Author Picks

FREE

Exploring Swift

Chapters selected by

Craig Grummitt

manning

Exploring Swift

Selected by Craig Grummitt

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Candace Gillhoolley, cagi@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617296215
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 23 22 21 20 19 18

contents

introduction

A lot's happened since June 2014, when Apple shocked the developer community by launching a new programming language called Swift. In 2015, another shock was in store when Apple made Swift open source, opening the doors for the developer community to get involved in its progress. In the years since, Swift evolved into the powerful, modern and expressive language it is today. According to the latest stackoverflow survey, Swift's ranked the sixth most-loved programming language. And although Swift's primarily used in iOS and Mac OS development, Swift isn't limited to building iPhone apps and Mac programs—you'll find it available in web development, server-side development and cloud-based services. Craig Federighi, Senior VP of Software Engineering at Apple, threw down the gauntlet in a podcast interview when he declared his hopes for Swift to become "*the* language, *the* major language for the next twenty years of programming in our industry."

It's obviously a good time to explore Swift! This sampler brings together sample chapters on Swift from three books available through Manning Publications. First—we'll look at an introduction to data types in Swift in a chapter from my own book, *iOS Development with Swift.*

Swift objects

It's not only Apple's company motto, working with Swift also helps us to think different! In the chapter "Swift objects" from my book *iOS Development with Swift*, we take a look at data types in Swift—but we don't linger too long on constructs you're probably already familiar with.

We take a look at powerful features in Swift that may be new to you or work differently than what you're used to, such as structs, protocols, initializers, extensions, operator overloading and generics. We also don't skim over the mechanics of these features, we also look at why you'd use them, and when they might be appropriate, with useful examples.

If you're interested in continuing your study of Swift into developing apps for iOS, this book then moves its focus from covering the basics in Swift to iOS development, following the progress of an app from coming up with an idea for an app right through to publishing it on the app store.

Chapter 3 from *iOS Development with Swift*
by Craig Grummitt

Swift objects

This chapter covers

- Exploring objects, methods, and parameters in Swift
- Initializing properties
- Comparing inheritance with protocols
- Differentiating between classes and structs
- Exploring ways to extend your code

It's impossible to do anything in iOS development without using objects. Views are objects, view controllers are objects, models are objects—even basic data types such as `String`, `Int`, and `Array` are objects in Swift!

An object in Swift is a specific instance of a type of thing. In this chapter, we'll look at different ways of building up and structuring these types of things in your code. From experience in other languages, you may know this "type of thing" (or type) as a class. While it's true that types can be represented by classes in Swift, they're not the only type of thing in Swift—other types called structures and enumerations also exist. We'll come back to those, but first let's look at classes.

Don't forget, you can refer to the Swift cheat sheets in appendix B. This chapter is summarized on the last page of the cheat sheets.

3.1 Classes

One approach for creating objects in Swift is with a class. A class defines what a type does with methods. A method is a function defined within a type. Along with methods, a class defines what a type is with properties. Properties are variables or constants stored in a type.

Let's say you've decided to build a distance converter app. Your app will accept distances in miles or kilometers, and will display the distance in either form of measurement, too.

You decide the best approach is to build a type that stores distances, regardless of the scale. You could create a distance with a miles or kilometers value, update the distance with a miles or kilometers value, or use the distance type to return its value as miles or kilometers (see figure 3.1).

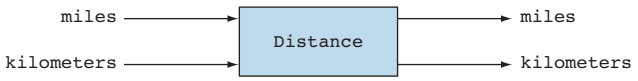


Figure 3.1 Distance type

3.1.1 Defining a class

Let's start by defining a simple `Distance` type with a class. In this chapter, you'll build up this class to contain a distance using different measurement types.

1. Create a new playground to follow along, and call it Distance. Classes are defined with the `class` keyword followed by the name of the class and the rest of the definition contained within curly brackets.
2. Create a `Distance` class.

```
class Distance {

}
```

3. Now that you have a class, you can create (or instantiate) your class with the name of the type, followed by parentheses, and assign this object to a variable:

```
var distance = Distance()
```

You might recognize the parentheses syntax from the previous chapter as an alternative syntax for creating or instantiating simple data types.

Now that you have a class definition for `Distance`, you can add properties and methods to it.

3.1.2 Properties

Variables that we've looked at so far have been global variables—defined outside the context of a class or function. Variables that are defined within a class are called properties, and fall into two broad categories: type properties and instance properties.

TYPE PROPERTIES

Type properties, also known as static properties, are relevant to all things of a certain type. It isn't even necessary that an instance of a type exist to access type properties. Type properties are connected to the type rather than the object. You instantiate a type property with the `static` keyword followed by a normal declaration of a variable.

For example, maybe you'd like to store the number of kilometers in a mile in a type property in your `Distance` class. In this case, a constant would make more sense, because the number of kilometers in a mile won't be changing any time soon. Use the keyword `let` instead of `var` to define a constant.

1. Add a type property constant to your simple `Distance` class:

```
class Distance {
    static let kmPerMile = 1.60934
}
```

   You could then retrieve or set this type property directly on the type.

2. Print to the console using the type property you created:

```
print ("2 miles = \(Distance.kmPerMile * 2) km")
```

INSTANCE PROPERTIES

Instance properties are relevant to specific objects or instances of a type.

Because the `miles` value will be relevant to specific instances of `Distance`, add `miles` as an instance property to your `Distance` class.

```
class Distance {
    static let kmPerMile = 1.60934
    var miles:Double
}
```

Whoops! If you're following along in the playground, you'll notice that this triggers a compiler error. Tap the red dot to see more information on the error (see figure 3.2). A pop-up appears below the line that describes the error along with Xcode's suggested fix.

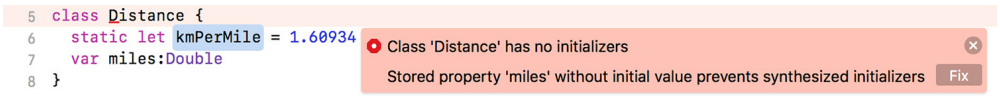


Figure 3.2 Non-optional variable can't equal `nil`

As we explored in the previous chapter, non-optionals can never equal `nil`. The `Distance` class can't contain a `miles` property that's equal to `nil`.

You have three possible alternatives to get rid of that red dot.

- One option is to give the property a default value. This is what Xcode suggests. If you tap Fix Button, Xcode will resolve the problem in this way for you.

  But a default value for the `miles` property doesn't make sense. There's no reason why 0 or any other value should be a default value for `miles`. Press Command-Z to undo this fix.
- Another option is to make the `miles` property an optional. This is easy to do; all you need to do is add a question mark:

  ```
  var miles:Double?
  ```

  This removes the error, but isn't appropriate for this example either. If you define a `Distance` object, you want it to have a value for miles! A distance with a miles value of `nil` doesn't make sense. Undo this fix too.
- You could pass a value to the `miles` property in an initializer. What's an initializer?

3.1.3 Initializers

An initializer is a special type of function that sets up a type. You can use an initializer to pass in values when you instantiate the type.

You can create an initializer with the `init` keyword followed by any parameters you want to pass in to initialize the instance properties.

1. Add an initializer to the `Distance` class to pass in a value to initialize the `miles` property.

   ```
   class Distance {
       static let kmPerMile = 1.60934
       var miles:Double
       init(miles:Double) {
           self.miles = miles
       }
   }
   ```

   Initializer

   Initializes the miles property

   As you can see, you can use the keyword `self` to differentiate between the instance property (`self.miles`) and the parameter (`miles`) that's passed in to the initializer.

   Now that the `miles` property is set in the initializer, the requirement that all non-optionals should contain non-`nil` values is satisfied, and the red dot should go away.
2. You can now instantiate a `Distance` object by passing in a value for miles.

   ```
   var distance = Distance(miles: 60)
   ```

NOTE By default you need to pass in the names of the arguments in initializers and functions. We'll look at this in more detail shortly.

3 Now that you have a `Distance` class, you could introduce a `km` property if you like, and initialize it in the initializer calculated from the `miles` value and the `kmPerMile` type property.

```
class Distance {
    static let kmPerMile = 1.60934
    var miles:Double
    var km:Double
    init(miles:Double) {
        self.miles = miles
        self.km = miles * Distance.kmPerMile
    }
}
```

Adds km property

Calculates km from miles

In case we need to calculate kilometers again, it may make sense to move this calculation to a method.

NOTE If all properties of a class have default values, Xcode will synthesize a default initializer automatically for you with no arguments.

3.1.4 Methods

Functions defined inside a class are called methods. Like variables and properties, methods can be divided into *instance methods* or *type methods*.

Instance methods are methods that are relevant to an instance of a type, whereas type methods apply to the type itself.

INSTANCE METHODS

Instance methods are relevant to each instance of a type.

In the future, you might want your `Distance` class to return a nicely formatted version of its data. Because the response will be different for each instance of `Distance`, this would be more relevant as an instance method.

1 Add an instance method to your `Distance` class that returns a nicely formatted miles string.

```
func displayMiles()->String {
    return "\(Int(miles)) miles"
}
```

2 You can call your instance method now using a `Distance` object.

```
var distance = Distance(miles: 60)
print(distance.displayMiles())
//prints "60 miles" to console
```

You currently calculate kilometers from miles in the `Distance` initializer. Let's refactor this calculation into a reusable method. You might be tempted to use an instance method, but you'll find this approach causes an error.

3 Add an instance method that calculates kilometers from miles, and call it from the initializer.

```
class Distance {
    static let kmPerMile = 1.60934
    var miles:Double
    var km:Double
    init(miles:Double) {
        self.miles = miles
        self.km = toKm(miles:miles)
    }
    func toKm(miles:Double)->Double {
        return miles * Distance.kmPerMile
    }
}
```

Call instance method; error here

Instance method

Curious! Why does calling an instance method in the initializer cause an error?

Until an initializer has fulfilled its duties to provide initial values for all non-optionals, the instance isn't designated as safe and therefore its instance properties and methods can't be accessed.

To solve this problem, one solution could be to ensure that all properties have values before using the instance method:

```
init(miles:Double) {
    self.miles = miles
    self.km = 0
    self.km = toKm(miles:miles)
}
```

Provides default value

No error now!

But stepping back from the problem, converting miles to kilometers could be as easily set up as a useful utility method on the type. Let's refactor our `toKm` method as a type method.

TYPE METHODS

Like type properties, type methods (also known as static methods) are methods that can be called directly on the type, rather than individual instances of the type.

1 Use the `static` keyword to refactor the `toKm` method as a type method. Type methods have implicit access to type properties, so we can remove the class name `Distance` before `kmPerMile`:

```
static func toKm(miles:Double)->Double {
    return miles * kmPerMile
}
```

Similar to the way you used type properties, call a type method by prefacing it with the type. For example, here's how you could call the `toKm` method we set up on the `Distance` class:

```
print(Distance.toKm(miles: 30))
```

Because type methods are called on the type and don't depend on an instance of a type, they can be used to initialize properties in the initializer.

2 Call your new static method in the initializer for `Distance`.

```
init(miles:Double) {
    self.miles = miles
    self.km = Distance.toKm(miles:miles)
}
```

OVERLOADING

It can be strange to developers new to Swift that it's completely valid in Swift to have two functions with the same name, as long as the names or types of the parameters are distinct. This is called *overloading* a function. "Overloading a function"—even the name sounds a little scary! Don't worry, this is standard practice in Swift and a useful tool.

At the moment, the `Distance` class has a static method called `toKm` that calculates kilometers from miles. What if later you find you need to calculate kilometers from another form of measurement, for example, feet? You'll probably want to name that method `toKm`, too. Well, in Swift you can do this by overloading the function by defining two functions with different parameter names, as shown in the following listing.

Listing 3.1 Overloading a function with different parameter names

```
static let feetPerKm:Double = 5280

static func toKm(miles:Double)->Double {
    return miles * kmPerMile
}
static func toKm(feet:Double)->Double {
    return feet / feetPerKm
}
```

Which method you use depends on the parameter name you pass:

```
let km = Distance.toKm(miles:60)    //96.5604
let km2 = Distance.toKm(feet:100)   // 0.03048
```

Similarly, perhaps in the future you want your `Distance` class to accept an `Int` value for `km` in your `toMiles` method. This time, you could overload the function by defining two functions with the same name that expect different data types, as shown in the following listing.

Listing 3.2 Overloading a function with different parameter data types

```
static func toMiles(km:Double)->Double {
    return km / kmPerMile
}
static func toMiles(km:Int)->Double {
    return Double(km) / kmPerMile
}
```

Again, the method you use depends on the data type of the parameter you pass. Initializers can be overloaded as well.

1 Add a second initializer for the `Distance` class to initialize the object based on kilometers. You'll need to add a type method to calculate miles from kilometers as well.

```
class Distance {
    static let kmPerMile = 1.60934
    var miles:Double
    var km:Double
    init(miles:Double) {
        self.miles = miles
        self.km = Distance.toKm(miles:miles)
    }
    init(km:Double) {
        self.km = km
        self.miles = Distance.toMiles(km:km)
    }
    static func toKm(miles:Double)->Double {
        return miles * kmPerMile
    }
    static func toMiles(km:Double)->Double {
        return km / kmPerMile
    }
}
```

Overloaded initializer

New type method

2 You can now use miles or kilometers to instantiate a `Distance` object:

```
var distance1 = Distance(miles: 60)
var distance2 = Distance(km: 100)
```

The `Distance` class is shaping up, but it has a bit of redundancy to it. Whether you store the distance in miles or kilometers, you're storing the same distance twice using two different measurement units. Shortly, we'll look at how to clean up that redundancy with computed properties.

Convenience initializers

The initializers we've looked at so far have been *designated* initializers—the main initializer for the class that ensures that all instance properties have their initial values. *Convenience* initializers are alternative initializers that add the keyword `convenience`, and, by definition, must ultimately call `self`'s designated initializer to complete the initialization process. Instead of overloading the initializer in the `Distance` class, we could have added a convenience initializer.

```
convenience init(km:Double) {
  self.init(miles:Distance.toMiles(km:km))
}
```

Convenience keyword

Calls designated initializer

3.1.5 Computed properties

Computed properties are properties that calculate their values from other properties.

As you saw earlier, there might be a point in the future when you want to add additional measurements to your `Distance` class—centimeters, feet, inches, cubits, yards, furlongs, nautical miles, light years, you get the idea. Should you keep all these versions of the same distance in memory? Probably not.

One solution to avoid this redundancy is to decide on one core property that will store the distance—in our `Distance` class, this could be `miles`. Then the other properties, rather than storing values, will calculate their value from the `miles` property. These types of properties will be computed properties.

Computed properties lie somewhere between properties and methods—they're methods implemented with the syntax of properties. They act similarly to getters and setters in other languages.

The computed property itself doesn't store any data. Rather, when the property's value is retrieved, the getter calculates a value to return. Calculations are performed in curly brackets {} and the value is returned using the `return` keyword.

1. To avoid redundancy, convert the `km` property to a read-only computed property. The `km` property will no longer store data; rather, it will calculate kilometers from the `miles` property at the moment it's requested. The initializers will no longer need to set the `km` property and will set the miles property directly.

```
class Distance {
    static let kmPerMile = 1.60934
    var miles:Double
    var km:Double {
        return Distance.toKm(miles:miles)
    }
    init(miles:Double) {
        self.miles = miles
        self.km = Distance.toKm(miles:miles)
    }
    init(km:Double) {
        self.km = km
        self.miles = Distance.toMiles(km:km)
    }
    static func toKm(miles:Double)->Double {
        return miles * kmPerMile
    }
    static func toMiles(km:Double)->Double {
        return km / kmPerMile
    }
}
```

2. Confirm that the `km` property can continue to be retrieved like a normal property.

```
var distance = Distance(km: 100)
print ("\(distance.km) km is \(distance.miles) miles")
```

This solves the redundancy, but unfortunately there's a problem. You want to be able to update a distance object by setting the kilometer value.

3 Check what happens when you update the `km` property.

```
distance.km = 90          <-- Error
```

Because `km` is a read-only property, attempting to update it causes an error.

Computed properties can optionally also implement a setter. A setter is a block of code that's called when a computed property is set. Because the computed property doesn't store any data, the setter is used to set the same values that derive the computed property's value in the getter.

The getter approach used in the previous example uses shorthand syntax to implement the getter. The longhand syntax uses a `get` keyword followed by curly brackets `{}`.

4 Convert the `km` computed property to use the longhand syntax.

```
var km:Double {
    get {                                  <-- Explicit getter syntax
        return Distance.toKm(miles:miles)
    }
}
```

The `set` syntax is similar to the `get` syntax, with the exception that the set syntax receives a variable representing the new value.

5 Convert the `km` computed property so that it now can be "set," as per the following code snippet:

```
class Distance {
    static let kmPerMile = 1.60934
    var miles:Double
    var km:Double {
        get {
            return Distance.toKm(miles:miles)
        }
        set(newKm) {
            miles = Distance.toMiles(km:newKm)
        }
    }
    init(miles:Double) {
        self.miles = miles
    }
    init(km:Double) {
        self.miles = Distance.toMiles(km:km)
    }
    static func toKm(miles:Double)->Double {
        return miles * kmPerMile
    }
    static func toMiles(km:Double)->Double {
        return km / kmPerMile
    }
}
```

As you can see, setting the `km` property doesn't store the value of kilometers. Instead, it calculates and stores a value in the `miles` property.

6 Confirm you can now update a distance object using either miles or kilometers:

```
var distance = Distance(km: 100)
distance.km = 35
distance.miles = 90
```

7 Confirm you can also retrieve the values of either miles or kilometers:

```
print("Distance is \(distance.miles) miles")
print("Distance is \(distance.km) km")
```

Mission complete!

DOWNLOAD You can check your `Distance` class with mine in the Distance.playground. Download all the code for this chapter by selecting Source Code > Clone and entering the repository location: https://github.com/iOSAppDevelopmentwithSwiftinAction/Chapter3.

CHALLENGE Confirm in the results sidebar that the distance object is instantiating, updating, and displaying correctly using miles or kilometers.

3.1.6 Class inheritance

If you're experienced in object-oriented programming (OOP), class inheritance and subtyping will most likely be a familiar topic. In Swift, multiple classes can inherit the implementation of one class through subclassing, forming an is-a relationship.

NOTE If you're familiar with class inheritance, you can skim through to the section called "Pros and cons."

Classes and subclasses form a hierarchy of relationships that looks like an upside-down tree. At the top of the tree is the base class from which all classes inherit, and every subclass inherits the methods and properties of its superclass and can add on implementation.

Let's explore inheritance by building up a class structure representing telephones. Different types of telephones exist—from older rotary phones to the latest iPhones, but they all share common functionalities: to make calls and to hang up.

See figure 3.3 for a simplified representation of the hierarchy of relationships of different types of telephones. At the base (top) of the tree is an abstract telephone, which can initiate and terminate calls. This branches into landline and cellular phones. Both landlines and cellular phones inherit the telephone's ability to initiate and terminate calls, but the cellular phone adds the ability to send an SMS. The various types of phones that inherit from landlines and cellular phones add (among other things) different input techniques. The various types of smartphones add their own implementation of an operating system.

NOTE This example isn't intended to be comprehensive. If I listed everything a smart phone could do, I'd be here all day!

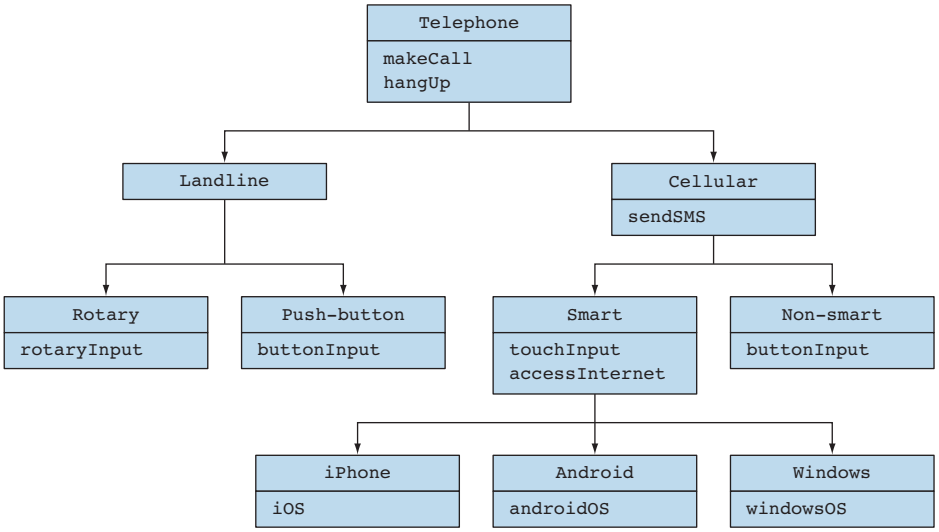


Figure 3.3 Telephone inheritance

You could model these relationships with classes. Subclasses indicate their superclass with a colon after their name, as shown in the following listing.

Listing 3.3 Class inheritance

```
class Telephone {
    func makeCall() {
        //make a call here
    }
    func hangUp() {
        //hang up here
    }

}
class landline:Telephone {

}
class Cellular:Telephone {
    func sendSMS() {
        //send SMS here
    }
}
//...
```

Landline subclasses Telephone

Cellular subclasses Telephone

Cellular adds functionality

After modeling this hierarchy, a method could receive a `Telephone` parameter, and regardless of whether the parameter passed is an Android, iOS, or even a rotary phone, the method knows that it can tell the telephone to `makeCall()` or `hangUp()`:

```
func hangUpAndRedial(telephone:Telephone) {
    telephone.hangUp()
    telephone.makeCall()
}
```

OPEN Explore the rest of the code in the Telephone-ClassInheritance.playground.

OVERRIDING

In addition to inheriting the implementation of a superclass, a subclass can override this implementation.

The `Cellular` class probably wants to implement its own version of making a call on cellular networks. It can do this by overriding the `makeCall` method, as shown in the following listing.

Listing 3.4 Override method

```
class Cellular:Telephone {
    override func makeCall() {
        //make cellular call
    }
    func sendSMS() {
        //send SMS here
    }
}
```

Overriding a method will, by default, prevent the superclass's implementation of that method from running. Sometimes, a subclass might want to add to the superclass's implementation rather than replace it. In this case, the subclass can use the `super` keyword to first call the method on the superclass, as shown in the following listing.

Listing 3.5 Call `super`

```
override func makeCall() {
    super.makeCall()
    //make cellular call
}
```

PROS AND CONS

Class inheritance is used extensively throughout Apple frameworks. For example, as you saw in chapter 1, the `UIButton` class subclasses the `UIControl` class, which, in turn, subclasses `UIView`.

Inheritance is a powerful technique for expressing relationships and sharing implementation between classes and lies at the heart of object-oriented programming.

Inheritance has issues, however, that are worth noting.

- *Swift only permits inheritance from one class.* iPhones aren't simply telephones any more. They're game consoles, e-readers, video players, compasses, GPS devices, step counters, heart rate monitors, fingerprint readers, earthquake detectors, and the list goes on. How can an iPhone share common functionality and implementation with these other devices? According to the simple inheritance model, they can't.
- *Sharing code can only happen between subclasses and superclasses.* Non-smart phones and push-button phones both have push-button input, but neither of them inherits from each other. iPads have iOS too, but they aren't telephones. These common implementations couldn't be shared, according to the pure inheritance model.
- *Sometimes it's not so clear which identity is the most relevant to subclass.* Should you have subclassed smartphones by operating system or by manufacturer? Both are important and could potentially contain different functionality or properties.

The trend in pure Swift has moved away from class inheritance and toward implementation of protocols.

3.1.7 Protocols

Protocols are similar to interfaces in other languages. They specify the methods and properties that a type that adopts the protocol will need to implement.

Protocol methods only indicate the definition of the method and not the actual body of the method, for example:

```
func makeCall()
```

If you rewrote the abstract `Telephone` class as a protocol, it would look like the following code snippet:

```
protocol Telephone {
    func makeCall()
    func hangUp()
}
```

Protocol methods

A type adopts a protocol with syntax similar to inheritance—a colon after the type name. As the methods in a protocol don't contain any implementation, a class that adopts the protocol must explicitly implement these methods. If you rewrote the `Landline` class to adopt the `Telephone` protocol, it would look like the following code snippet:

```
class Landline:Telephone {
    func makeCall() {
        //make a landline call here
    }
    func hangUp() {
        //hang up a landline call here
    }
}
```

Adopts the Telephone protocol

Implements the protocol methods

Protocol properties only indicate whether a property can be retrieved or set. For example, if you add a phone number property to `Telephone`, it looks like the following code snippet:

```
protocol Telephone {
    var phoneNo:Int { get set }
    func makeCall()
    func hangUp()
}
```

Protocol property

The protocol only specifies that the `phoneNo` property needs to exist in an adopting type, and that the property needs to get or set. Implementing the property is left to the adopting class.

```
class Landline:Telephone {
    var phoneNo:Int
    init(phoneNo:Int) {
        self.phoneNo = phoneNo
    }
    func makeCall() {
        //make a landline call here
    }
    func hangUp() {
        //hang up a landline call here
    }
}
```

Adopts the protocol property

Initializes the property

PROTOCOL EXTENSIONS

Okay. I have a confession to make.

I've been suggesting that protocols don't contain implementation, and that's not entirely true. Protocols are blessed with the magical ability to be *extended* to add actual functionality, which types that adopt the protocol will have access to.

In the previous example, the functionality of making a call and hanging up could be implemented in the `Telephone` protocol through use of an extension, as shown in the following listing.

Listing 3.6 Extending a protocol

```
protocol Telephone {
    var phoneNo:Int { get set }
    func makeCall()
    func hangUp()
}
extension Telephone {
    func makeCall() {
        print("Make call")
    }
    func hangUp() {
        print("Hang up")
    }
}
```

Extension of protocol

Implementation of methods in protocol

```
class Landline:Telephone {
    var phoneNo:Int
    init(phoneNo:Int) {
        self.phoneNo = phoneNo
    }
}
```

Because these methods are now implemented in the `Telephone` protocol, they no longer need to be implemented in a class that adopts that protocol. Note that the `Landline` class no longer implements the `makeCall` or `hangUp` methods.

Extended protocols still can't store properties, but because computed properties don't store properties, computed properties can be implemented in extended protocols.

Protocol relationships

This integration of protocols and protocol extensions into the Swift language made different and complex approaches possible for structuring relationships between types. This is due to several factors:

- Like classes, protocols can inherit other protocols.
- Types can adopt multiple protocols.
- Protocols can represent different types of relationships.

Class inheritance places the emphasis on is-a relationships. As you've seen, protocols can represent this relationship as well. When protocols represent an is-a relationship, the convention is to use a noun. In our example, `Landline` is-a `Telephone`.

But protocols aren't limited to identity or is-a relationships. Another common relationship that is represented is capabilities, or can-do. A common convention for protocols that represent a can-do relationship is to suffix its name with "able," "ible," or "ing."

Relationships in the real world are often not as simple as a pure inheritance model can handle. Complexity and nuance need to be addressed, and protocols and protocol extensions are useful for this.

Let's look again at telephones, converting subclasses to is-a and can-do protocols. Figure 3.4 illustrates one way you could redraw their relationships.

In this example, a protocol called `PushButtonable` could be written to handle the capability of button input. This protocol could then be adopted by both the push-button landline and the non-smart cellular phone. Despite not having an inheritance relationship, the two classes could still share implementation through the `PushButtonable` protocol extension.

The iPhone no longer inherits all its *smart* characteristics through the `Smart` class. Rather, it adopts specific capabilities through protocols such as `Touchable` or `Internetable`. In this way, it could go beyond traditional telephone capabilities and adopt protocols and share implementation through protocol extensions with completely different devices. Maybe it could share `VideoPlayable` along with `Television`, `Navigable` along with `GPSDevice`, or `GamePlayable` along with `GameConsole`.

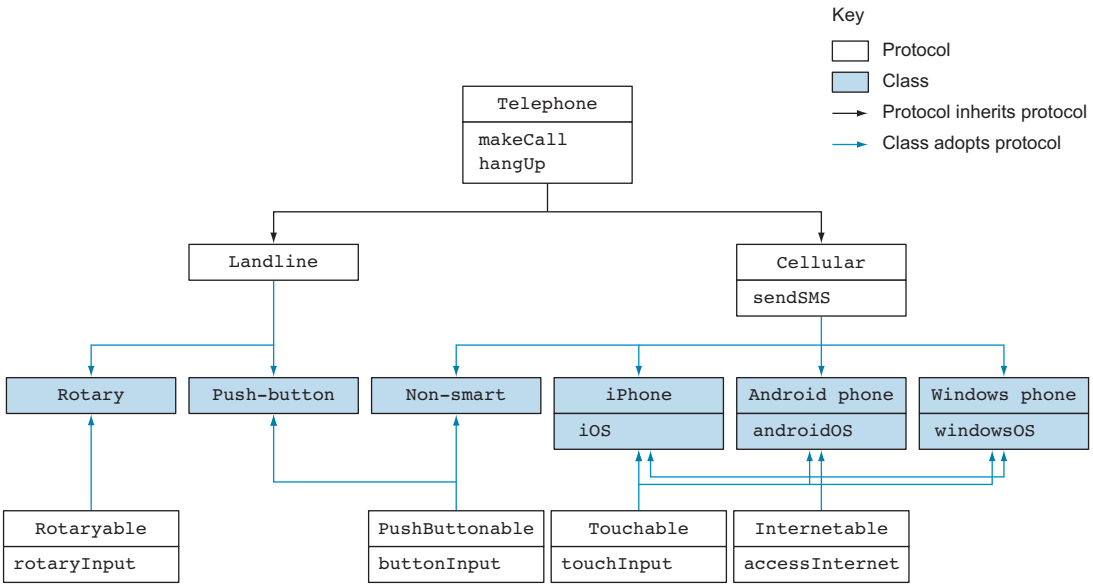


Figure 3.4 **Telephone using protocols**

Using protocols to structure the relationships in your code has been coined protocol-oriented programming. Sure, you could continue to program in Swift using familiar object-oriented programming techniques, but it's worth exploring the possibilities with protocols.

OPEN Explore the protocol relationships in code in the TelephoneProtocols.playground.

CHALLENGE Add a `Television` type that shares a `VideoPlayable` protocol with iPhones, Androids, and Windows phones.

3.2 Structures

Classes aren't the only "type of thing" in Swift. An alternative approach to creating objects in Swift is with a structure.

Structures have many similarities to classes. For example, they can

- Have properties
- Have methods
- Have initializers
- Adopt protocols

Define a structure with the `struct` keyword, for example:

```
struct Telephone {

}
```

Instantiation of a structure is identical to that of a class:

```
var telephone = Telephone()
```

3.2.1 Structures vs. classes

Structures have three main differences from classes worth noting:

- Structures can't inherit.
- Structures can have memberwise initializers.
- Structures are value types.

Each of these is explained in the following sections.

STRUCTURES CAN'T INHERIT

Structures can't inherit other structures. They can indirectly inherit functionality, however, by adopting protocols, which, as you've seen, can inherit other protocols.

MEMBERWISE INITIALIZERS

If you don't set up an initializer for a structure, an initializer that accepts all the structure's properties as parameters will automatically be generated for you. This automated initializer is called a memberwise initializer.

As you saw earlier in the chapter, when the `Distance` class didn't initialize its `miles` property, an error appeared. If you change the definition of this class to a `struct`, a memberwise initializer is automatically generated and the error disappears:

```
struct Distance {
    var miles:Double
}
```

You can now instantiate this structure using the memberwise initializer:

```
var distance = Distance(miles: 100)
```

STRUCTURES ARE VALUE TYPES

An important distinction between structures and classes is how they're treated when they're assigned to variables or passed to functions. Classes are assigned as *references*, and structures are assigned as *values*.

Look at the following listing. Predict the value of `color1.name` that will be printed to the console.

Listing 3.7 Changes to reference types

```
class Color {
    var name = "red"
}
var color1 = Color()
var color2 = color1
color2.name = "blue"
print(color1.name)
```

If you predicted `"blue"`, pat yourself on the back! Because classes are reference types, when `color1` was assigned to the `color2` variable, `color2` was assigned the reference to the underlying `Color` object (see figure 3.5).

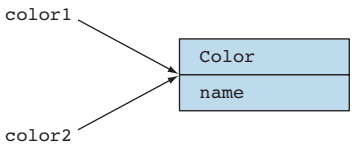


Figure 3.5 Reference types

In the end, both `color1` and `color2` refer to the same object, and any changes to `color2` are reflected in `color1` (and vice versa).

In Swift, core data types such as `String` are value types. Look at the following listing and predict the value of `letter1` that will be printed to the console.

Listing 3.8 Changes to value types

```
var letter1 = "A"
var letter2 = letter1
letter2 = "B"
print(letter1)
```

If you went with `"A"`, you're right. This time, when `letter2` was assigned to the `letter1` variable, `letter2` was assigned the *value* of `letter1`, instantiating a new String object. You're left with two String objects, as in figure 3.6.

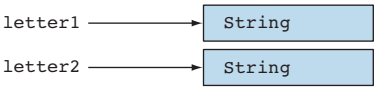


Figure 3.6 Value types

Because you now have two separate String objects, making a change to one of them doesn't affect the other.

Like `Strings`, when a structure is assigned to a new variable, it's copied. Let's look at the `Color` example again, but tweak one thing—it's now a structure rather than a class (to be clear, let's also rename it `ColorStruct`). Now, what is the value of `color1.name` that will be printed to the console in the following?

```
struct ColorStruct {
    var name = "red"
}
var color1 = ColorStruct()
var color2 = color1
color2.name = "blue"
print(color1.name)
```

If you predicted `"red"`, you're paying attention! Because structures are value types, when `color2` was assigned `color1`, only the value of `color1` was copied, two `ColorStruct` objects now exist, and any changes to `color2` aren't reflected in `color1`. Try it out in a playground and see for yourself!

Since Swift went open source, it's been fascinating to explore how the language looks "under the hood." One thing you'll discover if you look at the source of Swift is that many of the core data types are implemented as structs, explaining why types such as `String` are value types. Incidentally, this represents a change in direction from

Objective-C, where many types are implemented as classes (though references are implemented differently).

Constants

We've looked at constants in brief, but now's a good time to look at them a little closer.

You undoubtedly are familiar with constants—they're a special type of variable that will never be reassigned. In Swift, a constant is declared using the `let` keyword instead of `var`.

For example, if you assign an instance of a `Person` type to a constant, you can't later assign another instance of the `Person` type to the same constant:

```
let person = Person(name: "Sandra")
person = Person(name: "Ian")
```

Error—can't reassign constant

TIP If a variable is never reassigned, for performance reasons you should declare it a constant.

Here's a tricky question for you: is it permissible to modify a property of a constant of the `Person` type? For example:

```
person.name = "Ian"
```

If your answer was a confused expression and a shrug of the shoulders, you're right!

Whether a property of a constant can be modified depends on whether you have a value type or a reference type, and I wasn't clear in the question about whether `Person` was defined as a class or a structure. I did warn you it was going to be tricky!

For value types, the identity of the constant is tied up with the properties it contains. If you change a property, the variable is no longer the same value. For value types such as structures, it isn't permissible to modify a constant's properties.

For reference types, the identity of the constant is a reference to an object. There could be other constants or variables that point to that same object. For reference types such as classes, it's permissible to modify a constant's properties.

Which object type?

After learning the differences between classes and structures, the next question most people want the answer to is this: which should I use, and when?

To arrive at an answer of that complex question I find it helps to break it down into smaller questions:

- *Does the type need to subclass?* The choice may be clear—sometimes your type needs to subclass; therefore, you need a class.
- *Should instances of this type be one of a kind?* If you're storing data in a type, and want any changes to that data to be reflected elsewhere, it might make sense to use a class.
- *Is the value tied to the identity of this type?* Consider a `Point` type that stores an `x` and a `y` value. If you have two points that are both equal to `(x:0, y:0)`, would

they be equivalent? I suggest that they would. Therefore, the value is tied to its identity and it should probably be implemented as a structure.

Now, consider an `AngryFrog` type that among other properties also contains an `x` and a `y` value. If you have two angry frogs that both are positioned at `(x:0, y:0)`, would they be equivalent? I suggest probably not, because they're probably two distinct entities, maybe traveling in different directions, or may be controlled by different players. The identity of an `AngryFrog` would be tied to a reference to a specific instance rather than the current values of its properties, and therefore it should probably be implemented as a class.

For a visual representation of this decision process, see figure 3.7.

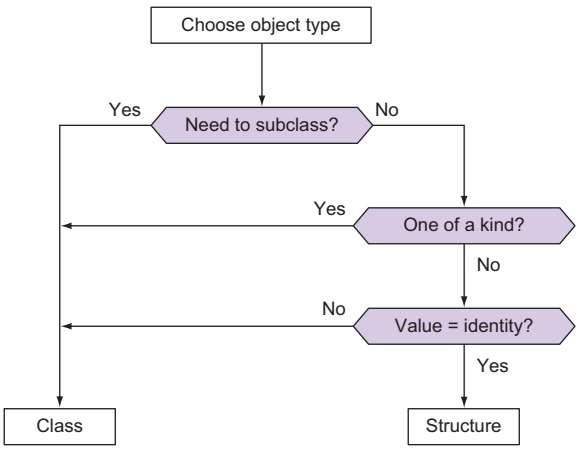


Figure 3.7 Structure or class decision

A complex codebase may have additional factors to consider, but I find these three questions a handy guide to arrive at an answer to the structure or class decision.

Let's practice this decision process with the `Distance` type you worked with earlier in the chapter:

- Does the `Distance` type need to subclass? No, it doesn't.
- Should there be only one `Distance` object? No, there can be more than one.
- Is the value equivalent to its identity? If you had two 100 km `Distance` objects, they should be treated as equivalent, so yes, the value is equivalent to identity.

Therefore, the `Distance` type should probably be implemented as a structure. Fortunately, changing a class to a structure or vice versa is straightforward. Swap the `class` keyword over for `struct`, and that's often all that's necessary. Go ahead and change the `Distance` class to a structure now.

We still haven't looked at all the object types available in Swift. To make things even more interesting, you have yet another alternative to classes and structures, called enums. We'll cover enums in chapter 10.

3.3 Extensions

We've looked at protocol extensions to add functionality to protocols. Extensions can also be used to add functionality to classes and structures.

There's much that extensions can do, but they do have limitations:

- Extensions can't override functionality.
- Extensions can add computed properties, but can't add stored properties.
- Extensions of classes can't add designated initializers.

3.3.1 Extensions of your type

When we looked at the `Distance` class earlier in the chapter, we considered that at a later point we may want to add additional measurements. Well, the time has come! Let's add feet to the `Distance` structure.

1. Open your Distance playground again.
2. Create an extension of your `Distance` structure.

```
extension Distance {
}
```

3. Add a feet computed property.

```
static let feetPerMile:Double = 5280
```

4. Add type methods to your extension to convert to miles and kilometers from feet, or back again to feet from miles.

```
static func toMiles(feet:Double)->Double {
    return feet / feetPerMile
}
static func toKm(feet:Double)->Double {
    return toKm(miles:toMiles(feet:feet))
}
static func toFeet(miles:Double)->Double {
    return miles * feetPerMile
}
```

5. You can set up a computed property now for feet.

```
var feet:Double {
    get {
        return Distance.toFeet(miles:miles)
    }
    set(newFeet) {
        miles = Distance.toMiles(feet: newFeet)
    }
}
```

6 Finally, create an initializer for the `Distance` structure.

```
    init(feet:Double) {
        self.miles = Distance.toMiles(feet:feet)
    }
}
```

Your `Distance` structure can now be initialized with `feet` and updated by setting `feet`.

OPEN Compare your `Distance` extension with mine in the DistanceExtensions.playground.

CHALLENGE To confirm it's now possible, create a new instance of `Distance` using feet, update this value, and then print this value to the console. Then extend the DistanceExtensions playground to include another form of measuring distance.

3.3.2 Extensions of their type

You aren't limited to extending your own code. You can also extend classes, structures, or protocols of third-party code, or even of Apple frameworks or the Swift language!

As you saw in the previous chapter, the dictionary doesn't contain a method to join with another dictionary. Let's rectify this situation!

1 Create a new playground, and call it Extensions.

2 Add an extension to `Dictionary` so that it can add to another dictionary.

Extends Dictionary

```
extension Dictionary {
    func add(other:Dictionary)->Dictionary {
        var returnDictionary:Dictionary = self
        for (key,value) in other {
            returnDictionary[key] = value
        }
        return returnDictionary
    }
}
```

Defines new method to extend Dictionary

3 To confirm your new extension works, create two sample dictionaries ready to add together:

```
var somelanguages = ["eng":"English","esp":"Spanish","ita":"Italian"]
var moreLanguages = ["deu":"German","chi":"Chinese","fre":"French"]
```

4 Now use your new method to join the two dictionaries:

```
var languages = somelanguages.add(other:moreLanguages)
```

From now on, whenever you want to join two dictionaries in a project that contains this extension, the `add` method is available to you. Because this method is defined directly on the `Dictionary` structure, you didn't need to define the datatypes of the key and value, making this method available for all `Dictionary` types.

OPEN Compare your code in this section with mine in the Extensions playground.

3.3.3 Operator overloading

I'm not completely happy with the `add` method. It's not intuitive that you're returning the union of the two dictionaries, rather than adding one dictionary directly to the other. I think it would be clearer if you'd used the add (+) operator, the way you can with `Arrays`. Fortunately, Swift makes it possible to define or redefine operators! Redefining functionality for an operator is called *operator overloading*.

The + operator function receives a `left` and `right` parameter and returns a value of the same type.

1. Redefine the `add` method in a `Dictionary` extension as an overloading of the + operator.

```
func +(left: [String:String], right:[String:String]) -> [String:String] {
    var returnDictionary = left
    for (key,value) in right {
        returnDictionary[key] = value
    }
    return returnDictionary
}
```

   Apart from how it's defined, not much has changed from the body of the method. The data types of the key and value need to be specified because you're no longer defining a generic `Dictionary` inside a `Dictionary` extension. Apart from that tweak, the code is similar, and you now can add two `Dictionarys` (with key/value `String/String`) with the plus (+) operator, which is much more intuitive!

2. You'll still need two sample dictionaries to add together:

```
var somelanguages = ["eng":"English","esp":"Spanish","ita":"Italian"]
var moreLanguages = ["deu":"German","chi":"Chinese","fre":"French"]
```

3. Add the two dictionaries together again, but this time use your overloaded add operator:

```
var languages = somelanguages + moreLanguages
```

CHALLENGE Overload the == operator to determine whether two `Distance` objects are equivalent. Tip: The == operator returns a `Bool` value.

3.3.4 Generics

It's a shame, however, that this new overloaded operator will only "operate" on a specific type of `Dictionary`—one with a key that's a `String`, and a value that's a `String`. What if you had another `Dictionary` with a key/value of `Int/String`? You'd need to define an overloaded operator again, for each combination of keys/values! How tiresome.

This is where a concept called *generics* is super useful. A generic can be substituted in a function for any type, but must consistently represent the same type. It turns a function that deals with a specific data type to a generic function that can work with any data type.

Pass in a list of generics between angle brackets <>, after the function or operator name. Like function parameters, generics can be given any name you like.

1. Make the overloaded + operator for adding `Dictionary`s generic for any datatype for key or value.

```
func +<Key,Value>(left: [Key:Value], right:[Key:Value]) -> [Key:Value]
{    var returnDictionary = left
     for (key,value) in right {
         returnDictionary[key] = value
     }
     return returnDictionary
}
```

2. Again, you'll need two sample dictionaries to add together.

```
let somelanguages = ["eng":"English","esp":"Spanish","ita":"Italian"]
let moreLanguages = ["deu":"German","chi":"Chinese","fre":"French"]
```

3. Check your generic method still adds these dictionaries of with a `String` key and `String` value.

```
var languages = somelanguages + moreLanguages
```

   Great, it still works! But will it add dictionaries of another type?

4. Create two sample dictionaries of another type to check. Let's try dictionaries with an `Int` key and `String` value:

```
let someRomanNumerals =
➥[1:"I",5:"V",10:"X",50:"L",100:"C",500:"D",1000:"M"]
let moreRomanNumberals = [1:"I",2:"II",3:"III",4:"IV",5:"V"]
```

5. Confirm your overloaded operator can now join this different type of `Dictionary`.

```
var romanNumerals = someRomanNumerals + moreRomanNumberals
```

Generics are another powerful tool to add to your programmer's arsenal. The Swift team themselves use them to define `Array`s and `Dictionary`s, which is why you didn't

need to define the data type of the `Dictionary` when you extended it. You were already using this powerful feature!

3.4 Summary

In this chapter, you learned the following:

- Use classes or structures to represent types.
- Classes are reference types; structures are value types.
- Use initializers to initialize values.
- Use computed properties as getters and setters.
- Consider protocols to share functionality between classes or structures.
- Use extensions to add functionality to classes and structures.
- Use operator overloading to redefine operators.
- Use generics to make functions more flexible.

Modeling data with enums

Another fascinating data type exists in Swift, called *enumerations*, or *enums*. You may have experience with enumerations in other languages—they're a brilliant way for storing a list of values or states in a type, that a variable of that type could then accept. For example, instead of storing a direction as a string, such as "North", you could create an enum type which defines all possible directions in what are called *cases*.

Those are the basics—but the enums get taken to another level in Swift. Enumerations are types in their own right—they can define initializers and methods or conform to protocols. Enumeration cases in Swift can also optionally be associated with values, which opens up a whole world of possibilities.

All of these features of enums make them an awesome alternative for modeling certain types of data. Tjeerd in 't Veen takes you on a deep dive into many topics in Swift in his book *Swift in Depth*. In the chapter "Modeling data with enums" we'll take a look at using enums for modeling data compared to subclassing and structs.

Chapter 2 from *Swift in Depth*
by Tjeerd in 't Veen

Modeling data with enums

This chapter covers

- You'll see how enums are an alternative to subclassing.
- How to use enums for polymorphism.
- You'll learn how enums are "or" types.
- How to model data with enums instead of structs.
- How enums and structs are algebraic types.
- How to convert structs to enums.
- How to safely handle enums with raw values.
- How to convert strings to enums to create robust code.

Enumerations, or enums for short, are a core tool to use as a Swift developer. Enums allow you to define a type by *enumerating* over its values, such as whether a HTTP method's 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 for each language. Enums in Swift, unlike in C and Objective-C, aren't representations of integer values. Instead, Swift borrows a lot of concepts

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

In fact, I'd argue enums are underused in Swift-land. This chapter hopes to change that and help you see how enums can be surprisingly useful in a number of ways.

You'll see how enums are a suited alternative to subclassing and how enums can help you fit multiple types inside a data structure, such as an array.

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

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

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

After reading this chapter, you may find yourself writing enums more often, ending up with safer and cleaner code in your projects.

2.1 Enums instead of subclassing

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

One of the limitations of modeling your software with hierarchies—e.g., subclassing—is that you're constrained into a specific direction which won't always match your needs.

For example, the aforementioned restaurant has been getting complaints of customers wanting to serve authentic Japanese sushi with their fries. The restaurant intends to accommodate them, but their subclassing model doesn't fit this new requirement.

In an ideal world, it makes sense to model your data hierarchically, but in practice, you'll sometimes hit edge cases and exceptions which may not fit your model.

In this section, we're going to explore the limitations of modeling our data via subclassing in a real-world scenario and solve these with the help of enums.

2.1.1 Forming a model for a workout app

Tip

Join me!

All code from this chapter can be found online. It's more educational and fun if you follow along. You can download the source code at https://github.com/tjeerdintveen/manning-swift-in-depth/tree/master/ch02- enums/

Next up we're building a model layer for a workout app, which tracks runs and cycles for someone. A workout includes the starttime, endtime and a distance.

We'll create a `Run` and a `Cycle` struct that represents the data we're modeling.

Listing 2.1 The Run struct

```
import Foundation // We need foundation for the Date type.

struct Run {
    let id: String
    let startTime: Date
    let endTime: Date
    let distance: Float
    let onRunningTrack: Bool
}
```

Listing 2.2 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 our data layer.

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

2.1.2 Creating a superclass

Many similarities exist between `Run` and `Cycle` which, at first look, make a good candidate for a superclass. The benefit with a superclass is that we can pass the superclass around in our methods and arrays etc. This saves us from creating specific methods and arrays for each workout type.

We could create a superclass called `Workout`, then turn `Run` and `Cycle` into a class and make them subclass `Workout`.

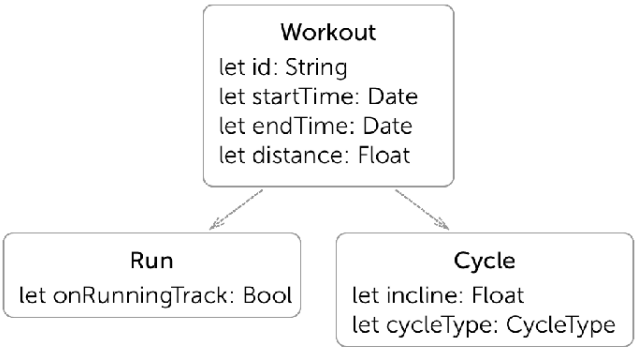


Figure 2.1 A subclassing hierarchy.

Hierarchically, the subclassing structure makes a lot of sense; because workouts share many values. We've a superclass called `Workout` with two subclasses, `Run` and `Cycle`, which inherit from `Workout`.

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

2.1.3 The downsides of subclassing

We'll quickly touch upon issues when it comes down to subclassing.

First of all, we're forced to use classes. Classes can be favorable, but the choice between classes, structs or other enums disappears when subclassing.

> **Note** **Structs and classes**
> We'll cover structs and classes in depth in later chapters.

Being forced to use classes 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. Its superclass `Workout` requires a `startTime`, `endTime` and `distance` value which `Pushups` doesn't need.

Subclassing `Workout` doesn't work because some properties on `Workout` don't apply to `Pushups`.

To allow `Pushups` to subclass `Workout`, we'd have to refactor the superclass and all its subclasses. We'd 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.

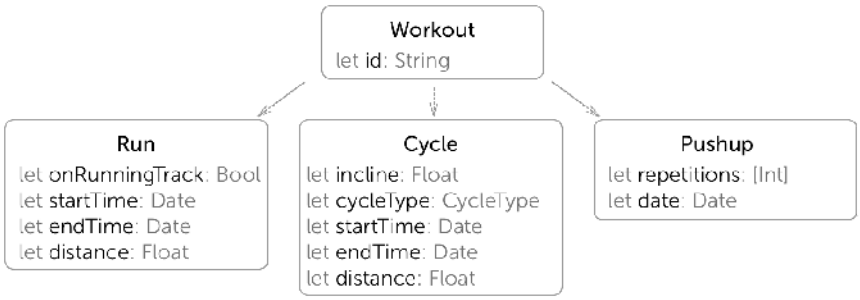


Figure 2.2 **A refactored subclassing hierarchy.**

Refactoring an entire data model shows the issue when subclassing. As soon as we introduce a new subclass, we risk the need to refactor the superclass and all its subclasses. That's a significant impact on existing architecture and a downside of subclassing.

Let's consider an alternate approach involving enums.

2.1.4 Refactoring a data model with enums

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

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

Listing 2.3 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.

We can create a `Workout` by passing it a `Run`, `Cycle` or `Pushups` workout.

For example, we can create a `Pushups` struct that contains the repetitions and the date of the workout. Then we can put this pushups struct inside a `Workout` enum.

Listing 2.4 Creating a workout

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

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

Listing 2.5 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's that we can add new workouts without refactoring existing ones. For example, if we introduce an `Abs` workout, we can add it to `Workout` without touching `Run`, `Cycle` or `Pushups`.

Listing 2.6 Adding a new workout to the Workout enum

```
enum Workout {
    case run(Run)
    case cycle(Cycle)
    case pushups(Pushups)
    case abs(Abs) // New workout introduced.
}
```

Not having to refactor other workouts in order to add a new one's a big benefit and worth considering using enums over subclassing.

2.1.5 Deciding on subclassing or enums

It's not always easy to determine when enums or subclasses fit your data model.

When types share a lot of properties, and if you predict that this won't change in the future, you can get far with classic subclassing. 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), then 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 on all cases in your entire application. Although this may require extra work when adding new cases, it's also 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 datamodel to a fixed manageable amount of cases, then enums can be a good choice.

2.1.6 Exercises

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

2.2 Enums for polymorphism

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

If we mix types in an array, we end up with an Array of type `[Any]`. Such as when we put a `Date`, `String`, and `Int` inside one array.

Listing 2.7 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're an `Any` type.

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

Listing 2.8 Matching on Any values at runtime

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

for element 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")
  }
}
```

We can still figure out what `Any` is at runtime, but we don't know what to expect when matching on an `Any` type; we must also implement a default case to catch the values in which we aren't interested.

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

2.2.1 Compile-time polymorphism

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

Note

What are these <Date> brackets?

Range's a type that represents a lower and upper bound. The `<Date>` notation indicates that `Range` is storing a *generic type*, which we'll explore deeply in a future chapter.

The `Range<Date>` notation tells us that we're working with a range of two `Date` types.

We can create a `DateType` representing either a single date *or* a range of dates. Then we can fill up an array of both a `Date` and `Range<Date>`.

Listing 2.9 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 own associated value.

Listing 2.10 Introducing a DateType enum

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

The array itself consists only out of `DateType` instances. In turn, each `DateType` harbors one of multiple types.

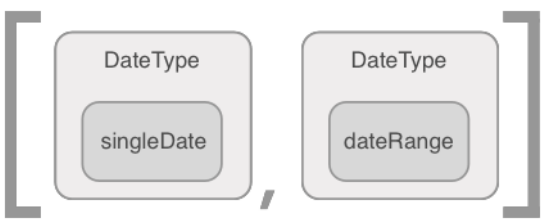


Figure 2.3 Array enums.

Thanks to the enum, we end up with an array containing multiple types, while maintaining compile-time safety.

Namely, if we read values from the array, we could switch on each value.

Listing 2.11 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 us if we ever modify the enum.

By way of illustration, if we added a year case to the enum, the compiler will tell us that we forgot to handle a case.

Listing 2.12 Adding a year case to DateType

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

Year's newly added.

The compiler throws:

Listing 2.13 The compiler notifies us

```
error: Enums.playground:27:5: error: switch must be exhaustive
    switch dateType {
    ^

Enums.playground:27:5: note: add missing case: '.year(_)'
    switch dateType {
```

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

We must beforehand know what kind of cases we expect and want to match against. But, when you know what you're working with, the added compile-time safety's a nice bonus.

2.3 Or versus and

Enums are based on something called *algebraic data types*, which is a term that comes from functional programming languages, where enums are sometimes referred to as *sum types*.

Enums—or sum types—can be thought of as an "or" type. Sum types can only be one thing at once, e.g., A traffic light can either be green *or* yellow *or* red. Or how a die can either be six-sided *or* twenty-sided, but not both at the same time.

On the other end of the spectrum, we've another concept called *product types*. A product type's a type that contains multiple values, such as a class, tuple or struct.

You can think of a product type as an "and" type. E.g., a `User` struct can have both a name *and* an id. Or an address class can have a street *and* a house number *and* a zip code.

2.3.1 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, we'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 or leave message. A message could even be a signal to send balloons across the screen. Because why not, Apple does it in their Messages app.

If we'd list the types of messages that our application supports, we'd have:

- A join message, such as "Mother-in-law has joined the chat."
- A text message that someone can write. Such as "Hello everybody!"
- A send balloons message, which includes some animations and annoying sounds that others see and hear.
- A leave message, such as "Mother-in-law has left the chat."
- When someone's drafting a message, such as "Mike is writing a message."

Let's create a data model to represent messages. Our first idea might be to use a `struct` to model our `Message`. We'll start by doing that and showcase the problems it brings. Then we'll solve these problems by using an enum.

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

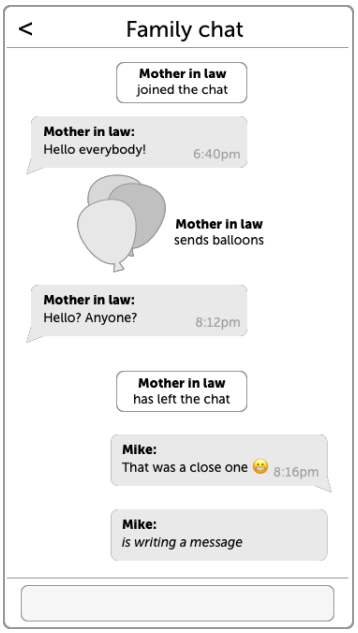


Figure 2.4 A chat application.

Listing 2.14 A join chatroom message

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

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

We can also create a regular text message.

Listing 2.15 A text message

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

In our hypothetical messaging app, we can pass this message data around to other users. Our `Message` struct looks as follows:

Listing 2.16 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 displays a problem. Because a struct can contain multiple values, we can run into bugs where the `Message` struct can both be a text message, as well as a `hasLeft` command as well as 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, we've a message in an invalid state. It represents a text message, but also a join and a leave message.

Listing 2.17 An invalid message with conflicting properties.

```
let brokenMessage = Message(userId: "1",
                            contents: "Hi there", // We 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, it's harder to run into invalid data, but it surely happens often enough in real-world projects. Imagine a `Message` being created from a local file or some function that combines two messages into one. No compile-time guarantees ensure that a message's in the right state.

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

2.3.2 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.

A message can be a join *or* leave message. A message can also be a draft *or* 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 our model by turning it into an enum.

Listing 2.18 Message as an enum (lacking values)

```
import Foundation

enum Message {
    case text
    case draft
    case join
    case leave
    case balloon
}
```

The total sum of variations of `Message` is five cases. Hence why enums are sometimes called *sum* types.

But, we're not there yet, because the cases have no values. We can add values by adding a tuple to each case.

A tuple's an ordered set of values. Such as `(userId: String, contents: String, date: Date)`.

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

Listing 2.19 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 have so-called *associated values* in Swift terms. Also, it's clear which properties belong together.

Now, whenever we want to create a `Message` as an enum, we can pick the proper case with related properties; there's no chance that we'll mix and match the wrong values.

Listing 2.20 Creating enum messages

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

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

Let's say we want to log the messages which have been sent.

Listing 2.21 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")
    }
}
```

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

Listing 2.22 Matching on a single case

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

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

Listing 2.23 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!
}
```

2.3.3 Deciding between structs and enums

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

Also, keep in mind that the associated values of an enum are containers without additional logic. If we had a struct or class, then we'd pass data via an initializer. This

initializer would perform some cleanup work. For instance, trimming the whitespace of a string.

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

2.4 Enums are algebraic data types

Previously, we touched briefly upon sum and product types. Let's use this section to go a little deeper allowing us to reason about enums better.

2.4.1 Algebraic data types

Enums, tuples, and structs are algebraic data types. Algebraic data types revolve around two operations: sum and product. Let's take a closer look at sum and product types before applying it to a data model.

2.4.2 Sum types

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

Listing 2.24 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, we *add* (sum) the possible values of the types inside. In the case of the `Day` enum, this means that the total sum's seven.

Another way to reason about possible values is the `Int8` type. Ranging from -127 to 128, the total number of possible values is 256 (127 + 128 + 1 for including 0). It isn't modeled this way, but you can think of an `Int8` as if it's an enum with 256 cases.

If we were to write an enum with two cases, and we added an `Int8` to one of the cases, then this enum's possible variations jumps from 2 to 257.

For instance, we can have an `Age` enum—representing someone's age—where if the age can be unknown, but if it's known, it contains an `Int8`.

Listing 2.25 The Age enum

```
enum Age {
    case known(Int8)
    case unknown
}
```

There are now 257 possible values can be represented by `Age`. Namely, the unknown case (1) + known case (256).

2.4.3 Product types

On the other end of the spectrum, we have product types. A product type multiplies the possible values of the values it contains.

As an example, if we were to store two booleans inside a struct, the total number of variations is the multiplication—also known as the product—of these two enums.

Listing 2.26 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.

In code this could be proven by showing all variations.

Listing 2.27 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)
```

This is good to keep in mind when modeling data. The higher the number of possible values a type has, the harder it's to reason about.

As a hyperbole, having a struct with 1000 strings for properties has a lot more possible states than a struct with a single boolean property.

2.4.4 Distributing a sum over an enum

Knowing about sum and product types won't stay all theory either. We're not here to write a dry, theoretically-based graduate paper, but to produce beautiful work.

Let's briefly cover a more in-depth look at the algebra behind the theory, and then you'll see how to use this knowledge as a tool to turn structs into enums and back again.

In arithmetic, as an example, we can write two variations of a calculation. We can have a sum (2 + 3) which we multiply by 4.

Listing 2.28 Arithmetic

```
4 * (2 + 3)
```

Alternatively, the multiplication (product) can be distributed over the sums for the same effect, in a different notation.

Listing 2.29 Distributing a multiplication over sums.

```
4 * 2 + 4 * 3
```

Enums, tuples, and structs are algebraic and follow the same principle. E.g., we can distribute a struct over an enum.

Imagine that we've a `PaymentType` enum containing three cases; This enum represents the three ways a customer can pay.

Listing 2.30 Introducing PaymentType

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

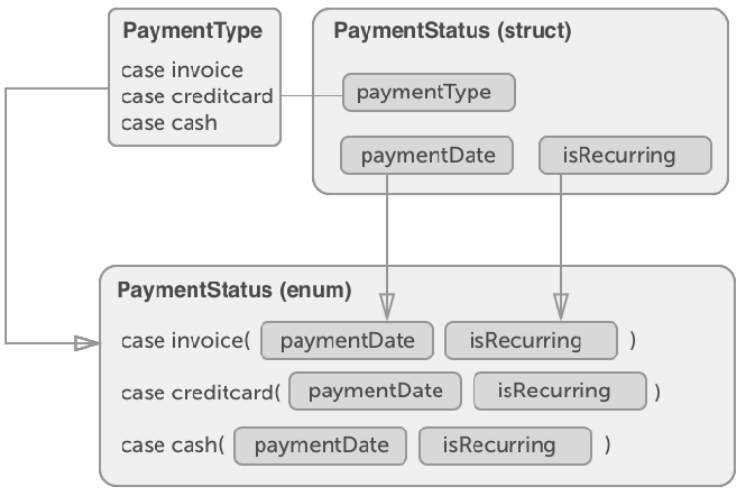
Next, we're going to represent the status of a payment. A struct's a suitable candidate to store some auxiliary properties besides the `PaymentStatus` enum, such as when a payment's completed and if it concerns a recurring payment.

Listing 2.31 A PaymentStatus enum

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

The product of all the variations are: All possible dates *times* 2 (boolean) *times* 3 (enum with three cases). This is a high number.

Like cream cheese on a bagel, we'll smear the properties of the struct out over the cases of the enum by following the rules of algebraic data types.



We 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 2.32 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's still there, and the number of possible variations is still the same. Except for this time, we flipped the types inside out!

As a benefit, we're only dealing with a single type; the price is repetition inside each case. Right or wrong isn't the issue; it's 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 us 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.

2.4.5 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's the number of possible variations of Bagel?
4. Turn Bagel into an enum while keeping the same amount of possible variations.
5. Given this enum representing a Puzzle game for a certain age range (baby, toddler, etc.) 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 could this enum be represented as a struct instead?

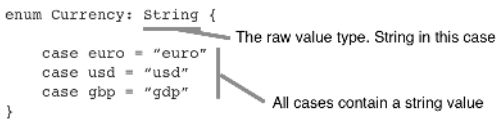
2.5 A safer use of strings

Dealing with strings and enums is quite common. Let's go ahead and pay some extra attention to them to do it correctly.

This section highlights some dangers when dealing with enums that hold a `String` raw value.

When an enums defined as a raw value type, then 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.



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 are values which are attached to enum's cases at compile time. In contrast, enums with associated types—which we'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. We don't need to add a string value if the `rawValue` is the same as the case name.

Listing 2.33 Enum with raw values, with string values omitted.

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

Because 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

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

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

We'll use the `Currency` enum to construct parameters for our API call. We can read the enum's raw value by accessing the raw value property and set up our 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, we'll change the `rawValue` of the `euro` case, from "euro" to "eur" (dropping the "o"), because *eur* is the currency notation of the euro.

Listing 2.35 Renaming a string

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

The problem, because the API call relied on the `rawValue` to creating our parameters, is that the parameters are now affected for the API call.

The compiler won't notify us, because the raw value's still valid code.

Listing 2.36 Unexpected parameters

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

Everything still compiles, but we silently introduced a bug in a part of our application.

It may sound obvious to make sure to update a string everywhere. 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 somewhere else in your application. These issues can sneak up on you, and it's easy to miss the newly introduced bugbecausewe don't know it happens at compile time.

We can play it safe and ignore an enum's raw values and match on the enum cases itself. When we set the parameters this way, we'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"]
}

// We're back to using "euro" again
print(parameters) // ["filter": "euro"]
```

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

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

These are all trade-off you'll have to make, but it's good to be extra conscious of the fact that you *lose* help from the compiler once you start using raw values from an enum.

2.5.2 Matching on Strings

Whenever we pattern match on a string, we're opening the door to missed cases. This section covers the downsides of matching on strings and showcase how to make an enum out of it for added safety.

In the next example, we'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, we 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, we'll only look at the extension.

The `iconName` function gives our application the name of the icon to display over an image, based on the file extension. E.g., a jpeg image has a little icon shown on it, this icon's name will be "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 there are a couple of problems with this approach, versus matching on enums.

It's easy to make a typo and 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 we 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'd repeatedly match on the same string, the chances of typos increase.

Also, if any bugs are introduced by matching on strings, we'll know it at *runtime*, but switching on enums are exhaustive. If we switched on an enum instead, we'd know about bugs (e.g., forgetting to handle a case) at compile time.

Let's create an enum out of it! We do this by introducing an enum with a `String` raw type, as indicated by `enum ImageType: String`.

Listing 2.40 Creating an enum with a String raw value

```
// Introducing the enum
enum ImageType: String {
  case jpg case bmp
  case gif
}
```

This time when we match in the `iconName` function, we turn the string into an enum first by passing a `rawValue`. This way we'll know if `ImageType` gets another case added to it. The compiler tells us 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 us compiler benefits if we missed a case.

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

Let's take care of that now by matching on multiple strings at once.

We can implement our own initializer which accepts a rawvalue 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's now case-insensitive, making it more forgiving.

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

Note

Optional init?

The initializer from `ImageType` returns an optional. An optional initializer indicates that it can fail. When the initializer fails—when we give it an unusable string—the initializer returns a nil value. Not to worry if this isn't clear yet, we'll handle optionals in-depth in a future chapter.

A couple of things to note here. We set the ImageType case depending on its passed rawValue, but not before turning it into a lowercased string to make the pattern matching case-insensitive.

Next, we give each case multiple options to match on. Such as `case "jpg", "jpeg"` to catch more cases. We could write it out by using more cases, but this is a clean way to group pattern matching.

Now our string matching's more robust and we 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 we have a bug in the conversion, we 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's now more idiomatic, the code has been made safer and more expressive. The tradeoff's that a new enum has to be created, which may be redundant if you pattern match on a string only once.

As soon as you see code matching on a string repeatedly, converting it to an enum's 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?

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.

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

In fact, perhaps next time as a pet project, see how far you can get by using only enums and structs. It'll be an excellent workout to think in *sum* and *product* types!

2.6 Summary

In this chapter, you learned that

- 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.
- Enums can be used 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.

Great job, you survived the first chapter! I hope you've got a share of "aha" moments already. There's a lot more to come; let's quickly move on.

2.7 Answers

1 A superclass prevents duplication, no need to declare the same property twice. With subclassing, you can also override existing functionality.
2 No need to refactor anything if you add another type. Whereas with subclassing you risk refactoring a superclass and its existing subclasses. Second, you aren't forced to use classes.

3 What are the number possible variations of Bagel? Twelve. Three for Topping times four for BagelType.

4 There are two ways, because Bagel contains two enums. You can pick which enum to store the data in.

Listing 2.44 Bagel as enum, two variations

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

// Alternatively, we can use the BagelType enum as the enum's cases.
enum Bagel {
    case cinnamonRaisin(Topping)
    case glutenFree(Topping)
    case oatMeal(Topping)
    case blueberry(Topping)
}
```

5 How could this enum be represented as a struct instead?

Listing 2.45 Puzzle as struct

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

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

6 String, Character and integer and floating-point types.

7 Raw type values are determined at compile time.

8 Associated values are set at runtime.

9 All types fit inside an associated value.

Graph problems

Finally, let's take a look at how we can apply what we've learned about Swift with a sample from the book *Classic Computer Science Problems in Swift*. In the chapter "Graph problems", author David Kopec introduces us to the problem of graphing networks—such as a train network—with vertices and edges (think train stations and train-lines).

David looks at implementing a data structure to represent such networks in code and then explores solving related problems, such as finding the shortest path between two stations, or minimizing the track needed to connect all the stations, using algorithms discovered 100 years ago!

This example doesn't only apply to train-networks. Plenty of real world examples exist of networks that this sort of discussion could apply to, and several are discussed at the end of the chapter. Beyond the specifics of this topic, I think it's fascinating to examine the process of how to approach complex problems such as these, a worthwhile exercise for those wanting to further their adventure in Swift!

Chapter 4 from *Classic Computer Science Problems in Swift* by David Kopec

Graph problems

A *graph* is an abstract mathematical construct that is used for modeling a real-world problem by dividing the problem into a set of connected nodes. We call each of the nodes a *vertex* and each of the connections an *edge*. For instance, a subway map can be thought of as a graph representing a transportation network. Each of the dots represents a station, and each of the lines represents a route between two stations. In graph terminology, we would call the stations "vertices" and the routes "edges."

Why is this useful? Not only do graphs help us abstractly think about a problem, they also let us apply several well-understood and performant search and optimization techniques. For instance, in the subway example, suppose we want to know the shortest route from one station to another. Or, suppose we wanted to know the minimum amount of track needed to connect all of the stations. Graph algorithms that you will learn in this chapter can solve both of those problems. Further, graph algorithms can be applied to any kind of network problem—not just transportation networks. Think of computer networks, distribution networks, and utility networks. Search and optimization problems across all of these spaces can be solved using graph algorithms.

In this chapter, we won't work with a graph of subway stations, but instead cities of the United States and potential routes between them. Figure 4.1 is a map of the continental United States and the fifteen largest metropolitan statistical areas (MSAs) in the country, as estimated by the U.S. Census Bureau.[1]

Famous entrepreneur Elon Musk has suggested building a new high-speed transportation network composed of capsules traveling in pressurized tubes. According to Musk, the capsules would travel at 700 miles per hour and be suitable

[1] Data from the United States Census Bureau's American Fact Finder, https://factfinder.census.gov/.

Figure 4.1 **A map of the 15 largest MSAs in the United States**

for cost-effective transportation between cities less than 900 miles apart.[2] He calls this new transportation system the "Hyperloop." In this chapter we will explore classic graph problems in the context of building out this transportation network.

Musk initially proposed the Hyperloop idea for connecting Los Angeles and San Francisco. If one were to build a national Hyperloop network, it would make sense to do so between America's largest metropolitan areas. In figure 4.2 the state outlines from figure 4.1 are removed. In addition, each of the MSAs is connected with some of its neighbors (not always its nearest neighbors, to make the graph a little more interesting).

Figure 4.2 is a graph with vertices representing the 15 largest MSAs in the United States and edges representing potential Hyperloop routes between cities. The routes were chosen for illustrative purposes. Certainly other potential routes could be part of a new Hyperloop network.

This abstract representation of a real-world problem highlights the power of graphs. Now that we have an abstraction to work with, we can ignore the geography of the United States and concentrate on thinking about the potential Hyperloop network simply in the context of connecting cities. In fact, as long as we keep the edges the same, we can think about the problem with a different looking graph. In figure 4.3, the location of Miami has moved. The graph in figure 4.3, being an abstract representation, can still address the same fundamental computational problems as the graph in figure 4.2, even if Miami is not where we would expect it. But for our sanity, we will stick with the representation in figure 4.2.

[2] Elon Musk, "Hyperloop Alpha," http://mng.bz/chmu.

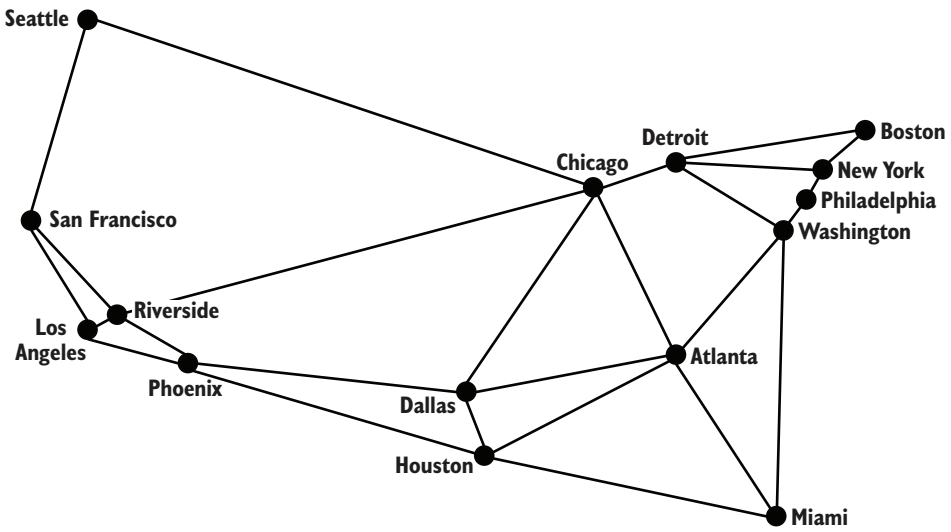


Figure 4.2 A graph with the vertices representing the 15 largest MSAs in the United States and the edges representing potential Hyperloop routes between them

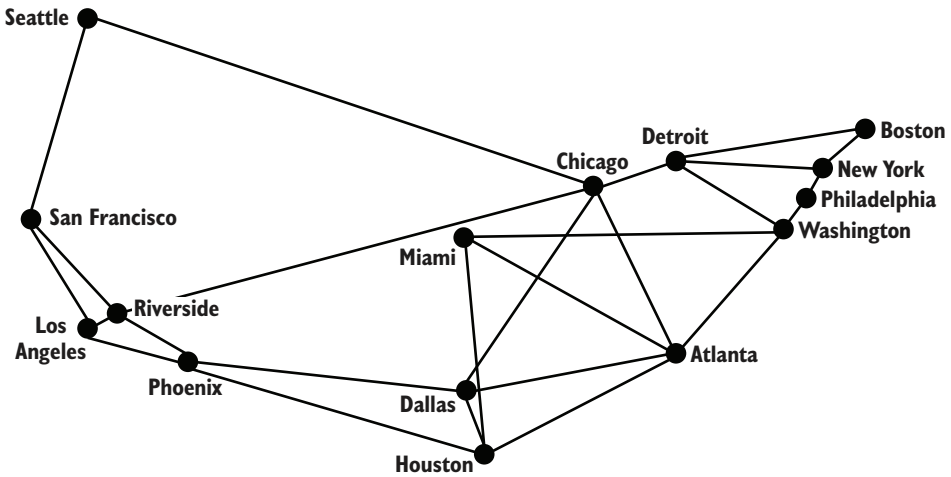


Figure 4.3 An equivalent graph to that in figure 4.2, with the location of Miami moved

4.1 Building a graph framework

Swift has been promoted as enabling a protocol-oriented style of programming (as opposed to the traditional object-oriented or functional paradigms).[3] Although the orthodoxy of this new paradigm is still being fleshed-out, what is clear is that it puts interfaces and composition ahead of inheritance. Whereas the class is the fundamental

[3] Dave Abrahams, "Protocol-Oriented Programming in Swift," WWDC 2015, Session 408, Apple Inc., http://mng.bz/zWP3.

building block in the object-oriented paradigm, and the function is the fundamental building block in functional programming, the protocol is the fundamental building block in protocol-oriented programming. In that light, we will try building a graph framework in a protocol-first style.

> **NOTE** The framework described in this section, and the examples that follow it, are largely based on a simplified version of my SwiftGraph open source project (https://github.com/davecom/SwiftGraph). SwiftGraph includes several features that go beyond the scope of this book.

We want this graph framework to be as flexible as possible, so that it can represent as many different problems as possible. To achieve this goal, we will use generics to abstract away the type of the vertices, and we will define an easy-to-adopt protocol for edges. Every vertex will ultimately be assigned an integer index, but it will be stored as the user-defined generic type.

Let's start work on the framework by defining the `Edge` protocol.

```
public protocol Edge: CustomStringConvertible {
    var u: Int { get set } // index of the "from" vertex
    var v: Int { get set } // index of the "to" vertex
    var reversed: Edge { get }
}
```

An `Edge` is defined as a connection between two vertices, each of which is represented by an integer index. By convention, `u` is used to refer to the first vertex, and `v` is used to represent the second vertex. You can also think of `u` as "from" and `v` as "to." In this chapter, we are only working with bidirectional edges (edges that can be travelled in both directions), but in *directed graphs*, also known as *digraphs*, edges can also be one-way, and the `reversed` property is meant to return an `Edge` that travels in the opposite direction. All `Edge` adoptees must implement `CustomStringConvertible` so they can be easily printed to the console.

The `Graph` protocol is about the essential role of a graph: associating vertices with edges. Again, we want to let the actual types of the vertices and edges be whatever the user of the framework desires. This lets the framework be used for a wide range of problems without needing to make intermediate data structures that glue everything together. In this light, we will use the Swift keyword `associatedtype` to define types that adopters of `Graph` can configure. For example, in a graph like the one for Hyperloop routes, we might define `VertexType` to be `String`, because we would use strings like "New York" and "Los Angeles" as the vertices. The only requirement of a potential `VertexType` is that it implements `Equatable`. `String` implements `Equatable`, so it is a valid `VertexType`.

```
protocol Graph: class, CustomStringConvertible {
    associatedtype VertexType: Equatable
    associatedtype EdgeType: Edge
    var vertices: [VertexType] { get set }
    var edges: [[EdgeType]] { get set }
}
```

The `vertices` array can be an array of any type that adopts `Equatable`. Each vertex will be stored in the array, but we will later refer to them by their integer index in the array. The vertex itself may be a complex data type, but its index will always be an `Int`, which is easy to work with. On another level, by putting this index between graph algorithms and the `vertices` array, it allows us to have two vertices that are equal in the same graph (imagine a graph with a country's cities as vertices, where the country has more than one city named "Springfield"). Even though they are the same, they will have different integer indexes.

There are many ways to implement a graph data structure, but the two most common are to use a *vertex matrix* or *adjacency lists*. In a vertex matrix, each cell of the matrix represents the intersection of two vertices in the graph, and the value of that cell indicates the connection (or lack thereof) between them. Our graph data structure uses adjacency lists. In this graph representation, every vertex has an array (or list) of vertices that it is connected to. Our specific representation uses an array of arrays of edges, so for every vertex there is an array of edges via which the vertex is connected to other vertices. `edges` is this two-dimensional array.

Notice, as well, that anything that adopts `Graph` must also adopt `class` and `CustomStringConvertible`. We want graph data structures to be reference types for memory-management purposes. It will also be slightly easier to write some of the protocol extensions if we know the adopters will be classes. `class` ensures that all graphs adopters are classes. `CustomStringConvertible` forces adopters of the protocol to be printable.

Introduced in Swift 2, protocol extensions allow fully fleshed out functions to be a part of a protocol. Amazingly, this will allow us to implement most of the functionality a graph needs before we actually define a concrete adopter of `Graph`. The following code shows the entirety of the protocol extension that adds this basic functionality, with in-source comments describing each of the functions.

```
extension Graph {
    /// How many vertices are in the graph?
    public var vertexCount: Int { return vertices.count }

    /// How many edges are in the graph?
    public var edgeCount: Int { return edges.joined().count }

    /// Get a vertex by its index.
    ///
    /// - parameter index: The index of the vertex.
    /// - returns: The vertex at i.
    public func vertexAtIndex(_ index: Int) -> VertexType {
        return vertices[index]
    }

    /// Find the first occurrence of a vertex if it exists.
    ///
    /// - parameter vertex: The vertex you are looking for.
    /// - returns: The index of the vertex. Return nil if it can't find it.
    public func indexOfVertex(_ vertex: VertexType) -> Int? {
```

```
    if let i = vertices.index(of: vertex) {
        return i
    }
    return nil
}

/// Find all of the neighbors of a vertex at a given index.
///
/// - parameter index: The index for the vertex to find the neighbors of.
/// - returns: An array of the neighbor vertices.
public func neighborsForIndex(_ index: Int) -> [VertexType] {
    return edges[index].map({self.vertices[$0.v]})
}

/// Find all of the neighbors of a given Vertex.
///
/// - parameter vertex: The vertex to find the neighbors of.
/// - returns: An optional array of the neighbor vertices.
public func neighborsForVertex(_ vertex: VertexType) -> [VertexType]? {
    if let i = indexOfVertex(vertex) {
        return neighborsForIndex(i)
    }
    return nil
}

/// Find all of the edges of a vertex at a given index.
///
/// - parameter index: The index for the vertex to find the children of.
public func edgesForIndex(_ index: Int) -> [EdgeType] {
    return edges[index]
}

/// Find all of the edges of a given vertex.
///
/// - parameter vertex: The vertex to find the edges of.
public func edgesForVertex(_ vertex: VertexType) -> [EdgeType]? {
    if let i = indexOfVertex(vertex) {
        return edgesForIndex(i)
    }
    return nil
}

/// Add a vertex to the graph.
///
/// - parameter v: The vertex to be added.
/// - returns: The index where the vertex was added.
public func addVertex(_ v: VertexType) -> Int {
    vertices.append(v)
    edges.append([EdgeType]())
    return vertices.count - 1
}

/// Add an edge to the graph.
```

```
    ///
    /// - parameter e: The edge to add.
    public func addEdge(_ e: EdgeType) {
        edges[e.u].append(e)
        edges[e.v].append(e.reversed as! EdgeType)
    }
}
```

Let's step back for a moment and consider why this protocol has two versions of most of its functions. We know from the protocol definition that the array `vertices` is an array of elements of type `VertexType`, which can be anything that implements `Equatable`. So we have vertices of type `VertexType` that are stored in the `vertices` array. But if we want to retrieve or manipulate them later, we need to know where they are stored in that array. Hence, every vertex has an index in the array (an integer) associated with it. If we don't know a vertex's index, we need to look it up by searching through `vertices`. That is why there are two versions of every function. One operates on `Int` indexes, and one operates on `VertexType` itself. The functions that operate on `VertexType` look up the relevant indices and call the index-based function.

Most of the functions are fairly self-explanatory, but `neighborsForIndex()` deserves a little unpacking. It returns the *neighbors* of a vertex. A vertex's neighbors are all of the other vertices that are directly connected to it by an edge. For example, in figure 4.2, New York and Washington are neighbors (the only neighbors) of Philadelphia. We find the neighbors for a vertex by looking at the ends (the vs) of all of the edges going out from it.

```
public func neighborsForIndex(_ index: Int) -> [VertexType] {
    return edges[index].map({self.vertices[$0.v]})
}
```

`edges[index]` is the adjacency list, the list of edges through which the vertex in question is connected to other vertices. In the closure of the map call, `$0` represents one particular edge, and `$0.v` represents the neighbor that the edge is connected to. `map()` will return all of the vertices (as opposed to just their indices), because `$0.v` is passed as an index into the `vertices` array.

Another important thing to note is the way `addEdge()` works. `addEdge()` first adds an edge to the adjacency list of the "from" vertex (`u`), and then adds a reversed version of itself to the adjacency list of the "to" vertex (`v`). The second step is necessary because this graph is not directed. We want every edge added to be bidirectional—that means that `u` will be a neighbor of `v` in the same way that `v` is a neighbor of `u`.

```
public func addEdge(_ e: EdgeType) {
    edges[e.u].append(e)
    edges[e.v].append(e.reversed as! EdgeType)
}
```

4.1.1 A concrete implementation of Edge

As was mentioned earlier, we are only dealing with bidirectional edges in this chapter. Beyond being bidirectional or unidirectional, edges can also be *unweighted* or *weighted*. A weighted edge is one that has some comparable value (usually numeric, but not always) associated with it. We could think of the weights in our potential Hyperloop network as being the distances between the stations. For now, though, we will deal with an unweighted version of the graph. An unweighted edge is simply a connection between two vertices. Another way of putting it is that in an unweighted graph we know which vertices are connected, whereas in a weighted graph we know which vertices are connected and we know something about those connections.

Our implementation of an unweighted edge, `UnweightedEdge`, will of course implement the `Edge` protocol. It must have a place for a "from" vertex (`u`), a place for a "to" vertex (`v`), and a way to reverse itself. It also must implement `CustomStringConvertible`, as required by `Edge`, which means having a `description` property.

```
open class UnweightedEdge: Edge {
    public var u: Int // "from" vertex
    public var v: Int // "to" vertex
    public var reversed: Edge {
        return UnweightedEdge(u: v, v: u)
    }

    public init(u: Int, v: Int) {
        self.u = u
        self.v = v
    }

    //MARK: CustomStringConvertable
    public var description: String {
        return "\(u) <-> \(v)"
    }
}
```

4.1.2 A concrete implementation of Graph

`UnweightedEdge` is pretty simple. Surprisingly, so is our concrete implementation of `Graph`. An `UnweightedGraph` is a `Graph` whose vertices can be any `Equatable` type (as per the `Graph` protocol) and whose edges are of type `UnweightedEdge`. By defining the types of the `vertices` and `edges` arrays, we are implicitly filling in the associated types `VertexType` and `EdgeType` in the `Graph` protocol.

```
open class UnweightedGraph<V: Equatable>: Graph {
    var vertices: [V] = [V]()
    var edges: [[UnweightedEdge]] = [[UnweightedEdge]]() //adjacency lists

    public init() {
    }

    public init(vertices: [V]) {
```

```
        for vertex in vertices {
            _ = self.addVertex(vertex)
        }
    }

    /// This is a convenience method that adds an unweighted edge.
    ///
    /// - parameter from: The starting vertex's index.
    /// - parameter to: The ending vertex's index.
    public func addEdge(from: Int, to: Int) {
        addEdge(UnweightedEdge(u: from, v: to))
    }

    /// This is a convenience method that adds an unweighted, undirected
        ➥ edge between the first occurrence of two vertices.
    ///
    /// - parameter from: The starting vertex.
    /// - parameter to: The ending vertex.
    public func addEdge(from: V, to: V) {
        if let u = indexOfVertex(from) {
            if let v = indexOfVertex(to) {
                addEdge(UnweightedEdge(u: u, v: v))
            }
        }
    }

    /// MARK: Implement CustomStringConvertible
    public var description: String {
        var d: String = ""
        for i in 0..<vertices.count {
            d += "\(vertices[i]) -> \(neighborsForIndex(i))\n"
        }
        return d
    }
}
```

The new abilities in `UnweightedGraph` are init methods, convenience methods for adding `UnweightedEdges` to the graph, and the property `description` for conformance with `CustomStringConvertible`.

Now that we have concrete implementations of `Edge` and `Graph` we can actually create a representation of the potential Hyperloop network. The vertices and edges in `cityGraph` correspond to the vertices and edges represented in figure 4.2.

```
var cityGraph: UnweightedGraph<String>
➥ = UnweightedGraph<String>(vertices: ["Seattle", "San
➥ Francisco", "Los Angeles", "Riverside", "Phoenix", "Chicago",
➥ "Boston", "New York", "Atlanta", "Miami", "Dallas", "Houston",
➥ "Detroit", "Philadelphia", "Washington"])

cityGraph.addEdge(from: "Seattle", to: "Chicago")
cityGraph.addEdge(from: "Seattle", to: "San Francisco")
cityGraph.addEdge(from: "San Francisco", to: "Riverside")
cityGraph.addEdge(from: "San Francisco", to: "Los Angeles")
```

```
cityGraph.addEdge(from: "Los Angeles", to: "Riverside")
cityGraph.addEdge(from: "Los Angeles", to: "Phoenix")
cityGraph.addEdge(from: "Riverside", to: "Phoenix")
cityGraph.addEdge(from: "Riverside", to: "Chicago")
cityGraph.addEdge(from: "Phoenix", to: "Dallas")
cityGraph.addEdge(from: "Phoenix", to: "Houston")
cityGraph.addEdge(from: "Dallas", to: "Chicago")
cityGraph.addEdge(from: "Dallas", to: "Atlanta")
cityGraph.addEdge(from: "Dallas", to: "Houston")
cityGraph.addEdge(from: "Houston", to: "Atlanta")
cityGraph.addEdge(from: "Houston", to: "Miami")
cityGraph.addEdge(from: "Atlanta", to: "Chicago")
cityGraph.addEdge(from: "Atlanta", to: "Washington")
cityGraph.addEdge(from: "Atlanta", to: "Miami")
cityGraph.addEdge(from: "Miami", to: "Washington")
cityGraph.addEdge(from: "Chicago", to: "Detroit")
cityGraph.addEdge(from: "Detroit", to: "Boston")
cityGraph.addEdge(from: "Detroit", to: "Washington")
cityGraph.addEdge(from: "Detroit", to: "New York")
cityGraph.addEdge(from: "Boston", to: "New York")
cityGraph.addEdge(from: "New York", to: "Philadelphia")
cityGraph.addEdge(from: "Philadelphia", to: "Washington")
```

`cityGraph` has vertices of type `String`, and we indicate each vertex with the name of the MSA that it represents. It is irrelevant in what order we add the edges to `cityGraph`. Because we implemented `CustomStringConvertible` in `UnweightedGraph` with a nicely printed description of the graph, we can now pretty-print (that's a real term!) the graph.

```
print(cityGraph)
```

You should get output similar to the following:

```
Seattle -> ["Chicago", "San Francisco"]
San Francisco -> ["Seattle", "Riverside", "Los Angeles"]
Los Angeles -> ["San Francisco", "Riverside", "Phoenix"]
Riverside -> ["San Francisco", "Los Angeles", "Phoenix", "Chicago"]
Phoenix -> ["Los Angeles", "Riverside", "Dallas", "Houston"]
Chicago -> ["Seattle", "Riverside", "Dallas", "Atlanta", "Detroit"]
Boston -> ["Detroit", "New York"]
New York -> ["Detroit", "Boston", "Philadelphia"]
Atlanta -> ["Dallas", "Houston", "Chicago", "Washington", "Miami"]
Miami -> ["Houston", "Atlanta", "Washington"]
Dallas -> ["Phoenix", "Chicago", "Atlanta", "Houston"]
Houston -> ["Phoenix", "Dallas", "Atlanta", "Miami"]
Detroit -> ["Chicago", "Boston", "Washington", "New York"]
Philadelphia -> ["New York", "Washington"]
Washington -> ["Atlanta", "Miami", "Detroit", "Philadelphia"]
```

4.2 Finding the shortest path

The Hyperloop is so fast that, for optimizing travel time from one station to another, it probably matters less how long the distances are between the stations and more how many hops it takes (how many stations need to be visited) to get from one station to another. Each station may involve a layover, so just like with flights, the fewer stops the better.

In graph theory, a set of edges that connects two vertices is known as a *path*. In other words, a path is a way of getting from one vertex to another vertex. In the context of the Hyperloop network, a set of tubes (edges) represents the path from one city (vertex) to another (vertex). Finding optimal paths between vertices is one of the most common problems that graphs are used for.

4.2.1 Defining a path

In our graphs, a path can simply be thought of as an array of edges.

```
public typealias Path = [Edge]
```

Every `Edge` knows the index of its "from" vertex (`u`) and its "to" vertex (`v`), so given a `Graph`, it is easy to deduce the vertices that it connects. There's a method in `Graph` for that, `vertexAtIndex()`. It would be nice to have a method to pretty-print a `Path` within a `Graph`. We can do that in a short extension to `Graph`.

```
extension Graph {
    /// Prints a path in a readable format
    public func printPath(_ path: Path) {
        for edge in path {
            print("\(vertexAtIndex(edge.u)) > \(vertexAtIndex(edge.v))")
        }
    }
}
```

4.2.2 Revisiting breadth-first search (BFS)

In an unweighted graph, finding the shortest path means finding the path that has the fewest edges between the starting vertex and the destination vertex. To build out the Hyperloop network, it might make sense to first connect the furthest cities on the highly populated seaboards. That raises the question, "what is the shortest path between Boston and Miami?"

Luckily, we already know an algorithm for finding shortest paths, and we can reuse it to answer this question. Breadth-first search, introduced in chapter 2, is just as viable for graphs as it is for mazes. In fact, the mazes we worked with in chapter 2 really are graphs. The vertices are the locations in the maze, and the edges are the moves that can be made from one location to another. In an unweighted graph, a breadth-first search will find the shortest path between any two vertices.

We can rewrite the breadth-first search implementation from chapter 2 to suit working with `Graph`. We can even reuse the same `Queue` class, unchanged.

```
public class Queue<T> {
    private var container: [T] = [T]()
    public var isEmpty: Bool { return container.isEmpty }
    public func push(_ thing: T) { container.append(thing) }
    public func pop() -> T { return container.removeFirst() }
}
```

The new version of `bfs()` will be an extension to `Graph`. It will no longer operate on `Node`s, as in chapter 2, but instead on vertices, referred to by their indices (`Int`s). Recall from chapter 2 that we used the `Node` class to keep track of the parent of each new `Node` we found. There was also a function, `nodeToPath()`, that used the `parent` property of each node to generate a path from the goal back to the start node (but reversed to start at the start). We will use a similar function, `pathDictToPath()`, to generate a `Path` from our starting vertex to the destination vertex.

```
/// Takes a dictionary of edges to reach each node and returns an array
    ➥ of edges
/// that goes from `from` to `to`
public func pathDictToPath(from: Int, to: Int, pathDict:
➥ [Int: Edge]) -> Path {
    if pathDict.count == 0 {
        return []
    }
    var edgePath: Path = Path()
    var e: Edge = pathDict[to]!
    edgePath.append(e)
    while (e.u != from) {
        e = pathDict[e.u]!
        edgePath.append(e)
    }
    return Array(edgePath.reversed())
}
```

In the new version of `bfs()`, in lieu of having access to the `parent` property on `Node`, we will use a dictionary associating each vertex index with the `Edge` that got us to it. This is what we will call `pathDict`. `pathDictToPath()` extrapolates from this dictionary the `Path` that connects the `from` vertex to the `to` vertex by looking at every `Edge` between `to` and `from` in `pathDict`.

As you study the implementation of `bfs()` on `Graph`, it may be helpful to flip back to the implementation of `bfs()` you are already familiar with from chapter 2. How has it changed? What has stayed the same? All of the basic machinery, aside from `pathDict`, is essentially the same, but several of the parameter types and generic types have been modified.

```
extension Graph {
    //returns a path to the goal vertex
    func bfs(initialVertex: VertexType, goalTestFn:
    ➥ (VertexType) -> Bool) -> Path? {
```

```
        guard let startIndex = indexOfVertex(initialVertex)
        ➥ else { return nil }
        // frontier is where we've yet to go
        let frontier: Queue<Int> = Queue<Int>()
        frontier.push(startIndex)
        // explored is where we've been
        var explored: Set<Int> = Set<Int>()
        explored.insert(startIndex)
        // how did we get to each vertex
        var pathDict: [Int: EdgeType] = [Int: EdgeType]()
        // keep going while there is more to explore
        while !frontier.isEmpty {
            let currentIndex = frontier.pop()
            let currentVertex = vertexAtIndex(currentIndex)
            // if we found the goal, we're done
            if goalTestFn(currentVertex) {
                return pathDictToPath(from: startIndex, to: currentIndex,
                ➥ pathDict: pathDict)
            }
            // check where we can go next and haven't explored
            for edge in edgesForIndex(currentIndex)
            ➥ where !explored.contains(edge.v) {
                explored.insert(edge.v)
                frontier.push(edge.v)
                pathDict[edge.v] = edge
            }
        }
        return nil // never found the goal
    }
}
```

The new `bfs()` takes a starting vertex, `initialVertex`, a function that will determine if the goal is reached, `goalTestFn()`, and returns an optional `Path`. The returned optional `Path` will be `nil` if `initialVertex` is not actually in the `Graph` (this is determined by the `guard` statement). It will also return `nil` if `goalTestFn()` never returns `true` for any of the searched vertices in the graph. `frontier` and `explored` are much the same as they were in chapter 2, except that now the generic type of each is set to `Int`—the index of a vertex in a `Graph`. This version of `bfs()` has no `successorFn()`. Instead, `edgesForIndex()` brings the next unexplored vertices onto the frontier. Finally, the last main difference between this version and the prior one is the use of `pathDict`, which gets updated when a new vertex is added to the queue, and which is used to return the final `Path` when the goal is found by calling `pathDictToPath()`.

We are now ready to find the shortest path (in terms of number of edges) between Boston and Miami. We can pass a closure to `bfs()` that tests for a goal of a vertex equivalent to the `String` `"Miami"`. If a `Path` is found, we can print it using the `printPath()` method introduced earlier as a protocol extension to `Graph`.

```
if let bostonToMiami = cityGraph.bfs(initialVertex: "Boston",
➥ goalTestFn: { $0 == "Miami" }) {
    cityGraph.printPath(bostonToMiami)
}
```

The output should look something like this:

```
Boston > Detroit
Detroit > Washington
Washington > Miami
```

Boston to Detroit to Washington to Miami, composed of three edges, is the shortest route between Boston and Miami in terms of number of edges. Figure 4.4 highlights this route.

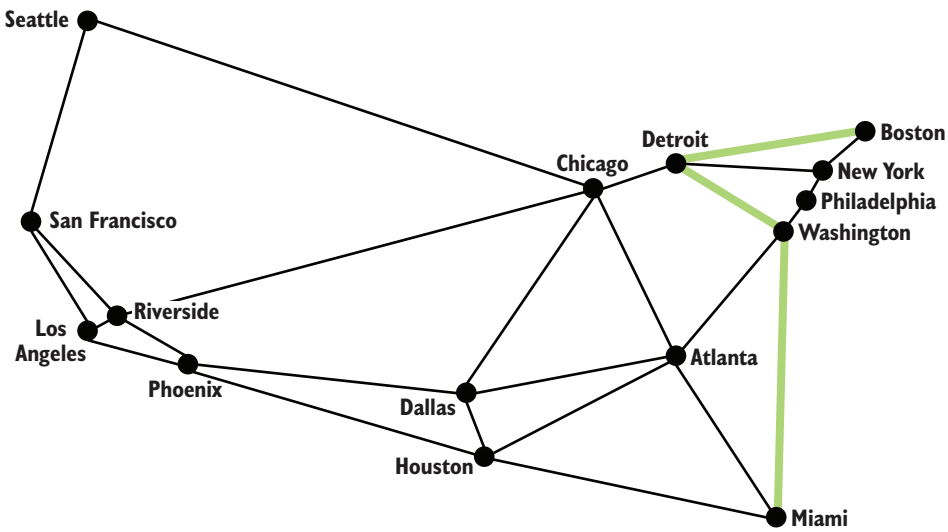


Figure 4.4 **The shortest route between Boston and Miami, in terms of number of edges, is highlighted.**

4.3 Minimizing the cost of building the network

Imagine we want to connect all 15 of the largest MSAs to the Hyperloop network. Our goal is to minimize the cost of rolling out the network, so that means using a minimum of track. The question is then, "how can we connect all of the MSAs using the minimum amount of track?"

4.3.1 Workings with weights

To understand the amount of track that a particular edge may require, we need to know the distance that the edge represents. This is an opportunity to re-introduce the concept of weights. In the Hyperloop network, the weight of an edge is the distance between the two MSAs that it connects. Figure 4.5 is the same as figure 4.2, except it has a weight added to each edge, representing the distance in miles between the two vertices that the edge connects.

To handle weights, we will need a new implementation of `Edge` and a new implementation of `Graph`. Once again, we want to design our framework in as flexible a way

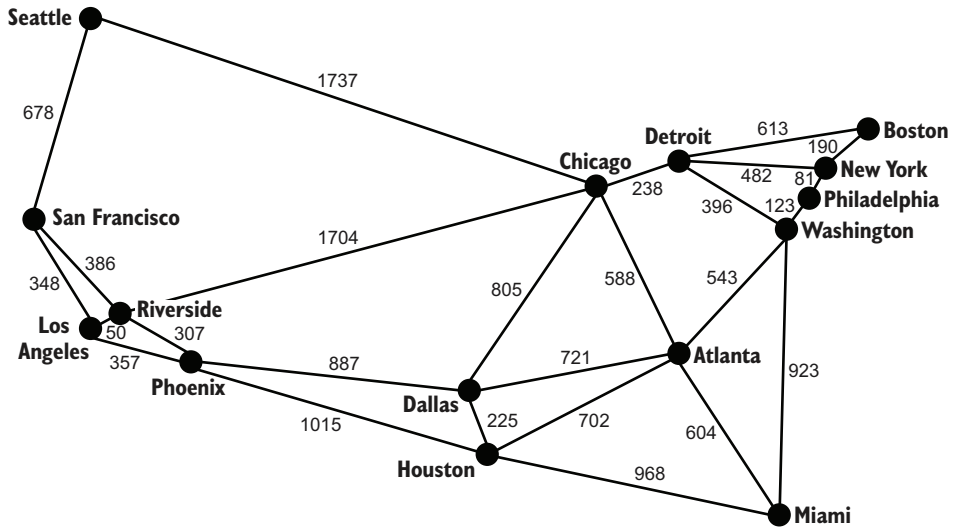


Figure 4.5 A weighted graph of the 15 largest MSAs in the United States, where each of the weights represents the distance between two MSAs in miles

as possible. To this end, we will allow the type of the weights associated with edges in our new `WeightedEdge` and `WeightedGraph` to be generic and therefore determined at creation time. But in order to execute several algorithms on weighted graphs, we do need the weights to have two properties: It must be possible to compare them, and it must be possible to add them together.

Any type that implements `Comparable` can be compared using operators like `==` and `<`. There is no built-in protocol in Swift for specifying that a type can be added, so we will create our own.

```
public protocol Summable {
    static func +(lhs: Self, rhs: Self) -> Self
}
```

If a type implements `Summable`, it means that instances of it can be added together. All of our weights must be `Summable`, meaning it must be possible to add them together, so they must implement the `+` operator. Of course, one category of types that can be added is numbers. Because the built-in number types in Swift already implement the `+` operator, it is possible to add `Summable` support to them without any work.

```
extension Int: Summable {}
extension Double: Summable {}
extension Float: Summable {}
```

A `WeightedEdge` will have a generic type, `W`, representing the type of its weight. It will also implement the protocols `Edge` and `Comparable`. Why does it implement `Comparable`?

The reason is that Jarnik's algorithm, which we will cover shortly, requires the ability to compare one edge with another.

```
open class WeightedEdge<W: Comparable & Summable>: Edge, Comparable {
    public var u: Int
    public var v: Int
    public let weight: W

    public var reversed: Edge {
        return WeightedEdge(u: v, v: u, weight: weight)
    }

    public init(u: Int, v: Int, weight: W) {
        self.weight = weight
        self.u = u
        self.v = v
    }

    //Implement CustomStringConvertible protocol
    public var description: String {
        return "\(u) <\(weight)> \(v)"
    }

    //MARK: Operator Overloads for Comparable
    static public func == <W>(lhs: WeightedEdge<W>,
    ➥ rhs: WeightedEdge<W>) -> Bool {
        return lhs.u == rhs.u && lhs.v == rhs.v && lhs.weight == rhs.weight
    }

    static public func < <W>(lhs: WeightedEdge<W>, rhs:
    ➥ WeightedEdge<W>) -> Bool {
        return lhs.weight < rhs.weight
    }
}
```

The implementation of `WeightedEdge` is not immensely different from the implementation of `UnweightedEdge`. It just has a new `weight` property and the implementation of `Comparable` via the `==` and `<` operators. The `<` operator is only interested in looking at weights, because Jarnik's algorithm is interested in finding the smallest edge by weight.

A `WeightedGraph` is a lot like an `UnweightedGraph`: It has init methods, it has convenience methods for adding `WeightedEdge`s, and it implements `CustomStringConvertible` via a `description` property. Where it differs is in the new generic type, `W`, that matches the type its weighted edges take. There is also a new method, `neighborsForIndexWithWeights()`, that returns not only each neighbor but also the weight of the edge that got to it. This method is useful for the new version of `description`.

```
open class WeightedGraph<V: Equatable & Hashable, W: Comparable & Summable>:
➥ Graph {
    var vertices: [V] = [V]()
    var edges: [[WeightedEdge<W>]] = [[WeightedEdge<W>]]() //adjacency lists
```

```
    public init() {
    }

    public init(vertices: [V]) {
        for vertex in vertices {
            _ = self.addVertex(vertex)
        }
    }

    /// Find all of the neighbors of a vertex at a given index.
    ///
    /// - parameter index: The index for the vertex to find the neighbors of.
    /// - returns: An array of tuples including the vertices as the first
        ➥ element and the weights as the second element.
    public func neighborsForIndexWithWeights(_ index: Int) -> [(V, W)] {
        var distanceTuples: [(V, W)] = [(V, W)]()
        for edge in edges[index] {
            distanceTuples += [(vertices[edge.v], edge.weight)]
        }
        return distanceTuples
    }

    /// This is a convenience method that adds a weighted edge.
    ///
    /// - parameter from: The starting vertex's index.
    /// - parameter to: The ending vertex's index.
    /// - parameter weight: the Weight of the edge to add.
    public func addEdge(from: Int, to: Int, weight:W) {
        addEdge(WeightedEdge<W>(u: from, v: to, weight: weight))
    }

    /// This is a convenience method that adds a weighted edge between the
        ➥ first occurrence of two vertices. It takes O(n) time.
    ///
    /// - parameter from: The starting vertex.
    /// - parameter to: The ending vertex.
    /// - parameter weight: the Weight of the edge to add.
    public func addEdge(from: V, to: V, weight: W) {
        if let u = indexOfVertex(from) {
            if let v = indexOfVertex(to) {
                addEdge(WeightedEdge<W>(u: u, v: v, weight:weight))
            }
        }
    }

    //Implement Printable protocol
    public var description: String {
        var d: String = ""
        for i in 0..<vertices.count {
            d += "\(vertices[i]) -> \(neighborsForIndexWithWeights(i))\n"
        }
        return d
    }
}
```

It is now possible to actually define a weighted graph. The weighted graph we will work with is a representation of figure 4.5, called `cityGraph2`.

```
let cityGraph2: WeightedGraph<String,
➥ Int> = WeightedGraph<String, Int>(vertices:
➥ ["Seattle", "San Francisco", "Los Angeles", "Riverside",
➥ "Phoenix", "Chicago", "Boston", "New York", "Atlanta",
➥ "Miami", "Dallas", "Houston", "Detroit", "Philadelphia", "Washington"])

cityGraph2.addEdge(from: "Seattle", to: "Chicago", weight: 1737)
cityGraph2.addEdge(from: "Seattle", to: "San Francisco", weight: 678)
cityGraph2.addEdge(from: "San Francisco", to: "Riverside", weight: 386)
cityGraph2.addEdge(from: "San Francisco", to: "Los Angeles", weight: 348)
cityGraph2.addEdge(from: "Los Angeles", to: "Riverside", weight: 50)
cityGraph2.addEdge(from: "Los Angeles", to: "Phoenix", weight: 357)
cityGraph2.addEdge(from: "Riverside", to: "Phoenix", weight: 307)
cityGraph2.addEdge(from: "Riverside", to: "Chicago", weight: 1704)
cityGraph2.addEdge(from: "Phoenix", to: "Dallas", weight: 887)
cityGraph2.addEdge(from: "Phoenix", to: "Houston", weight: 1015)
cityGraph2.addEdge(from: "Dallas", to: "Chicago", weight: 805)
cityGraph2.addEdge(from: "Dallas", to: "Atlanta", weight: 721)
cityGraph2.addEdge(from: "Dallas", to: "Houston", weight: 225)
cityGraph2.addEdge(from: "Houston", to: "Atlanta", weight: 702)
cityGraph2.addEdge(from: "Houston", to: "Miami", weight: 968)
cityGraph2.addEdge(from: "Atlanta", to: "Chicago", weight: 588)
cityGraph2.addEdge(from: "Atlanta", to: "Washington", weight: 543)
cityGraph2.addEdge(from: "Atlanta", to: "Miami", weight: 604)
cityGraph2.addEdge(from: "Miami", to: "Washington", weight: 923)
cityGraph2.addEdge(from: "Chicago", to: "Detroit", weight: 238)
cityGraph2.addEdge(from: "Detroit", to: "Boston", weight: 613)
cityGraph2.addEdge(from: "Detroit", to: "Washington", weight: 396)
cityGraph2.addEdge(from: "Detroit", to: "New York", weight: 482)
cityGraph2.addEdge(from: "Boston", to: "New York", weight: 190)
cityGraph2.addEdge(from: "New York", to: "Philadelphia", weight: 81)
cityGraph2.addEdge(from: "Philadelphia", to: "Washington", weight: 123)
```

Because `WeightedGraph` implements `CustomStringConvertible`, we can print out `cityGraph2`.

```
print(cityGraph2)
```

In the output, you will see both the vertices each vertex is connected to and the weight of those connections.

```
Seattle -> [("Chicago", 1737), ("San Francisco", 678)]
San Francisco -> [("Seattle", 678), ("Riverside", 386), ("Los Angeles", 348)]
Los Angeles -> [("San Francisco", 348), ("Riverside", 50), ("Phoenix", 357)]
Riverside -> [("San Francisco", 386), ("Los Angeles", 50), ("Phoenix", 307),
➥ ("Chicago", 1704)]
Phoenix -> [("Los Angeles", 357), ("Riverside", 307), ("Dallas", 887),
➥ ("Houston", 1015)]
Chicago -> [("Seattle", 1737), ("Riverside", 1704), ("Dallas", 805),
➥ ("Atlanta", 588), ("Detroit", 238)]
Boston -> [("Detroit", 613), ("New York", 190)]
```

```
New York -> [("Detroit", 482), ("Boston", 190), ("Philadelphia", 81)]
Atlanta -> [("Dallas", 721), ("Houston", 702), ("Chicago", 588),
➥ ("Washington", 543), ("Miami", 604)]
Miami -> [("Houston", 968), ("Atlanta", 604), ("Washington", 923)]
Dallas -> [("Phoenix", 887), ("Chicago", 805), ("Atlanta", 721),
➥ ("Houston", 225)]
Houston -> [("Phoenix", 1015), ("Dallas", 225), ("Atlanta", 702),
➥ ("Miami", 968)]
Detroit -> [("Chicago", 238), ("Boston", 613), ("Washington", 396),
➥ ("New York", 482)]
Philadelphia -> [("New York", 81), ("Washington", 123)]
Washington -> [("Atlanta", 543), ("Miami", 923), ("Detroit", 396),
➥ ("Philadelphia", 123)]
```

4.3.2 Finding the minimum spanning tree

A *tree* is a special kind of graph that has one, and only one, path between any two vertices. This implies that there are no *cycles* in a tree (which is sometimes called being *acyclic*). A cycle can be thought of as a circle (in the common sense, not the geometrical sense): If it is possible to traverse a graph from a starting vertex, never repeat any edges, and get back to the same starting vertex, then it has a cycle. Any graph that is not a tree can become a tree by pruning edges. Figure 4.6 illustrates pruning an edge to turn a graph into a tree.

A *connected* graph is a graph that has some way of getting from any vertex to any other vertex (all of the graphs we are looking at in this chapter are connected). A *spanning tree* is a tree that connects every vertex in a graph. A *minimum spanning tree* is a tree that connects every vertex in a weighted graph with the minimum total weight (compared to other spanning trees). For every weighted graph, it is possible to efficiently find its minimum spanning tree.

Whew, that was a lot of terminology! The point is that finding a minimum spanning tree is the same as finding a way to connect every vertex in a weighted graph with the minimum weight. This is an important and practical problem for anyone designing a network (transportation network, computer network, and so on)—how can every node in the network be connected for the minimum cost? That cost may be in terms of wire, track, road, or anything else. For instance, for a telephone network, another way of posing the problem is, "what is the minimum length of cable one needs to connect every phone?"

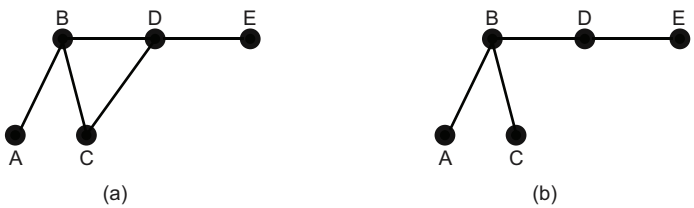


Figure 4.6 In (a), a cycle exists between vertices B, C, and D, so it is not a tree. In (b), the edge connecting C and D has been pruned, so the graph is a tree.

CALCULATING THE TOTAL WEIGHT OF A WEIGHTED PATH

Before we develop a method for finding a minimum spanning tree, we will develop a function we can use to test our future development. The solution to the minimum spanning tree problem will consist of an array of weighted edges that compose the tree. The function `totalWeight()` takes an array of `WeightedEdge<W>` and finds the total weight, `W`, that results from adding all of its edges' weights together.

```
public func totalWeight<W>(_ edges: [WeightedEdge<W>]) -> W? {
    guard let firstWeight = edges.first?.weight else { return nil }
    return edges.dropFirst().reduce(firstWeight) { (result, next) -> W in
        return result + next.weight
    }
}
```

`reduce()` is a higher-order function built in to most programming languages that can be programmed in a functional style. It takes a sequence of values and combines them via a closure. The closure is passed the result of each prior combination (the parameter `result` here) and the next value to be combined (`next` here). There's one problem—`reduce()` also requires a starting value. For most numbers, this would be 0, but because we don't know if `W` actually represents a number, we pull the first element out of `edges` and use it as the starting value. Because we do not want to re-add the first element after using it as the starting value, we call `dropFirst()` to ensure it is not added twice.

TIP `reduce()` is also known as "fold" in many other programming languages.

JARNIK'S ALGORITHM

Jarnik's algorithm for finding a minimum spanning tree works by dividing a graph into two parts: the vertices in the still-being-assembled minimum spanning tree, and the vertices not yet in the minimum spanning tree. It takes the following steps:

1. Pick an arbitrary vertex to be in the minimum spanning tree.
2. Find the lowest-weight edge connecting the minimum spanning tree to the vertices not yet in the minimum spanning tree.
3. Add the vertex at the end of that minimum edge to the minimum spanning tree.
4. Repeat steps 2 and 3 until every vertex in the graph is in the minimum spanning tree.

NOTE Jarnik's algorithm is commonly referred to as Prim's algorithm. Two Czech mathematicians, Otakar Borůvka and Vojtěch Jarník, interested in minimizing the cost of laying electric lines in the late 1920s, came up with algorithms to solve the problem of finding a minimum spanning tree. Their algorithms were "rediscovered" decades later by others.[4]

[4] Helena Durnova, "Otakar Boruvka (1899-1995) and the Minimum Spanning Tree" (Institute of Mathematics of the Czech Academy of Sciences, 2006), https://dml.cz/handle/10338.dmlcz/500001.

To run Jarnik's algorithm efficiently, a priority queue is used. Every time a new vertex is added to the minimum spanning tree, all of its outgoing edges that link to vertices outside the tree are added to the priority queue. The lowest-weight edge is always popped off the priority queue, and the algorithm keeps executing until the priority queue is empty. This ensures that the lowest-weight edges are always added to the tree first. Edges that connect to vertices already in the tree are ignored when they are popped.

The following code for `mst()` is the full implementation of Jarnik's algorithm,[5] along with a utility function for printing a `WeightedPath` and a new type defined in this extension of `WeightedGraph`.

WARNING Jarnik's algorithm will not necessarily work correctly in a graph with directed edges. It also will not work in a graph that is not connected.

```
/// Extensions to WeightedGraph for building a Minimum-Spanning Tree (MST)
public extension WeightedGraph {
    typealias WeightedPath = [WeightedEdge<W>]

    /// Find the minimum spanning tree in a weighted graph. This is the set
        ➥ of edges
    /// that touches every vertex in the graph and is of minimal combined
        ➥ weight. This function
    /// uses Jarnik's algorithm (aka Prim's algorithm) and so assumes the
        ➥ graph has
    /// undirected edges. For a graph with directed edges, the result may
        ➥ be incorrect. Also,
    /// if the graph is not fully connected, the tree will only span the
        ➥ connected component from which
    /// the starting vertex belongs.
    ///
    /// - parameter start: The index of the vertex to start creating
        ➥ the MST from.
    /// - returns: An array of WeightedEdges containing the minimum
        ➥ spanning tree, or nil if the starting vertex is invalid. If
        ➥ there are is only one vertex connected to the starting vertex,
        ➥ an empty list is returned.
    public func mst(start: Int = 0) -> WeightedPath? {
        if start > (vertexCount - 1) || start < 0 { return nil }
        var result: [WeightedEdge<W>] = [WeightedEdge<W>]() // the final
        ➥ MST goes in here
        var pq: PriorityQueue<WeightedEdge<W>> =
        ➥ PriorityQueue<WeightedEdge<W>>(ascending: true) // minPQ
        var visited: [Bool] = Array<Bool>(repeating: false, count:
        ➥ vertexCount) // already been to these

        func visit(_ index: Int) {
            visited[index] = true // mark as visited
            for edge in edgesForIndex(index) { // add all edges coming from
            ➥ here to pq
                if !visited[edge.v] { pq.push(edge) }
            }
        }
```

[5] Robert Sedgewick and Kevin Wayne, *Algorithms*, 4th Edition (Addison-Wesley Professional, 2011), p. 619.

```
        visit(start) // the first vertex is where everything begins

        while let edge = pq.pop() { // keep going as long as there are
        ➥ edges to process
            if visited[edge.v] { continue } // if we've been both places,
            ➥ ignore
            result.append(edge) // otherwise this is the current smallest
            ➥ so add it to the result set
            visit(edge.v) // visit where this connects
        }

        return result
    }

    /// Pretty-print an edge list returned from an MST
    /// - parameter edges The edge array representing the MST
    public func printWeightedPath(_ weightedPath: WeightedPath) {
        for edge in weightedPath {
            print("\(vertexAtIndex(edge.u)) \(edge.weight)>
            ➥ \(vertexAtIndex(edge.v))")
        }
        if let tw = totalWeight(weightedPath) {
            print("Total Weight: \(tw)")
        }
    }
}
```

Let's walk through `mst()`, line by line.

```
public func mst(start: Int = 0) -> WeightedPath? {
    if start > (vertexCount - 1) || start < 0 { return nil }
```

The algorithm returns an optional `WeightedPath` representing the minimum spanning tree. It does not matter where the algorithm starts (assuming the graph is connected and undirected), so the default is set to vertex index 0. If it so happens that the `start` is invalid, `mst()` returns `nil`.

```
var result: [WeightedEdge<W>] = [WeightedEdge<W>]() // the final MST goes
➥ in here
var pq: PriorityQueue<WeightedEdge<W>> =
➥ PriorityQueue<WeightedEdge<W>>(ascending: true) // minPQ
var visited: [Bool] = Array<Bool>(repeating: false, count: vertexCount)
➥ // already been to these
```

`result` will ultimately hold the weighted path containing the minimum spanning tree. This is where we will add `WeightedEdge`s, as the lowest-weight edge is popped off and takes us to a new part of the graph. Jarnik's algorithm is considered a *greedy algorithm* because it always selects the lowest-weight edge. `pq` is where newly discovered edges are stored and the next-lowest-weight edge is popped. `visited` keeps track of

vertex indices that we have already been to. This could also have been accomplished with a `Set`, similar to `explored` in `bfs()`.

```
func visit(_ index: Int) {
    visited[index] = true // mark as visited
    for edge in edgesForIndex(index) { // add all edges coming from here
    ➥ to pq
        if !visited[edge.v] { pq.push(edge) }
    }
}
```

`visit()` is an inner convenience function that marks a vertex as visited and adds all of its edges that connect to vertices not yet visited to `pq`. Note how easy the adjacency-list model makes finding edges belonging to a particular vertex.

```
visit(start) // the first vertex is where everything begins
```

It does not matter which vertex is visited first, unless the graph is not connected. If the graph is not connected, but is instead made up of disconnected *components*, `mst()` will return a tree that spans the particular component that the starting vertex belongs to.

```
while let edge = pq.pop() { // keep going as long as there are edges to
    process
    if visited[edge.v] { continue } // if we've been both places, ignore
    result.append(edge) // otherwise this is the current smallest so add
    ➥ it to the result set
    visit(edge.v) // visit where this connects
}
return result
```

While there are still edges on the priority queue, we pop them off and check if they lead to vertices not yet in the tree. Because the priority queue is ascending, it pops the lowest-weight edges first. This ensures that the result is indeed of minimum total weight. Any edge popped that does not lead to an unexplored vertex is ignored. Otherwise, because the edge is the lowest seen so far, it is added to the result set, and the new vertex it leads to is explored. When there are no edges left to explore, the result is returned.

Let's finally return to the problem of connecting all 15 of the largest MSAs in the United States by Hyperloop, using a minimum amount of track. The route that accomplishes this is simply the minimum spanning tree of `cityGraph2`. Let's try running `mst()` on `cityGraph2`.

```
if let mst = cityGraph2.mst() {
    cityGraph2.printWeightedPath(mst)
}
```

Thanks to the pretty-printing `printWeightedPath()` method, the minimum spanning tree is easy to read.

```
Seattle 678> San Francisco
San Francisco 348> Los Angeles
Los Angeles 50> Riverside
Riverside 307> Phoenix
Phoenix 887> Dallas
Dallas 225> Houston
Houston 702> Atlanta
Atlanta 543> Washington
Washington 123> Philadelphia
Philadelphia 81> New York
New York 190> Boston
Washington 396> Detroit
Detroit 238> Chicago
Atlanta 604> Miami
Total Weight: 5372
```

In other words, this is the cumulatively shortest collection of edges that connects all of the MSAs in the weighted graph. The minimum length of track needed to connect all of them is 5372 miles. Figure 4.7 illustrates the minimum spanning tree.

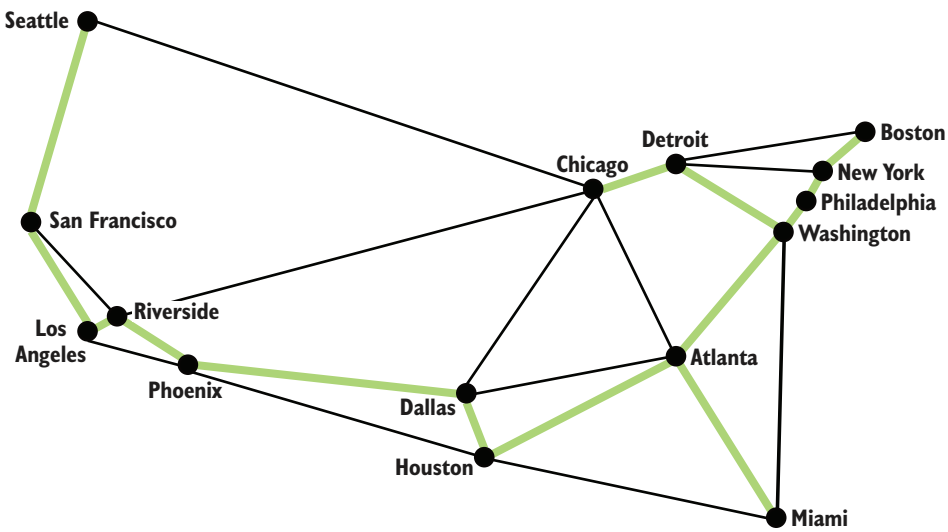


Figure 4.7 **The highlighted edges represent a minimum spanning tree that connects all 15 MSAs.**

4.4 Finding shortest paths in a weighted graph

As the Hyperloop network gets built, it is unlikely the builders will have the ambition to connect the whole country at once. Instead, it is likely the builders will want to minimize the cost to lay track between key cities. The cost to extend the network to particular cities will obviously depend on where the builders start.

Finding the cost to any city from some starting city is a version of the "single-source shortest path" problem. That problem asks, "what is the shortest path (in terms of total edge weight) from some vertex to every other vertex in a weighted graph?"

4.4.1 Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest path problem. It is provided a starting vertex, and it returns the lowest-weight path to any other vertex on a weighted graph. It also returns the minimum total weight to every other vertex from the starting vertex. Dijkstra's algorithm starts at the single-source vertex, and then continually explores the closest vertices to the start vertex. For this reason, like Jarnik's algorithm, Dijkstra's algorithm is greedy. When Dijkstra's algorithm encounters a new vertex, it keeps track of how far it is from the start vertex, and updates this value if it ever finds a shorter path. It also keeps track of what edge got it to each vertex, like a breadth-first search.

Here are all of the algorithm's steps:

1. Add the start vertex to a priority queue.
2. Pop the closest vertex from the priority queue (at the beginning this is just the start vertex)—we'll call it the current vertex.
3. Look at all of the neighbors connected to the current vertex. If they have not previously been recorded, or the edge offers a new shortest path to them, then for each of them record its distance from the start, record the edge that produced this distance, and add the new vertex to the priority queue.
4. Repeat steps 2 and 3 until the priority queue is empty.
5. Return the shortest distance to every vertex from the start vertex and the path to get to each of them.

The extension to `WeightedGraph` for Dijkstra's algorithm includes `DijkstraNode`, a simple data structure for keeping track of costs associated with each vertex explored so far and for comparing them. This is not dissimilar to the `Node` class in chapter 2. It also includes utility functions for converting the returned array of distances to something easier to use for looking up by vertex, and for calling `dijkstra()` without vertex indices.

Without further ado, here is the code for the extension. We will go over it line by line after.

```
public extension WeightedGraph {

    /// Represents a node in the priority queue used
    /// for selecting the next
    struct DijkstraNode: Comparable, Equatable {
        let vertex: Int
        let distance: W

        public static func < (lhs: DijkstraNode, rhs: DijkstraNode) -> Bool {
            return lhs.distance < rhs.distance
        }

        public static func == (lhs: DijkstraNode, rhs: DijkstraNode)
        ➥ -> Bool {
            return lhs.distance == rhs.distance
        }
```

```
}

/// Finds the shortest paths from some route vertex to every other
    ➥ vertex in the graph.
///
/// - parameter graph: The WeightedGraph to look within.
/// - parameter root: The index of the root node to build the shortest
    ➥ paths from.
/// - parameter startDistance: The distance to get to the root node
    ➥ (typically 0).
/// - returns: Returns a tuple of two things: the first, an array
    ➥ containing the distances, the second, a dictionary containing
    ➥ the edge to reach each vertex. Use the function
    ➥ pathDictToPath() to convert the dictionary into something
    ➥ useful for a specific point.
public func dijkstra(root: Int, startDistance: W) -> ([W?],
➥ [Int: WeightedEdge<W>]) {
    var distances: [W?] = [W?](repeating: nil, count: vertexCount)
    ➥ // how far each vertex is from start
    distances[root] = startDistance // the start vertex is
    ➥ startDistance away
    var pq: PriorityQueue<DijkstraNode> =
    ➥ PriorityQueue<DijkstraNode>(ascending: true)
    var pathDict: [Int: WeightedEdge<W>] = [Int: WeightedEdge<W>]()
    ➥ // how we got to each vertex
    pq.push(DijkstraNode(vertex: root, distance: startDistance))

    while let u = pq.pop()?.vertex { // explore the next closest vertex
        guard let distU = distances[u] else { continue } // should
        ➥ already have seen it
        for we in edgesForIndex(u) { // look at every edge/vertex
        ➥ from the vertex in question
            let distV = distances[we.v] // the old distance to
            ➥ this vertex
            if distV == nil || distV! > we.weight + distU { // if
            ➥ we have no old distance or we found a shorter path
                distances[we.v] = we.weight + distU
                ➥ // update the distance to this vertex
                pathDict[we.v] = we // update the edge on the shortest
                ➥ path to this vertex
                pq.push(DijkstraNode(vertex: we.v, distance:
                ➥ we.weight + distU)) // explore it soon
            }
        }
    }

    return (distances, pathDict)
}


/// A convenience version of dijkstra() that allows the supply of
    ➥ the root
/// vertex instead of the index of the root vertex.
public func dijkstra(root: V, startDistance: W)
➥ -> ([W?], [Int: WeightedEdge<W>]) {
```

```
        if let u = indexOfVertex(root) {
            return dijkstra(root: u, startDistance: startDistance)
        }
        return ([], [:])
    }

    /// Helper function to get easier access to Dijkstra results.
    public func distanceArrayToVertexDict(distances: [W?]) -> [V : W?] {
        var distanceDict: [V: W?] = [V: W?]()
        for i in 0..<distances.count {
            distanceDict[vertexAtIndex(i)] = distances[i]
        }
        return distanceDict
    }
}
```

The first few lines of `dijkstra()` use data structures you have become familiar with, except for `distances`, which is a placeholder for the distances to every vertex in the graph from the `root`. Initially all of these distances are `nil`, because we do not yet know how far each of them is—that is what we are using Dijkstra's algorithm to figure out!

```
public func dijkstra(root: Int, startDistance: W) -> ([W?],
➥ [Int: WeightedEdge<W>]) {
    var distances: [W?] = [W?](repeating: nil, count: vertexCount)
    ➥ // how far each vertex is from start
    distances[root] = startDistance // the start vertex is startDistance away
    var pq: PriorityQueue<DijkstraNode> =
    ➥ PriorityQueue<DijkstraNode>(ascending: true)
    var pathDict: [Int: WeightedEdge<W>] = [Int: WeightedEdge<W>]()
    ➥ // how we got to each vertex
    pq.push(DijkstraNode(vertex: root, distance: startDistance))
```

The first node pushed onto the priority queue contains the root vertex.

```
while let u = pq.pop()?.vertex { // explore the next closest vertex
    guard let distU = distances[u] else { continue } // should already have
    ➥ seen it
```

We keep running Dijkstra's algorithm until the priority queue is empty. `u` is the current vertex we are searching from, and `distU` is the stored distance for getting to `u` along known routes. Every vertex explored at this stage has already been found, so it must have a known distance. If it doesn't, something is wrong, hence the `guard` statement.

```
for we in edgesForIndex(u) { // look at every edge/vertex from the vertex
➥ in question
   let distV = distances[we.v] // the old distance to this vertex
```

Next, every edge connected to `u` is explored. `distV` is the distance to any known vertex attached by an edge to `u`.

```
if distV == nil || distV! >
➥ we.weight + distU { // if we have no old distance or we found a
➥ shorter path
    distances[we.v] = we.weight + distU // update the distance to this vertex
    pathDict[we.v] = we // update the edge on the shortest path to
    ➥ this vertex
    pq.push(DijkstraNode(vertex: we.v, distance: we.weight + distU))
    ➥ // explore it soon
}
```

If we have found a vertex that has not yet been explored (`distV == nil`), or we have found a new, shorter path to it, we record that new shortest distance to `v` and the edge that got us there. It is okay to force unwrap `distV` here, because the second part of the "or" operator (`||`) is short-circuited, and we know if we get to it that `distV` is not `nil`. Finally, we push any vertices that have new paths to them to the priority queue.

```
return (distances, pathDict)
```

`dijkstra()` returns both the distances to every vertex in the weighted graph from the root vertex, and the `pathDict` that can unlock the shortest paths to them. It is safe to run Dijkstra's algorithm now. Let's start by finding the distance from Los Angeles to every other MSA in the graph.

```
let (distances, pathDict) = cityGraph2.dijkstra(root: "Los Angeles",
➥ startDistance: 0)
var nameDistance: [String: Int?] =
➥ cityGraph2.distanceArrayToVertexDict(distances: distances)
for (key, value) in nameDistance {
    print("\(key) : \(String(describing: value!))")
}
```

Your output should look something like this:

```
Phoenix : 357
Detroit : 1992
Houston : 1372
Washington : 2388
Riverside : 50
Chicago : 1754
Dallas : 1244
Atlanta : 1965
New York : 2474
Philadelphia : 2511
Boston : 2605
San Francisco : 348
Seattle : 1026
Los Angeles : 0
Miami : 2340
```

We can use our old friend, `pathDictToPath()`, to find the shortest path between Los Angeles and a specific other MSA—say Boston. Finally, we can use `printWeightedPath()` to pretty-print the result.

```
let path = pathDictToPath(from:
➥ cityGraph2.indexOfVertex("Los Angeles")!, to:
➥ cityGraph2.indexOfVertex("Boston")!, pathDict: pathDict)
cityGraph2.printWeightedPath(path as! [WeightedEdge<Int>])
```

The shortest path from Los Angeles to Boston is

```
Los Angeles 50> Riverside
Riverside 1704> Chicago
Chicago 238> Detroit
Detroit 613> Boston
Total Weight: 2605
```

You may have noticed that Dijkstra's algorithm has some resemblance to Jarnik's algorithm. They are both greedy, and it is possible to implement them using quite similar code if one is sufficiently motivated. Another algorithm that Dijkstra's algorithm resembles is A* from chapter 2. A* can be thought of as a modification of Dijkstra's algorithm. Add a heuristic and restrict Dijkstra's algorithm to finding a single destination, and the two algorithms are the same.

4.5 Real-world applications

A huge amount of our world can be represented using graphs. You have seen in this chapter how effective they are for working with transportation networks, but many other kinds of networks have the same essential optimization problems: telephone networks, computer networks, utility networks (electricity, plumbing, and so on). As a result, graph algorithms are essential for efficiency in the telecommunications, shipping, transportation, and utility industries.

Retailers must handle complex distribution problems. Stores and warehouses can be thought of as vertices and the distances between them as edges. The algorithms are the same. The internet itself is a giant graph, with each connected device a vertex and each wired or wireless connection being an edge. Whether a business is saving fuel or wire, minimum spanning tree and shortest path problem-solving are useful for more than just games. Some of the world's most famous brands became successful by optimizing graph problems: think of Walmart building out an efficient distribution network, Google indexing the web (a giant graph), and FedEx finding the right set of hubs to connect the world's addresses.

Some obvious applications of graph algorithms are social networks and map applications. In a social network, people are vertices, and connections (friendships on Facebook, for instance) are edges. In fact, one of Facebook's most prominent developer tools is known as the "Graph API" (https://developers.facebook.com/docs/graph-api). In map applications like Apple Maps and Google Maps, graph algorithms are used to provide directions and calculate trip times.

Several popular video games also make explicit use of graph algorithms. MiniMetro and Ticket to Ride are two examples of games that closely mimic the problems solved in this chapter.

4.6 Exercises

1. Add support to the graph framework for removing edges and vertices.
2. Add support to the graph framework for directed graphs (digraphs).
3. Add an extension to `Graph` for depth-first search (see chapter 2).
4. Use this chapter's graph framework to prove or disprove the classic Bridges of Konigsberg problem.

index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W