Chapter 14: Conditionals

Conditional expressions, involving keywords such as if, else if, and else, provide Swift programs with the ability to perform different actions depending on a Boolean condition: True or False. This section covers the use of Swift conditionals, Boolean logic, and ternary statements.

Section 14.1: Optional binding and "where" clauses

Optionals must be *unwrapped* before they can be used in most expressions. `if let` is an *optional binding*, which succeeds if the optional value was **not** `nil`:

```
let num: Int? = 10 // or: let num: Int? = nil

if let unwrappedNum = num {
    // num has type Int?; unwrappedNum has type Int
    print("num was not nil: \(unwrappedNum + 1)")
} else {
    print("num was nil")
}
```

You can reuse the **same name** for the newly bound variable, shadowing the original:

```
// num originally has type Int?
if let num = num {
    // num has type Int inside this block
}
```

Version ≥ 1.2 Version < 3.0

Combine multiple optional bindings with commas (`,`):

```
if let unwrappedNum = num, let unwrappedStr = str {
    // Do something with unwrappedNum & unwrappedStr
} else if let unwrappedNum = num {
    // Do something with unwrappedNum
} else {
    // num was nil
}
```

Apply further constraints after the optional binding using a `where` clause:

`if let unwrappedNum = num` e unwrappedNum % 2 == 0 { print("num is non-nil, and it's an even number") }

If you're feeling adventurous, interleave any number of optional bindings and `where` clauses:

```
if let num = num                            // num must be non-nil
    where num % 2 == 1,                     // num must be odd
    let str = str,                          // str must be non-nil
    let firstChar = str.characters.first    // str must also be non-empty
    where firstChar != "x"                  // the first character must not be "x"
{
    // all bindings & conditions succeeded!
}
```

Version ≥ 3.0

In Swift 3, `where` clauses have been replaced (SE-0099): simply use another `,` to separate optional bindings and boolean conditions.

`if let unwrappedNum = num`wrappedNum % 2 == 0 { print("num is non-nil, and it's an even number") } if let num = num, // num must be non-nil num % 2 == 1, // num must be odd let str = str, // str must be non-nil let firstChar = str.characters.first, // str must also be non-empty firstChar != "x" // the first character must not be "x" { // all bindings & conditions succeeded! }

Section 14.2: Using Guard

`Version ≥ 2.0`

Guard checks for a condition, and if it is false, it enters the branch. Guard check branches must leave its enclosing block either via `return`, `break`, or `continue` (if applicable); failing to do so results in a compiler error. This has the advantage that when a `guard` is written it's not possible to let the flow continue accidentally (as would be possible with an `if`).

Using guards can help keep nesting levels low, which usually improves the readability of the code.

```
func printNum(num: Int) {
    guard num == 10 else {
        print("num is not 10")
        return
    }
    print("num is 10")
}
```

Guard can also check if there is a value in an optional, and then unwrap it in the outer scope:

```
func printOptionalNum(num: Int?) {
    guard let unwrappedNum = num else {
        print("num does not exist")
        return
    }
    print(unwrappedNum)
}
```

Guard can combine optional unwrapping and condition check using `where` keyword:

```
func printOptionalNum(num: Int?) {
guard let unwrappedNum = num, unwrappedNum == 10 else {
    print("num does not exist or is not 10")
    return
}
print(unwrappedNum)
}
```

Section 14.3: Basic conditionals: if-statements

An **`if` statement** checks whether a Bool condition is `true`:

```
let num = 10

if num == 10 {
    // Code inside this block only executes if the condition was true.
    print("num is 10")
}

let condition = num == 10   // condition's type is Bool
if condition {
```

```
    print("num is 10")
}
```

`if` statements accept `else if` and `else` blocks, which can test alternate conditions and provide a fallback:

```
let num = 10
if num < 10 {  // Execute the following code if the first condition is true.
    print("num is less than 10")
} else if num == 10 {  // Or, if not, check the next condition...
    print("num is 10")
} else {  // If all else fails...
    print("all other conditions were false, so num is greater than 10")
}
```

Basic operators like `&&` and `||` can be used for multiple conditions:

The logical AND operator

```
let num = 10
let str = "Hi"
if num == 10 && str == "Hi" {
    print("num is 10, AND str is \"Hi\"")
}
```

If `num == 10` was false, the second value wouldn't be evaluated. This is known as short-circuit evaluation.

The logical OR operator

```
if num == 10 || str == "Hi" {
    print("num is 10, or str is \"Hi\")
}
```

If `num == 10` is true, the second value wouldn't be evaluated.

The logical NOT operator

```
if !str.isEmpty {
    print("str is not empty")
}
```

Section 14.4: Ternary operator

Conditions may also be evaluated in a single line using the ternary operator:

If you wanted to determine the minimum and maximum of two variables, you could use if statements, like so:

```
let a = 5
let b = 10
let min: Int

if a < b {
    min = a
} else {
    min = b
}

let max: Int

if a > b {
    max = a
```

```
} else {
    max = b
}
```

The ternary conditional operator takes a condition and returns one of two values, depending on whether the condition was true or false. The syntax is as follows: This is equivalent of having the expression:

```
(<CONDITION>) ? <TRUE VALUE> : <FALSE VALUE>
```

The above code can be rewritten using ternary conditional operator as below:

```
let a = 5
let b = 10
let min = a < b ? a : b
let max = a > b ? a : b
```

In the first example, the condition is a < b. If this is true, the result assigned back to min will be of a; if it's false, the result will be the value of b.

Note: Because finding the greater or smaller of two numbers is such a common operation, the Swift standard library provides two functions for this purpose: max and min.

Section 14.5: Nil-Coalescing Operator

The nil-coalescing operator `<OPTIONAL> ?? <DEFAULT VALUE>` unwraps the `<OPTIONAL>` if it contains a value, or returns `<DEFAULT VALUE>` if is nil. `<OPTIONAL>` is always of an optional type. `<DEFAULT VALUE>` must match the type that is stored inside `<OPTIONAL>`.

The nil-coalescing operator is shorthand for the code below that uses a ternary operator:

```
a != nil ? a! : b
```

this can be verified by the code below:

```
(a ?? b) == (a != nil ? a! : b) // outputs true
```

Time For An Example

```
let defaultSpeed:String = "Slow"
var userEnteredSpeed:String? = nil

print(userEnteredSpeed ?? defaultSpeed) // outputs "Slow"

userEnteredSpeed = "Fast"
print(userEnteredSpeed ?? defaultSpeed) // outputs "Fast"
```