SAMPLE CHAPTER

Swift IN DEPTH

Tjeerd in 't Veen

MANNING

Swift in Depth

by Tjeerd in 't Veen

Chapter 2

brief contents

2 Modeling data with enums

This chapter covers

- How enums are an alternative to subclassing
- Using enums for polymorphism
- Learning how enums are "or" types
- Modeling data with enums instead of structs
- How enums and structs are algebraic types
- Converting structs to enums
- Safely handling enums with raw values
- Converting strings to enums to create robust code

Enumerations, or enums for short, are a core tool used by Swift developers. Enums allow you to define a type by *enumerating* over its values, such as whether an HTTP method is a *get*, *put*, *post*, or *delete* action, or denoting if an IP-address is either in *IPv4* or *IPv6* format.

Many languages have an implementation of enums, with a different type of implementation per language. Enums in Swift, unlike in C and Objective-C, aren't *only* representations of integer values. Instead, Swift borrows many concepts from

the functional programming world, which bring plenty of benefits that you'll explore in this chapter.

In fact, I would argue that enums are a little underused in Swift-land. I hope to change that and help you see how enums can be surprisingly useful in many ways. My goal is to expand your enum-vocabulary so that you can directly use these techniques in your projects.

First, you'll see multiple ways to model your data with enums and how they fare against structs and classes.

Enums are a way to offer polymorphism, meaning that you can work with a single type, representing more types. We shed some light on how we can store multiple types into a single collection, such as an array.

Then, you'll see how enums are a suitable alternative to subclassing.

We dive a little into some algebraic theory to understand enums on a deeper level; then you'll see how you can apply this theory and convert structs to enums and back again.

As a cherry on top, we explore raw value enums and how you can use them to handle strings cleanly.

After reading this chapter, you may find that you're modeling data better, writing enums just a bit more often, and ending up with safer and cleaner code in your projects.

2.1 Or vs. and

Enums can be thought of as an "or" type. Enums can only be one thing at once—for example, a traffic light can either be green *or* yellow *or* red. Alternatively, a die can either be six-sided *or* twenty-sided, but not both at the same time.

JOIN ME! All code from this chapter is online. It's more educational and fun if you follow along. You can download the source code at https://mng.bz/gNre.

2.1.1 Modeling data with a struct

Let's start off with an example that shows how to think about "or" and "and" types when modeling data.

In the upcoming example, you're modeling message data in a chat application. A message could be text that a user may send, but it could also be a join message or leave message. A message could even be a signal to send balloons across the screen (see figure 2.1). Because why not? Apple does it, too, in their Messages app.

Here are some types of messages that your application might support:

- Join messages, such as "Mother in law has joined the chat"
- Text messages that someone can write, such as "Hello everybody!"

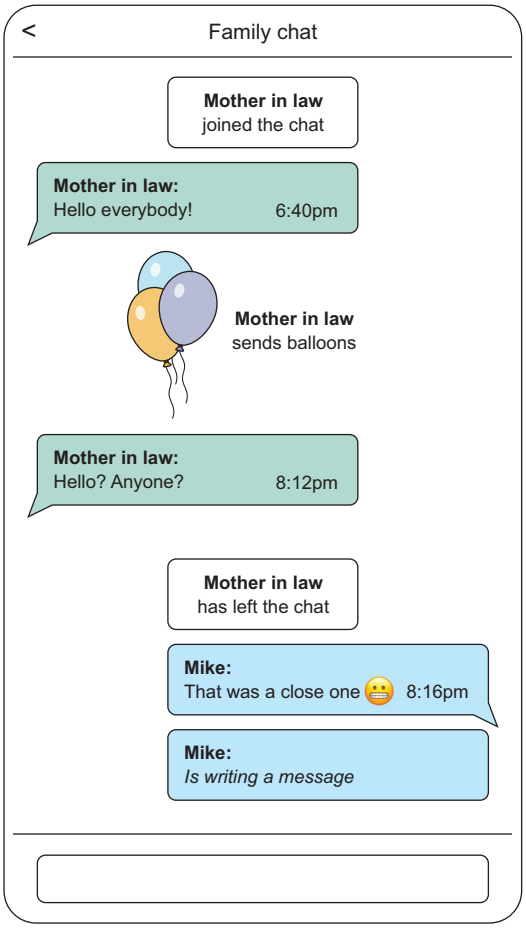

Figure 2.1 A chat application

- Send balloons messages, which include some animations and annoying sounds that others can see and hear
- Leave messages, such as "Mother in law has left the chat"
- Draft messages, such as "Mike is writing a message"

Let's create a data model to represent messages. Your first idea might be to use a struct to model your `Message`. You'll start by doing that and showcase the problems that come with it. Then you'll solve these problems by using an enum.

You can create multiple types of messages in code, such as a join message when someone enters a chatroom.

Listing 2.1 A join chatroom message

```
import Foundation // Needed for the Date type.

let joinMessage = Message(userId: "1",
                          contents: nil,
                          date: Date(),
                          hasJoined: true, // Set the joined Boolean
                          hasLeft: false,
                          isBeingDrafted: false,
                          isSendingBalloons: false)
```

You can also create a regular text message.

Listing 2.2 A text message

```
let textMessage = Message(userId: "2",
                          contents: "Hey everyone!", // Pass a message
                          date: Date(),
                          hasJoined: false,
                          hasLeft: false,
                          isBeingDrafted: false,
                          isSendingBalloons: false)
```

In your hypothetical messaging app, you can pass this message data around to other users.

The `Message` struct looks as follows.

Listing 2.3 The `Message` struct

```
import Foundation

struct Message {
    let userId: String
    let contents: String?
    let date: Date

    let hasJoined: Bool
    let hasLeft: Bool
```

```
    let isBeingDrafted: Bool
    let isSendingBalloons: Bool
}
```

Although this is one small example, it highlights a problem. Because a struct can contain multiple values, you can run into bugs where the `Message` struct can be a text message, a `hasLeft` command, and an `isSendingBalloons` command. An invalid message state doesn't bode well because a message can only be one *or* another in the business rules of the application. The visuals won't support an invalid message either.

To illustrate, you can have a message in an invalid state. It represents a text message, but also a join and a leave message.

Listing 2.4 An invalid message with conflicting properties

```
let brokenMessage = Message(userId: "1",
                            contents: "Hi there", // Have text to show
                            date: Date(),
                            hasJoined: true, // But this message also signals
     a joining state
                            hasLeft: true, // ... and a leaving state
                            isBeingDrafted: false,
                            isSendingBalloons: false)
```

In a small example, running into invalid data is harder, but it inevitably happens often enough in real-world projects. Imagine parsing a local file to a `Message`, or some function that combines two messages into one. You don't have any compile-time guarantees that a message is in the right state.

You can think about validating a `Message` and throwing errors, but then you're catching invalid messages at runtime (if at all). Instead, you can enforce correctness at compile time if you model the `Message` using an enum.

2.1.2 Turning a struct into an enum

Whenever you're modeling data, see if you can find *mutually exclusive* properties. A message can't be both a join and a leave message at the same time. A message can't also send balloons and be a draft at the same time.

But a message can be a join message *or* a leave message. A message can also be a draft, *or* it can represent the sending of balloons. When you detect "or" statements in a model, an enum could be a more fitting choice for your data model.

Using an enum to group the properties into cases makes the data much clearer to grasp.

Let's improve the model by turning it into an enum.

Listing 2.5 `Message` as an enum (lacking values)

```
import Foundation

enum Message {
    case text
```

```
    case draft
    case join
    case leave
    case balloon
}
```

But you're not done yet because the cases have no values. You can add values by adding a tuple to each case. A tuple is an ordered set of values, such as `(userId: String, contents: String, date: Date)`.

By combining an enum with tuples, you can build more complex data structures. Let's add tuples to the enum's cases now.

Listing 2.6 `Message` as an enum (with values)

```
import Foundation

enum Message {
    case text(userId: String, contents: String, date: Date)
    case draft(userId: String, date: Date)
    case join(userId: String, date: Date)
    case leave(userId: String, date: Date)
    case balloon(userId: String, date: Date)
}
```

By adding tuples to cases, these cases now have so-called *associated values* in Swift terms. Also, you can clearly see which properties belong together and which properties don't.

Whenever you want to create a `Message` as an enum, you can pick the proper case with related properties, without worrying about mixing and matching the wrong values.

Listing 2.7 Creating enum messages

```
let textMessage = Message.text(userId: "2", contents: "Bonjour!", date: Date())
let joinMessage = Message.join(userId: "2", date: Date())
```

When you want to work with the messages, you can use a `switch` case on them and unwrap its inner values.

Let's say that you want to log the sent messages.

Listing 2.8 Logging messages

```
logMessage(message: joinMessage) // User 2 has joined the chatroom
logMessage(message: textMessage) // User 2 sends message: Bonjour!

func logMessage(message: Message) {
    switch message {
    case let .text(userId: id, contents: contents, date: date):
        print("[\(date)] User \(id) sends message: \(contents)")
    case let .draft(userId: id, date: date):
        print("[\(date)] User \(id) is drafting a message")
```

```
    case let .join(userId: id, date: date):
        print("[\(date)] User \(id) has joined the chatroom")
    case let .leave(userId: id, date: date):
        print("[\(date)] User \(id) has left the chatroom")
    case let .balloon(userId: id, date: date):
        print("[\(date)] User \(id) is sending balloons")
    }
}
```

Having to switch on all cases in your entire application just to read a value from a single message may be a deterrent. You can save yourself some typing by using the `if case let` combination to match on a single type of `Message`.

Listing 2.9 Matching on a single case

```
if case let Message.text(userId: id, contents: contents, date: date) =
     textMessage {
    print("Received: \(contents)") // Received: Bonjour!
}
```

If you're not interested in specific properties when matching on an enum, you can match on these properties with an underscore, called a *wild card*, or as I like to call it, the "I don't care" operator.

Listing 2.10 Matching on a single case with the "I don't care" underscore

```
if case let Message.text(_, contents: contents, _) = textMessage {
    print("Received: \(contents)") // Received: Bonjour!
}
```

2.1.3 Deciding between structs and enums

Getting compiler benefits with enums is a significant benefit. But if you catch yourself pattern matching often on a single case, a struct might be a better approach.

Also, keep in mind that the associated values of an enum are containers without additional logic. You don't get free initializers of properties; with enums, you'd have to manually add these.

Next time you write a struct, try to group properties. Your data model might be a good candidate for an enum!

2.2 Enums for polymorphism

Sometimes you need some flexibility in the shape of *polymorphism*. Polymorphism means that a single function, method, array, dictionary—you name it—can work with different types.

If you mix types in an array, however, you end up with an array of type `[Any]` (as shown in the following listing), such as when you put a `Date`, `String`, and `Int` into one array.

Listing 2.11 Filling an array with multiple values

```
let arr: [Any] = [Date(), "Why was six afraid of seven?", "Because...", 789]
```

Arrays explicitly want to be filled with the same type. In Swift, what these mixed types have in common is that they are an `Any` type.

Handling `Any` types are often not ideal. Since you don't know what `Any` represents at compile time, you have to check against the `Any` type at runtime to see what it presents. For instance, you could match on any types via pattern matching, using a *switch* statement.

Listing 2.12 Matching on `Any` values at runtime

```
let arr: [Any] = [Date(), "Why was six afraid of seven?", "Because...", 789]

for element: Any in arr {
  // element is "Any" type
  switch element {
    case let stringValue as String: "received a string: \(stringValue)"
    case let intValue as Int: "received an Int: \(intValue)"
    case let dateValue as Date: "received a date: \(dateValue)"
    default: print("I am not interested in this value")
  }
}
```

You can still figure out what `Any` is at runtime. But you don't know what to expect when matching on an `Any` type; therefore, you must also implement a default case to catch the values in which you're not interested.

Working with `Any` types is sometimes needed when you can't know what something is at compile time, such as when you're receiving unknown data from a server. But if you know beforehand the types that you're dealing with, you can get compile-time safety by using an enum.

2.2.1 Compile-time polymorphism

Imagine that you'd like to store two different types in an array, such as a `Date` and a range of two dates of type `Range<Date>`.

> **WHAT ARE THESE <DATE> BRACKETS?** Range is a type that represents a lower and upper bound. The `<Date>` notation indicates that `Range` is storing a *generic type*, which you'll explore deeply in chapter 7.
>
> The `Range<Date>` notation tells you that you're working with a range of two `Date` types.

You can create a `DateType` representing either a single date *or* a range of dates. Then you can fill up an array of both a `Date` and `Range<Date>`, as shown next.

Listing 2.13 Adding multiple types to an array via an enum

```
let now = Date()
let hourFromNow = Date(timeIntervalSinceNow: 3600)

let dates: [DateType] = [
    DateType.singleDate(now),
    DateType.dateRange(now..<hourFromNow)
]
```

The enum itself merely contains two cases, each with its associated value.

Listing 2.14 Introducing a `DateType` enum

```
enum DateType {
  case singleDate(Date)
  case dateRange(Range<Date>)
}
```

The array itself consists only of `DateType` instances. In turn, each `DateType` harbors one of the multiple types (see figure 2.2).

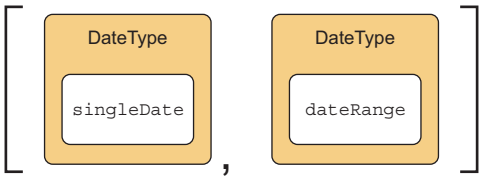

Figure 2.2 Array enums

Thanks to the enum, you end up with an array containing multiple types, while maintaining compile-time safety. If you were to read values from the array, you could switch on each value.

Listing 2.15 Matching on the `dateType` enum

```
for dateType in dates {
    switch dateType {
    case .singleDate(let date): print("Date is \(date)")
    case .dateRange(let range): print("Range is \(range)")
    }
}
```

The compiler also helps if you modify the enum. By way of illustration, if you add a year case to the enum, the compiler tells you that you forgot to handle a case.

Listing 2.16 Adding a year case to `DateType`

```
enum DateType {
  case singleDate(Date)
```

```
  case dateRange(Range<Date>)
  case year(Int)                    <-- Year is newly added.
}
```

The compiler is now throwing the following.

Listing 2.17 Compiler notifies you of an error

```
error: switch must be exhaustive
    switch dateType {
    ^

add missing case: '.year(_)'
    switch dateType {
```

Thanks to enums, you can bring back compile-time safety when mixing types inside arrays and other structures such as dictionaries.

Of course, you must know beforehand what kind of cases you expect. When you know what you're working with, the added compile-time safety is a nice bonus.

2.3 Enums instead of subclassing

Subclassing allows you to build a hierarchy of your data. For example, you could have a fast food restaurant selling burgers, fries, the usual. For that, you'd create a superclass of `FastFood`, with subclasses like `Burger`, `Fries`, and `Soda`.

One of the limitations of modeling your software with hierarchies (subclassing) is that doing so constrains you in a specific direction that won't always match your needs.

For example, the aforementioned restaurant has been getting complaints from customers wanting authentic Japanese sushi with their fries. They intend to accommodate the customers, but their subclassing model doesn't fit this new requirement.

In an ideal world, modeling your data hierarchically makes sense. But in practice, you'll sometimes hit edge cases and exceptions that may not fit your model.

In this section, we explore these limitations of modeling your data via subclassing in more of a real-world scenario and solve these with the help of enums.

2.3.1 Forming a model for a workout app

Next up, you're building a model layer for a workout app, which tracks running and cycling sessions for someone. A workout includes the start time, end time, and a distance.

You'll create both a `Run` and a `Cycle` struct that represent the data you're modeling.

Listing 2.18 The `Run` struct

```
import Foundation                   <-- Need Foundation for the Date type

struct Run {
    let id: String
    let startTime: Date
```

```
    let endTime: Date
    let distance: Float
    let onRunningTrack: Bool
}
```

Listing 2.19 The `Cycle` struct

```
struct Cycle {

    enum CycleType {
        case regular
        case mountainBike
        case racetrack
    }

    let id: String
    let startTime: Date
    let endTime: Date
    let distance: Float
    let incline: Int
    let type: CycleType
}
```

These structs are a good starting point for your data layer.

Admittedly, having to create separate logic in your application for both the `Run` and `Cycle` types can be cumbersome. Let's try to solve this via subclassing. Then you'll quickly learn which problems accompany subclassing, after which you'll see how enums can solve some of these problems.

2.3.2 Creating a superclass

Many similarities exist between `Run` and `Cycle`, which at first look make a good candidate for a superclass. The benefit of a superclass is that you can pass the superclass around, such as in your methods and arrays. A superclass saves you from creating specific methods and arrays for each workout subclass.

You could create a superclass called `Workout`; then you can turn `Run` and `Cycle` into classes and make them subclass `Workout`, which inherits from `Workout` (see figure 2.3).

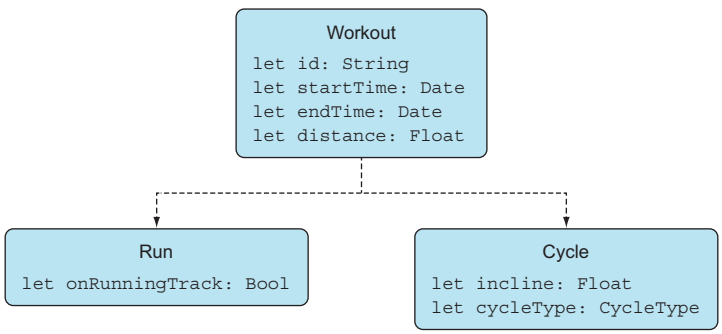

Figure 2.3 A subclassing hierarchy

Hierarchically, the subclassing structure makes a lot of sense because workouts share so many values.

The new `Workout` superclass contains the properties that both `Run` and `Cycle` share, specifically `id`, `startTime`, `endTime`, and `distance`.

2.3.3 The downsides of subclassing

Here we quickly touch upon issues related to subclassing. First of all, you're forced to use classes. Classes can be favorable, but having the choice between classes, structs, or other enums disappears when you use subclassing.

Being forced to use classes, however, isn't the biggest problem. Let's showcase another limitation by adding a new type of workout, called `Pushups`, which stores multiple repetitions and a single date.

Listing 2.20 The `Pushups` class

```
class Pushups: Workout {
    let repetitions: [Int]
    let date: Date
}
```

Pushups subclasses Workout

Subclassing `Workout` doesn't work properly because some properties of `Workout` don't apply to `Pushups`. `Workout` requires a `startTime`, `endTime`, and `distance` value, none of which `Pushups` needs.

To allow `Pushups` to subclass `Workout`, you'd have to refactor the superclass and all its subclasses. You would do this by moving `startTime`, `endTime`, and `distance` from `Workout` to the `Cycle` and `Run` classes because these properties aren't part of a `Pushups` class (see figure 2.4).

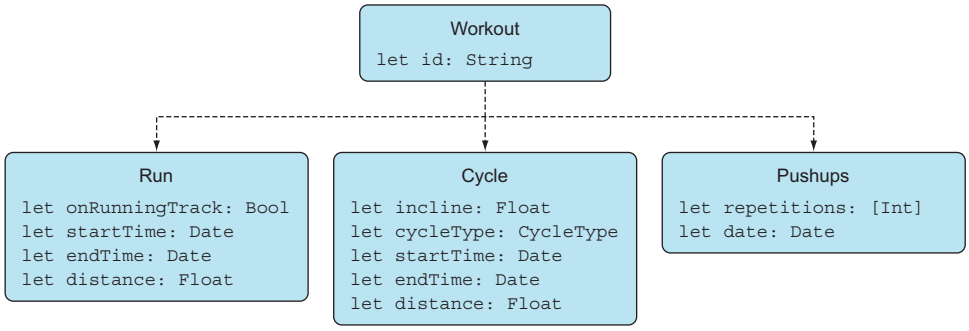

Figure 2.4 A refactored subclassing hierarchy

Refactoring an entire data model shows the issue when subclassing. As soon as you introduce a new subclass, you risk needing to refactor the superclass and all its subclasses, which is a significant impact on existing architecture.

Let's consider another approach involving enums.

2.3.4 Refactoring a data model with enums

By using enums, you stay away from a hierarchical structure, yet you can still keep the option of passing a single `Workout` around in your application. You'll also be able to add new workouts without needing to refactor the existing workouts.

You do this by creating a `Workout` enum instead of a superclass. You can contain different workouts inside the `Workout` enum.

Listing 2.21 `Workout` as an enum

```
enum Workout {
    case run(Run)
    case cycle(Cycle)
    case pushups(Pushups)
}
```

Now `Run`, `Cycle`, and `Pushups` won't subclass `Workout` anymore. In fact, all the workouts can be any type, such as a struct, class, or even another enum.

You can create a `Workout` by passing it a `Run`, `Cycle`, or `Pushups` workout. For example, you can convert `Pushups` to a struct, initialize it, and pass it to the `pushups` case inside the `Workout` enum.

Listing 2.22 Creating a workout

```
let pushups = Pushups(repetitions: [22,20,10], date: Date())
let workout = Workout.pushups(pushups)
```

Now you can pass a `Workout` around in your application. Whenever you want to extract the workout, you can pattern match on it.

Listing 2.23 Pattern matching on a workout

```
switch workout {
case .run(let run):
    print("Run: \(run)")
case .cycle(let cycle):
    print("Cycle: \(cycle)")
case .pushups(let pushups):
    print("Pushups: \(pushups)")
}
```

The benefit of this solution is that you can add new workouts without refactoring existing ones. For example, if you introduce an `Abs` workout, you can add it to `Workout` without touching `Run`, `Cycle`, or `Pushups`.

Listing 2.24 Adding a new workout to the `Workout` enum

```
enum Workout {
    case run(Run)
    case cycle(Cycle)
```

```
    case pushups(Pushups)
    case abs(Abs)                    <-- New workout is introduced
}
```

Not having to refactor other workouts to add a new one is a significant benefit and worth considering using enums over subclassing.

2.3.5 Deciding on subclassing or enums

Trying to determine when enums or subclasses fit your data model isn't always easy.

When types share many properties, and you predict that won't change in the future, you can get very far with classic subclassing. But subclassing steers you into a more rigid hierarchy. On top of that, you're forced to use classes.

When similar types start to diverge, or if you want to keep using enums and structs (as opposed to classes only), creating an encompassing enum offers more flexibility and could be the better choice.

The downside of enums is that now your code needs to match all cases in your entire application. Although this may require extra work when adding new cases, it also is a safety net where the compiler makes sure you haven't forgotten to handle a case somewhere in your application.

Another downside of enums is that at the time of writing, enums can't be extended with new cases. Enums lock down a model to a fixed number of cases, and unless you own the code, you can't change this rigid structure. For example, perhaps you're offering an enum via a third-party library, and now its implementers can't expand on it.

These are trade-offs you'll have to make. If you can lock down your data model to a fixed, manageable number of cases, enums can be a good choice.

2.3.6 Exercises

1. Can you name two benefits of using subclassing instead of enums with associated types?
2. Can you name two benefits of using enums with associated types instead of subclassing?

2.4 Algebraic data types

Enums are based on something called *algebraic data types*, which is a term that comes from functional programming. Algebraic data types commonly express composed data via something called sum types and product types.

Enums are *sum types*; an enum can be only one thing at once, hence the "or" way of thinking covered earlier.

On the other end of the spectrum are *product types*, types that contains multiple values, such as a tuple or struct. You can think of a product type as an "and" type—for example, a `User` struct can have both a name *and* an id. Alternatively, an address class can have a street *and* a house number *and* a zip code.

Let's use this section to cover a bit of theory so that you can reason about enums better. Then we move on to some practical examples where you'll turn an enum into a struct and vice versa.

2.4.1 Sum types

Enums are sum types, which have a fixed number of values they can represent. For instance, the following enum called `Day` represents any day in the week. There are seven possible values that `Day` can represent.

Listing 2.25 The `Day` enum

```
enum Day {
    case sunday
    case monday
    case tuesday
    case wednesday
    case thursday
    case friday
    case saturday
}
```

To know the number of possible values of an enum, you *add* (sum) the possible values of the types inside. In the case of the `Day` enum, the total sum is seven.

Another way to reason about possible values is the `UInt8` type. Ranging from 0 to 255, the total number of possible values is 256. It isn't modeled this way, but you can think of an `UInt8` as if it's an enum with 256 cases.

If you were to write an enum with two cases, and you added an `UInt8` to one of the cases, this enum's possible variations jump from 2 to 257.

For instance, you can have an `Age` enum—representing someone's age—where the age can be unknown, but if it is known, it contains an `UInt8`.

Listing 2.26 The `Age` enum

```
enum Age {
  case known(UInt8)
  case unknown
}
```

`Age` now represents 257 possible values, namely, the unknown case(1) + known case(256).

2.4.2 Product types

On the other end of the spectrum are product types. A product type multiplies the possible values it contains. As an example, if you were to store two Booleans inside a struct, the total number of variations is the product (multiplication) of these two enums.

Listing 2.27 A struct containing two Booleans

```
struct BooleanContainer {
  let first: Bool
  let second: Bool
}
```

The first Boolean (two possible values) *times* the second Boolean (two possible values) is four possible states that this struct may have.

In code, you can prove this by revealing all the variations.

Listing 2.28 `BooleanContainer` has four possible variations

```
BooleanContainer(first: true, second: true)
BooleanContainer(first: true, second: false)
BooleanContainer(first: false, second: true)
BooleanContainer(first: false, second: false)
```

When you're modeling data, the number of variations is good to keep in mind. The higher the number of possible values a type has, the harder it is to reason about a type's possible states.

As hyperbole, having a struct with 1,000 strings for properties has a lot more possible states than a struct with a single Boolean property.

2.4.3 Distributing a sum over an enum

I won't focus only on theory regarding sum and product types, either. You're not here to write a dry, theoretically based graduate paper, but to produce beautiful work.

Imagine that you have a `PaymentType` enum containing three cases, which represent the three ways a customer can pay.

Listing 2.29 Introducing `PaymentType`

```
enum PaymentType {
  case invoice
  case creditcard
  case cash
}
```

Next, you're going to represent the status of a payment. A struct is a suitable candidate to store some auxiliary properties besides the `PaymentType` enum, such as when a payment is completed and whether or not it concerns a recurring payment.

Listing 2.30 A `PaymentStatus` struct

```
struct PaymentStatus {
  let paymentDate: Date?
  let isRecurring: Bool
  let paymentType: PaymentType
}
```

The product of all the variations would be all possible dates *times* 2 (Boolean) *times* 3 (enum with three cases). You'd have a high number of variations because the struct can store many date variations.

Like cream cheese on a bagel, you're smearing the properties of the struct out over the cases of the enum by following the rules of algebraic data types (see figure 2.5).

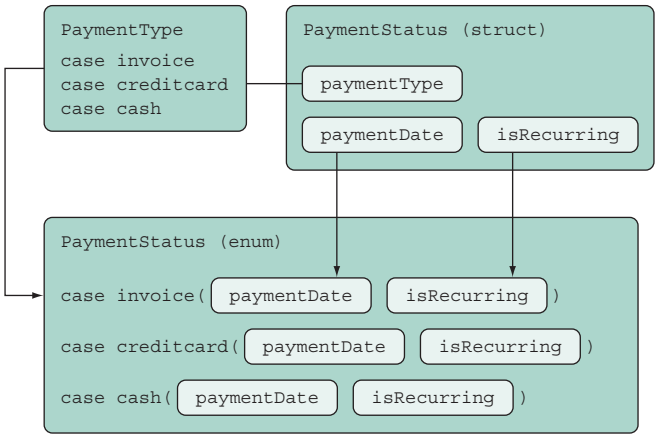

Figure 2.5 Turning a struct into an enum

You end up with an enum taking the same name as the struct. Each case represents the original enum's cases with the struct's properties inside.

Listing 2.31 `PaymentStatus` containing cases

```
enum PaymentStatus {
  case invoice(paymentDate: Date?, isRecurring: Bool)
  case creditcard(paymentDate: Date?, isRecurring: Bool)
  case cash(paymentDate: Date?, isRecurring: Bool)
}
```

All the information is still there, and the number of possible variations is still the same. Except this time you flipped the types inside out!

As a benefit, you're only dealing with a single type; the price is that you have some repetition inside each case. There's no right or wrong; it is merely a different approach to model the same data while leaving the same number of possible variations intact. It's a neat trick that displays the algebraic nature of types and helps you model enums in multiple ways. Depending on your needs, an enum might be a fitting alternative to a struct containing an enum, or vice versa.

2.4.4 Exercise

Given this data structure

```
enum Topping {
    case creamCheese
    case peanutButter
    case jam
}

enum BagelType {
    case cinnamonRaisin
    case glutenFree
    case oatMeal
    case blueberry
}

struct Bagel {
    let topping: Topping
    let type: BagelType
}
```

3. What is the number of possible variations of `Bagel`?
4. Turn `Bagel` into an enum while keeping the same amount of possible variations.
5. Given the following enum representing a puzzle game for a specific age range (such as baby, toddler, or teenager) and containing some puzzle pieces

```
enum Puzzle {
    case baby(numberOfPieces: Int)
    case toddler(numberOfPieces: Int)
    case preschooler(numberOfPieces: Int)
    case gradeschooler(numberOfPieces: Int)
    case teenager(numberOfPieces: Int)
}
```

How would this enum be represented as a struct instead?

2.5 A safer use of strings

Dealing with strings and enums is quite common. Let's go ahead and pay some extra attention to them so that you'll do it correctly. This section highlights some dangers when dealing with enums that hold a `String` raw value.

When an enum is defined as a raw value type, all cases of that enum carry some value inside them.

Enums with raw values are defined by having a type added to an enum's declaration.

Listing 2.32 Enums with raw values and string values

```
enum Currency: String {
     case euro = "euro"
     case usd = "usd"
     case gbp = "gbp"
}
```

String is the raw value type.

All cases contain string values.

The raw values that an enum can store are only reserved for `String`, `Character`, and integer and floating-point number types.

An enum with raw values means each case has a value that's defined at compile-time. In contrast, enums with associated types—which you've used in the previous sections—store their values at runtime.

When creating an enum with a `String` raw type, each raw value takes on the name of the case. You don't need to add a string value if the `rawValue` is the same as the case name, as shown here.

Listing 2.33 Enum with raw values, with string values omitted

```
enum Currency: String {
    case euro
    case usd
    case gbp
}
```

Since the enum still has a raw value type, such as `String`, each case still carries the raw values inside them.

2.5.1 Dangers of raw values

Use some caution when working with raw values, because once you read an enum's raw values, you lose some help from the compiler.

For instance, you're going to set up parameters for a hypothetical API call. You'd use these parameters to request transactions in the currency you supply.

You'll use the `Currency` enum to construct parameters for your API call. You can read the enum's raw value by accessing the raw value property, and set up your API parameters that way.

Listing 2.34 Setting a raw value inside parameters

```
let currency = Currency.euro
print(currency.rawValue) // "euro"

let parameters = ["filter": currency.rawValue]
print(parameters) // ["filter": "euro"]
```

To introduce a bug, change the `rawValue` of the `euro` case, from "euro" to "eur" (dropping the "o"), since *eur* is the currency notation of the euro.

Listing 2.35 Renaming a string

```
enum Currency: String {
    case euro = "eur"
    case usd
    case gbp
}
```

Because the API call relied on the `rawValue` to create your parameters, the parameters are now affected for the API call.

The compiler won't notify you, because the raw value is still valid code.

Listing 2.36 Unexpected parameters

```
let parameters = ["filter": currency.rawValue]
// Expected "euro" but got "eur"
print(parameters) // ["filter": "eur"]
```

Everything still compiles. Unfortunately, you silently introduced a bug in part of your application.

Always make sure to update a string everywhere, which may sound obvious. But imagine that you're working on a big project where this enum was created in a completely different part of the application, or perhaps offered from a framework. An innocuous change on the enum may be damaging elsewhere in your application. These issues can sneak up on you, and they're easy to miss because you don't get notified at compile time.

You can play it safe and ignore an enum's raw values and match on the enum cases. As shown in the following code, when you set the parameters this way, you'll know at compile time when a case changes.

Listing 2.37 Explicit raw values

```
let parameters: [String: String]
switch currency {
    case .euro: parameters = ["filter": "euro"]
    case .usd: parameters = ["filter": "usd"]
    case .gbp: parameters = ["filter": "gbp"]
}

// Back to using "euro" again
print(parameters) // ["filter": "euro"]
```

You're recreating strings and ignoring the enum's raw values. It may be redundant code, but at least you'll have precisely the values you need. Any changes to the raw values won't catch you off guard because the compiler will now help you. You could even consider dropping the raw values altogether if your application allows.

Perhaps even better is that you *do* use the raw values, but you add safety by writing unit tests to make sure that nothing breaks. This way you'll have a safety net and the benefits of using raw values.

These are all trade-offs you'll have to make. But it's good to be aware that you lose help from the compiler once you start using raw values from an enum.

2.5.2 Matching on strings

Whenever you pattern match on a string, you open the door to missed cases. This section covers the downsides of matching on strings and showcases how to make an enum out of it for added safety.

In the next example, you're modeling a user-facing image management system in which customers can store and group their favorite photos, images, and gifs. Depending on the file type, you need to know whether or not to show a particular icon, indicating it's a jpeg, bitmap, gif, or an unknown type.

In a real-world application, you'd also check real metadata of an image; but for a quick and dirty approach, you'll look only at the extension.

The `iconName` function gives your application the name of the icon to display over an image, based on the file extension. For example, a jpeg image has a little icon shown on it; this icon's name is `"assetIconJpeg"`.

Listing 2.38 Matching on strings

```
func iconName(for fileExtension: String) -> String {
    switch fileExtension {
    case "jpg": return "assetIconJpeg"
    case "bmp": return "assetIconBitmap"
    case "gif": return "assetIconGif"
    default: return "assetIconUnknown"
    }
}

iconName(for: "jpg") // "assetIconJpeg"
```

Matching on strings works, but a couple of problems arise with this approach (versus matching on enums). Making a typo is easy, and thus harder to make it match—for example, expecting "jpg" but getting "jpeg" or "JPG" from an outside source.

The function returns an unknown icon as soon as you deviate only a little—for example, by passing it a capitalized string.

Listing 2.39 Unknown icon

```
iconName(for: "JPG") // "assetIconUnknown", not favorable
```

Sure, an enum doesn't solve all problems right away, but if you repeatedly match on the same string, the chances of typos increase.

Also, if any bugs are introduced by matching on strings, you'll know it at runtime. But switching on enums are exhaustive. If you were to switch on an enum instead, you'd know about bugs (such as forgetting to handle a case) at compile time.

Let's create an enum out of it! You do this by introducing an enum with a `String` raw type.

Listing 2.40 Creating an enum with a `String` raw value

```
enum ImageType: String {
  case jpg
  case bmp
  case gif
}
```

Introducing the enum

This time when you match in the `iconName` function, you turn the string into an enum first by passing a `rawValue`. This way you'll know if `ImageType` gets another case added to it. The compiler will tell you that `iconName` needs to be updated and handle a new case.

Listing 2.41 `iconName` creates an enum

```
func iconName(for fileExtension: String) -> String {
    guard let imageType = ImageType(rawValue: fileExtension) else {
        return "assetIconUnknown"
    }
    switch imageType {
    case .jpg: return "assetIconJpeg"
    case .bmp: return "assetIconBitmap"
    case .gif: return "assetIconGif"
    }
}
```

The function tries to convert the string to ImageType; it returns "assetIconUnknown" if this fails.

iconName now matches on the enum, giving you compiler benefits if you missed a case.

But you still haven't solved the issue of slightly differing values, such as "jpeg" or "JPEG." If you were to capitalize "jpg," the `iconName` function would return `"assetIconUnknown"`.

Let's take care of that now by matching on multiple strings at once. You can implement your initializer, which accepts a raw value string.

Listing 2.42 Adding a custom initializer to `ImageType`

```
enum ImageType: String {
    case jpg
    case bmp
    case gif

    init?(rawValue: String) {
        switch rawValue.lowercased() {
        case "jpg", "jpeg": self = .jpg
        case "bmp", "bitmap": self = .bmp
        case "gif", "gifv": self = .gif
        default: return nil
        }
    }

}
```

The string matching is now case-insensitive, making it more forgiving.

The initializer matches on multiple strings at once, such as "jpg" and "jpeg."

OPTIONAL INIT? The initializer from `ImageType` returns an optional. An optional initializer indicates that it can fail. When the initializer does fail—when you give it an unusable string—the initializer returns a nil value. Don't worry if this isn't clear yet; you'll handle optionals in depth in chapter 4.

Note a couple of things here. You set the `ImageType` case depending on its passed `rawValue`, but not before turning it into a lowercased string so you make the pattern matching case-insensitive. Next, you give each case multiple options to match on—such as `case "jpg", "jpeg"`—so that it can catch more cases. You could have written it out by using more cases, but this is a clean way to group pattern matching.

Now your string matching is more robust, and you can match on variants of the strings.

Listing 2.43 Passing different strings

```
iconName(for: "jpg") // "Received jpg"
iconName(for: "jpeg") // "Received jpg"
iconName(for: "JPG") // "Received a jpg"
iconName(for: "JPEG") // "Received a jpg"
iconName(for: "gif") // "Received a gif"
```

If you do have a bug in the conversion, you can write a test case for it and only have to fix the enum in one location, instead of fixing multiple string-matching sprinkled around in the application.

Working with strings this way is now more idiomatic; the code has been made safer and more expressive. The trade-off is that a new enum has to be created, which may be redundant if you pattern-match on a string only once.

But as soon as you see code matching on a string repeatedly, converting it to an enum is a good choice.

2.5.3 Exercises

6 Which raw types are supported by enums?
7 Are an enum's raw values set at compile time or runtime?
8 Are an enum's associated values set at compile time or runtime?
9 Which types can go inside an associated value?

2.6 Closing thoughts

As you can see, enums are more than a list of values. Once you start "thinking in enums," you'll get a lot of safety and robustness in return, and you can turn structs to enums and back again.

I hope that this chapter inspired you to use enums in surprisingly fun and useful ways. Perhaps you'll use enums more often to combine them with, or substitute for, structs and classes.

In fact, perhaps next time as a pet project, see how far you can get by using only enums and structs. Limiting yourself to enums and structs is an excellent workout to help you think in sum and product types!

Summary

- Enums are sometimes an alternative to subclassing, allowing for a flexible architecture.
- Enums give you the ability to catch problems at compile time instead of runtime.
- You can use enums to group properties together.
- Enums are sometimes called sum types, based on algebraic data types.
- Structs can be distributed over enums.
- When working with enum's raw values, you forego catching problems at compile time.
- Handling strings can be made safer by converting them to enums.
- When converting a string to an enum, grouping cases and using a lowercased string makes conversion easier.

Answers

1 Can you name two benefits of using subclassing instead of enums with associated types?

A superclass prevents duplication; no need to declare the same property twice. With subclassing, you can also override existing functionality.

2 Can you name two benefits of using enums with associated types instead of subclassing?

No need to refactor anything if you add another type, whereas with subclassing you risk refactoring a superclass and its existing subclasses. Second, you're not forced to use classes.

3 Given the data structure, what is the number of possible variations of `Bagel`?

Twelve (3 toppings times 4 bagel types)

4 Given the data structure, turn `Bagel` into an enum while keeping the same amount of possible variations.

Two ways, because `Bagel` contains two enums. You can store the data in either enum:

```
// Use the Topping enum as the enum's cases.
enum Bagel {
    case creamCheese(BagelType)
    case peanutButter(BagelType)
    case jam(BagelType)
}

// Alternatively, use the BagelType enum as the enum's cases.
enum Bagel {
    case cinnamonRaisin(Topping)
```

```
    case glutenFree(Topping)
    case oatMeal(Topping)
    case blueberry(Topping)
}
```

5 Given the enum representing a puzzle game for a specific age range, how would this enum be represented as a struct instead?

```
enum AgeRange {
    case baby
    case toddler
    case preschooler
    case gradeschooler
    case teenager
}

struct Puzzle {
    let ageRange: AgeRange
    let numberOfPieces: Int
}
```

6 Which raw types are supported by enums?

String, character, and integer and floating-point types

7 Are an enum's raw values set at compile time or runtime?

Raw type values are determined at compile time.

8 Are an enum's associated values set at compile time or runtime?

Associated values are set at runtime.

9 Which types can go inside an associated value?

All types fit inside an associated value.

iOS DEVELOPMENT

Swift IN DEPTH

Tjeerd in 't Veen

It's fun to create your first toy iOS or Mac app in Swift. Writing secure, reliable, professional-grade software is a different animal altogether. The Swift language includes an amazing set of high-powered features, and it supports a wide range of programming styles and techniques. You just have to roll up your sleeves and learn Swift in depth.

Swift in Depth guides you concept by concept through the skills you need to build professional software for Apple platforms, such as iOS and Mac; also on the server with Linux. By following the numerous concrete examples, enlightening explanations, and engaging exercises, you'll finally grok powerful techniques like generics, efficient error handling, protocol-oriented programming, and advanced Swift patterns. Author Tjeerd in 't Veen reveals the high-value, difficult-to-discover Swift techniques he's learned through his own hard-won experience.

What's Inside

- Writing reusable code with generics
- Iterators, sequences, and collections
- Protocol-oriented programming
- Understanding map, flatMap, and compactMap
- Asynchronous error handling with Result
- Best practices in Swift

Written for advanced-beginner and intermediate-level Swift programmers.

Tjeerd in 't Veen is a senior software engineer and architect in the mobile division of a large international banking firm.

"An excellent guide to using the advanced features of Swift to produce clean, high-performing code. The content is masterfully delivered, making it easy to quickly level-up your skills."
—Jason Pike, Atlas RFID Solutions

"Highly recommended to anyone interested in the Apple platform. For the novice who wants to become an expert, this is definitely where you should start!"
—Helmut Reiterer
Revenue Recovery Solutions

"Because Swift is so new, it's hard to find good resources to learn it. Look no further than this book."
—Tyler Slater, Jolt

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/swift-in-depth

ISBN-13: 978-1-61729-518-8
ISBN 10: 1-61729-518-3
54999
9 781617 295188

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]