Second Edition

A Swift Kickstart

DANIEL H STEINBERG

Introducing the
Swift Programming Language

Editors Cut

Copyright

"A Swift Kickstart" Second Edition, by Daniel H Steinberg

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

Legal

This is version 1.1 for Swift 5.5, Xcode 13, and iOS 15 released July 2021.

Methods

Swift enumerations can contain functions. When a function is part of an enumeration, we call it a method.

In our current example, the `color()` function can accept a `PrimaryColor` and return the corresponding `Color`.

If we turn this function into a method that is part of `PrimaryColor`, then any `PrimaryColor` instance then we don't have to pass any information to the method - it already knows which case of `PrimaryColor` it is. We can now ask any instance of `PrimaryColor` to tell us its `Color`.

Set up

Let's start our new playground page with the same code from the previous section and modify it as needed.

05Enumerations/03Methods

```
import SwiftUI

enum PrimaryColor {
    case red
    case yellow
    case blue
}

func color(from primaryColor: PrimaryColor) -> Color {
    switch primaryColor {
    case .red:
        return Color.red
    case .yellow:
        return Color.yellow
    case .blue:
        return Color.blue
    }
}
```

Note that `color` is a function. It is not part of `PrimaryColor`. We're going to change that.

Introducing a method

There are only three small changes we need to make to `color()` to turn it from a free function to a method.

First, we need to move the function into the body of the enumeration. Note that we keep the keyword `func` even though we now call it a method. Don't worry if you see any errors along the way.

05Enumerations/03Methods

```
enum PrimaryColor {
    case red
    case yellow
    case blue
    // Step 1: Move the function into the enumeration
    func color(from primaryColor: PrimaryColor) -> Color {
        switch primaryColor {
        case .red:
            return Color.red
        case .yellow:
            return Color.yellow
        case .blue:
            return Color.blue
        }
    }
}
```

The second step is to remove the parameters from `color()`. The `color()` method is part of the `PrimaryColor` instance so it knows which `PrimaryColor` it is calculating the `Color` for.

05Enumerations/03Methods

```
func color(~~from primaryColor: PrimaryColor~~) -> Color {
```

Third, the `switch` statement now switches on `self`. `self` refers to the instance of `PrimaryColor` that the method will be called on.

05Enumerations/03Methods

```
    func color() -> Color {
        switch self {  // ...
        }
    }
```

Here's the current state of our `PrimaryColor` enumeration.

05Enumerations/03Methods

```
enum PrimaryColor {
    case red
    case yellow
    case blue

    func color() -> Color {
        switch self {
        case .red:
            return Color.red
        case .yellow:
            return Color.yellow
        case .blue:
            return Color.blue
        }
    }
}
```

The `color()` method is now referred to as a member of `PrimaryColor`. Yes we still use `func` even though `color()` is now a method.

Calling the method

We need an instance of `PrimaryColor` to call the `color()` method. With free functions, we just pass in whatever the function needs. Methods have access to the state of the receiving object.

Create two instances and call the `color()` method on them using dot notation, like this.

05Enumerations/03Methods

```
let crayon = PrimaryColor.blue
```

```
blue
```

```
crayon.color()
```

```
blue
```

```
let paintBrush = PrimaryColor.yellow
paintBrush.color()
```

```
yellow
```

A second method

Add a method to `Color` named `circle()` so that we can see one method use another one. The `circle()` method will display the colors in a circle instead of a rectangle.

05Enumerations/03Methods

```
enum PrimaryColor {
    case red
    case yellow
    case blue

    func color() -> Color {
        switch self {
        case .red:
            return Color.red
        case .yellow:
            return Color.yellow
        case .blue:
            return Color.blue
        }
    }

    func circle() -> some View {
        Circle()
            .foregroundColor(color())
    }
}
```

The `circle()` method returns a SwiftUI `Circle` with the color equal to the `Color` returned by the `color()` method. Note that `circle()` omits the `return` keyword.

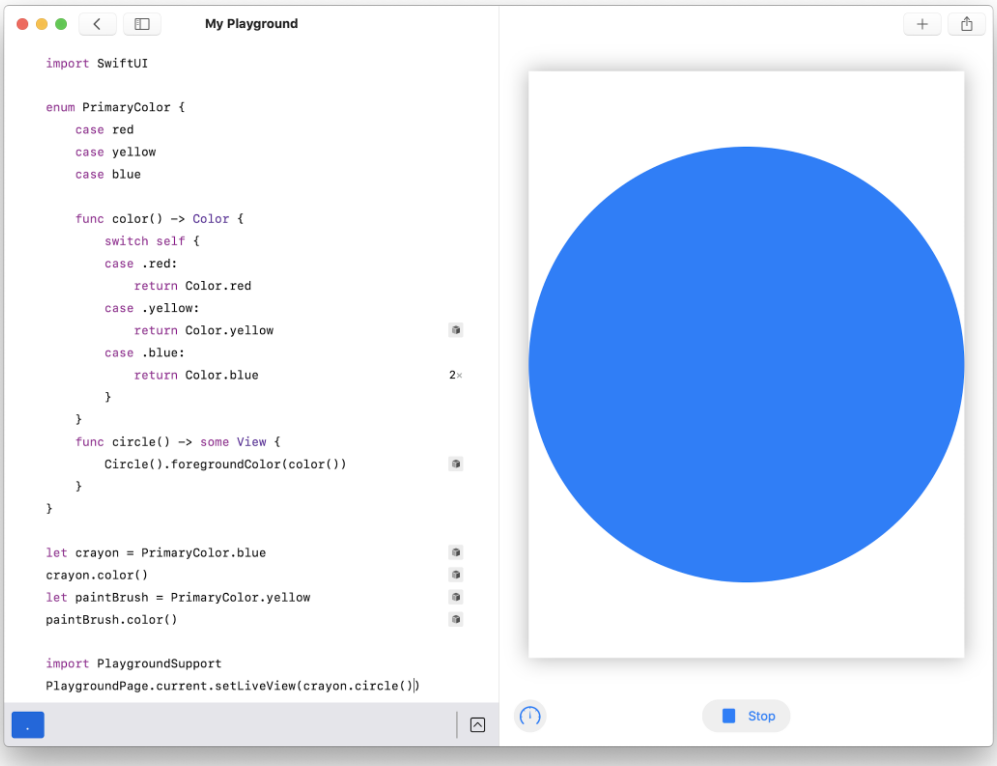
To see the result, we need to present the generated `Circle` in what is called a live view. Add this to the bottom of the playground page to see `crayon.circle()`.

05Enumerations/03Methods

```
import PlaygroundSupport
PlaygroundPage.current.setLiveView(crayon.circle())
```

Run the playground and you should see a blue circle on the right side.

Here's what it looks like in a Swift Playground.



Change the live view to `paintBrush.circle()` and you'll see a yellow circle instead.

Take a step back from the implementation of these methods for a moment and consider how cool it is that we can add methods to enumerations in Swift.

This means that, in Swift, enumerations contain both state and behavior. This is very different from the way enumerations work in many languages that we might be used to.

Swift enumerations have more tricks up their sleeves. They can also contain computed properties. I'll show you how in the next section.