Data binding

Data binding

Data binding permite definir variables en los layous. Así, mediante el uso de expresiones, se puede asignar el valor de dichas variables en los atributos de las views.

A través de data binding se simplifica la programación de nuestras activities puesto que, por ejemplo, no será necesario la utilización de `findViewById()`. Data binding vincula directamente el *layout* con los datos de la aplicación.

La utilización de LiveData como variables, implicaría la permanente actualización de la UI

Data binding

Activación

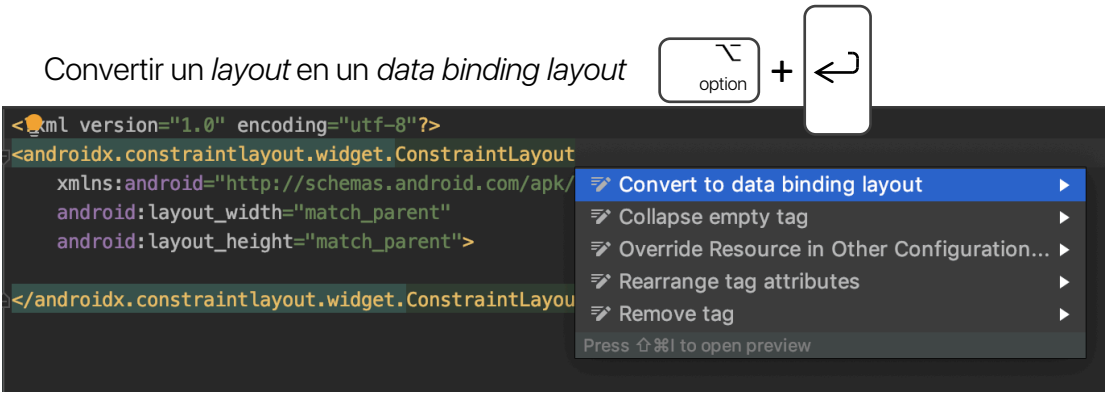
Module: build.gradle

```
: : :
android {
   : : :

   buildFeatures {
        dataBinding true

   }

}
```

Data binding

Data binding layout

Convertir un *layout* en un *data binding layout*




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

  <data>
    : : :
  </data>

  <androidx.constraintlayout.widget.ConstraintLayout
          xmlns:android="http://schemas.android.com/apk/res/android"
          android:layout_width="match_parent"
          android:layout_height="match_parent">

     : : :

  </androidx.constraintlayout.widget.ConstraintLayout>

</layout>
```

Aquí se definen las variables

Data binding

Data binding layout

```
import androidx.databinding.DataBindingUtil;

public class MainActivity extends AppCompatActivity
{
    private ActivityMainBinding dataBinding;


    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        dataBinding = DataBindingUtil.setContentView(this, R.layout.activity_main);
        : : :
    }
```

Se genera una clase cuyo nombre se deriva del identificador del layout (camel case sin subrayados más el sufijo ***Binding***)

Data binding

Data binding layout – Definición de variables

```
<?xml version="1.0" encoding="utf-8"?>
<layout >
    <data>
        <import type="es.upsa.mimo.v2021.app.BindingHelper"/>
        <import type="es.upsa.mimo.v2021.app.AppUtil"/>

        <variable
            name="indice"
            type="int" />

        <variable
            name="viewModel"
            type="es.upsa.mimo.v2021.app.MainViewModel" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity">
     : : :
```

También se permite realizar importaciones de datos. Generalmente estos tipos incluyen métodos static. La expresiones data binding podrán incluir la invocación a estos métodos static

Nombre variable

Definición de una variable

Tipo de dato

Si el propósitos de los view model es contener los datos que mostrará la activity, lo normal es que como varianles se definan los view model

Data binding

Data binding layout – Fijar valor de las variables

```xml
<?xml version="1.0" encoding="utf-8"?>
<layout>
  <data>
    <import type="es.upsa.mimo.v2021.app.BindingHelper"/>
    <import type="es.upsa.mimo.v2021.app.AppUtil"/>

    <variable
        name="indice"
        type="int" />

    <variable
        name="viewModel"
        type="es.upsa.mimo.v2021.app.MainViewModel"/>
  </data>

  <androidx.constraintlayout.widget.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity">
     : : :
```

```java
import androidx.databinding.DataBindingUtil;

public class MainActivity extends AppCompatActivity
{
  private ActivityMainBinding dataBinding;
  private MainViewModel viewModel;


  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    dataBinding = DataBindingUtil.setContentView(this, R.layout.activity_main);
    viewModel = new ViewModelProvider(this).get(MainViewModel.class);

    dataBinding.setIndice( 33 );
    dataBinding.setViewModel( viewModel );

    : : :
  }
```

El objeto data binding incluye setters por cada una de las variables definidas en le layout

Data binding

Expresiones. Asignación de valores – datos, arrays, List y Map

```
<layout>
  <data>
    <variable name="viewModel" type="es.upsa.mimo.app.MainViewModel"/>

  </data>

  <androidx.constraintlayout.widget.ConstraintLayout …>
    <TextView
       android:id="@+id/tvName"
       android:text="@{ viewModel.user.name }"
       : : : />
    <TextView
       android:id="@+id/tvAlias_3"
       android:text="@{ viewModel.user.alias[3] }"
       : : : />
    <TextView
       android:id="@+id/tvEmail2"
       android:text="@{ viewModel.user.emails[2] }"
       : : : />
    <TextView
       android:id="@+id/tvPhone_home"
       android:text="@{viewModel.user.phones[`phone`]) }"
       : : : />
  </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Arrays & Lists & Maps

es.upsa.mimo.app.MainViewModel

```
public class MainViewModel extends ViewModel
{
  private User user;

  : : :
  public User getUser() { return user; }
}
```

es.upsa.mimo.app.User

```
public class User
{
  private String name;
  private String[] alias;
  private List<String> emails;
  private Map<String, String> phones;
  : : :
  public String getName()
  {
    return name;
  }

  public String[] getAlias()
  {
    return alias;
  }

  public List[] getEmails()
  {
    return emails;
  }

  public Map<String, String> getPhones()
  {
    return phones;
  }
}
```

Data binding

Expresiones. Asignación de valores - *LiveData*

es.upsa.mimo.app.MainViewModel

```
public class MainViewModel extends ViewModel
{
  private MutableLiveData<User> user;

  : : :
  public LiveData<User> getUser() { return user; }
}
```

es.upsa.mimo.app.User

```
public class User
{
  private String name;
  private String[] alias;
  private List<String> emails;
  private Map<String, String> phones;
  : : :
  public String getName() { return name; }

  public String[] getAlias() { … }

  public List[] getEmails()  { … }

  public Map<String, String> getPhones() { … }
}
```

Si la expresión se evalúa como un LiveData, o contiene un LiveData, *data binding* crea un *observer* de forma que mantiene permanentemente actualizada la vista cuando el LiveData cambia su valor.

En la medida de lo posible, los valores de las vistas se facilitarán siempre a través de LiveData

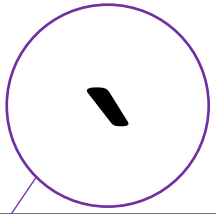
```
<layout>
  <data>
    <variable name="viewModel" type="es.upsa.mimo.app.MainViewModel"/>

  </data>

  <androidx.constraintlayout.widget.ConstraintLayout …>

    <TextView
       android:id="@+id/tvName"
       android:text="@{ viewModel.user.name }"
       : : : />

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

```
import androidx.databinding.DataBindingUtil;

public class MainActivity extends AppCompatActivity
{
  private ActivityMainBinding dataBinding;
  private MainViewModel viewModel;

  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    dataBinding = DataBindingUtil.setContentView(this, R.layout.activity_main);
    viewModel = new ViewModelProvider(this).get(MainViewModel.class);

    dataBinding.setViewModel( viewModel );
    dataBinding.setLifecycleOwner(this);
    : : :
  }
```

Si se trabaja con LiveData hay que indicar el LifeCycleOwnwer

Data binding

Expresiones. Asignación de valores - *recursos*



```
<layout>
  <data>
    <variable name="numero" type="int"/>
    <variable name="indice" type="int"/>
  </data>
  <androidx.constraintlayout.widget.ConstraintLayout …>
    <TextView
       android:id="@+id/tv1"
       android:text="@{ `Mes: ` + @string/enero }"
       : : : />
    <TextView
       android:id="@+id/tv2"
       android:text="@{ @string/formato(`Carlos`, `bueno`) }"
       : : : />
    <TextView
       android:id="@+id/tv3"
       android:text="@{ @plurals/mensajes(numero,numero) }"
       : : : />
    <TextView
       android:id="@+id/tv4"
       android:text="@{ @stringArray/numeros[indice] }"
       : : : />
    <TextView
       android:id="@+id/tv5"
       android:text="@{ @typedArray/meses.getString( indice ) }"
       : : : />
  </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

strings.xml

```
<resources>
  <string name="enero">Enero</string>
  <string name="febrero">Febrero</string>
  : : :
  <string name="diciembre">Diciembre</string>
  <string name="formato">Hola %1$s, hace %2$s </string>
</resources>
```

plurals.xml

```
<resources>
  <plurals name="mensajes">
    <item quantity="zero">No tienes mensajes</item>
    <item quantity="other">Tienes %1$d mensajes</item>
  </plurals>
</resources>
```

arrays.xml

```
<resources>
  <array name="numeros">
    <item>Uno</item>
    <item>Dos</item>
    : : :
    <item>Diez</item>
  </array>
</resources>
```

arrays.xml

```
<resources>
  <array name="meses">
    <item>@string/enero</item>
    <item>@string/febrero</item>
    : : :
    <item>@string/diciembre</item>
  </array>
</resources>
```

Data binding

Expresiones. Asignación de valores - *recursos*

Tipo	Referencia normal	Referencia en expresiones
`String[]`	`@array`	`@stringArray`
`int[]`	`@array`	`@intArray`
`TypedArray`	`@array`	`@typedArray`
`color int`	`@color`	`@color`
`ColorStateList`	`@color`	`@colorStateList`

Data binding

Expresiones. *Operadores*

Matemáticos	`+` `-` `/` `*` `%`
Concatenación Strings	`+`
Lógicos	`&&` `||`
Comparación	`==` `>` `>=` `<` `<=`
Binarios	`&` `|` `^`
Unarios	`+` `-` `!` `~`
Desplazamiento	`>>` `>>>` `<<`
Agrupación	`()`
Llamada a método	`()`
Acceso a campo	`.`
Acceso a elemento array	`[]`
Condicional	`?` `:`
Coalesce nulo	`??` Por ejemplo, la expresión: **@{vm.name ?? vm.lastName}** es equivalente a: **@{ (vm.name != null)? vm.name : vm.lastName }**

Data binding

Expresiones. Asignación de valores - *listeners*

```
<layout>
  <data>
    <variable name="handlers" type="es.upsa.mimo.app.Handlers"/>
    <variable name="viewModel" type="es.upsa.mimo.app.MainViewModel"/>
  </data>

  <androidx.constraintlayout.widget.ConstraintLayout …>
    <Buttom
       android:id="@+id/btOk"
       android:onClick="@{ handlers::onClickOk }"
       : : : />
    <Button
       android:id="@+id/btCancel"
       android:onClick="@{ (view) -> handlers.onClickCancel( view ) }"
       : : : />
    <Button
       android:id="@+id/btResume"
       android:onClick="@{ (view) -> handlers.onClickResume( view.id ) }"
       : : : />
    <Button
       android:id="@+id/btSendDefault"
       android:onClick="@{() -> viewModel.send( tvEmail.text ) }"
       : : : />
    <TextView
       android:id="@+id/tvEmail"
       android:text="@{viewModel.defaultEmail}"
       : : : />
  </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

es.upsa.mimo.app. Handlers

```
public class Handlers
{
  public void onClickOk(View view)
  {
   : : :
  }

  public void onClickCancel(View view)
  {
   : : :
  }

  public void onClickResume(int id)
  {
   : : :
  }


}
```

es.upsa.mimo.app.MainViewModel

```
public class MainViewModel extends ViewModel
{
  : : :
  public void send(String email)
  {
    : : :
  }

  public String getDefaultEmail()
  {
    : : :
  }
}
```

Data binding

Expresiones. Asignación de valores - **@BindingConversion**

Data Binding permite asignar (y actualizar dinámicamente) el valor de las propiedades de la *view*. Para ello, en el *layout*, a los atributos de la view se les vincula expresiones @{...} mediante las que se especifica el valor a asignar.

```
<TextView
        android:id="@+id/tvName"
        android:text="@{viewModel.user.name}"
        : : : />
```

A la propiedad text del **`TextView`** se le asigna el valor en el que se evalúa la expresión **`@{...}`**

Para realizar estas asignaciones *data binding* invoca al correspondiente método *setter* de la *view* (en el ejemplo invocaría al método `TextView#setText()`). Esto quiere decir que debe haber concordancia entre el tipo de dato de la expresión y el tipo de dato admitido en el setter, de forma que si fuesen tipos distintos se produciría un error.

Data binding permite definir *conversions* con las que evitar estos errores. Los *conversions* transforman el dato de la expresión a alguno de los tipos soportados por los *setter*. Los *conversions* serán invocados implícitamente por *data binding*.

Data binding

Expresiones. Asignación de valores - **@BindingConversion**

```
public class MainViewModel extends ViewModel
{
  public LiveData<User> user;
  : : :
}
```

```
<TextView
        android:id="@+id/tvName"
        android:text="@{viewModel.user.name}"
        : : : />
```

```
public class User
{
  private Name name;
  private String email;
  private String niick;
  : : :
  // getters & setters
}
```

Para que se pueda invocar a **TextView#setText(String)** es necesario definir una conversion que transforme **Name** a un **String**

```
public class Name
{
  private String firstName;
  private String lastName;
  : : :
  // getters & setters
}
```

```
import androidx.databinding.BindingConversion;

public class DataBindingHelpers
{
    @BindingConversion
    public static String nameToStringConverter(Name name)
    {
        return name.getFirstName() + " " + name.getLastname();
    }
}
```

"setter" parameter type

expression type

expression value

Data binding

Expresiones. Asignación de valores - **@BindingAdapter & @BindingMethod**

Data binding permite añadir nuevas propiedades a las *views*. Para ello es necesario indicar cómo se fijarán los valores en dichos propiedades (cómo realizar la función *setter*). Para realizar esta funcionalidad hay que definir o bien un *binding method* o bien un *binding adapter*.

1 **@BindingMethod**

Se utiliza un @BindingMethod cuando el setter es un método propio de la view

2 **@BindingAdapter**

Se utiliza un @BindingAdapter cuando cuando la view no posee el setter correspondiente

Data binding

Expresiones. Asignación de valores - **@BindingAdapter & @BindingMethod**

1 @BindingMethod

```
public class MainViewModel extends ViewModel
{
  public LiveData<User> user;
  : : :
}
```

```
<TextView
        android:id="@+id/tvName"
        name="@{viewModel.user.name}"
        : : : />
```

Se define un nuevo a tributo **name** en TextView

```
public class User
{
  private Name name;
  private String email;
  private String niick;
  : : :
  // getters & setters
}
```

```
public class Name
{
  private String firstName;
  private String lastName;
  : : :
  // getters & setters
}
```

```
import androidx.databinding.BindingConversion;

@BindingMethods({@BindingMethod(type = TextView.class,
                                attribute = "name",
                                method = "setText")
                 })
public class DataBindingHelpers
{
    @BindingConversion
    public static String nameToStringConverter(Name name)
    {
        return name.getFirstName() + " " + name.getLastname();
    }
}
```

"*setter*" parameter type

expression type

La anotación indica que en la *view* de tipo **TextView**, el valor de la propiedad **name** lo establece cualquiera de sus métodos **setText**

expression value

Data binding

Expresiones. Asignación de valores - **@BindingAdapter & @BindingMethod**

2 @BindingAdapter

Un *binding adapter* es un método **static** decorado con la anotación **@BindingAdapter**. El elemento **value** de la anotación indica el nombre de la propiedad (si en la *view* ya existiera dicha propiedad, el *binding adapter* redefiniría la forma en la que se haría el setter para dicha propiedad). El método, al menos, recibe dos parámetros: el primero representa el objeto *view* en el que se define la propiedad y el segundo representa el valor del atributo (valor en el que se evalúa ***expression*** en el *layout*)

```
<TextView
    android:id="@+id/widgetId"
    name="@{viewModel.user.name}"
    : : : />
```

```
public class AppBindingHelper
{
   @BindingAdapter("name")          1º          2º
   public static void setName(TextView textView, Name name)
   {
        textView.setText( name.getFullname() );
   }
}
```

Data binding

Expresiones. Asignación de valores - **@BindingAdapter & @BindingMethod**

2 @BindingAdapter

El *binding adapter también* podría recibir los valores anterior y nuevo de la propiedad

```
<TextView
   android:id="@+id/widgetId"
   name="@{viewModel.user.name}"
   : : : />
```

```
public class AppBindingHelper
{
  @BindingAdapter("name")          1º          2º          3º
  public static void setName(TextView textView, Name oldName , Name newName)
  {
      textView.setText( newName.getFullname() );
  }
}
```

Data binding

Expresiones. Asignación de valores - **@BindingAdapter & @BindingMethod**

2 @BindingAdapter

El *binding adapter también* podría definir propiedades "*listeners*"

```
<layout>
  <data>
    <variable name="viewModel" type="es.upsa.mimo.app.MainViewModel"/>
  </data>

  <androidx.constraintlayout.widget.ConstraintLayout …>
    <Buttom
       android:id="@+id/btSend"
       onClickListener="@{ () -> viewModel.send( tvEmail.text ) }"
       : : : />
    <Button
       android:id="@+id/btCancel"
       onClickListener="@{ viewModel::cancel }"
       : : : />
    <TextView
       android:id="@+id/tvEmail"
       android:text=…
       : : : />
 </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

es.upsa.mimo.app.MainViewModel

```
public class MainViewModel extends ViewModel
{
   : : :
   public void send(String email)
   {
    : : :
   }

   public void cancel()
   {
    : : :
   }
}
```

java.util.function.Consumer

```
public interface Consumer
{
   public void apply();
}
```

es.upsa.mimo.app. AppBindingHelper

```
public class AppBindingHelper
{
   @BindingAdapter("onClickListener")
   public static void setOnCliclListener(Button button, Consumer consumer)
   {
      button.setOnClickLister( new View.OnClickListener()
                                  {
                                    public void onClick(View view)
                                    {
                                       consumer.apply();
                                    }
                                  }
                                );
   }
}
```

Data binding

Expresiones. Asignación de valores - **@BindingAdapter & @BindingMethod**

2 @BindingAdapter

Un *binding adapter* puede fijar el valor de una o muchas propiedades.

El elemento **`value`** indica el nombre de todas las propiedades

El elemento **`requiredAll`** indica si es obligatorio que el *view* incluya los atributos (**`requiredAll=false`**).

Generalmente se utiliza esta opción cuando el *listener subyacente* (en el ejemplo **`TextWatcher`**) incluye muchos métodos abstractos.

```
<layout>
  <data>
    <variable name="viewModel" type="es.upsa.mimo.app.Ma
  </data>

 <androidx.constraintlayout.widget.ConstraintLayout …>

  <EditText
    android:id="@+id/etBody"
    afterTextChanged="@{(text)->viewModel.update(text)}"
    : : : />


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

es.upsa.mimo.app. AppBindingHelper

```
public class AppBindingHelper
{

  @BindingAdapter(value={"onTextChanged", "beforeTextChanged", "afterTextChanged"},
                  requiredAll=false
                  )
  public static void setListeners(EditText editText,
                                  BeforeTextChangedListener before,
                                  OnTextChangedListener on,
                                  AfterTextChangedListener after
                                 )
  {
    TextWatcher tw = new TextWatcher()
                     {
                       @Override
                       public void beforeTextChanged(CharSequence s, int s, int c, int a)
                       {
                         if (before!=null) before.beforeChange(s.toString());
                       }

                       @Override
                       public void onTextChanged(CharSequence s, int s, int b, int c)
                       {
                         if (on!=null) on.on(s.toString());
                       }

                       @Override
                       public void afterTextChanged(Editable s)
                       {
                         if (after!=null) after.afterChange(s.toString());
                       }
                     };
    editText.addTextWatcher(tw);
  }

  public interface OnTextChangedListener     { public void onChange(String text);      }
  public interface BeforeTextChangedListener { public void beforeChange(String text); }
  public interface AfterTextChangedListener  { public void afterChange(String text);  }
}
```

Data binding

Expresiones bidireccionales (*Two way*) - **@InverseBindingAdapter & @InverseBindingMethod**

Data binding actualiza la propiedad del *view* cuando la expresión incluye un LiveData.

Data binding también permite la operación inversa: actualizar el LiveData cuando el valor de la propiedad cambie (ya sea por la interacción del usuario con la *view* o por programación)

Para ello se deben utilizar expresiones **@={...}**

Se establece una relación bidireccional entre el atributo **`text`** de **`EditText`** y el **`LiveData email`** de MainViewModel: cuando **`email`** cambie, se actualiza el **`EditText`**; cuando el usuario interaccione con el **`EditText`** y modifique su valor, el cambio también quedará reflejado en el **`LiveData`**

```
<View
    android:id="@+id/widgetId"
    attributeName="@={liveData}"
    : : : />
```

```
<EditText
    android:id="@+id/etEmail"
    android:text="@={viewModel.email}"
    : : : />
```

```
public class MainViewModel extends ViewModel
{
  public LiveData<String> email;

}
```

Data binding

Expresiones bidireccionales (*Two way*) - **@InverseBindingAdapter & @InverseBindingMethod**

Es necesario definir :

1. Cómo se fija el valor de la propiedad (la funcionalidad ***setter***). Para ello se utilizará:
   - 1a **@BindingMethod**
     Cuando el *setter* pueda realizarse a través de un método que ya tenga definida la *view*
   - 1b **@BindingAdapter**
     Cuando la *view* no posea el método que realice la función del *setter* y sea necesario crearlo
2. Definir cómo se obtiene el valor de la propiedad (la funcionalidad ***getter***). Para ello se utilizará:
   - 2a **@InverseBindingMethod**
     Cuando el *getter* pueda realizarse a través de un método que ya tenga definida la *view*
   - 2b **@InverseBindingAdapter**
     Cuando la view no posea el método que realice la función del *getter* y sea necesario crearlo
3. **@BindingAdapter**
   @InverseBindingMethod y @InverseBindingAdaper crean adicionalmente otro atributo que representa un *event* a través del que notificar cambios en el valor de la propiedad bidireccional. El propósito de este *binding adapter* es fijar el valor de este nuevo atributo.

Data binding

Expresiones bidireccionales (*Two way*) - **@InverseBindingAdapter & @InverseBindingMethod**

2a @InverseBindingMethod

```
<EditText
        android:id="@+id/tvName"
        android:text="@={viewModel.email}"
        : : : />
```

Indica que la propiedad será bidireccional

```
public class MainViewModel extends ViewModel
{
  public MutableLiveData<String> email;
  : : :
}
```

```
import androidx.databinding.InverseBindingConversion;

@BindingMethods({@BindingMethod(type = EditText.class,
                                1a  attribute = "android:text",
                                    method = "setText")
                 })
@InverseBindingMethods({@InverseBindingMethod(type = EditText.class,
                                              attribute = "android:text",
                                2a            method = "getText"
                                              event = "textAttrChanged")
                 })
public class DataBindingHelpers
{
    : : :
```

El tipo de la *view*

El nombre de la propiedad

Indica el nombre del método de la *view* que hará de *getter*. Este elemento es opcional, de no indicarse se asume que el método la *view* a utilizar es el que cumple con las normas de nombrado: **getAttribute**

Define el nombre de un nuevo atributo "listener" que posteriormente se utilizará para notificar que el valor de la propiedad ha cambiado. En caso de no especificarse, se le asignaría un valor con el siguiente formato: **attributeAttrChanged**

Data binding

Expresiones bidireccionales (*Two way*) - **@InverseBindingAdapter & @InverseBindingMethod**

2b @InverseBindingAdapter

```
<EditText
        android:id="@+id/tvName"
        android:text="@={viewModel.email}"
        : : : />
```

Indica que la propiedad será bidireccional

```
public class MainViewModel extends ViewModel
{
  public MutableLiveData<String> email;
  : : :
}
```

```
import androidx.databinding.InverseBindingConversion;

@BindingMethods({@BindingMethod(type = EditText.class,
                                1a  attribute = "android:text",
                                    method = "setText")
                 })
public class DataBindingHelpers
{
2b  @InverseBindingAdapter(attribute="android:text", event="textAttrChanged")
   public static String getPropertyText(EditText editText)
   {
      return editText.getText().toString();
   }
}
```

El nombre de la propiedad

El tipo de la *view*

Define el nombre de un nuevo atributo de tipo **InverseBindingListener** que posteriormente se utilizará para notificar cambios de la propiedad bidireccional. En caso de no especificarse, se le asignaría por defecto un valor con el siguiente formato:

attributeAttrChanged

Data binding

Expresiones bidireccionales (*Two way*) - **@InverseBindingAdapter & @InverseBindingMethod**

3 @BindingAdapter

```
@BindingAdapter(value={"beforeTextChanged", "onTextChanged", "afterTextChanged", "textAttrChanged"},
                requiredAll=false
               )
public static void setListeners(EditText editText,
                                BeforeTextChangedListener before,
                                OnTextChangedListener on,
                                AfterTextChangedListener after,
                                InverseBindingListener inverseBindingListener
                               ) {
    editText.addTextChangedListener(new TextWatcher() {
                                        @Override
                                        public void beforeTextChanged(CharSequence s, int s, int c, int a) {
                                           if (before!=null) before.beforeChange(s.toString());
                                        }

                                        @Override
                                        public void onTextChanged(CharSequence s, int s, int b, int c) {
                                           if (on!=null) on.on(s.toString());
                                        }

                                        @Override
                                        public void afterTextChanged(Editable s) {
                                           if (after!=null) after.afterChange(s.toString());
                                           if (inverseListener != null) inverseBindingListener.onChange();
                                        }
                                    });

  public interface OnTextChangedListener     { public void onChange(String text);      }
  public interface BeforeTextChangedListener { public void beforeChange(String text); }
  public interface AfterTextChangedListener  { public void afterChange(String text);  }
}
```

El nombre del atributo definido en **@InverseBindingAdapter** o **@InverseBindingMethod**

Valor del atributo definido en **@InverseBindingAdapter** o **@InverseBindingMethod** que se utilizará para notificar cambios en la propiedad bidireccional

Este listener es el utilizado en **EditText** para detectar cambios en su valor

Se ejecuta este método cuando se ha modificado el valor de **EditText**

Se notifica el cambio de valor

Data binding

Expresiones bidireccionales (*Two way*) - Conversor **@InverseMethod**

Puede suceder que el origen de los datos sea de tipo **T** y la propiedad de la view lo sea de tipo **U**

```
<View
    android:id="@+id/tvName"
    data="@={ ● }"
    : : : />
```

```
package a.b.c;

public class MainViewModel extends ViewModel
{
  public MutableLiveData<T> data;
  : : :
}
```

data source

```
@BindingAdapter("data")
public static void setData(View view, U value)
{
   :::
}
```

En este caso debe realizarse **explícitamente** una conversión de tipos:

Data binding

Expresiones bidireccionales (*Two way*) - Conversor **@InverseMethod**

```
<data>
  <import type="a.b.c.Converter"/>
  <variable name="viewModel" type"a.b.c.MainViewModel"/>
</data>
: : :
<View
    android:id="@+id/tvName"
    data="@={ Converter.fromTtoU( viewModel.data ) }"
    : : : />
```

```
package a.b.c;

public class Converter
{
  @InverseMethod("fromUtoT")
  public static U fromTtoU(View view, T value)
  {
     :::
  }

  public static T fromUtoT(View view, U value)
  {
     :::
  }
```

Indica el nombre del método encargado de realizar la conversion inversa

```
package a.b.c;

public class MainViewModel extends ViewModel
{
  public MutableLiveData<T> data;
  : : :
}
```

data source

```
@BindingAdapter("data")
public static void setData(View view, U value)
{
  :::
}
```

Data binding

Expresiones bidireccionales (*Two way*) **– View que ya tienen definida la vinculación bidireccional**

View	Atributo(s)	Adaptador de vinculación (package androidx.databinding.adapters)
`AdapterView`	`android:selectedItemPosition` `android:selection`	`AdapterViewBindingAdapter`
`CalendarView`	`android:date`	`CalendarViewBindingAdapter`

`DatePicker`	`android:year`
	`android:month`
	`android:day`
		`DatePickerBindingAdapter`

`TimePicker`	`android:hour` `android:minute`	`TimePickerBindingAdapter`
`NumberPicker`	`android:value`	`NumberPickerBindingAdapter`
`CompoundButton`	`android:checked`	`CompoundButtonBindingAdapter`
`RadioButton`	`android:checkedButton`	`RadioGroupBindingAdapter`
`RatingBar`	`android:rating`	`RatingBarBindingAdapter`
`TextView`	`android:text`	`TextViewBindingAdapter`
`SeekBar`	`android:progress`	`SeekBarBindingAdapter`
`TabHost`	`android:currentTab`	`TabHostBindingAdapter`