First, we will initialise an initialState which will store all the initial attributes of the game to avoid global mutable state and maintain purity of the code. This InitialState is the model of the Model-View-Controller architecture. We are using this architecture because it can help us to separate the pure part of the code and the impure part of updating many svg elements. Then we will be using scan by taking the initialState as input with the reducer reduceState and will return a new output state object based on the changes on initial state. Since it only returns a new state instead of modifying the initial state, it helps to maintain the purity of the code.

I also create a constant class that stores all the constant values inside so the code will be easier to read and we can simply change all of the occurrences with one edit.

Then, I declare the type of state with readonly to make sure it is immutable and can only be read. Other than that, we also use readonly keyword for the array inside the State to make sure the array is also immutable. We store car and plank as a nested array to indicate different elements in different rows in the game and also declare plank and car as the same type Object because they both have similar attributes and behaviour. I use multiple recursive functions (createCars,createPlanks and createTurtle) to create multiple rows of objects to initialise the attribute in initialState to avoid repetition.

Then I use observeKey to observe the no repeating keydown keyboard event of different keys then creating and returning action classes. These are the controllers of the architecture and we make them return classes then we can manipulate the state by using instanceof to detect the controller inside reduceState. Using observables can help us to handle the multiple values asynchronously and reactively.

Inside the reduceState, score function will take the state object updated after receiving move action class and check if the frog is entering an unfilled score area. It will return a new state object with new userScore and modified scoreAreasArray. There is also a function boundary that will make sure the frog will not exceed the background by checking the position of the frog of the state object and the distance moved. There are four functions which are timer, dropIntoRiver, handleCollision and roundCompleted to handle the state when reduceState receives Tick action class. These four functions will check if a certain condition meets to restart the game by returning the initial state. To achieve that frog will dive into water, I declare 1 boolean diveIntoWater and change it periodically by tracking the timer of the state.There is also a function outOfBound to make the object moving outside the screen to reappear from the opposite site. Other than that, reduceState will return initialState when it receives a restart action class.

Then, we subscribe to the stream with updateView. This is the view part of the architecture and this part is impure. We can use effectful code to update the svg element in this part. We call this at the end of our observable to make sure we don't mutate any state in the observable. I render the svg elements with their new attributes in this part. Then, for the object that is not created in html, I will check if it is null, if it is null, we render it into the game, else we render it with its new attributes. I didn't use unsubscribe and just return the state into initial state when game over because I will need another subscription to observe the restart input when the original subscription is unsubscribed.