Swift Protocols

Associated Types:

What is an associated type?

An associated type can be seen as a replacement of a specific type within a protocol definition. In other words: it's a placeholder name of a type to use until the protocol is adopted and the exact type is specified.

What are the benefits of using associated types?

They prevent writing duplicate code by making it easier to define a common interface for multiple scenarios. This way, the same logic can be reused for multiple different types, allowing you to write and test logic only once.

Syntax:

```
protocol Screen {
    associatedtype ItemType
    var items: [ItemType] { get set }
}
```

```
class MainScreen: Screen {
    typealias ItemType = String
    var items = [String]()
}
```

Example 1:

We have a convenient way of defining colors.

```
let color = UIColor(hex: "FF7217")
```

We'll reuse that method and define a protocol around it

```
public protocol BrandColorSupporting {
    associatedtype ColorValue
    static func colorFor(hex: String, alpha: CGFloat) -> ColorValue
}
```

Some colors also required to be defined with a specific alpha value which we included in this protocol too. Then added support for this protocol for both UIColor and Color.

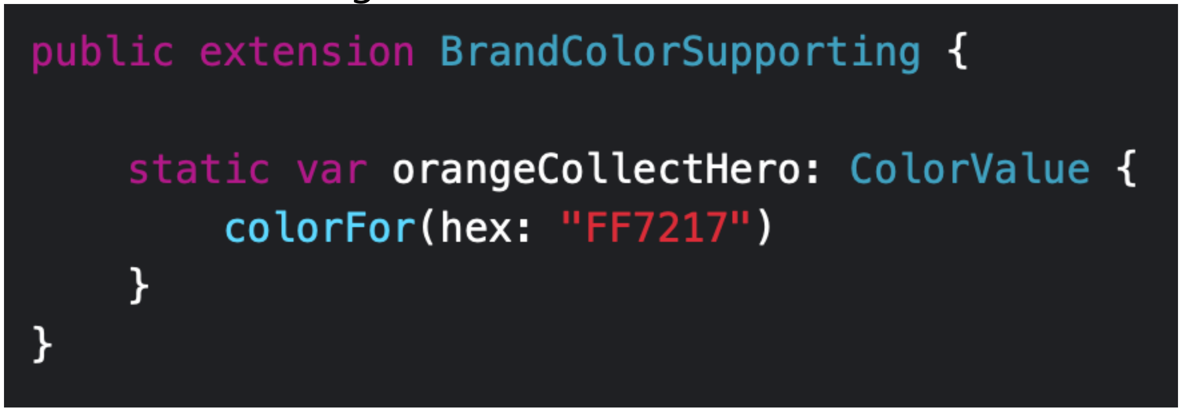
```swift
extension UIColor: BrandColorSupporting {
    public static func colorFor(hex: String, alpha: CGFloat) -> UIColor {
        return UIColor(hex: hex).withAlphaComponent(alpha)
    }
}

@available(iOS 13.0, *)
extension Color: BrandColorSupporting {
    public static func colorFor(hex: String, alpha: CGFloat) -> Color {
        return Color(UIColor.colorFor(hex: hex, alpha: alpha))
    }
}
```

This allows us to reuse the convenience initializer of UIColor to create colors using hex value.

As not all colors required to define an alpha component we added a default extension to our BrandColorSupporting Protocol:

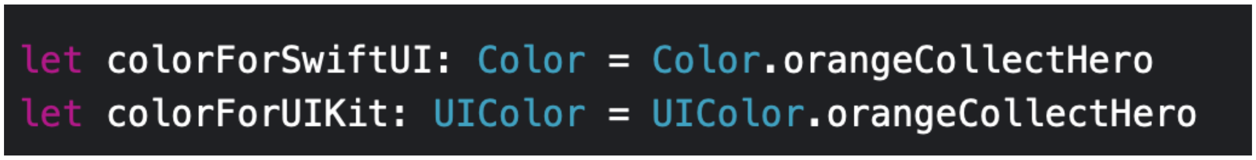
```swift
extension BrandColorSupporting {
    static func colorFor(hex: String) -> ColorValue {
        return colorFor(hex: hex, alpha: 1.0)
    }
}
```

We started defining colors



```swift
public extension BrandColorSupporting {

    static var orangeCollectHero: ColorValue {
        colorFor(hex: "FF7217")
    }
}
```

The best thing of using associated types is that we can make use of the same logic while the result type is changed based on context:



```swift
let colorForSwiftUI: Color = Color.orangeCollectHero
let colorForUIKit: UIColor = UIColor.orangeCollectHero
```

Type Erasure:

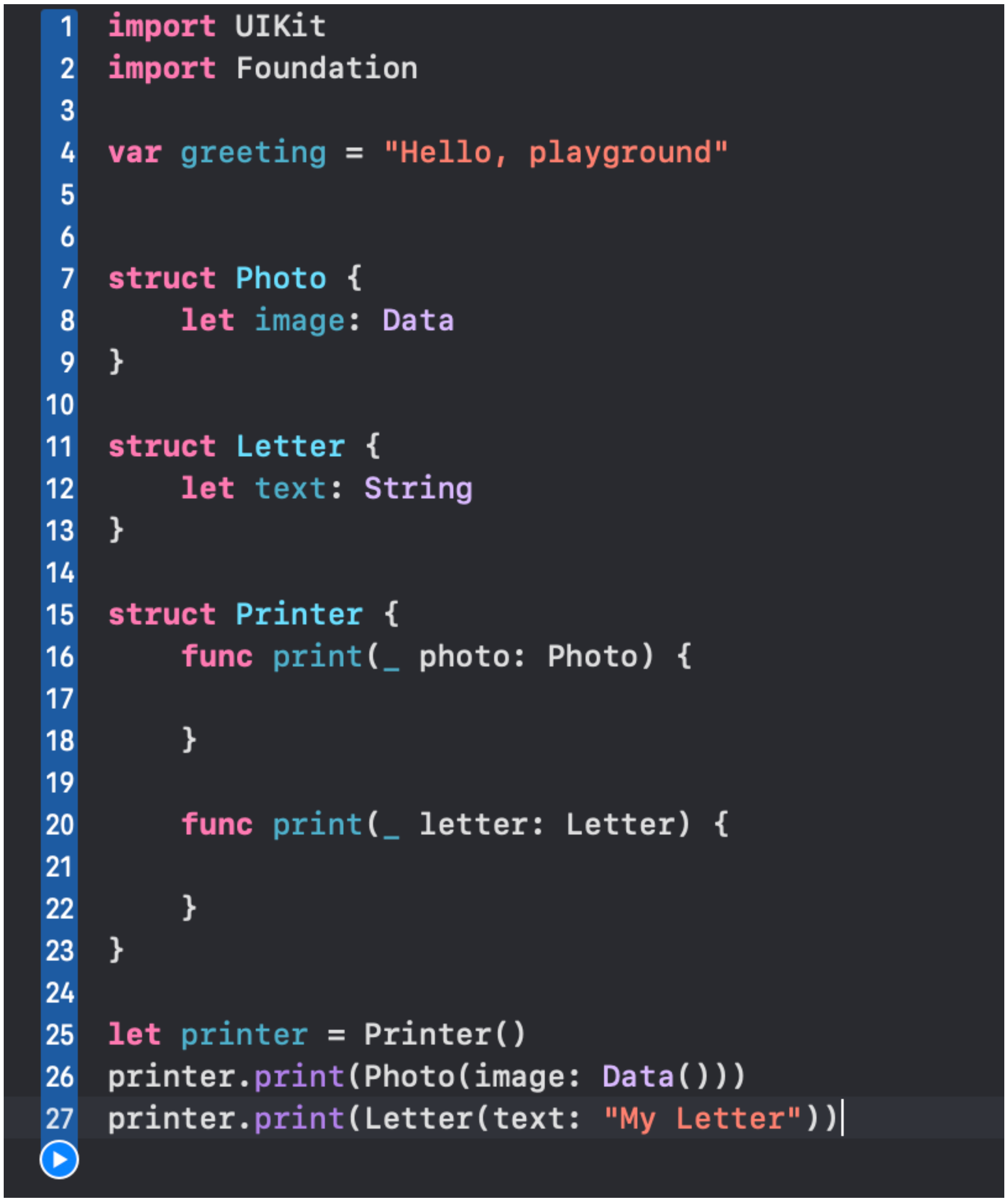
https://cocoacasts.com/understanding-type-erasure-in-swift-what-is-type-erasure

Swift is type-safe, meaning that each value has a type and a compiler that performs type checking at compile type. Your code won’t compile if it contains type errors. Swift’s type system is strict and that has its own drawbacks. It makes language less dynamic as other languages like JavaScript.

To work around the limitation of swift’s type system, it is necessary to hide type of a value. **<u>Hiding or erasing the type of a value is better known as type erasure.</u>**

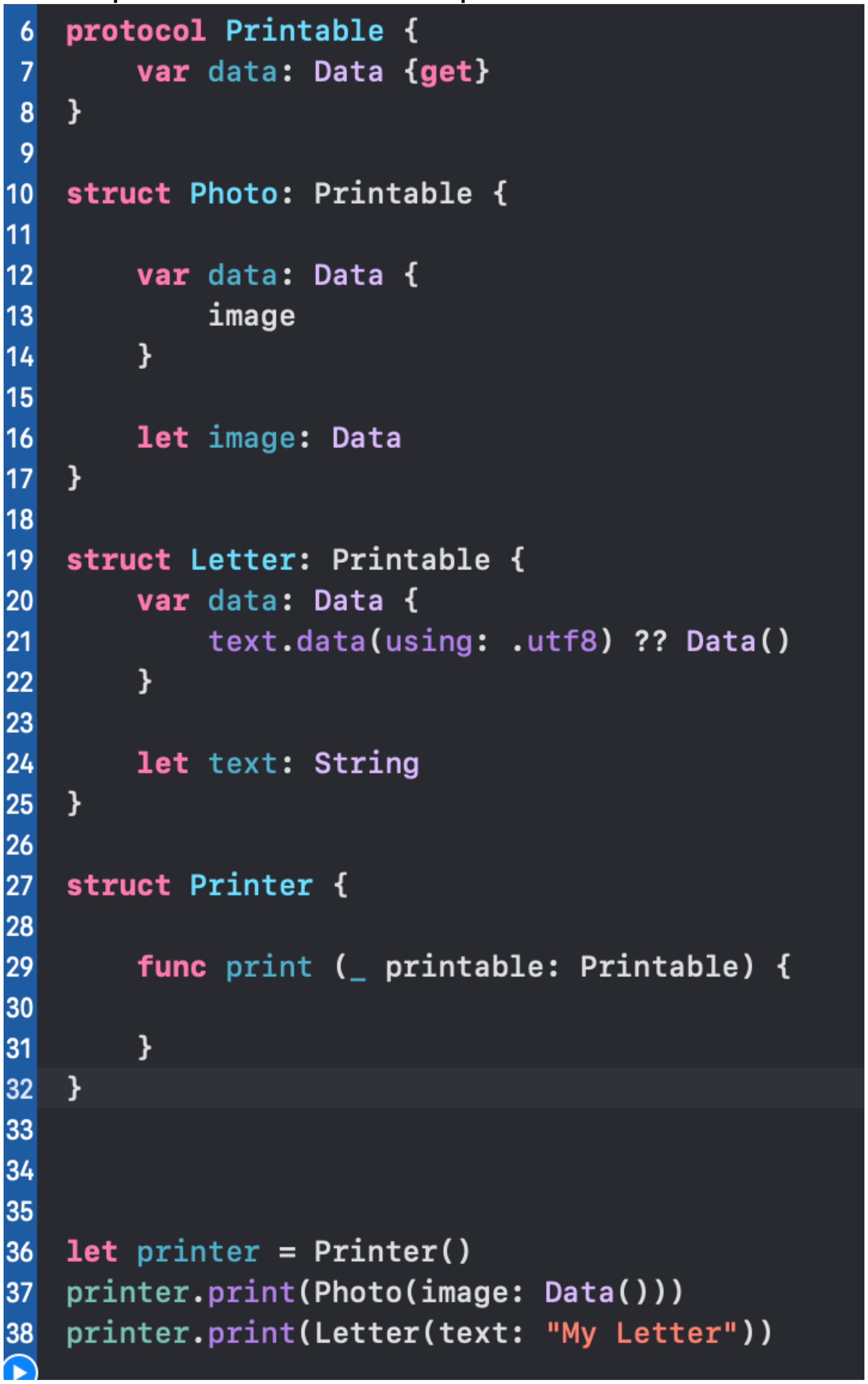
Even if you use type erasure, the compiler is still able to access the type of the value whose type is erased. Type safety is a fundamental concept of the language and type erasure does not change that.

An example:



```swift
import UIKit
import Foundation

var greeting = "Hello, playground"


struct Photo {
    let image: Data
}

struct Letter {
    let text: String
}

struct Printer {
    func print(_ photo: Photo) {

    }

    func print(_ letter: Letter) {

    }
}

let printer = Printer()
printer.print(Photo(image: Data()))
printer.print(Letter(text: "My Letter"))
```

This isn't a scalable solution because for instance if we need to add support for printing postcards. We need to extend the Printer struct with a method that accepts a Postcard object. The implementation also suffers from tight coupling. The Printer struct is tightly coupled to the photo and letter structs. Let us clean up the implementation with a protocol with name Printable.



```swift
protocol Printable {
    var data: Data {get}
}

struct Photo: Printable {

    var data: Data {
        image
    }

    let image: Data
}

struct Letter: Printable {
    var data: Data {
        text.data(using: .utf8) ?? Data()
    }

    let text: String
}

struct Printer {

    func print (_ printable: Printable) {

    }
}



let printer = Printer()
printer.print(Photo(image: Data()))
printer.print(Letter(text: "My Letter"))
```

We created the printable protocol to hide the type of the value that we pass to the print(_:) method. **This pattern is better known as protocol-oriented-programming.** While it isn't considered a form of type erasure, it clearly illustrates what hiding a value's type means and what benefits are doing so.