Практическая работа. Binding. Связывание представлений в коде активности.

Под капотом findViewById()

Как вам уже известно, каждый раз, когда вы хотите взаимодействовать с представлением в коде активности или фрагмента, сначала следует получить ссылку на него вызовом `findViewById()`. Например, следующий код активности получает ссылку на кнопку `Button` с идентификатором `start_button`, чтобы кнопка могла реагировать на щелчки:

```
class MainActivity : AppCompatActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 setContentView(R.layout.activity_main)
 val startButton = findViewById<Button>(R.id.start_button)
 startButton.setOnClickListener {
 //Код, который делает что-то полезное
 }
 }
}
```

Но что на самом деле происходит при вызове `findViewById()`?

При выполнении этого кода происходит следующее:

1. Файл макета `MainActivity` (activity_main.xml) заполняется, преобразуясь в иерархию объектов `View`. Если файл описывает линейный макет, содержащий текстовое представление и кнопку, то в результате заполнения будут созданы объекты `LinearLayout`, `TextView` и `Button`. `LinearLayout` становится корневым представлением этой иерархии.

2. Android ищет в иерархии представление с подходящим идентификатором. Мы используем вызов `findViewById<Button>(R.id.start_button)`, поэтому Android ищет в иерархии объект `View` с идентификатором `start_button`.

3. Android возвращает объект `View` с искомым идентификатором, ближний к верху иерархии, и преобразует его к типу, заданному при вызове `findViewById()`. В данном случае `findViewById<Button>(R.id.start_button)` находит в иерархии первый объект `View` с идентификатором `start_button` и преобразует его к типу `Button`. Теперь `MainActivity` может взаимодействовать с объектом.

Получается, что каждый раз, когда вы вызываете `findViewById()`, Android ищет в иерархии макета объект View с подходящим идентификатором и преобразует его к заданному типу.

Хотя метод `findViewById()` удобен для получения ссылок на представления, у него есть ряд недостатков.

- Удлинение кода Чем больше представлений, с которыми вам придется взаимодействовать, тем больше вызовов придется сделать. Это может привести к удлинению кода и затруднит его чтение.
- Неэффективность Каждый раз, когда вы вызываете findViewById(), Android приходится искать в иерархии макета представление с подходящим идентификатором. Это неэффективно, особенно если макет состоит из множества представлений, образующих глубокую иерархию.
- Небезопасность в отношении null Метод findViewById() используется для поиска во время выполнения, а это означает, что компилятор не может проверить типичные ошибки. Например, методу findViewById() может передаваться недействительный идентификатор, не существующий в макете. Если вы попытаетесь выполнить код

```
val message = view.findViewById<EditText>(R.id.message)
```

и в макете нет представления View с идентификатором message, выдается исключение null-указателя и в приложении происходит фатальный сбой.

- Небезопасность в отношении типов Другая проблема заключается в том, что компилятор не может проверить, правильно ли задан тип View, что может привести к исключению приведения типа. Допустим, у вас имеется группа переключателей pizza_group и вы пытаетесь получить ссылку на нее кодом следующего вида:

```
val pizzaGroup = view.findViewById<ChipGroup>(R.id.pizza_group)
```

Хотя вместо `RadioGroup` задан тип `ChipGroup`, код все равно компилируется. Компилятор не проверяет тип на правильность при построении кода. Во время выполнения приложения будет выдано исключение приведения типа, и в приложении произойдет фатальный сбой.

На помощь приходит связывание представлений

Вместо того чтобы вызывать `findViewById()` каждый раз, когда вам понадобится ссылка на `View`, можно воспользоваться связыванием представлений. Вы создаете объект связывания (об этом чуть позднее) и используете его для обращения к представлению. Допустим, имеется макет, содержащий кнопку с идентификатором `start_button`. Если вы хотите, чтобы кнопка что-то делала по щелчку, можно получить ссылку на нее вызовом `findViewById()`:

```
val startButton = findViewById<Button>(R.id.start_button)
startButton.setOnClickListener {
 //Код, который делает что-то полезное
}
```

Со связыванием представлений вам уже не придется вызывать `findViewById()` для получения ссылки на кнопку. Вместо этого можно использовать следующий код:

```
binding.startButton.setOnClickListener {
 //Код, который делает что-то полезное
}
```

Он делает то же самое, но проще записывается, а ваш код становится короче и лучше читается.

При использовании связывания представлений Android уже не нужно искать подходящий объект View в иерархии: для обращения к нему просто используется объект связывания. Такой способ намного эффективнее findViewById().

Другое преимущество заключается в том, что компилятор предотвращает исключения null-указателей и приведения типов на стадии компиляции. При обращении к представлениям с использованием объекта связывания компилятор знает, какие представления доступны и к каким типам они относятся. Он не позволит обратиться к несуществующему представлению, и вам уже не придется приводить представление к конкретному типу, потому что компилятору этот тип уже известен. В результате ваш код становится намного безопаснее. Итак, теперь вам известны преимущества связывания представлений. Разберемся в том, как же им пользоваться.

Как использовать связывание представлений

Код связывания представлений несколько отличается для активностей и фрагментов, и в этой главе мы продемонстрируем оба варианта. Основная последовательность действий выглядит так:

1. Включение связывания представлений в код активности приложения `Stopwatch`. Мы построили приложение `Stopwatch` для изучения методов жизненного цикла Android. Вернемся к этому приложению и обновим его код активности, чтобы в нем использовалось связывание представлений.

2. Включение связывания представлений во фрагмент приложения `Bits and Pizzas`. Вернемся к приложению `Bits and Pizzas`, и посмотрим, как реализовать связывание представлений в его коде фрагмента.

Снова к приложению Stopwatch

Начнем с модификации приложения `Stopwatch`. Откройте проект этого приложения. Как вы, вероятно, помните, приложение Stopwatch выводит простой секундомер, который можно запускать, приостанавливать и сбрасывать тремя кнопками. Приложение выглядит примерно так:

В приложении используется одна активность `MainActivity` с файлом макета `activity_main.xml`. Каждый раз, когда ей потребуется взаимодействовать с одним из представлений, она вызывает `findViewById()` для получения ссылки на него. Например, чтобы кнопка `Start` реагировала на щелчки, используется код следующего вида:

```
val startButton = findViewById<Button>(R.id.start_button)
startButton.setOnClickListener {
```

```
  //Код, выполняемый по щелчку на кнопке
}
```

Давайте посмотрим, как обновить приложение, чтобы вместо `findViewById()` в нем использовалось связывание представлений.

Включение связывания представлений в файле build.gradle приложения

Чтобы использовать связывание представлений, сначала следует включить его в разделе android файла `build.gradle` приложения. Код, включающий связывание представлений, выглядит примерно так:

```
android {
 ...
 buildFeatures {
 viewBinding true
 }
}
```

Мы собираемся использовать связывание представлений в приложении `Stopwatch`, поэтому убедитесь в том, что вы включили приведенное выше изменение в файл `Stopwatch/app/build.gradle`. Затем выберите команду `Sync Now`, чтобы синхронизировать изменения с остальными частями проекта.

При включении связывания представлений генерируется код для каждого макета

Когда вы включаете связывание представлений, для каждого файла макета в приложении автоматически создается класс связывания. Например, приложение `Stopwatch` содержит файл макета с именем `activity_main.xml`, поэтому при включении связывания представлений автоматически генерируется класс связывания с именем `ActivityMainBinding`:

Каждый класс связывания включает свойство для каждого представления в макете, обладающее идентификатором. Например, макет `activity_main.xml` включает кнопку с идентификатором `start_button`, поэтому класс связывания `ActivityMainBinding` включает свойство с именем `startButton` и типом Button. Классы связывания важны, потому что представления макетов ассоциируются со свойствами класса связывания. Вместо того чтобы вызывать `findViewById()` каждый раз, когда вам понадобится ссылка на представление, вы просто взаимодействуете со свойством этого представления в классе связывания. Итак, мы включили связывание представлений в приложении Stopwatch. Давайте разберемся, как использовать его в коде `MainActivity`.

Как добавить связывание представлений в активность

Код использования связывания представлений будет практически одинаковым для всех активностей, которые вы будете создавать. Он выглядит примерно так:

```
package com.hfad.stopwatch
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import com.hfad.stopwatch.databinding.ActivityMainBinding
class MainActivity : AppCompatActivity() {
 private lateinit var binding: ActivityMainBinding
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)

 binding = ActivityMainBinding.inflate(layoutInflater)
 val view = binding.root
 setContentView(view)
 }
}
```

Приведенный выше код объявляет свойство с именем `binding` и типом `ActivityMainBinding`. Значение этого свойства задается в методе `onCreate()` активности, для чего используется следующий код:

```
binding = ActivityMainBinding.inflate(layoutInflater)
```

Эта команда вызывает метод `inflate()` объекта `ActivityMainBinding`, который создает объект `ActivityMainBinding`, связанный с макетом активности.

Код:

```
val view = binding.root
setContentView(view)
```

получает ссылку на корневое представление объекта `binding` и использует метод `setContentView()` для его отображения. После того как связывание представлений будет добавлено в активность, вы можете использовать свойство `binding` для взаимодействия с представлениями макета. Давайте посмотрим, как это делается.

Использование свойства binding для взаимодействия с представлениями

Текущий код `MainActivity` взаимодействует со своими представлениями для управления отсчетом времени и реакцией кнопок на щелчки. Мы обновим этот код, чтобы вместо вызовов `findViewById()`

он обращался к представлениям через свойство `binding` активности. Чтобы понять, как это работает, воспользуемся кнопкой `Start` активности `MainActivity` в качестве примера.

Код макета

Кнопка Start определяется в файле `activity_main.xml` следующим кодом:

```
<Button
 android:id="@+id/start_button"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:text="@string/start" />
```

Как видите, ей присвоен идентификатор `start_button`.

Код активности

`MainActivity` обеспечивает реакцию кнопки на щелчки при помощи метода `findViewById()`:

```
val startButton = findViewById<Button>(R.id.start_button)
startButton.setOnClickListener {
 //Код, выполняемый по щелчку на кнопке
}
```

При использовании связывания представлений этот код можно заменить следующим:

```
binding.startButton.setOnClickListener {
 //Код, выполняемый по щелчку на кнопке
}
```

Код делает то же самое, что и исходный код, но использует свойство `binding` объекта `MainActivity` для взаимодействия с кнопкой. Обновим полный код `MainActivity`, чтобы в нем использовалось связывание представлений.

Полный код MainActivity.kt

Ниже приведен обновленный код `MainActivity`; измените свою версию `MainActivity.kt` (изменения выделены жирным шрифтом):

```
package com.hfad.stopwatch
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.os.SystemClock
```

```
import com.hfad.stopwatch.databinding.ActivityMainBinding
class MainActivity : AppCompatActivity() {
 private lateinit var binding: ActivityMainBinding

 var running = false //Хронометр работает?
 var offset: Long = 0 //Базовое смещение
 val OFFSET_KEY = "offset"
 val RUNNING_KEY = "running"
 val BASE_KEY = "base"
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)

 binding = ActivityMainBinding.inflate(layoutInflater)
 val view = binding.root
 setContentView(view)

 //Восстановление предыдущего состояния
 if (savedInstanceState != null) {
 offset = savedInstanceState.getLong(OFFSET_KEY)
 running = savedInstanceState.getBoolean(RUNNING_KEY)
 if (running) {
 binding.stopwatch.base = savedInstanceState.getLong(BASE_KEY)
 binding.stopwatch.start()
 } else setBaseTime()
 }

//Кнопка start запускает секундомер, если он не работал

 binding.startButton.setOnClickListener {
 if (!running) {
 setBaseTime()
 binding.stopwatch.start()
 running = true
 }
 }
 //Кнопка pause останавливает секундомер, если он работал

 binding.pauseButton.setOnClickListener {
 if (running) {
 saveOffset()
 binding.stopwatch.stop()
 running = false
 }
 }
 //Кнопка reset обнуляет offset и базовое время

 binding.resetButton.setOnClickListener {
 offset = 0
 setBaseTime()
 }
 }
 override fun onPause() {
 super.onPause()
 if (running) {
```

```
    saveOffset()
    binding.stopwatch.stop()
    }
    }

    override fun onResume() {
    super.onResume()
    if (running) {
    setBaseTime()
    binding.stopwatch.start()
    offset = 0
    }
    }
    override fun onSaveInstanceState(savedInstanceState: Bundle) {
    savedInstanceState.putLong(OFFSET_KEY, offset)
    savedInstanceState.putBoolean(RUNNING_KEY, running)
    savedInstanceState.putLong(BASE_KEY, binding.stopwatch.base)
    super.onSaveInstanceState(savedInstanceState)
    }
    //Обновляет время stopwatch.base
    fun setBaseTime() {
    binding.stopwatch.base = SystemClock.elapsedRealtime() - offset
    }
    //Сохраняет offset
    fun saveOffset() {
    offset = SystemClock.elapsedRealtime() - binding.stopwatch.base
    }
}
```

И это все изменения, которые необходимо внести в приложение `Stopwatch`, чтобы перевести его на связывание представлений. Давайте в общих чертах рассмотрим, что происходит при выполнении кода, и проведем его тест-драйв.

Что делает код

При выполнении приложения происходит следующее:

1. При запуске приложения создается `MainActivity`. Активность включает свойство `ActivityMainBinding` с именем `binding`.
2. При выполнении метода `onCreate()` активности `MainActivity` объект `ActivityMainBinding` присваивается свойству `binding`. Значение `binding` присваивается в `onCreate()`, потому что именно здесь `MainActivity` впервые получает доступ к приложениям
3. Объект `ActivityMainBinding` включает свойство для каждого представления в макете с идентификатором. Например, файл макета `activity_main.xml` включает кнопку с идентификатором `start_button`, поэтому объект `ActivityMainBinding` включает свойство `Button` с именем `startButton`, открывающее доступ к этому представлению.

4. `MainActivity` использует свойство `binding` для обращения к своим представлениям. Оно указывает, как кнопка `Start` должна реагировать на щелчки, например вызовом метода `setOnClickListener()` свойства `startButton`.

Проведем тест-драйв приложения.

При запуске приложения все работает так же, как и прежде. Если щелкнуть на кнопке `Start`, секундомер запускается, а при щелчках на кнопках `Pause` и `Reset` он приостанавливается и сбрасывается.

Однако в новой версии `MainActivity` использует для взаимодействия с представлениями механизм связывания представлений (вместо вызовов `findViewById()`).