Creating constants and variables

Swift can create constants and variables, but constants are generally preferable.

Use this to create then change a variable string:

```
var name = "Ted"
name = "Rebecca"
```

If you don't want to change a value, use a *constant* instead:

```
let user = "Daphne"
```

The **`print()`** function is helpful for learning and debugging, and shows some information about a variable:

```
print(user)
```

Strings

Swift's strings start and end with double quotes:

```
let actor = "Tom Cruise"
```

And they work great with emoji too:

```
let actor = "Tom Cruise 🏃"
```

If you want double quotes inside your string, place a backslash before them:

```
let quote = "He tapped a sign saying \"Believe\" and walked away."
```

If you want a string that spans multiple lines, start and end with three double quotes, like this:

```
let movie = """
A day in
the life of an
Apple engineer
"""
```

Swift provides many useful properties and methods for strings, including **`.count`** to read how many letters it has:

```
print(actor.count)
```

There are also `hasPrefix()` and `hasSuffix()`, which lets us know whether a string starts or ends with specific letters:

```
print(quote.hasPrefix("He"))
print(quote.hasSuffix("Away."))
```

Important: Strings are case-sensitive in Swift, so that second check will return false

Integers

Swift stores whole numbers using the type `Int`, which supports a range of standard mathematical operators:

```
let score = 10
let higherScore = score + 10
let halvedScore = score / 2
```

It also supports compound assignment operators that modify variables in place:

```
var counter = 10
counter += 5
```

Integers come with their own useful functionality, such as the `isMultiple(of:)` method:

```
let number = 120
print(number.isMultiple(of: 3))
```

You can also make random integers in a specific range, like this:

```
let id = Int.random(in: 1...1000)
```

Decimals

If you create a number with a decimal point, Swift will consider it a `Double`:

```
let score = 3.0
```

Swift considers `Double` to be a wholly different type of data to `Int`, and won't let you mix them together.

Booleans

Swift uses the type `Bool` to store true or false:

```
let goodDogs = true
let gameOver = false
```

You can flip a Boolean from true to false by calling its `toggle()` method:

```
var isSaved = false
isSaved.toggle()
```

Joining strings

You can create strings out of other data using *string interpolation*: write a backslash inside your string, then place the name of a variable or constant inside parentheses, like this:

```
let name = "Taylor"
let age = 26
let message = "I'm \(name) and I'm \(age) years old."
print(message)
```

When that code runs, it will print "I'm Taylor and I'm 26 years old."

Arrays

You can group items into an array like this:

```
var colors = ["Red", "Green", "Blue"]
let numbers = [4, 8, 15, 16]
var readings = [0.1, 0.5, 0.8]
```

Each of those hold different kinds of data: one strings, one integers, and one decimals. When we read data from arrays we will get the appropriate type back - a `String`, an `Int`, or a `Double`:

```
print(colors[0])
print(readings[2])
```

Tip: Make sure an item exists at the index you're asking for, otherwise your code will crash – your app will just stop working.

If your array is variable, you can use `append()` to add new items:

```
colors.append("Tartan")
```

The type of data you add must match whatever was already in there.

Arrays have useful functionality, such as `.count` to read how many items are in an array, or `remove(at:)` to remove one item at a specific index:

```
colors.remove(at: 0)
print(colors.count)
```

You can check whether an array contains a particular item by using `contains()`, like this:

```
print(colors.contains("Octarine"))
```

Dictionaries

Dictionaries store multiple values according to a key we specify. For example, we could create a dictionary to store information about a person:

```
let employee = [
    "name": "Taylor",
    "job": "Singer"
]
```

To read data from the dictionary, use the same keys you used when creating it:

```
print(employee["name", default: "Unknown"])
print(employee["job", default: "Unknown"])
```

The `default` value will be used if the key we've asked for doesn't exist.

Sets

Sets are similar to arrays, except you can't add duplicate items, and they don't store items in a particular order.

This makes a set of numbers:

```
var numbers = Set([1, 1, 3, 5, 7])
print(numbers)
```

Remember, the set will ignore duplicate values, and it won't remember the order used in the array.

Adding items to a set is done by calling its `insert()` method, like this:

```
numbers.insert(10)
```

Sets have one big advantage over arrays: using `contains()` on a set is effectively instant no matter how many items the set contains – even a set with 10,000,000 items will respond instantly.

Enums

An enum is a set of named values we can create and use to make our code more efficient and safer. For example, we could make an enum of weekdays like this:

```
enum Weekday {
    case monday, tuesday, wednesday, thursday, friday
}
```

That calls the new enum `Weekday`, and provides five cases to handle the five weekdays.

We can now make instances of that enum, then assign other possible cases to it:

```
var day = Weekday.monday
day = .friday
```

Type annotations

You can try to force a specific type for a new variable or constant by using *type annotation* like this:

```
var score: Double = 0
```

Without the `: Double` part Swift would infer that to be an `Int`, but we're overriding that and saying it's a `Double`.

Here are some type annotations based on the types covered so far:

```
let player: String = "Roy"
var luckyNumber: Int = 13
let pi: Double = 3.141
var isEnabled: Bool = true
var albums: Array<String> = ["Red", "Fearless"]
var user: Dictionary<String, String> = ["id": "@twostraws"]
var books: Set<String> = Set(["The Bluest Eye", "Foundation"])
```

Arrays and dictionaries are so common that they have special syntax that is easier to write:

```
var albums: [String] = ["Red", "Fearless"]
var user: [String: String] = ["id": "@twostraws"]
```

Knowing the exact types of things is important for creating empty collections. For example, both of these create empty string arrays:

```
var teams: [String] = [String]()
var clues = [String]()
```

Values of an enum have the same type as the enum itself, so we could write this:

```
enum UIStyle {
    case light, dark, system
}

var style: UIStyle = .light
```

Conditions

Use **if**, **else**, and **else if** statements to check a condition and run some code as appropriate:

```
let age = 16

if age < 12 {
    print("You can't vote")
} else if age < 18 {
    print("You can vote soon.")
} else {
    print("You can vote now.")
}
```

We can use **&&** to combine two conditions, and the whole condition will only be true if the two parts inside are true:

```
let temp = 26

if temp > 20 && temp < 30 {
    print("It's a nice day.")
}
```

Alternatively, **||** will make a condition be true if *either* subcondition is true.

Switch statements

Swift lets us check a value against multiple conditions using **switch**/**case** syntax, like this:

```
enum Weather {
    case sun, rain, wind
}

let forecast = Weather.sun

switch forecast {
case .sun:
    print("A nice day.")
case .rain:
    print("Pack an umbrella.")
default:
    print("Should be okay.")
}
```

switch statements *must* be exhaustive: all possible values must be handled so you can't miss one by accident.

The ternary conditional operator

The ternary operator lets us check a condition and return one of two values: something if the condition is true, and something if it's false:

```
let age = 18
let canVote = age >= 18 ? "Yes" : "No"
```

When that code runs, **canVote** will be set to "Yes" because **age** is set to 18.

Loops

Swift's **for** loops run some code for every item in a collection, or in a custom range. For example:

```
let platforms = ["iOS", "macOS", "tvOS", "watchOS"]

for os in platforms {
    print("Swift works on \(os).")
}
```

You can also loop over a range of numbers:

```
for i in 1...12 {
    print("5 x \(i) is \(5 * i)")
}
```

1...12 contains the values 1 through 12 inclusive. If you want to exclude the final number, use **..<** instead:

```
for i in 1..<13 {
    print("5 x \(i) is \(5 * i)")
}
```

Tip: If you don't need the loop variable, use **_**:

```
var lyric = "Haters gonna"

for _ in 1...5 {
    lyric += " hate"
}

print(lyric)
```

There are also **while** loops, which execute their loop body until a condition is false, like this:

```
var count = 10

while count > 0 {
    print("\(count)…")
    count -= 1
}

print("Go!")
```

You can use **continue** to skip the current loop iteration and proceed to the following one:

```
let files = ["me.jpg", "work.txt", "sophie.jpg"]

for file in files {
    if file.hasSuffix(".jpg") == false {
        continue
    }

    print("Found picture: \(file)")
}
```

Alternatively, use **break** to exit a loop and skip all remaining iterations.

Functions

To create a new function, write **func** followed by your function's name, then add parameters inside parentheses:

```
func printTimesTables(number: Int) {
    for i in 1...12 {
        print("\(i) x \(number) is \(i * number)")
    }
}

printTimesTables(number: 5)
```

We need to write **number: 5** at the call site, because the parameter name is part of the function call.

To return data from a function, tell Swift what type it is, then use the **return** keyword to send it back. For example, this returns a dice roll:

```
func rollDice() -> Int {
    return Int.random(in: 1...6)
}

let result = rollDice()
print(result)
```

If your function contains only a single line of code, you can remove the `return` keyword:

```
func rollDice() -> Int {
    Int.random(in: 1...6)
}
```

Returning multiple values from functions

Tuples store a fixed number of values of specific types, which is a convenient way to return multiple values from a function:

```
func getUser() -> (firstName: String, lastName: String) {
    (firstName: "Taylor", lastName: "Swift")
}

let user = getUser()
print("Name: \(user.firstName) \(user.lastName)")
```

If you don't need all the values from the tuple you can destructure the tuple to pull it apart into individual values, then `_` to tell Swift to ignore some:

```
let (firstName, _) = getUser()
print("Name: \(firstName)")
```

Customizing parameter labels

If you don't want to pass a parameter's name when calling a function, place an underscore before it:

```
func isUppercase(_ string: String) -> Bool {
    string == string.uppercased()
}

let string = "HELLO, WORLD"
let result = isUppercase(string)
```

An alternative is to write a second name before the first: one to use externally, and one internally:

```
func printTimesTables(for number: Int) {
    for i in 1...12 {
        print("\(i) x \(number) is \(i * number)")
    }
}

printTimesTables(for: 5)
```

In that code `for` is used externally, and `number` is used internally.

Providing default values for parameters

We can provide default parameter values by writing an equals after the type then providing a value, like this:

```
func greet(_ person: String, formal: Bool = false) {
    if formal {
        print("Welcome, \(person)!")
    } else {
        print("Hi, \(person)!")
    }
}
```

Now we can call `greet()` in two ways:

```
greet("Tim", formal: true)
greet("Taylor")
```

Handling errors in functions

To handle errors in functions you need to tell Swift which errors can happen, write a function that can throw errors, then call it and handle any problems.

First, define the errors that can occur:

```
enum PasswordError: Error {
    case short, obvious
}
```

Next, write a function that can throw errors. This is done by placing `throws` into the function's type, then by using `throw` to trigger specific errors:

```
func checkPassword(_ password: String) throws -> String {
    if password.count < 5 {
        throw PasswordError.short
    }

    if password == "12345" {
        throw PasswordError.obvious
    }

    if password.count < 10 {
        return "OK"
    } else {
        return "Good"
    }
}
```

Now call the throwing function by starting a **do** block, calling the function using **try**, then catching errors that occur:

```
let string = "12345"

do {
    let result = try checkPassword(string)
    print("Rating: \(result)")
} catch PasswordError.obvious {
    print("I have the same combination on my luggage!")
} catch {
    print("There was an error.")
}
```

When it comes to catching errors, you must always have a **catch** block that can handle every kind of error.

Closures

You can assign functionality directly to a constant or variable like this:

```
let sayHello = {
    print("Hi there!")
}

sayHello()
```

In that code, **sayHello** is a closure – a chunk of code we can pass around and call whenever we want. If you want the closure to accept parameters, they must be written inside the braces:

```
let sayHello = { (name: String) -> String in
    "Hi \(name)!"
}
```

The **in** is used to mark the end of the parameters and return type – everything after that is the body of the closure itself.

Closures are used extensively in Swift. For example, there's an array method called **filter()** that runs all elements of the array through a test, and any that return true for the test get returned in a new array.

We can provide that test using a closure, so we could filter an array to include only names that begin with T:

```
let team = ["Gloria", "Suzanne", "Tiffany", "Tasha"]

let onlyT = team.filter({ (name: String) -> Bool in
    return name.hasPrefix("T")
})
```

Inside the closure we list the parameter **filter()** passes us, which is a string from the array. We also say that our closure returns a Boolean, then mark the start of the closure's code by using **in** – after that, everything else is normal function code.

Trailing closures and shorthand syntax

Swift has a few tricks up its sleeve to make closures easier to read. Here's some code that filters an array to include only names beginning with "T":

```
let team = ["Gloria", "Suzanne", "Tiffany", "Tasha"]

let onlyT = team.filter({ (name: String) -> Bool in
    return name.hasPrefix("T")
})

print(onlyT)
```

Immediately you can see that the body of the closure has just a single line of code, so we can remove **`return`**:

```
let onlyT = team.filter({ (name: String) -> Bool in
    name.hasPrefix("T")
})
```

`filter()` must be given a function that accepts one item from its array, and returns true if it should be in the returned array.

Because the function we pass in *must* behave like that, we don't need to specify the types in our closure. So, we can rewrite the code to this:

```
let onlyT = team.filter({ name in
```

We can go further using special syntax called *trailing closure syntax*, which looks like this:

```
let onlyT = team.filter { name in
    name.hasPrefix("T")
}
```

Finally, Swift can provide short parameter names for us so we don't even write **`name in`** any more, and instead rely on a specially named value provided for us: **`$0`**:

```
let onlyT = team.filter {
    $0.hasPrefix("T")
}
```

Structs

Structs let us create our own custom data types, complete with their own properties and methods:

```
struct Album {
    let title: String
    let artist: String
    var isReleased = true

    func printSummary() {
        print("\(title) by \(artist)")
    }
}

let red = Album(title: "Red", artist: "Taylor Swift")
print(red.title)
red.printSummary()
```

When we create instances of structs we do so using an *initializer* – Swift lets us treat our struct like a function, passing in parameters for each of its properties. It silently generates this *memberwise initializer* based on the struct's properties.

If you want to have a struct's method change one of its properties, mark it as *mutating*:

```
mutating func removeFromSale() {
    isReleased = false
}
```

Computed properties

A computed property calculates its value every time it's accessed. For example, we could write an `Employee` struct that tracks how many days of vacation remained for that employee:

```
struct Employee {
    let name: String
    var vacationAllocated = 14
    var vacationTaken = 0

    var vacationRemaining: Int {
        vacationAllocated - vacationTaken
    }
}
```

To be able to write to `vacationRemaining` we need to provide both a *getter* and a *setter*:

```
var vacationRemaining: Int {
    get {
        vacationAllocated - vacationTaken
    }

    set {
        vacationAllocated = vacationTaken + newValue
    }
}
```

`newValue` is provided by Swift, and stores whatever value the user was assigning to the property.

Property observers

Property observers are pieces of code that run when properties change: `didSet` runs when the property just changed, and `willSet` runs *before* the property changed.

We could demonstrate `didSet` by making a `Game` struct print a message when the score changes:

```
struct Game {
    var score = 0 {
        didSet {
            print("Score is now \(score)")
        }
    }
}

var game = Game()
game.score += 10
game.score -= 3
```

Custom initializers

Initializers are special functions that prepare a new struct instance to be used, ensuring all properties have an initial value.

Swift generates one based on the struct's properties, but you can create your own:

```
struct Player {
    let name: String
    let number: Int

    init(name: String) {
        self.name = name
        number = Int.random(in: 1...99)
    }
}
```

Important: Initializers don't have `func` before them, and don't explicitly return a value.

Access control

Swift has several options for access control inside structs, but four are the most common:

- Use `private` for "don't let anything outside the struct use this."
- Use `private(set)` for "anything outside the struct can read this, but don't let them change it."
- Use `fileprivate` for "don't let anything outside the current file use this."
- Use `public` for "let anyone, anywhere use this."

For example:

```
struct BankAccount {
    private(set) var funds = 0

    mutating func deposit(amount: Int) {
        funds += amount
    }

    mutating func withdraw(amount: Int) -> Bool {
        if funds > amount {
            funds -= amount
            return true
        } else {
            return false
        }
    }
}
```

Because we used `private(set)`, reading `funds` from outside the struct is fine but writing isn't possible.

Static properties and methods

Swift supports static properties and methods, allowing you to add a property or method directly to the struct itself rather than to one instance of the struct:

```
struct AppData {
    static let version = "1.3 beta 2"
    static let settings = "settings.json"
}
```

Using this approach, everywhere we need to check or display something like the app's version number we can read `AppData.version`.

Classes

Classes let us create custom data types, and are different from structs in five ways.

The first difference is that we can create classes by inheriting functionality from other classes:

```
class Employee {
    let hours: Int

    init(hours: Int) {
        self.hours = hours
    }

    func printSummary() {
        print("I work \(hours) hours a day.")
    }
}

class Developer: Employee {
    func work() {
        print("I'm coding for \(hours) hours.")
    }
}

let novall = Developer(hours: 8)
novall.work()
novall.printSummary()
```

If a child class wants to change a method from a parent class, it must use **override**:

```
override func printSummary() {
    print("I spend \(hours) hours a day searching Stack Overflow.")
}
```

The second difference is that initializers are more tricky with classes. There's a lot of complexity here, but there are three key points:

1. Swift won't generate a memberwise initializer for classes.
2. If a child class has custom initializers, it must always call the *parent's* initializer after it has finished setting up its own properties.
3. If a subclass *doesn't* have any initializers, it automatically inherits the initializers of its parent class.

For example:

```
class Vehicle {
    let isElectric: Bool

    init(isElectric: Bool) {
        self.isElectric = isElectric
    }
}

class Car: Vehicle {
    let isConvertible: Bool

    init(isElectric: Bool, isConvertible: Bool) {
        self.isConvertible = isConvertible
        super.init(isElectric: isElectric)
    }
}
```

super allows us to call up to methods that belong to our parent class, such as its initializer.

The third difference is that all copies of a class instance share their data, meaning that changes you make to one will automatically change other copies.

For example:

```
class Singer {
    var name = "Adele"
}

var singer1 = Singer()
var singer2 = singer1
singer2.name = "Justin"
print(singer1.name)
print(singer2.name)
```

That will print "Justin" for both – even though we only changed one of them, the other also changed. In comparison, struct copies *don't* share their data.

The fourth difference is that classes can have a *deinitializer* that gets called when the last reference to an object is destroyed.

So, we could create a class that prints a message when it's created and destroyed:

```
class User {
    let id: Int

    init(id: Int) {
        self.id = id
        print("User \(id): I'm alive!")
    }

    deinit {
        print("User \(id): I'm dead!")
    }
}

for i in 1...3 {
    let user = User(id: i)
    print("User \(user.id): I'm in control!")
}
```

The final difference is that classes let us change variable properties even when the class itself is constant:

```
class User {
    var name = "Paul"
}

let user = User()
user.name = "Taylor"
print(user.name)
```

As a result of this, classes don't need the **`mutating`** keyword with methods that change their data.

Protocols

Protocols define functionality we expect a data type to support, and Swift ensures our code follows those rules.

For example, we could define a **`Vehicle`** protocol like this:

```
protocol Vehicle {
    func estimateTime(for distance: Int) -> Int
    func travel(distance: Int)
}
```

That lists the required methods for this protocol to work, but doesn't contain any code – we're specifying only method names, parameters, and return types.

Once you have a protocol you can make data types conform to it by implementing the required functionality. For example, we could make a **`Car`** struct that conforms to **`Vehicle`**:

```
struct Car: Vehicle {
    func estimateTime(for distance: Int) -> Int {
        distance / 50
    }

    func travel(distance: Int) {
        print("I'm driving \(distance)km.")
    }
}
```

All the methods listed in `Vehicle` must exist *exactly* in `Car`, with the same name, parameters, and return types.

Now you can write a function that accepts any kind of type that conforms to `Vehicle`, because Swift knows it implements both `estimateTime()` and `travel()`:

```
func commute(distance: Int, using vehicle: Vehicle) {
    if vehicle.estimateTime(for: distance) > 100 {
        print("Too slow!")
    } else {
        vehicle.travel(distance: distance)
    }
}

let car = Car()
commute(distance: 100, using: car)
```

Protocols can also require properties, so we could require properties for how many seats vehicles have and how many passengers they currently have:

```
protocol Vehicle {
    var name: String { get }
    var currentPassengers: Int { get set }
    func estimateTime(for distance: Int) -> Int
    func travel(distance: Int)
}
```

That adds two properties: one marked with `get` that might be a constant or computed property, and one marked with `get set` that might be a variable or a computed property with a getter and setter.

Now all conforming types must add implementations of those two properties, like this for `Car`:

```
let name = "Car"
var currentPassengers = 1
```

Tip: You can conform to as many protocols as you need, just by listing them separated with a comma.

Extensions

Extensions let us add functionality to any type. For example, Swift's strings have a method for trimming whitespace and new lines, but it's quite long so we could turn it into an extension:

```
extension String {
    func trimmed() -> String {
        self.trimmingCharacters(in: .whitespacesAndNewlines)
    }
}

var quote = "   The truth is rarely pure and never simple   "
let trimmed = quote.trimmed()
```

If you want to change a value directly rather than returning a new value, mark your method as `mutating` like this:

```
extension String {
    mutating func trim() {
        self = self.trimmed()
    }
}

quote.trim()
```

Extensions can also add computed properties to types, like this one:

```
extension String {
    var lines: [String] {
        self.components(separatedBy: .newlines)
    }
}
```

The `components(separatedBy:)` method splits a string into an array of strings using a boundary of our choosing, which in this case is new lines.

We can now use that property with all strings:

```
let lyrics = """
But I keep cruising
Can't stop, won't stop moving
"""

print(lyrics.lines.count)
```

Protocol extensions

Protocol extensions extend a whole protocol to add computed properties and method implementations, so any types conforming to that protocol get them.

For example, `Array`, `Dictionary`, and `Set` all conform to the `Collection` protocol, so we can add a computed property to all three of them like this:

```
extension Collection {
    var isNotEmpty: Bool {
        isEmpty == false
    }
}
```

Now we can put it to use:

```
let guests = ["Mario", "Luigi", "Peach"]

if guests.isNotEmpty {
    print("Guest count: \(guests.count)")
}
```

This approach means we can list required methods in a protocol, then add default implementations of those inside a protocol extension. All conforming types then get to use those default implementations, or provide their own as needed.

Optionals

Optionals represent the absence of data – for example, they distinguish between an integer having the value 0, and having no value at all.

To see optionals in action, think about this code:

```
let opposites = [
    "Mario": "Wario",
    "Luigi": "Waluigi"
]

let peachOpposite = opposites["Peach"]
```

That attempts to read the value attached to the key "Peach", which doesn't exist, so this can't be a regular string. Swift's solution is called *optionals*, which means data that might be present or might not.

An optional string might have a string waiting inside for us, or there might be nothing at all – a special value called **`nil`**, that means "no value". Any kind of data can be optional, including **`Int`**, **`Double`**, and **`Bool`**, as well as instances of enums, structs, and classes.

Swift won't let us use optional data directly, because it might be empty. That means we need to *unwrap* the optional to use it – we need to look inside to see if there's a value, and, if there is, take it out and use it.

Swift gives us several ways of unwrapping optionals, but the one you'll see most looks like this:

```
if let marioOpposite = opposites["Mario"] {
    print("Mario's opposite is \(marioOpposite)")
}
```

That reads the optional value from the dictionary, and if it has a string inside it gets *unwrapped* – the string inside gets placed into the **`marioOpposite`** constant, and isn't optional any more. Because we were able to unwrap the optional, the condition is a success so the **`print()`** code is run.

Unwrapping optionals with guard

Swift has a second way of unwrapping optionals, called **`guard let`**, which is very similar to **`if let`** except it flips things around: **`if let`** runs the code inside its braces if the optional had a value, and **`guard let`** runs the code if the optional *didn't* have a value.

It looks like this:

```
func printSquare(of number: Int?) {
    guard let number = number else {
        print("Missing input")
        return
    }

    print("\(number) x \(number) is \(number * number)")
}
```

If you use **`guard`** to check a function's inputs are valid, Swift requires you to use **`return`** if the check fails. However, if the optional you're unwrapping has a value inside, you can use it *after* the **`guard`** code finishes.

Tip: You can use guard with any condition, including ones that don't unwrap optionals.

Nil coalescing

Swift has a third way of unwrapping optionals, called the *nil coalescing operator* – it unwraps an optional and provides a default value if the optional was empty:

```
let tvShows = ["Archer", "Babylon 5", "Ted Lasso"]
let favorite = tvShows.randomElement() ?? "None"
```

The nil coalescing operator is useful in many places optionals are created. For example, creating an integer from a string returns an optional **`Int?`** because the conversion might have failed. Here we can use nil coalescing to provide a default value:

```
let input = ""
let number = Int(input) ?? 0
print(number)
```

Optional chaining

Optional chaining reads optionals inside optionals, like this:

```
let names = ["Arya", "Bran", "Robb", "Sansa"]
let chosen = names.randomElement()?.uppercased()
print("Next in line: \(chosen ?? "No one")")
```

Optional chaining is there on line 2: a question mark followed by more code. It allows us to say "if the optional has a value inside, unwrap it then…" and add more code. In our case we're saying "if we got a random element from the array, uppercase it."

Optional try?

When calling a function that might throw errors, we can use `try?` to convert its result into an optional containing a value on success, or `nil` otherwise.

Here's how it looks:

```
enum UserError: Error {
    case badID, networkFailed
}

func getUser(id: Int) throws -> String {
    throw UserError.networkFailed
}

if let user = try? getUser(id: 23) {
    print("User: \(user)")
}
```

The `getUser()` function will always throw `networkFailed`, but we don't care *what* was thrown – all we care about is whether the call sent back a user or not.

Wrap up

We've covered the majority of Swift language fundamentals here, but really we've only scratched the surface of what the language does. Fortunately, with what you've learned you already know enough to build some fantastic software with Swift and SwiftUI.