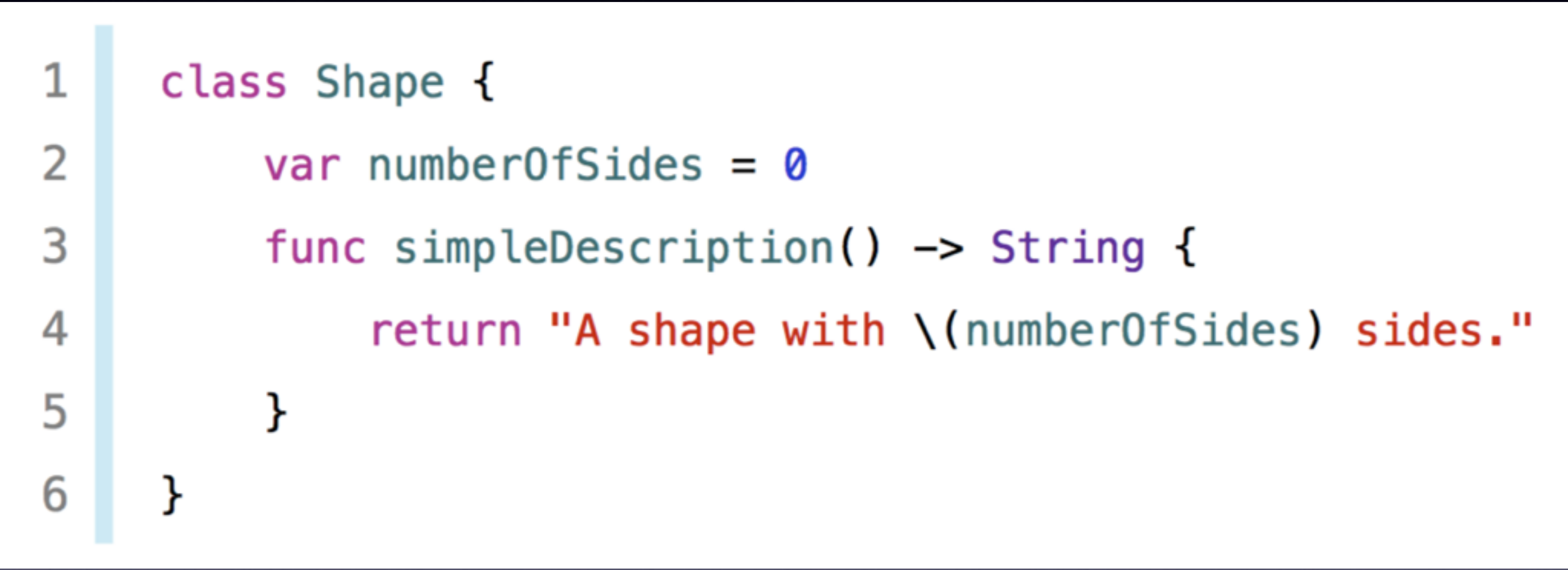
```swift
var array = ["apple", "banana", "dragonfruit"]
array.insert("cherry", atIndex: 2)
array
```

Classes and Initializers

- In object-oriented programming, the behavior of a program is based largely on interactions between objects.
- An object is an instance of a class, which can be thought of as a blueprint for that object.
- Classes store additional information about themselves in the form of properties, and define their behavior using methods.

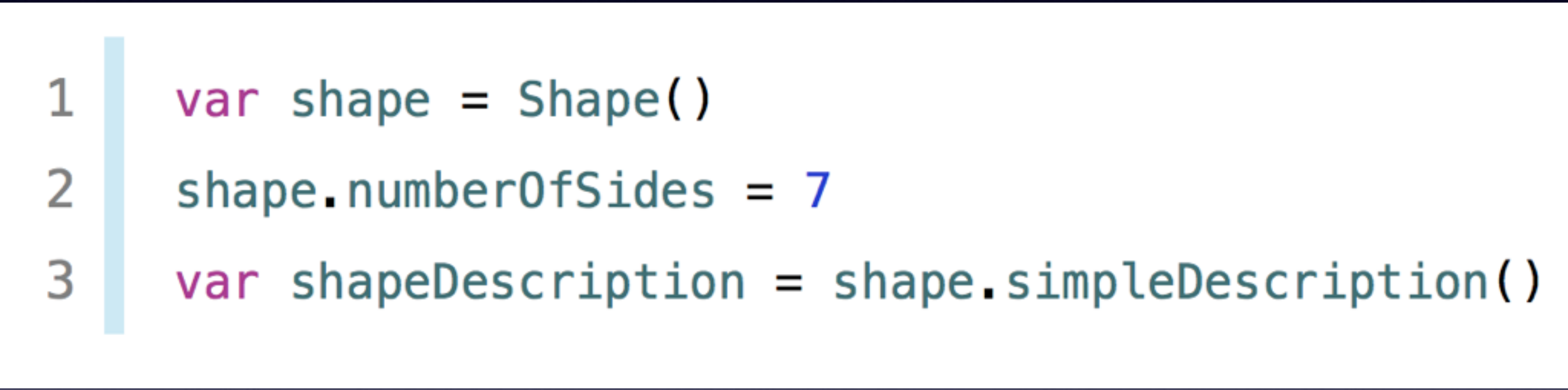
- Use **class** followed by the class's name to define a class.
- A property declaration in a class is written the same way as a constant or variable declaration, except that it's in the context of a class.
- Likewise, method and function declarations are written the same way.

- 構成 App 世界的主角—物件
- 建造物件的藍圖—類別
- class ＋ 類別名稱 ＋ { ＋ }
- 只需要一個檔案定義
- 不用加入（import）一堆 header 檔



```swift
class Shape {
    var numberOfSides = 0
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

- Create an instance of a class—an object—by putting parentheses after the class name.
- Use dot syntax to access the properties and methods of the instance.

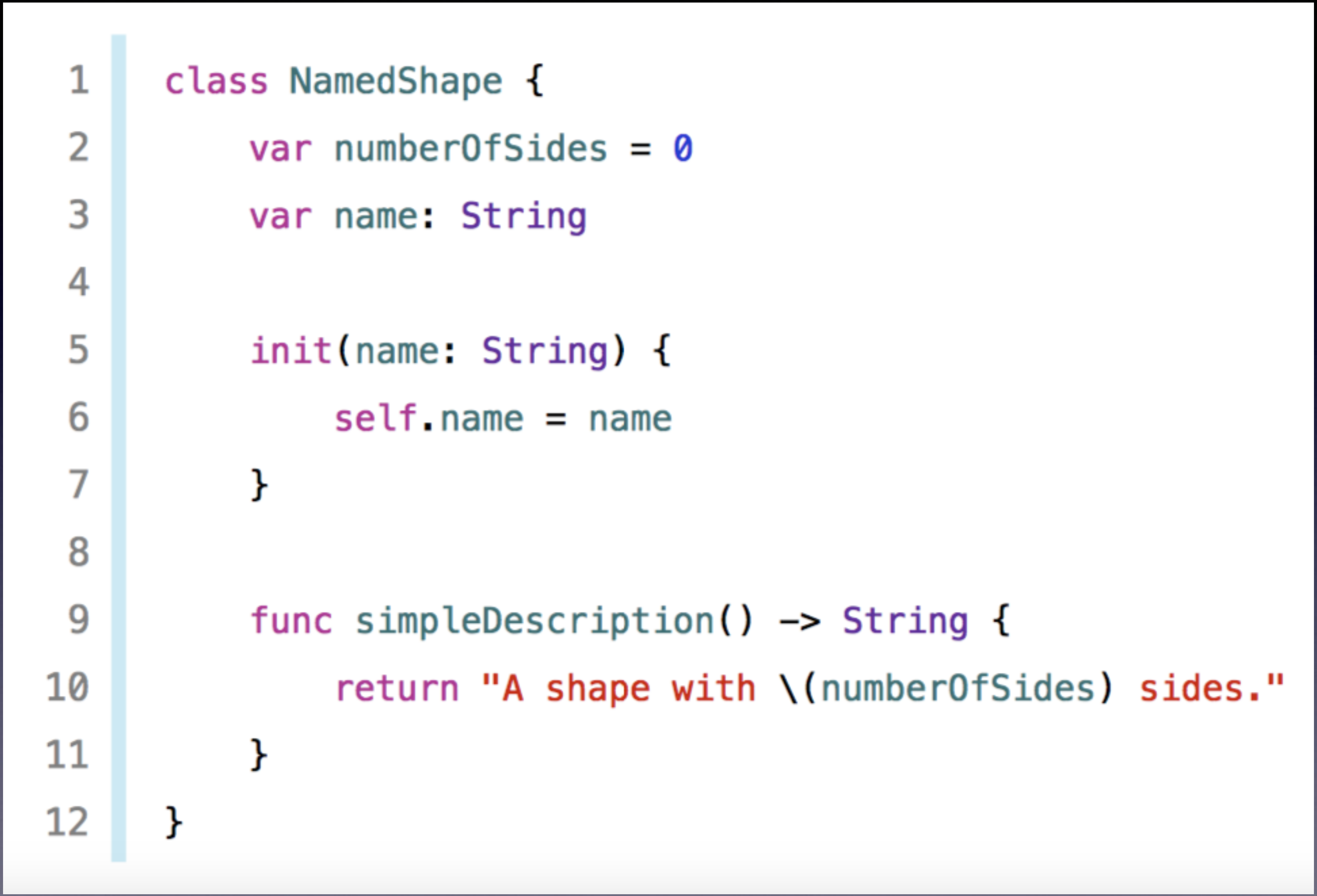


```swift
var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

- An **initializer** is a method that prepares an instance of a class for use, which involves setting an initial value for each property and performing any other setup.

- Use **init** to create one.

- 一定要初始化的物件屬性
- 初始化物件的方法— initializer
- 宣告時不須加 func
- 名稱一定是 init
- 將在物件建立時被呼叫
- 自動生成的無參數 initializer — default initializer
- 接受參數的彈性 initializer



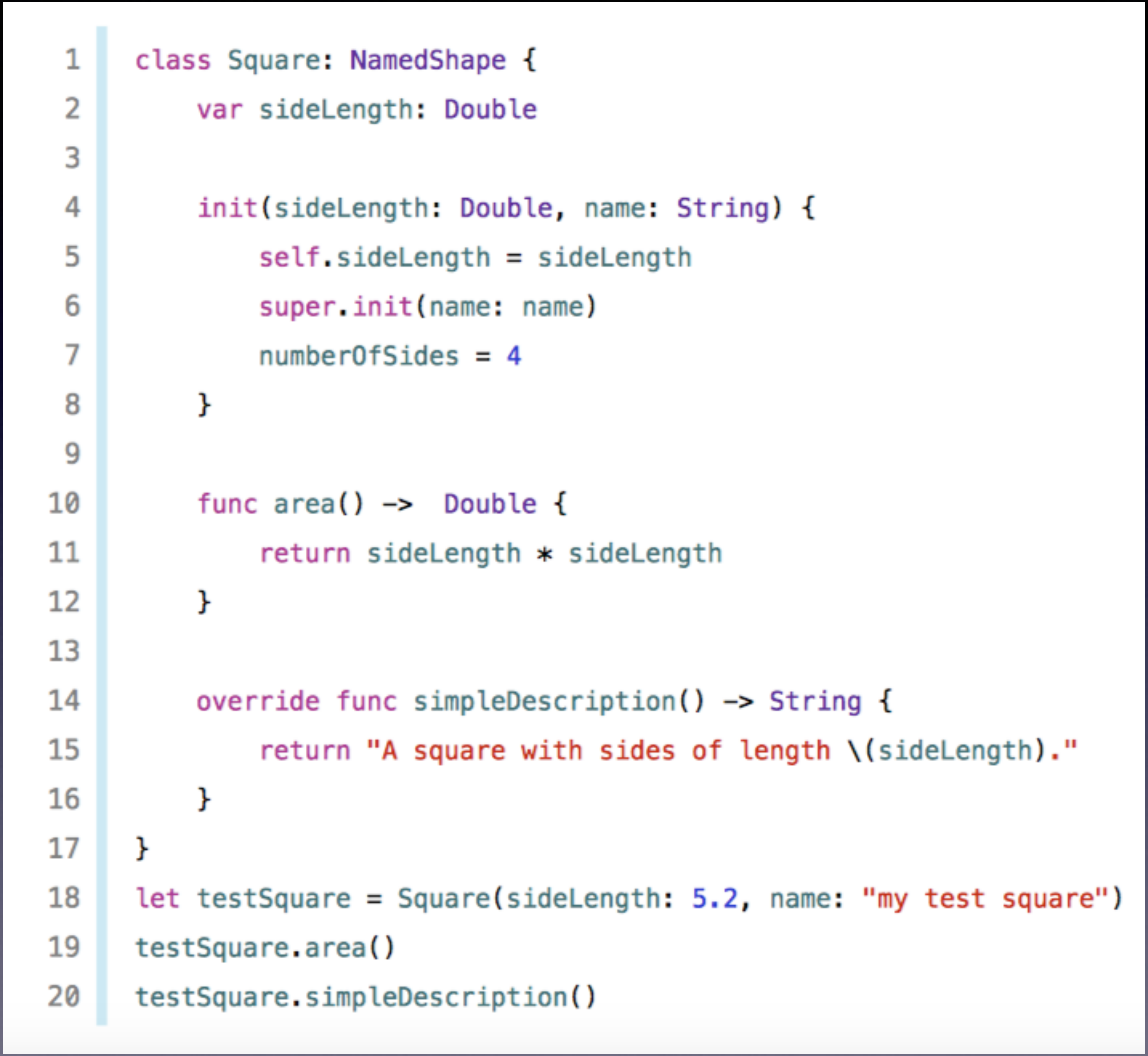
```swift
class NamedShape {
    var numberOfSides = 0
    var name: String

    init(name: String) {
        self.name = name
    }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

- Every property needs a value assigned—either in its declaration or in the initializer.
- You don’t call an initializer by writing **init**; you call it by putting parentheses with the appropriate arguments after the class name.
- When you call an initializer, you include all arguments names along with their values.

```
let namedShape = NamedShape(name: "my named shape")
```

- Classes **inherit** their behavior from their parent class.
- A class that inherits behavior from another class is called a subclass of that class, and the parent class is called a superclass.
- Subclasses include their superclass name after their class name, separated by a colon.
- A class can inherit from only one superclass, although that class can inherit from another superclass, and so on, resulting in a class hierarchy.

- 青出於藍的繼承
- 子類別 ： 父類別 {
    屬性宣告
    方法定義
  }
- 家庭革命！方法和屬性的覆寫（override)



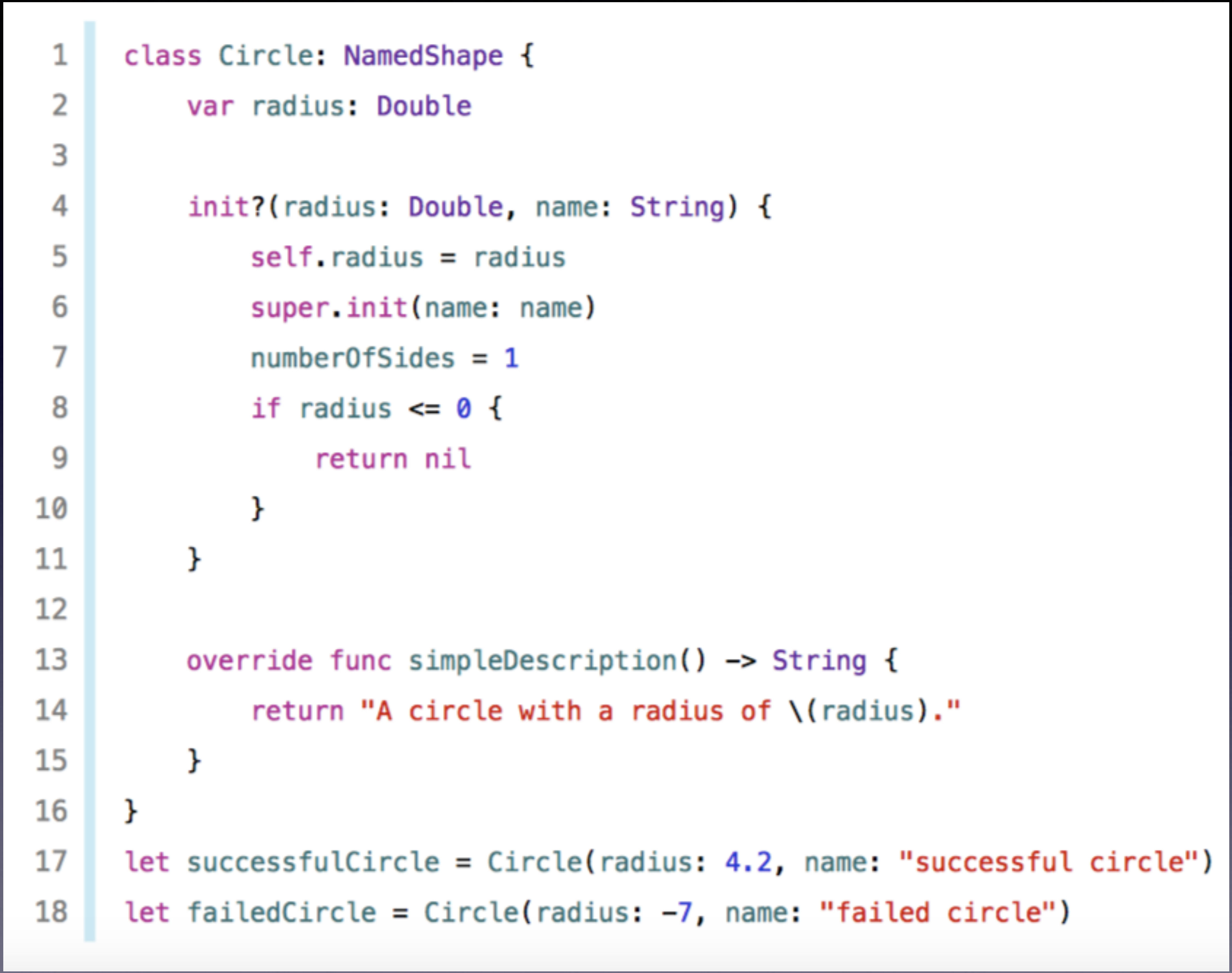
```swift
class Square: NamedShape {
    var sideLength: Double

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }

    func area() ->  Double {
        return sideLength * sideLength
    }

    override func simpleDescription() -> String {
        return "A square with sides of length \(sideLength)."
    }
}
let testSquare = Square(sideLength: 5.2, name: "my test square")
testSquare.area()
testSquare.simpleDescription()
```

- Sometimes, initialization of an object needs to fail, such as when the values supplied as the arguments are outside of a certain range, or when data that’s expected to be there is missing.

- Initializers that may fail to successfully initialize an object are called failable initializers.

- A failable initializer can return **nil** after initialization. Use **init?** to declare a failable initializer.

- 失敗的 failable initializer
- 可以回傳 nil，表示失敗
- 當 init 裡可能回傳 nil 時，init 的後頭則須加上？或！
- 在 Swift 裡唯有 optional 才能是 nil
- 加了？或！的 initializer 又稱為 failable initializer



```swift
class Circle: NamedShape {
    var radius: Double

    init?(radius: Double, name: String) {
        self.radius = radius
        super.init(name: name)
        numberOfSides = 1
        if radius <= 0 {
            return nil
        }
    }

    override func simpleDescription() -> String {
        return "A circle with a radius of \(radius)."
    }
}
let successfulCircle = Circle(radius: 4.2, name: "successful circle")
let failedCircle = Circle(radius: -7, name: "failed circle")
```

- Initializers can also have a number of keywords associated with them.
- A designated initializer does not require any keywords.
- This initializer acts as one of the primary initializers for a class; any initializer within a class must ultimately call through to a designated initializer.

- The **convenience** keyword next to an initializer indicates a convenience initializer.
- Convenience initializers are secondary initializers.
- They can add additional behavior or customization, but must eventually call through to a designated initializer.

- A **required** keyword next to an initializer indicates that every subclass of the class that has that initializer must implement its own version of the initializer (if it implements any initializer).

- 防止絕技失傳的 required initializer

- Type casting is a way to check the type of an instance, and to treat that instance as if it’s a different superclass or subclass from somewhere else in its own class hierarchy.

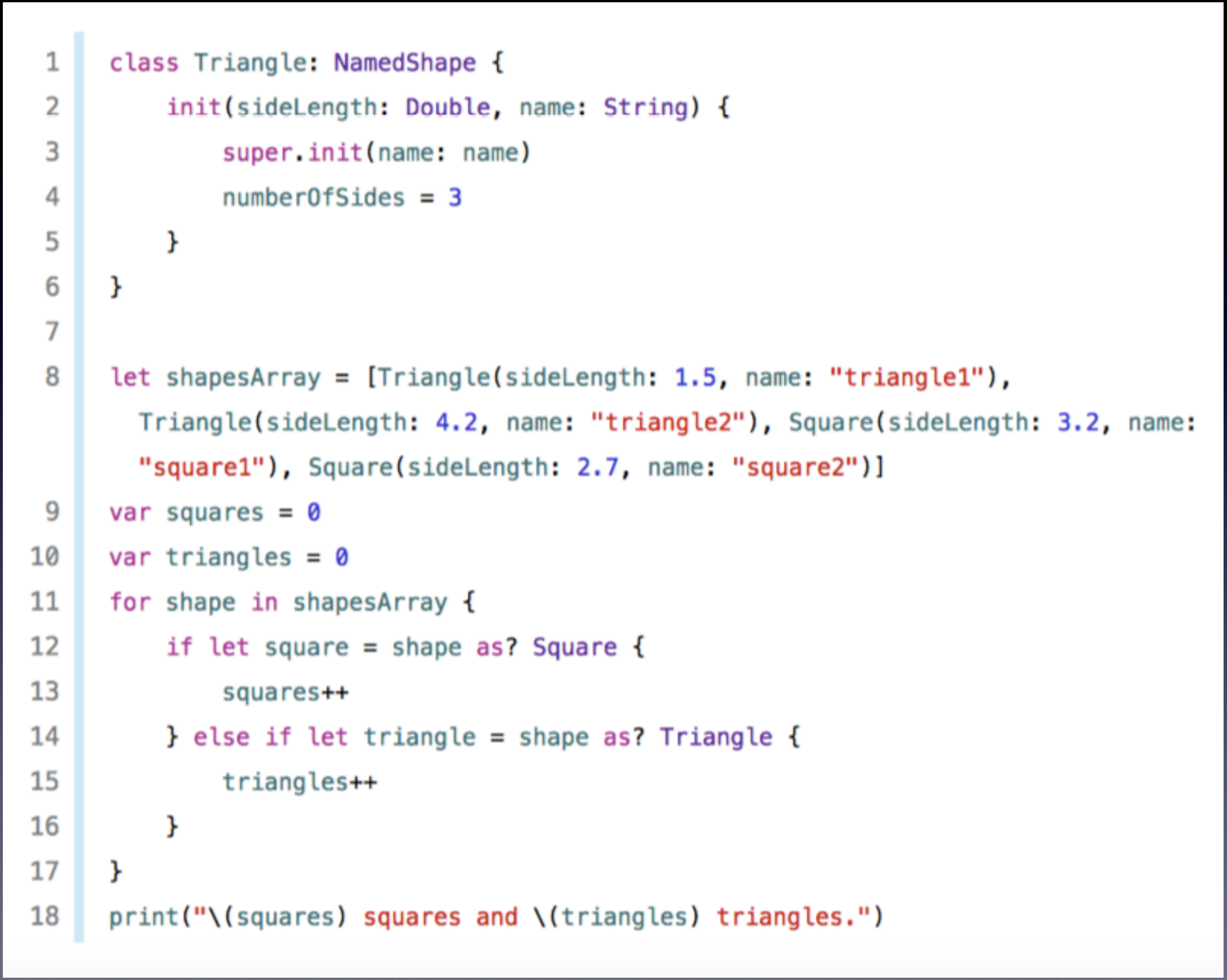
- A constant or variable of a certain class type may actually refer to an instance of a subclass behind the scenes.
- Where you believe this is the case, you can try to **downcast** to the subclass type using a type cast operator.

- Because downcasting can fail, the type cast operator comes in two different forms.
- The optional form, **as?**, returns an optional value of the type you are trying to downcast to.
- The forced form, **as!**, attempts the downcast and force-unwraps the result as a single compound action.

- Use the optional type cast operator (**as?**) when you’re not sure if the downcast will succeed.
- This form of the operator will always return an optional value, and the value will be **nil** if the downcast was not possible.
- This lets you check for a successful downcast.

- Use the forced type cast operator (**as!**) only when you’re sure that the downcast will always succeed.
- This form of the operator will trigger a runtime error if you try to downcast to an incorrect class type.

- 失敗並不可恥—只要以 as！轉型
- 如果不確定是否可以轉型成功，最好還是採用 as？轉型
- 可能轉型失敗，需使用 as！的例子為向下轉型
- 若是向上轉型則可放心地使用 as，子類別一定可以安全轉型為父類別，因為子類別繼承了父類別的屬性和方法



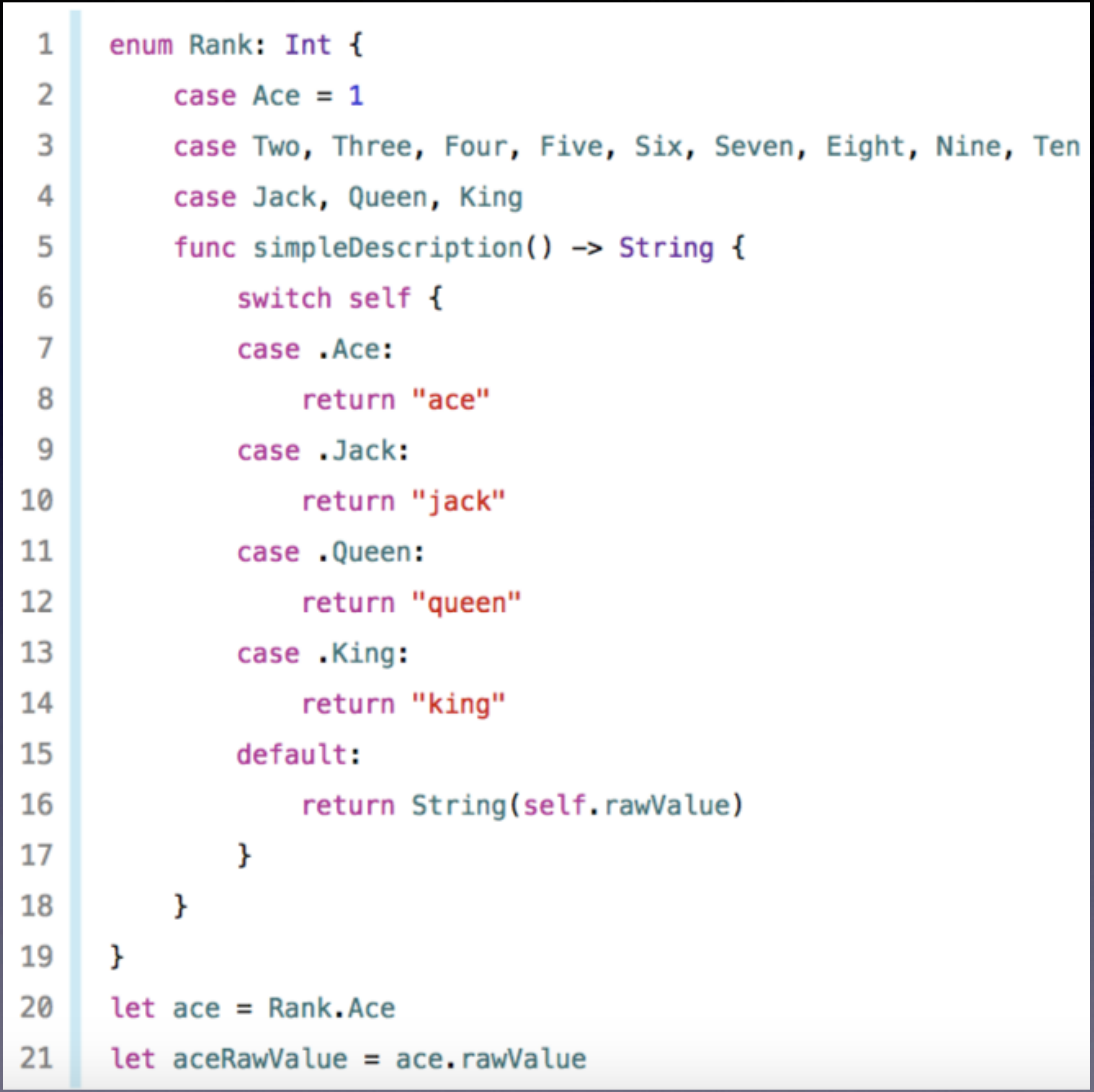
```swift
class Triangle: NamedShape {
    init(sideLength: Double, name: String) {
        super.init(name: name)
        numberOfSides = 3
    }
}

let shapesArray = [Triangle(sideLength: 1.5, name: "triangle1"), Triangle(sideLength: 4.2, name: "triangle2"), Square(sideLength: 3.2, name: "square1"), Square(sideLength: 2.7, name: "square2")]
var squares = 0
var triangles = 0
for shape in shapesArray {
    if let square = shape as? Square {
        squares++
    } else if let triangle = shape as? Triangle {
        triangles++
    }
}
print("\(squares) squares and \(triangles) triangles.")
```

Enumerations and Structures

- Classes aren’t the only ways to define data types in Swift.
- Enumerations and structures have similar capabilities to classes, but can be useful in different contexts.

- **Enumerations** define a common type for a group of related values and enable you to work with those values in a type-safe way within your code.
- Enumerations can have methods associated with them.

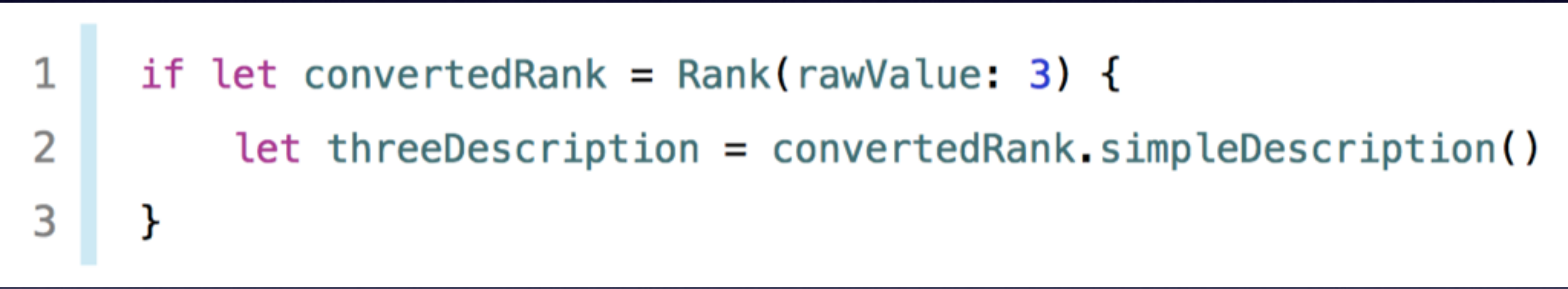
- 透過 enum 能以容易理解記憶的名稱取代無意義的數字，增加程式的可讀性
- 明明白白表達清單成員的 enum
- 為每一個成員設置專屬的 case，有幾個成員就準備幾個 case
- enum 的最佳拍檔 switch
- 以 enum 型別宣告的變數，其儲存的內容一定是 enum型別裡的成員



```swift
enum Rank: Int {
    case Ace = 1
    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
    case Jack, Queen, King
    func simpleDescription() -> String {
        switch self {
        case .Ace:
            return "ace"
        case .Jack:
            return "jack"
        case .Queen:
            return "queen"
        case .King:
            return "king"
        default:
            return String(self.rawValue)
        }
    }
}
let ace = Rank.Ace
let aceRawValue = ace.rawValue
```

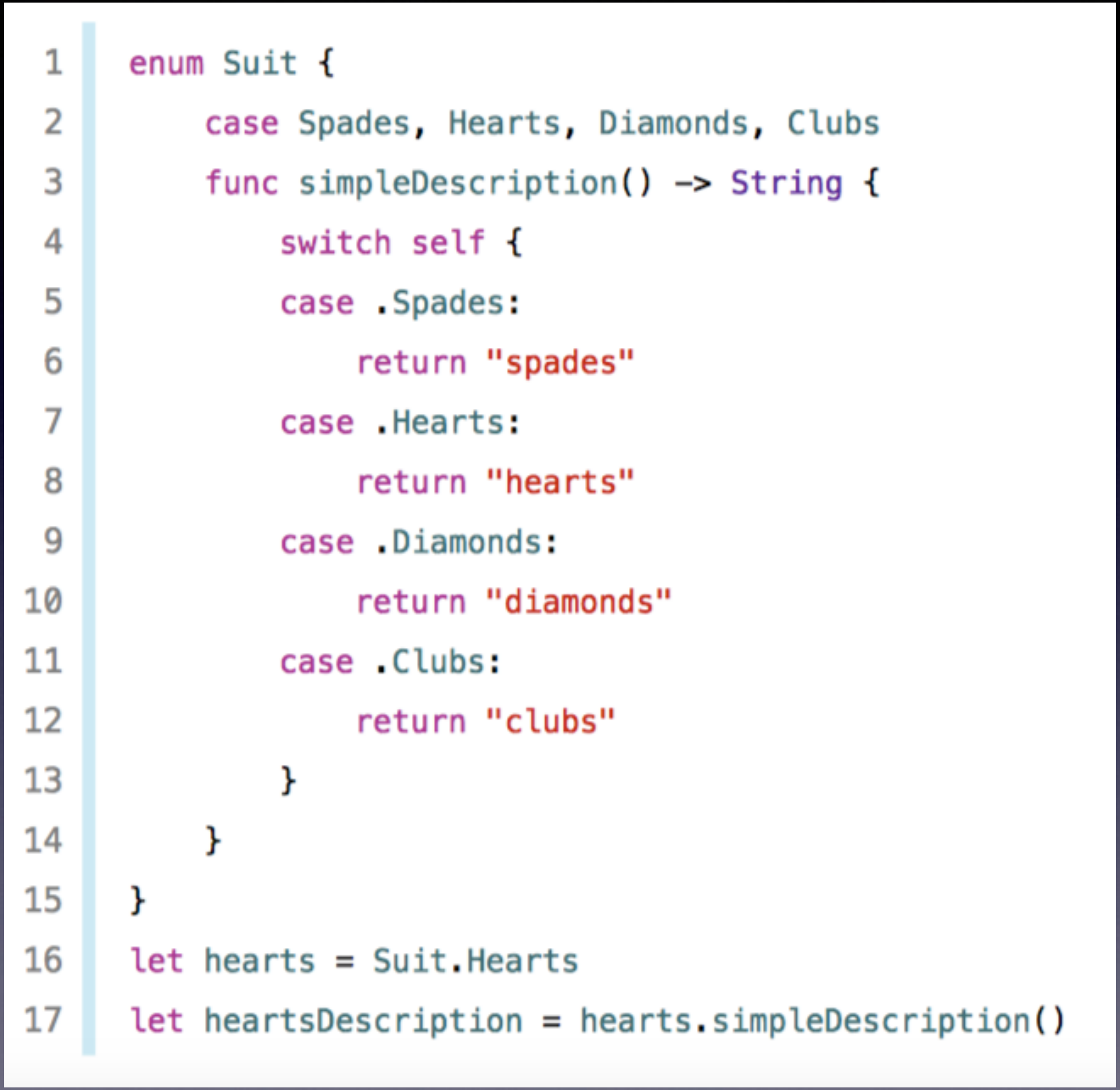
- Use the **rawValue** property to access the raw value of an enumeration member.
- Use the **init?(rawValue:)** initializer to make an instance of an enumeration from a raw value.

- enum 成員的另一面 — raw value
- 所有成員的 raw value 必須是同一個型別
- 定義 enum 時必須表明 raw value 的型別
- raw value 允許的型別只能為以下幾種：String、Character、Int、Float 和 Double



```swift
if let convertedRank = Rank(rawValue: 3) {
    let threeDescription = convertedRank.simpleDescription()
}
```

- 動態設定的相關聯資料 — associated value
- 一次能和多個 value 相關聯，享有如同一夫多妻的福利，不像 raw value 只能一對一，遵守傳統的一夫一妻制
- 相關聯的 value 可以是任何型別，甚至是物件



```swift
enum Suit {
    case Spades, Hearts, Diamonds, Clubs
    func simpleDescription() -> String {
        switch self {
        case .Spades:
            return "spades"
        case .Hearts:
            return "hearts"
        case .Diamonds:
            return "diamonds"
        case .Clubs:
            return "clubs"
        }
    }
}
let hearts = Suit.Hearts
let heartsDescription = hearts.simpleDescription()
```

- **Structures** support many of the same behaviors as classes, including methods and initializers.

- One of the most important differences between structures and classes is that structures are always copied when they are passed around in your code, but classes are passed by reference.

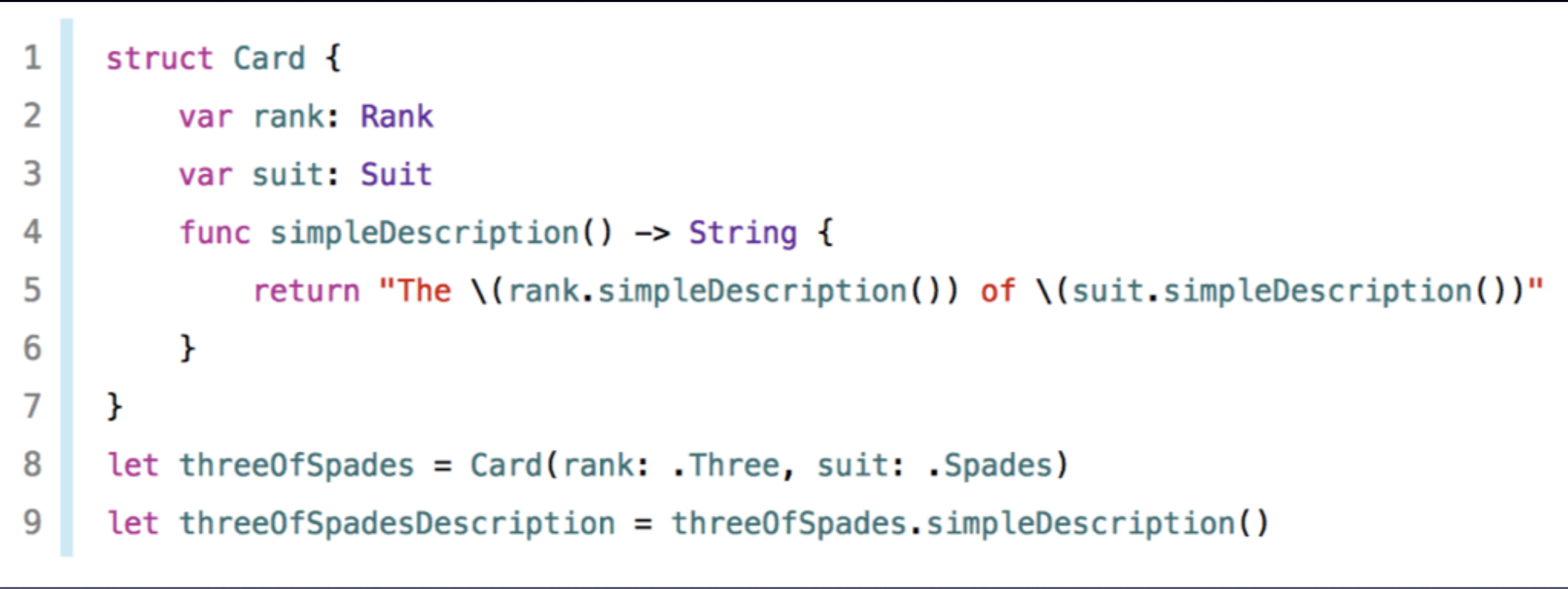
- Structures are great for defining lightweight data types that don’t need to have capabilities like inheritance and type casting.

- 模仿天王 — 和 class 百分之九十雷同的 struct
- 以 struct 定義
- 以 static 定義型別方法和屬性

- struct 無法繼承
- struct 能從 stored property 自動生成 memberwise initializer
- struct 是 value type，class 是 reference type

- 和 struct 百分之八十雷同的 enum
- Swift 定義型別有三種方法：class、struct 和 enum
- 方法：包含從 enum 建立的資料呼叫方法和直接從 enum 型別呼叫的方法
- 在 enum 裡定義型別方法和 struct 一樣，由 static 開頭

- 屬性：enum 可宣告屬性，但只能是 computed property，不能是 stored property
- 在 enum 裡定義的型別屬性則和 struct 一樣，由 static 開頭
- initializer



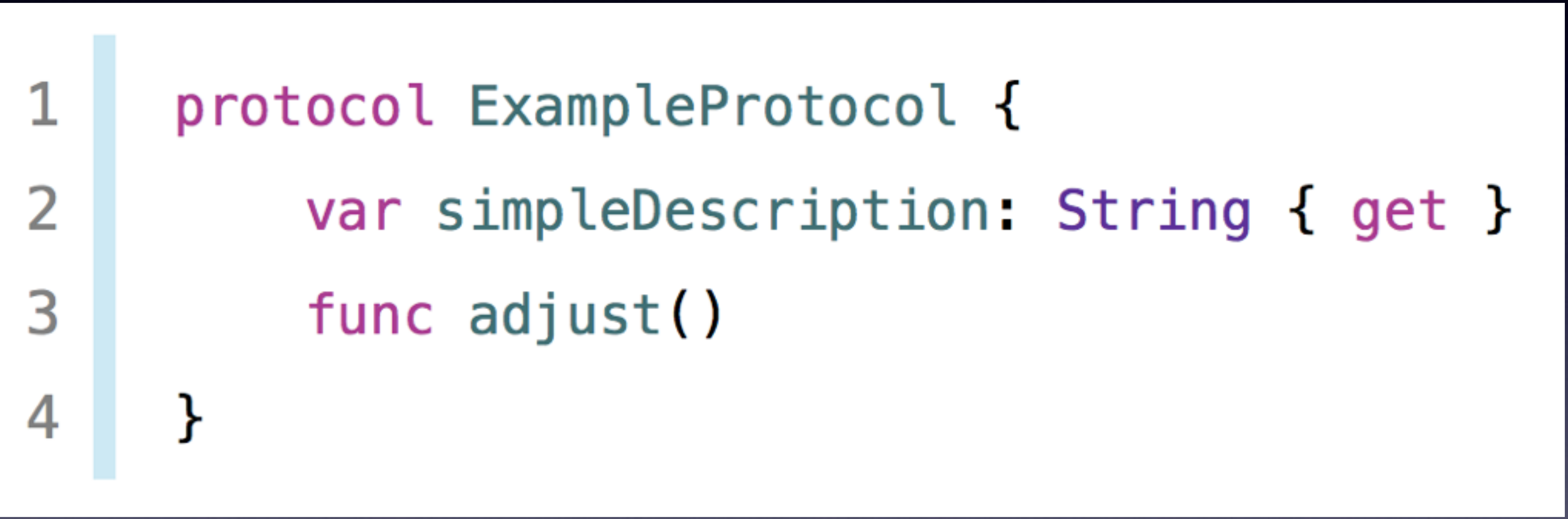
```swift
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"
    }
}
let threeOfSpades = Card(rank: .Three, suit: .Spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

Protocols

- A **protocol** defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality.
- The protocol doesn’t actually provide an implementation for any of these requirements—it only describes what an implementation will look like.

- The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements.
- Any type that satisfies the requirements of a protocol is said to **conform** to that protocol.

- 出一張嘴，只宣告不定義的 protocol
- protocol 如同 struct、class 和 enum 一樣可以宣告方法、屬性和 subscript
- 但是 protocol 只出一張嘴完全不會定義實際執行的程式碼內容
- 將定義程式碼的重責大任交給答應遵循從 protocol 的 struct、class 和 enum



```swift
protocol ExampleProtocol {
    var simpleDescription: String { get }
    func adjust()
}
```

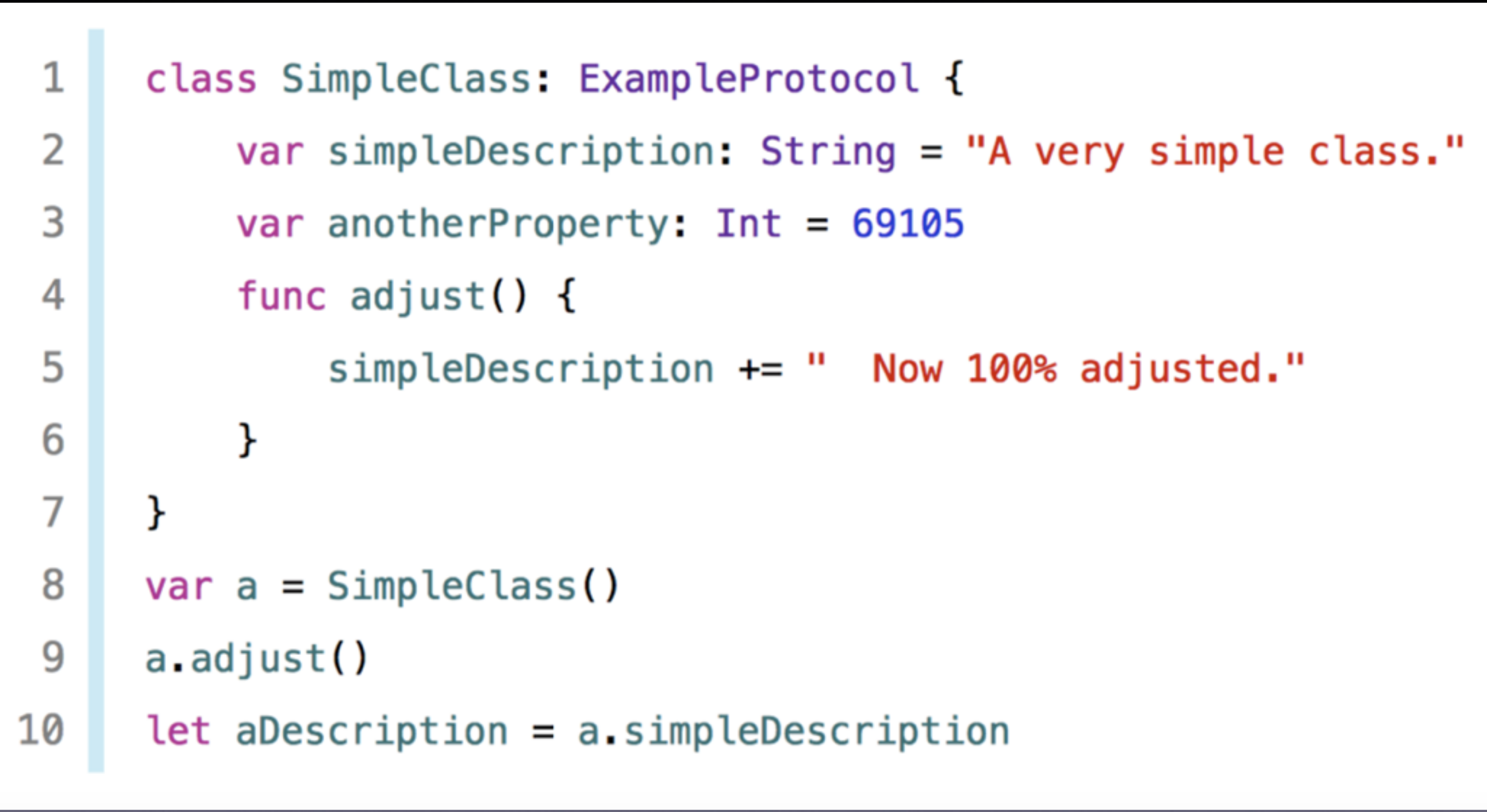
- The **{ get }** following the **simpleDescription** property indicates that it is read-only, meaning that the value of the property can be viewed, but never changed.

- Protocols can require that conforming types have specific instance properties, instance methods, type methods, operators, and subscripts.
- Protocols can require specific instance methods and type methods to be implemented by conforming types.

- These methods are written as part of the protocol’s definition in exactly the same way as for normal instance and type methods, but without curly braces or a method body.
- Classes, structures, and enumerations adopt a protocol by listing its name after their name, separated by a colon.

- A type can adopt any number of protocols, which appear in a comma-separated list.
- If a class has a superclass, the superclass's name must appear first in the list, followed by protocols.
- You conform to the protocol by implementing all of its requirements.

- 遵從 protocol 和繼承雖然是同一個語法，但 protocol 卻有更大的彈性
- 可以同時遵從多個 protocol 彼此以**逗號**分隔
- 同時繼承和遵從 protocol 是可行的，只要尊敬地把父類別擺在第一個，接著再以逗號串接遵從的 protocol 即可



```swift
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += "  Now 100% adjusted."
    }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription
```

- Protocols are first-class types, which means they can be treated like other named types.

```swift
class SimpleClass2: ExampleProtocol {
    var simpleDescription: String = "Another very simple class."
    func adjust() {
        simpleDescription += "  Adjusted."
    }
}

var protocolArray: [ExampleProtocol] = [SimpleClass(), SimpleClass(), SimpleClass2()]
for instance in protocolArray {
    instance.adjust()
}
protocolArray
```

參考資料

- Start Developing iOS Apps (Swift)
- The Swift Programming Language (Swift 2.1)