Enums

Principles of Functional Programming

Pure Data

In the previous sessions, you have learned how to model data with class hierarchies.

Classes are essentially bundles of functions operating on some common values represented as fields.

They are a very useful abstraction, since they allow encapsulation of data.

But sometimes we just need to compose and decompose *pure data* without any associated functions.

Case classes and pattern matching work well for this task.

A Case Class Hierarchy

Here's our case class hierarchy for expressions again:

```
trait Expr
object Expr:
  case class Var(s: String) extends Expr
  case class Number(n: Int) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr
```

This time we have put all case classes in the `Expr` companion object, in order not to pollute the global namespace.

So it's `Expr.Number(1)` instead of `Number(1)`, for example.

One can still "pull out" all the cases using an import.

```
import Expr.*
```

A Case Class Hierarchy

Here's our case class hierarchy for expressions again:

```
trait Expr
object Expr:
  case class Var(s: String) extends Expr
  case class Number(n: Int) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr
```

Pure data definitions like these are called *algebraic data types*, or ADTs for short.

They are very common in functional programming.

To make them even more convenient, Scala offers some special syntax.

Enums for ADTs

An *enum* enumerates all the cases of an ADT *and nothing else*.

Example

```
enum Expr:
  case Var(s: String)
  case Number(n: Int)
  case Sum(e1: Expr, e2: Expr)
  case Prod(e1: Expr, e2: Expr)
```

This enum is equivalent to the case class hierarchy on the previous slide, but is shorter, since it avoids the repetitive `class ... extends Expr` notation.

Pattern Matching on ADTs

Match expressions can be used on enums as usual.

For instance, to print expressions with proper parameterization:

```
def show(e: Expr): String = e match
  case Expr.Var(x) => x
  case Expr.Number(n) => n.toString
  case Expr.Sum(a, b) => s"${show(a)} + ${show(a)}}"
  case Expr.Prod(a, b) => s"${showP(a)} * ${showP(a)}"

def showP(e: Expr): String = e match
  case e: Sum => s"(${show(expr)})"
  case _ => show(expr)
```

Simple Enums

Cases of an `enum` can also be simple values, without any parameters.

Example

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

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

We can also combine several simple cases in one list:

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

Pattern Matching on Simple Enums

For pattern matching, simple cases count as constants:

```
enum DayOfWeek:
  case Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

import DayOfWeek.*

def isWeekend(day: DayOfWeek) = day match
  case Saturday | Sunday => true
  case _ => false
```

More Fun With Enums

Enumerations can take parameters and can define methods.

Example:

```
enum Direction(val dx: Int, val dy: Int):
  case Right extends Direction( 1,  0)
  case Up    extends Direction( 0,  1)
  case Left  extends Direction(-1,  0)
  case Down  extends Direction( 0, -1)

  def leftTurn = Direction.values((ordinal + 1) % 4)
end Direction

val r = Direction.Right
val u = x.leftTurn       // u = Up
val v = (u.dx, u.dy)     // v = (1, 0)
```

More Fun With Enums

Notes:

- Enumeration cases that pass parameters have to use an explicit `extends` clause
- The expression `e.ordinal` gives the ordinal value of the enum case `e`. Cases start with zero and are numbered consecutively.
- `values` is an immutable array in the companion object of an enum that contains all enum values.
- Only simple cases have `ordinal` numbers and show up in `values`, parameterized cases do not.

Enumerations Are Shorthands for Classes and Objects

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

```
abstract class Direction(val dx: Int, val dy: Int):
  def rightTurn = Direction.values((ordinal - 1) % 4)
object Direction:
  val Right = new Direction( 1,  0) {}
  val Up    = new Direction( 0,  1) {}
  val Left  = new Direction(-1,  0) {}
  val Down  = new Direction( 0, -1) {}
end Direction
```

There are also compiler-defined helper methods `ordinal` in the class and `values` and `valueOf` in the companion object.

Domain Modeling

ADTs and enums are particularly useful for domain modelling tasks where one needs to define a large number of data types without attaching operations.

Example: Modelling payment methods.

```
enum PaymentMethod:
  case CreditCard(kind: Card, holder: String, number: Long, expires: Date)
  case PayPal(email: String)
  case Cash

enum Card:
  case Visa, Mastercard, Amex
```

Summary

In this unit, we covered two uses of `enum` definitions:

- as a shorthand for hierarchies of case classes,
- as a way to define data types accepting alternative values,

The two cases can be combined: an enum can comprise parameterized and simple cases at the same time.

Enums are typically used for pure data, where all operations on such data are defined elsewhere.