18. La lógica del juego

Ya estamos listos para escribir un proyecto (`CCC18`) que nos permitirá jugar una partida completa al solitario y nos avisará con un mensaje cuando ya no podamos hacer mas movimientos.

El proyecto va a contar con dos clases:

- `Game.java`: con los métodos que implementan las reglas del juego como, por ejemplo, el método que comprueba si un salto es posible.
- `MainActivity.java`: que gestiona las pulsaciones del usuario y se las comunica al objeto de la clase `Game`.

Esta separación entre las reglas del juego y la interfaz gráfica permite mantener estos dos módulos de forma independiente. Más adelante podríamos por ejemplo modificar la interfaz gráfica sin necesidad de tocar la clase `Game`. Este es el enfoque utilizado en el patrón de diseño conocido como **modelo-vista-controlador**:

- El modelo es todo lo que tiene que ver con los datos, es decir, las reglas del juego en nuestro caso.
- La vista es la interfaz gráfica, es decir, el tablero del juego. En nuestra app la interfaz se especifica en el fichero de diseño `activity_main.xml`.
- El controlador es la actividad que sirve de intermediario entre la interfaz y el modelo: captura los eventos que tienen lugar en la vista, se los comunica al modelo para que se actualice y luego a la vista para que se redibuje.

El fichero de diseño utilizado en el proyecto `CCC18` es el mismo que el utilizado en la unidad anterior. Empecemos estudiando la actividad `MainActivity` que, como puedes comprobar a continuación, contiene muchos elementos conocidos.

18.1 La clase `MainActivity`

/src/MainActivity.java

```
package es.uam.eps.android.ccc18;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.RadioButton;
import android.widget.Toast;

public class MainActivity extends Activity implements OnClickListener{
    Game game;
    static final int SIZE = 7;

    private final int ids [][] = {
       {0, 0, R.id.f1, R.id.f2, R.id.f3, 0, 0},
       {0, 0, R.id.f4, R.id.f5, R.id.f6, 0, 0},
       {R.id.f7, R.id.f8, R.id.f9, R.id.f10, R.id.f11, R.id.f12, R.id.f13},
       {R.id.f14, R.id.f15, R.id.f16, R.id.f17, R.id.f18, R.id.f19, R.id.f20},
       {R.id.f21, R.id.f22, R.id.f23, R.id.f24, R.id.f25, R.id.f26, R.id.f27},
       {0, 0, R.id.f28, R.id.f29, R.id.f30, 0, 0},
       {0, 0, R.id.f31, R.id.f32, R.id.f33, 0, 0}};
```

```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        registerListeners();
        game = new Game();
        setFigureFromGrid();
    }

    private void registerListeners (){
        RadioButton button;

        for (int i=0; i<SIZE; i++)
            for (int j=0; j<SIZE; j++)
                if (ids[i][j]!=0){
                    button = (RadioButton) findViewById(ids[i][j]);
                    button.setOnClickListener(this);
                }
    }

    public void onClick (View v){
        int id = ((RadioButton) v).getId();

        for (int i=0; i<SIZE; i++)
            for (int j=0; j<SIZE; j++)
                if (ids[i][j] == id) {
                    game.play(i, j);
                    break;
                }

        setFigureFromGrid();
        if (game.isGameFinished())
            Toast.makeText(this, R.string.gameOverTitle, Toast.LENGTH_LONG).show();
    }

    private void setFigureFromGrid (){
        RadioButton button;

        for (int i=0; i<SIZE; i++)
            for (int j=0; j<SIZE; j++)
                if (ids[i][j] != 0){
                    int value = game.getGrid(i, j);
                    button = (RadioButton) findViewById(ids[i][j]);

                    if (value == 1)
                        button.setChecked(true);
                    else
                        button.setChecked(false);
                }
    }
}
```

El método `onCreate()`, una vez inflada la interfa gráfica especificada en el fichero de diseño, llama a `registerListeners()`, que se encarga de registrar la actividad como escuchador para todos los botones, incluido el central. También instancia un objeto de tipo `Game`, que se utilizará dentro del método `onClick()`. Finalmente, sitúa las fichas sobre el tablero con el método `setFigureFromGrid()` (poniendo a `true` el estado de los botones necesarios). Este método utiliza la información del miembro `grid` de la clase `Game`, que es un array 7x7 de enteros: 1 indica que la posición correspondiente del tablero está ocupada por una ficha y 0 refleja que la posición está vacía.

El método `onClick()` identifica las coordenadas del botón pulsado por el jugador y se las pasa al método `play()` de la clase `Game`. Este método se encarga de actualizar el array `grid` de `Game`, de tal forma que la actividad pueda redibujar el tablero con el método `setFigureFromGrid()`. Finalmente, si el método `isGameFinished()` devuelve `true`, se muestra un mensaje que indica el final del juego mediante la clase `Toast`.

Un toast es una ventana flotante que presenta rápidamente un pequeño mensaje. El método `makeText()` construye el objeto `Toast` y toma tres argumentos:

1. El contexto.
2. Una referencia a un objeto `CharSequence` que contiene el mensaje que se desea mostrar.
3. La duración de la vista: `Toast.LENGTH_SHORT` o `Toast.LENGTH.LONG`.

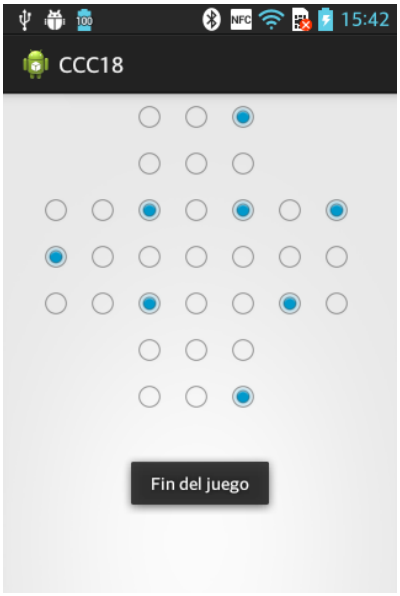
El método `makeText()` devuelve el objeto `Toast`. Es entonces cuando llamamos al método `show()` de `Toast` para mostrar el mensaje. El recurso `gameOverTitle` almacena el siguiente mensaje: Fin del juego.

/res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">CCC18</string>
    <string name="gameOverTitle">Fin del juego</string>

</resources>
```



Mas adelante aprenderemos cómo mostrar mensajes distintos dependiendo del idioma del dispositivo.

18.2 La clase `Game`

Veamos ahora cómo implementa `Game` las reglas del juego:

/src/Game.java

```
package es.uam.eps.android.ccc18;

public class Game {
    static final int SIZE = 7;
    private int grid[][];
    private static final int CROSS[][]={{0,0,1,1,1,0,0},
                                        {0,0,1,1,1,0,0},
```

```
                                        {1,1,1,1,1,1,1},
                                        {1,1,1,0,1,1,1},
                                        {1,1,1,1,1,1,1},
                                        {0,0,1,1,1,0,0},
                                        {0,0,1,1,1,0,0}};
    private static final int BOARD[][]={{0,0,1,1,1,0,0},
                                        {0,0,1,1,1,0,0},
                                        {1,1,1,1,1,1,1},
                                        {1,1,1,1,1,1,1},
                                        {1,1,1,1,1,1,1},
                                        {0,0,1,1,1,0,0},
                                        {0,0,1,1,1,0,0}};
    private int pickedI, pickedJ;
    private int jumpedI, jumpedJ;
    private enum State {READY_TO_PICK, READY_TO_DROP, FINISHED};
    private State gameState;

    public Game(){
        grid = new int [SIZE][SIZE];

        for (int i=0; i<SIZE; i++)
            for (int j=0; j<SIZE; j++)
               grid[i][j] = CROSS[i][j];

        gameState = State.READY_TO_PICK;
    }

    public int getGrid(int i, int j){
        return grid[i][j];
    }

    public boolean isAvailable(int i1, int j1, int i2, int j2) {

        if (grid[i1][j1]==0 || grid[i2][j2] == 1)
            return false;

        if (Math.abs(i2-i1) == 2 && j1 == j2)
        {
            jumpedI = i2 > i1 ? i1+1: i2+1;
            jumpedJ = j1;
            if (grid[jumpedI][jumpedJ] == 1)
               return true;
        }

        if (Math.abs(j2-j1) == 2 && i1 == i2)
        {
            jumpedI = i1;
            jumpedJ = j2 > j1 ? j1+1: j2+1;
            if (grid[jumpedI][jumpedJ] == 1)
                return true;
        }

        return false;
    }

    public void play (int i, int j) {
        if (gameState == State.READY_TO_PICK) {
            pickedI = i;
            pickedJ = j;
            gameState = State.READY_TO_DROP;
        } else if (gameState == State.READY_TO_DROP) {
            if (isAvailable(pickedI, pickedJ, i, j)) {
                gameState = State.READY_TO_PICK;

                grid[pickedI][pickedJ] = 0;
                grid[jumpedI][jumpedJ] = 0;
                grid[i][j] = 1;

                if (isGameFinished())
                   gameState = State.FINISHED;
            }
            else {
                pickedI=i;
                pickedJ=j;
            }
        }
    }
```

```
    public boolean isGameFinished (){

        for (int i=0; i<SIZE; i++)
            for (int j=0; j<SIZE; j++)
                for (int p=0; p<SIZE; p++)
                    for (int q=0; q<SIZE; q++)
                        if (grid[i][j]==1 && grid[p][q]==0 && BOARD[p][q]==1)
                            if (isAvailable(i, j, p, q))
                                return false;

        return true;
    }
}
```

Para empezar hay que tener en cuenta que el juego puede estar en uno de los tres estados siguientes:

- `READY_TO_PICK`: el juego está en este estado antes de pulsar una ficha que se desea mover.
- `READY_TO_DROP`: el juego está en este estado cuando espera que indiquemos dónde deseamos dejar caer una ficha previamente seleccionada.
- `FINISHED`: el juego está en este estado si no podemos mover ninguna ficha.

Veamos algunos de los miembros de `Game`. Tenemos tres arrays bidimensionales de enteros de dimension 7x7:

- El miembro `grid` es el único que se modifica a lo largo del juego ya que indica en cada momento si hay una ficha (1) o no (0) en cada posición.
- El miembro `CROSS` almacena la figura inicial, es decir, las posiciones del tablero que tienen ficha inicialmente (1).
- El miembro `BOARD` tiene a 1 las posiciones accesibles del tablero. Se diferencia de `CROSS` en la posición central, que es accesible pero no tiene ficha inicialmente.

Además, `(pickedI, pickedJ)` son las coordenadas de la última ficha que quiere mover el jugador y `(jumpedI, jumpedJ)` son las coordenadas de la última ficha que hemos intentado saltar.

El constructor de `Game` inicializa el estado del juego con el valor `READY_TO_PICK` y el array `grid` con el array `CROSS`. Mas adelante tendremos ocasión de inicializar el tablero con otras figuras iniciales a elección del jugador.

El método `isAvailable()` devuelve `true` si la ficha con coordenadas `(i1,j1)` puede saltar a la posición `(i2,j2)`. Para ello el método ejecuta las siguientes tareas:

- Comprueba que la posición de origen tiene ficha y que la posición de destino está vacía.
- Comprueba que las posiciones de origen y destino están a dos unidades de distancia en fila (segundo `if`) o en columna (primer `if`).
- Calcula la posición intermedia `(jumpedI, jumpedJ)` y comprueba que tiene ficha.

El método `play()` funciona de la siguiente manera:

- Si el estado del juego es `READY_TO_PICK`, guarda las coordenadas de la ficha seleccionada en `pickedI` y `pickedJ` para su uso mas adelante, en otra llamada a `play()`.
- Si el estado del juego es `READY_TO_DROP`, comprueba que el salto de la ficha seleccionada en una llamada anterior (`pickedI`, `pickedJ`) a la posición (`i`, `j`), la última pulsada, es posible. En ese caso, devuelve el estado del juego al valor `READY_TO_PICK` y actualiza el array `grid` para reflejar el salto, y que luego lo utilice la actividad para redibujar el tablero. Si el salto no es posible, actualiza (`pickedI`, `pickedJ`) con las coordenadas de la última posición, es decir, el destino seleccionado se convierte en origen para un movimiento posterior del jugador.

Por último el método `isGameFinished()` comprueba si existe algún salto válido con las fichas que quedan sobre el tablero. Si encuentra alguno, devuelve `false`. En caso contrario, devuelve `true`. Para ello recorre todos los posibles pares de posiciones inicial y final, incluso con repetición para simplificar el código. Antes de comprobar si el salto entre `(i,j)` y `(p,q)` es posible, comprueba que la casilla `(i,j)` está llena, que la casilla `(p,q)` está vacía y, además, es una casilla válida, lo cual está registrado en el array `BOARD`.