

# Persistencia y bases de datos - Parte I

---

## Persistencia de datos en Android

---

### Competencias:

- Entender las diferentes opciones para almacenar datos en Android.
- Decidir cuál es la mejor opción para otorgar persistencia de datos en las Aplicaciones.
- Saber qué opciones ofrece Android para compartir información entre aplicaciones.

### Introducción

El sistema operativo Android nos ofrece diferentes opciones para almacenar los datos que utiliza nuestra aplicación. Las opciones disponibles se utilizan en diferentes casos de uso, normalmente las opciones por defecto de una aplicación es más que suficiente para la mayoría de los casos. Las opciones por defecto de Android para almacenar información o datos en una aplicación son SharedPreferences o el Sandbox de la aplicación.

Qué opción de almacenamiento tomamos depende directamente de las necesidades de nuestra aplicación, sean estas necesidades de seguridad, espacio, disponibilidad, etc. Aprenderemos sobre las diferentes opciones que nos ofrece Android, y de qué manera podemos elegir dichas opciones.

## Persistencia de datos en Android

En Android, una aplicación cuenta con un ecosistema que le permite almacenar datos que son utilizados por nuestra aplicación. Cada aplicación cuenta con un Sandbox al cual sólo tiene acceso la aplicación, y el sistema operativo en modo root. Este Sandbox es el contenedor de ejecución seguro de cada aplicación, donde se almacena el código fuente, los assets, archivos de configuración, imágenes, etc. No confundir contenedor de ejecución de la aplicación con ART, que es Android Run Time. ART es donde se ejecuta el código en Android, pero el contenedor de cada aplicación es el Sandbox. Nadie tiene acceso a este contenedor o sandbox, el cual se ejecuta con el mínimo de privilegios necesarios en el dispositivo. Este concepto de Sandbox es gracias a linux, que crea un id para cada proceso que corre en el sistema operativo, con el mínimo de privilegios y sin intervención externa.

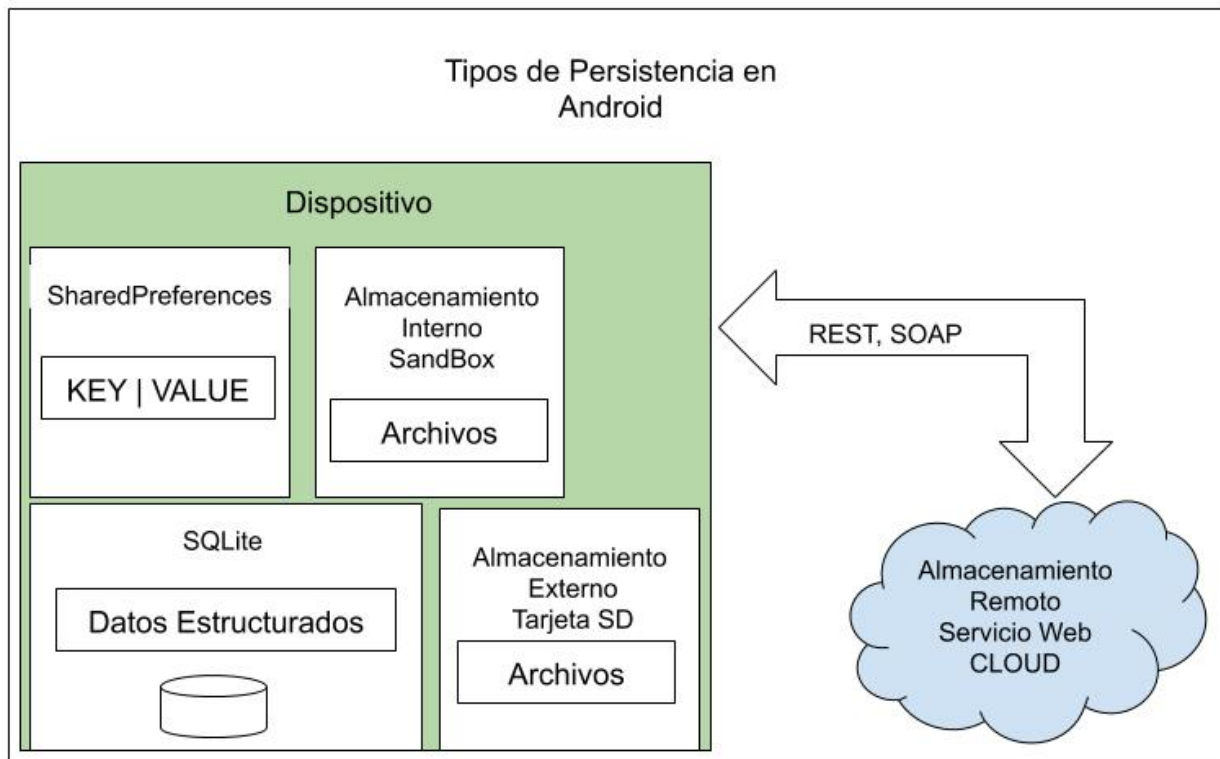


Imagen 1. Diagrama con los tipos de persistencia en Android.

Las diferentes opciones de persistencia de datos son las siguientes:

1. **SharedPreferences:** Nos permite almacenar datos primitivos en formato clave-valor. Este se profundizará más adelante.
2. **Almacenamiento Interno:** Nos permite almacenar información en la memoria interna del dispositivo. La información se puede almacenar en archivos dentro del Sandbox de la aplicación. Sólo la aplicación tiene acceso a esos archivos, a menos que el dispositivo se encuentre rooteado.
3. **Almacenamiento externo:** Los dispositivos Android permiten conectar memoria externa, normalmente en formato de memoria SD. Esta memoria es extraíble o se puede montar en un computador, por lo cual no se asegura su disponibilidad, y se recomienda almacenar información que no sea crítica para el funcionamiento de la aplicación. Se pueden almacenar archivos, los cuales son de acceso público para todos los usuarios. Cualquiera de estos usuarios puede eliminar o modificar dichos archivos. Este tipo de almacenamiento no tiene seguridad.
4. **Base de Datos SQLite:** Almacenamiento de datos, estructurado, privado. Esta implementación de SQL está optimizada para dispositivos móviles, es la opción por defecto de base de datos en Android. Desde los inicios se ha podido crear bases de dato en SQLite. La librería Room de Google nos permite interactuar con SQLite de manera más simple, con menos errores y menos boilerplate code, facilitando el uso y el desarrollo. Es decir, Room crea una capa de abstracción que nos permite interactuar directamente con la base de datos en SQLite, pero con la familiaridad de trabajar con objetos en vez de tablas y queries SQL. Room será visto en profundidad más adelante.
5. **Almacenamiento remoto:** Podemos almacenar información de modo remoto, ya sea a través de un servicio web, normalmente REST, o a través de un servicio tipo Cloud.

## Datos y Seguridad

¿Cómo podemos decidir qué tipo de almacenamiento nos conviene utilizar?. Para decidir esto debemos tener claro:

- ¿Qué tipo de información o datos estamos almacenando?.
- ¿Los datos son públicos o privados?
- ¿Qué disponibilidad debe tener esa información o datos?
- ¿Quién tiene acceso a esa información o datos ?

Estas interrogantes son claves para decidir qué tipo de almacenamiento utilizaremos. Debemos considerar no sólo el tamaño de los datos, también es necesario considerar la privacidad y disponibilidad de los mismos. Veamos algunos casos de ejemplo.

## **Almacenamiento de fotografías de la Cámara**

Por ejemplo la aplicación de la cámara necesita almacenar archivos de imagen de gran tamaño, en este caso podemos utilizar el almacenamiento interno o externo. Las imágenes que se toman con la cámara del dispositivo, con la aplicación por defecto de la cámara, son de acceso público y se encuentran disponibles en la galería del dispositivo, o los archivos multimedia del mismo. Estos mismos archivos son utilizados por aplicaciones para ser publicadas en internet, estas aplicaciones deben pedir permiso para acceder a las imágenes, pero nada impide que esos mismos archivos sean extraídos en un computador personal, incluso eliminados de la memoria del dispositivo, sea interna o externa.

## **Preferencias de una aplicación**

Consideremos el caso donde nuestra aplicación permite a un usuario configurar a su gusto algunas cosas, por ejemplo el tamaño de la letra o los colores de ciertos componentes. Como esta configuración es local, no representa problemas de privacidad, y no tiene mucha complejidad, podemos utilizar `SharedPreferences` para almacenar un conjunto de clave-valor y mantener los datos de la configuración entre instancias de nuestra aplicación. Si estos datos se llegan a perder, nuestra aplicación no sufre ninguna consecuencia severa, salvo pequeños inconvenientes para el usuario.

## **Aplicación Bancaria**

Imaginemos una aplicación bancaria, en la cual tenemos acceso a diferentes servicios, por ejemplo revisar nuestra cartola mensual en formato pdf. Esta cartola contiene información sensible para el usuario, y debe ser almacenada con la mayor seguridad posible, por lo que en este caso es necesario utilizar el almacenamiento interno, en el sandbox de la aplicación, para que nadie, salvo el usuario en nuestra aplicación, pueda ver el archivo descargado. Medidas adicionales son bienvenidas en casos de extrema sensibilidad de información, algunos son: claves de acceso, encriptación o cifrado de los archivos.

## **Aplicación con modo Online/Offline**

Lo más común actualmente es contar con aplicaciones que utilizan Internet para obtener información, pero muchas veces se ven afectadas por la intermitencia de las conexiones de los teléfonos celulares. En estos casos podemos desarrollar una versión Offline, normalmente con Caché o data almacenada localmente, que podemos mostrar al usuario mientras no tenemos conexión. Este caché, normalmente, se almacena en una base de datos local, y se va renovando con la última información, a medida que vamos cargando desde el servicio Online. Hay aplicaciones que para mejorar la experiencia del usuario, almacenan grandes cantidades de caché, lo cual termina resintiéndolo el desempeño del dispositivo por falta de espacio. Tener cuidado al implementar este tipo de almacenamiento en una aplicación.

# Compartiendo datos en Android

Como ya sabemos, los datos o información, en Android, sólo están disponibles para la aplicación. En el caso que necesitemos compartir información entre aplicaciones contamos con las siguientes opciones:

1. **Compartir a través de SharedPreferences:** Podemos compartir nuestro archivo de SharedPreferences con otras aplicaciones a través del modo global.
2. **ContentProvider:** los proveedores de contenido nos permiten acceder a datos de otra aplicación. Estos datos son compartidos para que sean usados por otras aplicaciones, incluso pueden ser modificados si el ContentProvider lo permite. Se pueden compartir imágenes, archivos, o datos de una base de datos. Si queremos compartir información desde una aplicación A a una aplicación B, la aplicación A debe implementar el ContentProvider. La aplicación B debe utilizar el ContentProvider implementado por A y pedir los permisos necesarios. Por ejemplo WhatsApp debe utilizar el ContentProvider de la aplicación de Contactos de el sistema, así obtiene los contactos necesarios para empezar a enviar mensajes o llamar por teléfono.

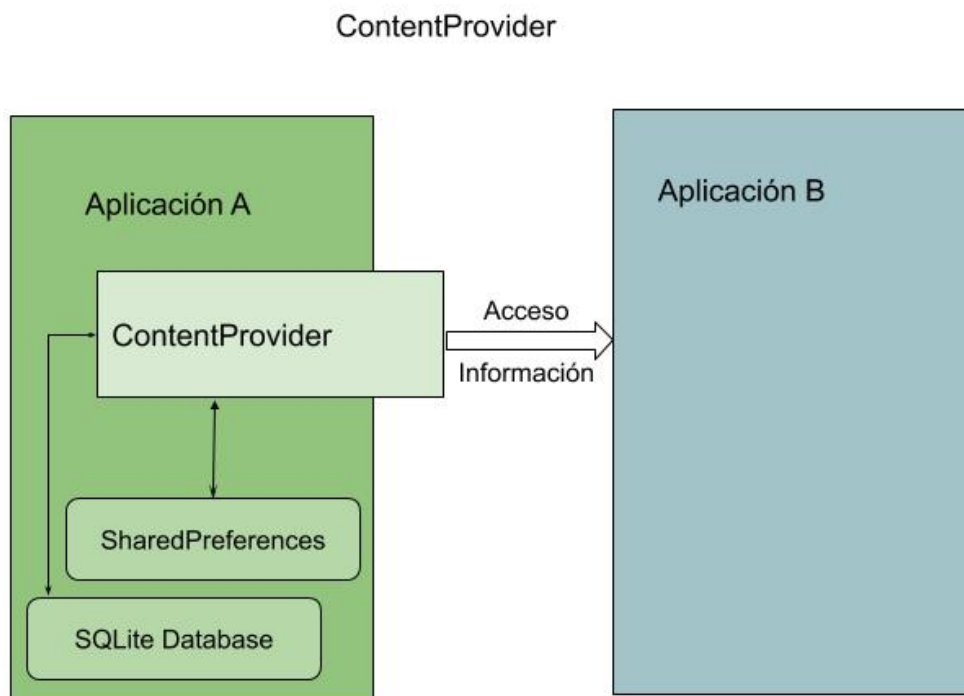


Imagen 2. Diagrama de un proveedor de contenidos que nos permite acceder a datos de otra aplicación.

# SharedPreferences

---

## Competencias:

- Añadir SharedPreferences a una aplicación
- Utilizar buenas prácticas en el uso de SharedPreferences
- Compartir información entre aplicaciones a través de SharedPreferences

## Motivación

A continuación nos adentraremos en una de las opciones más simples, y utilizadas, para almacenar datos en Android. Comprenderemos que tipo de almacenamiento tenemos disponible, que podemos almacenar, buenas y malas prácticas con respecto al uso de SharedPreferences. Aprenderemos a crear un Archivo de Preferencias, los diferentes modos en que podemos crear el mismo, y cómo hacer que ese archivo esté disponible para otras aplicaciones.

## SharedPreferences, o archivo de preferencias

SharedPreferences es la forma más simple de almacenar información en Android. Nos permite almacenar pequeñas cantidades de información, en un formato de clave-valor, en un archivo en nuestro dispositivo. Este archivo se puede crear con diferentes configuraciones que limitan el acceso de terceros a la información almacenada en este archivo. El archivo es directamente manejado por Android, todos los componentes de la aplicación tienen acceso al archivo, pero no está disponible para otras aplicaciones. El archivo de sharedPreferences se inicia de manera recomendada en modo privado, indicando que sólo los componentes de la aplicación dueña del archivo tienen acceso al mismo. Existe un modo público para este archivo, pero no está recomendado, y se han deprecado sus opciones desde hace varias versiones de Android, más información sobre esto en la sección donde explicamos como crear un archivo de preferencias.

SharedPreferences está recomendado sólo para pequeñas cantidades de información, si la información que se debe almacenar es más compleja que unas cuantas primitivas, por ejemplo un objeto, se recomienda utilizar una base de datos SQLite.

Los datos almacenados en SharedPreferences son privados para la aplicación. Los datos persisten más allá de una sesión de la aplicación, los datos almacenados en el archivo se mantienen incluso si la aplicación es cerrada explícitamente. Se utiliza normalmente para almacenar las preferencias del usuario, su nickname o su puntaje en un juego, nada muy complejo o delicado.

# Utilizando SharedPreferences en una aplicación

Para utilizar SharedPreferences en una aplicación, necesitamos de alguna clase que extienda Context, para los ejemplos y explicaciones de este contenido asumimos la existencia de una aplicación ficticia que cuenta con una clase MainActivity, que nos permite obtener el Context. Más adelante construiremos una aplicación de ejemplo, paso por paso, donde utilizaremos todo lo que veremos a continuación, pero primero debemos tener claro cómo crear un archivo, como utilizarlo y las operaciones que nos permite. Además, debemos entender lo siguiente:

## No debemos confundir SharedPreferences con Preference:

- **SharedPreferences** es un conjunto de APIs que nos permite almacenar datos primitivos (String, Int, Boolean ,etc), en formato clave-valor, estos datos persisten entre instancias de la aplicación.
- **Preference** es un conjunto de APIs que nos permiten crear una interfaz de usuario para una o varias pantallas de preferencias, por ejemplo la configuración de idioma o notificaciones push de la aplicación. Preference ocupa SharedPreferences para almacenar y persistir estos datos que son seleccionados en la pantalla de preferencias.

## Creando un archivo de SharedPreferences en nuestra aplicación ficticia

Normalmente necesitamos sólo un archivo por aplicación, el nombre está asociado con el paquete de la misma. De esta forma es más fácil asociar el archivo con la aplicación. Esto no es una restricción y podemos utilizar cualquier nombre, pero está recomendado que utilicemos el nombre del paquete de la aplicación.

Para crear un archivo de preferencias debemos tener acceso a un Context, esto significa que debemos tener acceso a una Actividad o algún otro componente de Android que nos permita obtener el objeto Context. Por ejemplo en nuestra Actividad en una clase llamada MainActivity agregaremos una variable que mantendrá disponible nuestro archivo de preferencias. Para hacer esto debemos agregar el siguiente código a esta clase de ejemplo que contiene la actividad:

```
class MainActivity : AppCompatActivity() {  
    lateinit var sharedPreferences: SharedPreferences  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        val fileName = "cl.desafiolatam.persistenciaandroid"  
        sharedPreferences = getSharedPreferences(fileName, Context.MODE_PRIVATE)  
    }  
}
```

Como vemos, nuestra MainActivity extiende AppCompatActivity que nos permite obtener un Context. Además, contiene una variable lateinit var llamada sharedPreferences, de tipo SharedPreferences. Esta variable tiene un modificador lateinit, que nos permite definir una propiedad de la clase, que no será iniciada inmediatamente en el constructor de la misma.

```
class MainActivity : AppCompatActivity() {  
    lateinit var sharedPreferences: SharedPreferences  
    ...  
}
```

Todas las variables tipo var en kotlin deben ser inicializadas en el constructor de la clase, pero cuando esto no resulta conveniente, como en este caso que inicializamos la variable en el método onCreate, podemos utilizar el modificador lateinit para inicializar la variable más adelante en nuestro código. Si queremos acceder a esta variable antes de que sea inicializada lo más recomendable es hacer lo siguiente:

```
if (this::sharedPreferences.isInitialized) {  
    //hacer la operación de forma segura  
}
```

De esta forma accedemos de manera segura a nuestra variable, en este caso usamos la referencia al nivel donde está declarada la variable `this::`, luego ocupamos el nombre de la variable y la propiedad isInitialized para saber si ya fue inicializada.

Para crear el archivo hacemos uso del método Context.getSharedPreferences. Este método es parte de un objeto instanciado que extiende Context en Android. En este caso nuestra MainActivity es una instancia de un objeto que extiende Context, por lo que podemos hacer referencia directa al método:

```
val fileName = "cl.desafiolatam.persistenciaandroid"  
//Modo recomendado, privado  
sharedPreferences = getSharedPreferences(  
    fileName,  
    Context.MODE_PRIVATE)  
//otros modos  
sharedPreferences = getSharedPreferences(  
    fileName,  
    Context.MODE_WORLD_READABLE)  
sharedPreferences = getSharedPreferences(  
    fileName,  
    Context.MODE_WORLD_WRITEABLE)  
sharedPreferences = getSharedPreferences(  
    fileName,  
    Context.MODE_MULTI_PROCESS)
```

El método recibe dos parámetros, las opciones para estos parámetros son las siguientes:



- **name:** El nombre del archivo de preferencia. Si el archivo no existe, se creará cuando se utilice un editor, `SharedPreferences.edit()`, y se haga commit a los cambios, `Editor.commit()`. Es decir, el sistema creará nuestro archivo, y después lo mantendrá, junto con sus datos almacenados, hasta que la aplicación sea eliminada del dispositivo o el usuario borre explícitamente las preferencias de la aplicación, al eliminar los datos de la aplicación desde el manager de aplicaciones del dispositivo.
- **mode:** El modo en el cual operará el archivo, en este caso utilizamos el modo privado pero tenemos las siguientes opciones:
  - **MODE\_PRIVATE:** El archivo puede sólo ser accedido por la aplicación, o todas las aplicaciones que comparten el mismo ID. Este es el modo por defecto, y recomendado para la creación y operación de archivos de preferencias.
  - **MODE\_WORLD\_READABLE:** El archivo puede ser leído por todas las aplicaciones, esto modo no es recomendado y fue deprecado en la API 17, tiene problemas de seguridad.
  - **MODE\_WORLD\_WRITEABLE:** El archivo puede ser modificado por todas las aplicaciones, no recomendado, con problemas de seguridad y deprecado en la API 17.
  - **MODE\_MULTI\_PROCESS:** El archivo en el disco es revisado por posibles modificaciones, aunque ya se encuentre cargado en el proceso actual de la aplicación. Este modo es deseado cuando tenemos una aplicación con múltiples procesos, los cuales escriben en el mismo archivo simultáneamente. Este modo es ocupado principalmente cuando se necesita comunicar datos entre procesos, pero existen alternativas mucho mejores. Este modo fue deprecado en la API 23, debido a problemas de desempeño, consistencia entre procesos y la poca fiabilidad de funcionamiento en varias versiones de Android. Las aplicaciones deben utilizar métodos más fiables para comunicar información entre procesos, lo recomendado es utilizar un `ContentProvider`.

## SharedPreferences.Editor

SharedPreferences es una Interfaz, que nos permite interactuar con el archivo de preferencias que creamos. Para poder almacenar datos en este archivo debemos utilizar un editor. Para esto debemos hacer lo siguiente:

```
sharedPreferences = getSharedPreferences(fileName, Context.MODE_PRIVATE)
sharedPreferences.edit().putString("TestString","Hola").commit()
sharedPreferences.edit().putString("TestString2","Hola 2").apply()
```

Al ocupar edit(), SharedPreferences nos retorna una interfaz SharedPreferences.Editor, esta interfaz nos permite ejecutar diferentes métodos para almacenar distintos datos. El método edit no se puede utilizar por sí solo, por lo cual debemos agregar alguno de los siguientes métodos, disponibles a través de la interfaz Editor:

- **apply:** Este método nos permite aplicar los cambios que hicimos sobre el archivo de preferencias que se encuentra en memoria. Hay que tener cuidado con este método ya que si bien ejecuta de forma atómica la operación de modificación, si hay dos editores modificando al mismo tiempo, el último que aplica apply termina sobrescribiendo los datos. Este método guarda en el disco de manera asíncrona, aunque modifica inmediatamente el archivo en memoria. Además, si se da el caso que más de un editor está haciendo un commit(), y hay algún apply que no ha terminado, el commit() bloqueará todo hasta que terminen todos los commit, incluido el que está bloqueando. Este método puede reemplazar de manera segura cualquier operación commit(), si estamos ignorando el resultado que retorna la operación. Este método retorna void. No se señala si la operación falló o fue exitosa. Recomendado si estamos ejecutando los cambios en el Thread principal del sistema.
- **clear:** este método elimina todos los valores que se encuentran en el archivo de preferencia. Después que hagamos commit() no quedará nada en el archivo de preferencias, salvo que hayamos salvado algo en el editor que ocupamos para ejecutar clear. Luego de que se ejecuta el método sobre el archivo de preferencias, este clear se ejecuta primero que cualquier otra operación, sin importar si clear se llamó primero o después. Este método retorna un SharedPreferences.Editor
- **commit:** Nos permite almacenar los datos de manera inmediata en el archivo de preferencias, al igual que con apply debemos considerar que si dos editores hacen commit, el último sobrescribirá los datos. Este método se ejecuta de modo síncrono al disco, por lo que no se recomienda su ejecución en el Thread principal de la aplicación. Todos los commit pueden ser reemplazados por apply de manera segura. Commit retorna un boolean verdadero si los valores fueron almacenados de forma correcta.
- **putBoolean:** método que permite almacenar un valor boolean en las preferencias, recibe una clave tipo String, que es el nombre de la preferencia, y el valor de la misma, en este caso un true o false. Si la llave no existe creará el valor en el archivo, si existe la reemplaza. Este método retorna un editor que es ocupado para hacer commit o apply.
- **putFloat:** método que permite almacenar un valor del tipo Float en las preferencias, recibe una clave tipo String y un valor de tipo Float. Se crea la clave de no existir, si existe se reemplaza. Este método retorna un editor que es ocupado para hacer commit o apply.

- **putInt**: método que permite almacenar un valor de tipo Int en las preferencias. Recibe una clave tipo String y un valor de tipo Int. Se crea si no existe, si existe se reemplaza. Este método retorna un editor que es ocupado para hacer commit o apply.
- **putLong**: método que permite almacenar un valor de tipo Long. Recibe una clave tipo String y un valor tipo Long. Se crea si no existe, si existe se reemplaza. Este método retorna un editor que es ocupado para hacer commit o apply.
- **putString**: método que permite almacenar un valor de tipo String. Recibe una clave tipo String y un valor tipo String. Se crea si no existe, si existe se reemplaza. Este método retorna un editor que es ocupado para hacer commit o apply.
- **putStringSet**: método que permite almacenar un Set de Strings o Set en las preferencias. Recibe una clave tipo String y un Set de String para almacenar. Si no existe se crea, si existe se reemplaza. Este método retorna un editor que es ocupado para hacer commit o apply.
- **remove**: método que nos permite eliminar claves específicas, se ejecuta cuando se aplica commit al archivo de preferencias. El método se ejecuta primero que cualquier otro método independiente del orden en que se ejecutó en el código.

A continuación podemos ver algunos ejemplos del uso de un Editor de SharedPreferences, recordando que los utilizaremos en nuestra ejemplo paso por paso.

```
sharedPreferences.edit().putString("TestString", "Hola").apply()
sharedPreferences.edit().putBoolean("TestBoolean", true).apply()
sharedPreferences.edit().putFloat("TestFloat", 1.0f).apply()
sharedPreferences.edit().putInt("TestInt", 1).apply()
sharedPreferences.edit().putLong("TestLong", 1L).apply()
val setString = setOf("one", "two", "three", "four")
sharedPreferences.edit().putStringSet("TestStringSet", setString).apply()
```

## Métodos de SharedPreferences

La interfaz nos permite no sólo editar, con la interfaz Editor, también nos permite leer y realizar otras operaciones sobre el archivo de preferencias. Las otras operaciones que podemos realizar son las siguientes:

- **contains:** método que recibe una clave y nos devuelve un boolean indicando si esta clave está presente o no en el archivo de preferencias. Sólo nos dice si el archivo contiene la clave, no el valor.
- **edit:** es el método que nos devuelve el Editor que describimos en el apartado anterior.
- **getAll:** este método nos entrega un Map con todas las claves y valores presentes en el archivo de preferencias. Podemos iterar sobre este Map y recorrer todas la claves del archivo, pero debemos saber específicamente qué tipo de valor está relacionado con cada clave, para poder operar correctamente con ese valor.
- **getBoolean:** nos permite leer un boolean desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor boolean de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo boolean, el método entrega una ClassCastException.
- **getFloat:** nos permite leer un Float desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor float de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Float, el método entrega una ClassCastException.
- **getInt:** nos permite leer un Int desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor int de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Int, el método entrega una ClassCastException.
- **getLong:** nos permite leer un Long desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor long de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Long, el método entrega una ClassCastException.

- **getString**: nos permite leer un String desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor String de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Int, el método entrega una ClassCastException.
- **getStringSet**: nos permite leer un Set de Strings desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el Set con los valores de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Set, el método entrega una ClassCastException.

Algunos ejemplos de estos métodos

```
//para saber si contiene cierta clave
sharedPreferences.contains("TestString")
//si quiero todos los valores y claves contenidos
//pasa de ser getAll() a all gracias a kotlin
sharedPreferences.all
//obtener un boolean
sharedPreferences.getBoolean("TestBoolean", false)
//obtener un int
sharedPreferences.getInt("TestInt", -1)
//obtener un long
sharedPreferences.getLong("TestLong", -1L)
//obtener un Float
sharedPreferences.getFloat("TestFloat", -1.0f)
//obtener un String
sharedPreferences.getString("TestString", "NotFound")
//obtener un Set<String>
sharedPreferences.getStringSet("TestStringSet", setOf())
```

Además SharedPreferences cuenta con una interfaz listener que nos permite ser notificados cuando una propiedad cambia en el archivo de preferencias.

## SharedPreferences.OnSharedPreferencesChangeListener

Esta interfaz nos permite escuchar los cambios que ocurren en un archivo de preferencias. Para que esto funcione, debemos implementar la Interfaz en nuestra clase que va a escuchar, para este caso digamos que nuestra MainActivity será la clase que escuchar y debe implementar la interfaz. El código queda de la siguiente manera:

```
class MainActivity : AppCompatActivity(),
SharedPreferences.OnSharedPreferencesChangeListener {
    override fun onSharedPreferencesChanged(
        changedSharedPreferences: SharedPreferences?,
        changedKey: String?) {

        if (changedKey.equals("UserNickName", true)) {
            val newNickName = changedSharedPreferences?
                .getString(changedKey, "")
            displayText.text = newNickName
        }
    }

    private lateinit var sharedPreferences: SharedPreferences
    private lateinit var displayText: TextView
    ...
}
```

Lo primero que hacemos es implementar la interfaz en la firma de nuestra Activity class, separando la herencia de AppCompatActivity(). Recordando que podemos heredar de sólo una clase, pero podemos implementar todas las interfaces que necesitemos.

```
class MainActivity : AppCompatActivity(),
SharedPreferences.OnSharedPreferencesChangeListener
```

Luego implementamos el método que se define en la interfaz, el método onSharedPreferencesChanged. Este método tiene dos parámetros, que se envían cuando el archivo de SharedPreferences es modificado. El primer parámetro es el archivo de SharedPreferences, el segundo es la clave que cambió. En este caso estamos esperando el cambio de la clave "UserNickName", para desplegar ese nuevo nickName en un TextView, que se llama displayText, una variable definida en la Actividad.

```

override fun onSharedPreferencesChanged(
    changedSharedPreferences: SharedPreferences?,
    changedKey: String?) {

    if (changedKey.equals("UserNickName", true)) {
        val newNickName = changedSharedPreferences?
            .getString(changedKey, "")
        displayText.text = newNickName
    }
}

```

Para registrar, y desregistrar, un listener en un archivo de SharedPreferences ocupamos dos métodos que están disponibles en SharedPreferences:

- **registerOnSharedPreferenceChangeListener**: nos permite registrar un listener en un archivo de preferencias. Este listener será notificado cuando ocurra un cambio en dicho archivo. Este listener, para ser notificado, debe no ser reciclado en alguna de las pasadas del recolector de basura del sistema. SharedPreferences guarda siempre una referencia débil de los listeners, para evitar memory leaks con referencias circulares. Una referencia circular ocurre cuando A refiere a B y B refiere a A, en ese caso, cuando el recolector de basura quiere reciclar, A y B siempre tienen una referencia. Cuando un objeto siempre tiene una referencias, esa memoria no se libera, por lo que se produce un memory leak si ese objeto está sin utilizar, pero no se puede reciclar. Por lo que se recomienda que el listener sea siempre el que tiene la referencia a SharedPreferences, esta referencia debe ser fuerte, normalmente una referencia a this es fuerte. El método de registro recibe como parámetro un OnSharedPreferenceChangeListener listener, normalmente una clase que implementa la interfaz, o una implementación de la interfaz como propiedad de otra clase.
- **unregisterOnSharedPreferenceChangeListener**: este método nos permite desregistrar un listener del archivo de preferencias. Recibe como parámetro el listener que se quiere remover.

Podemos registrar un listener de la siguiente forma

```

sharedPreferences.registerOnSharedPreferenceChangeListener(this)

```

En este caso estamos registrando nuestra MainActivity, que implementa la interfaz OnSharedPreferenceChangeListener. Esta actividad será notificada cuando cambie sharedPreferences, y cambiará el texto en displayText, cuando la clave que cambió sea igual a UserNickName.

```
override fun onSharedPreferenceChanged(  
    changedSharedPreferences: SharedPreferences?,  
    changedKey: String?) {  
  
    if (changedKey.equals("UserNickName", true)) {  
        val newNickName = changedSharedPreferences?  
            .getString(changedKey, "")  
        displayText.text = newNickName  
    }  
}
```

También podemos definir una variable que sea parte de la Actividad, en este caso se puede implementar como un lambda, debido a que es una interfaz con un sólo método, un ejemplo de cómo podemos hacer esto es:

```
val listener = SharedPreferences  
    .OnSharedPreferenceChangeListener { preferences, key ->  
        if (key.equals("TestString", true)) {  
            displayText.text = preferences.getString(key, "")  
        }  
}  
sharedPreferences.registerOnSharedPreferenceChangeListener(listener)
```

En este caso definimos una val listener, que es asignada a una clase anónima con `object:SharedPreferences.OnSharedPreferenceChangeListener`. Esta clase define el método `onSharedPreferenceChanged`. Finalmente ocupamos la variable para registrarla como listener de sharedPreferences.

Debemos tener una referencia a este listener para poder desregistrar el mismo cuando ya no sea necesario escuchar los cambios de ese archivo de preferencias. Recordar que es importante desregistrar el listener para evitar posibles referencias circulares que pueden generar memory leaks. Para desregistrar, ocupamos el método de SharedPreferences `unregisterOnSharedPreferenceChangeListener`, pasando la referencia del listener que queremos desregistrar. El registro y desregistro normalmente se hacen en `onResume` y `onPause` respectivamente

```
sharedPreferences.unregisterOnSharedPreferenceChangeListener(this)  
sharedPreferences.unregisterOnSharedPreferenceChangeListener(listener)
```



## Aplicación de Ejemplo

Desarrollaremos un pequeño ejemplo para entender el uso de SharedPreferences en un aplicación real. En este ejemplo crearemos una aplicación que nos permite almacenar datos entre diferentes corridas. La aplicación que haremos es la siguiente:

- Partimos de una aplicación vacía desde crear un nuevo proyecto en Android Studio
- Nuestra aplicación contará con una pantalla que contiene los siguiente
  - Un input para ingresar un número
  - Un input para ingresar un Texto
  - Un switch para seleccionar
  - Un input para ingresar un número decimal
  - Un botón para salvar todo y uno para borrar todo
  - Textos para desplegar los diferentes tipos guardados

En esta aplicación usaremos todo lo aprendido hasta ahora.

## Crear proyecto inicial

Este es el paso más simple, desde la pantalla de Bienvenida de Android Studio utilizamos la opción Start a new Android Studio, y seguimos los paso para crear un Empty activity.

La secuencia se muestra en la imágenes siguientes, los datos utilizados para el proyecto son los siguiente:

- Name: EjemploSharedPreferences
- Package name: cl.desafiolatam.ejemplosharedpreferences
- Lenguaje: Kotlin
- Minimum API level: 23, Marshmallow

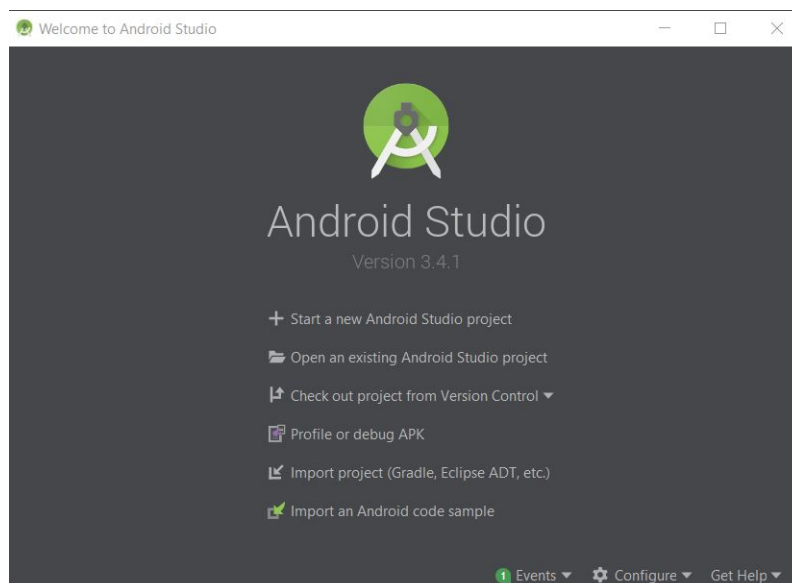


Imagen 3. Pantalla de bienvenida.

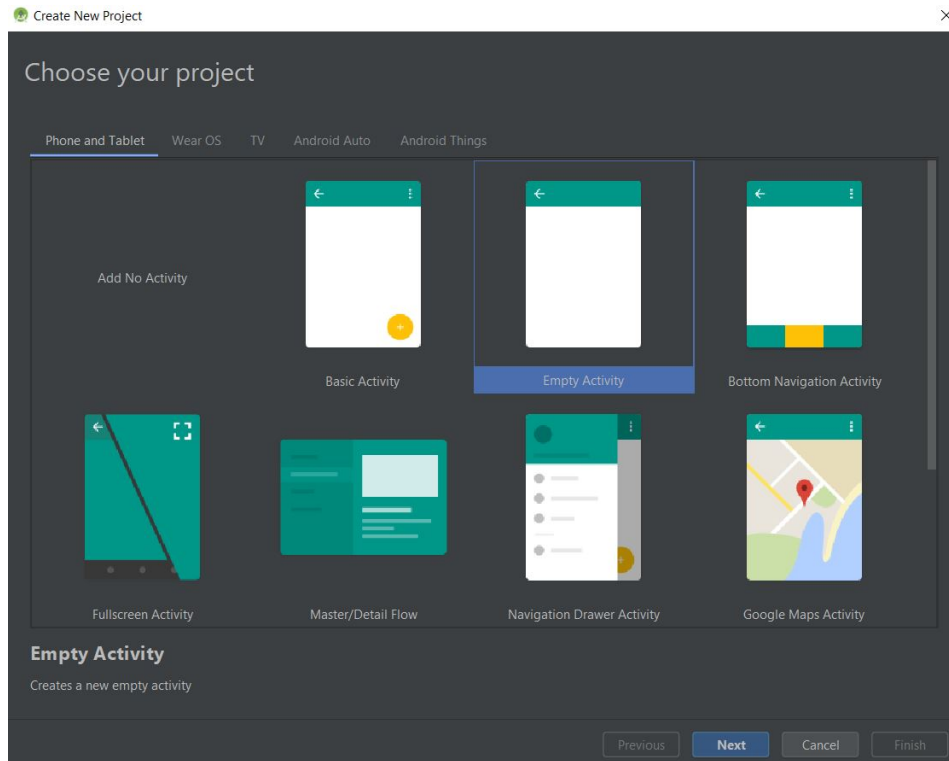


Imagen 4. Pantalla de selección de EmptyActivity.

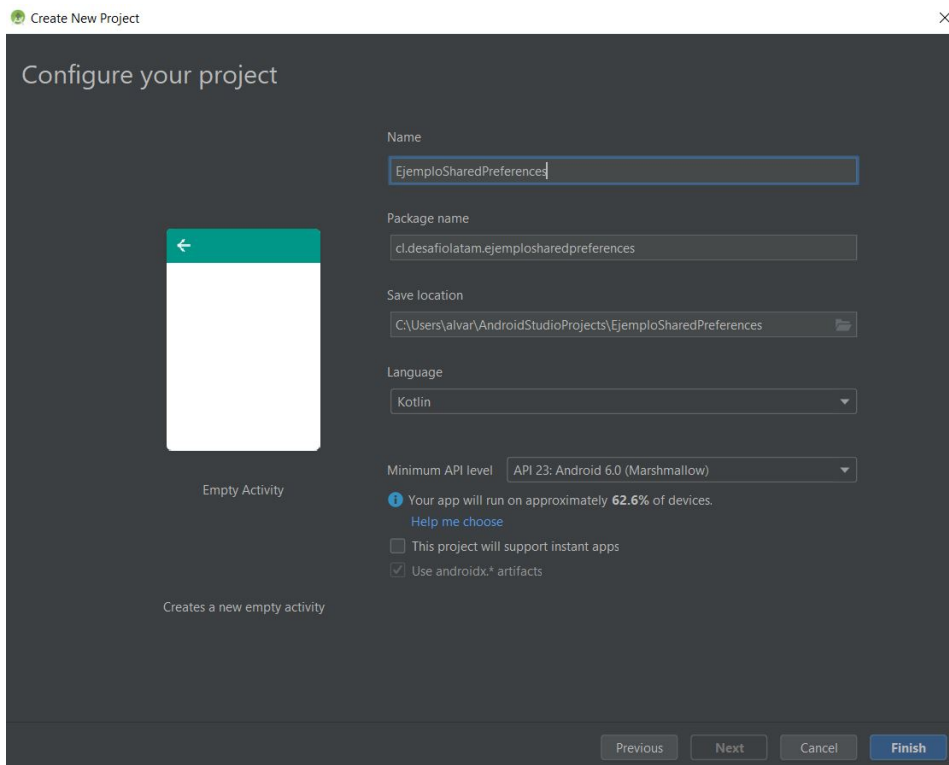
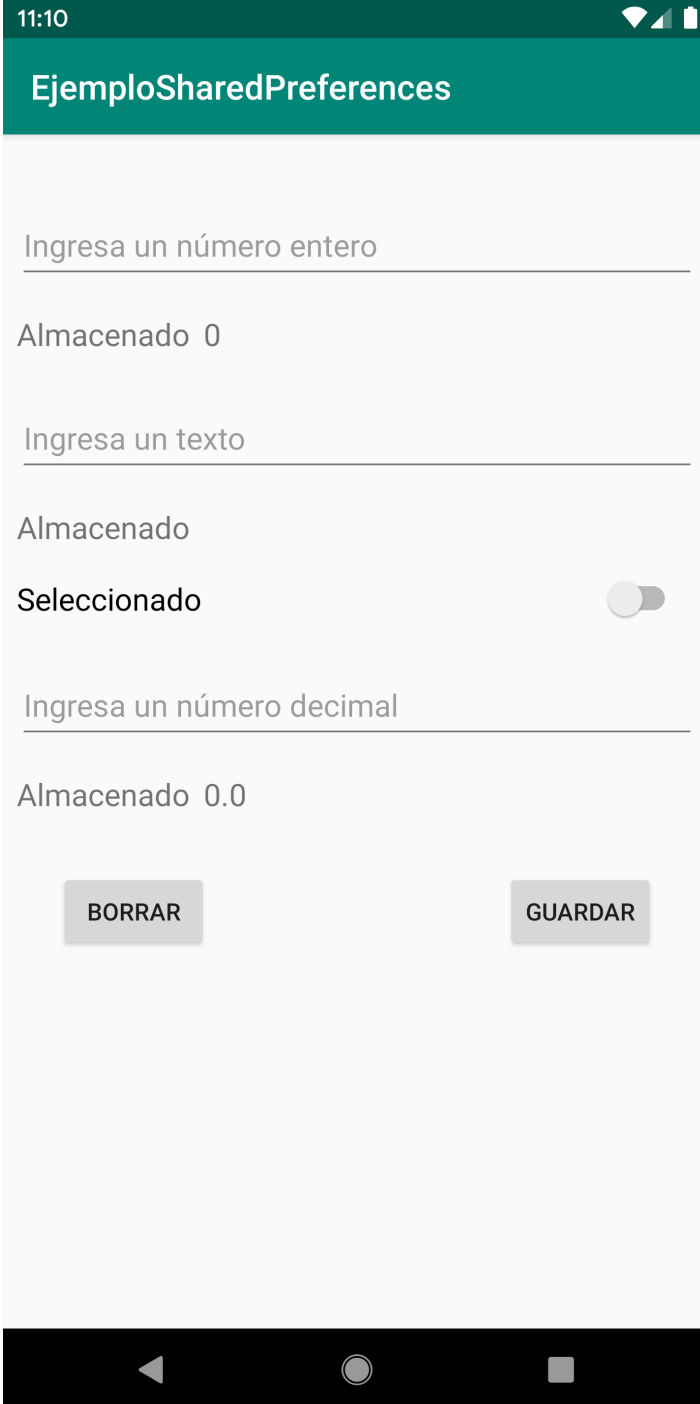


Imagen 5. Configurando el proyecto

Después de presionar Finish, nuestro proyecto inicial está listo. No necesitamos configurar nada más en este proyecto, pongamos manos a la obra.

## Pantalla de la aplicación

Para tener de referencia implementaremos una pantalla para esta aplicación con los componentes que podemos ver en la imagen



The screenshot shows an Android application interface with a teal header bar containing the title "EjemploSharedPreferences". The status bar at the top displays the time "11:10" and icons for Wi-Fi, cellular signal, and battery. The main content area is light gray and contains three input sections. The first section has a text input field with the placeholder "Ingresa un número entero", followed by the text "Almacenado 0". The second section has a text input field with the placeholder "Ingresa un texto", followed by the text "Almacenado" and a toggle switch labeled "Seleccionado" which is currently turned off. The third section has a text input field with the placeholder "Ingresa un número decimal", followed by the text "Almacenado 0.0". At the bottom of the form are two gray buttons: "BORRAR" on the left and "GUARDAR" on the right. The bottom of the screen shows the standard Android navigation bar with back, home, and recents icons.

Imagen 6. Ejemplo de pantalla a implementar.

Para no perder mucho tiempo en esta pantalla, debemos implementar rápidamente el siguiente código XML.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <com.google.android.material.textfield.TextInputLayout
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="32dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_marginEnd="8dp"
        app:layout_constraintStart_toStartOf="parent"
        android:layout_marginStart="8dp"
        android:id="@+id/numero_container">

        <com.google.android.material.textfield.TextInputEditText
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Ingresa un número entero"
            android:id="@+id/numero_input"
            android:inputType="number"/>
    </com.google.android.material.textfield.TextInputLayout>
    <TextView
        android:text="Almacenado"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/num_ent_title"
        app:layout_constraintTop_toBottomOf="@+id/numero_container"
        app:layout_constraintStart_toStartOf="parent"
        android:layout_marginStart="8dp"
        android:layout_marginTop="16dp"
        android:textSize="18sp"/>
    <TextView
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:id="@+id/num_ent_valor"
        app:layout_constraintStart_toEndOf="@+id/num_ent_title"
        android:layout_marginStart="8dp"
```

```

        app:layout_constraintEnd_toEndOf="parent"
        android:layout_marginEnd="8dp"
        app:layout_constraintTop_toBottomOf="@+id/numero_container"
        android:layout_marginTop="16dp"
        android:textSize="18sp"/>
<com.google.android.material.textfield.TextInputLayout
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    app:layout_constraintTop_toBottomOf="@+id/num_ent_valor"
    app:layout_constraintStart_toStartOf="parent"
    android:layout_marginStart="8dp"
    android:layout_marginEnd="8dp"
    app:layout_constraintEnd_toEndOf="parent"
    android:id="@+id/texto_container">

    <com.google.android.material.textfield.TextInputEditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Ingresa un texto"
        android:id="@+id/texto_input"
        android:inputType="text" />
</com.google.android.material.textfield.TextInputLayout>
<TextView
    android:text="Almacenado"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/text_title"
    app:layout_constraintTop_toBottomOf="@+id/texto_container"
    app:layout_constraintStart_toStartOf="parent"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:textSize="18sp" />
<TextView
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:id="@+id/text_valor"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_marginEnd="8dp"
    app:layout_constraintTop_toBottomOf="@+id/texto_container"
    app:layout_constraintStart_toEndOf="@+id/text_title"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:textSize="18sp" />
<Switch
    android:text="Seleccionado"

```

```

        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:id="@+id/switch_input"
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_marginEnd="16dp"
        app:layout_constraintStart_toStartOf="parent"
        android:layout_marginStart="8dp"
        app:layout_constraintTop_toBottomOf="@+id/text_valor"
        android:layout_marginTop="16dp"
        android:textSize="18sp" />
<com.google.android.material.textfield.TextInputLayout
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    app:layout_constraintTop_toBottomOf="@+id/switch_input"
    android:layout_marginEnd="8dp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    android:layout_marginStart="8dp"
    android:id="@+id/num_dec_container">

    <com.google.android.material.textfield.TextInputEditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Ingresa un número decimal"
        android:id="@+id/num_dec_input"
        android:inputType="number|numberDecimal" />
</com.google.android.material.textfield.TextInputLayout>
<TextView
    android:text="Almacenado"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/num_dec_title"
    android:layout_marginTop="16dp"
    app:layout_constraintTop_toBottomOf="@+id/num_dec_container"
    app:layout_constraintStart_toStartOf="parent"
    android:layout_marginStart="8dp"
    android:textSize="18sp" />
<TextView
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:id="@+id/num_dec_valor"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="16dp"
    app:layout_constraintTop_toBottomOf="@+id/num_dec_container"

```

```

        app:layout_constraintStart_toEndOf="@+id/num_dec_title"
        android:layout_marginStart="8dp"
        android:textSize="18sp" />

<Button
    android:text="Guardar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/guardar"
    android:layout_marginTop="32dp"
    app:layout_constraintTop_toBottomOf="@+id/num_dec_valor"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_marginEnd="32dp" />

<Button
    android:text="Borrar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/borrar"
    android:layout_marginTop="32dp"
    app:layout_constraintTop_toBottomOf="@+id/num_dec_valor"
    app:layout_constraintStart_toStartOf="parent"
    android:layout_marginStart="32dp"

/>
</androidx.constraintlayout.widget.ConstraintLayout>

```

Utilizamos un `ConstraintLayout` para mantener todos nuestros controles ordenados, con un id, que se utilizará en el código para referenciarlos. Notar que los inputs definen sus teclados para evitar problemas de valores, `number` para el número entero, `numberDecimal` para el número decimal. Esto se hace agregando la propiedad `android:inputType="number"` y `android:inputType="numberDecimal"` a los `TextInputEditText` correspondientes. Ahora veremos que es lo necesario para utilizar `SharedPreferences` en esta aplicación.

## Actividad Principal, variables y métodos

En nuestra aplicación abrimos el archivo de MainActivity.kt. Podemos ver que en su estado inicial sólo contamos con el esqueleto y un método onCreate donde se inicializa la actividad y se configura su vista en xml. El código debería verse de la siguiente manera

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
    }  
}
```

Lo primero que debemos hacer es crear una variable para mantener nuestro archivo referenciado durante todo el ciclo de esta aplicación, en este caso nos basta con utilizar una propiedad de tipo var en MainActivity. Agreguemos el siguiente código, recordado que esta variable debe ser lateinit para ser iniciada en onCreate, el código debe ser agregado antes de la función onCreate.

```
class MainActivity : AppCompatActivity() {  
    lateinit var sharedPreferences: SharedPreferences  
    override fun onCreate(savedInstanceState: Bundle?) {
```

También agregamos dentro del método onCreate, después de setContentView, la inicialización de sharedPreferences, con el nombre del paquete y en modo privado:

```
        override fun onCreate(savedInstanceState: Bundle?) {  
            super.onCreate(savedInstanceState)  
            setContentView(R.layout.activity_main)  
            val fileName = "cl.desafiolatam.ejemplosharedpreferences"  
            sharedPreferences = getSharedPreferences(fileName, Context.MODE_PRIVATE)  
        }
```



Hemos creado nuestro archivo de manera correcta, y en el modo correcto. Ahora necesitamos referenciar todos nuestros controles y los nombres de las claves que utilizaremos para almacenar y leer los datos desde el archivo, en este caso usaremos las siguientes variables que agregamos después de nuestra var de sharedPreferences y antes de onCreate. Además agregamos un companion object para definir nuestras claves, usamos el siguiente código:

```
class MainActivity : AppCompatActivity() {
    companion object {
        private const val baseClave = "EjemploSP"
        const val claveNumEnt = baseClave + "NumEnt"
        const val claveTexto = baseClave + "Texto"
        const val claveSwitch = baseClave + "Switch"
        const val claveNumDec = baseClave + "NumDec"
    }
    lateinit var sharedPreferences: SharedPreferences
    lateinit var numEntInput: TextInputEditText
    lateinit var numEntValue: TextView
    lateinit var textoInput: TextInputEditText
    lateinit var textoValue: TextView
    lateinit var switch: Switch
    lateinit var numDecInput: TextInputEditText
    lateinit var numDecValue: TextView
    lateinit var borrar: Button
    lateinit var guardar: Button
}
```

Recordemos que un companion object en Kotlin es lo mismo que definir variables estáticas en java, y estas variables son compartidas por todas las instancias de la clase que las implementa.

Para inicializar nuestros widgets variables escribimos un método, en el cuerpo de Main Activity, que llamaremos después de inicializar el archivo de sharedPreferences en el método onCreate, el método que utilizamos para inicializar los widgets o controles de la pantalla es el siguiente:

```
private fun initView() {
    numEntInput = findViewById(R.id.numero_input)
    numEntValue = findViewById(R.id.num_ent_valor)
    textoInput = findViewById(R.id.texto_input)
    textoValue = findViewById(R.id.text_valor)
    switch = findViewById(R.id.switch_input)
    numDecInput = findViewById(R.id.num_dec_input)
    numDecValue = findViewById(R.id.num_dec_valor)
    borrar = findViewById(R.id.borrar)
    guardar = findViewById(R.id.guardar)
}
```

```

borrar.setOnClickListener(object: View.OnClickListener {
    override fun onClick(p0: View?) {
        borrar()
    }
})

guardar.setOnClickListener(object: View.OnClickListener {
    override fun onClick(p0: View?) {
        guardar()
    }
})
}

```

En este método también definimos los listeners necesarios para que el botón borrar y guardar funcionen. En el método borrar() y guardar() utilizamos SharedPreferences para almacenar y eliminar lo que uno agrega como usuario al archivo que hemos creado. Además creamos un tercer método que nos permite leer los datos almacenados desde el archivo de SharedPreferences y actualizar los controles necesarios para mostrar esos datos al usuario. Veamos estos métodos y cómo utilizan SharedPreferences.

## Método: leerDesdeSharedPreferences()

Este método se utiliza para leer los diferentes datos almacenados en el archivo de sharedPreferences que creamos y cargar los datos iniciales en la aplicación. La primera vez que corremos la aplicación, tendremos sólo valores por defecto, pero en las siguientes corridas se mostrarán los datos almacenados por el usuario al presionar guarda. Definimos el método de la siguiente manera:

```

private fun leerDesdeSharedPreferences() {
    numEntValue.text = sharedPreferences.getInt(claveNumEnt, 0).toString()
    textoValue.text = sharedPreferences.getString(claveTexto, "")
    val checked = sharedPreferences.getBoolean(claveSwitch, false)
    switch.isChecked = checked
    switch.isSelected = checked
    numDecValue.text = sharedPreferences.getFloat(claveNumDec, 0.0f).toString()
}

```

En este método, que llamamos en onCreate después de inicializar el archivo en sharedPreferences, hacemos lo siguiente:

- En el TextView numEntValue mostraremos el valor almacenado para el número entero que podemos ingresar en la aplicación. Como vemos en el código utilizamos el método getInt, con la clave claveNumEnt definida en nuestro companion object, y tenemos un valor por defecto 0, para mostrar el texto en un TextView debemos convertir a string el valor, usamos el método toString para esto. Mientras el usuario no guarde un valor, el valor mostrado será 0.
- En el TextView textoValue mostramos el texto almacenado por el usuario. Utilizamos getString para leer el valor, con la clave claveTexto y un valor por defecto vacío. Mientras el usuario no almacene algo nuevo, el texto mostrado será vacío.
- En el Switch del mismo nombre mostramos si el usuario ha seleccionado o no este switch. El valor de esto lo almacenamos como un boolean, por lo que para obtener el valor almacenado utilizamos getBoolean, con la clave claveSwitch y un valor por defecto false. Notar que se debe setear el valor de isChecked y isSelected para que el switch muestre el estado correcto.
- En el TextView numDecValue mostramos el valor del número decimal almacenado, para obtener el valor utilizamos getFloat, con la clave claveNumDec y un valor por defecto de 0.0

En este método utilizamos lo que nos ofrece SharedPreferences para leer el archivo, ahora veamos cómo almacenar y eliminar datos.

## Método guarda()

En el método guardar ocupamos el editor para almacenar los diferentes datos que se pueden ingresar en la aplicación, el código de este método es el siguiente:

```
private fun guardar() {
    if (numEntInput.text!!.isEmpty()) {
        val numEnt = numEntInput.text.toString().toInt()
        sharedPreferences.edit().putInt(claveNumEnt, numEnt).apply()
    }
    if (textoInput.text!!.isEmpty()) {
        sharedPreferences.edit()
            .putString(claveTexto, textoInput.text.toString())
            .apply()
    }
    val checked = switch.isChecked
    sharedPreferences.edit().putBoolean(claveSwitch, checked).apply()
    if (numDecInput.text!!.isEmpty()) {
        val numDec = numDecInput.text.toString().toFloat()
        sharedPreferences.edit().putFloat(claveNumDec, numDec).apply()
    }
}
```

En este método utilizamos edit() y almacenamos diferentes tipos de datos de la siguiente manera:

- Lo primero es almacenar el número entero que el usuario ingresa en numEntInput. Como tenemos un texto, no podemos llegar y almacenar el valor, lo primero es ver si no está vacío. Si no es vacío, convertimos el valor a string y luego int, almacenando el mismo en numEnt. Luego lo almacenamos con edit.putInt(claveNumEnt, numEnt).apply(), de esta forma nos aseguramos de que la aplicación no se caiga y que no se almacenen datos vacíos.
- Para almacenar el texto en textInput, también vemos si no es vacío. Si pasa esta prueba podemos almacenar el valor desde el input, convirtiéndolo con toString. Para almacenar utilizamos putString(claveTexto,...).apply().
- Para el switch almacenaremos un boolean, rescatamos el valor de isChecked en val checked y luego utilizamos edit().putBoolean(claveSwitch, checked).apply() para almacenar el valor en el archivo. Nunca es vacío así que no necesitamos chequear este valor.
- Para el número decimal, chequeamos que no esté vacío. Utilizamos numDec para almacenar el valor no vacío convertido, y lo guardamos usando edit().putFloat(claveNumDec, numDec).apply().

Ahora veamos cómo borrar estos valores del archivo.

## Método: borrar()

En este método utilizamos edit().remove para eliminar los datos guardados, definimos este método de la siguiente forma:

```
private fun borrar() {  
    sharedPreferences.edit().remove(claveNumEnt).apply()  
    sharedPreferences.edit().remove(claveTexto).apply()  
    sharedPreferences.edit().remove(claveSwitch).apply()  
    sharedPreferences.edit().remove(claveNumDec).apply()  
}
```

Este método es bastante simple, sólo debemos notar que no hay problemas en borrar claves que no existe, si el archivo no las encuentra no hace nada y el método apply se ejecuta correctamente.

Para finalizar nuestro ejemplo debemos modificar el código en el método onCreate de la siguiente forma:

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
    val fileName = "cl.desafiolatam.ejemplosharedpreferences"  
    sharedPreferences = getSharedPreferences(fileName, Context.MODE_PRIVATE)  
    initView()  
    leerDesdeSharedPreferences()  
}
```

Notar el orden de los métodos `initViews` y `leerDesdeSharedPreferences`, si no utilizamos ese orden la aplicación se caerá por null pointers en las variables que referencian a los controles en pantalla. Luego de modificar el método `onCreate`, y hemos agregado todo el código a `MainActivity`, presionamos run app, el botón play en Android Studio. Si todo está bien debemos ver la siguiente pantalla:

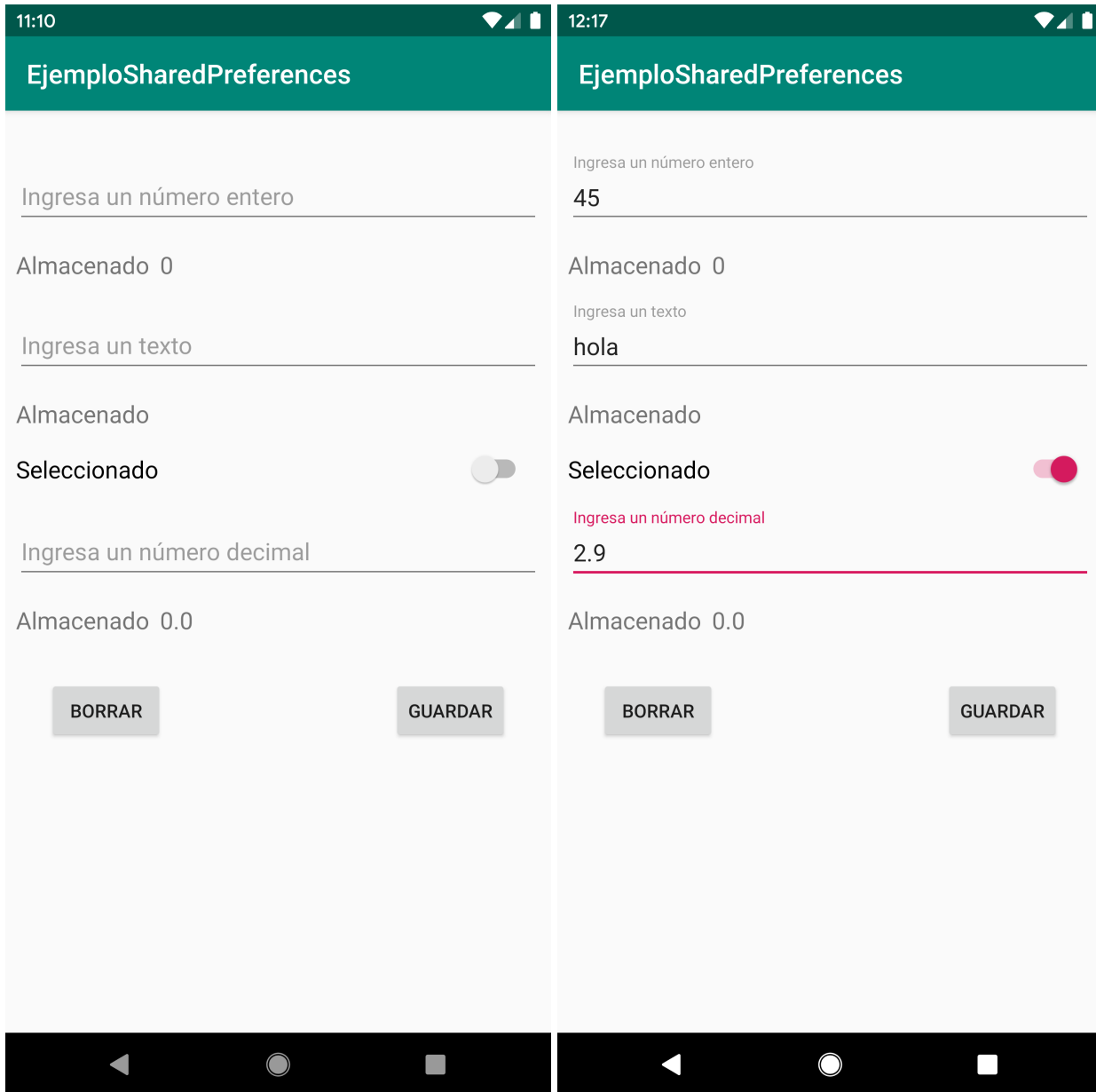


Imagen 7 y 8. Pantallas al ejecutar la aplicación.

Después de iniciar la aplicación podemos agregar algunos datos, para luego presionar guardar. Después de guardar debemos cerrar la aplicación. Al abrirla nuevamente veremos los datos cargados en los distintos controles como vemos en la imagen a continuación:

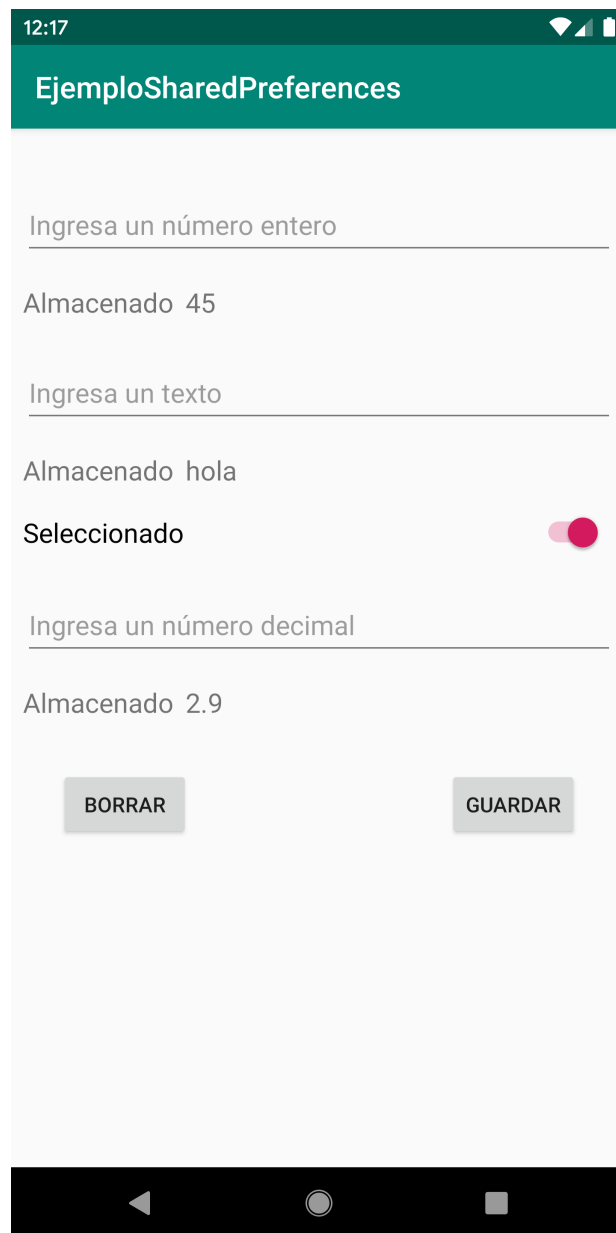


Imagen 9. Datos cargados posterior al haber cerrado la aplicación.

Así finalizamos el ejemplo. A continuación veremos Room.