Types, Operators

Sun*

Outline

1. Basic Data Types
2. Optionals

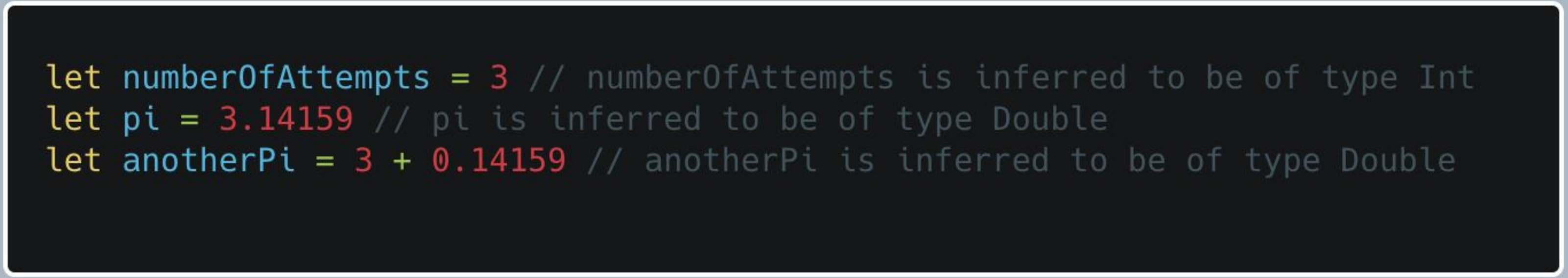
1. Data Types

1. Numeric Types
2. Booleans
3. Tuples
4. Type Aliases

Sun*

1.1 Numeric Types

❖ Numbers can be declared to be of type *Int*, *Double* (64-bit floating-point number), *Float* (32-bit floating-point number)

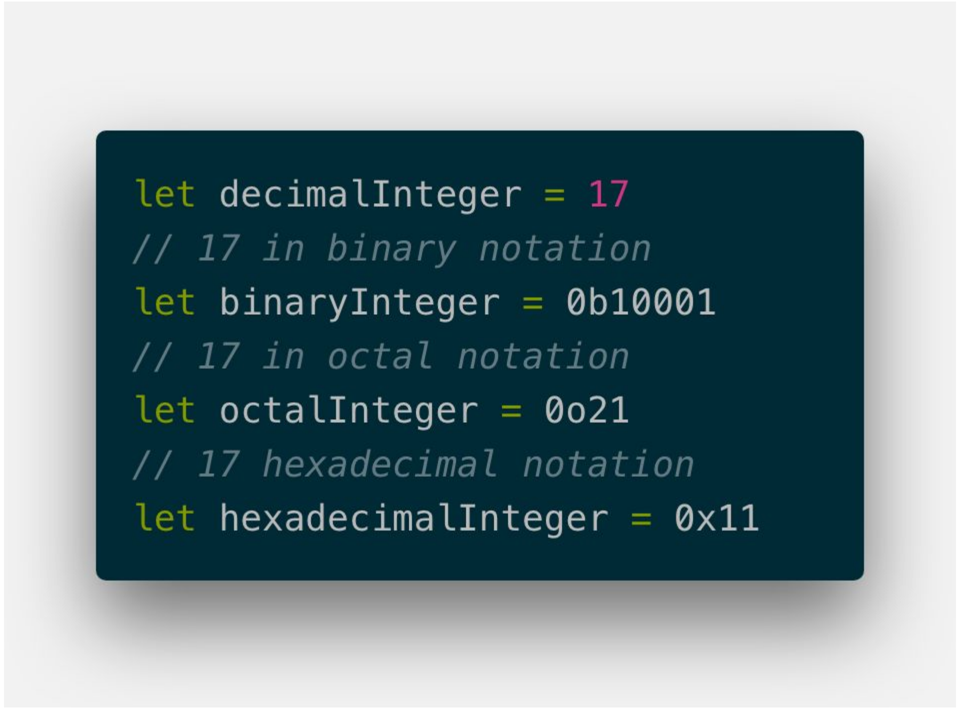


```
let numberOfAttempts = 3 // numberOfAttempts is inferred to be of type Int
let pi = 3.14159 // pi is inferred to be of type Double
let anotherPi = 3 + 0.14159 // anotherPi is inferred to be of type Double
```

1.1 Numeric Types (cont)

Integer literals can be written as:

- A decimal number, with no prefix
- A binary number, with a 0b prefix
- An octal number, with a 0o prefix
- A hexadecimal number, with a 0x prefix

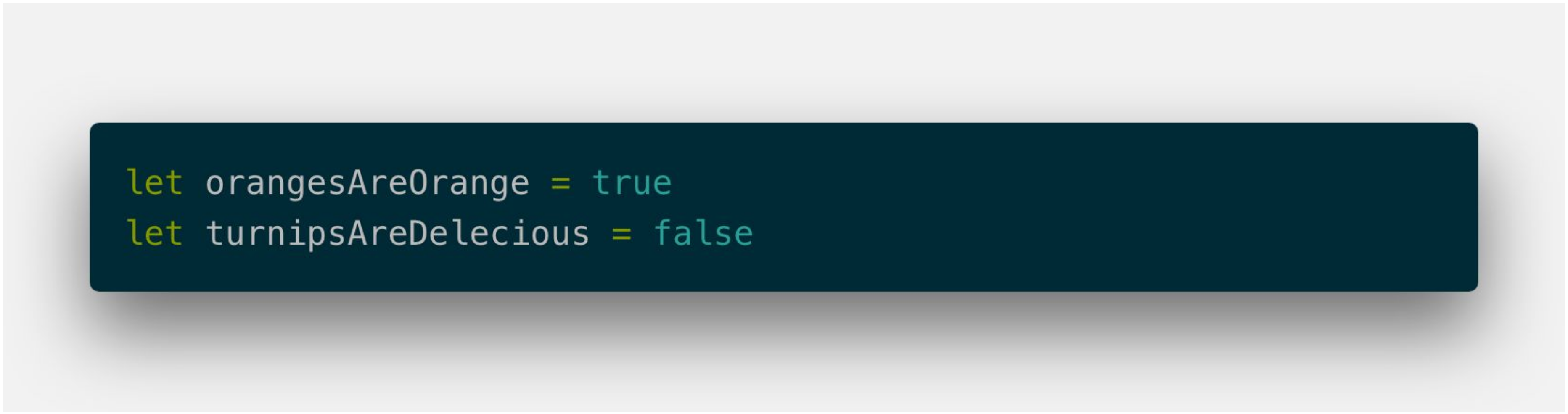


```
let decimalInteger = 17
// 17 in binary notation
let binaryInteger = 0b10001
// 17 in octal notation
let octalInteger = 0o21
// 17 hexadecimal notation
let hexadecimalInteger = 0x11
```

Sun*

1.2 Boolean

❖ Swift has a basic *Boolean* type, called *Bool*.

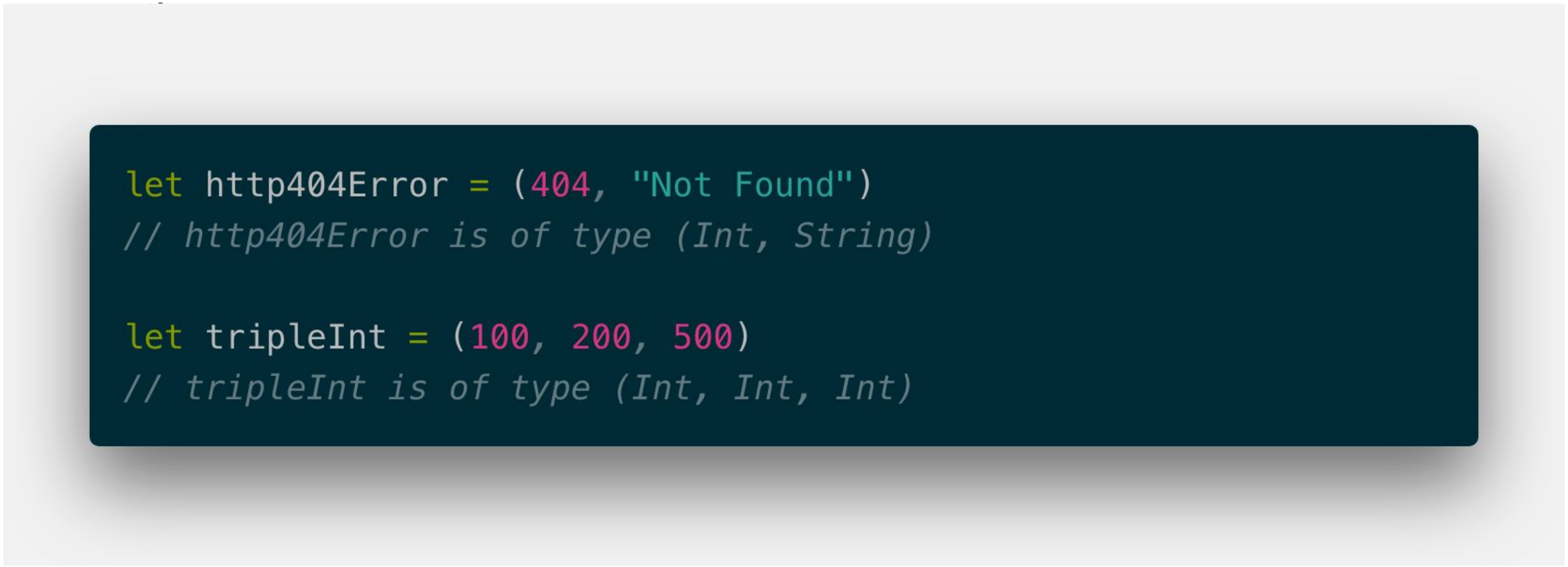


```
let orangesAreOrange = true
let turnipsAreDelecious = false
```

Sun*

1.3 Tuples

❖ *Tuples* group multiple values into a single compound value. The values within a tuple can be of any type and don’t have to be the save type as each



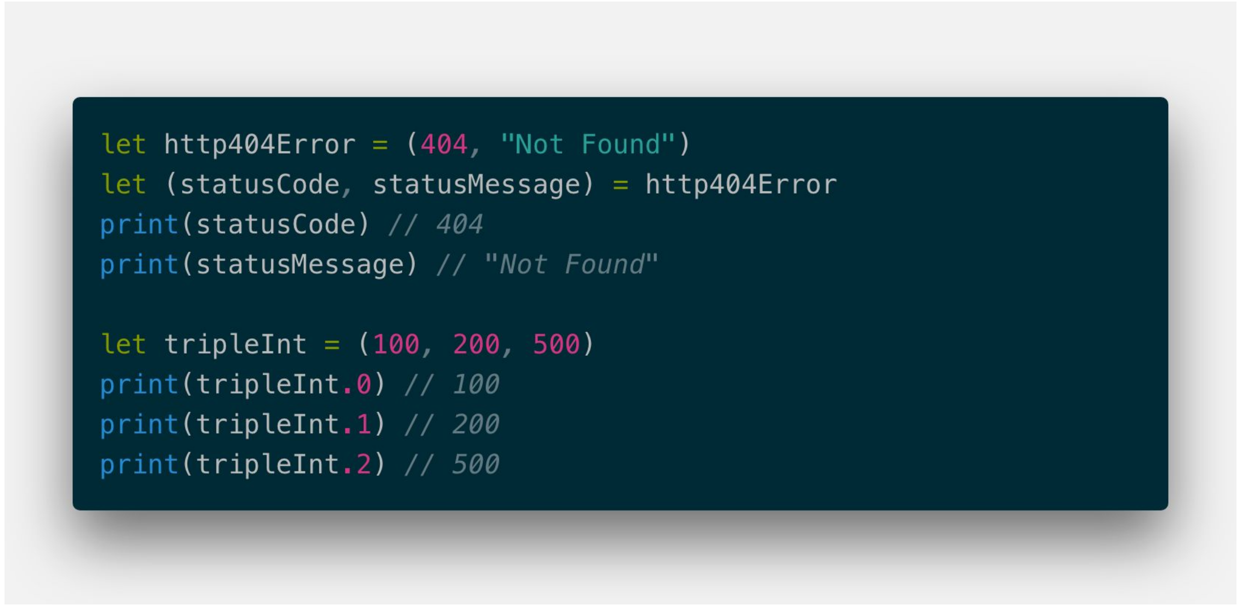
```
let http404Error = (404, "Not Found")
// http404Error is of type (Int, String)

let tripleInt = (100, 200, 500)
// tripleInt is of type (Int, Int, Int)
```

Sun*

1.3 Tuples (cont)

❖ You can decompose a tuple’s contents into separate constants or variables, which you then access as usual:



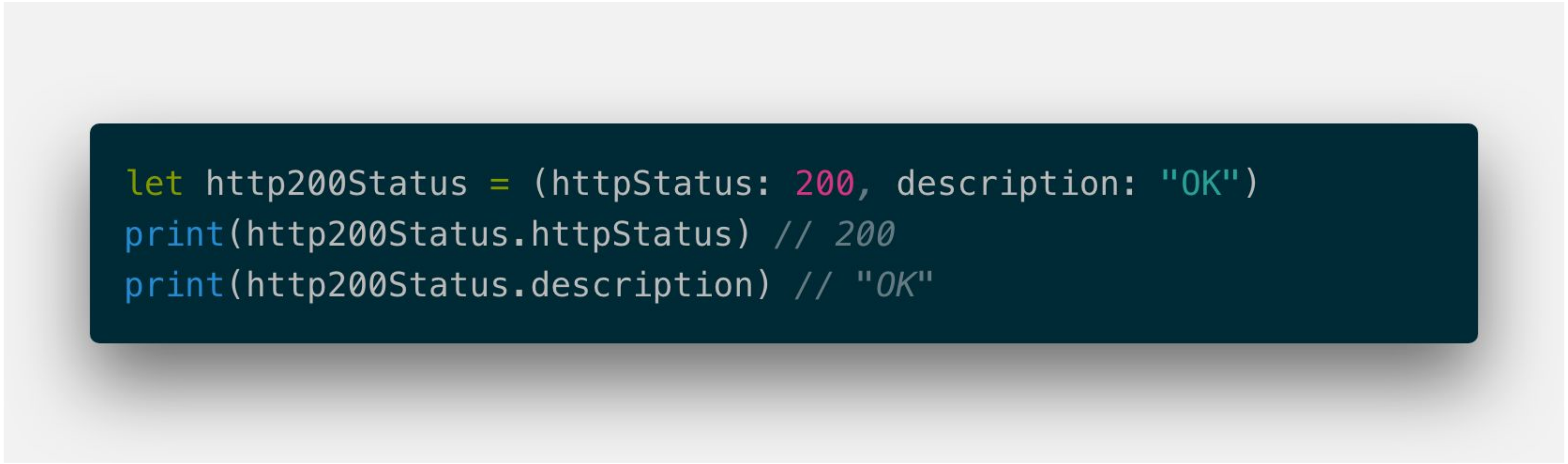
```
let http404Error = (404, "Not Found")
let (statusCode, statusMessage) = http404Error
print(statusCode) // 404
print(statusMessage) // "Not Found"

let tripleInt = (100, 200, 500)
print(tripleInt.0) // 100
print(tripleInt.1) // 200
print(tripleInt.2) // 500
```

Sun*

1.3 Tuples (cont)

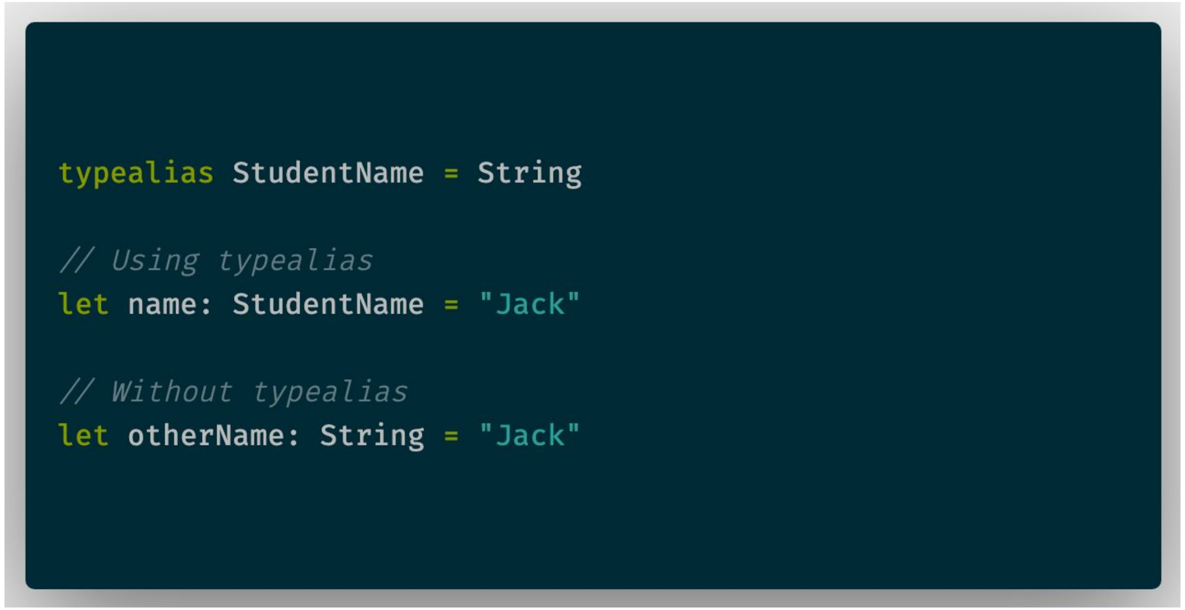
- ❖ You can name the individual elements in a tuple when the tuple is defined. It’s easier to access the values of those elements later:



```
let http200Status = (httpStatus: 200, description: "OK")
print(http200Status.httpStatus) // 200
print(http200Status.description) // "OK"
```

1.4 Type Aliases:

❖ *Type aliases* define an alternative name for an existing type by using `typealias` keyword.



```
typealias StudentName = String

// Using typealias
let name: StudentName = "Jack"

// Without typealias
let otherName: String = "Jack"
```

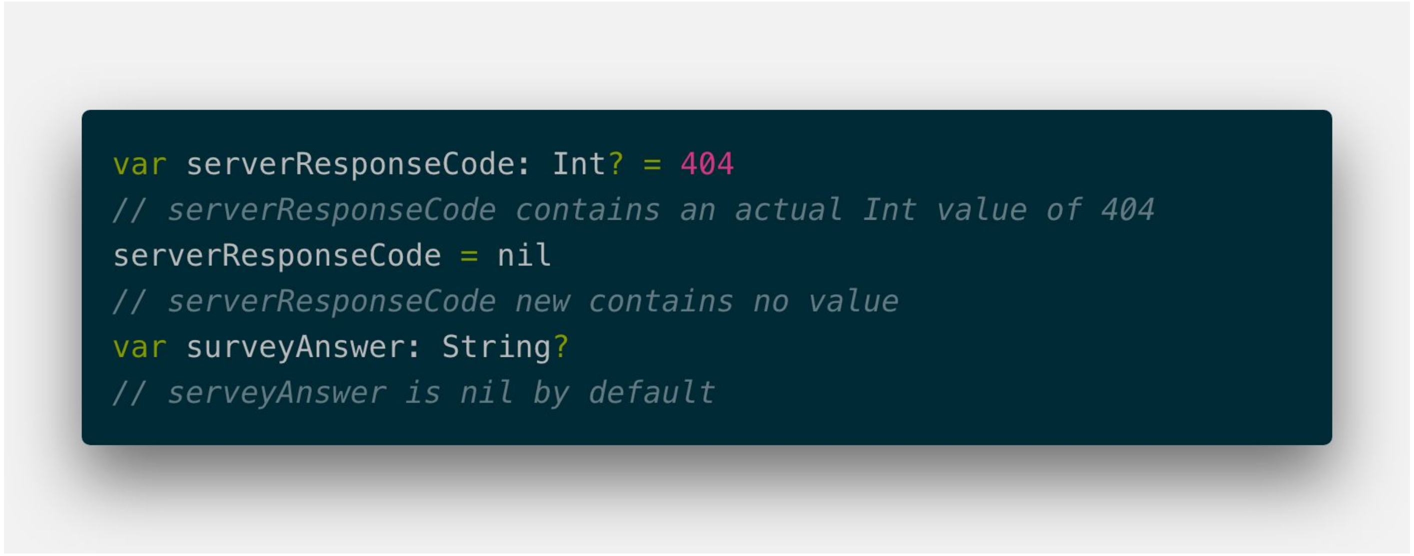
Sun*

2. Optionals

1. Definition
2. Unwrapping an optional
3. Compare ways to unwrap an optional

2.1 Definition

- *Optionals* are used in situations where a value may be absent.
- An optional represents two possibilities: Either there is a value, or there isn't a value at all.



```
var serverResponseCode: Int? = 404
// serverResponseCode contains an actual Int value of 404
serverResponseCode = nil
// serverResponseCode new contains no value
var surveyAnswer: String?
// serveyAnswer is nil by default
```

2.2 Unwrapping an optional

❖ If you defined a variable as optional, then to get the value from this variable, you will have to unwrap it.

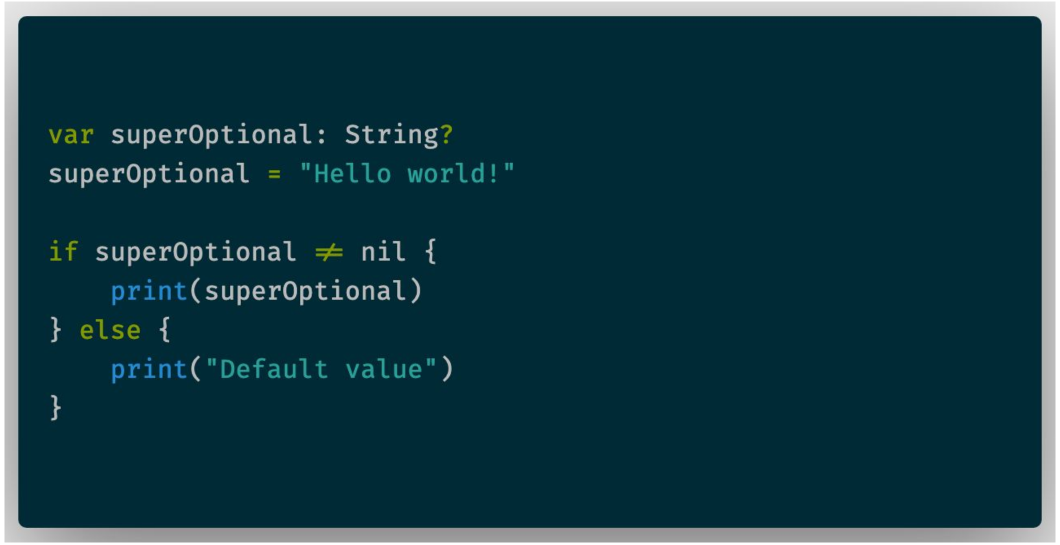
❖ There are six ways to unwrap actual values from an Optional:

- ➔ Unwrapping `Optionals` using if-else conditions
- ➔ Nil coalescing operator
- ➔ Force unwrapping
- ➔ Optional binding (`if let`)
- ➔ `guard let`
- ➔ Optional chaining

Sun*

2.2 Unwrapping an optional (cont)

❖ **Unwrapping Optionals using if-else conditions** is the way of unwrapping optionals in which we use if-else conditions to check if the `Optional` holds a `nil` value or not. If the `Optional` doesn't hold a `nil` value, we unwrap it, otherwise we print a string "Default value"



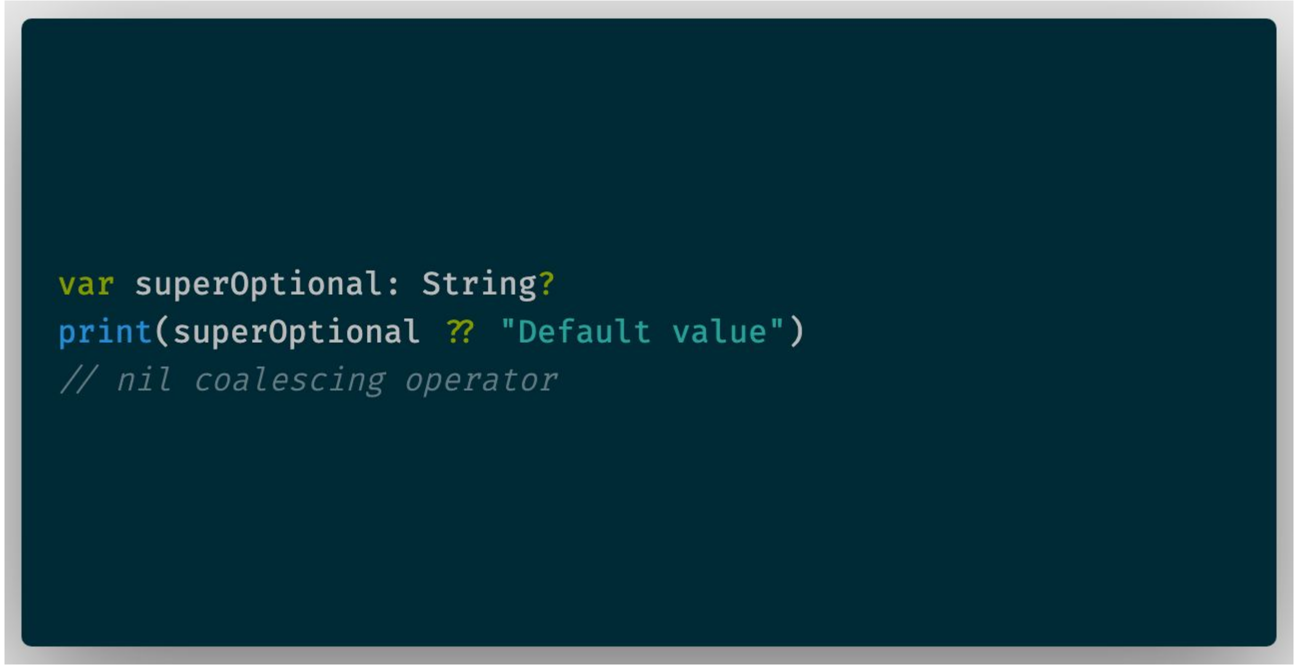
```swift
var superOptional: String?
superOptional = "Hello world!"

if superOptional ≠ nil {
    print(superOptional)
} else {
    print("Default value")
}
```

Sun*

2.2 Unwrapping an optional (cont)

❖ **Nil coalescing operator** is another way of unwrapping optionals. This is a better and shorter way of unwrapping than using **if-else conditions**:

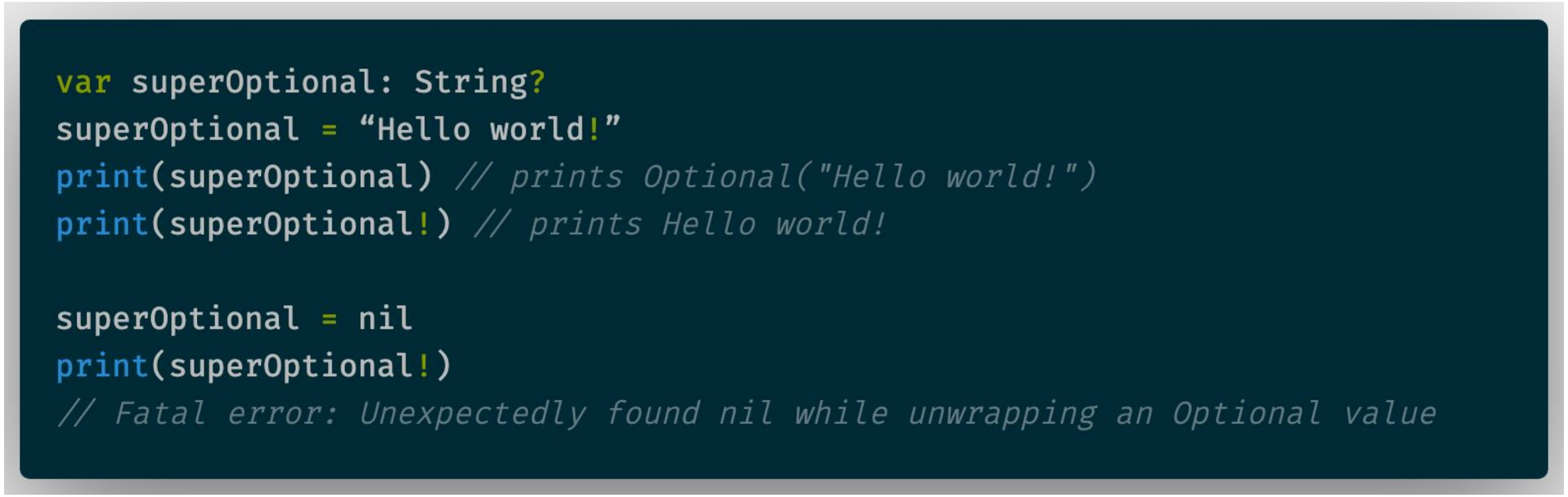


```swift
var superOptional: String?
print(superOptional ?? "Default value")
// nil coalescing operator
```

Sun*

2.2 Unwrapping an optional (cont)

❖ **Force unwrapping** is a way of unwrapping values from optionals where we forcibly unwrap the value from the variable without caring if the `Optional` has an actual value or a `nil` value.

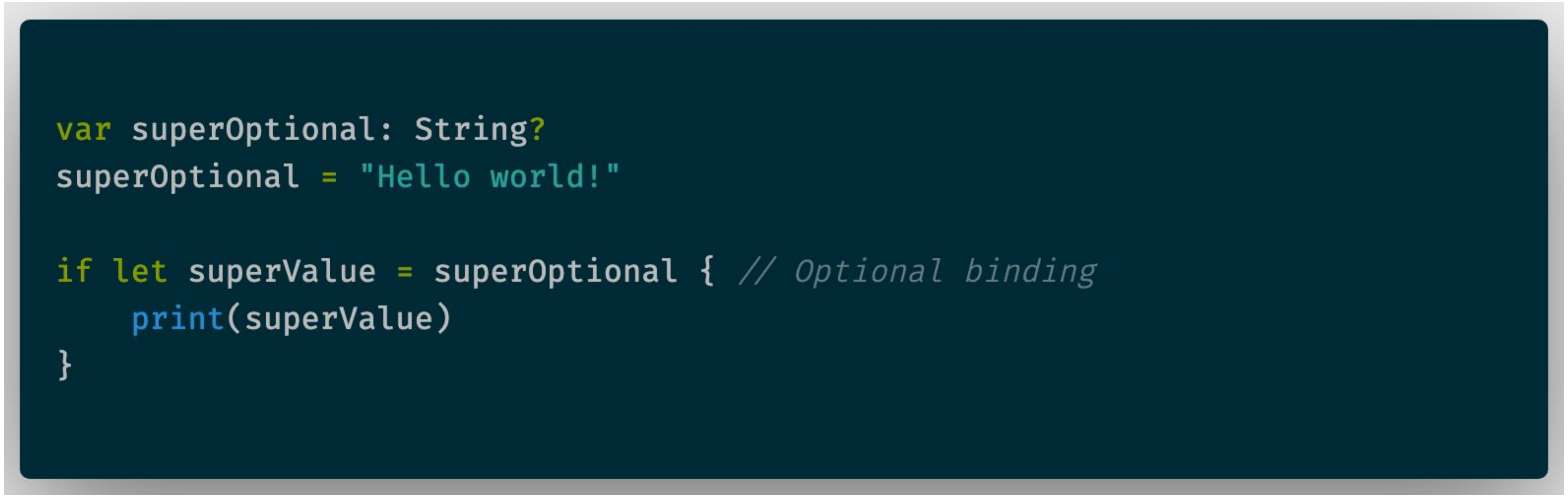


```
var superOptional: String?
superOptional = “Hello world!”
print(superOptional) // prints Optional("Hello world!")
print(superOptional!) // prints Hello world!

superOptional = nil
print(superOptional!)
// Fatal error: Unexpectedly found nil while unwrapping an Optional value
```

2.2 Unwrapping an optional (cont)

- **Optional binding** is a safe alternative to force unwrapping. In this method, we move to the unwrapping of the `Optional` only after making sure that it holds an actual value. For this purpose, we make use of `if let`.



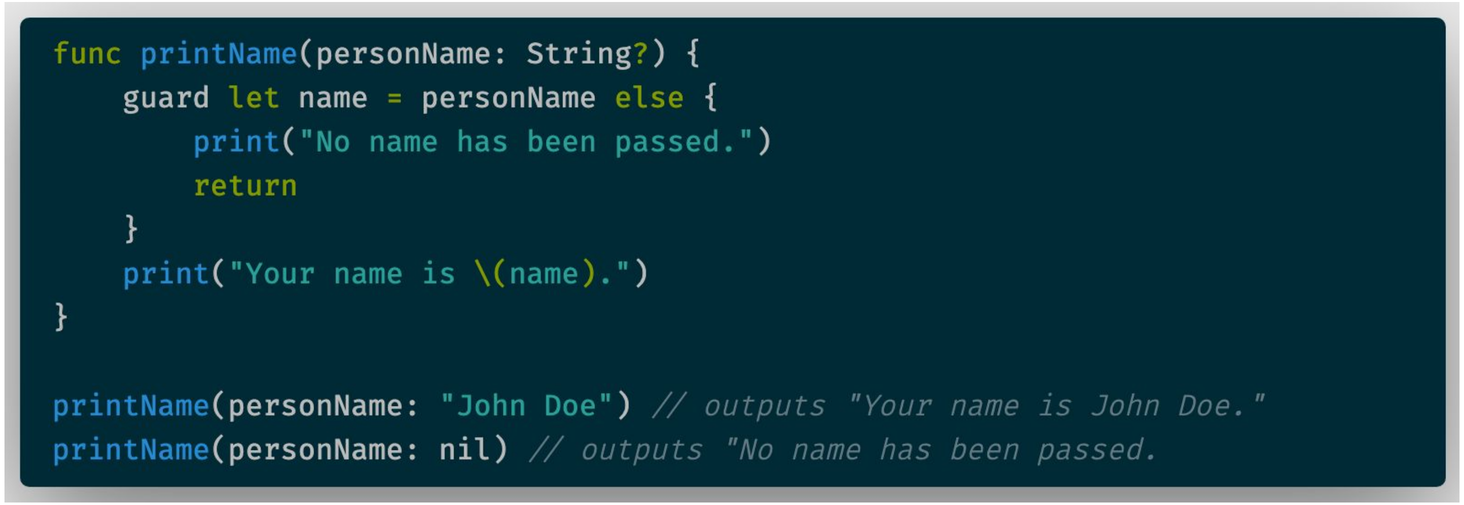
```swift
var superOptional: String?
superOptional = "Hello world!"

if let superValue = superOptional { // Optional binding
    print(superValue)
}
```

Sun*

2.2 Unwrapping an optional (cont)

- `guard let` is a nice alternative to if let for the purpose of unwrapping optionals. If guard let gets `nil` value of the `Optional` we are trying to unwrap, it expects us to exit the function, loop or condition we used it in. However, the difference between `if let` and `guard let` is that we can still make use of our unwrapped optional, even after the `guard let` code.

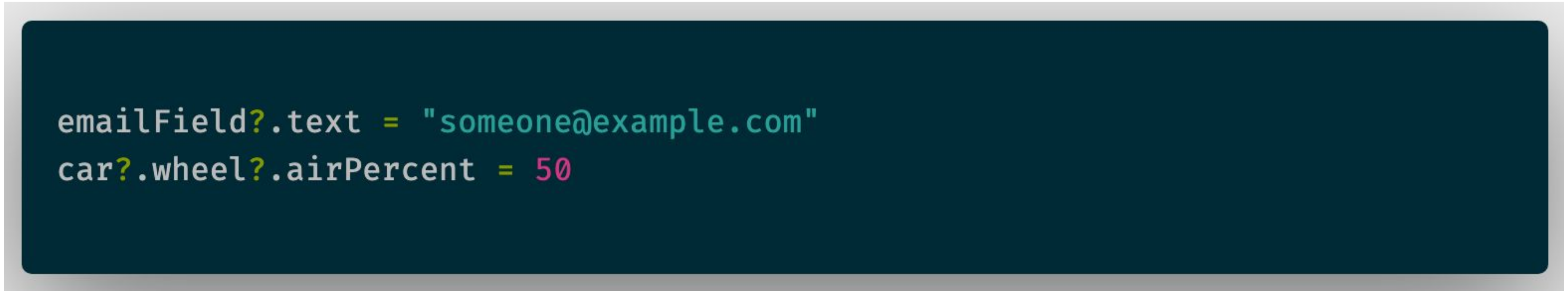


```
func printName(personName: String?) {
    guard let name = personName else {
        print("No name has been passed.")
        return
    }
    print("Your name is \(name).")
}

printName(personName: "John Doe") // outputs "Your name is John Doe."
printName(personName: nil) // outputs "No name has been passed.
```

2.2 Unwrapping an optional (cont)

- If we have to deal with multiple optionals at once, **optional chaining** can be a useful approach.



```
emailField?.text = "someone@example.com"
car?.wheel?.airPercent = 50
```

Sun*

2.3 Compare ways to unwrap an optional

	Pros	Cons
Unwrapping `Optionals` using if-else conditions	Familiar way to unwrap value	It's too long and messy and not recommended
Nil coalescing operator	Better and shorter than if-else conditions	Must provide a default value
Force unwrapping	The easiest way to unwrap values	It leads to a fatal error in case the `Optional` is holding a `nil` value
Optional binding (`if let`)	A safe alternative to force unwrapping	Binding value can only be used inside `if` block
`guard let`	Binding value can be used in the same block of `guard let` statement	Require else `block` and `return` statement
Optional chaining	Easily deal with multiple optionals at once	Return values are `Optional`

Sun*

Question & Answer?

Sun*