Design Decisions

- Chose to implement a RNG class instead of using Math.random() as it is impure.
    - Because it will produce different output each time the function is called.
    - The RNG class can be made completely pure by providing a constant seed.
    - In that case, the sequence of the tetrominos dropping will be the same each time the page is reloaded.
    - To prevent the players from being able to memorize the sequence, `Date().getMilliseconds()` is used as the initial seed.
    - This function is impure as the output for each call is different.
    - The RNG class implemented has deterministic randomness.
    - It is pure because the `next()` function doesn't modify the class attributes but returns a new RNG class object.
    - It also produce the same output given the same input.
    - This implementation is chosen over using observable stream of random numbers because this implementation is more lightweight and likely more efficient than setting up and tearing down observable streams.
- A grid made up of 2-dimensional array where each cell can either be `null` or a string representing the color of the block that occupies that cell.
    - After the block currently falling collide with the floor or other blocks, which is detected by `isCollision()`, the block will integrate into part of the grid using `updateGrid()`.
    - The string that represents color of that cell will be used to render out the game board.
    - Updating the grid maintains the purity of the program as it doesn't modify any external states, it simply just returns a new grid.
    - This reason I chose this implementation over others is, after the block has 'landed', we can just integrate it into the grid, forget about that block and spawn the new block.
    - This requires reasonably less operation and is more efficient compared to other implementation to ensure the smoothness of the game.

State management, Purity, Function Programming

- State is managed throughout the game using the apply function in each class implementing `Action` interface.
    - The apply function will not modify the old state but produce and return a new State object with some changes caused by action of the player.
    - After a new state is produced, it will be passed into the scan operator to then render the changes to let the player visualize.
    - It maintains purity by separating the side effects so that all the side effects are contained in `render` instead of in the state processing.
    - We maintain the immutability of the state objects so that it's easier to maintain.
    - As the separation of concerns and the clear flow of data make the code easier to understand, modify, and extend.
    - Pure functions are also easier to test for errors because they don't have hidden side effects.
- Mutable and loops are avoided to maintain purity.
    - It can be hard to track the state of mutable sometimes, especially when trying to solve bugs.

    - Loops avoided because it depends on mutable such as loop counters. Recursion is used instead of loops to achieve the same outcome.
    - Build in higher-order functions such as `map`, `filter` are also used in the program.
- Tetris wall kicks is also implemented.
    - It first determines the rotated shape of the block using `rotateMatrix` function.
    - If that results in a collision, it will use `tryRotationOffsets` to find a valid offset for the rotated block.
    - In short, the game tries shifting the block left, right, up or down to see if the rotation fits in any of those positions.
    - If a valid offset is found, it updates the game state using the `updateState` function.
    - Otherwise, the rotation attempt is ignored.

RxJS, Observable, FRP

- BehaviorSubject is used for `state$` so that every time there's a change in game level, it will update the tick rate of the game.
- The player controls implemented using Observables
    - `fromEvent` turns keypress events into a stream of events.
    - It is then filtered into different steams based on the key code.
    - Each streams is mapped into respective game action.
    - At last all streams are merged and `scan` operator is used to update the game state.
    - The final state is then passed to a `subscribe` function which renders the game state.
- Restart functionality is also implemented.
    - By listening to 'R' key press and mapping it to a Observable stream of Restart Action.
    - In the end, these Observable streams will get merged into one unified stream of actions.
    - This stream will get `pipe` into a `scan` operator where the `apply()` function of each Action class will be carried out and a resulting state will be returned and rendered.
    - Restart can be achieved by not unsubscribing to `source$` after the `gameEnd` condition becomes true.
    - Instead it will just show the `gameOver` element and a text to hint the player to press 'R' to restart.
    - To restart, the game state is reset to its initial state.
    - Except the high score is passed on even if a player restarts the game, this is done by passing the high score into the output state object.
    - The RNG is also kept the same to ensure that no matter how many times the player decides to restart, the sequence of blocks will remain consistent after reloading the page given that the seed used is constant.
- An additional SPACEBAR control is also introduced using similar implementation as other controls.
    - It allows the player to drop a block straight down.
    - It calculates the new Y coordinate of the block using recursion in `findNewY` function.
    - It checks whether placing the block one step further down would cause a collision.
    - If there's no collision, the method calls itself recursively with the Y-coordinate incremented by 1, effectively checking the next position down.
    - The block is then moved to that position using the `Move` action, given the new Y coordinates.