Enums

Principles of Functional Programming

Enums: Motivation

We have seen that classes can *aggregate several values* into a single abstraction. For instance, the `Rational` class aggregates a numerator and a denominator.

Conversely, how could we define an abstraction *accepting alternative values*?

Example

Define a `Color` type with values `Red`, `Green`, `Blue`, and `Magenta`.

Color Objects

```
trait Color
object Red extends Color
object Green extends Color
object Blue extends Color
object Magenta extends Color
```

Color Objects

```
trait Color
object Red extends Color
object Green extends Color
object Blue extends Color
object Magenta extends Color
```

This is getting tedious!

Is there a simpler way?

Enums

As a simpler and shorter alternative, we can define a type with its values in an *enum*:

```
enum Color:
  case Red, Green, Blue, Magenta
```

This definition introduces:

- A new *type*, named `Color`.
- Four possible *values* for this type, `Color.Red`, `Color.Green`, `Color.Blue`, and `Color.Magenta`.

Enumerate the Values of an Enumeration

It is possible to enumerate all the values of an enum by calling the `values` operation on the enum companion object:

```
Color.values               > Array(Red, Green, Blue, Magenta)
val c = Color.Green        > c: Color = Green
c == Color.values(1)       > true
```

Discriminate the Values of an Enumeration

You can discriminate between the values of an enum by using a *match* expression:

```scala
import Color._
def isPrimary(color: Color): Boolean =
  color match
    case Red | Green | Blue => true
    case Magenta => false
```

Match Syntax

- `match` is followed by a sequence of *cases*, `case value => expr`.
- Each case associates an *expression* `expr` with a *constant* `value`.
- Default cases are written with an underscore, e.g.

```
def isPrimary(color: Color): Boolean = color match
  case Magenta => false
  case _ => true
```

We will see later that pattern matching can do more than discriminating enums.

Enumerations Can Take Parameters

```scala
enum Vehicle(val numberOfWheels: Int) {
  case Unicycle extends Vehicle(1)
  case Bicycle  extends Vehicle(2)
  case Car      extends Vehicle(4)
}
```

- Enumeration cases that pass parameters have to use an explicit `extends` clause

Enumerations Are Shorthands for Classes and Objects

The `Color` enum is expanded by the Scala compiler to roughly the following structure:

```scala
abstract class Color
object Color {
  val Red     = Color()
  val Green   = Color()
  val Blue    = Color()
  val Magenta = Color()
  ...
}
```

(plus some helper methods in `Color` and its companion object)