Optionals

You use optionals in situations where a value may be absent. An optional represents two possibilities: Either there is a value, and you can unwrap the optional to access that value, or there isn't a value at all.

nil: You set an optional variable to a valueless state by assigning it the special value nil:

```
var serverResponseCode: Int? = 404
// serverResponseCode contains an actual Int value of 404
serverResponseCode = nil
// serverResponseCode now contains no value
```

You can't use nil with non-optional constants and variables. If a constant or variable in your code needs to work with the absence of a value under certain conditions, always declare it as an optional value of the appropriate type.

If you define an optional variable without providing a default value, the variable is automatically set to nil for you:

```
var surveyAnswer: String?
// surveyAnswer is automatically set to nil
```

<u>If Statements and Forced Unwrapping</u>

You can use an if statement to find out whether an optional contains a value by comparing the optional against nil. You perform this comparison with the "equal to" operator (==) or the "not equal to" operator (!=).

If an optional has a value, it's considered to be "not equal to" nil:

```
if convertedNumber != nil {
    print("convertedNumber contains some integer value.")
}
// Prints "convertedNumber contains some integer value."
```

Once you're sure that the optional *does* contain a value, you can access its underlying value by adding an exclamation point (!) to the end of the

optional's name. The exclamation point effectively says, "I know that this optional definitely has a value; please use it." This is known as *forced unwrapping* of the optional's value:

```
if convertedNumber != nil {
    print("convertedNumber has an integer value of \
(convertedNumber!).")
}
// Prints "convertedNumber has an integer value of 123."
```

Trying to use ! to access a nonexistent optional value triggers a runtime error. Always make sure that an optional contains a non-nil value before using ! to force-unwrap its value.

Optional Binding

You use *optional binding* to find out whether an optional contains a value, and if so, to make that value available as a temporary constant or variable. Optional binding can be used with if and while statements to check for a value inside an optional, and to extract that value into a constant or variable, as part of a single action.

Write an optional binding for an if statement as follows:

```
if let constantName = someOptional {
    statements
}
```

You can rewrite the possibleNumber example from the Optionals section to use optional binding rather than forced unwrapping:

```
if let actualNumber = Int(possibleNumber) {
    print("The string \"\(possibleNumber)\" has an integer value of \
(actualNumber)")
} else {
    print("The string \"\(possibleNumber)\" couldn't be converted to
an integer")
}
// Prints "The string "123" has an integer value of 123"
```

This code can be read as:

"If the optional Int returned by Int(possibleNumber) contains a value, set a new constant called actualNumber to the value contained in the

optional.”

If the conversion is successful, the actualNumber constant becomes available for use within the first branch of the if statement. It has already been initialized with the value contained *within* the optional, and so you don’t use the ! suffix to access its value. In this example, actualNumber is simply used to print the result of the conversion.

Guard Statement

In Swift, a **guard** statement is similar to **if** statement except that it is used to transfer the program control out of scope if the condition(s) are not met.
A guard statement has the following form:

```
guard condition else {
   statements
}
```

Where **condition** evaluates to **true** or **false** and:

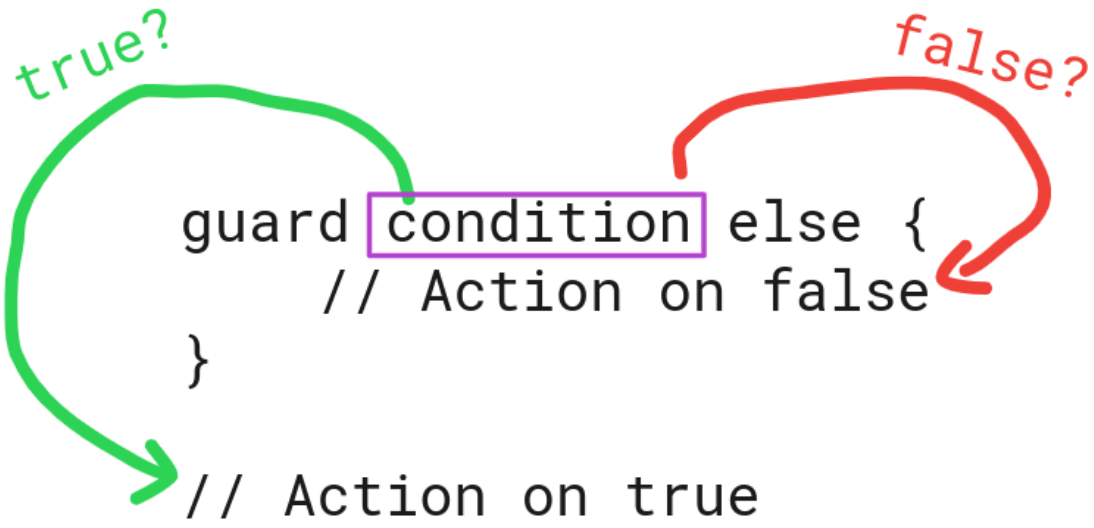
- If the **condition** is **true** the **guard** block is not executed.
- If the **condition** is **false** the **guard** block gets executed.

The else clause of a guard statement is required, and must either call a function with the Never return type or transfer program control outside the guard statement’s enclosing scope using one of the following statements:

- return
- break
- continue
- throw

For example, this piece of code continues execution if the age is 18 or more.

```
guard age >= 18 else { return false }
```



So, use **`if let`** if you just want to unwrap some optionals, but prefer **`guard let`** if you're specifically checking that conditions are correct before continuing.

Example:

```
func nameCheck() {
    var name: String? = "Alice"
    guard let myName = name else {
        print("Name is not defined")
        return
    }
    print("My name is \(myName)")
}

nameCheck()
```

The guard statement works such that it:

- Checks if the **name** variable is not **nil**.
- If the **name** is not **nil**, the **name** is assigned to a new constant **myName**.
- If the **name** is **nil**, the guard exits the function.