Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Implementación de un recurso docente sobre Android para el autoaprendizaje de un lenguaje de programación

Autor: Alfonso Molina Montilla

Tutor: Antonio J. Sierra

Dep. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2015

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de la Telecomunicación

Implementación de un recurso docente sobre Android para el autoaprendizaje de un lenguaje de programación

Autor:
Alfonso Molina Montilla

Tutor:
Antonio J. Sierra
Profesor tutor

Dep. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2015

Autor: Alfonso Molina Montilla

Tutor: Antonio J. Sierra

El tribunal nombrado para juzgar el Trabajo arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2015

El Secretario del Tribunal

Resumen

En este proyecto se muestra la implementación de una herramienta docente para aprender algunos conceptos básicos de programación. Se ha desarrollado un videojuego para móviles Android, con la idea de enseñar un lenguaje de programación de una forma intuitiva y cómoda.

Esta herramienta está orientada a usuarios con conocimientos mínimos de programación. El jugador deberá diseñar programas con el objetivo de completar diferentes misiones. Estos programas utilizarán bucles y condicionales, aumentando progresivamente su complejidad. Al completar niveles, el jugador irá desbloqueando nuevas herramientas de programación y se explicarán sus usos.

Una vez escrito el código, este será ejecutado por pantalla, línea a línea, y se verá de forma gráfica qué es lo que hacen las líneas de texto.

El objetivo es que el usuario aprenda las bases de un lenguaje de programación y se acostumbre a diseñar programas de una forma estructurada. El usuario tampoco deberá preocuparse de la sintaxis, ya que el código se escribirá de forma automatizada; de esta forma el jugador podrá centrarse en utilizar las estructuras de control y los elementos básicos de un lenguaje de programación.

Esta herramienta está orientada hacia el lenguaje de programación Java, aunque se proporciona soporte para incluir nuevos lenguajes en el futuro.

Índice

ÍNDICE DE CÓDIGO

ÍNDICE DE FIGURAS

1 OBJETIVOS

El objetivo de este proyecto es el desarrollo de un videojuego didáctico para dispositivos Android. El videojuego enseñará los elementos básicos de un lenguaje de programación, utilizando las características propias de los dispositivos móviles para facilitar el aprendizaje.

Aprender un lenguaje de programación, sin tener conocimientos previos de la materia, es difícil. En los grados de ciencias, la asignatura de programación suele resultar difícil para muchos alumnos, que no han tenido contacto con un lenguaje de programación anteriormente.

El videojuego puede resultar útil para ayudar a los alumnos en las primeras fases del aprendizaje. Se centra en aquellos en los que la programación difiere del resto de asignaturas; no en memorizar teoría, sino en aprender a pensar de forma estructurada y utilizar bucles y condicionales. Y una aplicación interactiva es el mejor medio para mostrarlos de forma intuitiva.

Este proyecto pretende crear un videojuego que aporte un conocimiento básico sobre un lenguaje de programación. Al finalizar el videojuego, el jugador conocerá algunas nociones acerca de la programación, la sintaxis de un lenguaje en concreto y estará acostumbrado a diseñar programas.

El videojuego estará dividido en niveles, cada uno con un determinado objetivo que se debe cumplir. En ellos, el jugador deberá escribir unas líneas de código que controlen a su personaje y lleven a este a superar el nivel. Cada nivel tendrá componentes de azar y repetición, diseñados para que el usuario incluya en su código estructuras de control.

El código se escribirá utilizando un teclado especial. Este teclado estará compuesto por los diferentes elementos que pueden ser necesarios al programar, y los escribirá automáticamente al pulsar las teclas correctas. Se mostrará la sintaxis correcta del lenguaje a la vez que el jugador se abstrae de la escritura.

Una vez que el código esté escrito, el videojuego mostrará de forma gráfica el desarrollo del mismo. Se mostrará el movimiento del personaje y cómo se recorren los bucles.

Además, el videojuego debe ser atractivo estéticamente y fácil de usar. Utilizarlo no debe aportar una dificultad añadida a la de aprender un lenguaje de programación.

Tras terminar el videojuego el jugador debe saber utilizar los siguientes elementos del lenguaje de programación Java:

- La estructura de clases de Java.

- Objetos y variables.

- Estructuras de control.

La aplicación aportará soporte para enseñar diversos lenguajes de programación, aunque esta primera versión únicamente incluirá Java.

En esta memoria se explicará cómo se ha desarrollado este videojuego. Se ha dividido en siete capítulos principales, siendo esta página el primero de ellos.

A continuación, se expone un estudio de las herramientas similares que ya existen en el mercado.

Después se dará una visión general de las tecnologías utilizadas, centradas especialmente en Android.

En el cuarto capítulo se explicará con más detenimiento cómo funciona el videojuego y, en el quinto, se detalla cómo se ha implementado.

Después se muestran las pruebas que se han realizado para comprobar su funcionamiento y, para terminar, se incluye un capítulo con las conclusiones y líneas futuras.

2 INTRODUCCIÓN

Los videojuegos para aprender programación no son una idea novedosa: hay muchos ejemplos de videojuegos y herramientas didácticas con ese mismo objetivo. En muchos de estos casos no solo se pretende enseñar un lenguaje de programación, sino que este es un elemento fundamental en la mecánica del videojuego.

Antes de realizar el proyecto, se han investigado estas alternativas. Se han dividido en dos tipos: las que están orientadas para ser usadas con un ordenador y las aplicaciones móviles. Aunque ambos tipos de herramientas tienen el mismo objetivo, lo abordan de forma diferente, dada la diferencia de plataformas. Las aplicaciones móviles son más cercanas a este proyecto, pero no por ello se deben dejar de lado las demás.

2.1 Herramientas para ordenador

Estas son algunas de las más herramientas más destacadas para ordenador [1] (muchas de ellas están en inglés). La mayoría de estas páginas funcionan proponiendo un reto o ejercicio que el usuario deberá completar escribiendo algo de código.

- **CodeAcademy** [3]. Es una herramienta didáctica que enseña los elementos básicos de muchos lenguajes de programación, de forma fácil y cómoda.

  En este caso, no se plantea el aprendizaje a través de un videojuego. CodeAcademy hace uso de lecciones más tradicionales, con ejercicios sencillos para ponerlas en práctica.

- **Code Maven** y **Game Maven** [4]. Esta página enseña interactivamente a utilizar JavaScript y a usar este lenguaje para programar un pequeño videojuego. Va guiando al usuario paso a paso, enseñando los elementos básicos del lenguaje, a utilizar un *canvas* y cómo darle movimiento a los elementos del videojuego.

- **Ruby Warrior** [5]. Este es un videojuego propiamente dicho. El jugador controla a un guerrero, con el objetivo de derrotar a unos monstruos y salir de la mazmorra.

  Para ello se utilizando el lenguaje de programación Ruby. En el interior de un bucle (que se repite una vez por turno), se deben escribir condicionales y acciones, para diseñar qué hará el guerrero en cada situación.

  Es entretenido y crear el código requiere un esfuerzo de lógica por parte del jugador, aunque, si se desconoce Ruby, es necesario consultar la sintaxis en algún medio externo.

- **CodeCombat** [6]. Similar al anterior, en este juego el jugador debe programar, esta vez en Java, para conseguir que un guerrero derrote a los enemigos y cumpla diversas misiones. Sin embargo, es menos entretenido que Ruby Warrior y tiene una progresión muy lenta a la hora de profundizar en el lenguaje de programación.

- **CodinGame** [7]. En este videojuego, disponible en muchos lenguajes de programación, se presentan retos de varios niveles de dificultad que se deben resolver escribiendo un pequeño programa en el lenguaje especificado.

Requiere un conocimiento previo del lenguaje en concreto, aunque sea básico. Este es un videojuego para programadores; tiene concursos online en los que gana el que realice el mejor programa, un ranking de jugadores y, en resumen, muchos juegos centrados en la programación. Es interesante, pero no la mejor opción si se está aprendiendo.

2.2 Herramientas para móvil

En el mundo de las aplicaciones móviles, que es donde este proyecto tiene lugar, este tipo de herramientas tienen menos acogida, aunque no son inexistentes.

Esta plataforma tiene la desventaja de no contar con un teclado físico. Esto complica la escritura de código, lo que lleva a las aplicaciones a tomar otro enfoque: no piden que el usuario escriba código, sino que utilizan formas alternativas de crear código o pseudocódigo, aprovechando la interfaz táctil de los móviles.

Estas son algunas de las más conocidas [2]:

- **CodeAcademy** [8]. Es una versión simplificada de su equivalente web.
- **Lightbot** [9]. Este juego, destinado a los más pequeños, está protagonizado por un pequeño robot que tiene que moverse por un tablero y encender determinadas zonas.

  Esta aplicación no utiliza un lenguaje de programación, sino que le da al jugador unos botones que representan las acciones que puede realizar el robot (andar, girar o saltar, por ejemplo). El jugador utiliza esos botones para diseñar el movimiento del robot, utilizando bucles y funciones.

  El objetivo de la aplicación no es enseñar un lenguaje de programación, sino acostumbrar a los niños a utilizar funciones y recursividad mientras juegan.
- **Hopscotch** [10]. Esta aplicación utiliza pseudocódigo para crear multitud de programas. El usuario puede utilizar diferentes órdenes y condicionales (como “cuando se inicia el proyecto” o “cuando toque el suelo”) para diseñar pequeños juegos.

  Es una propuesta interesante. Los jugadores aprenden a cómo diseñar un programa, empleando un pensamiento estructurado. El gran punto negativo que tiene es que no enseña un lenguaje de programación: únicamente se programa en pseudocódigo; si el jugador planea aprender un lenguaje de programación en concreto después de practicar con esta aplicación, tendrá que empezar desde cero.
- **Hakitzu Elite: Robot Hackers** [11]. El objetivo de este juego es controlar a un equipo de robots para que derrote al equipo contrario y hackée su núcleo. Cada robot se controla escribiendo un pequeño programa en cada turno, en el que se dicta con órdenes de JavaScript lo que debe hacer.

  Este videojuego no enseña un lenguaje de programación. Está ambientado en el mundo de la programación, las órdenes de los robots son de JavaScript y, entre nivel y nivel, se dan consejos de JavaScript; pero no se utiliza ningún tipo de estructuras de control y el videojuego no se centra en enseñar el lenguaje.

2.3 Visión general

Cada una de estas páginas y aplicaciones tiene puntos fuertes y débiles. Estas herramientas están entre un punto medio entre un videojuego y un recurso didáctico; esto hace que se decanten por un lado u otro. Las que enseñan de una forma más clara un lenguaje de programación (como CodeAcademy o Code Maven) carecen de la interacción de un videojuego, mientras que los

videojuegos que tienen mejor jugabilidad (como RubyWarrior o Hakitzu) dejan de lado la parte didáctica, y no enseñan de forma ordenada el lenguaje de programación.

El conocimiento inicial del jugador es una variable a tener en cuenta a la hora de elegir qué aplicación utilizar. Algunos de estos juegos son de nivel muy básico, destinado casi a niños (como Lightbot o CodeCombat), mientras que, por el otro lado, otras están destinadas a usuarios con un nivel de programación básico o avanzado (CodinGame).

Por último, hay que considerar qué es lo que queremos aprender: los principios de la programación de forma general (Hopscotch) o un lenguaje de programación concreto.

Una característica común que suelen tener las aplicaciones móviles investigadas es que no tienen código real. La carencia de un teclado físico en los terminales obliga a buscar otras soluciones para escribir el código, ya que el teclado software no es cómodo en este contexto. Son pocas las aplicaciones (como Hakitzu) que requieren teclear el código; la mayoría utilizan botones que escriben de forma automática el código o pseudocódigo.

En este proyecto se ha optado crear un videojuego que, aunque sea jugable, no deje de lado la parte didáctica. El objetivo será enseñar un lenguaje de programación en concreto, y estará destinado a usuarios con conocimientos mínimos o inexistentes del mismo. Mientras se juega se enseñarán sus elementos básicos, centrando los niveles en practicar las estructuras de control.

3 TECNOLOGÍAS UTILIZADAS

Esta aplicación ha sido desarrollada para Android, un sistema operativo presente en la mayoría de los dispositivos móviles. Se ha elegido esta plataforma debido a su fácil acceso, su expansión y su disponibilidad a la hora del desarrollo del proyecto.

A continuación se ofrece una breve explicación del sistema operativo Android, seguida de un apartado más concreto acerca de algunos elementos que hay que conocer para programar una aplicación. Esta sección explicará algunos conceptos que es necesario entender para poder seguir el capítulo cinco de esta memoria, que detallará la implementación de la aplicación.

Una vez detallado lo que hay que conocer para desarrollar una aplicación Android, se explicará JSON (*JavaScript Object Notation*), que es un formato de datos utilizado en el proyecto.

Y, para terminar, se incluye un breve resumen de lo visto en el apartado.

3.1 Introducción a Android

Android es, a día de hoy, el sistema operativo con mayor expansión en el mercado de teléfonos móviles.

Ilustración 1 – Gráfico del mercado de Sistemas Operativos en 2014 [14]

Android está basado en Linux y fue diseñado en el año 2007 por Android Inc., empresa financiada y comprada por Google. Inicialmente estaba destinado a terminales móviles, aunque actualmente se puede encontrar en todo tipo de dispositivos.

Android es código libre, pero con matices. Se distribuye bajo dos tipos de licencias: GNU GPLv2 para el kernel y APACHE v2 para el resto de los componentes del sistema. La licencia del kernel indica que éste está a disposición de todo el mundo, y puede modificarse como se desee. Sin embargo, las modificaciones deben hacerse bajo la misma licencia, estando a disposición del todo el mundo.

Con el resto de los componentes ocurre de forma diferente: los fabricantes pueden modificarlos

como les convenga, y estas modificaciones no tienen por qué ser registradas bajo la misma licencia. Esto da mayor libertad a los fabricantes, permitiéndoles modificar el sistema sin la necesidad de licenciar gratuitamente estos desarrollos [13].

En este proyecto nos interesa especialmente saber cómo funcionan las aplicaciones Android. Están escritas en Java y ejecutadas en una máquina virtual (donde se convierten en un código que entienda el kernel basado en Linux).

3.2 Aplicaciones Android

En este apartado se dará una breve introducción al desarrollo de aplicaciones, y en los siguientes subapartados se explicarán algunos conceptos concretos [15] [16] [17].

Para realizar una aplicación propia necesitamos un entorno de desarrollo y el kit de desarrollo software (conocido como SDK). Con estos dos elementos podremos crear una aplicación y, tras pagar una cuota de alta, subirla al mercado de aplicaciones de Google y ponerla a disposición de todo el mundo.

Android ha pasado por diferentes versiones desde su desarrollo. En la tabla siguiente se muestra el porcentaje de uso de cada una de las versiones, así como el número de versión, su nombre y el nivel de API (Interfaz de Programación de Aplicaciones):

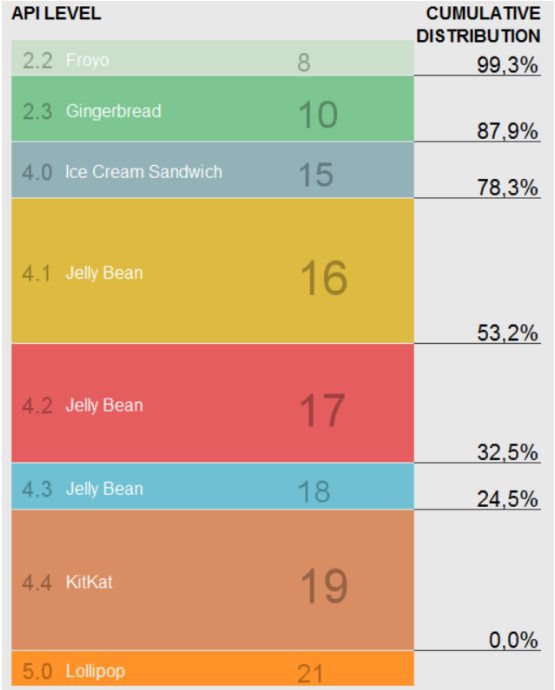

Ilustración 2 – Porcentaje de uso de las versiones de Android

Es interesante conocer el porcentaje de uso de cada una de las distintas versiones para elegir correctamente el nivel de API mínima para el que desarrollar la aplicación, ya que algunas de las bibliotecas incluidas en las API más recientes no son compatibles con las antiguas.

Las aplicaciones se apoyan en las bibliotecas de Android para funcionar y es deber del programador tener conocimiento de ellas. Gran parte del éxito a la hora de desarrollar una aplicación radica en la habilidad del programador para buscar en la documentación de Android y utilizar las prestaciones que ofrecen las diferentes bibliotecas.

A continuación se detallan algunas facetas concretas del desarrollo de una aplicación Android.

3.2.1 Android Studio

En este proyecto se ha utilizado Android Studio, el entorno de desarrollo creado por Google y destinado exclusivamente a Android. Otra opción válida habría sido descargar Eclipse y añadir el kit de desarrollo software.

Si iniciamos Android Studio podemos crear un nuevo proyecto, que contendrá todo el código fuente de la aplicación. Inicialmente se nos pide configurar los datos básicos de nuestra aplicación: nombre de la aplicación, nombre del paquete y el nivel de API mínimo.

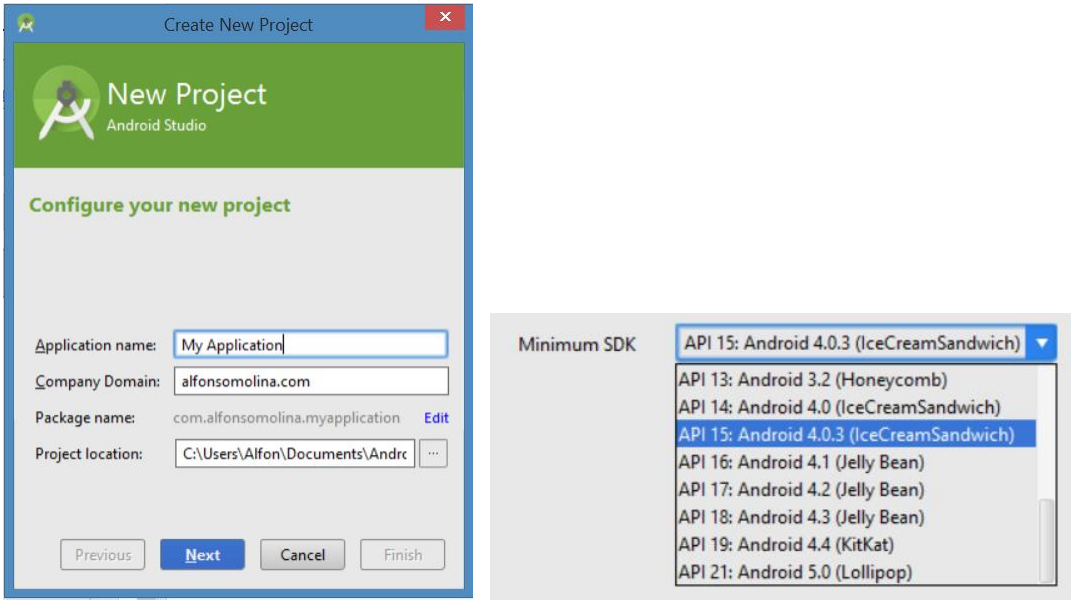

Ilustración 3 – Nuevo proyecto en Android Studio

El nombre de la aplicación es el nombre que aparecerá en la lista de aplicaciones una vez instalada en el teléfono.

El segundo elemento es el nombre del paquete Java. Android lo hace por defecto utilizando el dominio de la compañía que realiza la aplicación, aunque es únicamente una convención; el programador puede poner el nombre que quiera. Una opción común suele ser inventar un dominio ficticio o usar la dirección del repositorio del código (*github*, por ejemplo).

Si el dominio de la compañía es *dominio.com* y el nombre de la aplicación es *app*, el nombre del paquete será *com.dominio.app*. Este nombre de paquete debe ser único en el dispositivo Android en que esté instalado.

La API mínima determina a partir de qué dispositivos funcionará la aplicación. Los terminales que tenga una versión de Android anterior a la fijada no podrán utilizarla.

Una vez creado el proyecto y una actividad básica, Android Studio muestra así la estructura de ficheros (en las siguientes secciones se explicarán qué son los distintos elementos):

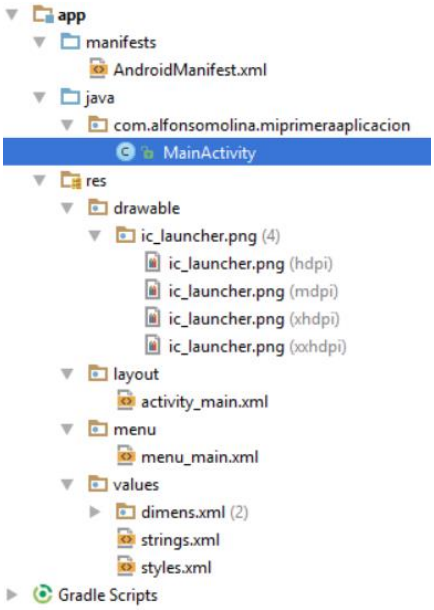

Ilustración 4 – Estructura de ficheros en Android Studio

Android Studio también tiene incluido un emulador de Android para probar las aplicaciones. Primero debemos crearlo con la versión de Android y el modelo de teléfono que queramos. Una vez creado, podremos iniciarlo y mandarle la aplicación fácilmente.

Otra opción disponible es probar la aplicación en un dispositivo físico. Hay que tener en cuenta dos cosas para utilizar un dispositivo real:

1) Los drivers tienen que estar instalados en el ordenador. Normalmente se instalarán automáticamente, aunque con algunos dispositivos será necesario descargarlos del fabricante.
2) Se tiene que haber activado "Depuración USB" en el terminal. Estas opciones normalmente están ocultas, y para mostrarlas hay que pulsar varias veces en algún sitio, como en "Número de compilación".

Android Studio tiene otras herramientas útiles: busca errores de sintaxis, recomienda cambios en el código, nos facilita un depurador y tiene un editor gráfico para crear los *layout*, entre otras cosas.

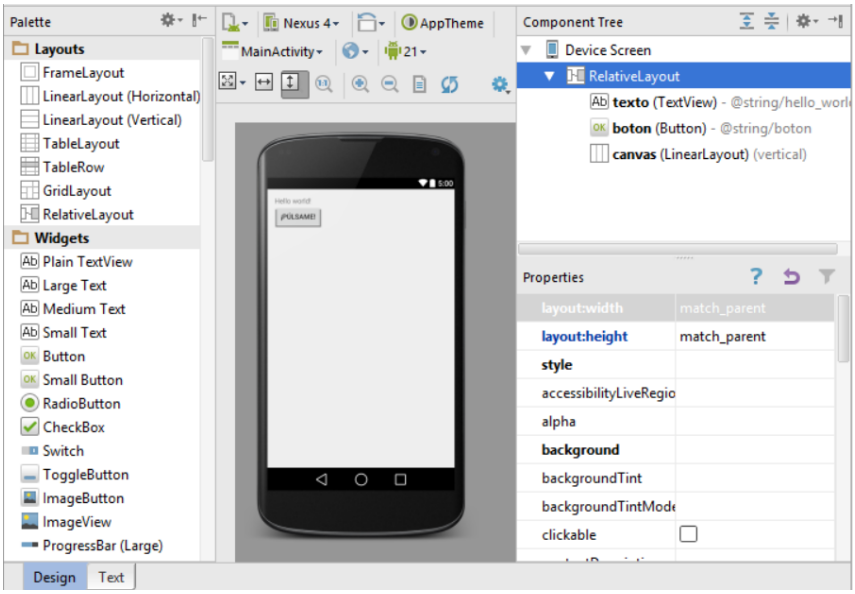

Ilustración 5 – Editor de Android Studio

3.2.2 Manifest

El archivo *AndroidManifest.xml* es, como el manifiesto de carga de un barco, una lista de todo lo que hay presente en nuestra aplicación. En este archivo debemos incluir el nombre del paquete Java, la API mínima, los componentes de la aplicación, los permisos que necesitamos para ejecutar la aplicación, el icono que se mostrará en la pantalla del móvil, cual es la actividad principal y otros puntos de información vital para la aplicación.

Este es el *manifest* de una aplicación simple:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.alfonsomolina.miprimeraaplicacion" >
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Código 1 – AndroidManifest.xml

En estas líneas se listan las actividades de la aplicación y se indica qué icono se usará (con el atributo “android:icon”), qué nombre tendrá (“android:label”) y cuál es la MainActivity (con el “intent-filter” en el interior de la actividad).

El archivo manifest admite otros atributos y elementos más específicos. Por ejemplo, dentro de cada actividad se puede utilizar el atributo “android:screenOrientation”, obligando a dicha actividad a mostrarse siempre en una orientación determinada (*portrait* o *landscape*). Por otra parte, el elemento “user-permission”, introducido como hijo del raíz “manifest”, especifica qué recursos son utilizados por la aplicación (por ejemplo, si se usa la cámara o internet).

El archivo *manifest* es fundamental que esté correctamente estructurado. Si se está utilizando Android Studio, el entorno de desarrollo ayudará en la creación del *manifest*, añadiendo las actividades conforme estas es vayan creando.

3.2.3 Actividades y layouts

Las actividades son uno de los cuatro componentes principales de Android (los otros tres son los servicios, los proveedores de contenidos y los receptores de difusión, que no se utilizan en este proyecto). Una actividad es una única pantalla de la aplicación, mostrada en el teléfono a través de una interfaz gráfica. Por lo general, una aplicación estará formada por varias actividades diferentes. Aunque trabajen conjuntamente para el usuario, cada una de ellas será independiente de las demás.

Una actividad se divide en dos partes: la parte lógica y la interfaz de usuario (IU). La IU es declarada en un *layout* donde se indican los elementos gráficos que se mostrarán en pantalla. La lógica, por su parte, es una clase java que extiende la clase *Activity*. Sus métodos indican qué es lo que hace cada uno de los elementos gráficos y los modifican cuando sea necesario.

El *layout* es un archivo xml. Está compuesto por dos tipos de elementos: las vistas y los grupos de vistas.

Las vistas son cada uno de los elementos que se muestran en pantalla: un botón (*Button*), un bloque de texto (*TextView*) o una imagen (*ImageView)*, por ejemplo.

Los grupos de vistas son contenedores de vistas, y el archivo xml tendrá siempre uno de ellos como raíz. Los dos grupos de vistas más utilizados son *LinearLayout* y *RelativeLayout*. En el primero las vistas hijas se dibujan una al lado de la otra en la pantalla, con orientación horizontal o vertical. En un layout relativo tenemos más libertad de acción, y podemos dibujar las vistas utilizando cualquier tipo de relación entre ellas.

El *layout* de una actividad que muestra un bloque de texto y un botón sería así:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:paddingLeft="16dp" android:paddingRight="16dp"
    android:paddingTop="16dp" android:paddingBottom="16dp"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/texto"
        android:text="@string/hello_world"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <Button
        android:id="@+id/boton"
        android:text="@string/boton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/texto"
        android:onClick="cambiarTexto"/>

</RelativeLayout>
```

Código 2 – layout_main_activity.xml

Es un *layout* relativo, así que indicamos que el botón está por debajo del texto usando el atributo "android:layout_below".

Cada vista tiene varios atributos disponibles para configurar sus propiedades. Hay algunos particulares de cada vista y otros comunes a casi todas, como el identificador ("android:id"). Este atributo es importante, ya que es la forma más fácil de referenciar una vista desde el código.

Otros atributos son "android:layout_width" y "android:layout_height", que se utilizan para determinar la anchura y la altura de las vistas, respectivamente. En este caso, las vistas ocupan en pantalla el espacio necesario para mostrar su contenido, incluido con el atributo "android:text" (este atributo únicamente está presente en aquellas vistas que muestran algún texto en pantalla).

Utilizando las vistas adecuadas y configurando los atributos correctamente es posible mostrar por pantalla cualquier interfaz que se le ocurra al programador.

Ahora que la parte gráfica está lista, hay que crear la lógica de la aplicación.

Como se ha dicho anteriormente, para programar cómo se comportaran esos elementos del *layout* necesitamos crear una clase que extienda a una clase derivada de *Activity* (como *ActionBarActivity*, que es una actividad con una barra con opciones en la parte superior). La llamaremos MainActivity.java.

```
public class MainActivity extends ActionBarActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }


    public void cambiarTexto(View view) {
        TextView texto = (TextView) findViewById(R.id.texto);
        texto.setText("Texto modificado.");
    }
}
```

Código 3 - MainActivity.java

Esta clase no debe confundirse con la clase *main* de Java. En Android, se conoce como MainActivity a la clase que se inicia al arrancar la aplicación. No tiene por qué iniciarse siempre la primera; dada la independencia de las actividades de Android puede lanzarse un *intent* e iniciar otra distinta. Tampoco es obligatorio llamarla MainActivity; este nombre es una convención.

"onCreate" es un método que se ejecuta cada vez que la aplicación es cargada en memoria. En este caso lo único que hace es elegir qué *layout* es el que se mostrará por pantalla. Esto es uno de los varios métodos que controlan su ciclo de vida. En el siguiente diagrama se muestra el ciclo de vida de una actividad y qué métodos se ejecutan en cada paso:

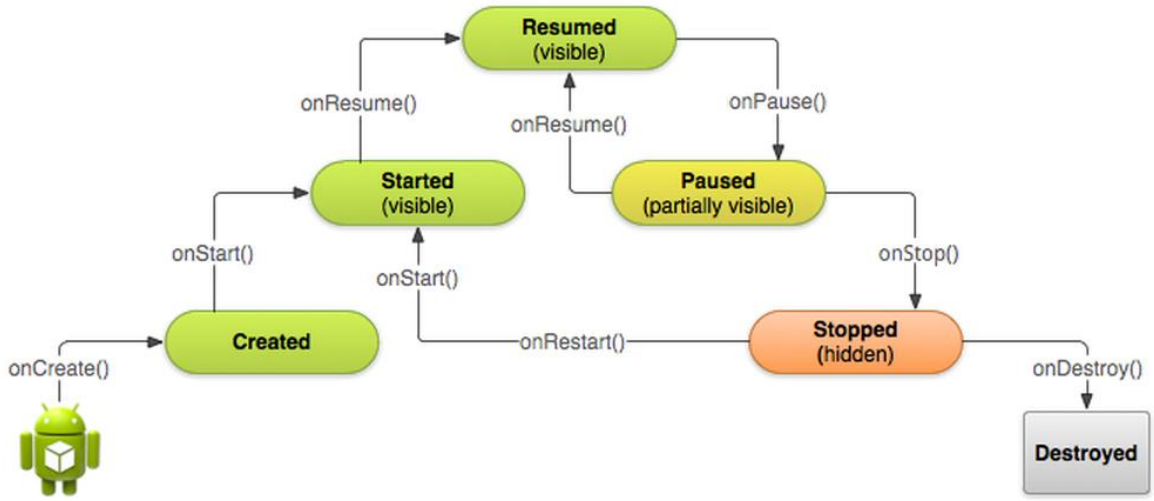

Ilustración 6 – Ciclo de vida de la actividad

El otro método que aparece se llama "cambiarTexto". Este es el método que se ejecuta cuando se pulsa el botón (lo hemos indicado así en el *layout*, utilizando el atributo "android:onClick"). Al pulsar el botón, la actvidida busca la vista *TextView* con el identificador "texto" y cambia su contenido.

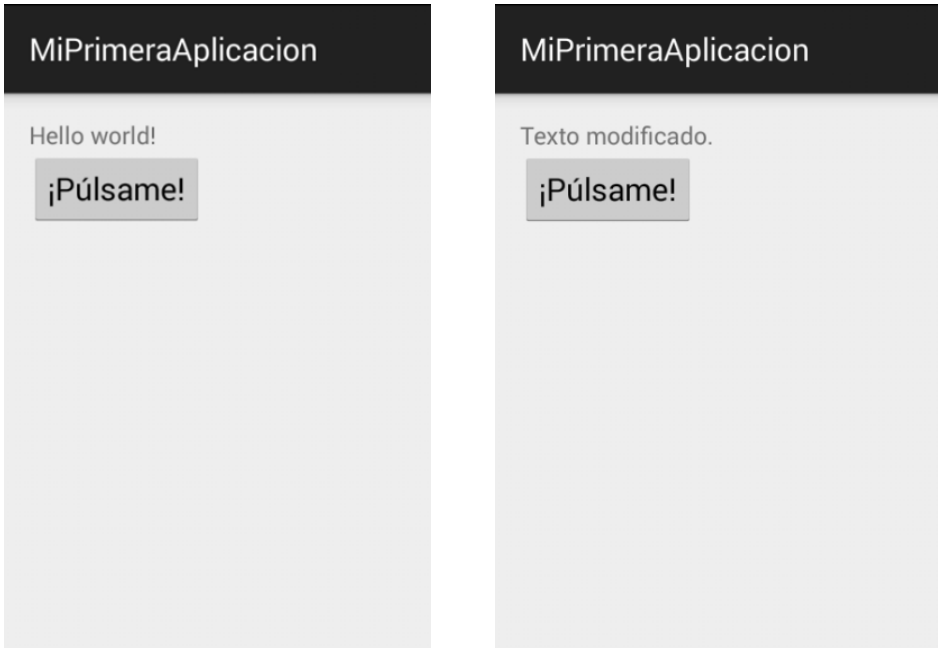

Ilustración 7 – Ejecución de MiPrimeraAplicacion

Para cambiar de actividad se utilizan los intents, ya sea para iniciar una actividad de la misma aplicación u otra de una aplicación distinta (por ejemplo, si queremos echar una foto podemos abrir la aplicación de la cámara utilizando un intent).

Cuando una actividad se inicia, la que estaba antes es pausada y añadida a la pila de actividades. Si pulsamos el botón de "Atrás", la actividad actual se destruye y vuelve a estar activa la última que fue pausada.

A un intent podemos añadirle datos extra, de muchos tipos distintos. En la nueva actividad se pueden extraer esos datos y utilizarlos.

```
public void iniciarActividad(View view) {
    int dato = 3;
    Intent intent = new Intent(this, Actividad2.class);
    intent.putExtra("DATO EXTRA", dato);
    startActivity(intent);
}
```

Código 4 – Ejemplo de intent

3.2.4 Recursos

Un proyecto Android se divide principalmente en dos partes: el código de la aplicación (en la carpeta "java") y los recursos (en la carpeta "res").

Los recursos son los diferentes archivos que utiliza la aplicación. Pueden ser de extensiones muy distintas (imágenes, texto plano o xml, por ejemplo) y se pueden utilizar desde cualquier lugar de la aplicación. Algunos de los diferentes tipos de recursos son:

- **Drawable**. Esta carpeta contiene las imágenes que se van a utilizar en la aplicación.
- **Layout**. En esta carpeta se encuentran los *layout* de las actividades.
- **Menu**. Si una actividad extiende *ActionBarActivity*, en la parte superior de la pantalla aparecerá una barra con un menú. Los elementos que aparecen en el menú estarán definidos en un fichero xml que se encontrarán aquí.

- **Values**. Una aplicación muchas veces necesita utilizar valores predefinidos de esta carpeta. Uno de los ficheros más utilizados es "strings.xml", que contiene las diferentes cadenas de caracteres que utilizamos en la aplicación.

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

    <string name="app_name">MiPrimeraAplicacion</string>
    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>
    <string name="boton">¡Púlsame!</string>

</resources>
```

Código 5 – strings.xml

- **Raw**. Para guardar archivos arbitrarios en su forma original.

Cada recurso tiene un identificador asignado automáticamente por Android. Podemos acceder al recurso utilizando dicho identificador, escribiendo "R.recurso.nombre" en las clases java y "@recurso/nombre" en los ficheros xml.

Utilizar la carpeta de recursos permite a Android dar sporte para diferentes idiomas, pantallas y orientaciones. Cada carpeta está a su vez dividida en subcarpetas, cada una de ellas con una copia de los recursos adaptada al soporte particular.

Es común que la carpeta de "values" sea divida en verias carpetas para dar soporte a diferentes idiomas: "values-en", "values-fr", etc. Siempre que en ambos casos el recurso tenga el mismo nombre, cuando se referencie el recurso desde la aplicación se accederá, si es posible, a la carpeta correspondiente al idioma del dispositivo, y no a la carpeta por defecto.

De forma similar, se pueden crear imágenes para diferentes densidades de pantalla en la carpeta "drawable" y *layouts* diferentes para la orientación vertical (*portrait)* y horizontal (*landscape)*.

3.2.5 Imágenes y animación

3.2.5.1 Imágenes 9.path

Las imágenes 9.path tienen la facultad de adaptarse automáticamente, sin distorsionarse, al tamaño de la vista en la que estén. Este tipo de imágenes son útiles para poner fondos personalizados para botones y vistas similares.

Este tipo de imágenes necesitan un borde de, como mínimo, un píxel de grosor de color transparente o blanco, en el que se dibujarán píxeles de color negro en las zonas que pueden estirarse, dejando el borde intacto en las zonas que no deben cambiar de tamaño.

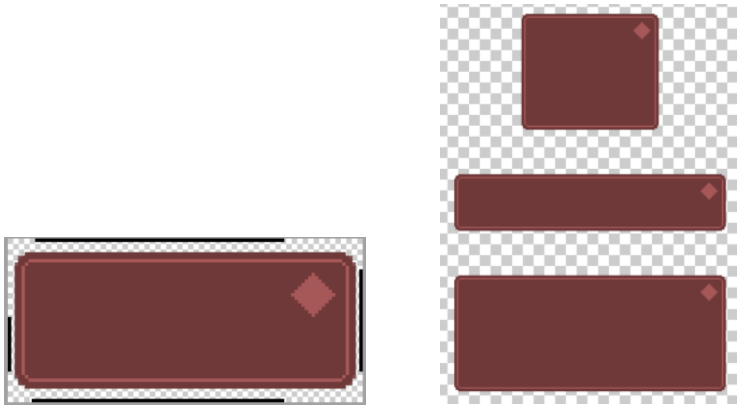

Ilustración 8 – Imágenes 9.path

La zona en la que se cruzan las líneas de los bordes superior e izquierdo es aquella que se puede estirar sin problemas. Los bordes inferior y derecho, por su parte, indican la zona en la que debe ir el contenido de la vista (si fuese un botón o un bloque de texto, en esa zona es donde se escribiría).

3.2.5.2 Drawable Animation

Esta es la técnica de animación más simple: se crean imágenes similares y se reproducen una tras otra. Para ello, una vez tengamos las imágenes, hay que crear un fichero xml que indica cuáles hay que mostrar en la animación y cuántos milisegundos dura cada una. Además, es de utilidad el atributo "android:oneshot", que indica si la animación se debe repetir indefinidamente o sólo una vez.

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false">
    <item android:drawable="@drawable/imagen1" android:duration="300" />
    <item android:drawable="@drawable/imagen2" android:duration="300" />
</animation-list>
```

Código 6 – animacionEjemplo.xml

A la hora de programar, estas imágenes se usan de la misma forma que una imagen normal. Si se quiere poner en un *layout*, únicamente habría que escribir "@drawable/nombre_de_la_imagen". La única diferencia es que, si queremos acceder a la animación desde la clase java, debemos usar la clase *AnimationDrawable*, en vez de *ImageView*.

3.2.5.3 Canvas

Esta es una forma de animación más compleja. Consiste en dibujar cosas sobre un *canvas*, o lienzo, y mostrar la imagen dibujada en una *SurfaceView*, una vista especial destinada a acoger dibujos.

Primero se necesita un *LinearLayout* como contenedor de la *SurfacteView* en la *layout* de la actividad, al que pondremos de identificador "canvas".

Después creamos una nueva clase que extienda *SurfaceView*:

```
public class Lienzo extends SurfaceView{

    public Lienzo(Context context){
        super(context);
    }

    public void dibujar(Canvas canvas){
        canvas.drawColor(Color.BLACK);
        Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.roca);
        canvas.drawBitmap(bitmap,0,0,null);
    }
}
```

Código 7 – Lienzo.java

El método "dibujar" recibe el canvas y dibuja en él una roca. Primero pinta el canvas de negro, después obtiene el mapa de bits de la imagen a partir del identificador del recurso y por último la dibuja en la posición (0,0) del canvas, correspondiente a la esquina superior izquierda.

Ahora que esta clase está creada hay que prepararla desde el código de la actividad. Desde el método "onCreate" se va a instanciar un objeto de esta clase y añadir al contenedor que hemos creado antes:

```
public class MainActivity extends ActionBarActivity {
    Lienzo actividad2;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        actividad2 = new Lienzo(this);
        LinearLayout surface = (LinearLayout) findViewById(R.id.canvas);
        surface.addView(actividad2);
    }


    public void dibujarRoca(View view) {
        Canvas c = actividad2.getHolder().lockCanvas();
        actividad2.dibujar(c);
        actividad2.getHolder().unlockCanvasAndPost(c);

    }
}
```

Código 8 - MainActivity.java (usando el Canvas)

Y desde el método "dibujarRoca" (que se ejecuta al clicar en el botón) ordenamos a la *SurfaceView* que dibuje su contenido. En el terminal se verá así:

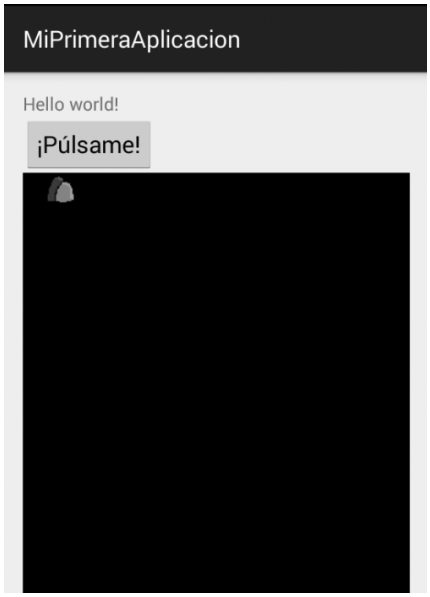

Ilustración 9 – SurfaceView en ejecución

A partir de estos conceptos se puede crear animación. Para ello hay que dibujar varias veces en el canvas e ir modificando el contenido en cada iteración. Si, por ejemplo, cada vez se dibuja la roca un poco más a la derecha se simulará movimiento.

3.2.6 Persistencia de datos

Las aplicaciones no mantienen información de una ejecución a otra. Si queremos que haya datos con persistencia entre ejecuciones, tenemos que implementarlo nosotros. Una de las opciones más utilizadas es hacer uso de las preferencias compartidas (o *SharedPreferences*).

Las *SharedPreferences* son pares nombre-valor, guardados en un directorio propio de la aplicación. Desde cualquier lugar de la aplicación podemos acceder a los valores guardados (con un valor por defecto por si no existe) y modificarlos, utilizando el editor.

En el siguiente ejemplo se muestra cómo utilizarlas. Primero se obtiene el fichero de preferencias con el identificador "PREF", que será abierto con el modo de operación "MODE_PRIVATE" (el modo por defecto, de esta forma el fichero sólo será accesible por la aplicación que le ha llamado).

A continuación, se obtiene el entero que está guardado bajo el nombre "NOMBRE", obteniendo el valor cero si este no existe.

```
SharedPreferences sharedPref = this.getSharedPreferences("PREF", Context.MODE_PRIVATE);
int valor1 = sharedPref.getInt("NOMBRE", 0);
```

Código 9 – Ejemplo SharedPreferences I

Para añadir o editar algún valor es necesario un editor. En el ejemplo siguiente se obtiene el editor con el método "edit" y se añade el entero llamado "valor2" con el nombre "NOMBRE". Si ya existía, será modificado.

Para que los cambios sean efectivos no se debe olvidar la orden "editor.apply()".

```
SharedPreferences.Editor editor = sharedPref.edit();
int valor2 = 0;
editor.putInt("NOMBRE",valor2);
editor.apply();
```

Código 10 – Ejemplo ShardePreferences II

3.3 Otras tecnologías

3.3.1 JSON

JSON [29] es un formato estándar para codificar información. Se compone de tres tipos de datos: *arrays* (con corchetes), objetos (con llaves) y pares nombre-valor.

Para ejemplificar este formato se ha creado un ejemplo muy simple de una agenda [28]:

```
{
    "agenda":[
    {
        "nombre":"Persona1",
        "telefono":"telefono1"
    },
    {
        "nombre":"Persona2",
        "telefono":"telefono2"
    }
  ]
}
```

Código 11 – agenda.json

En este pequeño ejemplo se pueden ver todos los elementos que se pueden usar en JSON. Hay un objeto raíz, delimitado por las comillas, y dos objetos más pequeños que representan a un contacto de la agenda. Los contactos son elementos de un array llamado "agenda", representado por los corchetes.

Leerlo es muy fácil, utilizando la biblioteca *org.json* [18]. Este es un ejemplo de cómo obtener el nombre del primer contacto de la agenda:

```
try{
    JSONObject jsonObject = new JSONObject(cadena);
    JSONArray agenda = jsonObject.getJSONArray("agenda");
    JSONObject contacto = agenda.getJSONObject(0);
    String nombre = contacto.getString("nombre");
} catch (JSONException e) {
    e.printStackTrace();
}
```

Código 12 – Ejemplo leer JSON

Primero se obtiene el objeto raíz del archivo JSON (guardado como una *string* en "cadena") y a partir de ahí vamos obteniendo los elementos hijos de cada objeto o *array*.

Siempre que utilizamos una función de esta biblioteca debemos tener en cuenta que lanzan una excepción *JSONException* si el elemento que queremos obtener no existe en el archivo JSON. Por ello hay que introducir cada consulta en un bloque *try-catch*, y hacer lo que sea necesaria si la lectura falla.

3.4 Conclusiones

En este capítulo se ha visto una introducción a las aplicaciones Android y a muchos de los elementos que es necesario conocer para programarlas. Más concretamente, el capítulo se ha centrado en aquellos elementos que se han utilizado en la realización de este proyecto. Este conocimiento es necesario para entender los capítulos siguientes, ya que explican la implementación de la aplicación.

En el capítulo siguiente se dará una visión general de la aplicación y, a continuación, se procederá con la implementación en sí.

4 APLICACIÓN PROPUESTA

La aplicación que se ha desarrollado en este proyecto es un videojuego con el que el jugador, a la vez que juega, aprende programación. Para jugar se utiliza un lenguaje de programación (Java, en este caso). En cada nivel del videojuego, el jugador deberá controlar al héroe escribiendo un fragmento de código con el que pueda alcanzar el objetivo del nivel.

El jugador no controlará al personaje paso a paso, sino que diseñará un código inicial que el personaje utilizará para moverse en el nivel. Según se va avanzando en el videojuego, se irán desbloqueando diversas herramientas de programación. Inicialmente únicamente están disponibles los métodos del personaje, y poco a poco se irán añadiendo condicionales, variables, bucles y otros elementos propios de Java.

Se ha realizado una aplicación de tal forma que sea un punto medio entre un videojuego y una herramienta didáctica. Un videojuego debe tener cierta historia, y ser divertido de jugar, pero no se debe olvidar que el objetivo principal del proyecto es enseñar un lenguaje de programación.

En lo referente a programación, se van aportando diferentes elementos y con cada uno de ellos se explica de forma breve qué son y cómo se utilizan. Cada nivel será progresivamente más complicado, y se espera que el jugador se acostumbre a utilizar los elementos que se proporcionan según los vaya superando.

El videojuego, además, tiene una pequeña trama. La protagonista es una chica llamada Ada que, como es corriente en los videojuegos, tiene que salvar su mundo. Este funciona siguiendo las reglas de la programación y se ha visto infestado por las malvadas excepciones.

4.1 Funcionamiento

En esta sección se va a explicar cómo se utiliza la aplicación. Se hará un recorrido por las distintas ventanas que el usuario visitará en el transcurso del videojuego, explicando cómo se utilizan y cual es su función en el mismo.

Para hacer progresos en el videojuego, el jugador deberá superar niveles (o “misiones”) que, a su vez, se dividen en subniveles (que en esta implementación se han definido como “etapas”). Según se van superando, el jugador va avanzando en la historia, desbloqueando nuevos elementos y consiguiendo experiencia, con la cual podrá desbloquear nuevos niveles.

Para superar los niveles, el jugador visitará las distintas actividades de la aplicación. A continuación se explica la primera de ella: el mapa.

4.1.1 Mapa

La actividad principal del videojuego es una vista del mapa. En ella se muestra una ciudad y la lista de misiones que se pueden hacer en ella. Al pulsar en una misión de la lista, aparecerán detalles de la misma, con un botón para ir a ella. Si el jugador ya había entrado en un nivel pero no lo ha completado, se muestra un botón para ir a la última etapa visitada.

Las misiones no desaparecerán una vez realizadas; seguirán presentes para que el jugador pueda volver a jugarlas cuando desee. Aunque las misiones nuevas sí tienen una distinción: aparece un signo de exclimación sobre su botón. Una vez que el jugador accede a ese nivel, aunque no lo supere, el signo de exclamación desaparece.

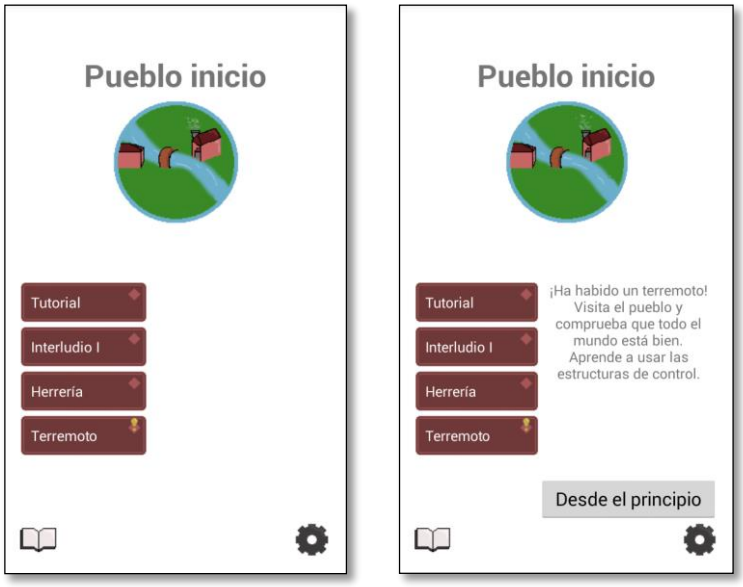

Ilustración 10 - MapaActivity

Esta actividad también tiene enlaces a las opciones (que permite reiniciar el juego y cambiar de idioma) y al libro de apuntes, representados con un engranaje y un libro, respectivamente.

4.1.2 Libro de apuntes

Conforme se avanza en el videojuego se van obteniendo lecciones acerca de distintos elementos de Java, enseñando al jugador a utilizarlos en el juego. Todas estas lecciones son guardadas en el libro de apuntes y pueden consultarse en cualquier momento:

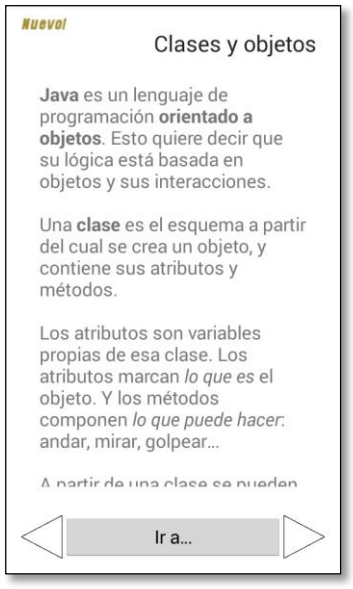

Ilustración 11 – ApuntesActivity

4.1.3 Introducción

Cada nivel empieza con dos personajes hablando, a modo de introducción y como contexto para el nivel. Este diálogo tiene dos funciones: guiará al jugador acerca de qué código tiene que escribir y avanzará en la trama del videojuego.

Ilustración 12 - IntroducciónActivity

4.1.4 Resumen del nivel

En esta actividad se muestra el tablero en el que se realiza la misión y los objetivos (principal y secundario) de la misma.

Los elementos aleatorios del mapa aparecen de forma semitransparente o con un signo de interrogación, indicando al jugador que tiene que tenerlos en cuenta y necesita un código que funcione en los distintos escenarios.

Ilustración 13 – ResumenMisionActivity

4.1.5 Escribir código

El siguiente paso es escribir un fragmento de código necesario para cumplir la misión.

Ilustración 14 – CrearCodigoActivity

El código se escribirá utilizando un teclado especial, en el cual las teclas escriben de forma automática el código utilizando la sintaxis adecuada.

Al iniciar el nivel aparece un globo de texto con información del nuevo elemento de programación desbloqueado y lo añade al libro de apuntes. Muchos niveles también incluyen unas líneas preliminares, para guiar la escritura de código la primera vez que se utiliza cada elemento.

También hay dos iconos a la derecha: uno muestra de nuevo la ayuda y otro el resumen.

4.1.6 Ver ejecución

Por último se realiza la ejecución del códio. En esta actividad se ejecuta una línea de código cada vez que se pulsa la pantalla, mostrando de forma gráfica qué hace el código [12]:

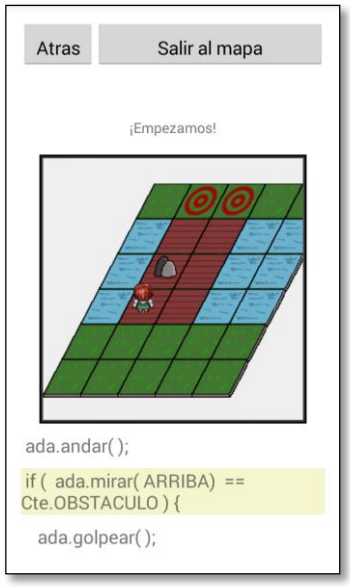

Ilustración 15 – ResolverCodigoActivity

Si el jugador le da al botón "Atrás" (al botón en pantalla o al botón físico) irá de vuelta a la actividad "CrearCódigo", donde podrá modificar lo que ha escrito.

4.2 Desglose de niveles

En esta sección se van a listar los diferentes niveles se han desarrollado para el videojuego y qué herramienta de programación se utilizan en cada una.

- **Tutorial**.

Este nivel es necesario para explicar el funcionamiento de la aplicación. Enseña cómo se utiliza el teclado.

- **Interludio I**.

Este segundo nivel sirve como introducción al lenguaje de programación Java. Explica la existencia de clases y objetos.

- **Herrería**.

Desbloquea algunos métodos nuevos, como "mirar", aumentando las acciones que puede realizar el personaje.

- **Terremoto**.

Se introducen las sentencias de control. Progresivamente se enseñan "if", "else" y "else if".

- **Interludio II**.

Clase de programación que introduce las variables y ejemplifica su uso con un pequeño nivel.

- **Asalta Caminos**.

Se introduce por primera vez un bucle. Se utilizan dos, en dos etapas distintas: while y do-while.

- **Asalta Caminos II**.

No desbloquea ninguna tecla nueva, sino que se pone a prueba la habilidad del usuario en la utilización de bucles y condicionales.

- **La historia de Boole**.

Se enseña la noción de las excepciones. Este nivel también se utiliza para avanzar en la historia del videojuego.

- **Torre**.

Excepciones y Try-catch.

4.3 Conclusiones

Este capítulo ha dado una visión general de cómo funciona el videojuego y, por lo tanto, de qué hay que implementar. Ahora que ya se tiene esta idea en mente se va a proceder con la implementación, explicando cómo se han desarrollado las distintas actividades que se han visto en este apartado.

5 IMPLEMENTACIÓN

En esta sección se va a explicar cómo se ha realizado la implementación del proyecto. Se va a dividir en diferentes secciones, para simplificar la explicación: visión general, interfaz de usuario, escritura de código, animación del tablero de juego, soporte de varios lenguajes y preferencias compartidas.

Cada una de las secciones se centra en una de las facetas del proyecto. Es una división por funcionalidad, y no por estructura; por lo general cada sección hace uso de varias clases y actividades.

En cada sección se va a intentar explicar cuál es el objetivo que se quiere lograr y cómo se ha realizado, mostrando los fragmentos más relevantes del código cuando estos sean necesarios para la explicación. El código no será una copia exacta del proyecto, sino que habrá sido simplificado. No sólo se han sustituido trozos de código por "[...]", sino que el código presente también será modificado para eliminar funcionalidades y centrarlo en las líneas que se quieren explicar.

5.1 Visión General

En la realización de este proyecto se han desarrollado muchos tipos de elementos diferentes: actividades, clases Java auxiliares, ficheros xml, drawables y archivos JSON con la descripción de los distintos niveles del videojuego. En esta sección se prentende dar una perspectiva de todos estos elementos y como trabajan conjuntamente para formar el videojuego.

Las actividades son:

MainActivity – Es la actividad que se muestra cada vez que se inicia la aplicación. Muestra una ventana de bienvenida y, tras un segundo, pasa a la siguiente actividad (en inglés este tipo de pantalla se conocen como *SplashActivity*). También realiza la configuración inicial del videojuego.

MapaActivity – Muestra el mapa del videojuego y las misiones disponibles en la ciudad actual. A partir de ella también se puede acceder al libro de apuntes, cambiar el lenguaje de programación y reiniciar el juego.

ApuntesActivity – Lee en las preferencias compartidas qué apuntes se han desbloqueado y los escribe en pantalla.

IntroduccionActivity – Esta actividad se inicia al pulsar en una misión del mapa. Dibuja en pantalla a dos personajes hablando.

ResumenMisionActivity – Muestra al jugador el tablero de la misión y cual es el objetivo de la misma. Se puede acceder a ella desde IntroduccionActivity, para ver por primera vez el objetivo, y desde CrearCodigoActivity, para recordarlo.

CrearCodigoActivity – En esta actividad es donde se escribe el código. Muestra las líneas escritas y proporciona un teclado para escribir.

ResolverCodigoActivity – Ejecuta el código línea a línea y lo muestra en la pantalla.

Las actividades se ayudan de otras clases. Estas son:

Codigo – Esta clase se utiliza como contenedor para guardar qué se ha escrito en cada línea de código. Se explicará con más detenimiento en su sección correspondiente.

Variable – Cuando en el código se crea una variable, se crea un objeto de este tipo.

Aviso – Mensaje que aparece en pantalla y requiere la confirmación del usuario cuando se quiere salir de una misión.

Tablero, **Sprite** e **Hilo** – Estas tres clases se encargan de dibujar en pantalla el tablero de juego. Tablero implementa *SurfaceView* y es la clase maestra. Cada uno de los elementos que haya en el nivel está dentro de un objeto Sprite, que se encargan de dibujarlo, y la clase Hilo trabaja en segundo plano y recuerda al tablero cada pocos milisegundos que redibuje su contenido.

ElementoModificado y **ListaModificadaAdapter** – Son utilizados para crear listas de forma personalizada. Más en la sección "Interfaz".

ParserJSON – Esta clase es la encargada de leer el archivo JSON y extraer la información pedida de él.

También se utilizan otros recursos, recogidos en la carpeta "res":

drawables – Las imágenes guardadas en esta carpeta se utilizan para formar la parte gráfica de la aplicación. Son las imágenes y archivos xml que se utilizan para dibujar las personas, animales, objetos y paisajes del juego, así como los personajes que hablen en IntroduccionActivity. También hay imágenes para los botones y listas personalizadas.

layout – Aquí se guardan los *layouts* de las actividades.

values – En esta carpeta se usa principalmente "strings.xml". Aquí están las diferentes cadenas que se utilizan en el juego y los apuntes, declarados para diferentes lenguajes de programación.

xml – Aquí se guardan los archivos que describen las teclas que hay en el teclado.

raw – En esta carpeta se guardan los ficheros JSON que describen los niveles del juego.

Todos estos archivos trabajan conjuntamente para formar el videojuego, interaccionando entre ellas. A continuación se muestra un diagrama de clases [27]:

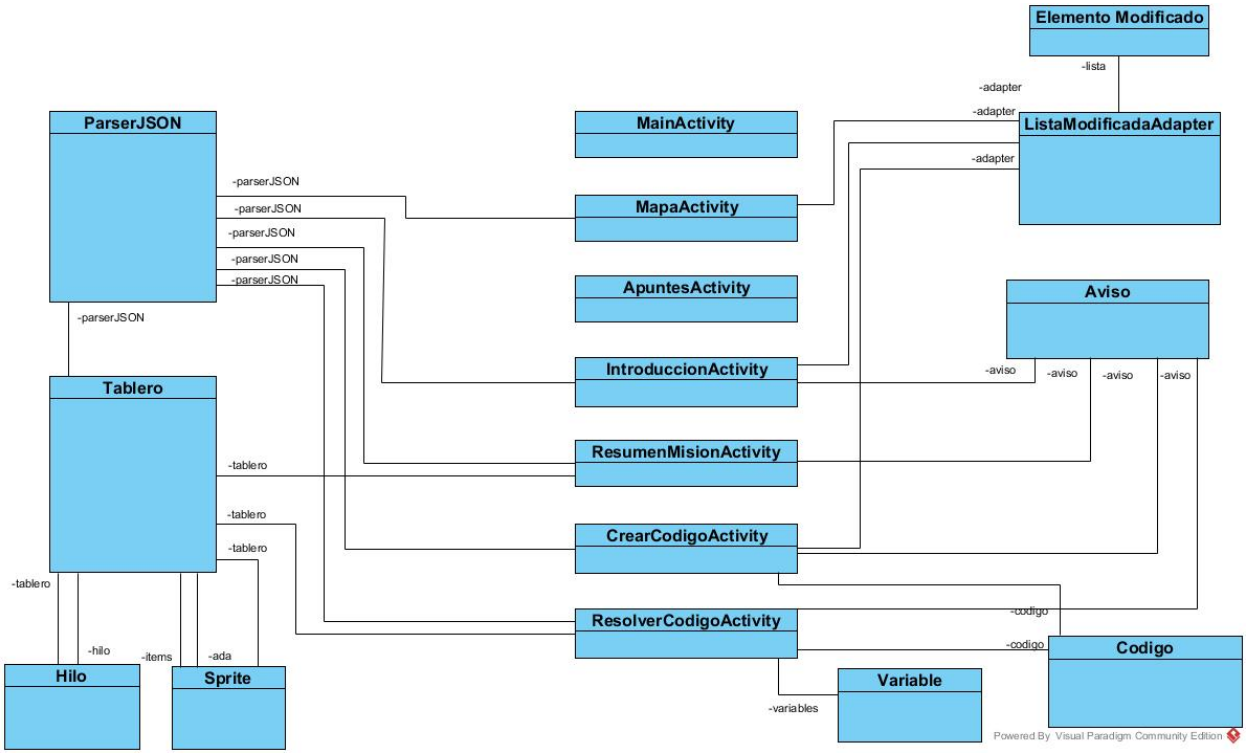

Ilustración 16 – Diagrama de clases

5.1.1 Archivos JSON

Para entender el funcionamiento de las actividades (que se verán en los puntos posteriores) hay que mencionar con mayor extensión el funcionamiento de los ficheros JSON, guardados en la carpeta "raw" de los recursos del sistema.

Hay dos tipos: los del mapa, que listan los niveles que hay, y los de los niveles, que ponen qué hay en cada uno de ellos. El archivo JSON del mapa tiene una estructura similar a esta:

```
{
    "ciudad_oeste": "pueblo_inicio",
    "exp_oeste": "0",
    "ciudad_sur": "torre",
    "exp_sur": "40",


    "nombre": "Camino en el bosque",

    "miniatura": "bosque",

    "niveles": [
       {
            "nombre": "Asalta caminos",
            "id": "combate1",
            "experiencia": "30",
            "descripcion": "¡Hay bestias en  el bosque!
Derrótalas usando bucles.",
            "lenguajes": ["Java","C"]
       }
    ]

}
```

Código 13 – bosque.json

En este archivo se recoge toda la información necesaria para dibujar el mapa. Se puede ver que indica qué ciudades hay a los lados (si no aparecen quiere decir que no hay nada en esa dirección), la experiencia necesaria para alcanzarlas, la miniatura que se dibujará y la lista de los niveles existentes con información acerca de ellos.

El archivo JSON de un nivel en concreto consiste en un *array* con las diferentes etapas que hay en el nivel. Indican el objetivo del nivel, qué se dice en el diálogo, cuál es el mapa y otros datos necesarios para configurar el nivel. Esta es la estructura de estos archivos JSON, con los elementos más importantes:

```
[
    {
        "teclas": [
            {
                "teclado": "1",
                "teclas": ["14"]
            }
        ],
        "fondo": "fondo_torre",
        "mensajes": [
            {
                "imagen": "monje",
                "posicion": "derecha",
                "texto": "¡Hola, mundo!",
            }
        ],
        "avatar": "monje_avatar",
        "resumen": "Resumen de la misión”,
        "objetivo": "objetivo primario",
        "secundario": "objetivo secundario",
        "victoria": "meta",
        "lineas": [ ["401" , "303"], ["401"], ["0"] ],
        "obligatorio": [ ["401"] ],
        "posicion_inicial": "1",
        "mapa": {
            "filas": "2",
            "columnas": "3",
            "baldosaBase": "baldosa_hierba",
            "baldosaSuelo": [
                {
                    "posicion": "5",
                    "bitmap": "baldosa_agua"
                }
            ],
            "item": [
                {
                    "aleatorio": "true",
                    "posicion": ["1","2","3"],
                    "tipo": "meta",
                    "bitmap": "meta"
                },
                {
                    "posicion": "3",
                    "direccion": "derecha",
                    "tipo": "ada",
                    "bitmap": "sprite_ada"
                }
            ]
        },
        "ayuda": ["interfaz"],
        "ayuda_extra": ["teclado"],
        "experiencia": "5"
    }
]
```

Código 14 - nivel1.json

Archivos similares a este describen los diferentes niveles que componen el juego. Cada uno de los elementos tiene una función concreta para diseñar el nivel. Algunos de estos son:

- Las teclas que se desbloquean se encuentran en el *array* "teclas".
- "fondo" y "mensajes" se utilizan en IntroduccionActivity. "fondo" contiene el nombre del *drawable* con el fondo que se mostrará, y "mensajes" es un *array* con todos los mensajes que se intercambian, con todo lo necesario para mostrarlos por pantalla.
- En "lineas" están, en formato de código, las líneas que se escribirán como muestra.
- "obligatorio" es un *array* con los elementos que hay que utilizar si se quiere completar el nivel. Si las líneas que siguen no se ejecutan en el código, el juego no permite la victoria.
- La parte de teoría que se incluirá en cada nivel está recogida en "ayuda" y "ayuda_extra". En la primera se indica el nombre de los *strings* que se utilizarán al principio de cada nivel para ayudar a resolverlo. Los apuntes de "ayuda_extra", sin embargo, se desbloquearán al ganar el nivel y únicamente se mostrarán en el libro de apuntes.
- En "mapa" se describe el tablero de juego. Este será de tantas filas y columnas como se indiquen, con una baldosa del tipo "baldosaBase" en cada posición, excepto aquellas en las que se especique otro tipo de baldosa en "baldosaSuelo".

  En el array "items" se listan todos los elementos interactuables que habrá en el tablero (objetos, enemigos, Ada, la meta…) con los atributos necesarios para crearlos.

  La posición de los elementos es un número que identifica da la baldosa sobre la que estarán. Se empieza a numerar en la esquina superior izquierda y se va hacia la derecha y hacia abajo.

  Algunos elementos serán posicionados en una posición aleatoria de las incluídas en un *array*. En este sentido habrá dos tipos de ítems que pueden estar en el mapa: los aleatorios y los determininstas. Desde el parserJSON se cogerá una posición aleatoria de todas las disponibles y se guardará en una tabla para no dibujar dos ítems en una misma posición.

  Aquí se ve un ejemplo de los dos tipos: Ada estará siempre en la posición 3, mientras que la meta puede estar en tres posiciones distintas.

5.2 Interfaz

En esta sección se va a hablar de cómo se ha implementado la interfaz de usuario. Se dará una visión general de la interfaz (centrada en IntroducciónActivity pero extrapolable a las demás actividades) y se mostrarán los elementos más llamativos que se hayan desarrollado y sean comunes para toda la aplicación. Otros elementos serán explicados en otra sección posterior (la parte gráfica del videojuego, por ejemplo, tiene una sección propia).

La interfaz de juego se crea de la forma usual en Android: una actividad implementa la lógica y utiliza un *layout* para representar los elementos por pantalla. En esta imagen se muestra la ventana del editor de Android Studio para la actividad IntroduccionActivity:

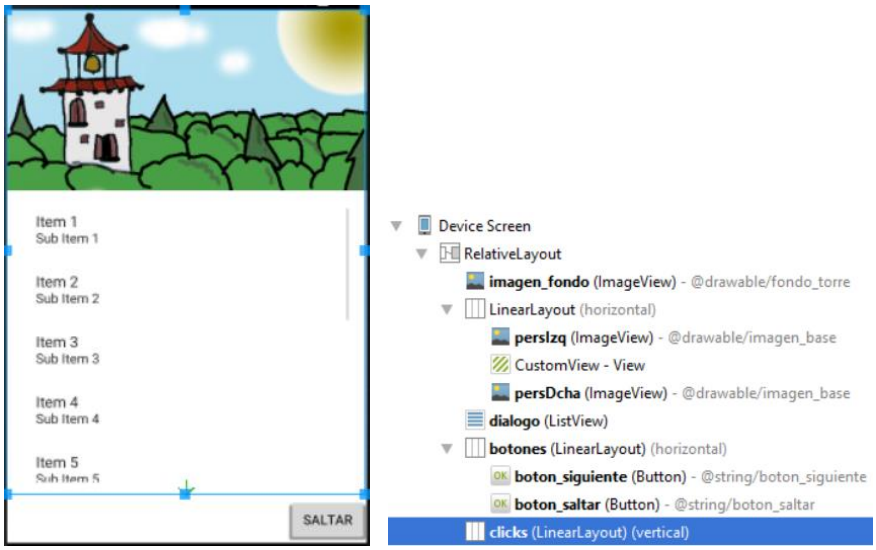

Ilustración 17 – Layout de IntroduccionActivity

Jugando con los *layouts* relativos y lineales se puede crear la interfaz adecuada. Algunas consideraciones:

- "boton_siguiente" inicialmente está oculto. Aparece cuando acaba el diálogo y al pulsarlo se inicia la siguiente actividad del nivel. "botón_saltar" tiene la misma función, pero está visible desde el principio (salvo si es la primera vez que se ha entrado en este nivel concreto, en cuyo caso se oculta).
- La cabecera de la actividad está formada por una imagen con el fondo y un *LinearLayout* superpuesto. En este *layout* hay tres elementos dispuestos horizontalmente: las imágenes de los personajes del nivel y otra vista vacía para crear un espacio vacío entre ellos.
- Se muestra un nuevo mensaje cada vez que el usuario pulsa la pantalla. "clicks" es una vista que ocupa toda la interfaz (excepto los botones, si están) y absorbe las pulsaciones. En código se le adjunta un OnClickListener que añade un nuevo mensaje a la lista.

Si es la primera vez que se entra en esta actividad, sin embargo, el comportamiento es diferente. Con la intención de servir de guía visual al usuario, se añade una imagen *9.path* al *layout*:

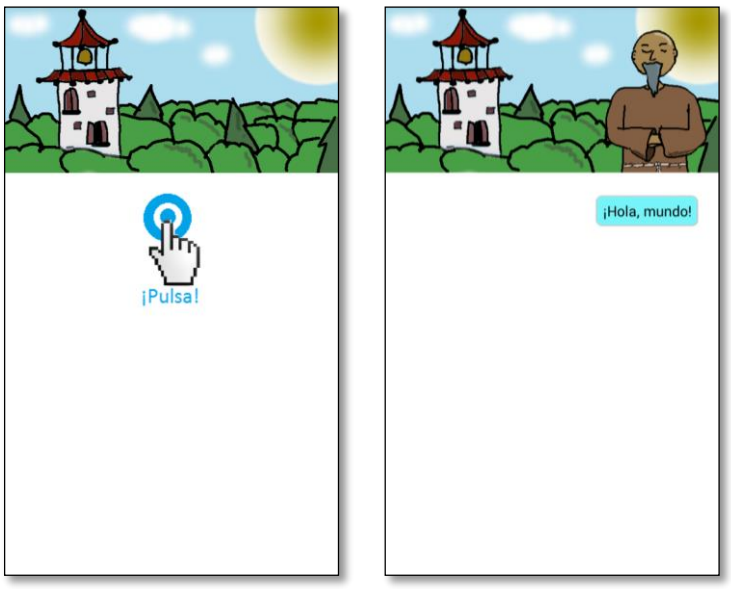

Ilustración 18 – Primer acceso a IntroduccionActivity

```
//Si es la primera vez que se entra, se pone la imagen de ayuda
if(!sharedPref.getBoolean("control_pulsar_chat", false)){
    //Se guarda para que no vuelva a salir
    editor.putBoolean("control_pulsar_chat",true);
    //Se pone la imagen de fondo y
    findViewById(R.id.clicks).setBackgroundResource(R.drawable.pulsa);
    //un listener. Al pulsar se quita la imagen, se muestra un mensaje y se pone el listener normal
    findViewById(R.id.clicks).setOnClickListener(new View.OnClickListener(){
        @Override
        public void onClick(View v){
            findViewById(R.id.clicks).setBackgroundResource(0);
            findViewById(R.id.clicks).setOnClickListener(pulsador);
            getMensaje();
        }
    });
}
```

Código 15 – Primer acceso (Fragmento de IntroduccionActivity I)

Si es la primera vez que se accede (y eso se sabe gracias a las preferencias compartidas), se añade la imagen "pulsa.9.png" como fondo del layout, que se expandirá automáticamente y quedará centrada. Cuando se pulsa, se quita la imagen, se muestra un mensaje y se vuelve a poner el Listener normal.

Hay otras ocasiones en que la interfaz tiene un comportamiento distinto la primera vez que se dan una serie de condiciones. El objetivo de esto es crear una interfaz más amigable para el usuario.

- ResolverCodigoActivity se asemeja a IntroduccionActivity en el sentido de que hay que pulsar la pantalla para que avance la ejecución. También muestra una imagen con las letras "Pulsa" la primera vez que se ejecuta.
- Cuando hay nuevas adiciones en el libro de apuntes aparece un "Toast" y, la primera vez, se muestra una flecha apuntando al libro.
- Cuando se falla por primera vez al ejecutar el código, una flecha aparece apuntando la tecla "Atrás".
- La primera vez que se inicia el juego, aparece la misión "Tutorial" automáticamente.

5.2.1 Lista

Un elemento interesante de la interfaz de IntroduccionActivity es la lista [19]. Una *ListView* es una clase de Android que muestra elementos por pantalla de forma ordenada, cada uno en una celda propia. En este caso se ha hecho de forma personalizada, para poder añadirle un globo de texto y poder orientarlos a la izquierda o a la derecha.

Para ello son necesarias dos clases: ElementoModificado y ListaModificadaAdapter. También será necesario un *layout* para cada uno de los tipos de celdas distintos y algunas imágenes *9.path*.

ElementoModificado tiene información de cada uno de los elementos que componen la lista. Se trata de una clase sencilla, formada por dos atributos: una cadena, con el texto que se mostrará, y una bandera booleana.

La *ListView* estará formada por una lista de ElementoModificado. ListaModificadaAdapter es la clase encargada de coger cada uno de los miembros de esa lista y mostrarlos por pantalla de la forma adecuada. Para ello necesita un *layout* con el que dibujar una fila de la lista.

En la aplicación hay tres lugares que utilizan estas clases para crear listas personalizadas en las que se permita alternar entre dos tipos de filas diferentes. Cada implementación es distinta. Podrían haberse creado tres adaptadores diferentes, pero se han agrupado en una misma clase por comodidad.

- En IntroduccionActivity, cada globo de texto puede tener un tipo de fondo diferente y estar alineado a la derecha o a la izquierda.
- En MapaActivity, se utiliza la lista para dar fondo a las misiones. Las misiones nuevas tendrán la bandera activada y el fondo dibujado será diferente.
- Y en CrearCodigoActivity, la línea de código que se esté editando tendrá un fondo amarillo (y la bandera será true), mientras que el resto serán blancas. Además, se escribirá el número de línea a la izquierda.

El tipo de lista se identifica gracias al layout proporcionado. En el caso que nos ocupa, este layout es "R.layout.lista_chat":

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/contenedor"
    >

    <TextView
        android:id="@+id/texto"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="5dip"
        android:textColor="@android:color/primary_text_light" />

</LinearLayout>
```

Código 16 – R.layout.lista_chat

Hay otros dos más: "R.layout.linea_de_codigo" y "R.layout.boton_mision". Para el funcionamiento del adaptador, el *layout* raíz debe tener como identificador "contenedor", y debe tener un *TextView* hijo con el identificador "texto".

ListaModificadaAdapter es la clase encargada de almacenar la lista de ElementoModificado, leer la bandera y, en función de su estado, dibujar la fila de la forma correspondiente.

Dado que hay tres tipos de filas distintos, el adaptador deberá tener tres comportamientos distintos en el método que dibuja las filas. En el caso de los globos de texto, el código es así:

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {

    View fila = convertView;

    if (convertView == null) {
        //Se añade una nueva view a la lista.
        LayoutInflater inflater = (LayoutInflater) this.getContext()
                .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        fila = inflater.inflate(resourceID, parent, false);
    }

    ElementoModificado l = lista.get(position);
    ((TextView) fila.findViewById(R.id.texto)).setText(l.getTexto());

    if (resourceID == R.layout.lista_chat){

        //Se elige el fondo y la posición según si está a la derecha o la izquierda
        fila.findViewById(R.id.texto).setBackgroundResource(l.isFlagged() ?
                R.drawable.globo_izq : R.drawable.globo_dcha);
        LinearLayout contenedor = (LinearLayout) fila.findViewById(R.id.contenedor);
        contenedor.setGravity(l.isFlagged()? Gravity.START : Gravity.END);
        contenedor.setPadding(l.isFlagged()? 0 : 40, 0, l.isFlagged() ? 40: 0,0);
    }
    //[...]
    return fila;
}
```

Código 17 – getView (Fragmento de ListaModificadaAdapter)

En este caso, el layout con el que se ha creado la lista es "R.layout.lista_chat". El adaptador escribe el texto y crea la celda de la forma correcta: añade el fondo adecuado y orienta el globo hacia la izquierda o la derecha.

Para utilizar la lista, desde la actividad hay que crearla, indicando el layout a partir del cual se generará la vista (y que se usará para identificar qué tipo de lista es):

```
ListView lista = (ListView) findViewById(R.id.dialogo);
ListaModificadaAdapter mensajeAdapter = new ListaModificadaAdapter(
        getApplicationContext(), R.layout.lista_chat);
lista.setAdapter(mensajeAdapter);
```

Código 18 – Creación de la lista (Fragmento de IntroduccionActivity II)

Luego, se utiliza el adaptador para añadir mensajes a la lista. Cada vez que se pulsa la pantalla, se coge el mensaje correspondiente (guardado en "numMensaje") y se muestra:

```
int resID = parserJSON.getPersona(numMensaje);

boolean posicion = parserJSON.getPosicion(numMensaje);
ImageView imagen = posicion ? imagenPersIzq : imagenPersDcha;

imagen.setBackgroundResource(resID);
AnimationDrawable animacion = (AnimationDrawable) imagen.getBackground();
animacion.start();
if (!parserJSON.getMensaje(numMensaje).equals("")) {

    mensajeAdapter.add(new ElementoModificado(posicion,
            parserJSON.getMensaje(numMensaje)));
    mensajeAdapter.notifyDataSetChanged();
}

numMensaje++;
```

Código 19 – Mostrar mensaje (Fragmento de IntroduccionActivity III)

Nótese cómo, al crear el ElementoModificado, se añaden el texto del mensaje y el *flag* indicando si está a la izquierda o a la derecha. A continuación se notifica al adaptador que hay elementos nuevos, lo que fuerza la actualización de la lista (a no ser que el mensaje esté vacío y sólo se quería mostrar la animación).

5.2.2 Aviso

Cuando se pulsa el botón de "atrás" del terminal, se destruye la actividad actual y se vuelve a iniciar la actividad superior en la pila de actividades. Si el jugador está escribiendo código, al ir hacia atrás lo perderá. Por ello se ha implementado un *Dialog*: una ventana emergente que aparace por encima de la actividad y solicita una respuesta por parte del usuario. Para poder salir el usuario necesita confirmar que no ha pulsado el botón por error.

Ilustración 19 – Confirmación para salir

Este *Dialog* se ha desarrollado creando una clase que extienda *DialogFragment*, a la que se ha llamado Aviso. Sólo necesita un método, "onCreateDialog", donde se crea y se especifica qué ocurre en cada una de las dos opciones.

Esta clase es usada en las cuatro actividades que componen la misión y salir podría producir pérdida de información: IntroduccionActivity, ResumenMisionAcivity, CrearCodigoActivity y ResolverCodigoActivity.

Al ser usada en más de una ocasión, se ha decidido implementar el código en una clase propia. También podría haberse definido como una subclase, como ocurre en MapaActivity. En esta actividad se puede pulsar un botón que reinicia el juego y, para evitar accidentes, al pulsarlo se solicita confirmación.

5.3 Código

La escritura y ejecución de código es la parte más importante de esta aplicación. El núcleo está en la clase Codigo: un objeto de este tipo representará a una línea de código escrita por el jugador.

Antes de continuar conviene decir que los diferentes elementos de código que se pueden escribir están identificados por un entero. Estos enteros están recogidos en la clase Codigo como una serie de

constantes:

- Del número 301 a 400 se utiliza para constantes dentro del código (arriba, derecha, enemigo etc.). Esto no quiere decir que haya cien constantes y se utilicen todos los números disponibles; se ha dividido así para hacer más intuitiva la separación y poder añadir más elementos en caso de que sea necesario.
- De 401 a 500 están los métodos que puede usar Ada (andar, mirar, escuchar…).
- De 501 a 550 están las estructuras de control, y de 551 a 600 sus respectivos cierres.
- De 601 a 700 se asignan los comparadores (igual, menor que…).
- Y, a partir de 1000, se asignan variables.

Toda la información de lo que se ha escrito en una línea de código está guardada en los atributos de la clase:

```
private int codigo;
private ArrayList<Integer> params;
private int nTab;
private ArrayList<String> cadena;
private String lenguaje;
```

Código 20 – Atributos (Fragmento de Codigo)

"codigo" guarda un entero con el elemento principal. Es el que se usa para identificar qué se hace en esa línea (se inicia un bucle, se realiza una acción, etc).

"params" es una lista de todo lo que se ha escrito en la línea, en orden (incluye el código principal).

"cadena" es una lista hermana de "params", en el sentido de que a cada elemento en la lista "params" le corresponde uno en la lista "cadena". Se utiliza porque algunos parámetros necesitan una cadena auxiliar (una variable *int* se identifica con el número 1001, pero necesita una cadena con su nombre).

"nTab" guarda el número de tabulaciones.

Y "lenguaje" es el lenguaje en el que se está jugando.

Sobre los métodos, cabe destacar "escribirLinea". Este método convierte en caracteres, en la sintaxis correcta, la línea escrita. Lo hace en tres pasos:

1) Añade las tabulaciones necesarias.
2) Recorre la lista de parámetros. Convierte a una cadena de caracteres cada elemento (de forma diferente según el lenguaje) y añade algo más si es necesario (un cierre de paréntesis si es un método, por ejemplo).
3) Termina de escribir la línea, normalmente añadiendo un punto y coma o la apertura de unas llaves.

Utilizando este método se mostrará por pantalla qué lleva escrito el jugador.

El código que el jugador tiene que crear será una lista de objetos de esta clase, que después se ejecutará. Se hace en dos actividades distintas: CrearCodigoActivity y ResolverCodigoActivity.

5.3.1 CrearCodigoActivity

Esta actividad es en la que se realiza la escritura del código. Estas son algunas de los atributos que utiliza para ello:

```
private int linea_actual;
private ArrayList<Codigo> codigo;              //Cada elemento es una línea de código
private ArrayList<String> variables;           //Lista con los nombres de las variables creadas
private String[] constantes;                   //Constantes
private int num_constantes;                    //Número de constantes desblqueadas
private String[] tipos_variables;              //Tipo de variables
private int num_variables;                     //Número de tipos desbloqueados
private String lenguaje;                       //Lenguaje del código
```

Código 21 – Atributos (Fragmento de CrearCodigoActivity I)

Hay más atributos, pero estos son los que están centrados en el desarrollo del código.

El código se escribe utilizando el teclado. Se han desarrollado tres teclados principales, distribuyendo el posible código que se puede escribir, y uno con letras para dar nombre a las variables creadas:

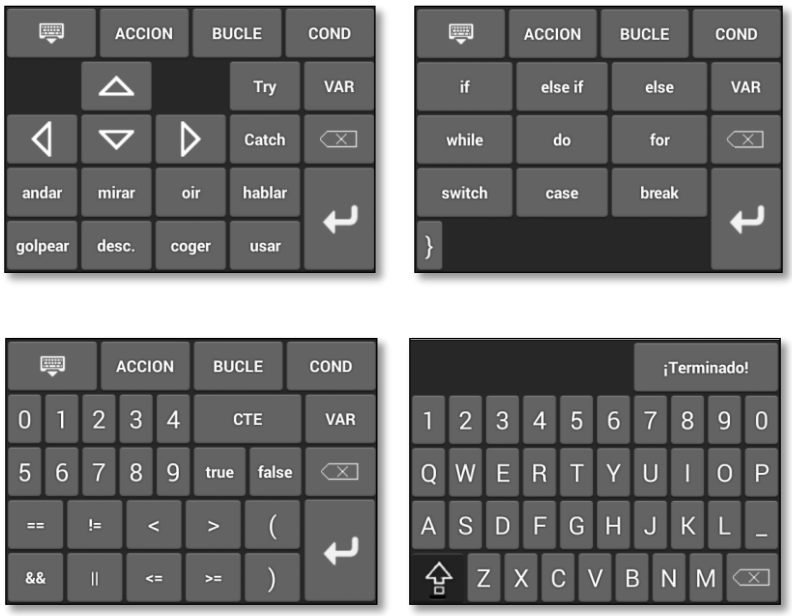

Ilustración 20 - Teclados

Para implementar el teclado se hace uso de las clases de Android *Keyboard* y *KeyboardView* [20] [21]. En el layout de la actividad se introduce la vista del teclado:

```
<android.inputmethodservice.KeyboardView
    android:id="@+id/teclado"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:focusable="true"
    android:focusableInTouchMode="true"
    android:visibility="invisible"
    android:keyBackground="@drawable/btn_keyboard_key_ics"
    android:layout_gravity="bottom"/>
```

Código 22 – KeyboardView en activity_crear_codigo.xml

El atributo más relevante para la representación del teclado es “android:keyBackground”. A este atributo se le proporciona un fichero XML con el estilo visual que tendrán las teclas [22] [23].

El fichero “btn_keyboard_key_ics”, guardado en la carpeta “drawable” es el siguiente. Lista, de forma ordenada, el fondo que se debe utilizar para los diferentes tipos y estados de teclas:

```
<selector xmlns:android="http://schemas.android.com/apk/res/android">

    <!-- Toggle keys. Use checkable/checked state. -->
    <item android:state_checkable="true" android:state_checked="true" android:state_pressed="true"
          android:drawable="@drawable/btn_keyboard_key_pressed_on_ics_dark" />
    <item android:state_checkable="true" android:state_pressed="true"
          android:drawable="@drawable/btn_keyboard_key_pressed_off_ics_dark" />
    <item android:state_checkable="true" android:state_checked="true"
          android:drawable="@drawable/btn_keyboard_key_normal_on_ics_dark" />
    <item android:state_checkable="true"
          android:drawable="@drawable/btn_keyboard_key_normal_off_holo_dark" />

    <!-- Empty background keys. -->
    <item android:state_empty="true"
          android:drawable="@android:color/transparent" />

    <!-- Normal keys. -->
    <item android:state_pressed="true"
          android:drawable="@drawable/btn_keyboard_key_pressed_ics_light" />
    <item android:drawable="@drawable/btn_keyboard_key_normal_holo_light" />

</selector>
```

Código 23 – btn_keyboard_key_ics.xml

Para las teclas normales se usará la imagen "btn_keyboard_key_normal_holo_light", a no ser que esté presionada, que se utilizará "btn_keyboard_key_pressed_ics_light". Para las teclas con dos posiciones (como la tecla *shift*) se utilizan cuatro fondos distintos, según el estado: sin estar activa ni pulsada, activa, pulsada y pulsada y activa.

Este es únicamente el contenedor del teclado. Ahora, desde el código, hay que añadir las teclas de la distribución correspondiente:

```
final Keyboard mKeyboard = new Keyboard(this,R.xml.teclado1);
teclado = (KeyboardView)findViewById(R.id.teclado);
teclado.setKeyboard(mKeyboard);
teclado.setPreviewEnabled(false);
teclado.setOnKeyboardActionListener(tecladoActionListener);
ocultarTeclas(1);
```

Código 24 – Añadir teclado (Fragmento de CrearCodigoActivity II)

El método "ocultarTeclas", como su propio nombre indica, oculta las teclas que aún no están desbloqueadas. Para ello recorre las teclas una a una y comprueba en las SharedPreferences si está desbloqueada. Si la tecla en concreto no está, se hace desaparecer:

```
for (int i = 0; i <= maxTeclas; i++)
    if(!sharedPref.getBoolean("tecla_" + teclado + "_" + i, false)) {
        mKeyboard.getKeys().get(i).label = "";
        mKeyboard.getKeys().get(i).height = 0;
    }
```

Código 25 – ocultarTeclas (Fragmento de CrearCodigoActivity III)

Para diseñar la distribución del teclado se utiliza un archivo xml. Hay varios, uno para cada uno de los teclados mostrados anteriormente. En el caso de "teclado1":

```
<?xml version="1.0" encoding="utf-8"?>
<Keyboard xmlns:android="http://schemas.android.com/apk/res/android"
    android:keyWidth="20%p"
    android:keyHeight="8%p" >

    <Row android:keyWidth="25%p">
        <Key android:codes="-3"   android:keyIcon="@drawable/tecla_ocultar"
            android:keyEdgeFlags="left" />
        <Key android:codes="-10"    android:keyLabel="ACCION" />
        <Key android:codes="-11"    android:keyLabel="BUCLE" />
        <Key android:codes="-12"    android:keyLabel="COND" android:keyEdgeFlags="right"/>
    </Row>

    <Row>
        <Key android:codes="304" android:keyIcon="@drawable/flecha_arriba"
            android:horizontalGap="20%p" android:keyEdgeFlags="left"/>
        <Key android:codes="520" android:keyLabel="Try" android:horizontalGap="20%p"/>
        <Key android:codes="998" android:keyLabel="VAR" android:keyEdgeFlags="right"/>
    </Row>

    <!--[...]-->
</Keyboard>
```

Código 26 – teclado1.xml

Como se ve, el teclado está dividido en filas, y cada una de ellas se divide en teclas. Hay varios atributos que conviene conocer:

- "keyWidth" y "keyHeight" determinan la altura y anchura de las teclas. Siempre se usará el más cercano a la tecla. Si el atributo está en la tecla, se usará ese. Si no lo tiene, se usará el de la fila y, si tampoco hay, se usará el del teclado.
- "codes" es el código que se enviará al Listener cuando se pulse esa tecla. Corresponde con el códido adecuado de la clase Codigo.
- "keyLabes" y "keyIcon" contienen el texto o la imagen, respectivamente, que se mostrará en la tecla.
- "keyEdgeFlags" son atributos de las teclas a la izquierda y derecha de la fila, utilizados para ayudar en la representación.
- "horizontalGap" introduce un espacio vacío antes de la tecla.

El último componente del teclado es un *Listener* que se activará cada vez que se pulse una tecla. El *Listener* recibe el código de la tecla, en el primer atributo del método "onKey", y otra lista de códigos secundarios de la tecla que no se utiliza en este proyecto.

Es en este paso donde hay que implementar la escritura del código. El teclado no escribirá nada por defecto, únicamente hará lo que se implemente en el *Listener*. Por cada tecla este actuará de una forma distinta. Algunas de las teclas son sencillas: por ejemplo, las teclas de la primera fila cambian de teclado:

```
private KeyboardView.OnKeyboardActionListener tecladoActionListener =
        new KeyboardView.OnKeyboardActionListener() {

    @Override
    public void onKey(int primaryCode, final int[] keyCodes) {
        Codigo c = codigo.get(linea_actual);
        //Ocultar teclado
        if (primaryCode == -3 ) {
            teclado.setVisibility(View.GONE);

        //Tres teclas cambian el teclado visible
        } else if (primaryCode == -10) {
            teclado.setKeyboard(new Keyboard(getApplicationContext(), R.xml.teclado1));
            ocultarTeclas(1);
        } else if (primaryCode == -11) {
            teclado.setKeyboard(new Keyboard(getApplicationContext(), R.xml.teclado2));
            ocultarTeclas(2);
        } else if (primaryCode == -12) {
            teclado.setKeyboard(new Keyboard(getApplicationContext(), R.xml.teclado3));
            ocultarTeclas(3);
        }
```

Código 27 – Cambiar teclado (tecladoActionListener I)

Otros se van complicando poco a poco. La tecla "Enter" crea una nueva línea de código justo después de la línea actual, y cambia la línea seleccionada a la nueva:

```
} else if (primaryCode == -1) {

    adapter.setFlag(linea_actual, false);
    linea_actual++;
    adapter.add(linea_actual, " ");
    adapter.setFlag(linea_actual, true);

    codigo.add(linea_actual, new Codigo(lenguaje, 0));

    //Si la línea anterior era el inicio de un bucle, se aumenta la tabulación de la siguiente
    if (c.getCodigo() >= Codigo.IF && c.getCodigo() <= Codigo.FIN_BUCLE_INICIO)
        codigo.get(linea_actual).setTab(c.getTab() + 1);
    else
        codigo.get(linea_actual).setTab(c.getTab());
}
```

Código 28 – Enter (tecladoActionListener II)

Primero modifica las dos líneas utilizando el adaptador. Se quita el *flag* de la línea antigua y lo activa en la nueva (se recuerda que, en este tipo de lista, el *flag* era *true* en la fila que se está modificando actualmente y tiene un fondo amarillo).

Después añade un nuevo objeto Codigo a la lista y, para terminar, le escribe el número de tabulaciones correctas (las mismas que la línea anterior, a no ser que ésta sea el inicio de un bucle, en cuyo caso aumenta en un valor).

La mayoría de las teclas están agrupadas por bloques. El *Listener* realiza lo mismo para la mayoría de los inicios de estructuras de control. Por ejemplo, si se escribe el inicio de un bucle:

```
}else if (primaryCode < Codigo.FIN_BUCLE_INICIO) {
    if (c.getCodigo() == Codigo.CIERRE) {
        adapter.setFlag(linea_actual, false);
        linea_actual++;
        adapter.add(linea_actual, " ");
        adapter.setFlag(linea_actual, true);
        codigo.add(linea_actual, new Codigo(lenguaje, 0));
        c = codigo.get(linea_actual);
        c.setTab(codigo.get(linea_actual - 1).getTab());
    }
    c.setCodigo(primaryCode);

    codigo.add(linea_actual + 1, new Codigo(lenguaje, Codigo.CIERRE));
    codigo.get(linea_actual + 1).addParam(primaryCode + 50);
    codigo.get(linea_actual + 1).setTab(c.getTab());
    adapter.add(linea_actual + 1, codigo.get(linea_actual + 1).escribirLinea());
}
```

Código 29 – Iniciar bucle o condicional (tecladoActionListener III)

Obviando el *if* inicial, lo que hace es escribir el código de la tecla que se ha pulsado en la línea actual y añadir una nueva línea, que contendrá el cierre del bloque.

En el *if* inicial lo que se hace es, si el usuario ha pulsado la tecla encima del cierre de otro bucle, añadir una nueva línea y escribir el nuevo inicio de bucle en esta nueva línea.

Esto se hace así porque, por cómo se ha desarrollado el proyecto, los cierres de llaves y los inicios de nuevos bucles han de estar en líneas separadas. Es decir, no puede se puede escribir "} else {" en una misma línea. Como esto puede resultar confuso para los jugadores, que tendrán la intención de iniciar el siguiente elemento del bloque en el cierre, se ha tenido en cuenta y se realiza automáticamente.

Para las teclas de acciones se hace algo similar. Sin embargo, también hay que tener consideraciones particulares:

```
}else if (primaryCode < Codigo.FIN_ACCIONES) {
    if (c.getCodigo() == 0)
        c.setCodigo(primaryCode);

    else if (c.getCodigo() < Codigo.FIN_ACCIONES) {
        adapter.setFlag(linea_actual, false);
        linea_actual++;
        adapter.add(linea_actual, " ");
        adapter.setFlag(linea_actual, true);
        codigo.add(linea_actual, new Codigo(lenguaje, primaryCode));
        codigo.get(linea_actual).setTab(c.getTab());
    } else if (c.getCodigo() != Codigo.CIERRE || c.getParam(1) == Codigo.DO + 50)
        c.addParam(primaryCode);
    else
        Toast.makeText(CrearCodigoActivity.this,
                "¡No puedes escribir ahí!", Toast.LENGTH_SHORT).show();
}
```

Código 30 – Acciones (tecladoActionListener IV)

Añade el código de la tecla de tres formas distintas, dependiendo de qué haya escrito previamente en la línea.

- Si la línea estaba vacía, el código se añade como código principal de la línea.
- Si había otra acción, se crea una nueva línea y se añade el código a la nueva. Esto se ha tenido en cuenta porque varios usuarios que probaron la aplicación olvidaban pulsar "Enter" cuando había que escribir varios métodos seguidos. Se espera que con esta modificación sea más fácil.
- En otros casos, se añade como parámetro (dentro de una condición, por ejemplo).

- Y, si no se puede escribir ahí (porque es el cierre de un bucle o el inicio de un do, que no admiten parámetros), se lanza un *Toast*.

Algunas teclas de por sí no aportan suficiente información como para poder escribir correctamente la línea de código. En algunos casos, lo que se hace es desplegar una pequeña lista para que el usuario seleccione de ella el elemento que quiere usar. Por ejemplo, cuando se pulsa la tecla "VAR" se muestran las variables creadas:

Ilustración 21 – Popup con la lista de variables

```
} else if (primaryCode == Codigo.VARIABLE) {

    popup_texto.clear();
    popup_texto.add("Crear");
    popup_texto.addAll(variables);
    popup_texto.add("Salir");
    popup_adapter.notifyDataSetChanged();
    popup.setVisibility(View.VISIBLE);
    popup.setOnItemClickListener((parent, view, position, id) -> {
            if (position == 0) {
                onKey(Codigo.DECLARAR, keyCodes);
            } else if (position != popup_adapter.getCount() - 1) {

                //Se guarda como parámetro el nombre de la variable
                if (codigo.get(linea_actual).getCodigo() == 0) {
                    codigo.get(linea_actual).setCodigo(Codigo.ASIGNACION);
                    codigo.get(linea_actual).setCadena(popup_texto.get(position));

                } else {
                    codigo.get(linea_actual).addCadena(Codigo.VARIABLE,
                            popup_texto.get(position));
                }
            }

            if (position != 0) {
                popup.setVisibility(View.GONE);
                adapter.set(linea_actual, codigo.get(linea_actual).escribirLinea());
                lista.invalidateViews();
            }
    });
}
```

Código 31 – Mostrar un popup (tecladoActionListener V)

Una vez que esta nueva lista se muestra por pantalla, pueden pasar tres cosas:

a) El usuario le da a crear variable (la posición del elemento pulsado es 0). En ese caso, se salta a otra sección del Listener que cambia el contenido del *popup* y muestra los diferentes tipos disponibles para crear una variable.

b) Se selecciona una variable ya creada. Ahora hay que mirar la línea de código: si está vacía (es decir, el código principal es 0), se crea un código como una asignación. En la pantalla, se verá el nombre de la variable y un igual a su derecha. Si ese no es el caso, se añade la variable como parámetro. Después, se oculta el *popup*.

c) Se le da a salir (es decir, la posición pulsada es la última). En ese caso no se escribe la variable y se oculta el *popup*.

Si el usuario decide crear una variable y elige el tipo que quiera, el teclado cambiará, mostrándose un teclado más tradicional con letras y números. Cuando se pulse una de estas teclas se guardará en una cadena el nombre que se vaya escribiendo.

```
if(teclado.getKeyboard().getKeys().get(31).on && primaryCode > 95)
    s = s + Character.toString((char) (primaryCode-32));
else
    s = s + Character.toString((char) primaryCode);
c.setCadena(s);
```

Código 32 – escribrir una letra (tecladoActionListener VI)

La variable "s" es la cadena correspondiente al nombre de la variable en el objeto Codigo de la línea. La letra pulsada se añadirá al final de la cadena y se guardará.

La tecla 31 del teclado es el *shift*. Cuando esté activado (su atributo "on" sea *true*) se escribirá en mayúsculas (que, en la tabla ASCII, está 32 puestos antes que las minúsculas).

Cuando el usuario haya completado el nombre del variable deberá pulsar la tecla "Terminado". En ese caso el nombre que se haya ido escribiendo se guardará en una lista con las variables creadas, para poder mostrarla más adelante cuando el jugador quiera utilizarla.

De esta forma se irá escribiendo el código. Las teclas irán creando la lista de Codigo, y estos se irán mostrando en pantalla gracias a su método "escribirLinea". Una vez terminado, el código se enviará a la siguiente actividad:

```
Intent intent = new Intent(this, ResolverCodigoActivity.class);
intent.putExtra("CODIGO", codigo);
intent.putExtra("NIVEL", nivel);
intent.putExtra("ETAPA", etapa);
startActivity(intent);
```

Código 33 – enviar codigo (fragmento de CrearCodigoActivity IV)

5.3.2 ResolverCodigoActivity

El objetivo de esta pantalla es mostrar al jugador qué hace su código mediante una representación gráfica. Para ello se va leyendo línea a línea, una por cada vez que el jugador pulsa la pantalla, y se muestra el resultado en el tablero.

Para realizar correctamente su función, son varios los elementos que la actividad debe tener en cuenta y hacer que funcionen correctamente entre ellos:

- El código, recibido desde CrearCodigoActivity. La actividad debe saber leerlo correctamente y mostrarlo por pantalla.
- La representación gráfica. El tablero debe dibujarse continuamente y mostrar una animación. Además, muchas líneas actúan con los elementos presentes en el tablero, y deben actualizarse.

- El objetivo del nivel. Cada uno de los misiones presentará un reto distinto; la actividad debe notar cuando éste se ha cumplido (o cuando ya es imposible de realizar).

En el siguiente diagrama se detalla de forma general qué hace la actividad.

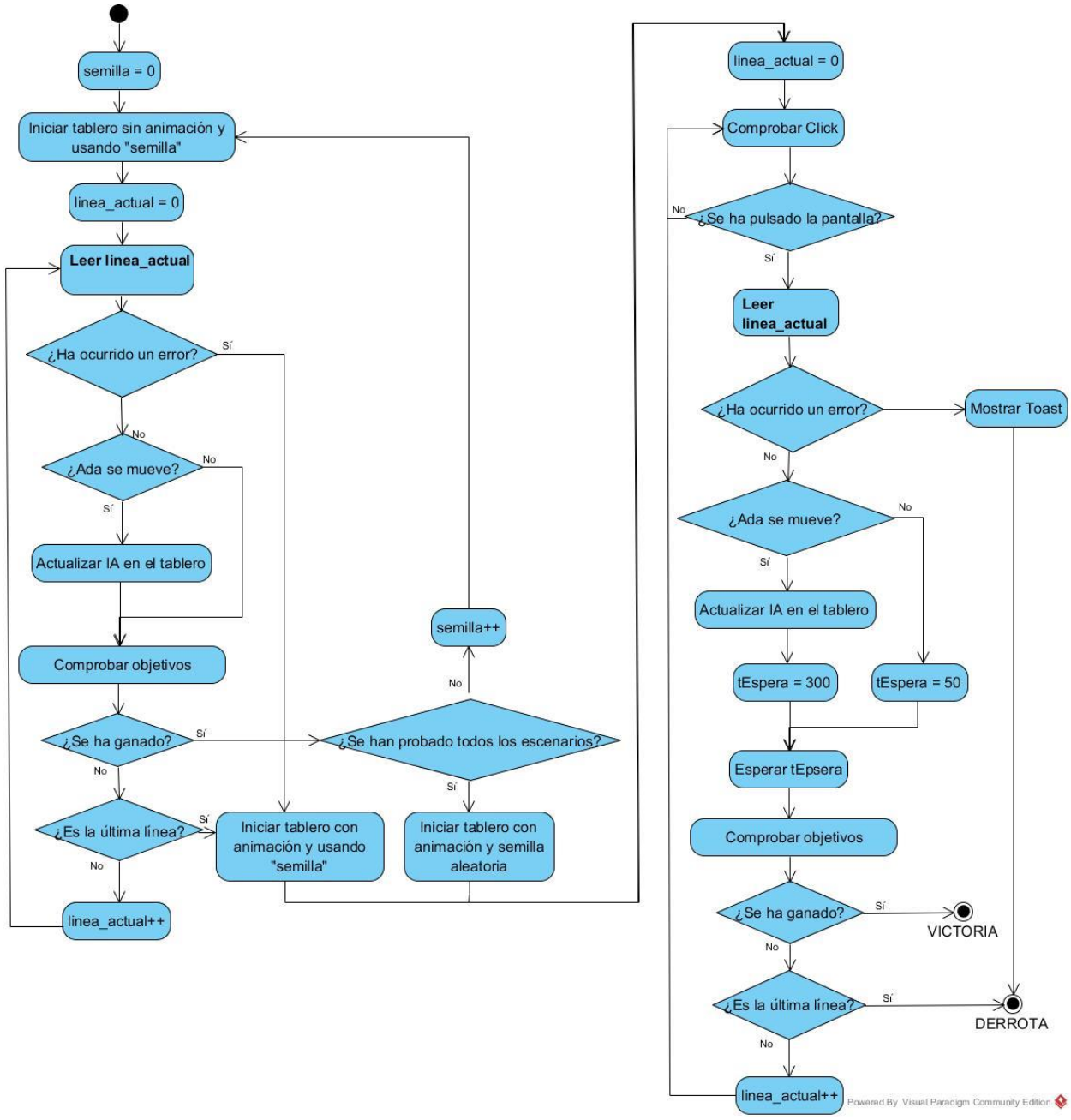

Ilustración 22 – Diagrama de Actividad de ResolverCodigoActivity

Lo primero que se hace es comprobar que el código sea válido para varias ejecuciones. Para ello se ejecutan rápidamente diferentes distribuciones del tablero con semillas distintas. En esta fase el tablero no dibuja nada, sino que el movimiento se realiza instantáneamente. Tampoco se muestran *Toasts* con errores, en caso de que los hubiera.

Se cambia la semilla hasta que se llegue al final o se falle en una ejecución. Si una distribución concreta ha dado error, esa la que se le mostrará al jugador, mientras que si ha pasado todas se utilizará una semilla aleatoria.

Una vez elegida la semilla, se mostrará por pantalla la distribución escogida. En este caso no será una ejecución automática, sino que se leerá una línea cada vez que el jugador pulse la pantalla. Cuando el jugador pulsa la pantalla, se activa este Listener:

```
final View.OnClickListener pulsador = (v) -> {
        if (!enEjecucion && !fin)
            ejecutar();
};
```

Código 34 – pulsador (Fragmento ResolverCodigoActivity I)

Únicamente se ejecuta una línea si las variables booleanas "enEjecucion" y "fin" son *false*. "fin" es *true* cuando ya se ha terminado la ejecución del nivel (se han recorrido todas las líneas o el jugador ha ganado o perdido). Es decir; si la partida ha terminado, no se permite que se contiúen ejecutando líneas de código.

"enEjecucion" es verdadera mientras se está ejecutando la línea. Se le asigna *true* cuando comienza a leerse la línea de código, y no vuelve a estar *false* hasta que termina la animación del tablero. Así se evita que se solapen varias ejecuciones.

Hay que tener en cuenta que la ejecución del tablero es independiente a la lectura del código. Desde ResolverCodigoActivity se mandan órdenes al tablero, que procederá a animarse tardando el tiempo que sea necesario. Para asegurar que no se solapan, después de leer la línea de código, ResolverCodigoActivity espera un tiempo antes de poner "enEjecucion" como *false*. Esto se consigue con estas líneas, colocadas al final del método "ejecutar":

```
if(esperar) {
    tEspera = 600;
}
else
    tEspera = 50;

handler.postDelayed(new Runnable() {
    public void run() {
        if (esperar)
            tablero.actualizarIA();

        victoria();
        enEjecucion = false;
        esperar = false;
    }
}, tEspera);
```

Código 35 – Leer línea (Fragmento ResolverCodigoActivity II)

"esperar" es una variable booleana que es *true* cuando en la línea de código se ordena algo de movimiento (andar, por ejemplo). Si hay que esperar a la animación, el *handler* no se activará hasta pasados seiscientos milisegundos. Si no hay animación, únicamente esperará cincuenta.

Dentro del handler se cambia "enEjecucion", "esperar" y se comprueba si el jugador ha ganado la partida. Además, actualiza el movimiento de los demás personajes no controlados por el jugador. Estos se mueven en el mismo turno que Ada haga un movimiento, y no en las demás. Las líneas que contengan estructuras de control no provocarán que los enemigos se lancen a por Ada.

En todo caso, una vez pulsada la pantalla se procede a la ejecución del código (el cuadrado "Leer línea_actual" del diagrama de actividad). Según el código principal de la línea se realizan unas acciones u otras. Si en el código se ha escrito que Ada ande, se ejecuta esto:

```
if (c.getCodigo() == Codigo.ANDAR) {
    esperar = true;

    //Si tiene un parámetro y es una dirección
    if (c.getNumParam() == 2 &&
            (c.getParam(1) >= Codigo.ABAJO && c.getParam(1) <= Codigo.ARRIBA)) {
        tablero.getAda().setAccion(Sprite.ANDAR, c.getParam(1) - Codigo.ABAJO);

    //O si no tiene parámetros
    } else if (c.getNumParam() == 1) {
        tablero.getAda().setAccion(Sprite.ANDAR);

    } else {
        if (!ejecucion_prev)
            Toast.makeText(this, "Error - Los parametros de \"andar\" están mal.",
                    Toast.LENGTH_LONG).show();
        fin = true;
    }
}
```

Código 36 – Leer línea (Fragmento de ResolverCodigoActivity II)

"c" es un objeto Codigo que contiene la línea actual. Se leen los parámetros que contiene, para comprobar si se ha escrito correctamente. Si hay algún fallo, se lanza un *Toast* que notifica del error al jugador.

Al leer este código, se accederá al tablero a través del método "setAccion". Esté accederá al mismo método del objeto que representará a Ada (se explicará con más detalle en las secciones siguientes).

Otras líneas de código son más complicadas, como los condicionales (*if*, *else if* o *else*). Para ejecutar estos se realizan una serie de pasos:

1) Si es un *else if* o un *else*, se comprueba que la línea anterior sea el cierre de un *if* o de un *else if*.
2) Se evalúa la condición, comparando poco a poco los parámetros de la línea. Si hay un fallo de sintaxis, se termina la partida. Si no:
    a. Si la condición es correcta, se pasa a la siguiente línea.
    b. Si es incorrecta, se salta al cierre del bloque y se avanza una línea más.
3) Si la línea contiene un cierre de llaves, significa que ese bloque de la condición se ha recorrido y, por lo tanto, se salta a la siguiente línea que no sea parte del bloque (saltando los *else if* y *else* que sean necesarios).

Cómo se evalúa una condición requiere una mayor explicación. Está implementada en el interior de un método, que recibe como parámetro el número del parámetro de inicio en que empieza la condición (si es un *if*, será en el segundo, puesto que el primero sería el código del *if*).

Para hacer la evaluación necesita estas variables:

```
boolean resultado = false;
Codigo c = codigo.get(linea_actual);
int j = paramInicio;

int[] cond = new int[2];
int indice = 0;
int comparador = 0;
```

Código 37 – evaluarCondicion (Fragmento de ResolverCodigoActivity III)

Los primeros parámetros son intuitivos: "resultado" es lo que devolverá la función, "c" es la línea de código sobre la que se trabaja y "j" es una variable para recorrer los parámetros.

Solo se ha implementado la comparación entre dos variables. No se han introducido los comandos *and* o *or*.

"cond" es una tabla de dos enteros. En ella se almacenarán las variables, números y otros elementos

que se vayan leyendo en “c”, y cada vez que se almacene uno aumentará “indice”, para saber dónde guardar el siguiente. En “comparador” se guarda el código del comparador que se usará.

Ahora, lo que se pretende hacer es leer los parámetros, almacenarlos en su variable correspondiente y, cuando se tengan dos elementos en “cond” y un comparador, se evaluará la condición. Ese momento debe coincidir con la lectura del último elemento de la línea; si hay alguno más significa que no se ha escrito correctamente.

```
while (j < c.getNumParam()) {

    if (c.getParam(j) == Codigo.VARIABLE) {
        Variable v = buscarVariable(c.getCadena(j));
        if (v != null)
            cond[indice] = v.getValor();
        indice++;
    } else if (c.getParam(j) >= Codigo.IGUAL && c.getParam(j) <= Codigo.FIN_COMP) {
        comparador = c.getParam(j);
    }
    //[...]

    //Si ya se tiene un juego completo
    if (indice == 2 && comparador != 0) {
        switch (comparador) {
            case (Codigo.IGUAL):
                resultado = (cond[0] == cond[1]);
                break;
            //[...]
        }
        indice = 0;
        comparador = 0;
    }
    j++;
}

if (indice == 1 && cond[0] == Codigo.TRUE)
    resultado = true;
else if (indice == 1 && cond[0] == Codigo.FALSE)
    resultado = false;
else if (indice != 0){
    fin = true;
    resultado = false;
}
```

Código 38 – evaluarCondicion II (Fragmento de ResolverCodigoActivity IV)

Al final, se comprueba si la lectura ha sido correcta. Si el índice no es cero (es decir, si no se ha entrado en el *if* al final del bucle, donde se reinicia) significa que ha habido un error, a no ser que sea debido a que en la condición únicamente haya un valor booleano.

Otra faceta a considerar de esta actividad es la gestión de las variables. Cada vez que se crea una variable, esta se escribe en un objeto Variable. Esta tienen un atributo “tipo” y varios atributos con el valor (“valorInt”, “valorBoolean”, etc).

Según el tipo de variable (utilizando el método “getTipo” de la clase) se debe actuar de forma distinta, y acceder a un determinado valor.

La ejecución del código continuará así hasta que acabe la partida. Se irán leyendo las líneas, una a una, y se representarán por pantalla.

En la parte superior de la pantalla siempre habrá dos botones. Uno de ellos tiene el texto “Atrás”, y lleva al usuario de vuelta a la actividad CrearCodigoActivity (el botón físico del móvil tiene la misma función). El otro botón lleva al usuario a la siguiente pantalla que le corresponde jugar, que cambia según el estado de la partida y el tipo de nivel.

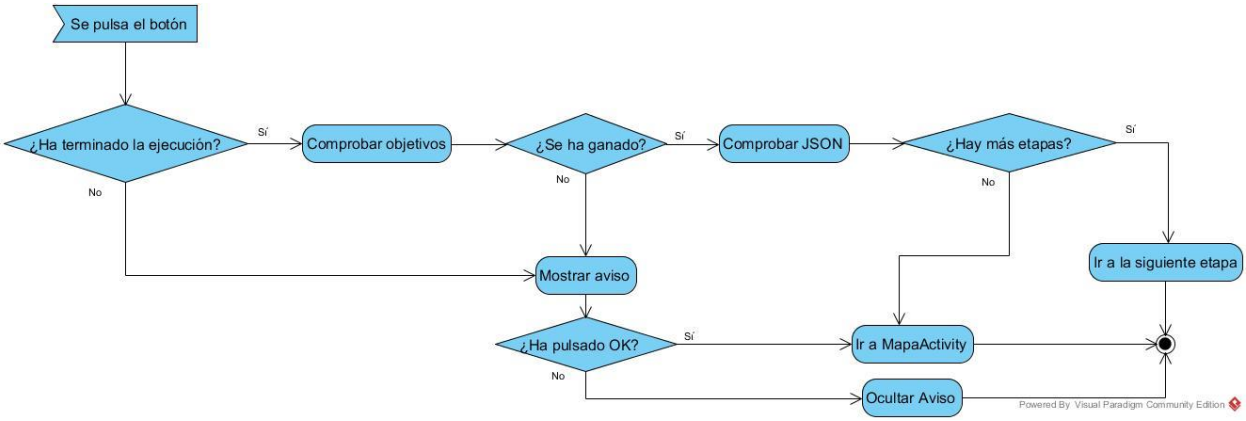

Ilustración 23 – Diagrama de actividad de botón_siguiente

5.4 Tablero y Sprites

Ilustración 24 - Tablero

El tablero de juego está compuesto por baldosas y *sprites*. Las baldosas componen el suelo, no son interactuables y únicamente determinan por dónde se pueden mover los personajes. Los *sprites* son los diferentes ítems que puede haber en el tablero (Ada, los enemigos, objetos, etc).

Para dibujar el tablero en el mapa necesitamos: la clase Tablero, la clase Sprite, un hilo y una *SurfaceView* donde dibujarlo en el *layout* de la actividad [24].

5.4.1 Sprite

Cada uno de lo selementos que se muestren en el mapa, con excepción de las baldosas, serán recogidos en un objeto de la clase Sprite.

Esta clase tiene como función controlar el movimiento, la interactuación con los demás elementos del mapa y la animación de los diferentes items. En este último caso, es fundamental un mapa de bits que contenga el dibujo del ítem [25]. Pueden tener diferentes estructuras, según el tipo de Sprite:

Ilustración 25 - Sprite_ada, sprite_cocodrilo y sprite_yunque

Los tipos de Sprite simples, que no tienen movimiento, tienen un mapa de bits más sencillo. Este es el caso de los objetos, como el yunque de la foto.

Los que tienen una animación necesitan, como mínimo, tres fotogramas. Estos se irán alternando, mostrando movimiento. Aunque el personaje esté quieto, se verá una animación estática. El caso más simple es el de los Sprites como el cocodrilo de la imagen: únicamente tiene la animación de reposo y siempre estarán mirando en un único sentido.

Los mapas de bits más complejos son los de los personajes. Estos se dividen, verticalmente, en tres secciones; una para cada acción que pueden realizar (reposo, andar y golpear). A su vez, se dividen en los tres fotogramas reglamentarios y, en horizontal, en cuatro filas; una para cada dirección.

A la hora de crear la imagen hay que tener en cuenta en qué posición de la baldosa quiere dibujarse. Por defecto, será dibujado en la esquina inferior izquierda, por lo que hay que dibujarle un espacio en blanco extra si se quiere que el personaje esté en el centro:

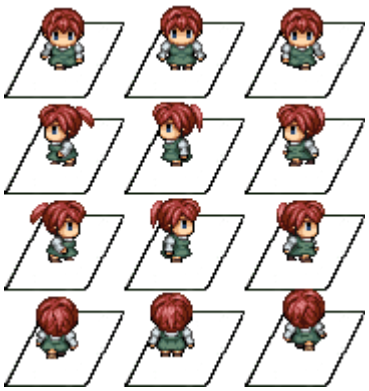

Ilustración 26 – Detalle sprite_ada

En la clase Sprite se han definido diferentes tipos elementos que puede haber en el tablero. Cada tipo un comportamiento distinto y, por lo general, también cambia el formato del mapa de bits.

```
public static final int ROCA = 0;
public static final int DIANA = 2;
public static final int COCODRILO = 3;
public static final int META = 4;
public static final int ADA = 5;
public static final int BESTIA = 6;
public static final int FIJO = 7;
public static final int OBJETO = 8;
public static final int NPC = 9;
```

Código 39 – Tipos de Sprite (Fragmento de Sprite I)

Los *sprites* del tipo “cocodrilo”, por ejemplo, no se mueven del sitio. Su mapa de bits sólo necesita tres fotogramas y, si un personaje pasa por encima, les hace daño.

El tipo “roca” y “diana” son similares. Los dos son objetos inanimados que se pueden romper, pero los *sprites* del tipo “diana” requieren ser rotos para ganar la partida, mientras que los “roca” no.

Los “NPC” son los personajes amistosos no controlables por el jugador. Tienen mapas de bits grandes y cada uno se comporta de forma diferente.

La principal función de la clase Sprite es coger el fotograma correcto del mapa de bits y dibujarlo en el tablero. La posición en el mapa de baldosas está determinada por dos atributos: “fila” y “columna”, contándose desde arriba y desde la izquierda.

A partir de la posición hay que calcular las coordenadas *(x,y)* en el *canvas* donde dibujar la imagen:

```
this.x = columna * tablero.getAnchuraBaldosa()
        + (tablero.getFilas() - 1 - fila) * tablero.getBaldosa_offset();
this.y = (fila + 1) * tablero.getAlturaBaldosa() - alto;

dest = new Rect(x + x_tablero_offset, y + y_tablero_offset,
        x + ancho + x_tablero_offset, y + alto + y_tablero_offset);
```

Código 40 – Calcular posición (Fragmento de Sprite.java II)

El primer operando de ambas asignaciones calcula la posición *x* e *y* de la esquina inferior izquierda de la baldosa sobre la que se posicionará el *sprite*.

En el caso de la coordenada *y*, hay que restar el alto del fotograma para obtener la posición para empezar a dibujarlo. El canvas empieza a dibujar a partir de la esquina superior izquierda de la imagen, y a partir de ahí continúa con el resto.

En el caso de la coordenada x, hay que sumar el desfase producido por las baldosas, ya que están inclinadas. Para ello se suma el tamaño del desfase por el número de baldosas que se mueven.

Una vez calculada la posición, se crea un rectángulo que delimitará el área del canvas en que se dibujará. A este hay que dibujarle un nuevo desfase, esta vez debido al *scroll* que tenga activado el tablero.

El tablero, en ocasiones, puede ser demasiado grande para la pantalla de un móvil y no podrá mostrarse entero. En esos casos se muestra únicamente una zona, centrada en Ada. Las variables "tablero_offset" indican cuánto hay que desplazar los sprites en cada dirección para que salgan correctamente dibujados.

Ahora que se tiene su posición, hay que controlar al personaje. Para cambiar la acción hay que llamar al método "setAccion":

```
public void setAccion(int acc, int dir) {
    accion = acc;
    fotogramaActual = 0;
    direccion = dir;
    bmp = Bitmap.createBitmap(bmp_full, ancho * bmp_columnas, 0,
            ancho * bmp_columnas, alto * bmp_filas);

    //Se configuran las opciones determinadas de cada acción, como la vida.
    switch (acc) {
        case ANDAR:
            switch (dir) {
                case DERECHA:
                    if (columna + 1 < tablero.getColumnas()) {
                        columna++;
                        vida = tablero.getAnchuraBaldosa() / velocidad;
                    } else {
                        vida = 0;
                        accion = QUIETO;
                    }
                    break;
                //[...]
            }
            break;
        //[...]
    }
}
```

Código 41 – setAccion (Fragmento de Sprite.java III)

Primero fija la acción (un entero de cero a dos, para identificar la zona del *bitmap*), la dirección (un entero de cero a tres, para identificar la fila), pone el fotograma actual a cero y guarda la sección del mapa de bits con la acción correspondiente en la variable "bmp".

"velocidad" es una variable que guarda el número de "posiciones" en el canvas que se mueve cada vez. La vida se fija para que el movimiento pare cuando se mueva la distancia requerida.

Con la posición calculada sólo queda dibujar el *sprite* en el canvas. Se realiza en dos pasos: actualizar y dibujar.

Al actualizar, normalmente sólo se cambia el fotograma de la animación, para dar un cierto movimiento cuando el personaje está quieto. Si el personaje está realizando alguna acción, se hacen cambios adicionales.

Esta es una versión abreviada del método:

```
public void actualizar() {
    fotogramaActual = ++fotogramaActual % fotogramas;
    int srcX = fotogramaActual * ancho;
    int srcY = direccion * alto;
    src.set(srcX, srcY, srcX + ancho, srcY + alto);

    if (accion == ANDAR || accion == CHOCAR) {
        switch (accion == ANDAR ? direccion : ARRIBA - direccion) {
            case DERECHA:
                dest.offset(velocidad, 0);
                x += velocidad;
                break;
            //[...]
        }
    }

    if (accion != QUIETO && --vida < 0) {
        setAccion(QUIETO, direccion);
        centrarDibujo();
    }
}
```

Código 42 – actualizar (Fragmento de Sprite.java II)

Primero se elige el fotograma que se va a dibujar, y se crea un rectángulo que lo delimite en el mapa de bits de la acción a realizar.

Si se mueve (si ha chocado, el movimiento es en la dirección contraria), y cuando la vida de la acción sea cero, se vuelve a quedar quieto.

Después de actualizar, se dibuja. Se utiliza el bitmap, el rectángulo del origen, el destino y una pintura (se utiliza para hacer la figura semi transparente).

```
public void dibujar(Canvas canvas) {
    if (salud > 0) {
        canvas.drawBitmap(bmp, src, dest, paint);
    }
}
```

Código 43 – dibujar (Fragmento de Sprite.java III)

5.4.2 Tablero

Tablero gestiona el movimiento de todos los elementos que hay en juego. Tiene tres funciones principales: dibujar los elementos en juego, controlarlos (su movimiento y las interacciones entre ellos) y actuar como interfaz para obtener datos del juego.

Hay dos tipos de tableros diferentes: el de muestra (que se dibuja en ResumenMisionActivity) y el definitivo (utilizado en ResolverCodigoActivity). En el tablero de muestra se dibujarán de forma semitransparente los *sprites* que tengan una posición aleatoria.

La clase Tablero utiliza cuatro atributos principales:

```
protected ArrayList<Sprite> items;
protected Sprite ada;
private Bitmap baldosaBase;
protected ArrayList<Bitmap> baldosas;
```

Código 44 – atributos del tablero (Fragmento de Tablero.java I)

Para dibujar el suelo utiliza "baldosas", que es una lista con el bitmap de cada baldosa del tablero. Como en la mayoría de los casos casi todas las baldosas son del mismo estilo, la baldosa principal se guarda en "baldosaBase", y el bitmap en la lista es es cero.

El *ArrayList* "items" contiene a todos los sprites del tablero, excepto Ada, que está aparte.

5.4.2.1 Creación

El archivo JSON lista cómo es el tablero y qué elementos hay en él.

```
"mapa": {
        "filas": "6",
        "columnas": "5",
        "baldosaBase": "baldosa_hierba",
        "baldosaSuelo": [
           {
               "posicion": "5",
               "bitmap": "baldosa_agua"
           },
           {
               "posicion": "8",
               "bitmap": "baldosa_agua"
           },
           […]
        ],

        "item": [
            {
                "posicion":"21",
                "tipo":"ada",
                "bitmap":"sprite_ada",
                "direccion": "arriba"
            },
            […]
        ]
    }
```

Código 45 – Tablero en JSON

El constructor del Tablero coge los diferentes elementos y los crea:

- Coge los datos del tablero (fila, columna, baldosaBase).
- Crea el *ArrayList* "baldosas" con tantos elementos como baldosas haya, y en aquella posición en que la baldosa del suelo sea distinta a la base, pone su bitmap.
- Se crean los Sprites, obteniendo los datos del array "item".
- Si es un tablero de muestra, se hacen semi transparentes los elementos aleatorios y se dibuja un símbolo en las demás posiciones posibles.

5.4.2.2 Hilo

El tablero en sí mismo no produce animación. Para ello hay que crear un hilo que le recuerde cada pocos milisegundos que debe actualizar y redibujar el contenido [26].

El ciclo de vida del hilo se une al ciclo de vida de la *SurfaceView*. Así, el hilo se creará a la vez que la superficie (no podemos crear el hilo en "onCreate", porque la superficie aún no está creada) y se eliminará cuando ésta sea eliminada.

```
holder = getHolder();
holder.addCallback(new SurfaceHolder.Callback() {
    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        hilo.cancel(true);
    }
    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        hilo = new Hilo(tablero);
        hilo.execute();
    }
    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {
    }
});
```

Código 46 – ciclo de vida del hilo (Fragmento de Tablero.java II)

El hilo extiende *AsyncTask*. Esta clase realiza un trabajo en segundo plano, sin entorpecer al hilo principal. Su método principal es "doInBackground", que en este caso es un bucle infinito que llama al método "dibujar" del tablero cada pocos milisegundos:

```
@Override
protected Void doInBackground(Void... sinusar) {

    long tEspera = 1000 / FPS;
    long inicio;
    long fin;

    while (true) {
        if(this.isCancelled())
            return null;
        inicio = System.currentTimeMillis();

        Canvas c = tablero.getHolder().lockCanvas();
        tablero.dibujar(c);
        tablero.getHolder().unlockCanvasAndPost(c);

        fin = tEspera - (System.currentTimeMillis() - inicio);
        try {
            if (fin > 0)
                Thread.sleep(fin);
            else
                Thread.sleep(10);
        } catch (Exception e) {
        }

    }
}
```

Código 47 – doInBackground de Hilo.java

Después de ordenar al tablero que se dibuje, el hilo se pausa un determinado tiempo. Este tiempo queda determinado por el número de fotogramas por segundo que se deseen. Se calcula el tiempo

dinámicamente para que se empiece a dibujar siempre cada un mismo tiempo, y no se empiece a contar el tiempo cuando el tablero termine de dibujar su contenido, que puede variar.

El bucle continuará redibujando el tablero hasta que la *SurfaceView* sea destruída y el hilo se cancele. En ese caso "isCancelled" devolverá true y el hilo terminará.

5.4.2.3 Dibujar

Cuando el hilo lo ordene, el tablero tiene que actualizar y dibujar todo su contenido. El dibujo se realiza varias veces en cada segundo. Se hace en una serie de pasos:

1) Comprobar interacciones y actualizar movimiento.
    1) Comprueba si Ada choca con algún ítem. Para ello comprueba la posición de Ada con cada uno de los demás *sprites* en juego, y comprueba si coincide.

       En caso de que sea cierto, ejecuta la reacción adecuada: disminuye la salud de Ada si se hace se hace daño (al pisar agua o un cocodrilo, por ejemplo) o cambia la acción a chocar si no puede pasar por encima de ese ítem.
    2) Si Ada está atacando, realiza el ataque y disminuye la salud del objetivo. Si el ataque falla (porque no había nada delante), es Ada quien recibe el daño. Así se penalizan los códigos mal estructurados.
    3) Si un enemigo ha iniciado su ataque y está en una posición colindante a la de Ada, computa el daño.
    4) Actualiza a Ada, utilizando el método de la clase Sprite.
    5) Si Ada se está moviendo, se actualiza el *scroll* del mapa para que contiúe estando centrado en ella.
    6) Acutaliza los ítems.
    7) Si es necesario, se actualiza la inteligencia de los personajes no controlados por el jugador (NPC). Hay dos tipos de NPC: los amistosos y los enemigos.

       Los personajes amistosos tienen comportamiento diferente según su naturaleza y el nivel que se esté jugando. Los enemigos suelen tener un único comportamiento: perseguir a Ada y atacarle.
2) Ahora que todos los personajes están actualizados, se dibujan:
    1) Se dibuja todo el canvas de un color plano (para tapar lo que hubiera antes).
    2) Se dibujan las baldosas.
    3) Se dibujan los ítems de la lista.
    4) Se dibuja a Ada.

5.4.2.4 Otros métodos

La clase Tablero también dispone de varios métodos que actúan como puente entre el código que ha realizado el jugador y el juego.

El método "mirar" método devuelve un *int* con el código que representa lo que hay en la baldosa especificada. Para ello recorre todos los ítems del tablero y devuelve el tipo de aquel que tenga una posición que coincida. Además, muestra un mensaje indicando lo que se ha visto, pero no se muestra en el código siguiente para simplificarlo:

```
public int mirar (int x, int y){
    int i = 0;
    int m = 0;
    boolean flag = true;
    if(x < columnas && y < filas ) {
        while (i < items.size() && flag) {
            if (items.get(i).getColumna() == x && items.get(i).getFila() == y) {
                switch (items.get(i).getTipo()) {
                    case (Sprite.META):
                        m = Codigo.CTE_META;
                        break;
                    //[...]
                }
                flag = false;
            }
            i++;
        }
        if (flag){
            m = Codigo.CTE_VACIO;
        }
    }
    return m;
}
```

Código 48 – mirar (Fragmento de Tablero.java III)

5.5 Multilenguaje

Se da la opción de elegir qué lenguaje enseñará el videojuego. Esta opción se cambia pulsando el engranaje en MapaActivity. El usuario podrá elegir el lenguaje que prefiera de una lista desplegable:

Ilustración 27 – Elegir lenguaje

Esto provocará dos cambios principales en el videojuego:

1) El código que se escribirá tendrá una sintaxis distinta.

2) Las ayudas que se muestran al inicio de cada nivel serán diferentes.

El primer punto lo implementa el método “escribirLinea” de la clase Código; escribirá de forma diferente según el lenguaje especificado.

El segundo es más complicado. Las ayudas están guardadas en la carpeta "res/strings.xml" de esta forma:

```
<string name="ayuda_interfaz_titulo">Teclado I</string>
<string name="ayuda_interfaz"><![CDATA[
<p>Texto de la ayuda</p>
]]></string>

<string name="ayuda_teclado_titulo_Java">Teclado</string>
<string name="ayuda_teclado_Java"><![CDATA[
<p>Texto de la ayuda</p>
]]></string>

<string name="ayuda_teclado_titulo_C">Teclado</string>
<string name="ayuda_teclado_C"><![CDATA[
<p>Texto de la ayuda.</p>
]]></string>
```

Código 49 – Ayudas en strings.xml

Hay dos tipos de ayudas: las genéricas y las que que son específicas de un lenguaje concreto.

Cuando, desde el archivo JSON, se indican qué ayudas mostrar en el nivel, no se dice el nombre completo, sino su identificador propio ("interfaz" o "teclado"). A partir de ahí, se busca el id de Android para acceder a los recursos.

Primero se comprueba si existe una ayuda para el lenguaje seleccionado. Si la hay, se coge; si no, se coge la genérica. Y si esta tampoco está, significa que esa ayuda no está destinada para este lenguaje concreto y no se pone ninguna

```
String [] lista_ayudas = parserJSON.getAyuda("ayuda");
titulos = new ArrayList<>();
ayudas = new ArrayList<>();
int titulo_id;
String lenguaje = sharedPref.getString("lenguaje","Java");

for (String nombre : lista_ayudas) {
    titulo_id = getResources().getIdentifier(
            "ayuda_" + nombre + "_titulo" + "_" + lenguaje, "string", getPackageName());

    //Si hay una string particular para este lenguaje:
    if (titulo_id > 0) {
        titulos.add(getString(titulo_id));
        ayudas.add(getString(getResources().getIdentifier(
                "ayuda_" + nombre + "_" + lenguaje, "string", getPackageName())));
        //Si no lo hay se coge el por defecto
    } else {

        titulo_id = getResources().getIdentifier(
                "ayuda_" + nombre + "_titulo", "string", getPackageName());

        //Si hay uno genérico, se pone
        if (titulo_id > 0) {
            titulos.add(getString(titulo_id));
            ayudas.add(getString(getResources().getIdentifier(
                    "ayuda_" + nombre, "string", getPackageName())));
        }
        //Y si esa ayuda sólo está disponible para un mensaje concreto no se hace nada.
    }
}
```

Código 50 – Obtener ayudas en CrearCodigoActivity

De esta forma, la teoría cambia según el lenguaje seleccionado.

A veces con sólo esta modificación no es suficiente. Por ello, algunos niveles sólo están disponibles para un lenguaje concreto. Esto permite crear nuevos niveles para las características propias de cada lenguaje, que no son literalmente traducibles de un lenguaje a otro.

5.6 Preferencias compartidas

En el desarrollo del viedojuego se han usado con extensión las preferencias compartidas. Se usan para guardar el progreso del jugador, datos del personaje, las teclas y ayudas desblqueadas, entre otras cosas.

Estas son algunos de los pares nombre-valor guardados:

- Los que empiezan por "control_" se utilizan para que únicamente se ejecute algo una vez. Por ejemplo, "control_inicializado" guarda el valor *true* cuando se realiza la configuración inicial del juego.
- Al iniciar por primera vez cada nivel hay que realizar algunas acciones de configuración (como desbloquear las teclas). Cuando esta configuración se realiza, se guarda *true* en la preferencia compartida "nombredelnivel_etapa". Este campo también se utiliza para saber si se ha entrado en este nivel alguna vez.
- Algunos pares son datos personales del personaje y el jugador: "ciudad" guarda la última ciudad visitada, "lenguaje" el lenguaje configurado, "experiencia" el nivel del jugador (al completar con éxito un nivel aumenta la experiencia del jugador, permitiéndole acceder a misiones nuevas) y "constantes" y "variables" guardan un entero con el número de constantes y variables desbloqueadas, respectivamente.
- "tecla_i_j" guarda *true* cuando la tecla *j* del teclado *i* ha sido desblqueada.
- "ayuda_titulo_i" y "ayuda_i" se utilizan para mostrar los apuntes ya vistos en el libro de apuntes. Guardan el identificador del recurso, para poder acceder a él directamente en la carpeta "strings.xml".

5.7 Conclusiones

En este capítulo se ha realizado un recorrido por los principales elementos que se han implementado. No se ha mostrado toda la aplicación, pero sí aquellas seecciones que son más complejas o interesantes.

En esta etapa de la memoria ya se ofrecido una visión completa del videojuego: se ha explicado su objetivo, cómo se juega al mismo y cómo es su código interno.

En el siguiente capítulo se comprobará que todo funciona correctamente, instalando la aplicación terminada en el móvil y realizando algunas pruebas.

6 PRUEBAS REALIZADAS

En este apartado se va a realizar un recorrido por la aplicación, mostrando en ejecución que funcionan las diferentes prestaciones que se han desarrollado. Esta sección se ha dividido en varias etapas, una para cada prueba que se quiere superar. En cada una de ellas se comienza explicando el objetivo a probar, seguido de capturas con la ejecución y, a veces, unas breves líneas explicando algún punto particular de la ejecución.

6.1 Primera ejecución

La primera vez que se inicia la aplicación, saldrá el logo del videojuego y de ahí saltará a la misión "Tutorial", donde se mostrará la ayuda visual para pulsar la pantalla. También se comprobará que esta ayuda visual aparezca en la actividad ResolverCodigoActivity.

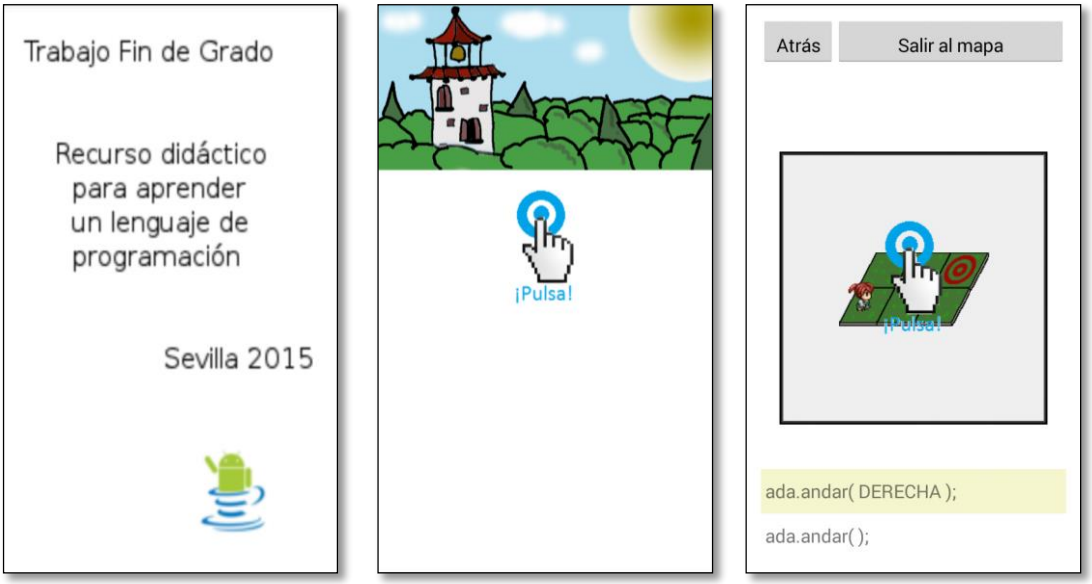

Ilustración 28 – Prueba 1 - Primera ejecución

6.2 Nuevos apuntes

Nuevos apuntes serán añadidos tras superar con éxito algunas misiones. Cuando esto ocurra aparecerá un *Toast* indicándolo al volver al mapa, el libro de apuntes tendrá un signo de exclamación y, la primera vez, habrá una flecha apuntando hacia el libro. Cuando se abra el libro de apuntes se mostrará inicialmente la nueva incorporación, con una señal que lo indique.

Ilustración 29 – Prueba 2 - Nuevos apuntes

6.3 Elementos al azar

Algunos niveles tienen elementos aleatorios. En ResumenMisionActivity estos deben mostrarse semitransparente o con un signo de interrogación, mientras que en ResolverCodigoActivity se debe mostrar una única posibilidad. Esta será aquella en la que el código falle, mientras que si se superan todos los escenarios se elegirá una al azar.

Ilustración 30 – Prueba 3 - Elementos aleatorios en ResumenMisionActivity

Si se desarrollo un código que cosista en andar en línea recta, la ejecución no llegará a la meta si la roca está a la izquierda. Por lo tanto, ese será el escenario que debe mostrarse:

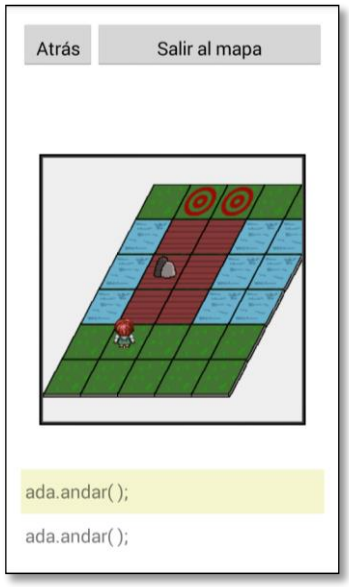

Ilustración 31 – Prueba 3 - Comprobar qué escenario no es superado

Si el código es versátil, puede ser cualquiera de los dos escenarios posible el que se utilice. Además, se comprueba que el condicional "if" funcione correctamente.

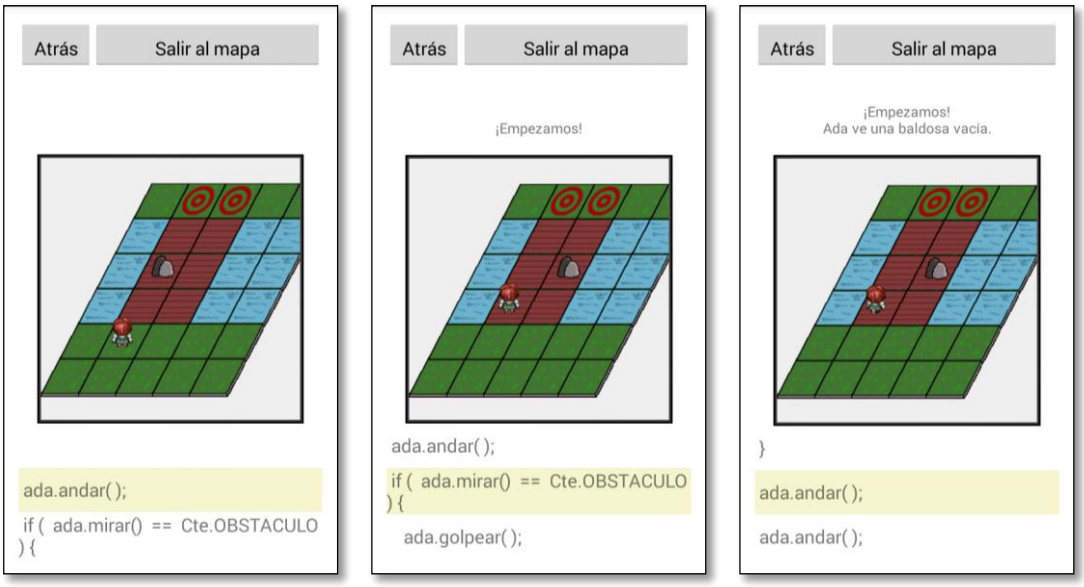

Ilustración 32 – Prueba 3 - Código funcional en varios escenarios

En las imágenes se muestran dos ejecuciones distintas: una de ellas en el escenario A y las dos imágenes siguientes en el escenario B.

6.4 Error de ejecución y victoria

Cuando se produce un error en el código, debe mostrarse con un Toast y, si es la primera vez que un nivel no es superado, mostrar una flecha apuntando al botón "Atrás". Una vez que ha ocurrido un error la ejecución parará y el jugador no podrá seguir ejecutando líneas de código.

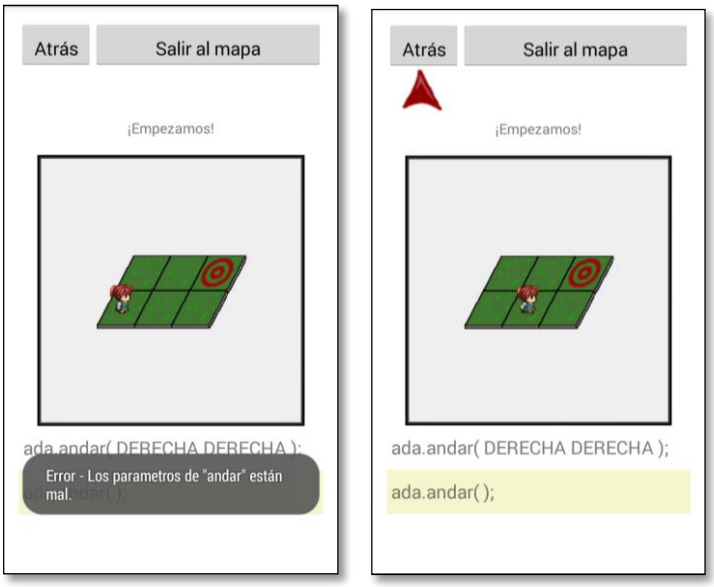

Ilustración 33 – Prueba 4 - Error

Por otra parte, cuando el código del jugador supera satisfactoriamente el nivel debe mostrarse un *Toast* con la palabra "VICTORIA". También se comprobará que al pisar la meta el nivel reconozca que el jugador ha ganado.

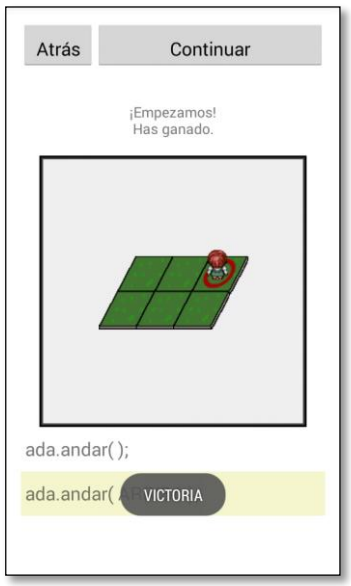

Ilustración 34 – Prueba 5 - Victoria

6.5 Nuevos niveles

Cuando se completa un nivel se desbloquea uno nuevo y, a veces, un nuevo lugar del mapa.

El nuevo nivel tendrá un signo de exclamación amarillo hasta que el usuario entre en él por primera vez, momento en que desaparecerá.

Ilustración 35 – Prueba 5 - Nuevos niveles

6.6 Multilenguaje

Si se decide optar por otro lenguaje de programación, la sintáxis cambiará y las líneas de código se escribirán de forma distinta al pulsar los mismos botones.

Ilustración 36 Prueba 6 - Ejemplo de ejecución en C

6.7 Conclusiones

Con estas pruebas se ha comprobado que la aplicación funciona y que no hay ningún fallo evidente, aunque no se descarta que bajo determinadas circunstancias pueda darse un fallo inesperado.

A continuación se presenta el capítulo final de la memoria, con unas conclusiones finales y líneas futuras para la aplicación.

7 CONCLUSIONES Y LÍNEAS FUTURAS

Aprender un lenguaje de programación no es fácil. Este proyecto puede ser útil para aquellos estudiantes que estén aprendiendo: un usuario que juegue a este videojuego y lo acabe debería tener un conocimiento básico de Java y saber utilizar algunas de las estructuras de control más comunes. A partir de ahí podrá aumentar sus conocimientos con otros medios lectivos.

Sin embargo, el proyecto realizado debe visualizarse como un prototipo. Para obtener un producto terminado y que sea posible utilizarlo con buenos resultados, es necesario seguir trabajando a partir del estado actual.

Se ha realizado lo mejor posible, y ha sido preparado para que sea fácil ampliarlo más adelante. Actualmente únicamente implementa algunos elementos de Java, centrados en las estructuras de control. Se han creado aquellos niveles necesarios necesarios para mostrar el funcionamiento de la aplicación y obtener una primera impresión de cómo se usan las estructuras de control.

Durante el desarrollo de este proyecto se han centrado los esfuerzos en diseñar el funcionamiento general y escribir el código que lo haga funcionar. Otras facetas del proyecto, como la parte gráfica y el diseño de los niveles del videojuego.

Por todo esto, en versiones siguientes de este proyecto hay varias secciones que será necesario ampliar o cambiar:

- El contenido del lenguaje a enseñar deberá ser ampliado con nuevas funciones, como por ejemplo las tablas o los objetos. Se añadirán más teclas dependiendo de hasta qué punto se quiera enseñar el lenguaje de programación.
- En esta versión sólo se enseña Java; en el futuro es probable que se desee dar soporte a varios lenguajes de programación.
- Será necesario añadir más niveles. Algunos de ellos para afianzar conocimientos, repitiendo lecciones de niveles anteriores, y otros para introducir nuevos elementos.
- Un cambio en la interfaz gráfica sería recomendable. Se ha intentado crear dibujos y gráficos agradables visualmente, pero estos no son suficientes para una aplicación comercial.

BIBLIOGRAFÍA

Herramientas y videojuegos para aprender programación

[1] Gonzalo Sánchez, “7 juegos para aprender a programar”.
http://blog.en1mes.com/2014/11/7-juegos-para-aprender-programar

[2] Richard Moss, "8 iOS Apps That Teach You How to Code".
http://www.maclife.com/article/gallery/8_ios_apps_teach_you_how_code

[3] CodeAcademy.
http://www.codecademy.com/es/learn

[4] CodeMaven y GameMaven.
http://www.crunchzilla.com

[5] Ruby Warrior.
https://www.bloc.io/ruby-warrior

[6] CodeCombat.
http://codecombat.com

[7] CodinGame.
https://www.codingame.com

[8] CodeAcademy versión móvil.
https://itunes.apple.com/us/app/codecademy-code-hour/id762950096?mt=8

[9] Lightbot.
https://play.google.com/store/apps/details?id=com.lightbot.lightbot

[10] Hopscotch.
https://itunes.apple.com/us/app/hopscotch-coding-for-kids/id617098629?mt=8

[11] Hakitzu Elite: Robot Hackers.
https://itunes.apple.com/es/app/hakitzu-elite-robot-hackers/id599976903?mt=8
https://play.google.com/store/apps/details?id=com.kuatostudios.hakitzu

[12] Cristobal Tapia García, Luis Dávila Gómez, Luis Castedo Cepeda, Cecilia García Cena y Basil Al Hadithi, "Aplicaciones Flash para el aprendizaje del Lenguaje de Programación C".
http://taee.euitt.upm.es/actas/2012/papers/2012S3A1.pdf

Android

[13] Manuel Gross, "Android: Origen, evolución y liderazgo del Sistema Operativo para smartphones".
http://manuelgross.bligoo.com/20121026-android-origen-evolucion-y-liderazgo-del-sistema-operativo-para-smartphones

[14] "Strategy Analytics: Android Shipped 1 Billion Smartphones Worldwide in 2014".
http://www.prnewswire.com/news-releases/strategy-analytics-android-shipped-1-billion-smartphones-worldwide-in-2014-300027707.html

[15] Página oficial de desarrolladores Android.
http://developer.android.com/index.html

[16] Stack Overflow.
http://stackoverflow.com/

[17] Jose Angel Zamora, "Aprende Android en 20 conceptos".
http://www.elandroidelibre.com/2014/02/aprende-android-en-20-conceptos-empezando-a-programar-para-android.html

[18] Ravi Tamada, "Android JSON Parsing Tutorial".
http://www.androidhive.info/2012/01/android-json-parsing-tutorial/

[19] Joe, "Android Chat Buble".
http://javapapers.com/android/android-chat-bubble/

[20] Maarten Pennings, "Android Development: Custom keyboard".
http://www.fampennings.nl/maarten/android/09keyboard/index.htm

[21] "Guide: The right soft keyboard".
http://forum.xda-developers.com/showthread.php?t=2497237

[22] Repositorio del teclado LatinIME.
https://android.googlesource.com/platform/packages/inputmethods/LatinIME/

[23] Repositorio de Android.
https://github.com/android/platform_frameworks_base/tree/master/core/res/res/drawable-hdpi

[24] "Android Game Programming".
http://www.edu4java.com/en/androidgame/androidgame1.html

[25] Generador de sprites.
http://www.famitsu.com/freegame/tool/chibi/index2.html

[26] Ramón Invarato, "AsynkTask en Android".
http://jarroba.com/asynctask-en-android

Otros

[27] Visual Paradigm, software de diseño de diagramas.
http://www.visual-paradigm.com/

[28] JSON
http://json.org/

[29] Verificador de ficheros JSON
http://json.org/

[30] Javadoc
https://en.wikipedia.org/wiki/Javadoc

ANEXO A

En este apartado se anexa la documentación generada a partir del código, utilizando la herramienta Javadoc [30]. En las páginas siguientes se mostrará toda la documentaciór acerca de cada clase.

com.alfonsomolina.tfg

Class MainActivity

java.lang.Object
    Activity
        com.alfonsomolina.tfg.MainActivity

```
public class MainActivity
extends Activity
```

"Splash Activity" que se muestra al iniciar la aplicacion. Muestra una animación de bienvenida, con el logo del juego, y al cabo de unos instantes redirige al mapa. Realiza la configuración inicial la primera vez que se ejecute.

Constructor Summary

Constructors

Constructor and Description
`MainActivity()`

Method Summary

Methods

Modifier and Type	Method and Description
void	`irMapa()` Inicia MapaActivity.
protected void	`onCreate(Bundle savedInstanceState)` Constructor.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

MainActivity

```
public MainActivity()
```

Method Detail

onCreate

```
protected void onCreate(Bundle savedInstanceState)
```

Constructor. Si es la primera vez ejecuta inicializarSharedPreferences, y, un segundo y medio mas tarde, irMapa.

Parameters:

`savedInstanceState` - Bundle necesario.

irMapa

```
public void irMapa()
```

Inicia MapaActivity. Si es la primera vez, inicia el tutorial en IntroduccionActivity.

com.alfonsomolina.tfg

Class MapaActivity

java.lang.Object
    Activity
        com.alfonsomolina.tfg.MapaActivity

```
public class MapaActivity
extends Activity
```

Permite recorrer el mapa y elegir a qué nivel acceder. Tiene botones para ir a los apuntes y la configuración.

Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
static class	**MapaActivity.AvisoCambiarIdioma** Controla el Dialog al cambiar el idioma.
static class	**MapaActivity.AvisoReinicio** Controla el Dialog al reiniciar el juego.

Constructor Summary

Constructors

Constructor and Description
MapaActivity()

Method Summary

Methods

Modifier and Type	Method and Description
void	**iniciarCiudad**(java.lang.String ciudad) Muestra los datos de la ciudad y los niveles disponibles.
void	**irApuntes**(View view) Inicia ApuntesActivity.
void	**irCiudad**(View view) Carga la ciudad existente en la direccion pulsada.
void	**irNivel**(View view) Inicia IntroduccionActivity con el nivel elegido.
void	**mostrarOpciones**(View view) Muestra las opciones del juego: reiniciar y cambiar idioma.
void	**mostrarSpinner**(View view) Muestra el spinner con los idiomas disponibles.
void	**onBackPressed**() Sale de la aplicacion.

protected void	**onCreate**(Bundle savedInstanceState) Constructor.
void	**onResume**() Oculta la flecha al botón de apuntes, en caso de que se vuelve desde ApuntesActivity.
void	**reiniciar**(View view) Reinicia el juego.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

MapaActivity

```
public MapaActivity()
```

Method Detail

onCreate

```
protected void onCreate(Bundle savedInstanceState)
```

Constructor. Carga el mapa con la configuracion adecuada. Muestra la última ciudad visitada y los niveles disponibles del jugador, según su experiencia.

Parameters:

`savedInstanceState` - Bundle necesario.

iniciarCiudad

```
public void iniciarCiudad(java.lang.String ciudad)
```

Muestra los datos de la ciudad y los niveles disponibles.

Parameters:

`ciudad` - con el identificador de la ciudad.

irCiudad

```
public void irCiudad(View view)
```

Carga la ciudad existente en la direccion pulsada.

Parameters:

`view` - vista del botón pulsado.

irNivel

```
public void irNivel(View view)
```

Inicia IntroduccionActivity con el nivel elegido. Si se ha decidido ir a la última etapa visitada, se mostrara esa.

Parameters:

`view` - vista del botón pulsado.

irApuntes

```
public void irApuntes(View view)
```

Inicia ApuntesActivity.

Parameters:

`view` - vista del botón pulsado.

mostrarOpciones

```
public void mostrarOpciones(View view)
```

Muestra las opciones del juego: reiniciar y cambiar idioma.

Parameters:

`view` - del botón pulsado.

reiniciar

```
public void reiniciar(View view)
```

Reinicia el juego.

Parameters:

`view` - vista del botón pulsado.

mostrarSpinner

```
public void mostrarSpinner(View view)
```

Muestra el spinner con los idiomas disponibles.

Parameters:

`view` - del botón pulsado.

onBackPressed

```
public void onBackPressed()
```

Sale de la aplicacion.

onResume

```
public void onResume()
```

Oculta la flecha al botón de apuntes, en caso de que se vuelve desde ApuntesActivity.

com.alfonsomolina.tfg

Class ApuntesActivity

java.lang.Object
    Activity
        com.alfonsomolina.tfg.ApuntesActivity

```
public class ApuntesActivity
extends Activity
```

Muestra los apuntes desbloqueados.

Constructor Summary

Constructors

Constructor and Description
`ApuntesActivity()`

Method Summary

Methods

Modifier and Type	Method and Description
void	`mostrarIndice(View view)` Muesta u oculta el índice.
void	`mostrarPagina()` Muestra la ayuda seleccionada.
void	`onBackPressed()` Al pulsar el boton de "atrás", oculta el índice si estaba visible; si no lo estaba, vuelve a MapaActivity.
protected void	`onCreate(Bundle savedInstanceState)` Constructor.
void	`pasarPagina(View view)`

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

ApuntesActivity

```
public ApuntesActivity()
```

Method Detail

onCreate

```
protected void onCreate(Bundle savedInstanceState)
```

Constructor. Crea la clase leyendo las preferencias compartidas y cargando las ayudas desbloqueadas.

Parameters:

`savedInstanceState` - bundle necesario.

mostrarIndice

```
public void mostrarIndice(View view)
```

Muesta u oculta el índice.

Parameters:

`view` - vista del boton pulsado

pasarPagina

```
public void pasarPagina(View view)
```

mostrarPagina

```
public void mostrarPagina()
```

Muestra la ayuda seleccionada.

onBackPressed

```
public void onBackPressed()
```

Al pulsar el boton de "atrás", oculta el índice si estaba visible; si no lo estaba, vuelve a MapaActivity.

com.alfonsomolina.tfg

Class IntroduccionActivity

java.lang.Object
    Activity
        com.alfonsomolina.tfg.IntroduccionActivity

```
public class IntroduccionActivity
extends Activity
```

Muestra a dos personajes hablando, introduciendo el nivel y avanzando en la trama del videojuego.

Constructor Summary

Constructors

Constructor and Description
`IntroduccionActivity()`

Method Summary

Methods

Modifier and Type	Method and Description
`void`	`getMensaje()` Muestra un mensaje más cuando se pulsa la pantalla.
`void`	`irNivel(View view)` Lanza la siguiente actividad del nivel.
`void`	`onBackPressed()` Al pulsar el botón de "atrás" se solicita una confirmación para salir.
`protected void`	`onCreate(Bundle savedInstanceState)` Constructor.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

IntroduccionActivity

```
public IntroduccionActivity()
```

Method Detail

onCreate

```
protected void onCreate(Bundle savedInstanceState)
```

Constructor. Realiza la inicializacion de la etapa.

Parameters:

`savedInstanceState` - Bundle necesario.

getMensaje

```
public void getMensaje()
```

Muestra un mensaje más cuando se pulsa la pantalla. Se encarga de mestrar la animación y el texto.

irNivel

```
public void irNivel(View view)
```

Lanza la siguiente actividad del nivel. Suele ser ResumenMisionActivity, pero si el nivel es únicamente un diálogo se hace la configuracion final de la etapa y se va a la siguiente. Puede darse el caso de que la etapa final sea un diálogo, por lo que hay que hacer la configuración final del nivel (aumentar la experiencia y las ayudas extra).

Parameters:

`view` - vista del boton pulsado.

onBackPressed

```
public void onBackPressed()
```

Al pulsar el botón de "atrás" se solicita una confirmación para salir.

com.alfonsomolina.tfg

Class ResumenMisionActivity

java.lang.Object
    Activity
        com.alfonsomolina.tfg.ResumenMisionActivity

```
public class ResumenMisionActivity
extends Activity
```

Muestra un resumen de la misiso, los objetivos de ella y el tablero de juego. El tablero será de muestra, los elementos aleatorios no estarán decididos y serán semitransparentes.

Constructor Summary

Constructors

Constructor and Description
`ResumenMisionActivity()`

Method Summary

Methods

Modifier and Type	Method and Description
void	`irNivel(View view)` Inicia CrearCodigoActivity.
void	`onBackPressed()` Al pulsar el botón de "atrás" vuelve a CrearCodigoActivity, si la actividad anterior fue esa, o solicita una confirmación para salir al mapa.
protected void	`onCreate(Bundle savedInstanceState)` Constructor.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

ResumenMisionActivity

```
public ResumenMisionActivity()
```

Method Detail

onCreate

```
protected void onCreate(Bundle savedInstanceState)
```

Constructor. Muestra la información del nivel y dibuja un tablero de muestra, con los elementos aleatorios aún no decididos.

Parameters:

`savedInstanceState` - Bundle necesario

irNivel

```
public void irNivel(View view)
```

Inicia CrearCodigoActivity.

Parameters:

`view` - vista del botón pulsado

onBackPressed

```
public void onBackPressed()
```

Al pulsar el botón de "atrás" vuelve a CrearCodigoActivity, si la actividad anterior fue esa, o solicita una confirmación para salir al mapa.

com.alfonsomolina.tfg

Class CrearCodigoActivity

java.lang.Object
    Activity
        com.alfonsomolina.tfg.CrearCodigoActivity

```
public class CrearCodigoActivity
extends Activity
```

Muesta un teclado para escribir código y las líneas que ya han sido escritas.

Constructor Summary

Constructors

Constructor and Description
`CrearCodigoActivity()`

Method Summary

Methods

Modifier and Type	Method and Description
void	`cerrar(View view)` Cierra las ayudas.
void	`mostrarAyuda(View view)` Muestra la ayuda.
void	`mostrarCabecera(View view)` Muestra el código escrito dentro de una clase.
void	`mostrarResumen(View view)` Lanza de nuevo la actividad con el resumen de la misión.
void	`ocultarTeclas(int teclado)` Oculta las teclas que no hayan sido desbloqueadas.
void	`onBackPressed()` Al pulsar el boton de "atrás", oculta las ayudas y el popup y, si no estaban visibles, muestra un aviso solicitando una confirmación para salir.
protected void	`onCreate(Bundle savedInstanceState)` Constructor.
void	`pasarDerecha(View view)` Muestra la siguiente ayuda de las disponibles.
void	`pasarIzquierda(View view)` Muestra la ayuda anterior de las disponibles.
void	`resolver(View view)` Inicia ResolverCodigoActivity para ejecutar las líneas escritas.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

CrearCodigoActivity

```
public CrearCodigoActivity()
```

Method Detail

onCreate

```
protected void onCreate(Bundle savedInstanceState)
```

Constructor. Crea la actividad obteniendo el mensaje de ayuda y las líneas de muestra.

Parameters:

`savedInstanceState` - bundle necesario

mostrarAyuda

```
public void mostrarAyuda(View view)
```

Muestra la ayuda.

Parameters:

`view` - vista del botón pulsado.

pasarDerecha

```
public void pasarDerecha(View view)
```

Muestra la siguiente ayuda de las disponibles.

Parameters:

`view` - vista del botón pulsado.

pasarIzquierda

```
public void pasarIzquierda(View view)
```

Muestra la ayuda anterior de las disponibles.

Parameters:

`view` - vista del botón pulsado.

cerrar

```
public void cerrar(View view)
```

Cierra las ayudas.

Parameters:

`view` - vista del botón pulsado.

ocultarTeclas

```
public void ocultarTeclas(int teclado)
```

Oculta las teclas que no hayan sido desbloqueadas.

Parameters:

`teclado` - entero con el número del teclado activo.

mostrarResumen

```
public void mostrarResumen(View view)
```

Lanza de nuevo la actividad con el resumen de la misión.

Parameters:

`view` - vista del botón pulsado.

mostrarCabecera

```
public void mostrarCabecera(View view)
```

Muestra el código escrito dentro de una clase.

Parameters:

`view` - vista del botón pulsado.

resolver

```
public void resolver(View view)
```

Inicia ResolverCodigoActivity para ejecutar las líneas escritas.

Parameters:

`view` - vista del botón pulsado.

onBackPressed

```
public void onBackPressed()
```

Al pulsar el boton de "atrás", oculta las ayudas y el popup y, si no estaban visibles, muestra un aviso solicitando una confirmación para salir.

com.alfonsomolina.tfg

Class ResolverCodigoActivity

java.lang.Object
    Activity
        com.alfonsomolina.tfg.ResolverCodigoActivity

```
public class ResolverCodigoActivity
extends Activity
```

Muestra la ejecución del código de forma gráfica en una SurfaceView con el tablero.

Field Summary

Fields

Modifier and Type	Field and Description
static int	HABLAR
static int	LIMPIAR
static int	META
static int	METAYLIMPIAR
static int	USAR

Constructor Summary

Constructors

Constructor and Description
ResolverCodigoActivity()

Method Summary

Methods

Modifier and Type	Method and Description
void	atras(View view) Vuelve a EscribirCodigoActivity sin perder las líneas escritas.
void	ejecutar() Controla la ejecución de la linea.
void	escribir() Escribe la línea que se esta ejecutando y las dos adyacentes.
void	leerLinea() Lee la línea actual y la ejecuta.
protected void	onCreate(Bundle savedInstanceState) Constructor.
void	salir(View view) Sale de la actividad.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

ResolverCodigoActivity

```
public ResolverCodigoActivity()
```

Method Detail

onCreate

```
protected void onCreate(Bundle savedInstanceState)
```

Constructor. Prueba en diferentes distribuciones del tablero para comprobar si el código cumple en todas.

Parameters:

`savedInstanceState` - bundle necesario.

ejecutar

```
public void ejecutar()
```

Controla la ejecución de la linea. Se asegura de no haber llegado al final, comprueba si se ha ganado o perdido y obliga al jugador a esperar un tiempo entre ejecuciones.

escribir

```
public void escribir()
```

Escribe la línea que se esta ejecutando y las dos adyacentes.

leerLinea

```
public void leerLinea()
```

Lee la línea actual y la ejecuta.

salir

```
public void salir(View view)
```

Sale de la actividad. Va a una u otra distinta según el estado de la ejecución. Si se ha ganado va a la siguiente, si se ha perdido o se sigue ejecutando vuelve al mapa tras mostrar un aviso.

Parameters:

`view` - vista del boton pulsado.

atras

```
public void atras(View view)
```

Vuelve a EscribirCodigoActivity sin perder las líneas escritas.

Parameters:

`view` - vista del botón pulsado.

com.alfonsomolina.tfg

Class Codigo

java.lang.Object
    com.alfonsomolina.tfg.Codigo

All Implemented Interfaces:

java.io.Serializable

```
public class Codigo
extends java.lang.Object
implements java.io.Serializable
```

Guarda información del codigo escrito en una línea de código. Implementa serializable para poder mandarlo en un intent.

See Also:

Serialized Form

Field Summary

Fields

Modifier and Type	Field and Description
static int	ABAJO
static int	AND
static int	ANDAR
static int	ARRIBA
static int	ASIGNACION
static int	BREAK
static int	CASE
static int	CATCH
static int	CIERRE
static int	COGER
static int	CTE
static int	CTE_BESTIA
static int	CTE_EXCEPCION
static int	CTE_FIJO
static int	CTE_META
static int	CTE_PERSONA
static int	CTE_ROCA
static int	CTE_VACIO
static int	DECLARAR
static int	DERECHA
static int	DESCANSAR
static int	DESIGUAL
static int	DO
static int	ELSE
static int	ELSE_IF
static int	FALSE

static int	FIN_ACCIONES
static int	FIN_BUCLE
static int	FIN_BUCLE_INICIO
static int	FIN_COMP
static int	FIN_CONSTANTES
static int	FIN_LETRAS
static int	FOR
static int	FOR_SEP
static int	GOLPEAR
static int	HABLAR
static int	IF
static int	IGUAL
static int	IZQUIERDA
static int	MAYOR
static int	MAYOR_IGUAL
static int	MENOR
static int	MENOR_IGUAL
static int	MIRAR
static int	NUMERO
static int	OIR
static int	OR
static int	SWITCH
static int	TRUE
static int	TRY
static int	USAR
static int	VAR_BOOLEAN
static int	VAR_INT
static int	VARIABLE
static int	WHILE

Constructor Summary

Constructors

Constructor and Description
Codigo(java.lang.String lenguaje, int c) Constructor.

Method Summary

Methods

Modifier and Type	Method and Description
void	addCadena(int parametro, java.lang.String cadena) Introduce un nuevo parámetro con una cadena complementaria.
void	addParam(int param)

	Añade un nuevo parametro.
boolean	**eliminarParam**() Borra una condición.
java.lang.String	**escribirLinea**() Devuelve la íinea de código en caracteres.
java.lang.String	**getCadena**() Devuelve la cadena delúultimo parametro de la línea.
java.lang.String	**getCadena**(int i) Devuelve la cadena de un parámetro en una posición determinada.
int	**getCodigo**() Devuelve el código principal de la línea.
int	**getNumParam**() Devuelve el número de paáametros.
int	**getParam**() Devuelve el último parámetro de la línea.
int	**getParam**(int i) Devuelve el parámetro de la posición escogida.
int	**getTab**() Devuelve el número de tabulaciones de la línea.
void	**setCadena**(java.lang.String s) Cambia la cadena del último parámetro.
void	**setCadena**(java.lang.String s, int i) Cambia la cadena del parámetro elegido.
void	**setCodigo**(int codigo) Cambia el código principal.
void	**setTab**(int nTab) Cambia el número de tabulaciones.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Codigo

```
public Codigo(java.lang.String lenguaje,
      int c)
```

Constructor.

Parameters:

`lenguaje` - String con el lenguaje de programacóon elegido.

`c` - entero con el óodigo principal de la linea.

Method Detail

escribirLinea

```
public java.lang.String escribirLinea()
```

Devuelve la íinea de código en caracteres.

Returns:

devuelve un String con la línea de código.

setCodigo

```
public void setCodigo(int codigo)
```

Cambia el código principal.

Parameters:

`codigo` - entero con el nuevo código a fijar.

addParam

```
public void addParam(int param)
```

Añade un nuevo parametro.

Parameters:

`param` - entero con el código del parámetro a incluir.

getCodigo

```
public int getCodigo()
```

Devuelve el código principal de la línea.

Returns:

devuelve un entero con el código principal de la línea.

eliminarParam

```
public boolean eliminarParam()
```

Borra una condición. Si tras borrarla no queda ninguna, pone el código principal a cero. Si ha borrado algo devuelve true.

Returns:

devuelve un boolean que vale "true" si ha borrado un parámetro, "false" en caso contrario

getParam

```
public int getParam()
```

Devuelve el último parámetro de la línea.

Returns:

devuelve un entero con el código del íltimo parámetro de la línea.

getParam

```
public int getParam(int i)
```

Devuelve el parámetro de la posición escogida.

Parameters:

`i` - entero con la posición del parámetro en la línea.

Returns:

devuelve un entero con el código del parámetro.

getNumParam

```
public int getNumParam()
```

Devuelve el número de paáametros.

Returns:

devuelve un entero con el número de parámetros.

addCadena

```
public void addCadena(int parametro,
            java.lang.String cadena)
```

Introduce un nuevo parámetro con una cadena complementaria.

Parameters:

`parametro` - entero con el código del parámetro.

`cadena` - String con la cadena complementaria.

setCadena

```
public void setCadena(java.lang.String s)
```

Cambia la cadena del último parámetro.

Parameters:

`s` - String con la nueva cadena.

setCadena

```
public void setCadena(java.lang.String s,
            int i)
```

Cambia la cadena del parámetro elegido.

Parameters:

`s` - String con la nueva cadena.

`i` - entero con la posición del parámetro en la línea.

getCadena

```
public java.lang.String getCadena()
```

Devuelve la cadena delúultimo parametro de la línea.

Returns:

devuelve un String con la última cadena de la lista.

getCadena

```
public java.lang.String getCadena(int i)
```

Devuelve la cadena de un parámetro en una posición determinada.

Parameters:

`i` - entero con la posición del parámetro en la línea.

Returns:

devuelve un String con la cadena elegida.

getTab

```
public int getTab()
```

Devuelve el número de tabulaciones de la línea.

Returns:

devuelve un entero con el número de tabulaciones.

setTab

```
public void setTab(int nTab)
```

Cambia el número de tabulaciones.

Parameters:

`nTab` - entero con el nuevo número de tabulaciones.

com.alfonsomolina.tfg

Class ParserJSON

java.lang.Object
    com.alfonsomolina.tfg.ParserJSON

```
public class ParserJSON
extends java.lang.Object
```

Lee los archivos JSON para obtener información acerca del nivel.

Constructor Summary

Constructors

Constructor and Description
`ParserJSON(Context ctx, java.lang.String ciudad)` Crea un objeto JSONObject a partir del archivo JSON de una ciudad.
`ParserJSON(Context ctx, java.lang.String nombre, int etapa)` Constructor.

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`dialogo()` Devuelve "true" si la etapa es solo un dialogo, sin escritura de codigo.
`boolean`	`esAleatorio(int i)` Indica si el sprite tiene una posición aleatoria.
`int[]`	`getAleatorio()` Devuelve las posciones en las que puede aparecer un sprite con carácter aleatorio.
`int`	`getAvatar()` Devuelve el avatar del personaje que plantea el nivel.
`java.lang.String[]`	`getAyuda(java.lang.String s)` Devuelve la lista de ayudas del nivel.
`int`	`getBaldosaBase()` Devuelve la baldosa base del suelo.
`int`	`getBaldosaBmp(int i)` Devuelve el identificador de la imagen de una baldosa que es diferente a la baldosa base.
`int`	`getBaldosaPosicion(int i)` Devuelve la posicion de una baldosa que es diferente a la baldosa base.
`java.lang.String`	`getCiudad(java.lang.String s)` Devuelve el identificador de la ciudad que está esa dirección en el mapa.
`int`	`getCiudadMiniatura()` Devuelve el identificador de la miniatura del mapa.
`int`	`getExperiencia()` Devuelve la experienca ganada en el nivel.
`java.lang.String`	`getExtra()`

java.lang.String	**getExtra**() Devuelve el identificador del fondo de la etapa.
int	**getFondo**() Devuelve el identificador del fondo de la etapa.
java.lang.String	**getID**(int i) Devuelve el identificador del sprite.
java.lang.String	**getInfoNivel**(java.lang.String s) Devuelve la descripción del nivel y sus objetivos.
int	**getInfoTablero**(java.lang.String s) Devuelve las medidas del tablero.
int	**getItemBmp**(int i) Devuelve el identificador de la imagen del sprite.
int	**getItemDireccion**(int i) Devuelve la dirección a la que está mirando inicialmente el sprite.
int	**getItemFuerza**(int i) Devuelve la fuerza del sprite.
int	**getItemPosicion**(int i) Devuelve la posición del sprite en el tablero.
int	**getItemSalud**(int i) Devuelve la salud máxima del sprite.
int	**getItemTipo**(int i) Devuelve el tipo del sprite.
java.util.ArrayList<**Codigo**>	**getLineasCodigo**() Devuelve las líneas de código de muestra del nivel.
java.lang.String[]	**getListaNiveles**() Muestra los niveles disponibles que hay en cada ciudad.
java.lang.String	**getMensaje**(int i) Devuelve un mensaje concreto.
java.lang.String	**getNivel**(java.lang.String nombre, java.lang.String s) Devuelve, del nivel elegido, la descripción o el id.
java.lang.String	**getNombreCiudad**() Devuelve el nombre de la ciudad.
int	**getNumeroConstantes**() Devuelve el número de constantes desbloqueadas.
int	**getNumeroVariables**() Devuelve el número de tipos de variables desbloqueados.
java.util.ArrayList<int[]>	**getObligatorio**() Devuelve las líneas de código que es obligatorio escribir.
int	**getPersona**(int i) Devuelve el identificador de la animación de la persona que dice un mensaje determinado.
boolean	**getPosicion**(int i) Devuelve la posición del personaje que dice el mensaje.
int	**getPosicionInicial**() Devuelve la linea de código inicial para escribir.
boolean	**getSig**(int i) Indica si el siguiente mensaje debe monstrarse automaticamente.
int[][]	**getTeclas**() Devuelve las teclas a desbloquear en el nivel.
int	**getVictoria**() Devuelve el tipo de victoria que le corresponde al nivel.
boolean	**haySiguienteNivel**() Indica si hay más etapas o es la última.
java.lang.String	**leerFichero**(java.lang.String res) Convierte el fichero en una String.

boolean	**setSemilla**(int semilla) Configura una semilla para los elementos aleatorios.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

ParserJSON

```
public ParserJSON(Context ctx,
          java.lang.String nombre,
          int etapa)
```

Constructor. Crea un objeto JSONObject a partir del archivo JSON de un nivel.

Parameters:

`ctx` - contexto de la aplicación.

`nombre` - string con el identificador del nivel.

`etapa` - entero con el número de etapa.

ParserJSON

```
public ParserJSON(Context ctx,
          java.lang.String ciudad)
```

Crea un objeto JSONObject a partir del archivo JSON de una ciudad.

Parameters:

`ctx` - contexto de la aplicación.

`ciudad` - string con el identificador de la ciudad.

Method Detail

leerFichero

```
public java.lang.String leerFichero(java.lang.String res)
```

Convierte el fichero en una String.

Parameters:

`res` - cadena con el identificador del fichero.

Returns:

devuelve un String.

setSemilla

```
public boolean setSemilla(int semilla)
```

Configura una semilla para los elementos aleatorios.

Parameters:

`semilla` - una semilla, entre 0 y el número de elementos aleatorios.

Returns:

devuelve "true" si se ha configurado una semilla y "false" si ya se han probado todas las combinaciones

getCiudadMiniatura

```
public int getCiudadMiniatura()
```

Devuelve el identificador de la miniatura del mapa.

Returns:

devuelve un entero con el identificador de la miniatura del mapa.

getNombreCiudad

```
public java.lang.String getNombreCiudad()
```

Devuelve el nombre de la ciudad.

Returns:

devuelve un String con el nombre de la ciudad.

getCiudad

```
public java.lang.String getCiudad(java.lang.String s)
```

Devuelve el identificador de la ciudad que está esa dirección en el mapa. Devuelve "" si no hay ninguna.

Parameters:

`s` - String con la dirección cardinal elegida.

Returns:

devuelve un String con el identificador de la ciudad o "" si no hay ninguna.

getListaNiveles

```
public java.lang.String[] getListaNiveles()
```

Muestra los niveles disponibles que hay en cada ciudad.

Returns:

devuelve un array con los nombres de los niveles.

getNivel

```
public java.lang.String getNivel(java.lang.String nombre,
                        java.lang.String s)
```

Devuelve, del nivel elegido, la descripción o el id.

Parameters:

`nombre` - nombre de la mision.

`s` - dato a obtener ("descripcion" o "id")

Returns:

devuelve un String con la descripción o el identificador del nivel.

getFondo

```
public int getFondo()
```

Devuelve el identificador del fondo de la etapa. Se mostrará cuando los personajes hablen, en IntroduccionActivity.

Returns:

devuelve un entero con el identificador del fondo de la etapa.

getPersona

```
public int getPersona(int i)
```

Devuelve el identificador de la animación de la persona que dice un mensaje determinado. Si es la misma animación que el mensaje anterior de esa persona, devuelve 0. Devuelve -1 si no hay mas mensajes.

Parameters:

`i` - entero con el número del mensaje.

Returns:

devuelve un entero con el idenficador, 0 o -1.

getPosicion

```
public boolean getPosicion(int i)
```

Devuelve la posición del personaje que dice el mensaje.

Parameters:

`i` - entero con el número del mensaje.

Returns:

devuelve "true" si está a la izquierda y "false" si está a la derecha.

getMensaje

```
public java.lang.String getMensaje(int i)
```

Devuelve un mensaje concreto.

Parameters:

`i` - entero con el número del mensaje.

Returns:

devuelve un string con el texto que se dice.

getSig

```
public boolean getSig(int i)
```

Indica si el siguiente mensaje debe monstrarse automaticamente.

Parameters:

`i` - entero con el número del mensaje.

Returns:

devuelve "true" si hay que mostrar el siguiente mensaje.

dialogo

```
public boolean dialogo()
```

Devuelve "true" si la etapa es solo un dialogo, sin escritura de codigo.

Returns:

devuelve "true" si la etapa es un dialogo.

getExperiencia

```
public int getExperiencia()
```

Devuelve la experienca ganada en el nivel.

Returns:

devuelve un entero con la experiencia ganada.

getAvatar

```
public int getAvatar()
```

Devuelve el avatar del personaje que plantea el nivel.

Returns:

devuelve un entero con una imagen para el avatar.

getInfoNivel

```
public java.lang.String getInfoNivel(java.lang.String s)
```

Devuelve la descripción del nivel y sus objetivos.

Parameters:

`s` - string con la información a obtener ("resumen", "objetivo" u objetivo "secundario")

Returns:

devuelve una cadena con la información requerida.

getVictoria

```
public int getVictoria()
```

Devuelve el tipo de victoria que le corresponde al nivel. Esta puede ser llegar a la meta, eliminar a todos los enemigos, usar unas determinadas líneas de código o una combinación de varias.

Returns:

devuelve un entero que identifica el tipo de victoria.

getInfoTablero

```
public int getInfoTablero(java.lang.String s)
```

Devuelve las medidas del tablero.

Parameters:

`s` - String con la medida a obtener ("filas" o "columnas").

Returns:

devuelve un entero con la información requerida.

getBaldosaBase

```
public int getBaldosaBase()
```

Devuelve la baldosa base del suelo. Esta será la baldosa mas común.

Returns:

devuelve un entero con el identificador al drawable de la baldosa.

getBaldosaPosicion

```
public int getBaldosaPosicion(int i)
```

Devuelve la posicion de una baldosa que es diferente a la baldosa base.

Parameters:

`i` - entero con el número de la baldosa.

Returns:

devuelve un entero con la posición la baldosa.

getBaldosaBmp

`public int getBaldosaBmp(int i)`

Devuelve el identificador de la imagen de una baldosa que es diferente a la baldosa base.

Parameters:

`i` - entero con el número de la baldosa.

Returns:

devuelve un entero con el identificador del drawable de la baldosa.

getItemBmp

`public int getItemBmp(int i)`

Devuelve el identificador de la imagen del sprite.

Parameters:

`i` - entero con el número del sprite en la lista de elementos en el tablero.

Returns:

devuelve un entero con el identificador del drawable del sprite.

getItemPosicion

`public int getItemPosicion(int i)`

Devuelve la posición del sprite en el tablero. Puede ser un valor aleatorio de entre una serie de posiciones válidas.

Parameters:

`i` - entero con el número del sprite en la lista de elementos en el tablero.

Returns:

devuelve un entero con la posición en el tablero.

getItemTipo

`public int getItemTipo(int i)`

Devuelve el tipo del sprite. Puede ser objeto, bestia, npc, roca, etc.

Parameters:

`i` - entero con el número del sprite en la lista de elementos en el tablero.

Returns:

devuelve un entero que identifica el tipo de sprite.

getID

`public java.lang.String getID(int i)`

Devuelve el identificador del sprite. Permite programar diferentes comportamiento en un mismo tipo de sprite.

Parameters:

`i` - entero con el numero del sprite en la lista de elementos en el tablero.

Returns:

devuelve un String con el id particular.

getItemDireccion

```
public int getItemDireccion(int i)
```

Devuelve la dirección a la que está mirando inicialmente el sprite.

Parameters:

`i` - entero con el número del sprite en la lista de elementos en el tablero.

Returns:

devuelve un entero con la dirección a la que mira el sprite.

getItemSalud

```
public int getItemSalud(int i)
```

Devuelve la salud máxima del sprite.

Parameters:

`i` - entero con el número del sprite en la lista de elementos en el tablero.

Returns:

devuelve un entero con la salud.

getItemFuerza

```
public int getItemFuerza(int i)
```

Devuelve la fuerza del sprite.

Parameters:

`i` - entero con el número del sprite en la lista de elementos en el tablero.

Returns:

devuelve un entero con la fuerza.

esAleatorio

```
public boolean esAleatorio(int i)
```

Indica si el sprite tiene una posición aleatoria.

Parameters:

`i` - entero con el número del sprite en la lista de elementos en el tablero.

Returns:

devuelve "true" si el sprite tiene una posición aleatoria.

getAleatorio

```
public int[] getAleatorio()
```

Devuelve las posciones en las que puede aparecer un sprite con carácter aleatorio.

Returns:

devuelve un array de enteros con las posiciones que pueden tener sprites aleatorios.

getExtra

```
public java.lang.String getExtra()
```

getAyuda

```
public java.lang.String[] getAyuda(java.lang.String s)
```

Devuelve la lista de ayudas del nivel. Devuelve los nombres, a partir de los cuales se podra acceder a las ayudas de la carpeta /res/strings.

Parameters:

s - string con el tipo de ayuda ("ayuda" o "ayuda_extra").

Returns:

devuelve un array de string con los nombres de las ayudas.

haySiguienteNivel

```
public boolean haySiguienteNivel()
```

Indica si hay más etapas o es la última.

Returns:

devuelve "true" si hay más etapas y "false" si es la última.

getTeclas

```
public int[][] getTeclas()
```

Devuelve las teclas a desbloquear en el nivel.

Returns:

devuelve un array bidimensional de enteros con las teclas a desbloquear. La primera columna indica el teclado, la segunda la tecla.

getLineasCodigo

```
public java.util.ArrayList<Codigo> getLineasCodigo()
```

Devuelve las líneas de código de muestra del nivel.

Returns:

devuelve una lista inteligente de Codigo, con las líneas de muestra.

getPosicionInicial

```
public int getPosicionInicial()
```

Devuelve la linea de código inicial para escribir.

Returns:

devuelve un entero con la línea de código inicial.

getObligatorio

```
public java.util.ArrayList<int[]> getObligatorio()
```

Devuelve las líneas de código que es obligatorio escribir.

Returns:

devuelve una lista inteligente con las líneas de código que hay que escribir, cada una de ellas compuesta por un arrays de enteros, con los códigos que hay en cada línea.

getNumeroConstantes

```
public int getNumeroConstantes()
```

Devuelve el número de constantes desbloqueadas.

Returns:

devuelve un entero con las constantes desbloqueadas.

getNumeroVariables

```
public int getNumeroVariables()
```

Devuelve el número de tipos de variables desbloqueados.

Returns:

devuelve un entero con los tipos de variables desbloqueados.

com.alfonsomolina.tfg

Class Tablero

java.lang.Object
    SurfaceView
        com.alfonsomolina.tfg.Tablero

```
public class Tablero
extends SurfaceView
```

Dibuja el tablero y los personajes y controla las interacciones entre ellos.

Constructor Summary

Constructors

Constructor and Description
`Tablero(Context context)`
`Tablero(Context context, ParserJSON parserJSON, boolean muestra)` Constructor.

Method Summary

Methods

Modifier and Type	Method and Description
`void`	`actualizar()` Actualiza la posición de todos los sprites y comprueba si interactúan entre ellos.
`void`	`actualizarIA()` Actualiza el comportamiento de los personajes no controlados por el jugador.
`void`	`actualizarOffset()` Calcula cuanto se mueve el tablero para centrarlo en la pantalla.
`void`	`centrarDibujo()` Centra cada sprite en el centro de su baldosa.
`void`	`coger()` Coge el objeto en la dirección que esté mirando Ada y lo equipa.
`void`	`coger(int direccion)` Coge el objeto en la dirección seleccionada y lo equipa.
`protected void`	`dibujar(Canvas canvas)` Dibuja el tablero en el canvas.
`protected void`	`dibujarTablero(Canvas canvas)` Dibuja las baldosas del tablero.
`void`	`eliminar(int x, int y)` Elimina el Sprite que haya en la coordenada seleccionada.
`int`	`escuchar()` Devuelve un entero con lo que se ha escuchado.
`Sprite`	`getAda()` Devuelve el Sprite de Ada.
`int`	`getAlturaBaldosa()` Devuelve la altura de cada baldosa.

int	**getAnchuraBaldosa**() Devuelve la anchura de cada baldosa.
int	**getBaldosa_offset**() Devuelve el desfase de cada baldosa.
boolean	**getBestias**() Indica si se ha derrotado a todos los enemigos del tablero.
int	**getColumnas**() Devuelve el número de columnas del tablero.
int	**getFilas**() Devuelve el número de filas del tablero.
java.lang.String	**getLog**() Devuelve el log que se ha generado.
boolean	**getMeta**() Indica si se ha alcanzado la meta.
int	**getXoffset**() Devuelve cuánno hay que mover el tablero en la cooredanad X para centrarlo.
int	**getYoffset**() Devuelve cuánto hay que mover el tablero en la cooredanad Y para centrarlo.
int	**mirar**(int x, int y, boolean log) Devuelve lo que hay en una posicion determinada en el tablero.
void	**setLog**(java.lang.String s) Modifica el log introduciendo una nueva entrada.
void	**setVelocidadRapida**() Acelera la animacion, haciendo que los sprites no tarden en moverse.
void	**usar**() Usa el objeto equipado.
void	**usar**(int direccion) Usa el objeto equipado en la dirección seleccionada.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Tablero

```
public Tablero(Context context)
```

Tablero

```
public Tablero(Context context,
       ParserJSON parserJSON,
       boolean muestra)
```

Constructor. Crea el tablero a partir de los datos en el JSON.

Parameters:

`context` - contexto de la actividad

`parserJSON` - parser para leer los datos del archivo JSON.

`muestra` - boolean que vale "true" si es un tablero de muestra y los sprites aleatorios deben ser transparentes.

Method Detail

dibujar

```
protected void dibujar(Canvas canvas)
```

Dibuja el tablero en el canvas. Primero actualiza todos los Sprites y después los dibuja.

Parameters:

`canvas` - Canvas en el que dibujar.

actualizar

```
public void actualizar()
```

Actualiza la posición de todos los sprites y comprueba si interactúan entre ellos.

actualizarIA

```
public void actualizarIA()
```

Actualiza el comportamiento de los personajes no controlados por el jugador.

dibujarTablero

```
protected void dibujarTablero(Canvas canvas)
```

Dibuja las baldosas del tablero.

Parameters:

`canvas` - canvas donde dibujar.

actualizarOffset

```
public void actualizarOffset()
```

Calcula cuanto se mueve el tablero para centrarlo en la pantalla.

mirar

```
public int mirar(int x,
        int y,
        boolean log)
```

Devuelve lo que hay en una posicion determinada en el tablero.

Parameters:

`x` - entero con la coordenada x.

`y` - entero con la coordenada y.

`log` - boolean que vale "true" si hay que mostrar en el log lo que se ha visto.

Returns:

devuelve un entero con un número que representa lo que había en la baldosa seleccionada.

eliminar

```
public void eliminar(int x,
            int y)
```

Elimina el Sprite que haya en la coordenada seleccionada.

Parameters:

`x` - entero con la coordenada x.

`y` - entero con la coordenada y.

coger

```
public void coger()
```

Coge el objeto en la dirección que esté mirando Ada y lo equipa.

coger

```
public void coger(int direccion)
```

Coge el objeto en la dirección seleccionada y lo equipa.

Parameters:

`direccion` - entero con la dirección seleccionada

usar

```
public void usar()
```

Usa el objeto equipado.

usar

```
public void usar(int direccion)
```

Usa el objeto equipado en la dirección seleccionada.

Parameters:

`direccion` - entero con la dirección seleccionada.

escuchar

`public int escuchar()`

Devuelve un entero con lo que se ha escuchado. Cambia según el nivel.

Returns:

devuelve un entero con información acerca del nivel.

centrarDibujo

`public void centrarDibujo()`

Centra cada sprite en el centro de su baldosa.

getBaldosa_offset

`public int getBaldosa_offset()`

Devuelve el desfase de cada baldosa.

Returns:

devuelve un entero con el desfase de cada baldosa.

getYoffset

`public int getYoffset()`

Devuelve cuánto hay que mover el tablero en la cooredanad Y para centrarlo.

Returns:

devuelve un entero con el desfase en la coordenada Y.

getXoffset

`public int getXoffset()`

Devuelve cuánno hay que mover el tablero en la cooredanad X para centrarlo.

Returns:

devuelve un entero con el desfase en la coordenada X.

getFilas

`public int getFilas()`

Devuelve el número de filas del tablero.

Returns:

devuelve un entero con el número de filas del tablero.

getColumnas

`public int getColumnas()`

Devuelve el número de columnas del tablero.

Returns:

devuelve un entero con el número de columnas del tablero.

getMeta

`public boolean getMeta()`

Indica si se ha alcanzado la meta.

Returns:

devuelve un boolean que vale "true" si se ha pisado la meta.

getAlturaBaldosa

`public int getAlturaBaldosa()`

Devuelve la altura de cada baldosa.

Returns:

devuelve un entero con la altura de cada baldosa.

getAnchuraBaldosa

`public int getAnchuraBaldosa()`

Devuelve la anchura de cada baldosa.

Returns:

devuelve un entero con la anchura de cada baldosa.

getLog

`public java.lang.String getLog()`

Devuelve el log que se ha generado.

Returns:

devuelve un String con el log generado.

setLog

`public void setLog(java.lang.String s)`

Modifica el log introduciendo una nueva entrada.

Parameters:

`s` - String con la nueva entrada del log.

setVelocidadRapida

```
public void setVelocidadRapida()
```

Acelera la animacion, haciendo que los sprites no tarden en moverse. Este método se utiliza para probar las combinaciones antes de ejecutar cada nivel.

getAda

```
public Sprite getAda()
```

Devuelve el Sprite de Ada.

Returns:

devuelve el Sprite de Ada.

getBestias

```
public boolean getBestias()
```

Indica si se ha derrotado a todos los enemigos del tablero.

Returns:

devuelve un boolean que vale "true" si se han derrotado a todos los enemigos.

com.alfonsomolina.tfg

Class Sprite

java.lang.Object
    com.alfonsomolina.tfg.Sprite

```
public class Sprite
extends java.lang.Object
```

Almacena toda la información referente a un Sprite y lo dibuja en el tablero.

Field Summary

Fields

Modifier and Type	Field and Description
static int	ABAJO
static int	ADA
static int	ANDAR
static int	ARRIBA
static int	ATACAR
static int	BESTIA
static int	CHOCAR
static int	COCODRILO
static int	DERECHA
static int	DIANA
static int	EXCEPCION
static int	FIJO
static int	IZQUIERDA
static int	META
static int	NPC
static int	OBJETO
static int	QUIETO
static int	ROCA
static int	TEMP

Constructor Summary

Constructors

Constructor and Description
Sprite(Tablero tablero, Bitmap bmp, int x, int y, int dir, int tipo) Constructor.

Method Summary

Methods

Modifier and Type	Method and Description
void	**actualizar**() Cambia el fotograma en el que está el Sprite.
void	**addTempSprite**(Bitmap bmp, int filas, int columnas, int life, boolean ultimo) Cambia la animación del Sprite.
void	**centrarDibujo**() Centra el dibujo en el centro de la baldosa.
boolean	**choca**(**Sprite** s) Comprueba si un Sprite choca con otro en el tablero.
void	**dibujar**(Canvas canvas) Dibuja el Sprite en un canvas.
int	**getAccion**() Devuelve la acción que esta realizando el Sprite.
int	**getColumna**() Devuelve la columna en la que se encuentra el Sprite.
int	**getDireccion**() Devuelve la dirección que orienta el Sprite.
java.lang.String	**getEquipo**() Devuelve el objeto equipado.
int	**getFila**() Devuelve la fila en la que se encuentra el Sprite.
int	**getFuerza**() Devuelve la fuerza del Sprite.
java.lang.String	**getID**() Devuelve el identificador propio del Sprite.
int	**getSalud**() Devuelve la salud del Sprite.
int	**getTipo**() Devuelve el tipo de Sprite.
int	**getVida**() Devuelve los fotogramas que le quedan a la animacion del movimiento.
int	**getX**() Devuelve la posición del canvas del eje X en que se encuentra el Sprite.
int	**getY**() Devuelve la posición del canvas del eje X en que se encuentra el Sprite.
void	**orientar**(int dir) Cambia la dirección en la que se orienta el Sprite.
boolean	**quitarVida**(int puntos) Quita algo de vida al Sprite.
void	**setAccion**(int acc) Cambia la acción que está realizando el sprite.
void	**setAccion**(int acc, int dir) Cambia la acción que está realizando el Sprite y la dirección en la que la realizará.
void	**setAlpha**(int alpha) Fija la transparencia del Sprite.
void	**setBmp_medidas**(int bmp_columnas, int bmp_filas, int bmp_acciones) Fija las medidas del bitmap.
void	**setEquipo**(java.lang.String equipo) Fija el equipo del Sprite.
void	**setFuerza**(int fuerza)

	Fija la fuerza del Sprite.
void	**setID**(java.lang.String iD) Fija el identificador propio del Sprite.
void	**setSalud**(int salud) Fija la salud del Sprite.
void	**setVelocidadRapida**() Acelera la animacioó, haciendo que los sprites no tarden en realizar acciones.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Sprite

```
public Sprite(Tablero tablero,
      Bitmap bmp,
      int x,
      int y,
      int dir,
      int tipo)
```

Constructor.

Parameters:

`tablero` - tablero en el que se juega.

`bmp` - Bitmap con los fotogramas del Sprite.

`x` - entero con la coordenada X inicial.

`y` - entero con la coordenada Y inicial.

`dir` - entero con la dirección inicial en la que se orienta.

`tipo` - entero con el tipo de Sprite.

Method Detail

dibujar

```
public void dibujar(Canvas canvas)
```

Dibuja el Sprite en un canvas.

Parameters:

`canvas` - canvas en el que dibujar.

actualizar

```
public void actualizar()
```

Cambia el fotograma en el que está el Sprite. Actualiza su movimiento si es necesario.

setAccion

```
public void setAccion(int acc)
```

Cambia la acción que está realizando el sprite.

Parameters:

`acc` - entero que identifica la nueva acción.

setAccion

```
public void setAccion(int acc,
             int dir)
```

Cambia la acción que está realizando el Sprite y la dirección en la que la realizará.

Parameters:

`acc` - entero que identifica la nueva acción.

`dir` - entero con la dirección en la que se orienta.

addTempSprite

```
public void addTempSprite(Bitmap bmp,
                 int filas,
                 int columnas,
                 int life,
                 boolean ultimo)
```

Cambia la animación del Sprite. Utiliza otra imagen para dibujar el Sprite durante algunos fotogramas. Después de esto vuelve al original o se elimina.

Parameters:

`bmp` - Bitmap con la nueva animación.

`filas` - entero con el número de filas de la imagen.

`columnas` - entero con el número de columnas de la imagen.

`life` - entero con el número de fotogramas que se usará esta imagen.

`ultimo` - boolean que vale "true" si tras terminar esta animación hay que eliminar el Sprite.

centrarDibujo

```
public void centrarDibujo()
```

Centra el dibujo en el centro de la baldosa.

choca

```
public boolean choca(Sprite s)
```

Comprueba si un Sprite choca con otro en el tablero.

Parameters:

`s` - Sprite con el que se comprueba si choca.

Returns:

devuelve "true" si chocan y "false" en caso contrario.

orientar

```
public void orientar(int dir)
```

Cambia la dirección en la que se orienta el Sprite.

Parameters:

`dir` - entero con la nueva dirección.

quitarVida

```
public boolean quitarVida(int puntos)
```

Quita algo de vida al Sprite. Si no queda salud, devuelve true.

Parameters:

`puntos` - entero con los puntos de vida a quitar.

Returns:

devuelve "true" si, tras quitar vida, la salud es igual o inferior a cero.

setAlpha

```
public void setAlpha(int alpha)
```

Fija la transparencia del Sprite.

Parameters:

`alpha` - entero de 0 a 255 con la transparencia.

setFuerza

```
public void setFuerza(int fuerza)
```

Fija la fuerza del Sprite.

Parameters:

`fuerza` - entero con el valor de fuerza.

setSalud

```
public void setSalud(int salud)
```

Fija la salud del Sprite.

Parameters:

`salud` - entero con la salud.

setID

```
public void setID(java.lang.String iD)
```

Fija el identificador propio del Sprite.

Parameters:

`iD` - String con el id.

getID

```
public java.lang.String getID()
```

Devuelve el identificador propio del Sprite.

Returns:

devuelve un String con el identificador del Sprite.

getVida

```
public int getVida()
```

Devuelve los fotogramas que le quedan a la animacion del movimiento.

Returns:

devuelve un entero con los fotogramas que le quedan a la animacion del movimiento.

getColumna

```
public int getColumna()
```

Devuelve la columna en la que se encuentra el Sprite.

Returns:

devuelve un entero con la columna en que se encuentra el Sprite.

getFila

```
public int getFila()
```

Devuelve la fila en la que se encuentra el Sprite.

Returns:

devuelve un entero con la fila en que se encuentra el Sprite.

getX

```
public int getX()
```

Devuelve la posición del canvas del eje X en que se encuentra el Sprite.

Returns:

devuelve un entero con la posición del canvas del eje X en que se encuentra el Sprite.

getY

```
public int getY()
```

Devuelve la posición del canvas del eje X en que se encuentra el Sprite.

Returns:

devuelve un entero con la posición del canvas del eje X en que se encuentra el Sprite.

getTipo

```
public int getTipo()
```

Devuelve el tipo de Sprite.

Returns:

devuelve un entero que identifica el tipo de Sprite.

getSalud

```
public int getSalud()
```

Devuelve la salud del Sprite.

Returns:

devuelve un entero con la salud del Sprite.

setEquipo

```
public void setEquipo(java.lang.String equipo)
```

Fija el equipo del Sprite.

Parameters:

`equipo` - String que identifica el objeto a equipar.

getEquipo

```
public java.lang.String getEquipo()
```

Devuelve el objeto equipado.

Returns:

devuelve un String que identifica el objeto equipado.

getFuerza

```
public int getFuerza()
```

Devuelve la fuerza del Sprite.

Returns:

devuelve un entero con la fuerza del Sprite.

getDireccion

```
public int getDireccion()
```

Devuelve la dirección que orienta el Sprite.

Returns:

devuelve un entero que identifica la dirección que orienta el Sprite.

getAccion

```
public int getAccion()
```

Devuelve la acción que esta realizando el Sprite.

Returns:

devuelve un entero que identifica la acción que esta realizando el Sprite.

setVelocidadRapida

```
public void setVelocidadRapida()
```

Acelera la animacioó, haciendo que los sprites no tarden en realizar acciones. Este metodo se utiliza para probar las combinaciones antes de ejecutar cada nivel.

setBmp_medidas

```
public void setBmp_medidas(int bmp_columnas,
                  int bmp_filas,
                  int bmp_acciones)
```

Fija las medidas del bitmap. Se utiliza para aquellos que no siguen la estructura tipica de su tipo de Sprite.

Parameters:

`bmp_columnas` - entero con las columnas de la imagen (fotogramas por accion).

`bmp_filas` - entero con las filas de la imagen (una o cuatro, una por cada direccion).

`bmp_acciones` - entero con las acciones de la imagen.

com.alfonsomolina.tfg

Class Hilo

java.lang.Object
    com.alfonsomolina.tfg.Hilo

```
public class Hilo
extends
```

Crea un hilo que actualiza el tablero, recordándole cada pocos milisegundos que debe actualizarse y redibujar su contenido.

Constructor Summary

Constructors

Constructor and Description
`Hilo(Tablero tablero)` Constructor.

Method Summary

Methods

Modifier and Type	Method and Description
`protected java.lang.Void`	`doInBackground(java.lang.Void... sinusar)` Un bucle se ejecuta hasta ser cancelado, actualizando el tablero cada poco tiempo.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Hilo

`public Hilo(Tablero tablero)`

Constructor. Construye el hilo para el tablero ya creado.

Parameters:

`tablero` - Tablero que se debe actualizar.

Method Detail

doInBackground

```
protected java.lang.Void doInBackground(java.lang.Void... sinusar)
```

Un bucle se ejecuta hasta ser cancelado, actualizando el tablero cada poco tiempo.

Parameters:

`sinusar` - No necesita parámetros.

Returns:

No devuelve nada.

com.alfonsomolina.tfg

Class Aviso

java.lang.Object
    DialogFragment
        com.alfonsomolina.tfg.Aviso

```
public class Aviso
extends DialogFragment
```

Dialog que solicita una confirmación cuando se pulsa el boton "atrás" dentro de una misión.

Constructor Summary

Constructors

Constructor and Description
`Aviso()`

Method Summary

Methods

Modifier and Type	Method and Description
`Dialog`	`onCreateDialog(Bundle savedInstanceState)` Crea el mensaje de aviso.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Aviso

```
public Aviso()
```

Method Detail

onCreateDialog

```
public Dialog onCreateDialog(Bundle savedInstanceState)
```

Crea el mensaje de aviso.

Parameters:

`savedInstanceState` - Bundle necesario.

Returns:

devuelve el Dialog

com.alfonsomolina.tfg

Class MapaActivity.AvisoCambiarIdioma

java.lang.Object
    DialogFragment
        com.alfonsomolina.tfg.MapaActivity.AvisoCambiarIdioma

Enclosing class:

MapaActivity

```
public static class MapaActivity.AvisoCambiarIdioma
extends DialogFragment
```

Controla el Dialog al cambiar el idioma. Pide una confirmación para hacerlo.

Constructor Summary

Constructors

Constructor and Description
`MapaActivity.AvisoCambiarIdioma()`

Method Summary

Methods

Modifier and Type	Method and Description
`Dialog`	`onCreateDialog(Bundle savedInstanceState)` Crea el mensaje de aviso.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

MapaActivity.AvisoCambiarIdioma

```
public MapaActivity.AvisoCambiarIdioma()
```

Method Detail

onCreateDialog

```
public Dialog onCreateDialog(Bundle savedInstanceState)
```

Crea el mensaje de aviso.

Parameters:

`savedInstanceState` - Bundle necesario.

Returns:

devuelve el Dialog

com.alfonsomolina.tfg

Class MapaActivity.AvisoReinicio

java.lang.Object
    DialogFragment
        com.alfonsomolina.tfg.MapaActivity.AvisoReinicio

Enclosing class:

MapaActivity

```
public static class MapaActivity.AvisoReinicio
extends DialogFragment
```

Controla el Dialog al reiniciar el juego. Pide una confirmación para hacerlo.

Constructor Summary

Constructors

Constructor and Description
`MapaActivity.AvisoReinicio()`

Method Summary

Methods

Modifier and Type	Method and Description
`Dialog`	`onCreateDialog(Bundle savedInstanceState)` Crea el mensaje de aviso.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

MapaActivity.AvisoReinicio

```
public MapaActivity.AvisoReinicio()
```

Method Detail

onCreateDialog

```
public Dialog onCreateDialog(Bundle savedInstanceState)
```

Crea el mensaje de aviso.

Parameters:

`savedInstanceState` - Bundle necesario.

Returns:

devuelve el Dialog

com.alfonsomolina.tfg

Class ListaModificadaAdapter

java.lang.Object
    ArrayAdapter
        com.alfonsomolina.tfg.ListaModificadaAdapter

```
public class ListaModificadaAdapter
extends ArrayAdapter
```

Adaptador creado para las diferentes listas. Tiene un comportamiento diferente según la naturaleza de la lista.

Constructor Summary

Constructors

Constructor and Description
`ListaModificadaAdapter(Context context, int resourceID)` Constructor.

Method Summary

Methods

Modifier and Type	Method and Description
`void`	`add(ElementoModificado e)` Añade un nuevo elemento al final de la lista.
`void`	`add(int posicion, java.lang.String s)` Añade un nuevo elemento en una posicion concreta de la lista.
`void`	`add(java.lang.String texto)` Añade un nuevo elemento al final de la lista.
`int`	`getCount()` Devuelve el número de filas en la lista.
`java.lang.Object`	`getItem(int position)` Devuelve el elemento de la lista de la posición elegida.
`View`	`getView(int position, View convertView, ViewGroup parent)` Crea la vista del elemento de la lista.
`void`	`remove(int posicion)` Elimina una fila de la lista.
`void`	`setFlag(int position, boolean b)` Cambia la bandera de un elemento de la lista.
`void`	`setText(int position, java.lang.String s)` Cambia el texto de un elemento de la lista

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

ListaModificadaAdapter

```
public ListaModificadaAdapter(Context context,
                      int resourceID)
```

Constructor.

Parameters:

`context` - contexto de la actividad.

`resourceID` - recurso con el layout de cada fila.

Method Detail

getCount

```
public int getCount()
```

Devuelve el número de filas en la lista.

Returns:

devuelve un entero con el número de elementos en la lista.

getItem

```
public java.lang.Object getItem(int position)
```

Devuelve el elemento de la lista de la posición elegida.

Parameters:

`position` - entero con la posición del elemento en la lista.

Returns:

devuelve el objeto en la fila elegida.

getView

```
public View getView(int position,
           View convertView,
           ViewGroup parent)
```

Crea la vista del elemento de la lista.

Parameters:

`position` - entero con la posición del elemento en la lista.

`convertView` - vista de la fila.

`parent` - vista de la lista padre.

Returns:

devuelve la fila modificada.

add

```
public void add(java.lang.String texto)
```

Añade un nuevo elemento al final de la lista.

Parameters:

`texto` - String con el texto del nuevo elemento.

add

```
public void add(ElementoModificado e)
```

Añade un nuevo elemento al final de la lista.

Parameters:

`e` - ElementoModificado con el nuevo elemento a introducir.

add

```
public void add(int posicion,
       java.lang.String s)
```

Añade un nuevo elemento en una posicion concreta de la lista.

Parameters:

`posicion` - entero con la posición del nuevo elemento

`s` - String con el texto del nuevo elemento.

remove

```
public void remove(int posicion)
```

Elimina una fila de la lista.

Parameters:

`posicion` - entero con la posición del elemento a eliminar

setFlag

```
public void setFlag(int position,
           boolean b)
```

Cambia la bandera de un elemento de la lista.

Parameters:

`position` - entero con la posición del elemento en la lista.

`b` - boolean con el nuevo valor de la bandera.

setText

```
public void setText(int position,
           java.lang.String s)
```

Cambia el texto de un elemento de la lista

Parameters:

`position` - entero con la posición del elemento en la lista.

`s` - String con el nuevo texto.

com.alfonsomolina.tfg

Class ElementoModificado

java.lang.Object
    com.alfonsomolina.tfg.ElementoModificado

```
public class ElementoModificado
extends java.lang.Object
```

Clase con el objeto contenido en las diferentes listas a usar. Tiene dos atributos, "texto" y "flag", que varían según la lista utilizada.

Constructor Summary

Constructors

Constructor and Description
`ElementoModificado(boolean flag, java.lang.String texto)` Construye la clase con el texto a mostrar y la bandera elegida.
`ElementoModificado(java.lang.String texto)` Constructor.

Method Summary

Methods

Modifier and Type	Method and Description
`java.lang.String`	`getTexto()` Devuelve el texto.
`boolean`	`isFlagged()` Devuelve la bandera.
`void`	`setFlag(boolean flag)` Modifica la bandera por una nueva.
`void`	`setTexto(java.lang.String texto)` Modifica el texto por uno nuevo.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

ElementoModificado

```
public ElementoModificado(java.lang.String texto)
```

Constructor. Construye la clase con el texto a mostrar y la bandera desactivada.

Parameters:

`texto` - String con el texto.

ElementoModificado

```
public ElementoModificado(boolean flag,
                  java.lang.String texto)
```

Construye la clase con el texto a mostrar y la bandera elegida.

Parameters:

`flag` - boolean con la bandera.

`texto` - String con el texto.

Method Detail

getTexto

```
public java.lang.String getTexto()
```

Devuelve el texto.

Returns:

devuelve un String con el texto.

setTexto

```
public void setTexto(java.lang.String texto)
```

Modifica el texto por uno nuevo.

Parameters:

`texto` - String con el texto

isFlagged

```
public boolean isFlagged()
```

Devuelve la bandera.

Returns:

boolean con la bandera.

setFlag

```
public void setFlag(boolean flag)
```

Modifica la bandera por una nueva.

Parameters:

`flag` - boolean con la bandera.

com.alfonsomolina.tfg

Class Variable

java.lang.Object
    com.alfonsomolina.tfg.Variable

```
public class Variable
extends java.lang.Object
```

Clase para guarda las variables que se hayan creado en la ejecución. Tiene diferentes valores para cada tipo de variable diferente.

Constructor Summary

Constructors

Constructor and Description
`Variable(int tipo, java.lang.String nombre)` Constructor.

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`getBoolean()` Devuelve el valor boolean de la variable.
`int`	`getInt()` Devuelve el valor entero de la variable.
`java.lang.String`	`getNombre()` Devuelve el nombre de la variable.
`int`	`getTipo()` Devuelve el tipo de la variable.
`void`	`setBoolean(boolean valorBoolean)` Fija el valor boolean de la variable.
`void`	`setInt(int valorInt)` Fija el valor entero de la variable.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Variable

```
public Variable(int tipo,
        java.lang.String nombre)
```

Constructor. Crea una variable con un tipo y nombre concretos.

Parameters:

`tipo` - entero que indica el tipo de la variable.

`nombre` - String con el nombre de la variable.

Method Detail

getInt

```
public int getInt()
```

Devuelve el valor entero de la variable.

Returns:

devuelve un entero con el valor de la variable.

setInt

```
public void setInt(int valorInt)
```

Fija el valor entero de la variable.

Parameters:

`valorInt` - entero con el valor de la variable.

getBoolean

```
public boolean getBoolean()
```

Devuelve el valor boolean de la variable.

Returns:

devuelve un boolean con el valor de la variable.

setBoolean

```
public void setBoolean(boolean valorBoolean)
```

Fija el valor boolean de la variable.

Parameters:

`valorBoolean` - boolean con el valor de la variable.

getTipo

```
public int getTipo()
```

Devuelve el tipo de la variable.

Returns:

devuelve un entero con el tipo de la variable.

getNombre

```
public java.lang.String getNombre()
```

Devuelve el nombre de la variable.

Returns:

devuelve un String con el nombre de la variable.