

# Model View Controller meets Monad

“ ***It is all about composition*** ”



# Goals

- show an end-to-end **functional** application
- leverage some well-consolidated functional libraries
- understand limitations (if any) and the improvements

# Repository

<https://github.com/cric96/scala-functional-gui>

# Target Application

## Tic Tac Toe



# OOP Design

## Everything is an object

Clean interface, state incapsulated, side effect as methods call.

Let's try to build an *old-fashion* application 😊

# Model

```

/* two players (X, O) */
enum Player {
    X, O, None;
}
/* a board 3x3 */
interface TicTacToe {
    Player get(int X, int Y);
    TicTacToe (or void??) update(int x, int y, Player p);
    boolean isOver;
    Player getTurn;
}

```

# View

```
//a là view model?  
interface ViewBoard {  
    List<String> getRow(int row);  
    List<List<String>> getAllBoard();  
}  
  
interface View extends ClickCellSource {  
    void render(ViewBoard board);  
    void winner(String player);  
}
```



# Put some design pattern 😊

```
public interface ClickCellSource {  
    void attach(Observer observer);  
    interface Observer {  
        void notify(int X, int Y);  
    }  
}
```

# Controller

```
public interface Game extends ClickCellSource.Observer {
    void start();
}

public class TicTacToeGame implements Game {
    private final TicTacToe ticTacToe;
    private final TicTacToeView ticTacToeView;

    public static TicTacToeGame playWith(
        final TicTacToe ticTacToe,
        final TicTacToeView ticTacToeView) {...}
    ....
}
```

# Putting all together

```
public static void main(String[] args) {
    final TicTacToeView view = SwingView.createAndShow(800, 600);
    final TicTacToe model = TicTacToeFactory.startX();
    final Game game = TicTacToeGame.playWith(model, view);
    game.start();
}
```

# **Clean enough right?**

## **What do you think?**

Try to rethink using "functional" abstractions  
(Monads? Functions? Algebraic Data Type?)

# Libraries

- **Cats**: provides abstractions for functional programming in the Scala programming language
- **Monix**: high-performance Scala / Scala.js library for composing **asynchronous, event-based** programs

# Task

“ Task represents a specification for a possibly lazy or asynchronous computation, which when executed will produce an  $A$  as a result, along with possible side-effects. ”

# What does it refer you to?

```
trait Task[+A] {  
  final def flatMap[B](f: A => Task[B]): Task[B] = ...  
  final def map[B](f : A => B): Task[B] = ...  
  //some interesting extesions  
  def memoize: Task[A] = ...  
}  
object Task {  
  def pure[A](a : A) : Task[A]  
  def defer[A](a : Task[A]) : Task[A]  
}
```

# A Little taste

```
val scheduler = monix.execution.Scheduler.Implicits.global

def someComputation(data : Long) : Task[Long] =
  Task.pure(data * 1000)

def log(value : String) : Task[Unit] = Task { println(value) }

val main = for {
  data <- someComputation(4)
  _ <- log(s"computations ends with value $data")
} yield (data)

main.runToFuture(scheduler)
```



# Observable

“ a data type for modelling and processing asynchronous and reactive streaming of events with non-blocking back-pressure. ”

We use it to implement **Functional Reactive Programming**

# Functional Reactive Programming

“ The program is expressed as a reaction to its inputs,  
or as a flow of data. ”

- Functional Reactive Programming

# Functional Reactive Programming

Out of this talk, If you are interested [Conal Elliot](#) makes a lot of materials about this topics.

# Books

1. **Scala with Cats Book**
2. **Category Theory for Programmers**
3. **Functional Reactive Programming**

# References

