Enums

Pure Data

- 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 is our case class hierarchy for expressions:

```
trait Expression
object Expression:
  case class Variable(s: string) extends Expression
  case class Number(n: Int) extends Expression
  case class Sum(e1: Expression, e2: Expression) extends Expression
  case class Product(e1: Expression, e2: Expression) extends Expression
```

- This time we have put all case classes in the Expression companion object, in order not to pollute the global namespace
- One can still pull out all the cases using an import
- Note that there are no methods in this definition, all we've done is define a base trait and case classes that extend the base trait
- Pure data definitions like these are called algebraic data types (ADTs)
- They are very common in functional programming, so Scala offers some special syntax

Enums for ADTs

- An ***enum*** enumerates all the cases of an ADT and nothing else
- Here is the redefinition of the example written above:

```
enum Expression:
  case Variable(s: string)
  case Number(n: Int)
  case Sum(e1: Expression, e2: Expression)
  case Product(e1: Expression, e2: Expression)
```

Pattern Matching on ADTs

- Match expressions can be used on enums as usual
- For instance, to print expressions with proper parametrization:

```
def show(e: Expression): String = e match
  case Variable(s) => s
  case Number(n) => n.toString
  case Sum(e1, e2) => s"${show(e1)} + ${show(e2)}"
  case Product(e1, e2) => s"${showP(e1)} + ${showP(e2)}"
```

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

Simple Enums

- Cases of an enum can also be simple values, without any parameters
- Example:

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

- We can also combine several simple cases in one list:

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

- For pattern matching, simple cases count as constants

Parameters and Methods inside an Enum

- 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)
```

- 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 contain all enum values
- Only simple cases have ordinal numbers and show up in values, parametrized cases do not
- The Direction enum is expanded by the Scala complier 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: String, expires: Date)
  case PayPal(email: String)
  case Cash

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

- 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