Business Logic

Effective Programming in Scala

Julien Richard-Foy

Business Logic

The previous lectures introduced language features such as **methods**, **conditions**, **case classes**, and **sealed traits**.

These features are the building blocks for modeling the **business domain** of a program and implementing its **business logic**.

This lecture shows how to use them in larger examples and provides some advice on how to manage the complexity of a programming problem.

1st Example: Video Streaming Footprint

The impact on the environment of video streaming varies according to the video quality and duration and the type of internet connection.

A simplified model of footprint is the following:

- Data centers consume 0.000072 kWh / MB of video,
- Mobile networks consume 0.00088 kWh / MB of video whereas fixed networks consume 0.00043 kWh / MB of video,
- Producing 1 kWh of electricity emits 0.5 kg of CO_2 (world average).

1st Example: Video Streaming Footprint

The impact on the environment of video streaming varies according to the video quality and duration and the type of internet connection.

A simplified model of footprint is the following:

- Data centers consume 0.000072 kWh / MB of video,
- Mobile networks consume 0.00088 kWh / MB of video whereas fixed networks consume 0.00043 kWh / MB of video,
- Producing 1 kWh of electricity emits 0.5 kg of CO_2 (world average).

What is the impact of watching a 30 minutes series:

1. in high definition (0.6 MB/s) from a mobile phone?
2. in low definition (0.3 MB/s) from a desktop computer?

Video Streaming Footprint: Domain

One way to start is to sketch a model of the concepts that are relevant to our problem.

In our case, we want to talk about the experience of watching videos of some duration, some quality, and from some type of network.

```
enum Network:
  case Fixed, Mobile

case class Experience(duration: Int, definition: Double, network: Network)
```

Video Streaming Footprint: Instances

We can validate that our model is expressive enough by constructing instances of cases we are interested in.

```
val lowQuality  = 0.3 // MB/s
val highQuality = 0.6 // MB/s

val thirtyMinutes = 30 * 60 // seconds

val highQualityAndMobile = Experience(thirtyMinutes, highQuality, Network.Mobile)
val lowQualityAndFixed   = Experience(thirtyMinutes, lowQuality, Network.Fixed)
```

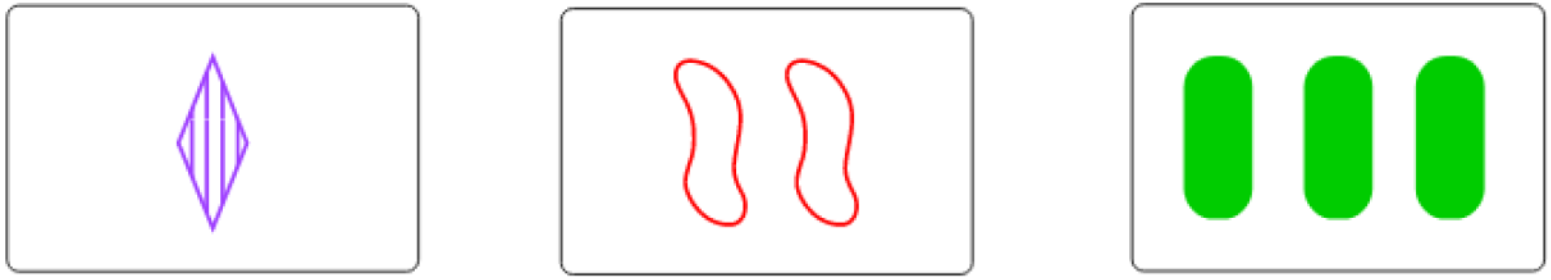
Video Streaming Footprint: Logic

```
val dataCenterEnergy = 0.000072   // kWh/MB
val kgCO2perkWh      = 0.5        // kg CO2 emitted per kWh consumed
def networkEnergy(network: Network) = network match
  case Network.Mobile => 0.00088  // kWh/MB
  case Network.Fixed  => 0.00043  // kWh/MB
def footprint(experience: Experience): Double =
  val megabytes = experience.duration * experience.definition
  val energy    = dataCenterEnergy + networkEnergy(experience.network)
  energy * megabytes * kgCO2perkWh

footprint(highQualityAndMobile)  // 0.51408 kg of CO2
footprint(lowQualityAndFixed)    // 0.13554 kg of CO2
```

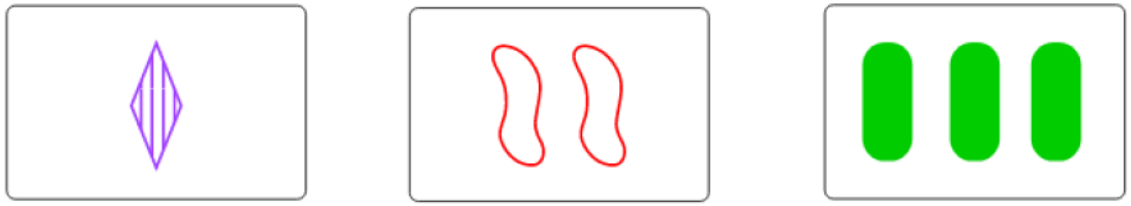
2nd Example: Set Card Game

Set[1] is a card game where three cards can make up a "set" if they satisfy some properties.



[1] https://en.wikipedia.org/wiki/Set_(card_game)

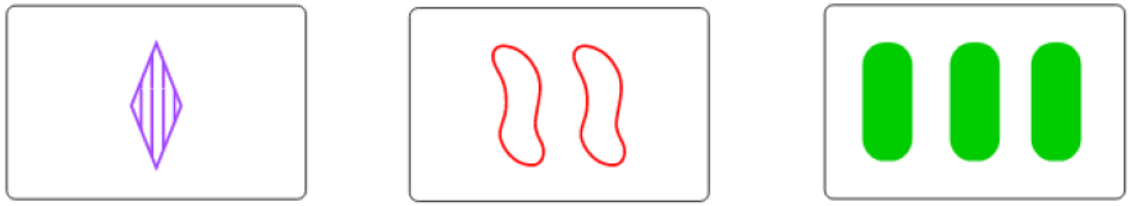
Set: Rules (1)



Cards have **four features** across **three possibilities** for each kind of feature:

- shape: diamond, squiggle, or oval
- number of shapes: 1, 2, or 3
- shading: solid, striped, or open
- color: red, green, or purple

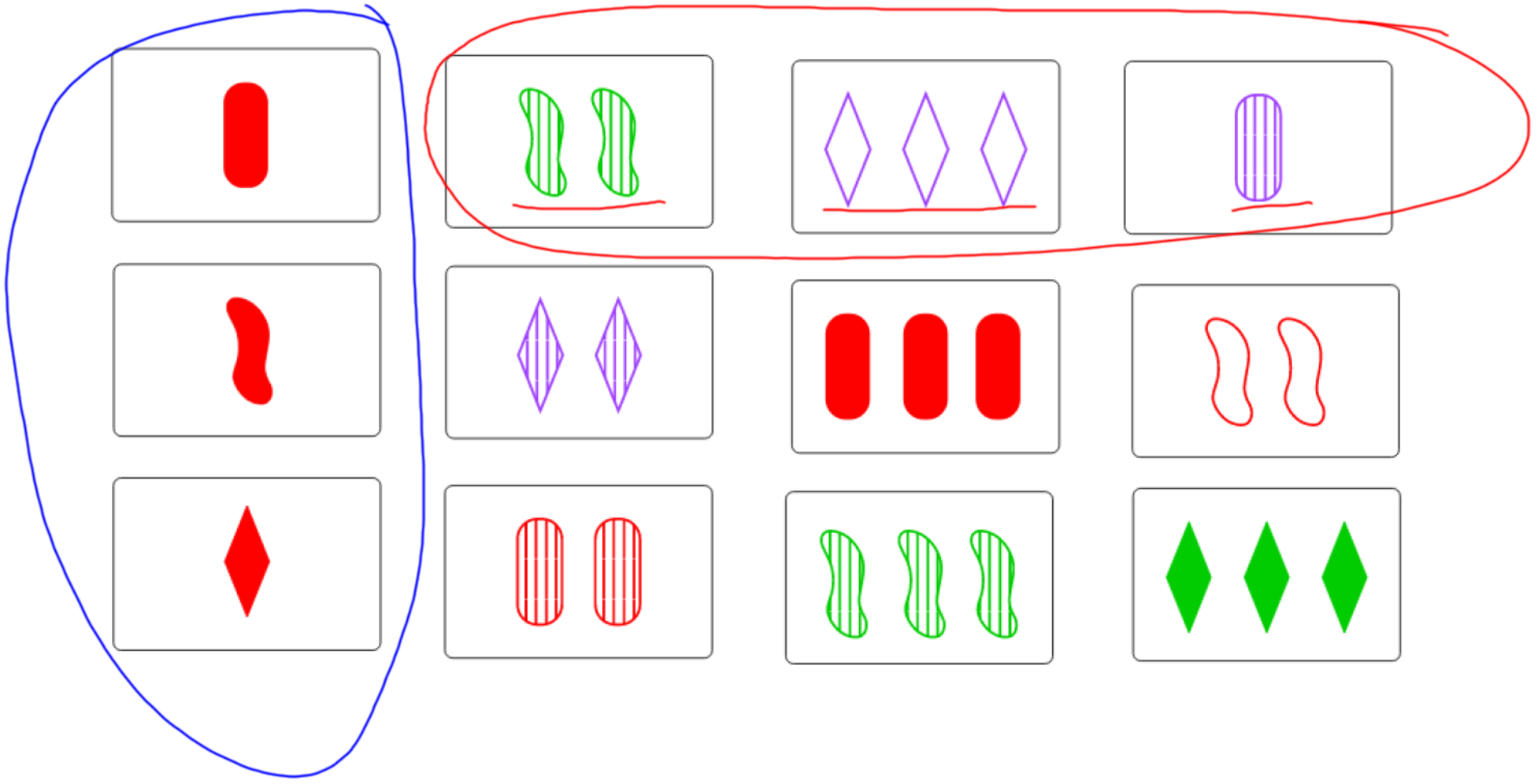
Set: Rules (2)



Three cards make up a **set** if and only if all the following properties are satisfied:

- they all have the same number of shapes or have three different numbers of shapes,
- they all have the same shape or have three different shapes,
- they all have the same shading or have three different shadings,
- they all have the same color or have three different colors.

Set: Let's Practice!



Set: Exercise

Implement a method that takes three cards of the set game and returns whether they form a valid “set” or not.

Set: Domain

```scala
case class Card(shape: Shape, number: Int, color: Color, shading: Shading)
```

Set: Domain

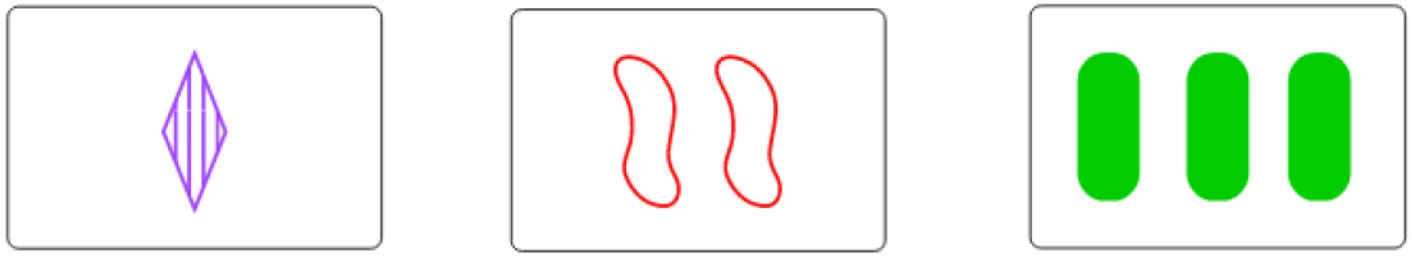
```scala
case class Card(shape: Shape, number: Int, color: Color, shading: Shading)

enum Shape:
  case Diamond, Squiggle, Oval

enum Color:
  case Red, Green, Purple

enum Shading:
  case Open, Striped, Solid
```

Set: Instances



```
val deck = List(
  Card(Shape.Diamond,  1, Color.Purple, Shading.Striped),
  Card(Shape.Squiggle, 2, Color.Red,    Shading.Open),
  Card(Shape.Oval,     3, Color.Green,  Shading.Solid)
)
```

Set: Nonsensical Instances

```
val nonsensicalCard = Card(Shape.Diamond, 0, Color.Purple, Shading.Striped)
```

Set: Nonsensical Instances

```
val nonsensicalCard = Card(Shape.Diamond, 0, Color.Purple, Shading.Striped)
```

We can construct cards with zero shapes, or a negative number of shapes, or more than three shapes!

Set: Stricter Model

```
case class Card(shape: Shape, number: Number, color: Color, shading: Shading)

enum Number:
  case One, Two, Three
```

Set: Logic

```
def isValidSet(card1: Card, card2: Card, card3: Card): Boolean = ???
```

Set: Logic

```
def isValidSet(card1: Card, card2: Card, card3: Card): Boolean =
  checkShapeProperty(card1, card2, card3)   &&
  checkNumberProperty(card1, card2, card3)  &&
  checkColorProperty(card1, card2, card3)   &&
  checkShadingProperty(card1, card2, card3)
```

Set: Logic

```
def isValidSet(card1: Card, card2: Card, card3: Card): Boolean =
  checkShapeProperty(card1, card2, card3)   &&
  checkNumberProperty(card1, card2, card3)  &&
  checkColorProperty(card1, card2, card3)   &&
  checkShadingProperty(card1, card2, card3)

def checkShapeProperty(card1: Card, card2: Card, card3: Card): Boolean =
  def allSame =
    card1.shape == card2.shape && card1.shape == card3.shape
  def allDifferent =
    card1.shape != card2.shape &&
    card1.shape != card3.shape &&
    card2.shape != card3.shape
  allSame || allDifferent
```

Set: Play!

```
isValidSet(
  Card(Shape.Diamond,  Number.One,   Color.Purple, Shading.Striped),
  Card(Shape.Squiggle, Number.Two,   Color.Purple, Shading.Open),
  Card(Shape.Oval,     Number.Three, Color.Purple, Shading.Solid)
)
// true

isValidSet(
  Card(Shape.Diamond,  Number.Two,   Color.Purple, Shading.Striped),
  Card(Shape.Squiggle, Number.Two,   Color.Purple, Shading.Open),
  Card(Shape.Oval,     Number.Three, Color.Purple, Shading.Solid)
)
// false
```

Modeling Methodology (1)

There is no systematic methodology: often, a same set of concepts can be modeled in multiple ways.

But here are some advice.

Modeling Methodology (1)

There is no systematic methodology: often, a same set of concepts can be modeled in multiple ways.

But here are some advice.

- Identify the concepts (in general, *nouns*) that you are interested in
- Identify the relations between them
  - Does a concept *belong to* another one?
    - e.g., "a rectangle *has* a width and a height"
    - e.g., "a 'post message' action *has* a channel and a message"
  - Does a concept *generalize* another one?
    - e.g., "a shape can either *be* a rectangle or a circle"
    - e.g., "a subscription *is* a possible action"

Modeling Methodology (2)

- Translate each concept into a type definition
  - Concepts belonging to others become **parameters** of case classes
  - Concepts generalizing others become **sealed traits**
- Check that you can construct meaningful values from your model
  - Conversely, check that you can **not** construct nonsensical values from your model
- Sometimes you need several iterations of domain model and logic implementation to find the right solution

Summary

Reify the concepts (nouns) of a business domain into case classes and sealed traits.

Check that values derived from your model are valid by construction and that they are meaningful for your problem.

Implement the business logic as operations on your model.