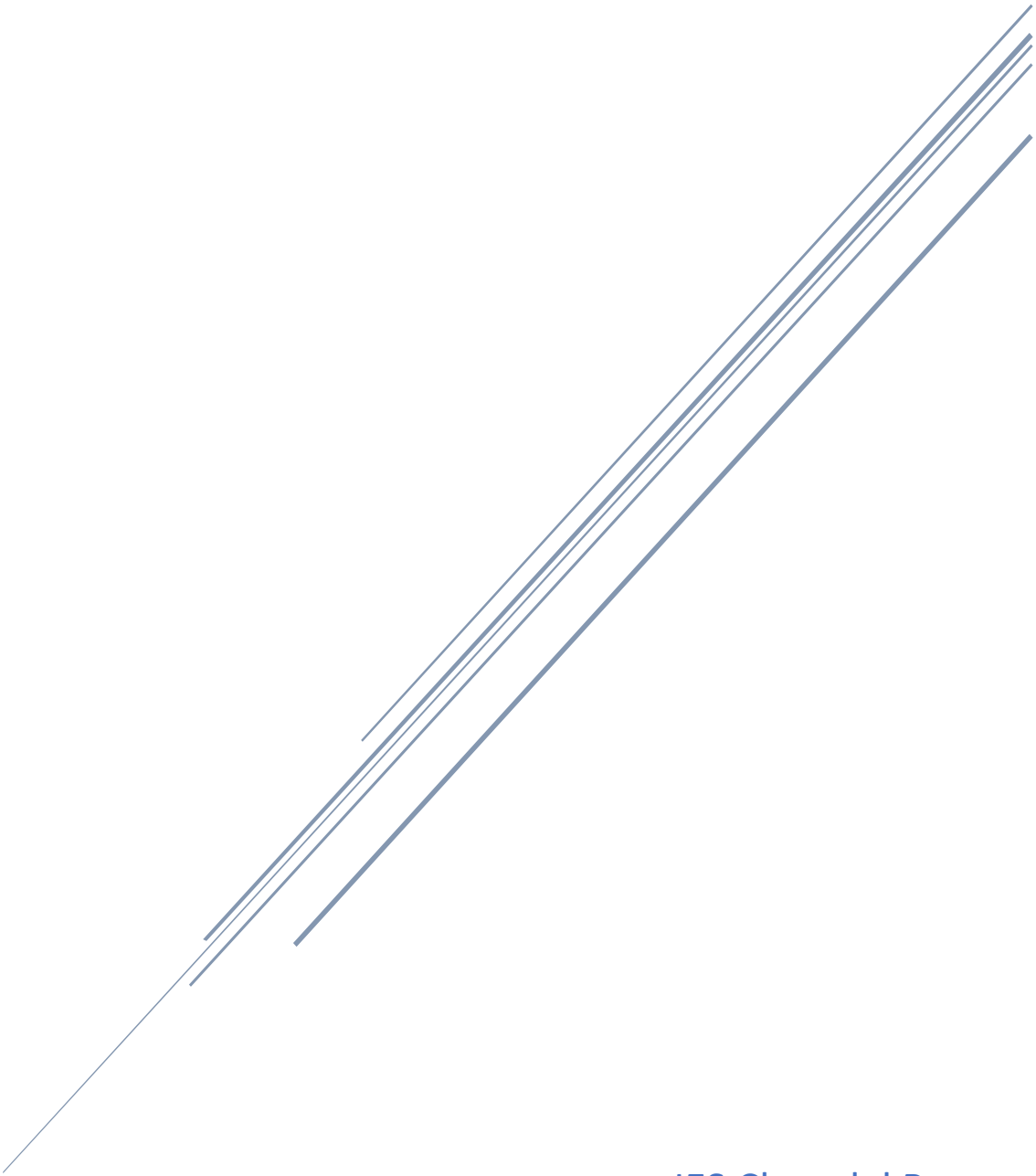
LABERINTO

Proyecto fin de curso



IES Clara del Rey
Adrián Caballero Orasio y Carlos del Valle Peláez

INDICE

1. INTRODUCCIÓN

1.1. DESCRIPCIÓN DEL PROYECTO

Para realizar nuestro proyecto de fin de curso hemos desarrollado un videojuego en Android, Laberinto.

Durante el desarrollo de este proyecto, hemos procurado simplificar al máximo los procesos y diseñar un juego lo mas optimo posible.

En cuanto a la parte de programación, el proyecto está desarrollado en Java y Android.

1.2. JUSTIFICACIÓN DEL PROYECTO

Hemos optado por este proyecto para ampliar y profundizar el uso de Java y Android, reforzando de este modo los conocimientos que hemos adquirido en las asignaturas de Programación y Android y ampliando dichos conocimientos mediante información añadida a través de internet.

1.3. REQUISITOS PARA EL USO DE LA APP LABERINTO

Tener instalado en el ordenador eclipse (para poder comprobar si se desea el código base en Java) y Android Studio (para poder visualizar la app final y poder interaccionar con el juego).

Otra opción es descargar en un móvil Android dicho juego para poder visualizarlo.

2. METODOLOGÍA

2.1. ESTRUCTURA DEL PROYECTO

La primera parte del proyecto será crear el código base en Java el cual, una vez realizado, mostrará un laberinto en el que el movimiento del jugador se efectuará indicando el comando por teclado.

Una vez completado, realizaremos el proyecto en Android añadiendo y optimizando el código el cual tendrá algunas mejoras tales como un menú inicial para introducir el nombre, laberinto que se desea ejecutar y se cambiarán los controles por una cruceta.

La segunda versión será seguir desarrollando y optimizando el código en Android. Podremos seleccionar el nivel y como queremos visualizar el laberinto, en formato texto o en formato imágenes (añadiendo obstáculos y el emoticono del jugador, el cual se cambiará más adelante), tendremos unos botones de ayuda e instrucciones.

En la versión final nuestro objetivo es depurar y optimizar pequeños fallos o mejoras que deseemos implantar en el juego.

3. CREACIÓN DEL CÓDIGO BASE EN JAVA

3.1. POINT

Con esta clase inicializamos el punto, reservando memoria y devolviendo el punto inicializado en caso de ser correcto y devolviendo NULL en caso de no serlo.

El primer paso será declarar nuestras variables y los constructores, uno con los parámetros x, y, simbol y otro vacío.

```
3  public class Point
4  {
5      private int x;
6      private int y;
7      private char symbol;
8      private Point parent;
9
10
11     public Point(int x, int y, char symbol)
12     {
13         this.x = x;
14         this.y = y;
15         this.symbol = symbol;
16         this.parent = null;
17     }
18
19     public Point()
20     {
21         this.x = -1;
22         this.y = -1;
23         this.symbol = FileChars.ERRORCHAR.c;
24         this.parent = null;
25     }
```

3.1. Imagen variables y constructor clase Point

A continuación, crearemos los **getters** y **setters**, en donde **getX** y **getY** devolverán la coordenada X o Y de un punto dado o -1 si se produce algún error y **setX** y **setY** modificarán la coordenada X o Y de un punto dado, devolviendo error si se produce algún fallo.

Dentro de esta clase encontraremos también una serie de funciones derivadas que indicarán el tipo de punto que tenemos, estas funciones serán **isInput()**, **isOutput()**, **isBarrier()**, **isSpace()** e **isActual().**

La función **copy()** se encargará de, si todo ha ido bien, copiar los datos de un punto a otro devolviendo el punto copiado(incluyendo la reserva necesaria de memoria), en caso contrario devolverá NULL.

3.2. MAZE

La clase **Maze.java** sirve para crear el laberinto. Dentro de ella encontraremos su constructor, el cual inicializa un mapa reservando memoria y devolviendo el mapa inicializado en el caso de que se haya hecho correctamente, siendo NULL en caso contrario.

Dentro de esta clase encontraremos las funciones **getInput()** y **getOutput()**, las cuales se encargarán de devolver el punto de entrada o salida en un mapa dado.

Tendremos también **getNeighbor()**, que devolverá el punto resultante al realizar un movimiento en un mapa a partir de un punto inicial.

La función **setPoint()** añade un punto a un mapa dado reservando nueva memoria o modifica el punto si ya se encuentra. Debe comprobar de que tipo es el punto, Output o Input, para guardarlo como corresponda.

```java
public boolean setPoint(Point p)
{
    if (p == null)
    {
        System.err.println("error en setPoint");
        return false;
    }

    int x = p.getX(), y = p.getY();
    if (x < 0 || x > getNx() || y < 0 || y > getNy())
    {
        return false;
    }

    if (p.getSymbol() == FileChars.INPUT.c && getInput() != null/* && p == getInput() */)
    {
        this.getInput().setSymbol(FileChars.SPACE.c);
    }

    if (p.getSymbol() == FileChars.OUTPUT.c && getOutput() != null/* && p == getOutput() */)
    {
        this.getOutput().setSymbol(FileChars.SPACE.c);
    }

    this.coord[x][y] = p;
    return (getPoint(x, y).equals(p));
}
```

3.2. Imagen función setPoint clase Maze

Las funciones mas importantes dentro de esta clase son **deepSearchStack()**, **breadthSearchQueue()** y **deepSearchRec()**, todas ellas realizan el recorrido completo de un mapa en busca del Output desde el Input, usando una pila.

```
public int deepSearchStack(Point input, Movements[] strat)
{
    Stack<Point> stack = new Stack<Point>();
    Point cp = null, neighbor = null;
    stack.push(input);

    while (stack.isEmpty() == false)
    {
        cp = (Point) stack.pop();
        if (cp.getSymbol() != Movements.VISITED.c)
        {
            if (this.getPoint(cp.getX(), cp.getY()).setSymbol(Movements.VISITED.c) == false)
            {
                return -1;
            }

            for (int i = strat.length - 1; i >= 0; i--)
            {
                /* stack is LIFO-> movements with higher preference have to be checked last */
                neighbor = this.getNeighbor(cp, strat[i]);

                if (neighbor != null)
                {
                    if (neighbor.isOutput() == true)
                    {
                        neighbor.setParent(this.getPoint(cp.getX(), cp.getY()));
                        return this.pathPaint(neighbor);
                    }

                    if (neighbor.isSpace() == true)
                    {
                        neighbor.setParent(this.getPoint(cp.getX(), cp.getY()));
                        if (stack.push(neighbor) == null)
                        {
                            System.err.println("Error in stack_push.");
                            return -1;
                        }
                    }
                    neighbor = null;
                }

             }
        }
        cp = null;
    }
    return -1;
}
```

3.2. Imagen función deepSearchStack clase Maze

```
public int breadthSearchQueue(Point input, Movements strat[])
{
    Point cp = null, neighbor = null;
    Queue<Point> queue = new LinkedList<Point>();
    queue.add(input);
    while (queue.isEmpty() == false)
    {
        cp = (Point) queue.remove();
        if (cp.getSymbol() != Movements.VISITED.c)
        {
            if (cp.isInput() != true)
            {
                if (this.getPoint(cp.getX(), cp.getY()).setSymbol(Movements.VISITED.c) == false)
                {
                    return -1;
                }
            }

            for (int m = 0; m < strat.length; m++)
            {
                /* queue is FIFO-> movements with higher preference have to be checked first */
                neighbor = this.getNeighbor(cp, strat[m]);
                if (neighbor != null)
                {
                    if (neighbor.isOutput() == true)
                    {
                        neighbor.setParent(this.getPoint(cp.getX(), cp.getY()));
                        return this.pathPaint(neighbor);
                    }

                    if (neighbor.isSpace() == true)
                    {
                        neighbor.setParent(this.getPoint(cp.getX(), cp.getY()));
                        if (queue.add(neighbor) == false)
                        {
                            System.err.println("Error in queue_insert.\n");
                            return -1;
                        }

                    }
                    neighbor = null;
                }
            }
        }
        cp = null;
    }
    return -1;
}
```

3.2. Imagen función breadthSearchQueue clase Maze

```
Point deepSearchRec(Point input, Movements strat[], boolean compare)
{
    Point neighbor = null, auxpoint = null;
    if (compare == true)
    {
        System.out.println("Current maze:");
        System.out.println(this.toString());
    }

    if (input.isOutput())
    {
        return input;
    }
    input.setSymbol(Movements.VISITED.c);

    for (int i = 0; i < strat.length; i++)
    {
        neighbor = this.getNeighbor(input, strat[i]);
        if (neighbor != null)
        {
            if (neighbor.getSymbol() != Movements.VISITED.c && neighbor.getSymbol() != Movements.ACTUAL.c
                    && neighbor.getSymbol() != FileChars.BARRIER.c)
            {
                neighbor.setParent(this.getPoint(input.getX(), input.getY()));

                if (neighbor.isOutput() == false)
                {
                    neighbor.setSymbol(Movements.ACTUAL.c);
                }
                auxpoint = this.deepSearchRec(neighbor, strat, compare);
                if (auxpoint != null)
                {
                    if (auxpoint.getParent().equals(input))
                    {
                        /* if(compare==true){System.out.println("Solution!");paintPath();} */}
                    return auxpoint;
                }
            }
        }
    }
    return null;
}
```

3.2. Imagen función Point deepSearchRec clase Maze

3.3. MAZESOLVER

La clase **Mazesolver.java** se encargará de comprobar que se puede llegar a la salida.

Podemos encontrar dentro de ella el método **runSolver(),** el cual llama a **stackSolver**, **queueSolver** y **recursiveSolver** con una matriz de movimientos.

```
public static void runSolver(String mfile, Movements strat[][]) throws Exception
{
    for (int i = 0; i < strat.length; i++)
    {
        if (strat[i].length != 4)
        {
            System.err.println("Numero de movimientos incorrecto.");
            return;
        }
        System.out.printf("ESTRATEGIA %s, %s, %s, %s\n", strat[i][0], strat[i][1], strat[i][2], strat[i][3]);
        struct("pila", mfile, strat[i], false, 2);
        struct("cola", mfile, strat[i], false, 2);
        struct("algoritmo recursivo", mfile, strat[i], false, 3);
    }
}
```

3.3. Imagen método runSolver clase MazeSolver

La función **stackSolver()** se encarga de resolver un mapa usando una pila.

```
public static int stackSolver(String mfile, Movements strat[]) throws Exception
{
    File file = new File(mfile);// file = fopen(mfile,"r");
    Maze maze = new Maze();

    if (file == null || maze == null || strat.length != 4)
    {
        return -1;
    }

    if (maze.read(file) == false)
    {
        System.err.println("Error en recursiveSolver.");
        return -1;
    }

    System.out.println("Maze:\n" + maze.toString());
    System.out.println("Input:\n" + maze.getInput().print());
    int pathlength = 0;
    pathlength = maze.deepSearchStack(maze.getInput(), strat);
    System.out.println(((pathlength != -1) ? "Output found, length of path: " + pathlength : "No path found."));
    System.out.println(maze.toString());
    System.out.printf("Final maze: %s%n", maze.pathPaint(maze.getOutput()));
    return pathlength;
}
```

3.3. Imagen función stackSolver() clase MazeSolver

La función **queueSolver()** se encarga de resolver un mapa usando una cola.

```
public static int queueSolver(String mfile, Movements strat[]) throws Exception
{
    File file = new File(mfile);
    Maze maze = new Maze();

    if (file == null || maze == null || strat.length != 4)
    {
        return -1;
    }

    if (maze.read(file) == false)
    {
        System.err.println("Error en recursiveSolver.");
        return -1;
    }

    System.out.println("Maze:\n" + maze.toString());
    System.out.println("Input:\n" + maze.getInput().print());
    int pathlength = 0;
    pathlength = maze.breadthSearchQueue(maze.getInput(), strat);
    System.out.println(((pathlength != -1) ? "Output found, length of path: " + pathlength : "No path found."));
    System.out.println(maze.toString());
    System.out.printf("Final maze: %s%n", maze.pathPaint(maze.getOutput()));
    return pathlength;
}
```

3.3. Imagen función queueSolver clase MazeSolver

La función **recursiveSolver()** se encarga de resolver un mapa usando un algoritmo recursivo.

```
public static int recursiveSolver(String mfile, Movements[] strat, boolean compare) throws Exception
{
    File file = new File(mfile);
    Maze maze = new Maze();

    if (file == null || maze == null || strat.length != 4)
    {
        return -1;
    }

    if (maze.read(file) == false)
    {
        System.err.println("Error en recursiveSolver.");
        return -1;
    }

    System.out.println("Maze:\n" + maze.toString() + "\n" + "Input:\n" + maze.getInput().print());
    int pathlength = 0;
    Point out = maze.deepSearchRec(maze.getInput(), strat, compare);

    if (out != null)
    {
        pathlength = maze.pathPaint(out);
        System.out.printf("Output is in point: %s%n", out.toString());
    }
    else
    {
        pathlength = -1;
    }

    System.out.println(((pathlength != -1) ? "Output found, length of path: " + pathlength : "No path found."));
    System.out.println(maze.toString());
    System.out.printf("Final maze: %s%n", maze.pathPaint(maze.getOutput()));
    return pathlength;
}
```

3.3. Imagen función recursiveSolver clase MazeSolver

3.4. FILECHARS

En la clase **FileChars.java** nos encontramos cada uno de los puntos disponibles en los txt , pared, suelo, entrada y salida.

```
public /* static */ enum FileChars
{
    ERRORCHAR('E'),
    INPUT('i'),
    OUTPUT('o'),
    BARRIER('+'),
    SPACE(' '),
    ;

    public final char c;

    private FileChars(char label)
    {
        this.c = label;
    }

    public static FileChars valueOf(char c)
    {
        for (FileChars m : values())
        {
            if (m.c == c)
            {
                return m;
            }
        }
        return null;
    }
}
// typedef enum {RIGHT = 0, UP = 1, LEFT = 2, DOWN = 3, STAY = 4} Move;
```

3.4. Imagen clase FileChars

3.5. MOVEMENTS

La clase **Movements.java** almacena lo movimientos y caracteres del personaje.

```
public enum Movements
{
    RIGHT('>'), //
    UP('^'), //
    LEFT('<'), //
    DOWN('v'), //
    /**
     * Simbolo para un espacio ya visitado<br>
     */
    VISITED('x'), //
    /**
     * Simbolo para un espacio actualmente ocupado<br>
     */
    ACTUAL('P'), //
    ;

    public final char c;

    private Movements(char label)
    {
        this.c = label;
    }

    public static Movements valueOf(char c)
    {
        for (Movements m : values())
        {
            if (m.c == c)
            {
                return m;
            }
        }
        return null;
    }
}
```

3.5. Imagen clase Movements

3.6. UTILS

La clase **Utils.java** contiene el limite que creamos para la lectura de los txt, consiguiendo de este modo que tengan un maximo a la hora de crearlo.

```
public class Utils
{
    /**
     * Tamaño maximo de un string en C.
     */
    static int MAX = 4096;

    public static String formatC(String txt, char c, int n)
    {
        return String.format(String.format("%" + (n - txt.length()) / 2 + "s%" + -(n - txt.length()) / 2 + "s", "%s", "").replace(' ', c),txt);
    }

    public static String formatL(String txt, char c, int n)
    {
        return String.format(String.format("%" + (n - txt.length()) + "s", "%s").replace(' ', c), txt);
    }

    public static String formatR(String txt, char c, int n)
    {
        return String.format(String.format("%" + -(n - txt.length()) + "s", "%s").replace(' ', c), txt);
    }
}
```

3.6. Imagen clase Utils

3.7. MAIN

La clase Main.java nos permitirá ejecutar nuestro programa. En ella encontraremos en primer lugar una llamada a la clase Movements.java, la cual nos permitirá ejecutar los movimientos de nuestro jugador.

```
private final static Movements strat[][] =
{ //
        { Movements.UP, Movements.DOWN, Movements.LEFT, Movements.RIGHT, }, //
        { Movements.RIGHT, Movements.UP, Movements.DOWN, Movements.LEFT, }, //
        { Movements.LEFT, Movements.RIGHT, Movements.UP, Movements.DOWN, }, //
        { Movements.DOWN, Movements.LEFT, Movements.RIGHT, Movements.UP, },//
};

private final static Movements strat1[] =
{Movements.LEFT, Movements.UP, Movements.RIGHT, Movements.DOWN, };
```

3.7. Imagen del método Movements clase Main

Dentro del main tendremos la ejecución de nuestro fichero .txt y llamamos a la función **play()**, la cual se encarga de leer el fichero y nos mostrará los controles posibles a introducir y, de ser correcto el control introducido, ejecutará el movimiento del personaje, esto lo realiza con **maze.startMaze()**, que pertenece a la clase **Maze.java**.

```
public static void main(String[] args) throws Exception
{
    if (args.length == 0)
    {
        main(new String[] { "m1.txt" });
        main(new String[] { "m2.txt" });
        main(new String[] { "m3.txt" });
        main(new String[] { "m4.txt" });
        return;
    }

    if (args.length < 1)
    {
        System.err.println("Falta el fichero de datos.");
        return;
    }

    /* ejsOld(args, 0); */
    play(args);
}
```

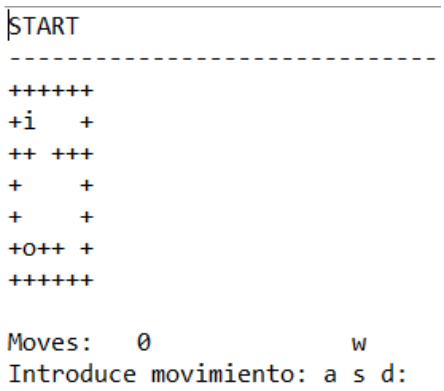
3.7. Imagen del main clase Main

```
public static void play(String[] args) throws Exception
{
    String mfile = args[0];
    File file = new File(mfile);// file = fopen(mfile,"r");
    Maze maze = new Maze();

    if (file == null || maze == null || strat.length != 4)
    {
        return;
    }

    if (maze.read(file) == false)
    {
        System.err.println("Error en play.read");
        return;
    }

    if (checkFile(args[0], strat1) != -1)
    {
        maze.startMaze();
    }
    else
    {
        System.err.println("SALIDA NO ENCONTRADA.");
        return;
    }
}
```

3.7. Imagen de la función play clase Main

4. MUESTRA PARCIAL DEL LABERINTO EN PANTALLA

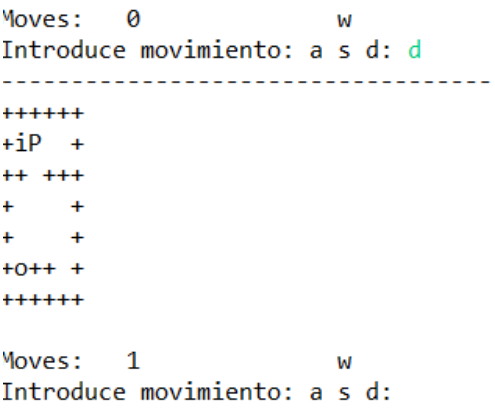
A continuación, veremos una visualización inicial de la ejecución de nuestro programa en Java.



4. Imagen terminal clase Main

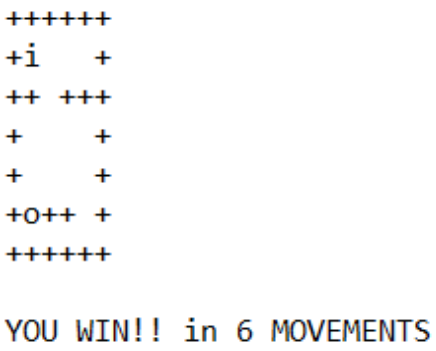
En la siguiente imagen podemos observar que, al introducir el movimiento **d**, nuestro personaje (**P**) se mueve a la derecha.

Una vez realizado el movimiento, cuenta dicho movimiento y vuelve a imprimir la instrucción para insertar un nuevo movimiento.



4. Imagen terminal clase Main

En la siguiente imagen podemos observar que, una vez que el personaje llega al punto o, finaliza el programa, mostrando un mensaje y el numero de movimientos realizados en total.



4. Imagen terminal clase Main

5. TRASPASO PROGRAMA A ANDROID STUDIO Y CREACIÓN DE CLASES JAVA

Una vez tenemos nuestro código en Java, exportamos todo el programa a Android Studio creando las siguientes clases para su correcto funcionamiento.

5.1. ACTUALIZACION MAIN ACTIVITY

En esta clase encontraremos un **método** llamado **onPulsar,** el cual tendrá la función de llevarnos a otra pantalla una vez que el jugador haya introducido su nombre y seleccionado un tipo de laberinto.

```
public void onPulsar(View v)
{
    String nombre = ((EditText) findViewById(R.id.etNombre)).getText().toString().trim();
    if (nombre.equals(""))
        muestraMensaje("Nombre inválido",Toast.LENGTH_SHORT);
    else if (laberinto.equals(""))
        muestraMensaje("Selecciona un laberinto",Toast.LENGTH_SHORT);
    else
    {
        Intent intent = new Intent( packageContext: this, SegundaActividad.class);
        intent.putExtra( name: "nombre", nombre);
        intent.putExtra( name: "laberinto", laberinto);
        startActivityForResult(intent, SECONDARY_ACTIVITY_TAG);
    }
}
```

5.1. Imagen método onPulsar clase Main Activity

Tendremos un segundo método que servirá para seleccionar un tipo de laberinto, el cual llamaremos **selectLaberinto,** cuya única función será la selección de dicho laberinto.

Encontraremos un método el cual servirá para mostrar un mensaje mediante Toast.

5.2. ACTUALIZACION CLASE MAZE

En esta clase añadiremos el **método printMaze**, el cual se encarga de dibujar el laberinto y de asegurarse que el tamaño de las celdas tenga valores positivos.

```
public String printMaze(Point p, int cellsX, int cellsY)
{
    int pX = (p != null) ? p.getX() : -1, pY = (p != null) ? p.getY() : -1;
    StringBuilder txt = new StringBuilder();
    int yCoord = (pY - 1 < cellsY / 2) ? 0 : (pY - 1 + cellsY <= getNrows()) ? pY - cellsY / 2 : getNrows() - cellsY;
    for (int y = 0, nY = 0, flag = 0; y < getNrows() && nY < cellsY; y++)
    {
        int xCoord = (pX - 1 < cellsX / 2) ? 0 : (pX - 1 + cellsX <= getNcols()) ? pX - cellsX / 2 : getNcols() - cellsX;
        for (int x = 0, nX = 0; x < getNcols() && nX < cellsX; x++)
        {
            if (x == xCoord && y == yCoord)
            {
                try
                {
                    if (x == pX && y == pY)
                        txt.append(Movements.ACTUAL.c);
                    else
                        txt.append(getPoint(x, y).getSymbol());
                }
                catch (Exception e)
                {
                    txt.append(FileChars.ERRORCHAR.c);
                }
                xCoord++;
                nX++;
                flag = 1;
            }
        }
        if (flag == 1)
        {
            if (y + 1 < getNrows() && nY < cellsY - 1)
                txt.append("\n");
            yCoord++;
            nY++;
        }
    }
    return txt.toString();
}
```

5.2. Imagen método printMaze de la clase Maze

5.3. SEGUNDA ACTIVIDAD

Esta clase será la encargada de mostrar el laberinto, contará con diferentes métodos entre los que podemos destacar:

- **Método onReset**, el cual será un botón situado junto a la cruceta el cual permitirá reiniciar el laberinto.

```java
public void onReset(View v)
{
    moves = 0;
    hasGanado = false;
    TextView tvC = findViewById(R.id.tvContador);
    tvC.setText(String.format("Movimientos: %1$d", moves));
    ((TextView) findViewById(R.id.tvGanardores)).setText("");
    play(getResourceId(laberintoType,  defType: "raw"));
}
```

5.3. Imagen método onReset de la clase Segunda Actividad

- **Método onBotonClick,** encargado del movimiento del jugador mediante la cruceta y de ir sumando cada movimiento que hace.

```java
public void onBotonClick(View v)
{
    if (!hasGanado)
    {
        Point neighbor;
        switch (v.getId())
        {
            case R.id.up:
            {
                neighbor = maze.getNeighbor(actual, Movements.UP);
                break;
            }
            case R.id.down:
            {
                neighbor = maze.getNeighbor(actual, Movements.DOWN);
                break;
            }
            case R.id.left:
            {
                neighbor = maze.getNeighbor(actual, Movements.LEFT);
                break;
            }
            case R.id.right:
            {
                neighbor = maze.getNeighbor(actual, Movements.RIGHT);
                break;
            }
            default:
            {
                throw new IllegalStateException("Unexpected value: " + v.getId());
            }
        }
```

```
        if (neighbor != null && !neighbor.isBarrier())
        {
            if (!neighbor.isOutput())
            {
                if (!actual.isInput() && !actual.isOutput())
                    actual.setSymbol(FileChars.SPACE.c);
                if (!neighbor.isInput())
                    neighbor.setSymbol(Movements.ACTUAL.c);
            }
            else
                actual.setSymbol(FileChars.SPACE.c);
            moves++;
            actual = neighbor;
        }
        printMaze();
        if (actual.equals(output))
        {
            TextView tv = findViewById(R.id.tvContador);
            tv.setText(String.format("Movimientos: %1$d", moves));
            muestraMensaje(String.format("Has ganado en %1$d movimientos", moves));
            onGanar();
        }
    } else onFinish(v);
}
```

5.3. Imagen método onBotonClick de la clase Segunda Actividad

- **Método actualizarBD,** para actualizar el ranking y almacenarlos en una base de datos.

```
protected void actualizarBD()
{
    tvLaberinto.setText("");
    String txt = "Rankings:\n";
    try
    {
        SQLiteHelper_Ranking db = SQLiteHelper_Ranking.getInstance(this);
        List<Ranking> ranking = db.getAllRankings();
        for (Ranking r : ranking)
            txt = String.format("%s%s\n", txt, r.toString( cont: this));
        ((TextView) findViewById(R.id.tvGanardores)).setText(txt);
    }
    catch (Exception e)
    {
        muestraMensaje("Error al mostrar ganadores");
    }
}
```

5.3. Imagen método actualizarBD de la clase Segunda Actividad

- **Método musica y onPause,** ambos se encargan del control de la musica y de apagarla encenderla.

```
public void musica(int music)
{
    try
    {
        Musica.musicaRaw( contexto: this, music);
        Musica.setLooping(true);
    }
    catch (IllegalArgumentException e)
    {
        Toast.makeText(getApplicationContext(),  text: "" + e, Toast.LENGTH_LONG).show();
        e.printStackTrace();
    }
}

@Override
public void onPause()
{
    if (Musica.getMp() != null && Musica.getMp().isPlaying())
    {
        Musica.onPause();
        Musica.retirar();
    }
    super.onPause();
}
```

5.3. Imagen método musica y onPause de la clase Segunda Actividad

5.4. RANKING

Esta clase se encarga exclusivamente de seleccionar los valores que queremos que se muestren a la hora de mostrar el resultado, los cuales se almacenan en la base de datos del programa.

```
public class Ranking
{
    public Integer id;
    public String laberinto;
    public String nombre;
    public Integer puntuacion;



    public Ranking(String nombre, String laberinto, Integer puntuacion)
    {
        this.nombre = nombre;
        this.laberinto = laberinto;
        this.puntuacion = puntuacion;
    }

    public Ranking(Integer id, String nombre, String laberinto, Integer puntuacion)
    {
        this.id = id;
        this.nombre = nombre;
        this.laberinto = laberinto;
        this.puntuacion = puntuacion;
    }

    public String toString(Context cont)
    {
        return String.format(cont.getResources().getString(R.string.rankingTxt), this.nombre, this.laberinto, this.puntuacion);
    }
}
```

5.4. Imagen clase Ranking

5.5. SQLITEHELPER_RANKING

Esta clase será nuestra base de datos, en la que tendremos diferentes métodos encargados de añadir, actualizar, y borrar los diferentes datos.

El **método addRanking** se encargará de añadir los datos introducidos por el jugador y su puntuación, así como el control de posibles errores a la hora de introducir dichos datos en la Base de Datos.

```java
public long addRanking(Ranking ranking)
{
    SQLiteDatabase db = getWritableDatabase();
    long id = -1;
    try
    {
        db.beginTransaction();
        ContentValues values = new ContentValues();
        values.put(key_id, ranking.id);
        values.put(key_nombre, ranking.nombre);
        values.put(key_laberinto, ranking.laberinto);
        values.put(key_puntuacion, ranking.puntuacion);
        id = db.insertOrThrow(TABLA,  nullColumnHack: null, values);
        db.setTransactionSuccessful();
    }
    catch (Exception e)
    {
        Log.d(TAG,  msg: "Error while trying to add ranking to database");
    }
    finally
    {
        db.endTransaction();
    }
    return id;
}
```

5.5. Imagen método addRanking de la clase SQLiteHelper Ranking

Una vez que dichos datos están introducidos en la Base de Datos, para evitar el duplicado de los mismos, tendremos el método addOrUpdateRanking que será el encargado de controlarlo.

```java
public long addOrUpdateRanking(Ranking ranking)
{
    Ranking r = getRanking(ranking);
    long id = -1;
    if (r == null)
    {
        id = addRanking(ranking);
    }
    else if (r.puntuacion > ranking.puntuacion)
    {
        id = updateRankingPuntuacion(ranking);
    }
    return id;
}
```

5.5. Imagen método addOrUpdateRanking de la clase SQLiteHelper Ranking

5.6. MÚSICA

Será la encargada de gestionar la música en nuestra aplicación, alguno de estos métodos es:

- **musicaRaw**.

  Encargado de iniciar la reproducción.

- **onPause**

  Para pausar la musica.

- **create**

  Método para crear un MediaPlayer para un Uri determinado.

- **seekto**

  Mueve el medio a la posición de tiempo especificada considerando el modo dado.

- **getTimestamp**

  Obtener la posición de reproducción actual.

- **setAudioStreamType**

  Establece el tipo de secuencia de audio para este MediaPlayer.

- **setLooping**

  Establece el reproductor para que esté en bucle o sin bucle.

6. CREACIÓN DE LAYOUTS

En esta parte nos encargaremos de crear los diferentes xml para mostrar gráficamente nuestro laberinto.

6.1. ACTIVITY_MAIN

Será el encargado de mostrar la ventana principal en la que aparece un **TexView** con el mensaje: ¿Cómo te llamas?, un **EditText** para introducir el nombre por teclado, un **Button** que contiene el mensaje Empezar Laberinto el cual solo se ejecutará si hemos seleccionado uno de los botones de selección de

laberinto, un **RadioGroup** que contiene varios **RadioButton** para elegir el tipo de laberinto

6.2. DPAD_VIEW

El siguiente xml se encarga de pintar la cruceta para poder realizar el movimiento y el botón R para reiniciar el nivel. Tendremos un **ImageView** que pintará una cruceta y, dentro de ella, tendremos 4 **Button** que serán los encargados de realizar el movimiento dependiendo de cual pulsemos. Tendremos también otro **Button** que pertenece al botón R, encargado de reiniciar el nivel.

6.3. MAZE_VIEW

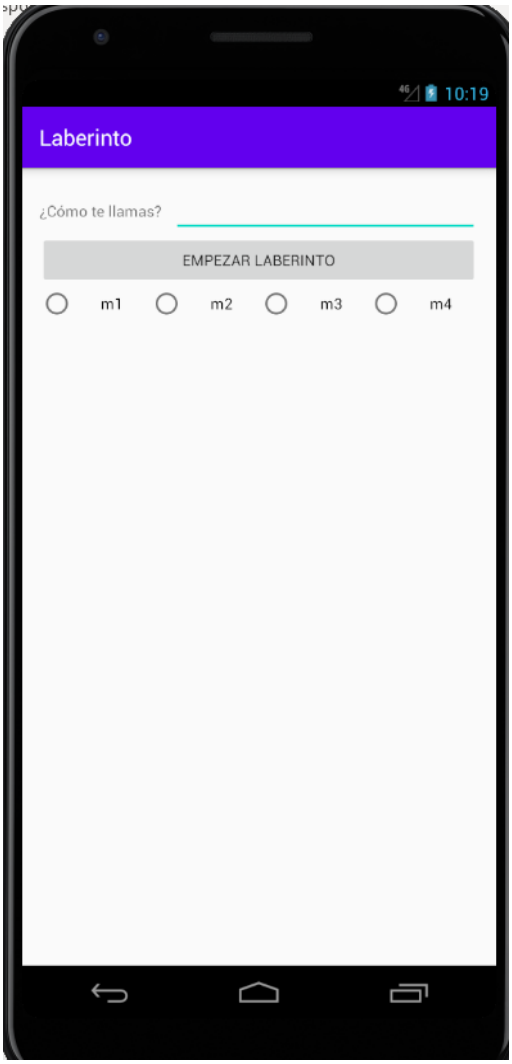
Este xml será el encargado de dibujar el laberinto, un **TextView** que mostrará el mensaje: Suerte y el nombre del jugador, otro **TextView** que mostrará el número de movimientos realizados y otro **TextView** que será el encargado de pintar el laberinto.

6.4. ACTIVITY_SEGUNDA_ACTIVIDAD

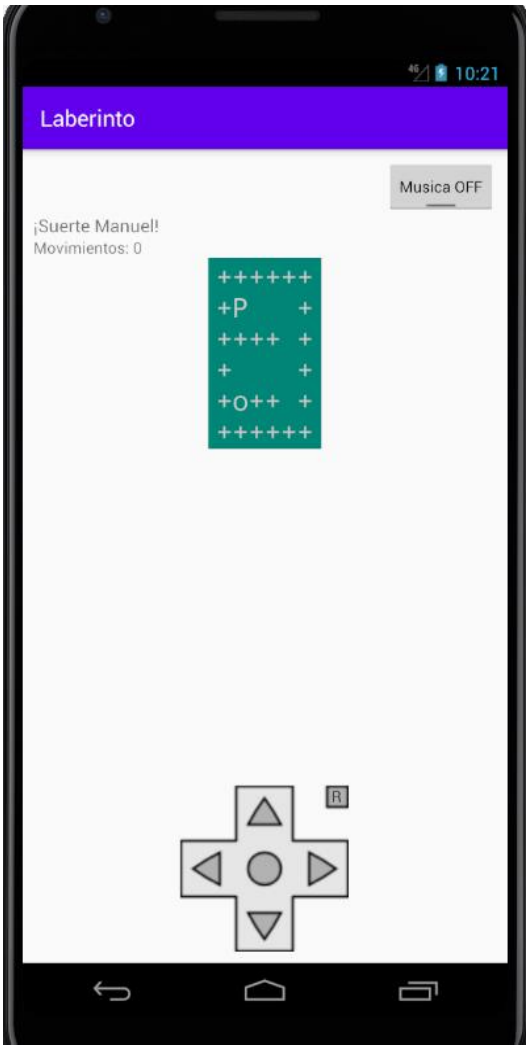
Encontraremos un **ToggleButton** el cual estará situado en la parte superior izquierda y será el encargado de permite silenciar la musica median ON/OFF. Tendremos dos **include** que se encargaran de llamar a los **layout** de **maze_view** y **dpad_view**.

7. PROTOTIPO LABERINTO FUNCIONANDO EN ANDROID

A continuación, veremos diferentes capturas del funcionamiento del laberinto en Android.



7. Imagen funcionamiento juego en Android



7. Imagen funcionamiento juego en Android

7. Imagen funcionamiento juego en Android

8. CLASES PARA SPRITE GRÁFICOS

8.1. MODIFICACIÓN CLASE MAZE A MAZEVIEW

Eliminaremos la clase Maze y crearemos nuestra nueva clase MazeView, la cual se encargará de la visualización del personaje, las casillas, su tamaño… Ya que la ejecución del juego, al contar con la posibilidad de ejecutarlo en formato texto o dibujo, tendrá dos clases más, encargadas de ejecutar dicha opción en base a la elección del jugador.

8.2. CLASE MAZE PLAYER DRAW ACTIVITY

Esta nueva clase se encargará de poder ejecutar el laberinto en forma dibujo llamando al **layout** a **activity_mazeplayer_draw** y llamando a su vez al **MazeView.**

Esta clase tendrá un contador y los movimientos de la cruceta, así como varios métodos entre los que destacan:

- **Método onOptionsItemSelected**, el cual nos permitirá mostrar un icono se play stop para la música, redimensionar nuestro laberinto, reiniciar la partida y mostrar una ayuda al jugador.

```java
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.musica: {
            item.setIcon(Utils.musicaPlayStop( c: this,  ini: false));
            return true;
        }
        case R.id.x_size: {
            item.setTitle("X: {mazeViewer.addCellsX()}");
            printMaze();
            return true;
        }
        case R.id.y_size: {
            item.setTitle("Y: {mazeViewer.addCellsY()}");
            printMaze();
            return true;
        }
        case R.id.restart: {
            Utils.muestraMensaje( c: this,  txt: "restart");
            iniPlay(this.laberintoType);
            return true;
        }
        case R.id.help: {
            //Utils.muestraMensaje(this,"help, por hacer");
            final View v = findViewById(R.id.action_settings);/*item.getActionView()*/
            openPopup(v, R.id.help);
            return true;
        }
        default: {
            return super.onOptionsItemSelected(item);
        }
    }
}
```

8.2. Imagen método onOptionsItemSelected

8.3. CLASE MAZE PLAYER TEXT ACTIVITY

Esta nueva clase se encargará de poder ejecutar el laberinto en formato texto, llamando al **layout** a **activity_mazeplayer_text.** Los métodos de esta clase serán muy similares al de la clase anterior, exceptuando cambios en alguno de ellos como, por ejemplo, en el **método printMaze(),** el cual se encargará de las dimensiones de cada laberinto y pintarlo.

9. LAYOUTS PARA SPRITE GRÁFICOS

9.1. ACTIVITY MAZEPLAYER DRAW

Este fichero nos permite visualizar el laberinto en formato dibujo, en el cual podemos observar un **maze_draw_view** (que se encarga de mostrar el laberinto), la cruceta, el contador de movimientos y un fondo de pantalla, quedando del siguiente modo:

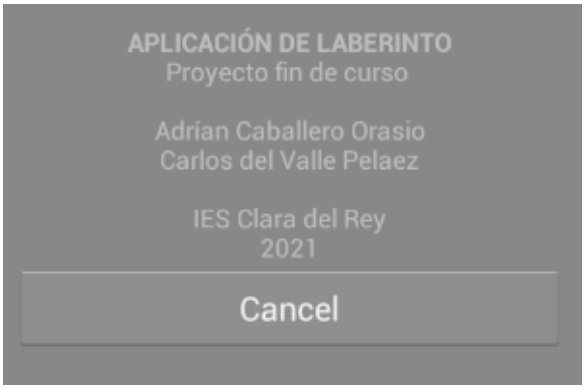
9.1. Imagen vista activity_mazeplayer_draw

9.2. ACTIVITY MAZEPLAYER TEXT

Este fichero nos permite visualizar el laberinto en formato texto, en el cual podemos observar un **maze_text_view** (que se encarga de mostrar el laberinto). El resto de código será igual que el de **activity_mazeplayer_draw.**

9.3. POPUP_WINDOW

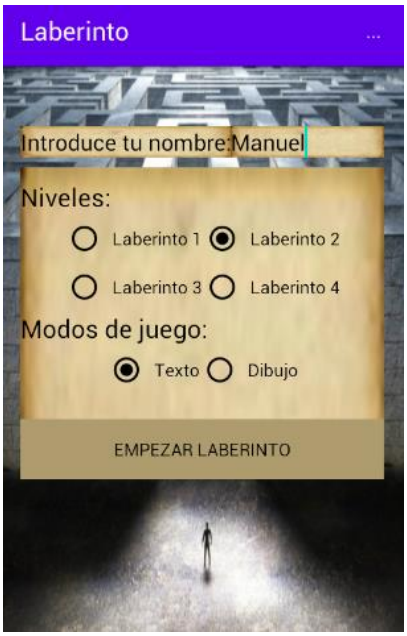
Este fichero sirve para mostrar una ventana emergente al pulsar sobre los tres botones de la pantalla de inicio (Rankings, Help, Information), quedando del siguiente modo:



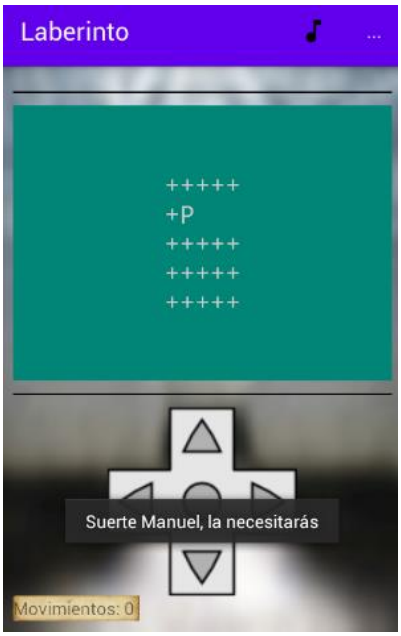
9.3. Imagen popup_window

10. VISUALIZACIÓN LABERINTO CON SPRITE GRÁFICOS Y MEJORAS

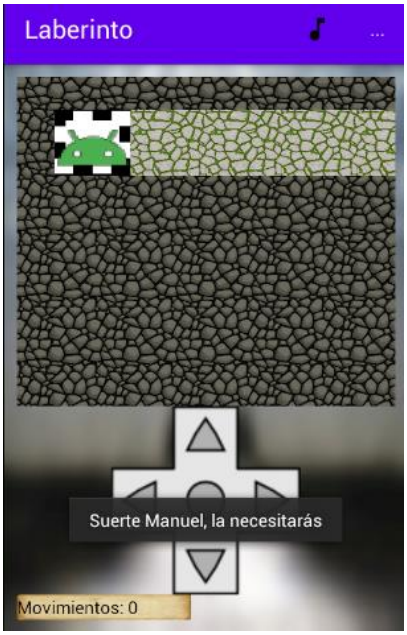
A continuación, veremos una visualización de como queda nuestro juego en ejecución:



10. Imagen pantalla inicial



10. Imagen juego en formato texto



10. Imagen juego en formato dibujo

11. CLASE AUTORESIZE TEXT VIEW

Esta clase es un **textView** que reduce el tamaño del texto, permitiendo de este modo que entre en la ventana y poder visualizar la totalidad del laberinto mediante un botón llamado **Zoom Out** que nos permite aumentar el angulo de visión del jugador, quedando del siguiente modo:

12. RETOQUES ESTÉTICOS FINALES

Para terminar, se realizan diferentes retoques para darle un mejor acabado a nuestra app, los cuales serán visualizados el día de la presentación, entre estos retoques podemos destacar:

- Introducción a la hora de carga del juego (Splash)

- Mejora de gráficos.

- Logotipo de la app.