Second Edition

A Swift Kickstart

DANIEL H STEINBERG

Introducing the
Swift Programming Language

Editors Cut

Copyright

ISBN-13: 978-0-9830669-8-9

Legal

This is version 1.0 for Swift 5.3, Xcode 12, and iOS 14 released September 2020.

Subclasses

This example comes out of a workshop I used to do at conferences. At these conferences there were `Attendee`s and people who paid extra to take the training as well. Here I'm calling them `TutorialAttendee`s. Because every `TutorialAttendee` is also an `Attendee`, we can model this relationship using subclasses.

The Liskov substitution principle guides how we think of subclasses. It says that if we have an instance of a `TutorialAttendee` then when it comes to things that an `Attendee` can do, the `TutorialAttendee` should behave no differently. It can have additional features and behaviors but it must be substitutable for an ordinary `Attendee`.

Set up

Let's start our new playground page with our most recent version of `Attendee`.

07Classes/04Subclasses

```
class Attendee {
    let name: String
    let hometown: String

    init(name: String,
         hometown: String = "Cupertino") {
        self.name = name
        self.hometown = hometown
    }
}
```

Every `Attendee` has a `name` and a `hometown`. A `TutorialAttendee` will also have a `tutorial`. Let's create our subclass.

Create a subclass

Below this add this subclass named `TutorialAttendee`. A `TutorialAttendee` is an `Attendee` who's also taking a tutorial. This will generate an error that we can ignore for now.

07Classes/04Subclasses

```
class TutorialAttendee: Attendee, CustomStringConvertible {
    // this is wrong but we'll fix it
}
```

The colon followed by `Attendee` indicates that `TutorialAttendee` is a subclass of `Attendee`.

`CustomStringConvertible`, after `Attendee`, indicates that `TutorialAttendee` also conforms to the `CustomStringConvertible` protocol.

We can mix and match listing a superclass and one or more protocols without confusion using the following pattern.

```
class MyClass: SuperClass, ProtocolOne, ProtocolTwo
```

A class can have zero or one superclasses. If there's a superclass, then we list it directly after the colon.

A class can have zero or more protocols. We list them comma-separated as shown above. If there's a superclass then we list the protocols after the superclass and a comma. If our class is a base class, then it won't have a superclass, so we list only the protocols if there are any.

I included `CustomStringConvertible` to illustrate the syntax of conforming to a protocol. Remove `CustomStringConvertible` from the top line of `Attendee`.

07Classes/04Subclasses

```
class TutorialAttendee: Attendee, CustomStringConvertible {

}
```

The code builds fine without warnings. Run the code and nothing happens but we also don't see any errors. Now, add a new property named `tutorial` of type `String`. This causes an error.

07Classes/04Subclasses

```
class TutorialAttendee: Attendee {
    let tutorial: String
}
```

We've seen this error message before.

```
Class 'TutorialAttendee' has no initializers
```

Before we added `tutorial`, `TutorialAttendee` could just use `Attendee`'s `init()`. Now `TutorialAttendee` needs an `init()` that initializes `tutorial` as well.

Initializer for subclass

The message is a bit confusing because we know we inherit an `init()` from `Attendee`. What the error is trying to tell us is that there's no initializer that sets the value for the `tutorial` property.

We don't want to repeat the code that we already inherit from the superclass, so the strategy is to create an initializer that takes parameters for `name`, `hometown`, and `tutorial`. This initializer sets the value of `tutorial` and then calls the `init()` method in the superclass, passing along the values for `name` and `hometown`.

The order is crucial. Swift strives to keep us safe. It's checking to make sure that we set all of the property values and it knows that the initializer in the superclass can't set the value of `tutorial`, so it insists that we set the value of `tutorial` before we call the `init()` in the superclass.

The rule is that if we introduce a new property in a subclass, that property must be initialized before we call up to the superclass' `init()` method.

I tend to list parameters that have default values at the end so I would insert `tutorial` as the second parameter.

07Classes/04Subclasses

```
class TutorialAttendee: Attendee {
  let tutorial: String

  init(name: String,
       tutorial: String,
       hometown: String = "Cupertino") {
    self.tutorial = tutorial
    super.init(name: name,
               hometown: hometown)
  }
}
```

I deliberately shuffled the order of the parameters in `TutorialAttendee` to show you that you don't have to preserve the order from the super class. On the other reason there's no reason you can't keep the same order and put the new properties at the start or end of the list.

Creating instances

Now we can create instances of `Attendee` and `TutorialAttendee` and check on their `name`, `hometown`, and, in the case of the `TutorialAttendee`, `tutorial`.

07Classes/04Subclasses

```
let daniel = Attendee(name: "Daniel",
                      hometown: "Shaker Heights")
let kimberli = TutorialAttendee(name: "Kimberli",
                                tutorial: "Swiftiness")
daniel.name
                                                        "Daniel"
daniel.hometown
                                                "Shaker Heights"
kimberli.name
                                                      "Kimberli"
kimberli.hometown
                                                     "Cupertino"
kimberli.tutorial
                                                   "Swiftiness"
```

I said before that a designated initializer must initialize all of the properties in its class. It doesn't need to do so on its own. To be a designated initializer, a subclass must initialize all properties that are specific to the subclass. Next, it must call up to a designated initializer in its superclass to initialize the properties that are specific to its superclass. The process continues until we get to a designated initializer in the base class.

Understanding initialization

The initialization takes place in two phases. First we move up the chain initializing all of the properties. Once we are at the base class and all properties are initialized we can do any other work by calling

methods. Then we return to the line after the call to `super.init()` in the first subclass and work our way down the chain.

Here's a summary of the process for a base class with a single subclass. I've indicated the order out front.

- Base class initializiation:
    - (3) Initialize properties declared in the base class
    - (4) Do any other setup required now that all properties are initialized
- Subclass initialization:
    - (1) Initialize properties introduced in this subclass
    - (2) Call `super.init()`
    - (5) Do any other setup now that all superclasses have done their setup

One last point about initializers and subclasses. There may be cases where you want to make sure that subclasses implement the same initializer with the same signature. We can enforce that by adding the keyword `required` to the `init()` in the superclass.

07Classes/04Subclasses

```
class Attendee {
  let name: String
  let hometown: String

  required init(name: String,
                          hometown: String = "Cupertino") {
    self.name = name
    self.hometown = hometown
  }
}
```

Subclass init

We get an error telling us that we must implement this same `init()` in `TutorialAttendee`. We have to use the `required` keyword in the subclass as well.

07Classes/04Subclasses

```
class TutorialAttendee: Attendee {
  let tutorial: String
  init(name: String,
     tutorial: String,
     hometown: String = "Cupertino"){
     self.tutorial = tutorial
     super.init(name: name, hometown: hometown)
  }
  required init(name: String,
                hometown: String = "Cupertino"){
    tutorial = "iOS Development"
    super.init(name: name, hometown: hometown)
  }
}
```

That's a quick look at subclasses and initialization. In the next section we add a method to `Attendee`.