Creating and Using Enums

We'll cover the following

In the previous example, we used `String` to represent a `suit`, but that's smelly. We don't need arbitrary values for `suit`. We may create a `sealed` class `Suit` and derived classes for each of the four permissible types. In fact, there can be only four values for `suit`. In short, we don't need classes, we simply need four instances. The `enum` class solves that problem elegantly.

Using enums

Here's an excerpt of code where the `suits` properties are converted to use an `enum` class `Suit` instead of being a `String`:

```
enum class Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

sealed class Card(val suit: Suit)

class Ace(suit: Suit) : Card(suit)

class King(suit: Suit) : Card(suit) {
  override fun toString() = "King of $suit"
}

//...
```

CardWithEnum.kt

Likewise, instead of passing a `String` to the constructor, we can now pass an instance of the `enum` class `Suit`:

```
// UseCardWithEnum.kt
println(process(Ace(Suit.DIAMONDS)))    // Ace of DIAMONDS
println(process(Queen(Suit.CLUBS)))    // Queen of CLUBS
println(process(Pip(Suit.SPADES, 2)))   // 2 of SPADES
println(process(Pip(Suit.HEARTS, 6)))  // 6 of HEARTS
```

The reference `Suit.DIAMONDS` represents an instance of class `Suit` and is a `static` property in the `enum` class.

Customizing enums

Not only are `enum` classes suitable for creating a bunch of enumerated values, we may customize them and iterate over them easily as well.

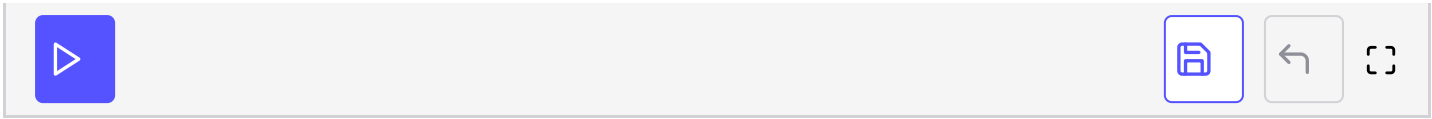
Given a String we can obtain the corresponding `enum` instance using the `valueOf()` method:

```
// iteratesuit.kts
val diamonds = Suit.valueOf("DIAMONDS")
```

If the `String` argument provided to `valueOf()` doesn't match for any of the values defined for the target `enum` class, then a runtime exception will be thrown.

We can also iterate over all the values for a `enum` class:

```
for (suit in Suit.values()) {
  println("${suit.name} -- ${suit.ordinal}") //CLUBS -- 0, etc.
}
```



iteratesuit.kts

The `values()` method provides an array of all the instances of the `enum` class. The `name` and `ordinal` properties of an `enum` instance will return the name and an index of the instance in the definition.

We may also hold state and provide methods in `enum` classes, but a semicolon has to separate the list of values from the methods. Let's add a `symbol` property for each value of the `Suit` `enum` and provide a method to return the `name` and `symbol`.

```
enum class Suit(val symbol: Char) {
  CLUBS('\u2663'),
  DIAMONDS('\u2666'),
  HEARTS('\u2665') {
    override fun display() = "${super.display()} $symbol"
  },
  SPADES('\u2660');
```

```
  open fun display() = "$symbol $name"
}
```

initlizeenum.kts

The `enum` class `Suit` now takes a parameter for the `symbol` property of type `Char`. It also has a method `display()` to return the `name` and `symbol` for a `suit`. Had we defined it as `abstract`, then each of the `suit` values would be required to implement that method. Had we defined it without `open`, then none of the suit values could override it. We took the middle ground here—those suits that want to override that method may do so and the other suits will use the implementation provided.

Each of the values `CLUBS`, `DIAMONDS`, and so on, pass the appropriate Unicode value to the constructor as argument. The `HEARTS` is special, as you'd suspect. Instead, being an instance of `Suit`, it's an anonymous inner class which overrides its own `display()` method. After the last `enum` value, `SPADES`, a semicolon indicates the end of values and the beginning of properties and methods of the `enum` class.

To see the above changes in action, let's iterate over the values of `Suit` and call the `display()` method for each value:

```
for (suit in Suit.values()) {
  println(suit.display())
}
```

The output below shows that the specialized `display()` method is called where available:

```
♣ CLUBS
♦ DIAMONDS
♥ HEARTS ♥
♠ SPADES
```

If you query for the `javaClass` on an instance of `Suit` with the call `suit.javaClass`, you'll see that `CLUBS`, `DIAMONDS`, and `SPADES` are instances of `Suit` but `HEARTS` is an instance of an anonymous inner class of `Suit`.

The Kotlin compiler takes care of minimally creating instances of the `enum` class where possible and specializes with anonymous inner classes when needed. Without regard to that, we can use `enums` in a type-safe manner, to create

expressive code that's easier to maintain.

The next lesson concludes the discussion for this chapter.