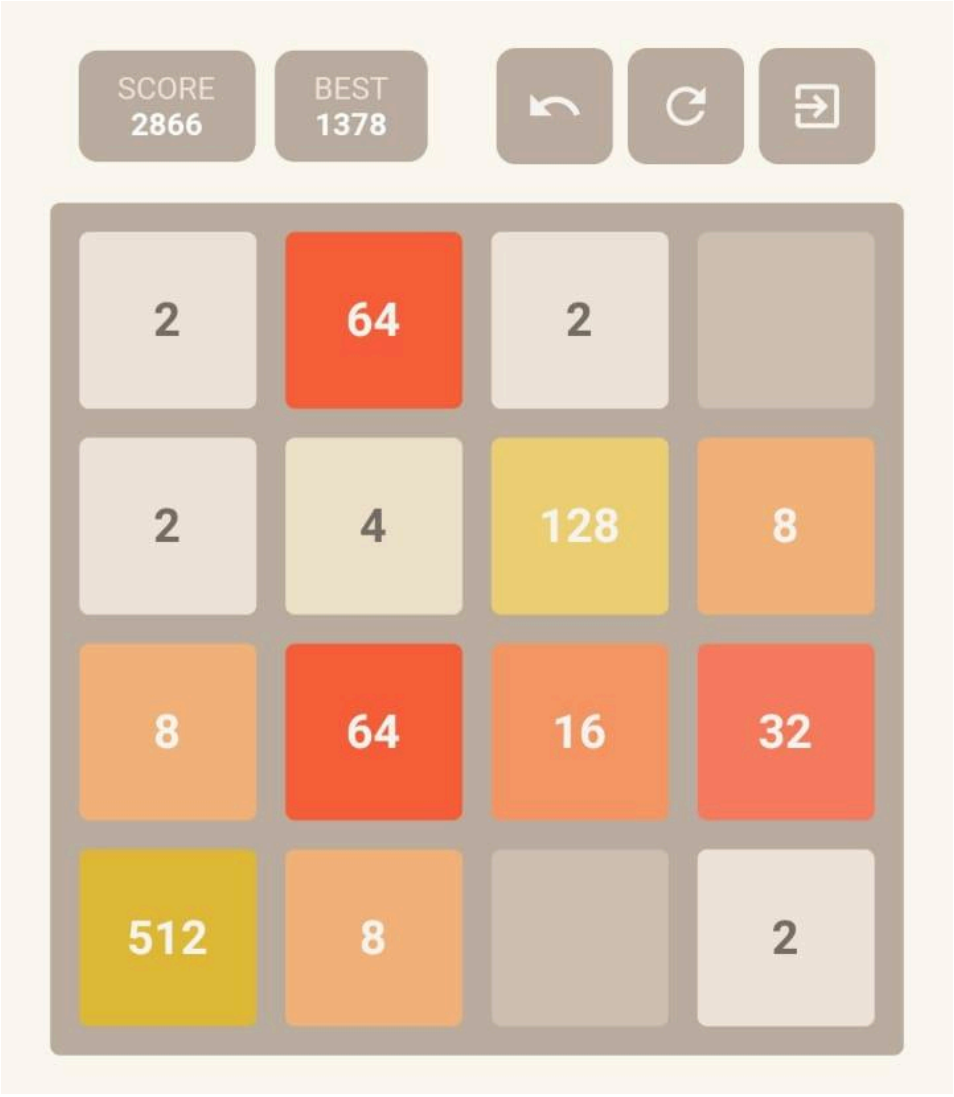
2048

Proyecto final de Flutter

Juego móvil de ingenio con persistencia de datos a través de JSON/Hive



Juan Martínez Justicia | IES Pablo Picasso
2º DAM | Programación Multimedia y Desarrollo de Aplicaciones Móviles

Índice

1. Descripción del proyecto

2048 es un juego que emplea un panel de **16 casillas (4x4)** para deslizar casillas, con un **valor inicial de 2**, que van apareciendo aleatoriamente tras cada movimiento. **El objetivo es fusionarlas por parejas** para alcanzar una casilla con valor 2048, el mayor que se puede obtener en un panel del tamaño dado.

La aplicación tendrá 2 pantallas:

- **Una de inicio** con el título del juego y los créditos, y con botones para comenzar el juego y salir de la app.
- **La pantalla del juego** a la que se accede con el botón previamente mencionado, en la que se desarrolla la propia acción del mismo.

Para la pantalla de inicio tan solo necesitaremos botones y texto, pero para la del juego vamos a utilizar diversos widgets personalizados:

- **Panel de 16 casillas**.
- **Casillas**, que tendremos que controlar en todo momento para saber hacia dónde moverlas y cuándo fusionarlas.
- **Panel de puntuación:** tanto la actual como el récord deberán mostrarse. El récord también deberá persistir aun cerrando la aplicación, para lo que utilizaremos **JSON**. La puntuación irá aumentando en función de los valores fusionados: si dos casillas de valor 4 se fusionan en una casilla de valor 8, se sumarán 4 puntos a la puntuación actual.
- **Botones con texto**.
- **Botones con iconos** de **Rehacer** (deshacer acción), **Reiniciar** y **Salir** (al menú principal).

Además contaremos con controladores para el panel y las rondas que se realicen, y una clase donde almacenaremos los colores que emplearemos para el juego y así tenerlos fácilmente accesibles.

La aplicación se exportará como el archivo **2048.apk**, que instalará la aplicación bajo el nombre **2048**.

2.a. Estructura de clases: Raíz

Tras haber descrito las pantallas y los widgets que debemos realizar, la estructura del proyecto quedaría de la siguiente manera:

lib: es la carpeta raíz de las clases **DART** del proyecto.

lib/**main.dart**: es la clase que se cargará al iniciar la aplicación. Contiene la pantalla de inicio, desde la que se puede acceder a la pantalla del juego.

lib/**game**: es la carpeta que contiene todos los archivos relacionados a la acción del juego, incluída la propia pantalla del juego.

lib/game/**game.dart**: es la clase que contiene la pantalla del juego. En ella se dispondrán los distintos widgets personalizados que realizaremos.

lib/game/**palette.dart**: esta clase contiene todos los colores que utilizaremos en la aplicación con el objetivo de facilitar el acceso a los mismos.

lib/game/**exporter.dart**: esta clase nos servirá como atajo para importar todos los widgets personalizados allá donde se necesiten (game.dart u otros widgets) con una sola línea.

2.b. Estructura de clases: Modelos

lib/game/**models**: es la carpeta que contiene todos los widgets personalizados.

lib/game/models/**board.dart**: el panel del juego. Contiene 3 clases diferentes; todas referidas al panel, pero con implicaciones diferentes (visual, control, relación a casillas). Estas son:

- **EmptyBoard**: calcula el tamaño del panel y conforma su "esqueleto".
- **Board**: contiene la información pertinente a la lógica del juego: casillas activas, puntuaciones, comprobación de fin de juego al cabo de cada ronda...
- **SquareBoard**: calcula el tamaño de las casillas y les da forma, y también asigna valores a los botones que se mostrarán al final del juego.

lib/game/models/**square.dart**: cada una de las casillas que se muestren en el panel. Contiene información de su valor, su índice en el panel, el próximo índice que tendría tras un movimiento, si se va a fusionar tras dicho movimiento, y diversos métodos para calcular los índices actual y próximo.

lib/game/models/**score.dart**: el panel de puntuaciones. Contiene 2 clases, pero una de ellas solo se utiliza localmente en la otra, que es la que luego servirá de widget en **game.dart**.

lib/game/models/**button.dart**: los botones, tanto de texto como de iconos, personalizados para que encajen con el aspecto de la aplicación.

lib/game/models/**animated_square.dart**: cada una de las casillas, pero su aplicación con animaciones. Tan solo contiene métodos para activar las animaciones de estirar y rebotar en los casos de movimiento y fusión, respectivamente.

2.c. Estructura de clases: Controladores

lib/game/**controllers**: es la carpeta que contiene aquellas clases que nos servirán para manejar el flujo del juego.

lib/game/controllers/**board.dart**: contiene una sola clase con múltiples métodos para iniciar una partida, mover las casillas en la dirección deseada, fusionar casillas contingentes y detectar si el movimiento realizado implica el fin de la partida o no.

lib/game/controllers/**round.dart**: contiene 2 clases simples que marcan el inicio y final de una ronda y se aseguran de que la dirección solicitada es la efectuada. De esta manera se previenen errores de animación o de flujo del juego en caso de que el jugador vaya demasiado rápido.

lib/game/controllers/**adapter.dart**: contiene una sola clase que permite leer y escribir información almacenada en **JSON** para lograr la persistencia.

3. Lógica del programa

Todas las clases, tanto de la raíz como los modelos y controladores, están comentados para guiar en el proceso. La lógica general del programa sería:

1. **Iniciar el juego**: se crea una casilla de valor 2 con índice aleatorio dentro del panel de 16.

2. **Mover la casilla**: se puede realizar con un desliz de dedos en el móvil, o con el ratón o las flechas del teclado en el ordenador. Antes de esto, se guardará el estado actual del tablero para poder volver a él con el botón de Deshacer.

3. **Comprobar el resultado del movimiento**: desde el controlador del panel, cada casilla calcula su próximo índice, es decir, la posición que tendrá tras este movimiento.

4. **Comprobar si se fusionan**: una vez estén todas las casillas colocadas, sabiendo la dirección del movimiento, comprobarán si se fusionarán o no, en cuyo caso se convertirían en una sola casilla que sumaría sus valores y ocuparía una sola posición.

5. **Finaliza la ronda** con la creación de una nueva casilla de valor 2 y se actualiza el estado de la aplicación para que el tablero muestre la nueva posición de las casillas.

6. Tras esto, podemos realizar otro movimiento con un **nuevo desliz**, volver al estado anterior con el botón de **Deshacer**, reiniciar el juego con el botón de **Reiniciar**, o volver al menú principal con el botón de **Salir** (se guardará el estado de la partida actual).

4. Aplicación de persistencia

Para realizar la persistencia de datos en Flutter, lo más recomendado es utilizar **JSON**. Para ello, importaremos **Hive Flutter**, un gestor de datos para Dart que permite guardar el estado de la aplicación, o más bien, de sus modelos, en formato JSON y cargarlos de nuevo al reabrirla.

Tan solo necesitamos "mapear" los modelos del panel (**board.dart**) y las casillas (**square.dart**), con lo que se crearán clases autogeneradas que utilizará Hive para la persistencia. Estas clases serán **board.g.dart** y **square.g.dart**, siendo **g** el indicador de que son generadas.

Tras esto, añadimos la funcionalidad necesaria a la clase adapter.dart para leer y escribir en formato JSON, y configuramos los métodos de carga **load()** y guardado **save()** en el controlador **board.dart** y sus correspondientes llamadas en **game.dart** (en cada actualización del estado y al salir al menú principal).

5. Posibles mejoras

Una mejora que me habría gustado implementar es añadir **paneles más grandes para una mayor dificultad al juego** y así permitir puntuaciones más altas, pero por problemas de complejidad no he podido llevarlo a cabo a tiempo. También sería interesante **añadir sonidos a las acciones** de botones, deslices y fusiones.

Como implementación de última hora, he añadido un **icono personalizado** utilizando la dependencia **Flutter Launch Icons**, que refactoriza las líneas de código necesarias de manera automática tras indicarle en **pubspec.yaml** la imagen deseada.