

*Paula Madrid Alonso, 100406260*  
*Delia Sánchez Gómez, 100383339*  
*Paula Capel Silveiro, 100383474*



Universidad  
Carlos III de Madrid

# APLICACIONES MÓVILES

## MEMORIA FINAL PROYECTO: MI NEVERA

GRUPO 6

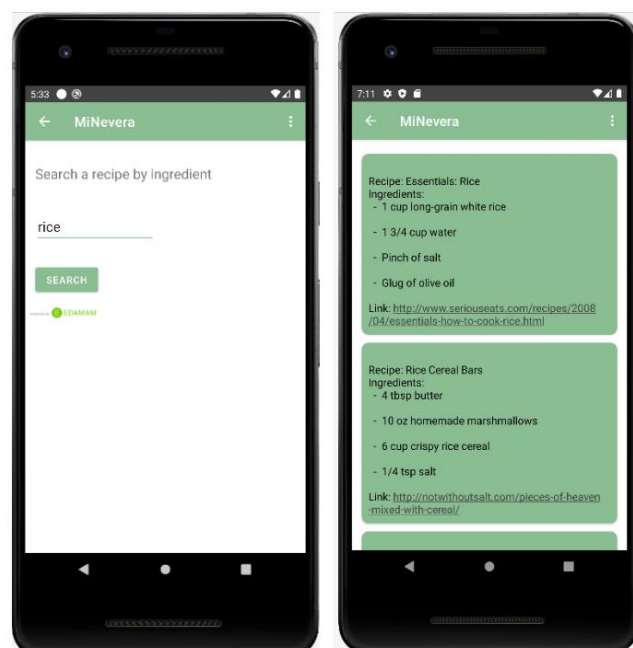


La aplicación MiNevera está diseñada para ayudar a los usuarios que la utilicen a reducir sus desechos controlando la fecha de caducidad de los productos de su nevera. Las funcionalidades utilizadas en la aplicación son las siguientes:

- La actividad *MainActivity* es la actividad principal y muestra una lista con los productos que están guardados en la base de datos. También incluye botones que llevan al resto de funcionalidades.



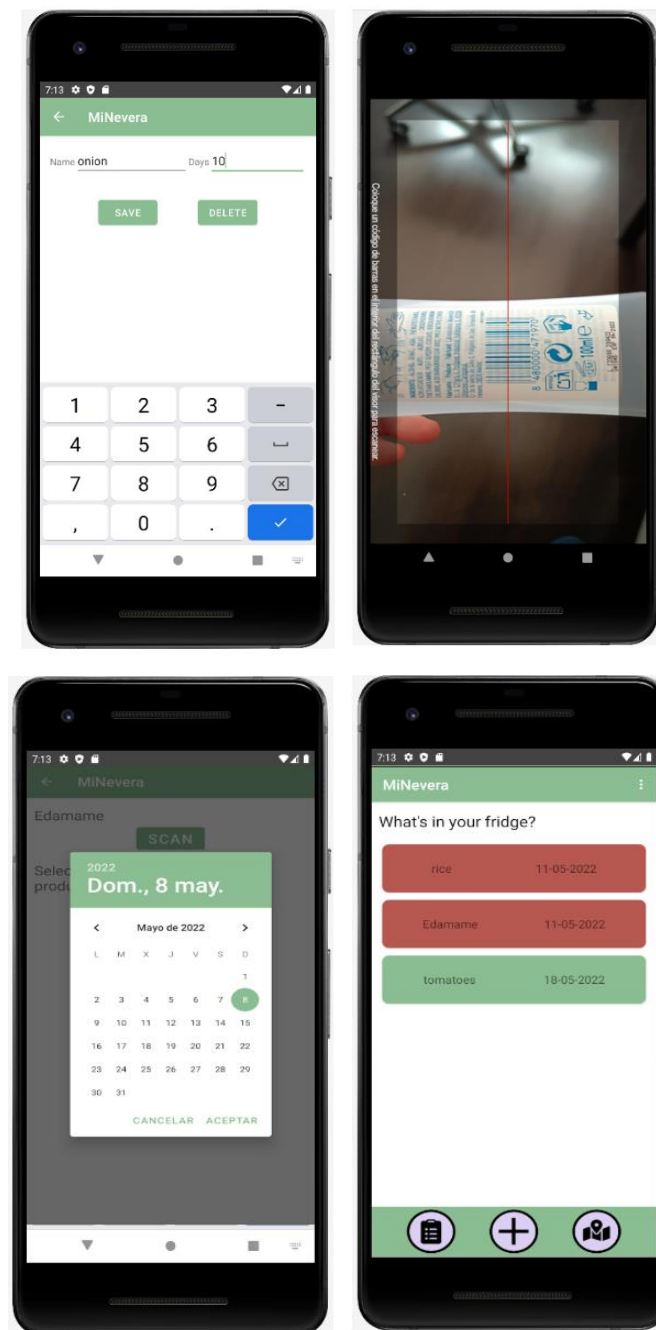
- La funcionalidad de búsqueda de recetas se implementa en 2 actividades, *SearchIngr*, en la que se introduce el nombre de un ingrediente, y *Recipe* en la que se hace una búsqueda en una base de datos de recetas gracias a la API de EDAMAM Recipes.



- La base de datos se crea en la actividad *dbProducts* y los productos se añaden a ella de dos maneras diferentes: manualmente gracias a la actividad *AddProducts* y

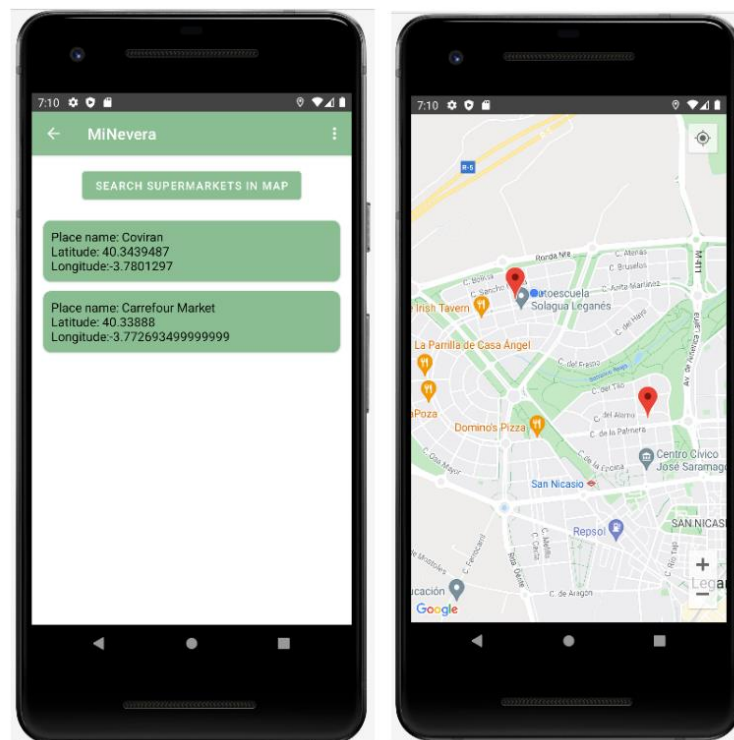
escaneando el código de barras con la cámara del dispositivo utilizando la actividad *Camara*. Dentro de esta última actividad se usa un calendario interactivo que se crea en la clase *DatePickerFragment*.

Para la funcionalidad de añadir productos escaneando un código de barras se implementan las APIs *Zxing* y *OpenFoodFacts*. *Zxing* se usa para procesar el código de barras capturado con la cámara como un *String* de números. A *OpenFoodFacts* se accede a través de una URL que se pide con terminación *.json* para buscar el nombre del producto.






- La funcionalidad de la búsqueda de supermercados cercanos al usuario se realiza con las siguientes actividades: *Map* en la que se accede a la localización del usuario y se

hace una búsqueda en la API *GooglePlaces* de los supermercados cercanos y *MapMarkers* en la que se infla un mapa gracias a la API SDK de *Maps* y se colocan marcadores de los supermercados encontrados.



- Además se usa el servicio *NotificationService* para avisar al usuario por medio de notificaciones de cuál es el producto en su nevera que va a caducar antes.

<p> <b>MiNevera</b> • 18:00</p> <p><b>Something in your fridge is about to go bad!</b></p> <p>The product Edamame expires today.</p>
<p> <b>MiNevera</b> • 18:01</p> <p><b>Something in your fridge is about to go bad!</b></p> <p>The product Edamame has 1 day left before it expires.</p>
<p> <b>MiNevera</b> • 18:06</p> <p><b>Something in your fridge is about to go bad!</b></p> <p>The product carrot has 2 days left before it expires.</p>

El trabajo que se ha llevado a cabo durante la creación de esta aplicación se ha dividido de la siguiente manera entre los integrantes del grupo 6:

Paula Madrid Alonso	Creación del AppBar, implementación de la funcionalidad de búsqueda de recetas a partir de un ingrediente y de la funcionalidad de búsqueda de supermercados en un mapa a partir
---------------------	--

	de la localización del usuario. Optimización de código.
Delia Sánchez Gómez	Creación y gestión de la base de datos, implementación de la pantalla principal, ajustes con preferencias compartidas y navegación entre las diferentes plantillas. Optimización de código.
Paula Capel Silveiro	Implementación del servicio de notificaciones, implementación de la barra inferior de la main activity, implementación de funcionalidad de añadir productos escaneando el código de barras. Optimización de código.

### Consideraciones a tener en cuenta para probar la aplicación:

Los emuladores empleados para probar la aplicación han sido Pixel 2, con API 29 y 30.

En dispositivos reales se ha comprobado con OnePlus Nord 2 (Android 11), y LG Q60 (Android 10).

A continuación se comenta el código con más detalle, pasando por todas las funcionalidades previamente mencionadas.

### Pantalla principal (Main Activity):

Esta pantalla será la principal de la aplicación. Es lo primero que se ve una vez se inicia la aplicación y muestra la lista de productos almacenados en la nevera, que emplea un código de colores para mostrar el número de días que faltan para que se alcance la fecha de caducidad. Esta lista se construye a partir de la base de datos de la aplicación, cuyo funcionamiento será explicado con mayor profundidad en apartados posteriores.

```
private void fillData() throws ParseException { //Recorre la BBDD y
coloca los productos uno por uno, determinando la diferencia de días y el
color de cada uno
    Context context = getApplicationContext();
    SharedPreferences prefs =
PreferenceManager.getDefaultSharedPreferences(context);
    boolean notificationsOff = prefs.getBoolean("switch", false); //Se
recoge el estado del switch
    productList = new ArrayList<ProductObject>();
    Cursor notesCursor = dbAdapter.fetchAllProducts();
    int count = 0;
    Cursor aux = notesCursor;
    if(aux.getCount() != 0) {
        aux.moveToFirst();
        while (count < aux.getCount()) {
            count++;
            Integer i = aux.getInt(0); // Recoge id
            String n = aux.getString(1); // Recoge name
            String d = aux.getString(2); // Recoge days
            dbAdapter.updateDifference(i, n, d); // Actualiza diff
            String diff_days = aux.getString(3); // Recoge diff
            if(count == 1) {
                if (!notificationsOff) {
                    startNotifications(n, diff_days); // Lanza
```

```

    notificación
        }
    }
    if(Integer.parseInt(diff_days)<0){
        dbAdapter.deleteProduct(i); // Borra si se ha pasado
        aux.moveToNext();
        continue;
    }
    ProductObject product = new ProductObject(
        Integer.toString(i), n, d, diff_days);
    productList.add(product);
    aux.moveToNext();
}
aux.close();
}
CardAdapter adapter = new CardAdapter(this,productList);
p_listview.setAdapter(adapter);
}

```

Gracias al método *fillData*, se recogen todos los productos contenidos en la base de datos y se evalúan uno a uno, comprobando los valores de cada producto y actualizando la diferencia entre la fecha actual y la fecha de caducidad guardada. Como los productos se evalúan individualmente, se podrá colocar una tarjeta de un color u otro dependiendo del número de días restantes. Esta característica se explicará a continuación.

Tras haber extraído los valores de un producto, se crea un objeto de tipo producto para ser almacenado en una *ArrayList*. El uso de esta clase permitirá crear un adaptador para agregar elementos en la lista con un *layout* específico. El adaptador se crea en la clase *CardAdapter.java*, recoge el producto y le asigna el *layout* correspondiente.

Una vez todos los productos han sido añadidos a la *ArrayList*, se asignarán a la *ListView* para que puedan ser mostrados en la actividad principal, desde la cual se podrán pulsar para editarlos o borrarlos, para lo cual se redirigirá a la actividad.

### Barra de botones inferior:

Para facilitar la navegación entre las diferentes actividades que se presentan en MiNevera se ha creado una barra de botones inferior. Esta barra ha sido definida en el archivo xml *activity\_main.xml* y se ha definido como tres botones contenidos en un *RelativeLayout*.

```

<RelativeLayout
    android:layout_width="match_parent"
    android:id="@+id/rel_lay"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:background="@color/sage_green">
    <ImageButton
        android:id="@+id/boton_lista"
        android:layout_width="70dp"
        android:layout_height="80dp"
        android:layout_marginRight="49dp"
        android:layout_toLeftOf="@id/boton_mas"

```

```

        android:adjustViewBounds="true"
        android:background="@color/sage_green"
        android:contentDescription="@string/b_lista_desc"
        android:onClick="Recetas"
        android:scaleType="fitCenter"
        android:src="@mipmap/b_lista" />
<ImageButton
    android:id="@+id/boton_mas"
    android:layout_width="70dp"
    android:layout_height="80dp"
    android:layout_centerHorizontal="true"
    android:adjustViewBounds="true"
    android:background="@color/sage_green"
    android:contentDescription="@string/b_mas_desc"
    android:onClick="AñadirNevera"
    android:scaleType="fitCenter"
    android:src="@mipmap/b_mas" />
<ImageButton
    android:id="@+id/boton_mapa"
    android:layout_width="70dp"
    android:layout_height="80dp"
    android:layout_alignParentEnd="false"
    android:layout_marginLeft="49dp"
    android:layout_marginRight="0dp"
    android:layout_toRightOf="@id/boton_mas"
    android:adjustViewBounds="true"
    android:background="@color/sage_green"
    android:contentDescription="@string/b_mapa_desc"
    android:onClick="AbrirMapa"
    android:scaleType="fitCenter"
    android:src="@mipmap/b_mapa" />
</RelativeLayout>

```

Todos los botones funcionan de la siguiente manera. En el archivo *activity\_main.xml* se crea el elemento o View <Button> en el Layout correspondiente para que la composición del menú sea la deseada. Una vez que se han definido los atributos del botón se especifica en el atributo `android:onClick` el nombre de un método que va a ser llamado cuando el usuario haga click en el botón. El método llamado debe estar definido como un *public void* al que se le pasa un elemento View. Esta funcionalidad también se puede implementar usando el método `setOnClickListener` de View, como se verá más tarde en la actividad *Camara.java*.

El botón de la derecha, que presenta el icono de una lista de comprobación, se abre el método *Recetas* que infla la actividad de *SearchIngr*. Esta actividad implementa un buscador de recetas por ingredientes.

```

//Botón de funcionalidad de buscador de recetas
public void Recetas(View view) {
    Intent listIntent = new Intent(this, SearchIngr.class);
    startActivity(listIntent);
}

```

El botón de la izquierda, que presenta el icono de un mapa con un marcador, se abre el método *AbrirMapa* que infla la actividad *Map*. Dicha actividad implementa un sistema de localización de supermercados cercanos a la posición del usuario.

```
//Botón de abrir el mapa
public void AbrirMapa(View view) {
    Intent mapIntent = new Intent(this, Map.class);
    startActivity(mapIntent);
}
```

El botón del medio del *RelativeLayout*, representado con un símbolo más, a través del método *AñadirNevera* hace visibles dos nuevos botones que representan dos formas distintas de añadir nuevos productos a la base de datos de MiNevera.

```
//Botón de funcionalidad de añadir alimentos, muestra un menú extra con
dos opciones para añadir productos
public void AñadirNevera(android.view.View view){
    ImageButton boton_camara = findViewById(R.id.boton_camara);
    ImageButton boton_bbdd = findViewById(R.id.boton_bbdd);
    if(boton_camara.getVisibility() == View.GONE){
        boton_camara.setVisibility(View.VISIBLE);
        boton_bbdd.setVisibility(View.VISIBLE);
    } else {
        boton_camara.setVisibility(View.GONE);
        boton_bbdd.setVisibility(View.GONE);
    }
}
```

El botón con el icono ABC representa la funcionalidad de añadir manualmente elementos que no tengan una fecha de caducidad definida. Se usaría introduciendo a mano el nombre del producto que se quiere añadir y el número de días aproximado que el usuario considera que tarda en caducar el producto. Este botón infla la actividad *AddProducts* a través del método *AñadirManualmente*.

```
//Botón de añadir a la lista de la compra manualmente
public void AñadirManualmente(View view) {
    ImageButton boton_camara = findViewById(R.id.boton_camara);
    ImageButton boton_bbdd = findViewById(R.id.boton_bbdd);
    Intent listIntent = new Intent(this, AddProducts.class);
    startActivity(listIntent);
    boton_camara.setVisibility(View.INVISIBLE);
    boton_bbdd.setVisibility(View.INVISIBLE);
}
```

El botón con el icono de la cámara representa la posibilidad de añadir elementos a la nevera escaneando el código de barra del producto y seleccionando la fecha de caducidad en un calendario interactivo. Al hacer click en el botón, infla la actividad *Camara* a través del método *AbrirCamara*.



```
//Botón de añadir a la lista por medio del escaneado del código de barras
public void AbrirCamara(View view) {
    ImageButton boton_camara = findViewById(R.id.boton_camara);
    ImageButton boton_bbdd = findViewById(R.id.boton_bbdd);
    Intent addIntent = new Intent(this, Camara.class);
    startActivity(addIntent);
    boton_camara.setVisibility(View.INVISIBLE);
    boton_bbdd.setVisibility(View.INVISIBLE);
}
```

De las actividades que se acaban de presentar se hablará a continuación en detalle, explicando la funcionalidad y componentes de cada una de ellas.

### Añadir productos manualmente:

Esta funcionalidad ha sido creada a partir de los códigos de ejemplo proporcionados por los profesores. En la actividad *AddProducts*, a la que es posible acceder desde *MainActivity* pulsando en el botón más y seleccionando el botón izquierdo emergente, aparecen en la parte de arriba dos *EditText* en los que es necesario introducir datos para que funcione.

En el cuadro de la izquierda se debe introducir el nombre del producto, y no podrá dejarse en blanco; en el cuadro de la derecha se introducirá el número de días estimado, entre 1 y 25, que aguantará el producto antes de ponerse en mal estado. En el caso de que alguno de los dos campos se quede vacío o que el número de días introducido no esté en el rango establecido, no se podrá introducir el producto y se informará al usuario del problema mediante un *Snackbar*, que aparecerá en la parte inferior de la pantalla, o inmediatamente encima del teclado si este se encuentra en uso en ese momento.

Una vez el usuario ha introducido los datos necesarios para introducir el producto en la aplicación, se procederá a calcular la fecha de caducidad y la diferencia de días y a introducirlo en la base de datos de la aplicación.

```
public String getDate(String num_days) throws ParseException { //Calcula
la fecha de caducidad añadiendo los días a la fecha actual
    Calendar c = Calendar.getInstance();
    Date cDate = c.getTime();
    c.add(Calendar.DATE, Integer.parseInt(num_days));
    String exp_date = DateFormat.format("dd-MM-yyyy", c).toString();
    Date nDate = new SimpleDateFormat("dd-MM-yyyy").parse(exp_date);
    return exp_date;
}
```

```
public String getDiff(String exp) throws ParseException { //Calcula la
diferencia de días entre la fecha de caducidad y la fecha actual
    Calendar c = Calendar.getInstance();
```

```

Date cDate = c.getTime(); // fecha actual
Date expDate = new SimpleDateFormat("dd-MM-yyyy").parse(exp);
long diff = expDate.getTime() - cDate.getTime(); // tiempo en
milisegundos
diff = ((diff/1000)/3600)/24;
return Long.toString(diff);
}

```

El cálculo de la fecha de caducidad y de la diferencia de días se realiza mediante dos métodos, *getDate* y *getDiff*, respectivamente. La primera de ellas recoge el parámetro *String* introducido mediante *EditText* numérico y, añadiendo dichos días a la fecha actual, calcula la fecha de caducidad del producto. El segundo método, a partir de esta fecha de caducidad, calcula cuántos días quedan para que el producto venza. Ambos valores se almacenarán en la base de datos creada para la aplicación.

### Añadir productos por escaneo de código de barras:

La funcionalidad de introducir productos mediante el escaneo de su código de barras se ha implementado en la actividad *Camara*. Cuando la actividad se abre, usando el método *onCreate*, se crean las referencias a los botones que el usuario debe pulsar para escanear y añadir la fecha. También se crean las referencias a los *TextView* que van a contener el código de barras y el nombre del producto una vez que se realice la consulta a la API de *OpenFoodFacts*.

```

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_camara); //Inflamos el layout
    //Botón de escaner y de recogida de fecha
    boton_escaner=findViewById(R.id.boton_escaner);
    datebutton = (Button) findViewById(R.id.introducir_fecha);
    // Creamos tres textview que van a recoger el código de barras, la
    fecha y el nombre de la consulta
    cod_barras=(TextView) findViewById(R.id.cod_barras);
    datetext= (TextView) findViewById(R.id.info_introducefecha);
    product_name = (TextView) findViewById(R.id.product_name);
    //Creamos el adaptador de la base de datos y la abrimos
    dbAdapter = new dbProducts(this);
    dbAdapter.open();
}

```

Cuando el usuario pulsa el botón *Scan* se hace una llamada al método *Escanear* que a su vez hace una llamada al *IntentIntegrator* de *Zxing* y llama al método *initiateScan*.

```

//Botón de escanear código de barras
public void Escanear(View view) {
    new IntentIntegrator(Camara.this).initiateScan();
}

```

El resultado del escaneado se recoge en el método *onActivityResult* y se guarda el resultado en el *TextView* *cod\_barras*. Cuando esto sucede se ejecuta *DownloadBarcodeTask*.

```

@Override

```

```

protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
    super.onActivityResult(requestCode, resultCode, data);
    IntentResult result =
IntentIntegrator.parseActivityResult(requestCode, resultCode, data);
    if(result != null)
        if (result.getContents() != null){
            // Cuando se escanea el código de barras se guarda en
cod_barras
            cod_barras.setText(result.getContents());
            new DownloadBarcodeTask().execute();

        }else{
            cod_barras.setText("Error while scanning the product.");
        }
    }
}

```

El resultado del escaneo se recoge del `TextView cod_barras` en la clase `DownloadBarcodeTask` que, como se puede observar en el código, extiende la clase `AsyncTask`. Esto nos permite hacer que el escaneo del código de barras con Zxing se ejecute en un hilo de fondo en nuestra aplicación. Dentro de la clase `DownloadBarcodeTask` encontramos dos métodos, `doInBackground` y `onPostExecute`.

```

private class DownloadBarcodeTask extends AsyncTask<String, Void, String>
{
    @Override
    protected String doInBackground(String... urls) {
        barcode=cod_barras.getText().toString();
        // Se hace una búsqueda en OpenFoodFacts usando makeCall con la
url y el código de barras
        String temp;
        String
url="https://world.openfoodfacts.org/api/v0/product/["+barcode+"].json";
        temp=
makeCall("https://world.openfoodfacts.org/api/v0/product/["+barcode+"].js
on");
        return temp;
    }
    @Override
    protected void onPreExecute() { }
    @Override
    protected void onPostExecute(String result) {

        // El nombre obtenido en makeCall se guarda como el nombre del
producto
        product_name.setText(result);
        name = product_name.getText().toString();
        if(name.equals("") || name.equals(" ")){
            Snackbar.make(findViewById(R.id.camara_act),
R.string.null_name_camara, Snackbar.LENGTH_SHORT).show();
            return;
        }else{
            //Se muestra el calendario para seleccionar la fecha
            showDatePicker();
        }
    }
}
}

```

El método *doInBackground* se encarga de recoger la serie de números que se han obtenido del código de barras, y con este código hace una búsqueda en OpenFoodFacts a través del método *makeCall* que se explicará a continuación.

```
public static String makeCall(String urlString) {
    URL url = null;
    BufferedInputStream is = null;
    String temp = null;
    JsonReader jsonReader;
    //Nos aseguramos de que la url sea correcta
    try {
        url = new URL(urlString);
    } catch (Exception ex) {
        System.out.println("Malformed URL");
    }
    try {
        if (url != null) {
            HttpURLConnection urlConnection = (HttpURLConnection)
url.openConnection();
            is = new BufferedInputStream(urlConnection.getInputStream());
        }
    } catch (IOException ioe) {
        System.out.println("IOException");
    }
    // Se hace una petición de un archivo .json en el que buscamos el
nombre del producto
    if (is != null) {
        try {
            jsonReader = new JsonReader(new InputStreamReader(is, "UTF-
8"));
            jsonReader.beginObject();
            while (jsonReader.hasNext()) {
                String name = jsonReader洗洗Name();
                if (name.equals("product")) {
                    jsonReader.beginObject();
                    while (jsonReader.hasNext()) {
                        name = jsonReader洗洗Name();
                        if (name.equals("product_name_es") ||
name.equals("generic_name_es") ) {
                            temp = jsonReader洗洗String();
                        } else {
                            jsonReader.skipValue();
                        }
                    }
                    jsonReader.endObject();
                } else {
                    jsonReader.skipValue();
                }
            }
            jsonReader.endObject();
        } catch (Exception ex) {
            System.out.println("Error reading URL");
        }
    }
    return temp;
}
```

El método *makeCall* tiene como input una url para realizar una consulta en la página web de OpenFoodFacts con una terminación .json que es el formato de texto en el cual se devuelven

los resultados de la consulta. Cuando se inicializa el método, se verifica que la url está bien formada y se hace una *HttpsURLConnection* para buscar la url en internet. El resultado de esta consulta es un archivo json en el cual se busca la etiqueta *product\_name\_es* o *generic\_name\_es* que se guarda como el nombre del producto escaneado.

*MakeCall* devuelve como resultado el nombre del producto escaneado que pasa como input al método *onPostExecute*. El nombre obtenido en *makeCall* se guarda como el nombre del producto y tras comprobar que no sea un *string* vacío se hace una llamada al método *showDatePicker*.

```
public void showDatePicker(){// Se hace visible el botón de seleccionar la fecha
    datetext.setVisibility(View.VISIBLE);
    datebutton.setVisibility(View.VISIBLE);
    //Pulsando el botón se llama a un calendario interactivo (DatePickerFragment)
    datebutton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            DialogFragment datePicker = new DatePickerFragment();
            datePicker.show(getSupportFragmentManager(), "date picker");
        }
    });
}
```

*ShowDatePicker* hace visible en pantalla el botón en el cual se selecciona la fecha de caducidad. Este botón *datebutton* usa el método *setOnClickListener* para que al pulsar sobre él se haga una llamada a un objeto *DatePickerFragment*. Este objeto *datePicker* permite al usuario introducir la fecha en un calendario interactivo y una vez que la fecha se ha introducido se procesa en el método *OnDataSet*.

```
public void onDataSet(DatePicker view, int year, int month, int dayOfMonth) {
    // una vez que se mete una fecha en el DatePickerFragment se procesa
    Calendar c = Calendar.getInstance(); // current date
    month = month+1;
    String m = Integer.toString(month);
    String d = Integer.toString(dayOfMonth);
    String y = Integer.toString(year);
    if(Integer.toString(dayOfMonth).length()<2){
        d = "0" + Integer.toString(dayOfMonth);
    }
    if(Integer.toString(month).length()<2){
        m = "0" + Integer.toString(month);
    }
    String expDate = d+"-"+m+"-"+y;
    // y se guarda el producto
    try {
        saveProduct(name,expDate);
    } catch (ParseException e) {
        e.printStackTrace();
    }
}
```

En *OnDataSet* la fecha se procesa para que aparezca en el formato dd-mm-yyyy y se procede a guardar el producto en el método *saveProduct*.

```

public void saveProduct(String name, String days) throws ParseException {
    String diff = getDiff(days);
    if(Integer.parseInt(diff)<0) {
        // Si la fecha de caducidad seleccionada no es correcta, se vuelve
a pedir
        showDatePicker();
        Snackbar.make(findViewById(R.id.camara_act), R.string.wrong_date,
Snackbar.LENGTH_SHORT).show();
    }else{
        // Si es correcta, se le asigna un id a la card, se guarda en la
base de datos y se vuelve al main activity
        if (mRowId == null) {
            long id = dbAdapter.createCard(name, days, diff);
            if (id > 0) {
                mRowId = id;
            }
            setResult(RESULT_OK);
            dbAdapter.close();
            Intent mainIntent = new Intent (this, MainActivity.class);
            startActivity(mainIntent);
            finish();
        }
    }
}

```

Dentro de *saveProduct* lo primero que se hace es calcular el número de días que quedan hasta la fecha de caducidad con el método *getDiff*.

```

// getDiff calcula los días que quedan hasta la fecha de caducidad
public String getDiff(String exp) throws ParseException {
    Calendar c = Calendar.getInstance();
    Date cDate = c.getTime(); // fecha actual
    Date expDate = new SimpleDateFormat("dd-MM-yyyy").parse(exp);
    long diff = expDate.getTime() - cDate.getTime(); // tiempo en
milisegundos
    diff = ((diff/1000)/3600)/24;
    return Long.toString(diff);
}

```

Si el número de días es menor que cero, se vuelve a pedir que el usuario introduzca la fecha. Si la diferencia es positiva la información del producto, nombre y fecha de caducidad, se guarda en la base de datos en un *CardView*. Además se le asigna un id al nuevo producto añadido para una fácil identificación. Con el producto correctamente añadido se vuelve a la *MainActivity*.

### Creación y gestión de la base de datos:

Para la aplicación se ha creado una única tabla en la base de datos *data*, llamada *products*. El código empleado para su implementación también ha sido creado a partir de los códigos de ejemplo proporcionados por los profesores y puede consultarse en la clase *dbProducts.java*.

*products* tiene como objetivo almacenar los distintos productos introducidos a la aplicación, ya sea a través del escáner por código de barras o manualmente. Cuenta con las columnas de **id**, **name**, **days** y **diff**, siendo el primero un entero auto incremental, y el resto tipo *String*.

```
// Sentencia SQL para crear las tablas de las bases de datos
private static final String DATABASE_CREATE = "create table " +
DATABASE_TABLE + " (" +
    KEY_ROWID + " integer primary key autoincrement, " +
    KEY_TITLE + " text not null, " +
    KEY_DATE + " text not null, " +
    KEY_DIFF + " text not null);";
```

En la columna name se insertará el *String* correspondiente. De igual manera, en las columnas *days* y *diff* se almacenarán la fecha de caducidad y el número de días restantes hasta dicha fecha, respectivamente.

En la misma clase también se incluyen métodos para recoger, modificar y eliminar los productos de la base de datos, siendo estos *fetchProduct*, que devuelve un cursor con un único producto de la tabla; *fetchAllProducts*, que recoge todos los productos contenidos en la tabla y ordenados por la diferencia de días de menor a mayor; *updateProduct*, que se empleará para modificar el nombre y fecha de un producto ya almacenado en la base de datos, y *deleteProduct*, que borrará el producto seleccionado de la base de datos.

```
public Cursor fetchProduct(long rowId) throws SQLException {
    Cursor mCursor =
        mDb.query(true, DATABASE_TABLE, new String[] {KEY_ROWID,
            KEY_TITLE, KEY_DATE, KEY_DIFF}, KEY_ROWID + "="
+ rowId, null, null, null, null, null);
    if (mCursor != null) {
        mCursor.moveToFirst();
    }
    return mCursor;
}
```

```
public Cursor fetchAllProducts() {
    //Al recoger todas las n
    return mDb.query(DATABASE_TABLE, new String[] {KEY_ROWID,
        KEY_TITLE, KEY_DATE, KEY_DIFF}, null, null, null, null, KEY_DIFF);
}
```

```
public boolean updateProduct(long rowId, String name, String date, String
diff) {
    ContentValues args = new ContentValues();
    args.put(KEY_TITLE, name);
    args.put(KEY_DATE, date);
    args.put(KEY_DIFF, diff);
    return mDb.update(DATABASE_TABLE, args, KEY_ROWID + "=" + rowId,
null) > 0;
}
```

```
public boolean deleteProduct(long rowId) {
    return mDb.delete(DATABASE_TABLE, KEY_ROWID + "=" + rowId, null) > 0;
}
```

### Sistema de notificaciones:

Desde la pantalla de ajustes, se podrán activar las notificaciones de la aplicación. Se compone de un *switch*, que cuenta con dos estados. De esta manera, si el *switch* está activado (hacia la derecha), las notificaciones se desactivarán.

```
SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(context);
boolean notificationsOff = prefs.getBoolean("switch", false);
//Se recoge el estado del switch
```

Esto se consigue mediante el uso de preferencias compartidas (*Shared Preferences*), que guardarán el estado del switch y se consultarán en el momento de llamar a la notificación en la pantalla principal.

Si el usuario tiene las notificaciones activadas estas mostrarán el elemento de la nevera que más próximo se encuentre a su fecha de caducidad. Esta funcionalidad se implementa por medio del uso de servicios, estos servicios permiten crear componentes de la aplicación que realizan operaciones de forma invisible. Se crea el servicio *NotificationService* que al arrancarse, siguiendo lo descrito en el método *onStartCommand*, crea el canal de notificaciones y dependiendo del número de días que le queden al producto más cercano a caducar manda un mensaje u otro. El nombre del producto que va a caducar y la diferencia de días que quedan hasta que caduque se pasan como *Extras* en el *intent* que se pasa al servicio.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    // Creamos notificación
    NotificationManager notificationManager;
    // Se crea canal de notificaciones
    NotificationCompat.Builder mBuilder =
        new NotificationCompat.Builder(this.getApplicationContext(),
            "com.example.minevera.notify_001");
    // pendingIntent para abrir la actividad cuando se pulse la
    notificación
    Intent ii = new Intent(this.getApplicationContext(),
        MainActivity.class);
    PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, ii,
        PendingIntent.FLAG_IMMUTABLE);
    mBuilder.setContentIntent(pendingIntent);
    mBuilder.setSmallIcon(R.mipmap.ic_launcher_round);
    mBuilder.setTitle("Something in your fridge is about to go
        bad!");
    //Dependiendo del número de días que le queden al elemento que va a
    caducar se manda un mensaje distinto
    if(Integer.parseInt(intent.getStringExtra("DAYS"))==0){
        mBuilder.setContentText("The product " +
            intent.getStringExtra("NAME") + " expires today. ");
    }else if(Integer.parseInt(intent.getStringExtra("DAYS"))==1){
        mBuilder.setContentText("The product " +
            intent.getStringExtra("NAME") + " has 1 day left before it expires. ");
    }
```



```

    }else {
        mBuilder.setContentText("The product " +
intent.getStringExtra("NAME") + " has " + intent.getStringExtra("DAYS") +
" days left before it expires. ");
    }

    notificationManager =
        (NotificationManager)
this.getSystemService(Context.NOTIFICATION_SERVICE);

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        String channelId = "YOUR_CHANNEL_ID";
        NotificationChannel channel = new NotificationChannel(channelId,
            "Canal de MiNevera",
            NotificationManager.IMPORTANCE_DEFAULT);
        notificationManager.createNotificationChannel(channel);
        mBuilder.setChannelId(channelId);
    }
    notificationManager.notify(0, mBuilder.build());
    return Service.START_STICKY;
}

```

El servicio de notificaciones se arranca desde el *MainActivity* en el método *startNotifications* que se llama cuando los productos están siendo añadidos a la *productList* en *fillData*. En *startNotifications* creamos el *intent* que va a lanzar *NotificationService*, se crea un *Bundle* que va a contener la información extra que hay que pasar al servicio y se arranca el servicio de notificaciones.

```

//Las notificaciones se activan activando el servicio creado en
NotificationService
public void startNotifications(String name, String diff) {
    // Creamos el Intent que va a lanzar el servicio
    Intent intent = new Intent(this, NotificationService.class);
    // Creamos la informacion a pasar entre actividades
    Bundle b = new Bundle();
    b.putString("NAME", name);
    b.putString("DAYS", diff);
    // Asociamos esta informacion al intent
    intent.putExtras(b);
    // Iniciamos el servicio
    startService(intent);
}

```

## Búsqueda de recetas a partir de un ingrediente:

Al pulsar el botón de la izquierda situado en la parte inferior de la vista se lleva a cabo un método llamado *Recetas* en el cual se realiza un *intent* para pasar a la siguiente actividad que será *SearchIngr*. En esta actividad se le pedirá al usuario introducir el nombre de un ingrediente en un *EditView*. Este valor introducido será pasado a la siguiente actividad como parámetro con el método *sendName*, que está basado en el código proporcionado por los profesores de la asignatura. Este método también se utilizará para pasar a la siguiente actividad (*Recipe*). En la vista de esta actividad también se da el crédito necesario para utilizar la API de EDAMAM Recipes que se utilizará a continuación para obtener las recetas.

```

public void sendName(View view) {
    // Creamos el Intent que va a lanzar la actividad Recipe
    Intent intent = new Intent(this, Recipe.class);
    // Obtenemos referencias a los elementos del interfaz grafico
    EditText nameText = (EditText) findViewById(R.id.edit_message);
    Button searchButton = (Button) findViewById(R.id.button_search);

    if(nameText.getText().toString().equals("") || nameText.getText().toString().equals(" ")) {
        Snackbar.make(findViewById(R.id.search_ingr),
            R.string.message_dialog_ingr, Snackbar.LENGTH_SHORT).show();
    } else {
        // Creamos la informacion a pasar entre actividades
        Bundle b = new Bundle();
        b.putString("ingredient", nameText.getText().toString());

        // Asociamos esta informacion al intent creado
        intent.putExtras(b);

        // Iniciamos la nueva actividad
        startActivity(intent);
    }
}

```

Al pasar a la siguiente actividad, *Recipes*, se recogerá el parámetro mandado desde la actividad *SearchIngr*, el cual se utilizará para construir el enlace que permitirá acceder al fichero JSON elegido. Al principio de la actividad se creará un tipo de objeto llamado *EdamamRecipe* con los atributos necesarios de las recetas: el nombre, o *label*; la lista de ingredientes, o *ingredientLines*, y el enlace a la receta completa, o *link*.

```

//creamos una clase para crear objetos posteriormente
class EdamamRecipe {
    String label;
    ArrayList<String> ingredientLines;
    URL link;

    public EdamamRecipe() { //String name, String ingredients, String link
        this.label = "";
        this.ingredientLines = new ArrayList<String>();
        this.link = null;
    }

    public void setLabel(String label) {
        this.label = label;
    }

    public void setIngredientLines(String ingredient) {
        ingredientLines.add(ingredient);
    }

    public void setLink(String longitude) {
        try {
            this.link = new URL(longitude);
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }
}

```

```

    }
}

public String getName() {
    return label;
}

public String getIngredientLines() {
    String aux = "";
    for (int i=0; i<ingredientLines.size();i++) {
        aux += "\n - "+ingredientLines.get(i) +"\n";
    }
    return aux;
}

public String getLink() {
    return link.toString();
}

@Override
public String toString(){
    return "Recipe: " + label + "\nIngredients: " + ingredientLines +
"\nLink: " + link;
}
}

```

Dentro de la clase *Recipe* se creará una clase, *EdamamRecipes*, que al ser ejecutada realizará la búsqueda en la base de datos de recetas de EDAMAM. Lo primero que se hace en *EdamamRecipes* es el método *doInBackground*, que se encarga de construir el enlace que dirige al fichero JSON. Este se introduce como parámetro de tipo *String* a un método llamado *makeCall*, que tiene una función similar al método *makeCall* utilizado a la hora de escanear productos con la cámara.

```

public class EdamamRecipes extends AsyncTask<View, Void,
ArrayList<EdamamRecipe>> {

    @Override
    protected ArrayList<EdamamRecipe> doInBackground(View... urls) {
        ArrayList<EdamamRecipe> temp;
        //print en consola para comprobar que la string del url se ha
        creado bien

        System.out.println("https://api.edamam.com/api/recipes/v2?type=public&q="
            + ingredient + "&app_id=" + EDAMAM_ID + "&app_key=" +
            EDAMAM_KEY + "&ingr=3-8");

        //llamamos al método makeCall() que devuelve un arraylist de
        objetos EdamameRecipe
        temp =
        makeCall("https://api.edamam.com/api/recipes/v2?type=public&q=" +
            ingredient + "&app_id=" + EDAMAM_ID + "&app_key=" + EDAMAM_KEY +
            "&ingr=3-8");
        return temp;
    }
}

```

El método crea un objeto *URL* con la *String* pasada como parámetro y conecta con ese enlace utilizando *HttpsURLConnection*. Posteriormente se guarda el contenido del fichero JSON en un buffer. Con un objeto *JsonReader* se recorre el fichero JSON hasta encontrar los elementos necesarios para formar un objeto *EdamamRecipe*. Al formar cada objeto se añade a un *ArrayList* de objetos *EdamameRecipe* y una vez se haya acabado de recorrer el fichero, se devuelve ese *ArrayList* al terminar el método.

```
public ArrayList<EdamamRecipe> makeCall(String urlString) { //

    URL url = null;
    BufferedInputStream is = null;
    ArrayList<EdamamRecipe> temp = new ArrayList<EdamamRecipe>();

    try {
        //se crea una url con el string que se pasa como parámetro
        url = new URL(urlString);
    } catch (Exception ex) {
        System.out.println("Malformed URL");
    }

    try {
        if (url != null) {
            //se conecta con el url y se guarda el json en un buffer para
            //poder trabajar con él
            HttpsURLConnection urlConnection = (HttpsURLConnection)
            url.openConnection();
            is = new BufferedInputStream(urlConnection.getInputStream());
        }
    } catch (IOException ioe) {
        System.out.println("IOException");
    }

    if (is != null) {
        try {
            //vamos recorriendo el fichero json que está guardado en el
            //buffer
            //y vamos seleccionando los campos que nos interesen
            JsonReader jsonReader = new JsonReader(new
            InputStreamReader(is, "UTF-8"));
            jsonReader.beginObject();
            while (jsonReader.hasNext()){
                String name = jsonReader.nextName();
                if (name.equals("hits")){
                    jsonReader.beginArray();
                    while (jsonReader.hasNext()){
                        //creamos un objeto EdamameRecipe auxiliar en el
                        //que guardaremos los campos que nos interesen
                        EdamamRecipe receta = new EdamamRecipe();
                        jsonReader.beginObject();
                        while (jsonReader.hasNext()){
                            name = jsonReader.nextName();
                            if (name.equals("recipe")){
                                jsonReader.beginObject();
                                while (jsonReader.hasNext()){
                                    name = jsonReader.nextName();
                                    if (name.equals("label")){
                                        //guardamos el nombre de la receta
                                        receta.setLabel(jsonReader.nextString());
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```

receta.getName());
                                System.out.println("Receta: " +
                                } else if
(name.equals("ingredientLines")) {
                                jsonReader.beginArray();
                                while (jsonReader.hasNext()) {
                                    //guardamos la lista de los
ingredientes
receta.setIngredientLines(jsonReader.nextString());
                                }
                                jsonReader.endArray();
                                } else if (name.equals("url")) {
                                    //guardamos la url de la página de
la que viene la receta
receta.setLink(jsonReader.nextString());
                                System.out.println("Link: " +
receta.getLink());
                                } else {
                                    jsonReader.skipValue();
                                }
                                }
                                jsonReader.endObject();
                                } else {
                                    jsonReader.skipValue();
                                }
                                }
                                jsonReader.endObject();
                                //se guarda cada receta en el arraylist
temp.add(receta);
                                }
                                jsonReader.endArray();
                                } else {
                                    jsonReader.skipValue();
                                }
                                }
                                } catch (Exception e) {
                                    System.out.println("Exception");
                                    return new ArrayList<EdamamRecipe>();
                                }
                                }
                                //se devuelve el arraylist con todas las recetas
return temp;
}

```

Después se pasa al método *onPostExecute* que se encarga de pasar el *ArrayList* de objetos *EdamameRecipe* a una *List* para poder añadirlo a una *ListView* y que se pueda mostrar en la vista de la actividad.

```

@Override
protected void onPostExecute(ArrayList<EdamamRecipe> result) {
    // Aquí se actualiza el interfaz de usuario
    List<String> listTitle = new ArrayList<String>();

    for (int i = 0; i < result.size(); i++) {

```

```

        //hacemos una lista con las recetas
        listTitle.add(i, "\nRecipe: " + result.get(i).getName() +
"\nIngredients: " + result.get(i).getIngredientLines() + "\nLink: " +
result.get(i).getLink());
    }

    //adaptamos la lista para poder mostrarlo en card views y meterlo
    en una list view
    ArrayAdapter<String> myAdapter;
    myAdapter = new ArrayAdapter<String>(Recipe.this,
R.layout.row_layout, R.id.listText, listTitle);
    m_listview.setAdapter(myAdapter);
}
}

```

### Búsqueda de supermercados cercanos:

Al pulsar el botón situado en la parte inferior derecha se lleva a cabo el método *AbrirMapa* el cual realiza un *intent* que empieza la actividad *Map*.

En esta actividad en el método *onCreate* se pedirá permiso para acceder a la localización del mapa si no se tiene previamente y se guardarán los valores de la latitud y la longitud del usuario en las variables *latitude* y *longitude* para ser enviado después a la siguiente actividad.

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    //inflamos el layout
    setContentView(R.layout.activity_map);

    // Creamos un listview que va a contener los resultados de las
    consulta a Google Places
    m_listview = (ListView) findViewById(R.id.id_list_places);

    // Comprobamos si tenemos permiso para acceder a la localización
    if
    (checkSelfPermission(android.Manifest.permission.ACCESS_FINE_LOCATION) ==
PackageManager.PERMISSION_GRANTED) {
        System.out.println("HELLOLOCATION: Tenemos permisos...");
    } else {
        // no tiene permiso, solicitarlo
        requestPermissions(new
String[]{android.Manifest.permission.ACCESS_FINE_LOCATION},
REQUEST_PERMISSION_ACCESS_FINE_LOCATION);
    }

    //Accedemos al servicio de localización
    LocationManager servicioLoc = (LocationManager)
getSystemService(Context.LOCATION_SERVICE);

    //Obtenemos la lista de proveedores disponibles
    boolean soloActivos = true;

```

```

List<String> proveedores = servicioLoc.getProviders(soloActivos);

if (proveedores.isEmpty()) { // No hay ninguno activo y no se puede
hacer nada
    return;
}

// Vemos si está disponible el proveedor de localización que queremos
usar
String proveedorElegido = LocationManager.GPS_PROVIDER;
boolean disponible = proveedores.contains(proveedorElegido);

// Otra opción es utilizar uno cualquiera de la lista (por ejemplo, el
primero)
if (!disponible) {
    proveedorElegido = proveedores.get(0);
}

//Pedimos la última localización conocida por el proveedor
Location localizacion =
servicioLoc.getLastKnownLocation(proveedorElegido);

//Tiempo mínimo entre escuchas de nueva posición
int tiempo = 1000; //milisegundos
//Distancia mínima entre escuchas de nueva posición
int distancia = 100; //metros

//Pedimos escuchar actualizaciones de posicion que reciba el proveedor
elegido, cada
//1000ms o 100m, que serán procesadas por el escuchador (implementado
en esta misma clase)
servicioLoc.requestLocationUpdates(proveedorElegido, tiempo,
distancia, (LocationListener) this);

//guardamos los valores de nuestra posicion para pasarlos a la
siguiente activity
latitude = String.valueOf(localizacion.getLatitude());
longitude = String.valueOf(localizacion.getLongitude());

new GooglePlaces().execute();
}

```

Al buscar los supermercados más cercanos y devolver los resultados en una *ListView*, la estructura de esta actividad es muy similar al de la actividad de *Recipe*. Consiste en un objeto en el que se van a guardar los datos de los supermercados que son buscados en la API de Google Places (*GooglePlace*). Este objeto incluye el nombre del supermercado o *name*, la latitud o *latitude* y la longitud o *longitude*. También se incluye una clase llamada *GooglePlaces* que se ejecuta para realizar la búsqueda de supermercados. Esta clase consiste en un método *doInBackground* en el que se crea un *ArrayList* de objetos *GooglePlaces*.

```

//creamos una clase para crear objetos posteriormente
class GooglePlace {
    private String name;
    private String latitude;
    private String longitude;
}

```

```

public GooglePlace() {
    this.name = "";
    this.latitude = "";
    this.longitude = "";
}

public void setName(String name) {
    this.name = name;
}

public void setLatitude(String latitude) {
    this.latitude = latitude;
}

public void setLongitude(String longitude) {
    this.longitude = longitude;
}

public String getName() {
    return name;
}

public String getLatitude() {
    return latitude;
}

public String getLongitude() {
    return longitude;
}
}

```

Los objetos son creados en el método *makeCall*, que funciona igual que el método *makeCall* de la actividad *Recipes*.

Los datos que se pasan a la siguiente actividad gracias al método *sendLocations* son *latitude*, *longitude* y el *ArrayList<GooglePlace> supermarkets*, para tener los valores de las latitudes y longitudes de todos los supermercados cercanos en la actividad. En este método también se empieza la siguiente actividad *MapMarkers*.

```

public void sendLocations(View view) {
    // Creamos el Intent que va a lanzar la actividad MapMarkers
    Intent intent = new Intent(this, MapMarkers.class);

    // Creamos la informacion a pasar entre actividades
    //se van a pasar tanto los valores de nuestra latitud y longitud
    //como un arraylist con todas las latitudes y longitudes de los
    supermercados más cercanos
    Bundle b = new Bundle();
    b.putString("latitude", latitude);
    b.putString("longitude", longitude);
    b.putStringArrayList("supermarkets", supermarkets);

    // Asociamos esta informacion al intent
    intent.putExtras(b);
}

```



```
// Iniciamos la nueva actividad
startActivity(intent);
}
```

En *MapMarkers* se recogen todos los valores enviados desde *Map* y se infla el mapa en pantalla en el método *onReadyMap*, que es un método de la interfaz *OnMapReadyCallback*. En este método se dibuja el mapa y se añaden todos los marcadores de los supermercados.

```
@Override
public void onMapReady(@NonNull GoogleMap googleMap) {
    mMap = googleMap;

    UiSettings settings = mMap.getUiSettings();

    //ponemos el boton de zoom a un lado del mapa
    settings.setZoomControlsEnabled(true);
    //ponemos la brújula
    settings.isCompassEnabled();

    //ponemos el punto azul con la localización actual
    if (ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED &&
ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_COARSE_LOCATION) !=
PackageManager.PERMISSION_GRANTED) {
        return;
    }

    mMap.setMyLocationEnabled(true);

    for(int i=0; i<supermarkets.size();i++){
        //empezamos a sacar solo los datos que necesitamos de cada string
        de cada supermercado
        lat_market =
supermarkets.get(i).substring(supermarkets.get(i).indexOf("Latitude:"),
supermarkets.get(i).indexOf("\nLongitude"));
        lon_market =
supermarkets.get(i).substring(supermarkets.get(i).indexOf("Longitude:"));
        place_name =
supermarkets.get(i).substring(supermarkets.get(i).indexOf("name:"),
supermarkets.get(i).indexOf("\nLatitude"));

        //continuamos sacando solo los datos que necesitamos
        lat_market = lat_market.substring(lat_market.indexOf(":"));
        lon_market = lon_market.substring(lon_market.indexOf(":"));
        place_name = place_name.substring(place_name.indexOf(":"));

        lat_market = lat_market.replace(":", "");
        lon_market = lon_market.replace(":", "");
        place_name= place_name.replace(":", "");

        //pasamos la latitud y longitud a Double para poder hacer un
        objeto LatLng con ellos
        latd = Double.parseDouble(lat_market);
        lond = Double.parseDouble(lon_market);
    }
}
```

```

        //creamos el marcados del supermercado
        LatLng market = new LatLng(latd, lond);
        MarkerOptions markerOpts = new MarkerOptions();
        markerOpts.position(market);
        //si se hace click en el marcador muestra el nombre del
supermercado
        mMap.addMarker(markerOpts).setTitle(place_name);
    }

    //centramos el mapa en nuestra localizacion
    centerMap(Double.parseDouble(latitude),
Double.parseDouble(longitude));
}

```

Utilizando la función *centerMap*, que recibe como parámetros la posición del usuario, se centra el mapa en su posición actual.

```

public void centerMap(double latitude, double longitude){

    // A partir de una pareja de coordenadas (tipo double) creamos un
objeto LatLng,
    // que es el tipo de dato que debemos usar al tratar con mapas
    LatLng position = new LatLng(latitude, longitude);

    // Obtenemos un objeto CameraUpdate que indique el movimiento de
cámara que queremos;
    // en este caso, centrar el mapa en unas coordenadas con el método
newLatLng()
    CameraUpdate update = CameraUpdateFactory.newLatLng(position);

    float zoom = 16;
    update = CameraUpdateFactory.newLatLngZoom(position, zoom);

    // Pasamos el tipo de actualización configurada al método del mapa que
mueve la cámara
    mMap.moveCamera(update);
}

```

**Archivos adicionales:**

**Enlace al .zip en [Drive](#)**

**Enlace al video de Youtube de [MiNevera](#)**