Enumerations

Effective Programming in Scala

Enumerations

Sometimes, the alternative values of a type are not *classes* of values, but singleton values. Scala 3 provides a special construct for this use case:

```
enum PrimaryColor:
  case Red, Blue, Green
```

This defines a type `PrimaryColor` with exactly three possible values: `PrimaryColor.Red`, `PrimaryColor.Blue`, or `PrimaryColor.Green`.

Pattern Matching on Enumerations

We use **literal patterns** on enumerations:

```
def isProblematicForColorBlindPeople(color: PrimaryColor): Boolean =
  color match
    case PrimaryColor.Red   => true
    case PrimaryColor.Blue  => false
    case PrimaryColor.Green => true
```

Pattern Matching on Enumerations

We use **literal patterns** on enumerations:

```
def isProblematicForColorBlindPeople(color: PrimaryColor): Boolean =
  color match
    case PrimaryColor.Red   => true
    case PrimaryColor.Blue  => false
    case PrimaryColor.Green => true
```

Example of use:

```
isProblematicForColorBlindPeople(PrimaryColor.Blue) // : Boolean = false
```

Enumerating the Values of an Enumeration

Enumerations provide a `values` operation that enumerates all their possible values:

```
PrimaryColor.values // Array(PrimaryColor.Red, PrimaryColor.Blue,
                    //   PrimaryColor.Green)
```

We will see next week how to use collections such as `Array`.

Finding a Value From its Label

You can find an enumeration value from its `String` label:

```
PrimaryColor.valueOf("Green") // : PrimaryColor = PrimaryColor.Green
```

If the argument passed to `PrimaryColor.valueOf` is not a valid enumeration label, it's a runtime error.

Enumerations (Scala 2 Compatibility)

It is worth noting that an enum definition is just a syntactic sugar for a sealed trait and case objects that extend this trait:

```
enum PrimaryColor:
  case Red, Blue, Green
```

Is equivalent to:

```
sealed trait PrimaryColor
object PrimaryColor:
  case object Red extends PrimaryColor
  case object Blue extends PrimaryColor
  case object Green extends PrimaryColor
  val values = Array(Red, Blue, Green)
  def valueOf(label: String): PrimaryColor = ...
```

Aside: Object Definitions

```
sealed trait PrimaryColor
object PrimaryColor:
  case object Red extends PrimaryColor
  case object Blue extends PrimaryColor
  case object Green extends PrimaryColor
```

This code defines:

- a type `PrimaryColor`, with a fixed number of possible values,
- an **object** `PrimaryColor`, containing the definitions of the values of the type `PrimaryColor`,
- the three possible values, `Red`, `Blue`, and `Green` for `PrimaryColor`.

Aside: Companion Objects

Note that the same name `PrimaryColor` can refer to either a type or a value, depending on where it is used.

It refers to the object `PrimaryColor` when it is used on the right-hand side of a definition, or when it is passed as an argument to an operation:

```
val color = PrimaryColor.Green
isProblematicForColorBlindPeople(PrimaryColor.Green)
```

It refers to the type `PrimaryColor` when it is used on the right-hand side of a type annotation:

```
val color: PrimaryColor = PrimaryColor.Green
```

Aside: Case Objects

A `case object` that extends a `sealed trait` defines one possible value for that trait.

Unlike a `case class`, which defines a constructor, a `case object` **is** already a value.

Like with enums, we use literal patterns to match on case objects:

```
def isProblematicForColorBlindPeople(color: PrimaryColor): Boolean =
  color match
    case PrimaryColor.Red   => true
    case PrimaryColor.Blue  => false
    case PrimaryColor.Green => true
```

Summary

`enum` is a convenient construct for modeling a type that has one of several possible singleton values.

An enumeration defines a type and its companion object, which in turn defines the possible values of the enumeration, and additional operations such as `values` and `valueOf`.