ViewModel

ViewModel

```
dependencies
{
    : : :
    implementation 'androidx.lifecycle:lifecycle-viewmodel-savedstate:2.5.1'
    implementation 'androidx.activity:activity-ktx:1.6.1'
    implementation 'androidx.fragment:fragment-ktx:1.5.5'
    : : :
}
```

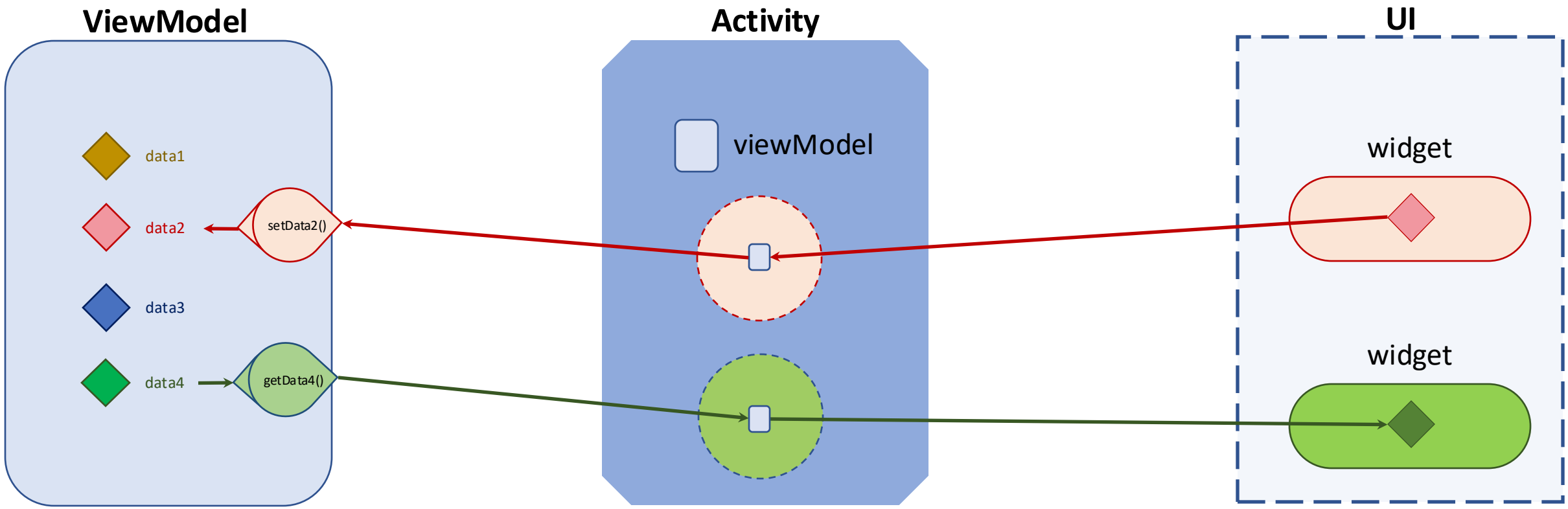
ViewModel

Es un objeto que contiene los datos y lógica de la vista (Activity)

Su ámbito trasciende el ciclo de vida de la Activity, ya que pervive a los cambios de configuración del dispositivo (por ejemplo rotaciones)

ViewModel

Esquema de uso



1 Los datos se encuentran en el ViewModel

2 La lógica se encuentra también en ViewModel

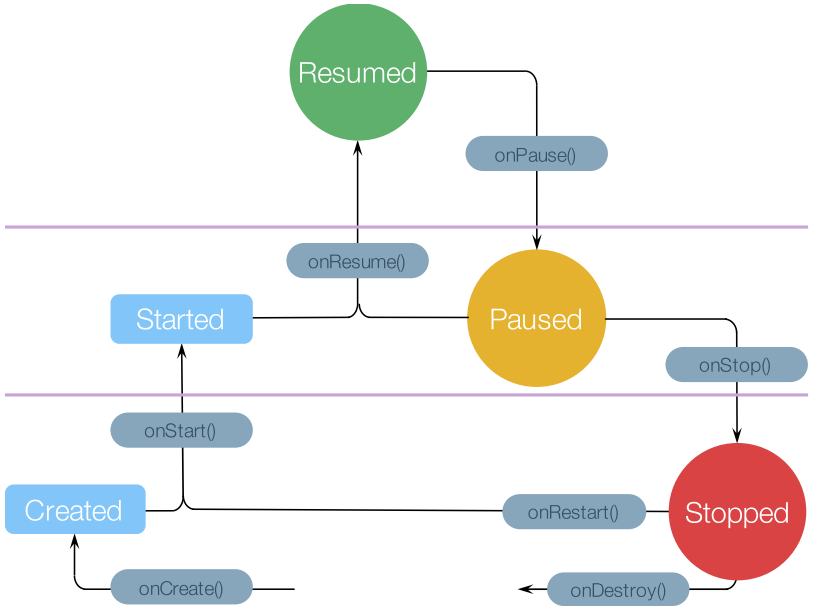
a La actividad tiene la referencia del ViewModel

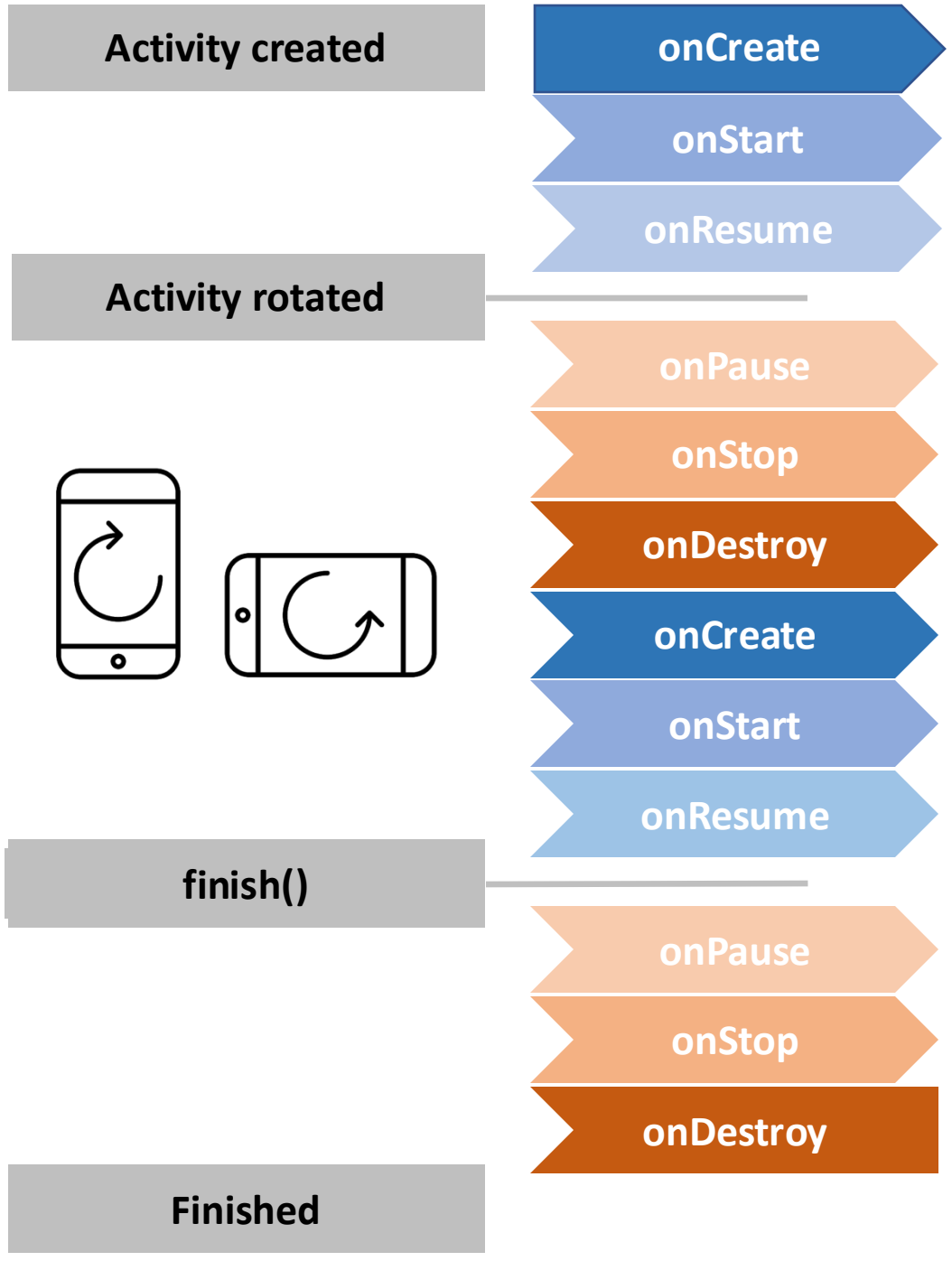
b Invoca a métodos de ViewModel para acceder a los datos →

c Invoca a métodos de ViewModel para notificar eventos de la UI ←

ViewModel

Escenario A - *scope*





ViewModel

Escenario A - Instanciación

1 2 3

El *viewmodel* debe extender de **androidx.lifecycle.ViewModel**

ViewModel

```
import androidx.lifecycle.ViewModel

class MainViewModel : ViewModel()
{

    : : :

}
```

La creación del *viewmodel* en la actividad debe realizarse delegándola en la función **viewModels()**

Activity

```
class MainActivity : AppCompatActivity()
{
  private val viewModel : MainViewModel by viewModels()

  override fun onCreate(savedInstanceState: Bundle?)
  {
    super.onCreate(savedInstanceState)
    : : :
  }
}
```

La creación del *viewmodel* en un composable debe realizarse delegándola en la función **viewModel()**

Compose

```
@Composable
fun Screen()
{
  val viewModel: MainViewModel = viewModel();
  : : :
}
```

ViewModel

Escenario A - Instanciación

1 2 3

Application

```
class CustomersApp: Application()
{
    :::
}
```

Si el *viewmodel* necesitase de *application*, tendría que extender de **androidx.lifecycle.AndroidViewModel**

ViewModel

```
import androidx.lifecycle.ViewModel

class CustomersViewModel(application: Application) : AndroidViewModel(application)
{
    : : :
    {
        CustomersApp customersApp = getApplication<CustomerApp>();
    }
}
```

Compose

```
@Composable
fun CustomersScreen()
{
  val viewModel: CustomersViewModel = viewModel();
  :::
}
```

En este caso, no habría cambios en la creación del *viewmodel* dentro del *acomposable*

Activity

```
class CustomersActivity : AppCompatActivity()
{
  private val viewModel : CustomersViewModel by viewModels()
   : : :
}
```

En este caso, no habría cambios en la creación del *viewmodel* dentro de la *activity*

ViewModel

Escenario A - Instanciación

1 2 3

Compose

```
@Composable
fun CustomersScreen()
{
  val factory = CustomersViewModel.Factory()
  val viewModel: CustomersViewModel = viewModel( factory )
  : : :
}
```

Application

```
class CustomersApp: Application()
{
   val repository: Repository = Repository()
   :::
}
```

Activity

```
class CustomersActivity : AppCompatActivity()
{
  val factory = CustomersViewModel.Factory()
  val viewModel: CustomersViewModel by viewModels { factory }
  : : :
}
```

Al delegado **viewModels** hay que proporcionarle como parámetro el *factory*.

Si el constructor del *viewmodel* define otros parámetros, para poder instanciarlo será preciso utilizar un *factory*. Este *factory* debe implementar **androidx.lifecycle.ViewModelProvider.Factory**

ViewModel

```
class CustomersViewModel(val repository: Repository) : ViewModel()
{
  : : :
```

Factory

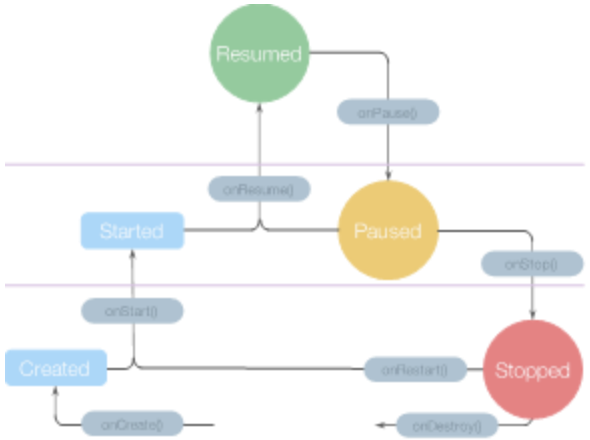
```
    class Factory : ViewModelProvider.Factory
    {
       override fun <T : ViewModel> create(modelClass: Class<T>, extras: CreationExtras): T
       {
          val application : Application = extras[ViewModelProvider.AndroidViewModelFactory.APPLICATION_KEY]!!
          return CustomersViewModel((application as CustomersApp).repository) as T
       }
    }
}
```

Su método **ViewModelProvider.Factory#create()** será el encargado de instanciar y devolver el objeto *viewmodel*. Al constructor del *factory* habría que proporcionarle el valor de los parámetros requeridos por *viewmodel* que no pudieran obtenerse a través de **android.app.Application**.

CreationExtras un *map* que facilita el acceso a datos útiles para la creación del *viewmodel*. Entre otras, incluye una entrada con la clave **ViewModelProvider.AndroidViewModelFactory.APPLICATION_KEY** que almacena la instancia de **Application**

ViewModel

Escenario B - *scope*



Activity created

onCreate

onStart

onResume

Se cambia la actividad

onPause

onStop

El S.O. necesita recursos y para obtenerlos decide matar la Activity

onDestroy

Se regresa a la Activity

onCreate

onStart

onResume

finish()

onPause

onStop

onDestroy

Finished

ViewModel *scope* 1

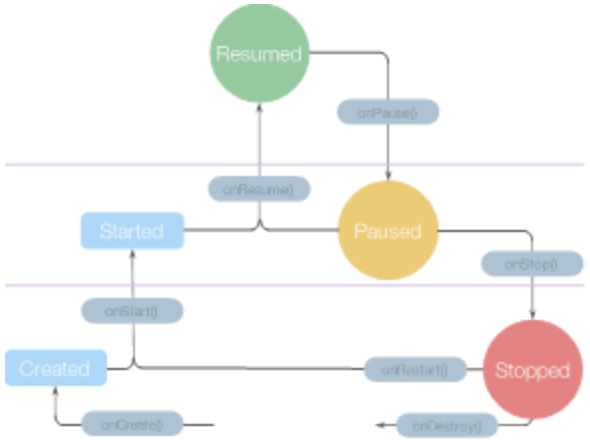
onCleared

ViewModel *scope* 2

onCleared

ViewModel

Escenario B - *scope*



Activity created

onCreate

onStart

onResume

Se cambia la actividad

onPause

onSaveInstanceState(Bundle bundle)

onStop

El S.O. necesita recursos y para obtenerlos decide matar la Activity

onDestroy

Se regresa a la Activity

onCreate

onStart

onResume

finish()

onPause

onStop

onDestroy

Finished

ViewModel *scope*

1

SavedStateHandle

onCleared

SavedStateHandle

ViewModel *scope*

2

onCleared

ViewModel

Escenario B - Instanciación

1 2 3

El constructor del *viewmodel* recibe como parámetro un **`SavedStateHandle`**

`SavedStateHandle` se comporta como un pseudo Map (tiene métodos **`contains()`**, **`keys()`**, **`get()`** y **`set()`**) donde podrá almacenarse propiedades. Estas propiedades tienen las mismas restricciones que en **`Bundle`** (la clave debe ser un String y los valores podrían ser tipos fundamentales de dato, **`String`**, objetos **`Serializable`**, **`Parcelable`** y **`Arrays`** de cualquiera de los tipos anteriores).

Los datos del viewmodel que necesiten «persistencia» deberán almacenarse como propiedades en **`SavedStateHandle`**.

La «persistencia» se realizará en **`Activity#onSaveInstanceState()`** del *activity* que creo el *viewmodel*

`SavedStateHandle` incluye entre sus propiedades todos los **`EXTRAS`** del **`Intent`** que arrancó la *activity* que creo el *viewmodel*

ViewModel

```
class CustomersViewModel(val ssh: SavedStateHandle) : ViewModel()
{
   : : :
}
```

Compose

```
@Composable
fun CustomersScreen()
{
 val viewModel: CustomersViewModel = viewModel()
  : : :
}
```

La creación del *viewmodel* en la actividad debe realizarse delegándola en la función **viewModels()**

Activity

```
class MainActivity : AppCompatActivity()
{

  private val viewModel : CustomersViewModel by viewModels()

  : : :
}
```

ViewModel

Escenario B - Instanciación

1 2 3

Si el *viewmodel* necesitase tanto del **SavedStateHandle** como del objeto **android.app.Application**, al igual que sucedía en el escenario anterior, la clase del *viewmodel* tendrá que extender de la clase **AndroidViewModel**

Posteriormente se podrá acceder a este objeto a través del método **Application getApplication<>()**

El orden de los parámetros obligatoriamente debe ser este: 1º **Application**, 2º **SavedStateHandle**

Application

```
class CustomersApp: Application()
{
    :::
}
```

ViewModel

```
class CustomersViewModel(application: Application , val ssh: SavedStateHandle) : AndroidViewModel(application)
{
   : : :
        CustomersApp customersApp = getApplication<CustomerApp>();
}
```

La creación del *viewmodel* en la actividad debe realizarse delegándola en la función **viewModels()**

Compose

```
@Composable
fun CustomersScreen()
{
 val viewModel: CustomersViewModel = viewModel()
  :::
}
```

Activity

```
class CustomersActivity : AppCompatActivity()
{
  private val viewModel : CustomersViewModel by viewModels()
   : : :
}
```

ViewModel

Escenario B - Instanciación

1 2 3

Compose

```
@Composable
fun CustomersScreen()
{
  val factory = CustomersViewModel.Factory()
  val viewModel: CustomersViewModel = viewModel( factory )
  : : :
}
```

Application

```
class CustomersApp: Application()
{
   val repository: Repository = Repository()
   : : :
}
```

Activity

```
class CustomersActivity : AppCompatActivity()
{
  val factory = CustomersViewModel.Factory()
  val viewModel: CustomersViewModel by viewModels { factory }
  : : :
}
```

Si el constructor del *viewmodel* incluye, además de **SavedStateHandle**, otros parámetros, para poder instanciarlo será preciso utilizar un *factory*. Este *factory* debe implementar **androidx.lifecycle.ViewModelProvider.Factory**

ViewModel

```
class CustomersViewModel(val repository: Repository, val ssh: SavedStateHandle) : ViewModel()
{
  : : :
    class Factory : ViewModelProvider.Factory
    {
       override fun <T : ViewModel> create(modelClass: Class<T>, extras: CreationExtras): T
       {
          val application : Application = extras[ViewModelProvider.AndroidViewModelFactory.APPLICATION_KEY]!!
          val repository : Respository = (application as CustomersApp).repository
          val savedStateHandle: SavedStateHandle = extras.createSavedStateHandle()
          return CustomersViewModel(repository, savedStateHandle) as T
       }
    }
}
```

Al constructor del *factory* habría que proporcionarle el valor de los parámetros requeridos por *viewmodel* que no pudieran obtenerse a través de **android.app.Application**.

CreationExtras#createSavedStateHandle() crea el objeto **SavedStateHandle** que será necesario pasarle al constructor del viewmodel

ViewModel

Escenario B - Instanciación

SavedStateHandle

`boolean contains(String key)`

Devuelve verdadero o falso en función de que el SavedStateHandle incluya o no una entrada con la clave key

`Set<String> keys()`

Obtiene un conjunto incluyendo la clave de todas las entradas del SavedStateHandle

`T get<T>(String key)`

Obtiene el valor vinculado a la clave key. Si no existiese devuelve null

`void set<T>(String key, T value)`

Añade una nueva entrada vinculando el valor *value* a la clave *key*.
Si ya existiese una entrada con dicha clave, se actualizaría su valor
Si se hubiese obtenido un LiveData para la clave key, también se actualizaría el LiveData y éste notificaría el cambio a los observers te tuviese suscritos

`T remove<T>(String key)`

Elimina del SavedStateHandle la entrada con clave key y devuelve el valor que tenía asociado.
Si no existiese una entrada para esa clave, se devolvería null.
Si previamente se hubiese obtenido un LiveData para esa clave, éste quedaría desvinculado del SavedStateHandle; así, si posteriormente se volviese a añadir una nueva entrada al SavedSateHandle con la misma clave, aquel LiveData no recibiría actualizaciones

`LiveData<T> getLiveData<T>(String key)`

Obtiene un LiveData incluyendo el valor vinculado a la clave key.
Si no existiese una entrada con la clave key, el LiveData devuelto estaría vacío

`LiveData<T> getLiveData<T>(String key, T defaultValue)`

Obtiene un LiveData incluyendo el valor vinculado a la clave key.
Si no existiese una entrada con la clave key, el LiveData devuelto incluiría el valor defaultValue

`StateFlow<T> getStateFlow<T>(String key, T initialValue)`

Obtiene un StateFlow incluyendo el valor vinculado a la clave key.
Si no existiese una entrada con la clave key, el StateFlow devuelto incluiría el valor initialValue