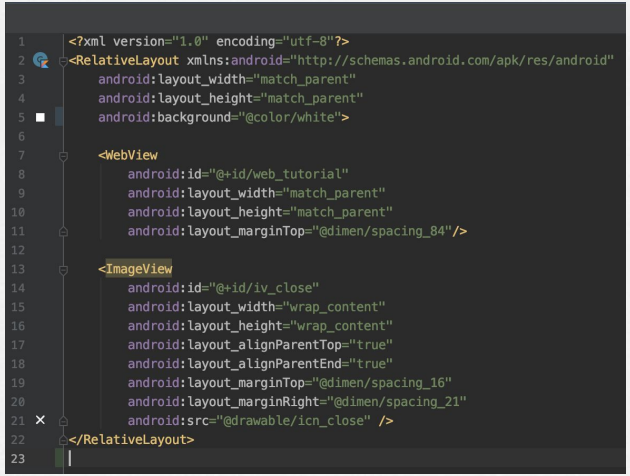
# EL PARADIGMA DECLARATIVO

# LA PROGRAMACIÓN IMPERATIVA

- Hasta ahora siempre se ha usado el paradigma imperativo:
  - Definir paso a paso la casuística de nuestra aplicación.
  - La interfaz se actualiza recorriendo el árbol de la UI:
    - *findViewById()*
    - *button.setText(text)*
    - *img.setBitmap(img)*

# LA PROGRAMACIÓN IMPERATIVA

- Árbol de widgets para inicializar la UI.
- Archivo de diseño XML.
- Widget con estado propio.
- Exponen métodos **get()** y **set()**.



```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/white">

    <WebView
        android:id="@+id/web_tutorial"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="@dimen/spacing_84"/>

    <ImageView
        android:id="@+id/iv_close"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_alignParentEnd="true"
        android:layout_marginTop="@dimen/spacing_16"
        android:layout_marginRight="@dimen/spacing_21"
        android:src="@drawable/icn_close" />
</RelativeLayout>
```

# LA PROGRAMACIÓN IMPERATIVA

- Manipular las vistas de forma manual aumenta la probabilidad de errores:
  - Es fácil olvidarse de actualizar estados de vistas.
  - Es fácil crear estados ilegales (conflicto de actualizaciones)
  - El mantenimiento de los estados de las vistas se hace complejo.

# LA PROGRAMACIÓN DECLARATIVA

- La industria está migrando a un modelo de UI declarativo:
  - La pantalla se regenera desde cero y solo se aplican los cambios necesarios.
  - Enfoque que evita la complejidad de actualizar manualmente una jerarquía de vistas.
  - Es costoso en términos computacionales: **Recomposición**.

# COMPOSICIÓN

- Funciones que reciben datos y emiten elementos de UI:

```
@Composable
fun Greeting(name: String) {
    Text("Hello $name")
}
```

# COMPOSICIÓN

- Funciones en Kotlin.
- Puedes usar **for**, **if**… aprovechando la potencia del lenguaje

```
@Composable
fun Greeting(names: List<String>) {
    for (name in names) {
        Text("Hello $name")
    }
}
```

# COMPOSICIÓN

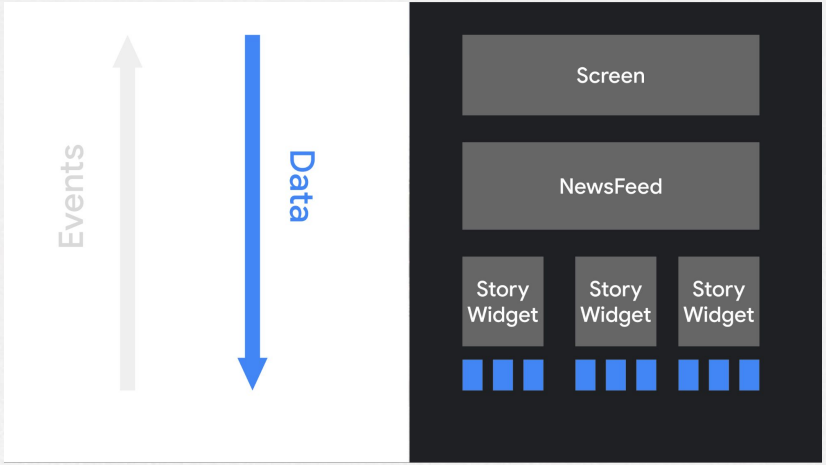
- Todas las funciones que admiten composición deben ser anotadas con **@Composable**.
- Las funciones que admiten composición pueden aceptar parámetros. **La lógica de la aplicación describe la UI**.
- **Text()** es también una función que admite composición y que se encarga de crear el elemento en la UI.

# COMPOSICIÓN

- Widgets sin estado.
- No se exponen como objetos.
- Para actualizar la UI se usa la misma función que admite composición con distintos argumentos.

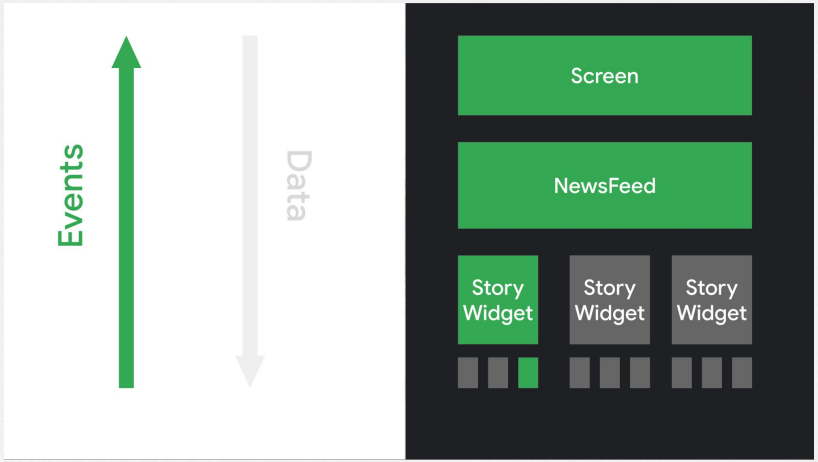
# COMPOSICIÓN

- Lógica de la app manda datos a la función que admite composición.
- Esa función utiliza otras funciones que admiten composición para describir la UI.



# COMPOSICIÓN

- El usuario interactúa con la UI.
- Se genera un evento *onClick()*.
- El estado de los datos cambia y se vuelven a llamar a las funciones que admiten composición.
- Se vuelven a dibujar los elementos de la UI -> **Recomposición**.



# RECOMPOSICIÓN

- Cuando los estados cambian, el framework vuelve a recomponer de forma inteligente solo los eventos que varían.
- Las funciones que admiten recomposición puede ejecutarse con la misma frecuencia que los fotogramas de una animación.
- Para realizar operaciones costosas es importante realizar la operación en un hilo separado del hilo de la UI.
- Siempre utiliza funciones lambda para devolver eventos.

# RECOMPOSICIÓN

```
@Composable
fun SharedPrefsToggle(
    text: String,
    value: Boolean,
    onValueChanged: (Boolean) -> Unit
) {
    Row {
        Text(text)
        Checkbox(checked = value, onCheckedChange = onValueChanged)
    }
}
```