2.5. Многоэлементный макет активности

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 2.5. Многоэлементный макет активности

Напечатано:: Murad Rezvan
Дата: понедельник, 3 июня 2024, 17:46

Оглавление

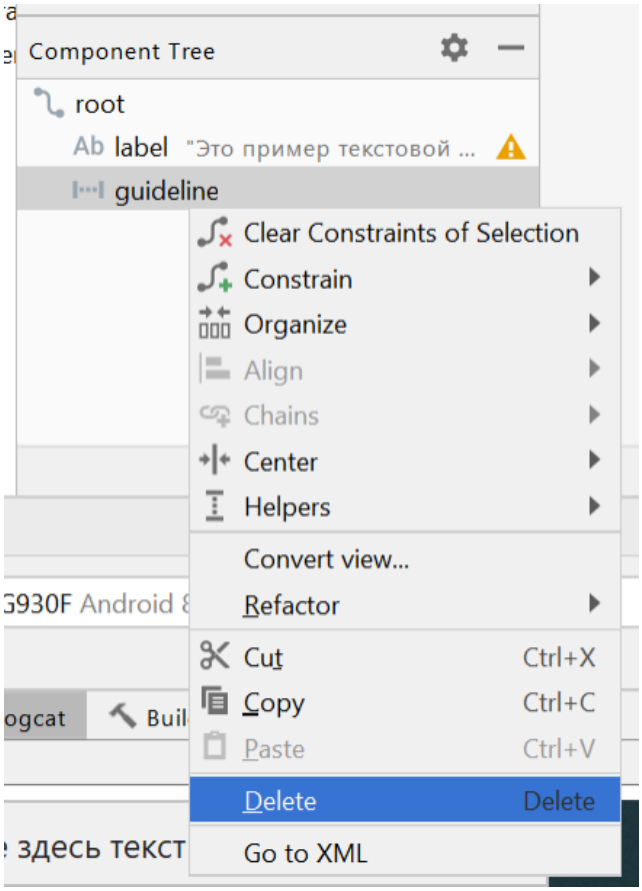
2.5.1 ConstraintLayout

Относительная разметка ConstraintLayout предусматривает создание сложных графических интерфейсов. Не смотря на то, что Android Studio использует данную разметку по умолчанию, для создания простых экранов лучше заменить её на линейную LinearLayout. Если же графический интерфейс приложения использует иерархическую вложенность экранных элементов, то целесообразно применение ConstraintLayout.

Для создания constraint-интерфейса в редакторе макета удобно использовать режим графического дизайна и дерево компонентов. Таким образом, данный режим редактора ориентирован именно на работу с относительной разметкой ConstraintLayout.

Добавление элементов на экран выполняется как и в любой разметке. Но относительная разметка требует обязательной привязки хотя бы с двух соседних сторон элемента (например, сверху и слева). Привязки задаются при помощи ограничителей (*constraints*). Описание таких ограничений в текстовом виде очень громоздко, по этому для настройки привязок чаще используется графический макет или панель атрибутов редактора макета.

По причине объёмности текстового описания разметки экрана в ConstraintLayout поиск элемента в xml-файле также значительно затруднён, не говоря уже о размещении элементов и перемещении их между контейнерами. Эти действия удобнее выполнять в дереве компонентов Component Tree. Для добавления элемента в активность достаточно выбрать нужный компонент в Palette и перенести его с помощью мыши на графический макет или в панель дерева компонентов. Так же с помощью мыши можно выбирать и перетаскивать экранные объекты по Component Tree., а в контекстном меню выбирать действия, совершаемые с выделенным элементом.



Не смотря на рекомендации Google об использовании ConstraintLayout, данная библиотека не входит в состав проекта и нуждается в подключении:

```
dependencies {
implementation 'androidx.constraintlayout:constraintlayout:$version_library_constraint'``
}
```

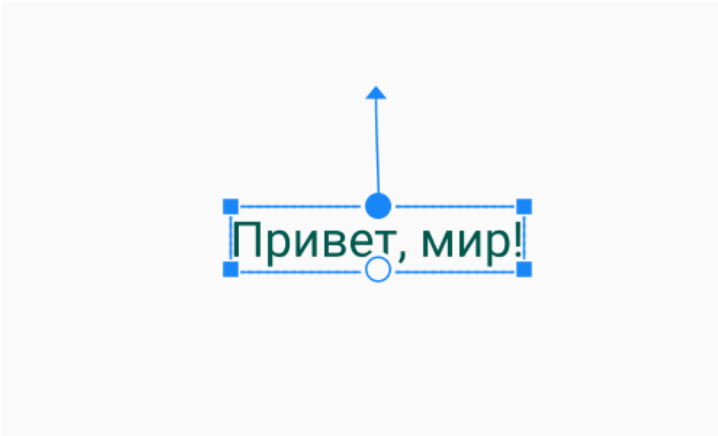
2.5.2 Настрока ограничений

Constraint определяет границу расположения и выравнивание конкретного объекта относительно второго объекта или границ родительского контейнера. Привязки бывают вертикальные (сверху или снизу) и горизонтальные (слева или справа).

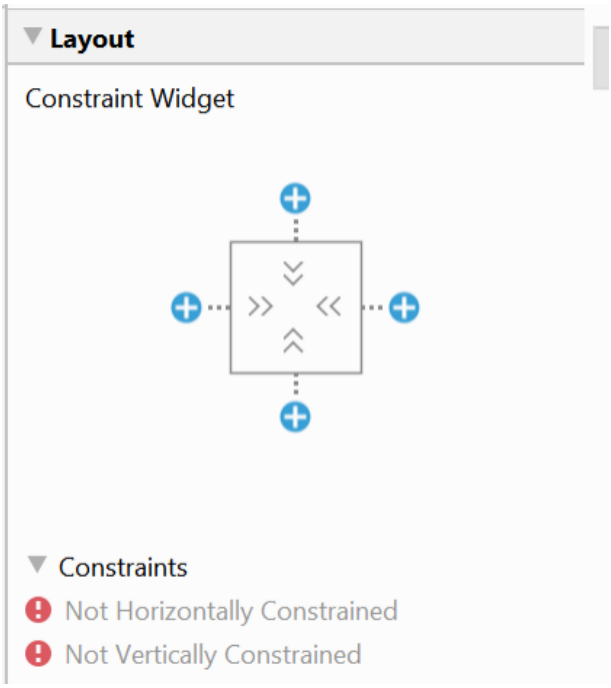
Между привязками устанавливаются ограничения.

Способы настройки ограничений

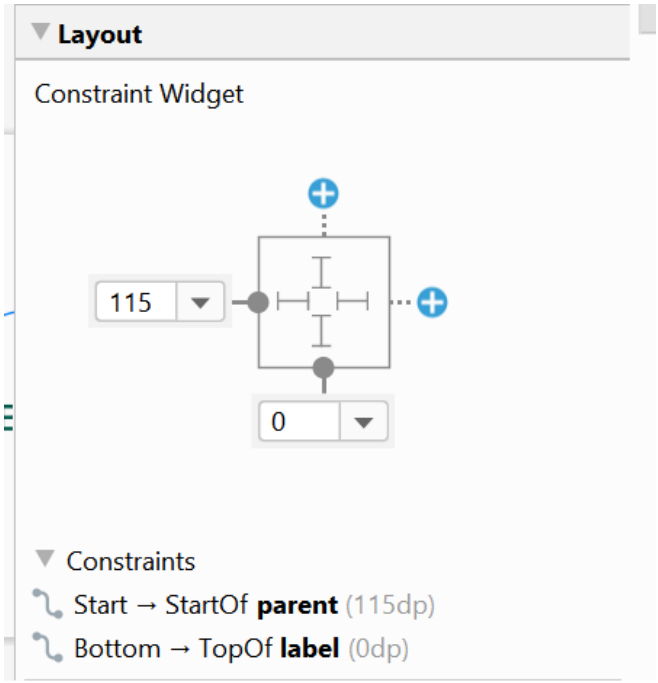
- Ограничения можно добавить *на графическом макете*, сначала выбрав привязываемый объект, а затем выделив и протянув мышью настраиваемую привязку к нужной привязке второго элемента или в любую точку желаемой границы родительского контейнера.



- Второй вариант настройки привязок возможен в панели свойств объекта (Attributes) на закладке Layout



В разделе Constraint Widget при щелчке мыши по знаку "+" синего цвета устанавливается привязка элемента к ближайшему объекту, расположенному на линии создаваемого ограничения constraint. То есть, если снизу под объектом располагался другой объект, то щелчок по нижнему "+" добавит ограничитель от верхнего элемента к нижнему. Если слева от объекта никаких объектов не располагается, то при установке левой привязки ограничитель поставится к левой границе контейнера (экрана). При этом в появившемся спиннере можно выбрать или ввести значение ограничения (в dp) для привязки. При этом в разделе *Constraints* появятся значения установленных связей.



Привязываемые объекты именуются их id, родительский контейнер именуется parent.

Из рисунка 2.5.4 видно, что у привязок есть свои имена

Привязка	Constraint
слева	Start
сверху	Top
справа	End
снизу	Bottom
по уровню текста надписи	Baseline

По этим же именам указываются ограничивающие привязки. Если элемент слева привязан к левой границе родительского контейнера, то это ограничение описывается как `Start -> StartOf parent (размер отступа)`

Отсюда видно, что для установки одного ограничения нужны два свойства: привязываемый элемент и отступ до этого элемента. Именно по этому xml -описание макета становится очень громоздким.

- Текстовое описание привязок выполняется с использованием пространства имен `app:`

```
xmlns:app="http://schemas.android.com/apk/res-auto"
```

и настройкой соответственных полей `margin`, указывающих размер ограничения.

Ограничения, представленные на рисунке 2.5.4 в xml-описании определяются так:

для левой привязки

```
android:layout_marginStart="115dp"
app:layout_constraintStart_toStartOf="parent"
```

для нижней привязки

```
app:layout_constraintBottom_toTopOf="@+id/label"
```

Поскольку снизу отступ равен 0, то и свойство `android:layout_marginBottom` у объекта отсутствует.

Все возможные ограничения между объектами описываются свойствами

```
layout_constraintLeft_toLeftOf

layout_constraintLeft_toRightOf

layout_constraintRight_toLeftOf

layout_constraintRight_toRightOf

layout_constraintTop_toTopOf

layout_constraintTop_toBottomOf

layout_constraintBottom_toTopOf

layout_constraintBottom_toBottomOf

layout_constraintBaseline_toBaselineOf

layout_constraintStart_toEndOf

layout_constraintStart_toStartOf

layout_constraintEnd_toStartOf

layout_constraintEnd_toEndOf
```

Размеры описываются по каждой привязке:

```
android:layout_marginStart

android:layout_marginEnd

android:layout_marginLeft

android:layout_marginTop

android:layout_marginRight

android:layout_marginBottom
```

- Привязки можно настроить и в коде, используя константы класса ConstraintLayout.LayoutParams

```
val lp = ConstraintLayout.LayoutParams(ConstraintLayout.LayoutParams.MATCH_CONSTRAINT_WRAP,
ConstraintLayout.LayoutParams.WRAP_CONTENT)
        txtView.layoutParams = lp
```

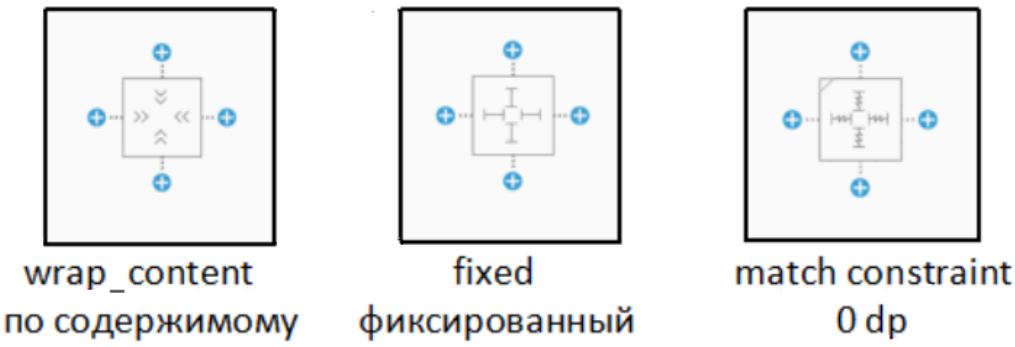
Размеры представления

В разметке ConstraintLayout к трём уже известным размерам (фиксированный, по размеру данных и по размеру родителя) добавлен четвёртый размер - *по размеру свободного пространства* `0dp (match constraint)`, часто заменяющий match_parent.

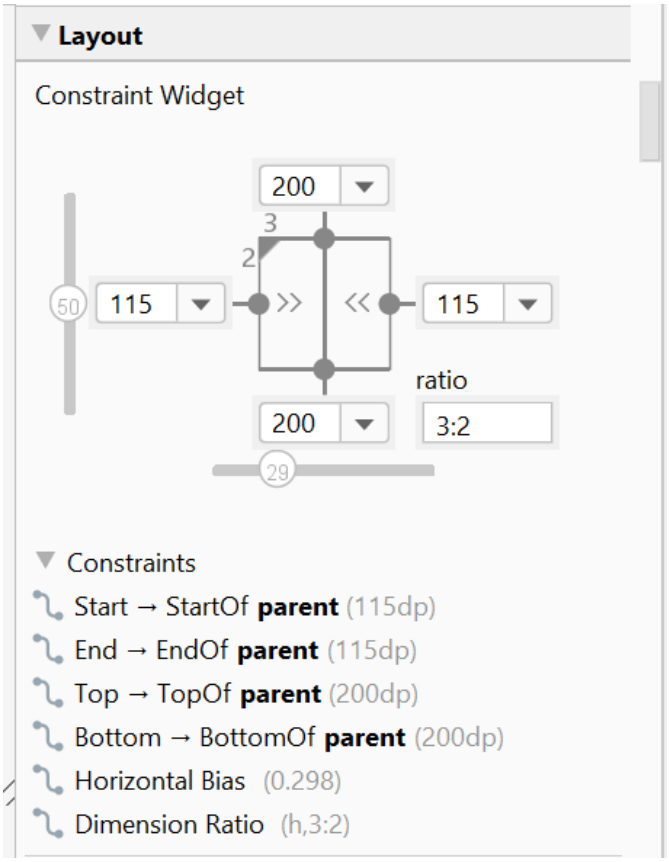
Изменять размеры кроме привычного редактирования xml-файла и раздела Declared Attributes панели свойств можно и в закладке Layout -> Constraint Widget. Каждый размер имеет своё графическое изображение:

Размер	Обозначение	Описание
wrap_content	двойные угловые скобки	размер представления соответствует размеру размещённых в нём элементов
fixed	отрезок	задаётся фиксированное значение размера представления
match constraint	зигзаг	отступы полей равны 0 dp, представление займет всё предоставленное ему пространство

Для смены размера достаточно щелчка мыши по типу размера в представлении.



У константы match constraint есть возможность настроить размеры представления так, чтобы оно занимало не просто всё свободное пространство между ограничивающими объектами, а чтобы стороны этого представления сохраняли пропорции. Для этого нужны две противоположные привязки и размер 0dp у соответственного размера представления. Если высота match constraint. нужны два вертикальных ограничителя: сверху и снизу; если же ширина match constraint. то горизонтальные ограничители: слева и справа. При этом в разделе Layout -> Constraint Widget в левом верхнем углу эскиза элемента появится пустой треугольник. Щелчок мышью по этому треугольнику сделает его закрашенным и выведет на панель поле для свойства отношения `ratio`, в котором можно установить пропорцию `ширина : высота`. На рисунке 2.5.6 размеры виджета установлены как 3:2.



Теперь при изменении ширины элемента его высота будет меняться с соблюдением установленной пропорции.

При этих настройках в xml-описании представления появился новый атрибут

```
app:layout_constraintDimensionRatio="h,3:2"
```

который оказывает, что при изменении размера высота (h) виджета будет меняться в зависимости от ширины в отношении 3:2. Если в этой строке заменить высоту на ширину

```
app:layout_constraintDimensionRatio="w,3:2"
```

то высота будет иметь фиксированный размер, а ширина со свойством 0dp будет зависеть от высоты в пропорциональном соотношении 3:2.

Выравнивания представлений

Цепочки

В ConstraintLayout имеется возможность линейной группировки объектов без использования LinearLayout. Функционалом, аналогичным возможностям линейной разметки обладает группировка представлений в цепочку (chain). Как и LinearLayout, chain имеет горизонтальную или вертикальную ориентацию, свойство весов элементов, входящих в цепочку и возможность их выравнивания относительно родительского контейнера.

Для создания цепочки нужно выбрать в дереве компонентов или на графическом макете объединяемые элементы, в контекстном меню выбрать раздел Chains, а затем выбрать ориентацию цепочки. После создания группы цепочке можно задать один из четырех стилей

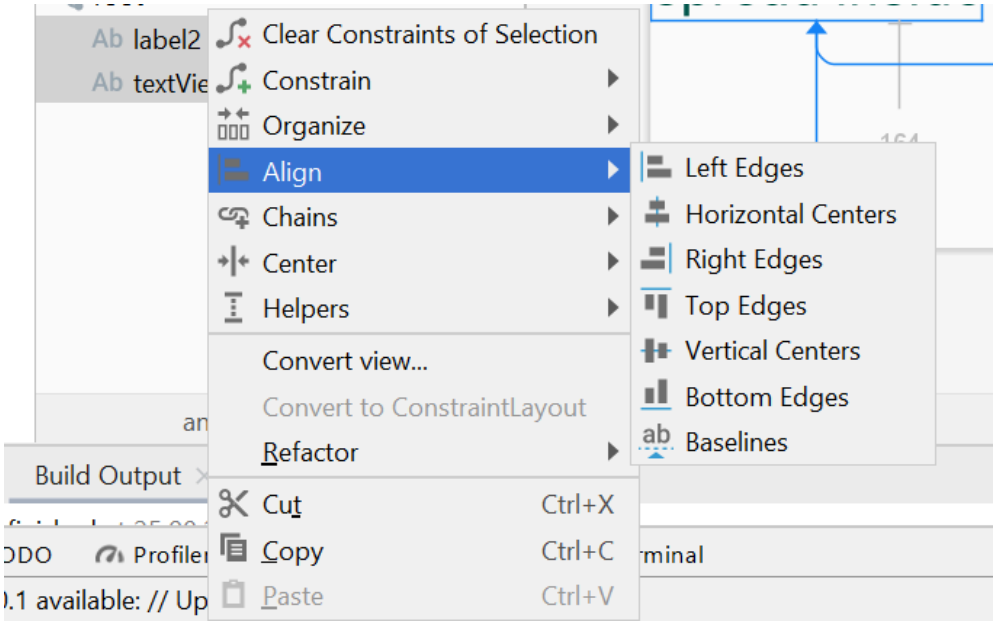
Стиль	Описание
spread	объекты внутри цепочки распределяются равномерно, сама цепочка центрируется относительно родительского контейнера
spread inside	объекты внутри цепочки распределяются равномерно, цепочка растянута по ширине родительской разметки. В этом стиле крайние представления цепочки привязаны к границам ConstraintLayout без отступов
packed	представления цепочки соединяются без отступов, сама цепочка центрируется относительно родительской разметки
weighted	у каждого виджета цепочки устанавливается вес, в соответствии с которым представление занимает пропорциональную весу область. При этом стиле хотя бы один элемент должен иметь размер match constraint

В текстовом формате стиль цепочки описывается свойством

```
app:layout_constraintHorizontal_chainStyle
```

Ещё одним преимуществом использования цепочек является возможность их пересечения. Это значит, что один и тот же элемент может входить в несколько цепочек, по этому появляется возможность фиксации большого количества объектов на экране. Нужно помнить, что элементы в цепочке могут перемещаться независимо друг от друга. Чтобы избежать наслоений представлений используют выравнивание цепочек. Для этой цели предусмотрен раздел Align, содержащий следующие возможности выравниваний в группе

Тип выравнивания	Описание
Left Edges	Левые границы представлений расположены на одно линии
Right Edges	Правые границы представлений расположены на одно линии
Top Edges	Верхние границы представлений расположены на одно линии
Bottom Edges	Нижние границы представлений расположены на одно линии
Horizontal Centers	Цепочка размещена по центру по ширине родителя
Vertical Centers	Цепочка размещена по центру по высоте родителя
Baselines	Текст в представлениях расположен на одной линии



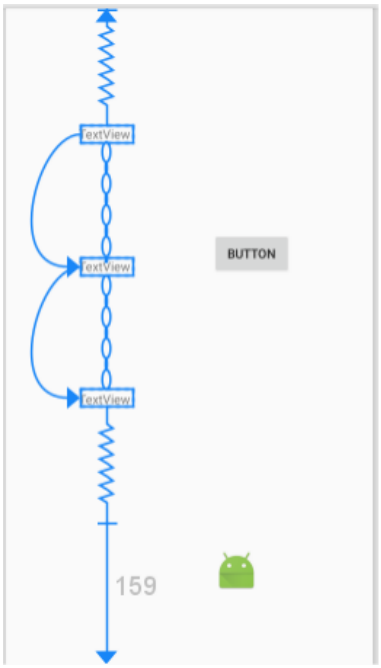
Ограничительные линии

При необходимости ограничения перемещения представления на экране используют разделители экрана: направляющие (guidelines) и барьеры (barriers). Эти объекты находятся в группе помощников (Helpers) и являются невидимыми экранными элементами. Барьер, в отличие от направляющих, способен менять своё положение на экране в зависимости от изменения размера привязанных к нему представлений. Направляющая имеет строго фиксированное положение и не меняет его ни в какой ситуации. Назначение у ограничительных линий одно: определить область отображения виджета на экране. То есть элемент ни в коем случае не может пересекать такой ограничитель.

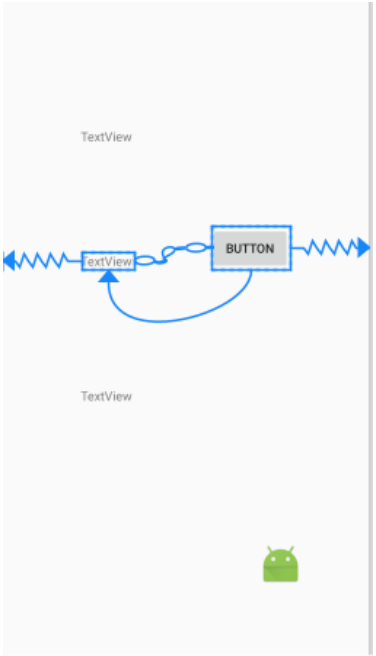
Упражнение 2.5.1 Установка привязок

1. Создайте приложение с корневой разметкой ConstraintLayout. Добавьте на экран еще два текстовых поля TextView, кнопку Button и картинку ImageView. Выполняйте задания в графическом редакторе макета.

2. Настройте выравнивание текстовых полей по левому краю и организуйте из них вертикальную цепочку.

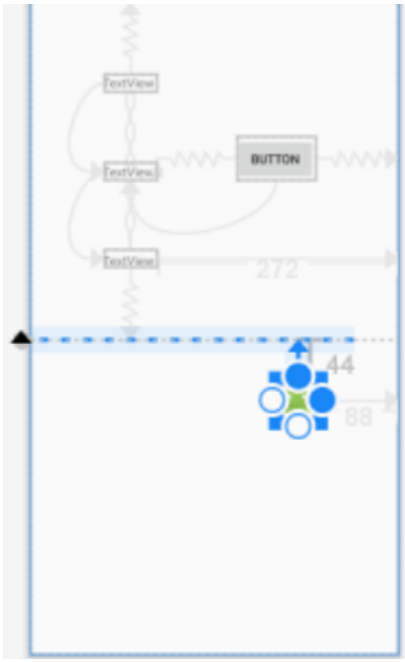


Из кнопки и одного из текстовых полей создайте горизонтальную цепочку, выровняв их предварительно по нижнему краю.



Установите недостающие привязки к границам родителей.

3. Разместите guideline так, чтобы картинка находилась всегда ниже текстовых полей и кнопки. Для этого привяжите нижнее текстовое поле снизу, imageview сверху к guideline.



Содержимое файла макета должно выглядеть примерно так:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/textView6"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="TextView"
        app:layout_constraintBottom_toTopOf="@+id/textView7"
        app:layout_constraintStart_toStartOf="@+id/textView7"
        app:layout_constraintTop_toBottomOf="@+id/textView5" />

    <TextView
        android:id="@+id/textView5"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="TextView"
        app:layout_constraintBottom_toTopOf="@+id/textView6"
        app:layout_constraintStart_toStartOf="@+id/textView6"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/textView7"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="272dp"
        android:text="TextView"
        app:layout_constraintBottom_toTopOf="@+id/guideline3"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView6" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button"
        app:layout_constraintBottom_toBottomOf="@+id/textView6"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toEndOf="@+id/textView6" />

    <ImageView
        android:id="@+id/imageView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="44dp"
        android:layout_marginEnd="88dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="@+id/guideline3"
        app:srcCompat="@android:mipmap/sym_def_app_icon" />

    <androidx.constraintlayout.widget.Guideline
        android:id="@+id/guideline3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        app:layout_constraintGuide_begin="379dp" />

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

2.5.3 Связывание данных

Разметка ConstraintLayout предусматривает большое количество экранных элементов, с которыми обязательно будет реализовываться пользовательское взаимодействие. Это значит, что в коде будет создано такое же количество переменных (под каждое активное представление) и все эти переменные необходимо будет связать с объектом экрана из xml-описания. То есть в методе onCreate() появляется малофункциональный блок вида

```
val view1 = findViewById<Type_view1>(R.id.view1)
...
val viewn = findViewById<Type_viewn>(R.id.viewn)
```

а в других местах кода множественные сеттеры значений, изменяющие текущие значения отображающих данные представлений. Понятно, что все эти "магические заклинания" очень перегружают код приложения и он становится плохо читаемым, поскольку программирование основного функционала теряется среди рутинных описаний и объявлений. К тому же такое количество объектов, находящихся в памяти с момента создания активности, снижает быстродействие приложения. При малом количестве экранных элементов это незаметно, но при сложно-структурированном макете активности такая проблема сразу же становится заметной. Конечно, можно создавать объект непосредственно перед его использованием, но при срабатывании `findViewById()` просматривается весь раздел id класса R для поиска нужной константы, то есть при постоянном обновлении значений представлений приложение будет работать ещё медленнее.

Для устранения большого количества однотипных стандартных строк кода разработана специальная библиотека связывания данных с экранными элементами. Библиотека DataBinding производит "упаковку" экрана приложения в экземпляр класса, построенного на основании xml-описания макета экрана, тем самым инкапсулируя все экранные элементы в один объект. Если поля класса сделать закрытыми (private), то в классе необходимо определить сеттеры нужных полей и обращение к их значениям будет производиться через эти методы. Если же поля не закрывать, то к ним можно будет обращаться напрямую из кода. В качестве полей создаваемого класса указываются значения только тех представлений, у которых будут изменяться свойства. Но это не ограничивает доступ через объект к остальным (не описанным в классе) представлениям. Использование данной библиотеки значительно сократит количество строк кода при разработке приложения с многоэлементным экраном и время выполнения операций в этом приложении.

Шаги использования DataBinding

1. Подключение библиотеки
2. Описание класса по макету экрана
3. Связывание данных в макете экрана
4. Создание в коде переменной связывания и её использование для доступа к экранным элементам

Подключение библиотеки

Добавление библиотеки происходит в файле сборщика текущего модуля `build.gradle(Module)` в разделе `android { ... }`.

Имеется два варианта подключения:

```
buildFeatures {
    dataBinding true
}   // рекомендован в документации Google
```

или

```
dataBinding{
        enabled = true
   } // тоже подключает библиотеку, использовалось для подключения в проект на java, с Kotlin также работает
```

Создание класса

Полями класса являются данные, которые будут изменяться в процессе работы приложения и выводиться в качестве значений экранных элементов. Все поля должны иметь начальную инициализацию, которая будет значением по умолчанию на случай, если в какое-либо поле не будет передано значение.

По разметке из трёх текстовых полей вывода и кнопке

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="88dp"
        android:text="TextView1"
        app:layout_constraintBottom_toTopOf="@+id/textView2"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_chainStyle="spread" />

    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="0dp"
        android:text="TextView2"
        app:layout_constraintBottom_toTopOf="@+id/textView3"
        app:layout_constraintStart_toStartOf="@+id/textView1"
        app:layout_constraintTop_toBottomOf="@+id/textView1" />

    <TextView
        android:id="@+id/textView3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="0dp"
        android:text="TextView3"
        app:layout_constraintBottom_toTopOf="@+id/button"
        app:layout_constraintStart_toStartOf="@+id/textView2"
        app:layout_constraintTop_toBottomOf="@+id/textView2" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="@+id/textView1"
        app:layout_constraintTop_toBottomOf="@+id/textView3" />

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

один из вариантов описания класса следующий:

```
class UsingData {
    val tfield1 = ""
    val tfield2 = ""
    val nfield1 = 0
}
```

На самом деле полей в классе может быть и больше, так как в текстовые представления можно в качестве данных передавать вычислимые выражения: конкатенацию строк, арифметические выражения, логические выражения и т.п.

Поскольку кнопка выступает управляющим элементом и не предназначается для вывода данных на экран, то её в поля класса добавлять не нужно.

Можно в классе определить конструктор, чтобы при связывании макета с данными была возможность сразу передать заполняемые значения.

```
    constructor(tf1:String, tf2:String, nf1:Int) {
        tfield1 = tf1
        tfield2 = tf2
        nfield1 = nf1
    }
```

Изменение макета

Когда класс готов, можно перестраивать макет экрана под структуру класса. Для этого нужно:

1. Сделать корневым элементом объект <layout> и перенести в него все пространства имён;
2. Добавить в корневой элемент объект <data> и определить в нём переменную связываемого класса;
3. Установить выражения-связывания для подстановки в свойство `android:text` из полей объявленной в <data> переменной.

Выполнить все эти действия можно вручную, но Android Studio и здесь помогает в конвертации.

- Для автоматического перевода файла макета экрана в формат data-binding нужно у корневой разметки вызвать контекстное меню и выбрать последовательность пунктов `Show Context Actions (Alt+Enter) -> Convert to data binding layout`.

В формате связывания данных файл разметки выглядит так:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <data>

    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity">

        <TextView
            android:id="@+id/textView1"
            ... />
        <TextView
            android:id="@+id/textView2"
           ... />
        <TextView
            android:id="@+id/textView3"
            ... />
        <Button
            android:id="@+id/button"
           ... />

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

- Объект <data> содержит описание переменных и необходимых импортов пакетов. Каждая переменная и каждый импорт представляются отдельным объектом контейнера <data>.

Переменная служит для связывания макета с данными, по этому имеет тип построенного ранее класса. Для указания типа нужно прописать полный путь к этому классу.

```
<variable
    name="mydata"
    type="ru.samsung.itacademy.mdev.databindingsample.UsingData" />
```

Можно создать переменные других типов:

```
<variable
    name="text"
    type="String" />

<variable
    name="number"
    type="Double" />
```

Импорт просто содержит имя подключаемого пакета

```
<import type="androidx.fragment.app.Fragment"/>
```

- Поскольку объявленная переменная является экземпляром построенного ранее класса, то она имеет все поля этого класса. Значит, значения этих полей можно установить в качестве надписи в текстовые представления. Для этого используется xml - описание подстановки `@{подстановочное выражение}`.

  При наличии в подстановке двойных кавычек ", значение выражения заключается в апострофы ':

  "@{обращение к значению текстового поля переменной}"

  '@{обращение к значению текстового поля переменной + "?!"}'

Подстановочным выражением являются:

Выражение	Пример	Примечание
обращение к текстовому полю переменной	"@{mydata.tfield1}"	текстовое значение поля tfield1 объекта mydata
обращение к числовому полю (поддерживается приведение типа)	"@{String.valueOf(mydata.nfield1)}"	приведение типа к строковому
вычислимое выражение	'@{mydata.tfield2 + ", " + mydata.tfield1.charAt(0) + "."}'	
тернарная операция	'@{mydata.nfield1 == 0? "ddd": "fds" }'	
неполная тернарная операция	"@{mydata.tfield1 ?? String.valueOf(345)}"	значение выводится, если в условии не null
ссылка на данные из другого поля	"@{textView1.text}"	значение совпадает с текстом из поля с id = textView1

Полный список операций, разрешенных и запрещённых в подстановочных выражениях, приведён в документации по языку выражений

Связывание макета с данными

Теперь в методе `onCreate()` нужно заменить вызов `setContentView()` на объявление и инициализацию объекта связывания, тип данных которого будет иметь имя, составленное из имени xml-файла макета с приписанным окончанием *Binding*. Если xml-описание экрана размещается в файле **sample.xml**, то класс связывания будет именоваться **SampleBinding**.

То есть при создании data binding layout файла для него генерируется класс `ИмяXMLфайлаBinding`, на основании которого создаётся java-класс `ИмяXMLфайлаBindingImpl`, содержащий определения всех методов, применимых к объекту связывания.

Само связывание создаётся методом `setContentView()` вспомогательного класса DataBindingUtil

```
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
val bind: ActivityMainBinding = DataBindingUtil.setContentView(this,R.layout.activity_main)
    }
```

Теперь все экранные элементы активности являются полями объекта `bind`. Это значит, что обращенит к кнопке с id = "button" будет происходить через объект bind

```
bind.button.setOnClickListener({ ... })
```

Кроме этого, у объекта `bind` есть поле - объект класса, созданного для связывания макета с данными, позволяющее создавать объекты этого класса.

```
bind.mydata = UsingData("text1", "text2" , Random.nextInt(-15, 15))
```

Следующий пример при нажатии кнопки перечисляет месяцы первой половины года с их порядковыми номерами

без использования библиотеки DataBinding

```
class MainActivity : AppCompatActivity() {
    val tf1: Array<String> = arrayOf("jan", "feb", "mar","apr","may","jun")
    val tf2: Array<String> = arrayOf("январь", "февраль", "март","апрель","май","июнь")
    var index = 0
        override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
       setContentView(R.layout.activity_main)
       val text1 = findViewById<TextView>(R.id.textView1)
       val text2 = findViewById<TextView>(R.id.textView2)
       val text3 = findViewById<TextView>(R.id.textView3)
       val button = findViewById<Button>(R.id.button)
            button.setOnClickListener({text2.text = tf1[index%6]
                                                    text3.text = tf2[index%6]
                                                    text1.text = ((index++)%6+1).toString()
                                    })
        }
}
```

с использованием библиотеки

```
class MainActivity : AppCompatActivity() {
    val tf1: Array<String> = arrayOf("jan", "feb", "mar","apr","may","jun")
    val tf2: Array<String> = arrayOf("январь", "февраль", "март","апрель","май","июнь")
    var index = 0
        override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val bind: ActivityMainBinding = DataBindingUtil.setContentView(this,R.layout.activity_main)
       bind.button.setOnClickListener({ bind.mydata = UsingData(tf1[index%6],tf2[index%6],(index++)%6+1)})
    }
}
```

Пример наглядно демонстрирует двукратное сокращение кода при небольшом количестве экранных элементов. Когда же речь идёт о сложных графических интерфейсах, применение библиотеки DataBinding оказывается более эффективным.

Упражнение 2.5.2 Применение библиотеки DataBinding

1. Используя линейную разметку нарисуйте экран приложения с двумя текстовыми полями TextView и кнопкой Button.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/name"
        android:text="Название клуба"/>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/rating"
        android:text="Рейтинг клуба"/>
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/view"
        android:text="Следующий клуб"/>
</LinearLayout>
```

2. В ресурсах приложения опишите строковый массив с названиями спортивных клубов Российской Федерации.

```
<string-array name="clubs">
        <item>Динамо</item>
        <item>Зенит</item>
        <item>Ротор</item>
        <item>Спартак</item>
    </string-array>
```

3. В классе MainActivity прочитайте ресурс в переменную типа Array.

```
var spClub: Array<String> = resources.getStringArray(R.array.clubs)
```

4. Запрограммируйте кнопку так, чтобы при нажатии на неё в первое текстовое поле выводились по очереди названия клубов, а во втором поле появлялась надпись: "Рейтинг клуба: N", где N - случайное число от 0 до 10.

```
 val tvname = findViewById<TextView>(R.id.name)
        val tvrating = findViewById<TextView>(R.id.rating)
        val click = findViewById<Button>(R.id.click)
        click.setOnClickListener {
            tvname.setText(spClub[(i++)%spClub.size])
            tvrating.setText(((Math.random()*10).roundToInt()).toString())
        }
```

5. Подключите к проекту библиотеку DataBinding.

```
    buildFeatures {
        dataBinding true
    }
```

6. Измените проект на использование подключенной библиотеки.

В первую очередь избавимся от `setContentView()` и всех `findViewById()`. То есть удалим весь код, выполненный в п.4.

6.1. Создадим класс данных с двумя полями: текстовое поле для названия клуба *name* и целочисленное поле *rating* для рейтинга и конструктором.

```
class Clubs{
        var name = ""
        var rating: Int = 0

        constructor(name: String, rating: Int) {
            this.name = name
            this.rating = rating
        }
    }
```

6.2. Изменим файл макета, добавив в него раздел `data` и заменив данные в текстовых полях на подстановочные выражения. Поскольку поле rating числовое, то его необходимо приводить в текст: `android:text="@{String.valueOf(club.rating)}"`

6.3. Создаём переменную связывания данных кода с макетом

```
val dbind: ActivityMainBinding = DataBindingUtil.setContentView(this,R.layout.activity_main)
```

6.4. Посредством созданной переменной обращаемся к экранным элементам:

```
dbind.click.setOnClickListener {
            dbind.club = Clubs(spClub[(i++)%spClub.size], (Math.random()*10).roundToInt())
        }
```

Теперь класс MainActivity.kt будет выглядеть так:

```
class MainActivity : AppCompatActivity() {
    var  i = 0
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
       var spClub: Array<String> = resources.getStringArray(R.array.clubs)
        val dbind: ActivityMainBinding = DataBindingUtil.setContentView(this,R.layout.activity_main)
       dbind.click.setOnClickListener {
            dbind.club = Clubs(spClub[(i++)%spClub.size], (Math.random()*10).roundToInt())
        }
    }
}
```

Начать тур для пользователя на этой странице