Nested Types

On This Page

Enumerations are often created to support a specific class or structure’s functionality. Similarly, it can be convenient to define utility classes and structures purely for use within the context of a more complex type. To accomplish this, Swift enables you to define *nested types*, whereby you nest supporting enumerations, classes, and structures within the definition of the type they support.

To nest a type within another type, write its definition within the outer braces of the type it supports. Types can be nested to as many levels as are required.

Nested Types in Action

The example below defines a structure called `BlackjackCard`, which models a playing card as used in the game of Blackjack. The `BlackjackCard` structure contains two nested enumeration types called `Suit` and `Rank`.

In Blackjack, the Ace cards have a value of either one or eleven. This feature is represented by a structure called `Values`, which is nested within the `Rank` enumeration:

```
struct BlackjackCard {

    // nested Suit enumeration
    enum Suit: Character {
        case spades = "♠", hearts = "♡", diamonds = "♢", clubs = "♣"
    }

    // nested Rank enumeration
    enum Rank: Int {
        case two = 2, three, four, five, six, seven, eight, nine, ten
        case jack, queen, king, ace
        struct Values {
            let first: Int, second: Int?
        }
        var values: Values {
            switch self {
            case .ace:
                return Values(first: 1, second: 11)
            case .jack, .queen, .king:
                return Values(first: 10, second: nil)
            default:
                return Values(first: self.rawValue, second: nil)
            }
        }
    }

    // BlackjackCard properties and methods
    let rank: Rank, suit: Suit
    var description: String {
        var output = "suit is \(suit.rawValue),"
        output += " value is \(rank.values.first)"
        if let second = rank.values.second {
            output += " or \(second)"
        }
        return output
    }
}
```

The `Suit` enumeration describes the four common playing card suits, together with a raw `Character` value to represent their symbol.

The `Rank` enumeration describes the thirteen possible playing card ranks, together with a raw `Int` value to represent their face value. (This raw `Int` value is not used for the Jack, Queen, King, and Ace cards.)

As mentioned above, the `Rank` enumeration defines a further nested structure of its own, called `Values`. This structure encapsulates the fact that most cards have one value, but the Ace card has two values. The `Values` structure defines two properties to represent this:

- `first`, of type `Int`
- `second`, of type `Int?`, or “optional `Int`”

`Rank` also defines a computed property, `values`, which returns an instance of the `Values` structure. This computed property considers the rank of the card and initializes a new `Values` instance with appropriate values based on its rank. It uses special values for `jack`, `queen`, `king`, and `ace`. For the numeric cards, it uses the rank’s raw `Int` value.

The `BlackjackCard` structure itself has two properties—`rank` and `suit`. It also defines a computed property called `description`, which uses the values stored in `rank` and `suit` to build a description of the name and value of the card. The `description` property uses optional binding to check whether there is a second value to display, and if so, inserts additional description detail for that second value.

Because `BlackjackCard` is a structure with no custom initializers, it has an implicit memberwise initializer, as described in Memberwise Initializers for Structure Types. You can use this initializer to initialize a new constant called `theAceOfSpades`:

```
let theAceOfSpades = BlackjackCard(rank: .ace, suit: .spades)
print("theAceOfSpades: \(theAceOfSpades.description)")
// Prints "theAceOfSpades: suit is ♠, value is 1 or 11"
```

Even though `Rank` and `Suit` are nested within `BlackjackCard`, their type can be inferred from context, and so the initialization of this instance is able to refer to the enumeration cases by their case names (`.ace` and `.spades`) alone. In the example above, the `description` property correctly reports that the Ace of Spades has a value of `1` or `11`.

Referring to Nested Types

To use a nested type outside of its definition context, prefix its name with the name of the type it is nested within:

```
let heartsSymbol = BlackjackCard.Suit.hearts.rawValue
// heartsSymbol is "♡"
```

For the example above, this enables the names of `Suit`, `Rank`, and `Values` to be kept deliberately short, because their names are naturally qualified by the context in which they are defined.