```
import UIKit

//: # Basics
//: ## Constants and Variables
//: ### The value of a constant can't be changed once it's set, whereas a
 variable can be set to a different value in the future.
//: ### Constants and variables must be declared before they're used. You
 declare constants with the let keyword and variables with the var keyword.
//:
//:     var age = 20
//:     var isReady = true
//:     let myName = "Mohammad"
//:     let pi = 3.14
//:     let dateOfBirth = "X-X-2018"
//: ### Use var when you expect the value to change
var myAge = 20
myAge = 25
myAge = 30
//: ### Use let when you know the value will stay constant
let encouragement = "You can do it!"
// encouragement cannot be changed later
//: ### Example 2a
let birthYear = 2008
var currentYear = 2015
var age = currentYear - birthYear
//: ### You can declare multiple constants or multiple variables on a single
 line, separated by commas:
var x = 0.0, y = 0.0, z = 0.0
//: ## Type Annotations
var welcomeMessage: String
//: ### The colon in the declaration means "…of type…," so the code above can
 be read as: "Declare a variable called welcomeMessage that is of type String."
//: ### You can define multiple related variables of the same type on a single
 line, separated by commas, with a single type annotation after the final
 variable name:
var red, green, blue: Double
//: ### Constant and variable names can contain almost any character,
 including Unicode characters:
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
//: ## Printing Constants and Variables
//: ### You can print the current value of a constant or variable with the
 print(_:separator:terminator:) function:
//: ### Declaration:
//: ### func print(_ items: Any..., separator: String = default, terminator:
 String = default)
var friendlyWelcome = "Hello!"
print(friendlyWelcome)
// Prints "Hello!"
```

```
//: ### Swift uses string interpolation to include the name of a constant or
 variable as a placeholder in a longer string, and to prompt Swift to replace
 it with the current value of that constant or variable.
print("The current value of friendlyWelcome is \(friendlyWelcome)")
// Prints "The current value of friendlyWelcome is Hello!"
//: ## Comments
//: ### Use comments to include nonexecutable text in your code, as a note or
 reminder to yourself.
// This is a comment.

/* This is also a comment
 but is written over multiple lines. */
//: ### Multiline comments in Swift can be nested inside other multiline
 comments.
/* This is the start of the first multiline comment.
 /* This is the second, nested multiline comment. */
 This is the end of the first multiline comment. */
//: ## Semicolons
//: ### Swift doesn't require you to write a semicolon (;) after each
 statement in your code
//: ### semicolons are required if you want to write multiple separate
 statements on a single line:
let cat = "🐱"; print(cat)
// Prints "🐱"
//: ## Integers
//: ### Integers are whole numbers with no fractional component, such as 42
 and -23.
var num1 = 42
//: ## UInt
//: ### Swift also provides an unsigned integer type, UInt
var num2 : UInt = 23
//: ## Floating-Point Numbers
//: ### Floating-point numbers are numbers with a fractional component, such
 as 3.14159
//: ### -> Double represents a 64-bit floating-point number.
//: ### -> Float represents a 32-bit floating-point number.
//: ## Type Safety and Type Inference
var distance = 23
// distance is infered to be of type Int
let pi = 3.14
// pi is inferred to be of type Double
//: ## Numeric Literals
//: ### 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
//: All of these integer literals have a decimal value of 17:
let decimalInteger = 17
let binaryInteger = 0b10001       // 17 in binary notation
```

```
let octalInteger = 0o21           // 17 in octal notation
let hexadecimalInteger = 0x11     // 17 in hexadecimal notation
//: ### For decimal numbers with an exponent of exp, the base number is
 multiplied by 10exp:
//:     1.25e2 means 1.25 x 102, or 125.0.
//:     1.25e-2 means 1.25 x 10-2, or 0.0125.
//: ## Type Aliases
//: ### Type aliases define an alternative name for an existing type. You
 define type aliases with the typealias keyword.
typealias AudioSample = UInt16
//: ### Once you define a type alias, you can use the alias anywhere you might
 use the original name:
var maxAmplitudeFound = AudioSample.min
// maxAmplitudeFound is now 0
//: ## Booleans
//: ### Swift has a basic Boolean type, called Bool.
//: Swift provides two Boolean constant values, true and false:
let orangesAreOrange = true
let isRainy = false
//: Boolean values are particularly useful when you work with conditional
 statements such as the if statement:
if isRainy {
    print("we have to stay at home")
} else {
    print("let's play soccer")
}
//: ## 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 of the same type as
 each other.
let http404Error = (404, "Not Found")
// http404Error is of type (Int, String), and equals (404, "Not Found")
//: ### A tuple can contain as many different types as you like
//:     (Int, Int, Int)
//:     (String, Bool)
let (statusCode, statusMessage) = http404Error
print("The status code is \(statusCode)")
// Prints "The status code is 404"
print("The status message is \(statusMessage)")
// Prints "The status message is Not Found"
//: ### If you only need some of the tuple’s values, ignore parts of the tuple
 with an underscore (_) when you decompose the tuple:
let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)")
// Prints "The status code is 404"
//: ### Alternatively, access the individual element values in a tuple using
 index numbers starting at zero:
print("The status code is \(http404Error.0)")
// Prints "The status code is 404"
print("The status message is \(http404Error.1)")
// Prints "The status message is Not Found"
```

```
//: ### You can name the individual elements in a tuple when the tuple is
 defined:
let http200Status = (statusCode: 200, description: "OK")
print("The status code is \(http200Status.statusCode)")
// Prints "The status code is 200"
print("The status message is \(http200Status.description)")
// Prints "The status message is OK"
//: ## Optional
//: ### You use optionals 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.
//: ### The example below uses the initializer to try to convert a String into
 an Int:
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
// convertedNumber is inferred to be of type "Int?", or "optional Int"
//: Because the initializer might fail, it returns an optional Int, rather
 than an Int. An optional Int is written as Int?, not Int.
//: ## 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
//: ### 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
//: ## If Statements and Forced Unwrapping
//: ### 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."
//: ### Forced Unwrapping
if convertedNumber != nil {
    print("convertedNumber has an integer value of \(convertedNumber!).")
}
// Prints "convertedNumber has an integer value of 123."
//: ### The exclamation mark effectively says, "I know that this optional
 definitely has a value; please use it."
//: ## Optional Binding
//: ### Optional binding is used to find out whether an optional contains a
 value, and if so, to make that value available as a temporary constant or
 variable.
//:     if let constantName = someOptional {
//:         statements
//:     }
if let actualNumber = Int(possibleNumber) {
    print("\"\(possibleNumber)\" has an integer value of \(actualNumber)")
```

```
} else {
    print("\"\(possibleNumber)\" could not be converted to an integer")
}
// Prints ""123" has an integer value of 123"
//: ### You can include as many optional bindings and Boolean conditions in a
 single if statement as you need to, separated by commas.
//: ### The following if statements are equivalent:
if let firstNumber = Int("4"), let secondNumber = Int("42"), firstNumber <
 secondNumber && secondNumber < 100 {
    print("\(firstNumber) < \(secondNumber) < 100")
}
// Prints "4 < 42 < 100"

if let firstNumber = Int("4") {
    if let secondNumber = Int("42") {
        if firstNumber < secondNumber && secondNumber < 100 {
            print("\(firstNumber) < \(secondNumber) < 100")
        }
    }
}
// Prints "4 < 42 < 100"
//: ## Implicitly Unwrapped Optionals
//: ### Sometimes it's clear from a program's structure that an optional will
 always have a value
let possibleString: String? = "An optional string."
let forcedString: String = possibleString! // requires an exclamation mark

let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString // no need for an exclamation mark
```