Enumerations and Structures

Use `enum` to create an enumeration. Like classes and all other named types, enumerations can have methods associated with them.

```
enum Rank: Int {
    case ace = 1
    case two, three, four, five, six, seven, eight, nine, ten
    case jack, queen, king

    func simpleDescription() -> String {
        switch self {
        case .ace:
            return "ace"
        case .jack:
            return "jack"
        case .queen:
            return "queen"
        case .king:
            return "king"
        default:
            return String(self.rawValue)
        }
    }
}
let ace = Rank.ace
let aceRawValue = ace.rawValue
```

> EXPERIMENT
>
> Write a function that compares two `Rank` values by comparing their raw values.

By default, Swift assigns the raw values starting at zero and incrementing by one each time, but you can change this behavior by explicitly specifying values. In the example above, `Ace` is explicitly given a raw value of `1`, and the rest of the raw values are assigned in order. You can also use strings or floating-point numbers as the raw type of an enumeration. Use the `rawValue` property to access the raw value of an enumeration case.

Use the `init?(rawValue:)` initializer to make an instance of an enumeration from a raw value. It returns either the enumeration case matching the raw value or `nil` if there's no matching `Rank`.

```
if let convertedRank = Rank(rawValue: 3) {
    let threeDescription = convertedRank.simpleDescription()
}
```

The case values of an enumeration are actual values, not just another way of writing their raw values. In fact, in cases where there isn't a meaningful raw value, you don't have to provide one.

```
enum Suit {
    case spades, hearts, diamonds, clubs

    func simpleDescription() -> String {
        switch self {
        case .spades:
            return "spades"
        case .hearts:
            return "hearts"
        case .diamonds:
            return "diamonds"
        case .clubs:
            return "clubs"
        }
    }
}
let hearts = Suit.hearts
let heartsDescription = hearts.simpleDescription()
```

> EXPERIMENT
>
> Add a `color()` method to `Suit` that returns "black" for spades and clubs, and returns "red" for hearts and diamonds.

Notice the two ways that the `hearts` case of the enumeration is referred to above: When assigning a value to the `hearts` constant, the enumeration case `Suit.hearts` is referred to by its full name because the constant doesn't have an explicit type specified. Inside the switch, the enumeration case is referred to by the abbreviated form `.hearts` because the value of `self` is already known to be a suit. You can use the abbreviated form anytime the value's type is already known.

If an enumeration has raw values, those values are determined as part of the declaration, which means every instance of a particular enumeration case always has the same raw value. Another choice for enumeration cases is to have values associated with the case—these values are determined when you make the instance, and they can be different for each instance of an enumeration case. You can think of the associated values as behaving like stored properties of the enumeration case instance. For example, consider the case of requesting the sunrise and sunset times from a server. The server either responds with the requested information, or it responds with a description of what went wrong.

```
enum ServerResponse {
    case result(String, String)
    case failure(String)
}

let success = ServerResponse.result("6:00 am", "8:09 pm")
let failure = ServerResponse.failure("Out of cheese.")

switch success {
case let .result(sunrise, sunset):
    print("Sunrise is at \(sunrise) and sunset is at \(sunset).")
case let .failure(message):
    print("Failure...  \(message)")
}
// Prints "Sunrise is at 6:00 am and sunset is at 8:09 pm."
```

> EXPERIMENT
>
> Add a third case to `ServerResponse` and to the switch.

Notice how the sunrise and sunset times are extracted from the `ServerResponse` value as part of matching the value against the switch cases.

Use `struct` to create a structure. Structures support many of the same behaviors as classes, including methods and initializers. One of the most important differences between structures and classes is that structures are always copied when they're passed around in your code, but classes are passed by reference.

```
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \
  (suit.simpleDescription())"
    }
}
let threeOfSpades = Card(rank: .three, suit: .spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

EXPERIMENT

Write a function that returns an array containing a full deck of cards, with one card of each combination of rank and suit.