Ввод и вывод текста. Кнопка. Слушатели

При разработке программного обеспечения с пользовательским интерфейсом (*англ.* User Interface, UI, это сокращение часто используется и полезно его знать) есть два основных подхода к тому где именно формируется пользовательский интерфейс.

Первый подход описывает пользовательский интерфейс прямо в коде программы. Это имеет свои плюсы и минусы: код становится перегруженным и тяжелее воспринимается, но зато можно более гибко формировать свойства элементов, используя условные операции, циклы и т.д. В Android такой подход реализуется с помощью Jetpack Compose. В данном курсе он не затрагивается.

Второй подход разделяет интерфейс и программный код. В Android для описания интерфейса используются XML-файлы, в которых элементы управления (кнопки, надписи, списки и т.д.) описаны в виде тегов, говорящих что это за элемент, и атрибутов, которые описывают его свойства. Например, кнопка может быть описана следующим образом:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Нажми меня!"/>
```

Здесь элемент управления называется `Button` (*англ.* кнопка). Начинается тег с открывающей угловой скобки, а завершается закрывающей угловой. Если тег одиночный, то есть внутрь него не вложены другие элементы, то перед закрывающим тегом ставится прямой слеш:

```
<Button ... />
```

Если тег имеет вложенные элементы (например, в контейнер могут быть вложены другие элементы управления, то слеш не ставится, а в после вложенных элементов повторяется название тега со слешем перед ним:

```
<LinearLayout>
    <Button … />
    <TextView … />
</LinearLayout>
```

Атрибуты внутри тега описывают свойства элемента: как он выглядит, как с ним можно взаимодействовать и т.д. Многие свойства являются характерными только для определённых элементов, например, свойство `textSize` указывает размер шрифта, оно может использоваться в поле ввода текста `EditText`, но оно не имеет смысла для изображения `ImageView`.

Но некоторые свойства являются общими для любых элементов. Например, свойство `id` определяет уникальный идентификатор элемента, с помощью этого идентификатора можно ссылаться на элемент как в разметке (например, указать что кнопка должна располагаться под полем ввода с указанным идентификатором), так и в коде (чтобы получить ссылку на элемент и читать или менять его свойства). Примеры такого взаимодействия будут описаны далее.

Android Studio подсказывает разработчику если что-то сделано не совсем правильно, или даже совсем неправильно. Например, если тег подчёркивается красной волнистой линией, значит, отсутствуют какие-то обязательные атрибуты:

```
<Button />
```

Обычно можно установить текстовый курсор на такой проблемный тег и нажать Alt+Enter (или в контекстном меню выбрать пункт «Show Context Actions») и в появившемся контекстном меню будут пункты вроде «Insert required attribute 'android:layout_height'», которые позволяют быстро добавить недостающие атрибуты, или хотя бы выяснить чего именно не хватает.

А если у атрибута меняется фон, это означает что технически всё допустимо, но нарушены какие-то важные рекомендации для разработчиков:

```
android:text="Нажми меня!"
```

В данном случае свойство `text` задано жёстко закодированным текстом, а так делать не рекомендуется, ведь программа может быть локализована на много языков. Все текстовые строки должны быть разнесены по разным языковым ресурсным файлам, более подробно такой подход разбирается в одной из следующих тем курса.

Чтобы обратиться к элементу из кода, нужно присвоить ему идентификатор. Он представляет собой текстовую строку в таком формате:

```
<TextView
    android:id="@+id/myTextView"
    .../>
```

Префикс `@+id/` означает что создаётся новый идентификатор, а не идёт отсылка к уже существующему. При компиляции все идентификаторы попадают в специальный класс `R`, который заново создаётся при каждой сборке программы. Получить ссылку на объект можно с помощью метода `findViewById`:

```
val myTextView = findViewById<TextView>(R.id.myTextView)
```

Нужно чётко различать идентификатор `R.id.myTextView` и сам объект `myTextView`, который возвращает метод `findViewById`. Идентификатор представляет собой лишь число для поиска объекта, а объект – это настоящий экземпляр класса, к которому можно обращаться для изменения свойства элемента управления, например, изменить текст на экране:

```
val myTextView = findViewById<TextView>(R.id.myTextView)
myTextView.text = "Новый текст"
```

Контейнер LinearLayout

Если просто добавить множество элементов в разметку, то все элементы будут идти один за другим, без каких-либо отступов и логики, и рано или поздно уйдут на край экрана. Чтобы размещать элементы по задуманной логике, используются контейнеры – специальные элементы, которые сами на экране не видны, но расставляют элементы по своим местам. Android предоставляет много контейнеров, каждый из них имеет свои особенности компоновки элементов, например:

- `LinearLayout` – размещает элементы один из другим по горизонтали или вертикали.
- `RelativeLayout` – размещает элементы относительно друг друга или родительского элемента («под текстовым полем с id=…», «выровнять правый край по правому краю элемента с id=…» и т.д.)
- `ConstraintLayout` – «наследник» `RelativeLayout` с множеством новых возможностей, но менее толерантный к ошибкам
- `FrameLayout` – размещает элементы один над другим, бывает полезен если некоторые элементы должны «плавать» над другими
- `GridLayout` – размещает элементы в виде таблицы

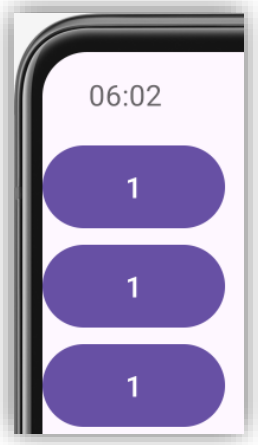
Многие из этих контейнеров будут рассмотрены в дальнейших темах курса, а для начала будет разобран контейнер `LinearLayout`.

Как уже было сказано выше, `LinearLayout` размещает элементы либо по горизонтали, либо по вертикали. Определяет направление размещения свойство `orientation`. Например, если требуется горизонтальное размещение, то можно использовать такой код:

```
<LinearLayout
    android:orientation="horizontal"
    android:layout_height="wrap_content"
    android:layout_width="match_parent">
    <Button
        android:text="1"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"/>
    <Button
        android:text="1"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"/>
    <Button
        android:text="1"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"/>
</LinearLayout>
```

Результат будет выглядеть примерно так:

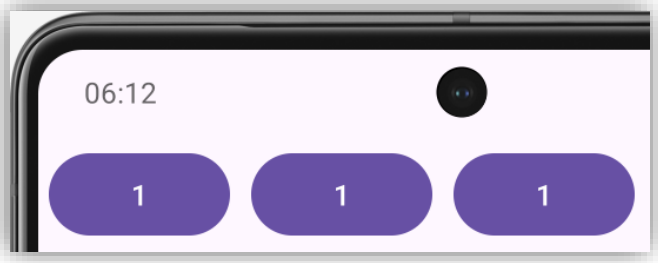
Кнопки размещаются одна за другой по горизонтали, поскольку свойство `orientation` имеет значение `horizontal`. Если требуется разместить элементы по вертикали, то свойству следует присвоить значение `vertical`:



На скриншотах видно, что элементы управления «прилипают» друг к другу или к краю экрана – по умолчанию у них отсутствует отступ от соседних элементов. Чтобы добавить такой отступ, нужно каждому элементу задать свойство `layout_margin` – на какое расстояние нужно отступить от соседнего элемента. Размеры и расстояния в Android задаются условной единицей `dp`, которая является одинаковой на разных устройствах независимо от плотности точек на экране:

```
<LinearLayout
    android:orientation="horizontal"
    android:layout_height="wrap_content"
    android:layout_width="match_parent">
    <Button
        android:text="1"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:layout_margin="5dp"/>
    <Button
        android:text="1"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:layout_margin="5dp"/>
    <Button
        android:text="1"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:layout_margin="5dp"/>
</LinearLayout>
```

Результат будет примерно таким:



Теперь элементы находятся на некотором расстоянии друг от друга, это выглядит приятнее. Можно манипулировать отступами с помощью свойства margin, а также его разновидностями `layout_marginTop`, `layout_marginBottom`, `layout_marginStart` и `layout_marginEnd`, это позволит сделать интерфейс более стильным.

Контейнеры можно вкладывать друг в друга (для экономии места здесь и далее код может быть сокращен, оставлены только важные для понимания атрибуты):

```
<LinearLayout android:orientation="vertical">
    <LinearLayout android:orientation="horizontal">
        <Button android:text="1"/>
        <Button android:text="2"/>
        <Button android:text="3"/>
    </LinearLayout>
    <LinearLayout android:orientation="horizontal">
        <Button android:text="4"/>
        <Button android:text="5"/>
        <Button android:text="6"/>
    </LinearLayout>
    <LinearLayout android:orientation="horizontal">
        <Button android:text="7"/>
        <Button android:text="8"/>
        <Button android:text="9"/>
    </LinearLayout>
</LinearLayout>
```

Результат будет примерно таким:

Вложенные контейнеры позволяют довольно гибко формировать интерфейс, но увлекаться этим не стоит: чем больше вложенных контейнеров, тем больше времени система тратит на формирование интерфейса, это может приводить к снижению производительности программы.

Важно разобраться с атрибутами `layout_width` и `layout_height`. Они определяют размеры элемента. Однако программа может работать на самых разных устройствах, и размеры у экранов могут очень сильно отличаться – одно дело смартфон, другое – смарт-часы, и

третье – умный телевизор. Поэтому размеры очень редко задаются в конкретных числовых величинах, чаще используются специальные константы:

- `wrap_content` – размер будет подобран таким образом, чтобы вместить содержимое элемента;
- `match_parent` – элемент постарается занять всё доступное пространство экрана или контейнера;
- `0dp` – специальная константа, которая означает не нулевой размер элемента, а специальный размер, зависящий от используемого контейнера. Чаще всего `0dp` означает что размером элемента должен управлять не сам элемент, а контейнер.

В примерах выше контейнер `LinearLayout` по ширине (`layout_width`) занимает всё доступное пространство (`match_parent`), а по высоте (`layout_height`) лишь столько места, сколько нужно чтобы влезли все его дочерние элементы. Кнопки же и по ширине, и по высоте занимают лишь минимально требуемое место (`wrap_content`).

Атрибуты `layout_width` и `layout_height` в большинстве случаев являются строго обязательными, без них программа компилироваться не будет.

Контейнер `LinearLayout` умеет распределят ширину или высоту согласно указанным *весам* элементов. Как раз здесь и пригодится особая константа `0dp`. В следующем примере ширина каждой кнопки установлена в `0dp`, а элемент `layout_weight` определяет вес элемента:

```
<LinearLayout
    android:orientation="horizontal"
    android:layout_height="wrap_content"
    android:layout_width="match_parent">
    <Button
        android:text="1"
        android:layout_height="wrap_content"
        android:layout_width="0dp"
        android:layout_weight="1"/>
    <Button
        android:text="2"
        android:layout_height="wrap_content"
        android:layout_width="0dp"
        android:layout_weight="2"/>
    <Button
        android:text="3"
        android:layout_height="wrap_content"
        android:layout_width="0dp"
        android:layout_weight="3"/>
</LinearLayout>
```

Вес первой кнопки 1, второй 2, третьей 3. Система просуммирует веса всех элементов – получится 6, и выделит место каждому элементу: 1/6, 2/6 и 3/6 и общей ширины экрана:

Отступы у элементов убраны чтобы результат был более наглядным.

Надпись TextView

Элемент `TextView` представляет собой текстовую надпись. Его основным атрибутом является text, содержимое этого атрибута появится на экране:

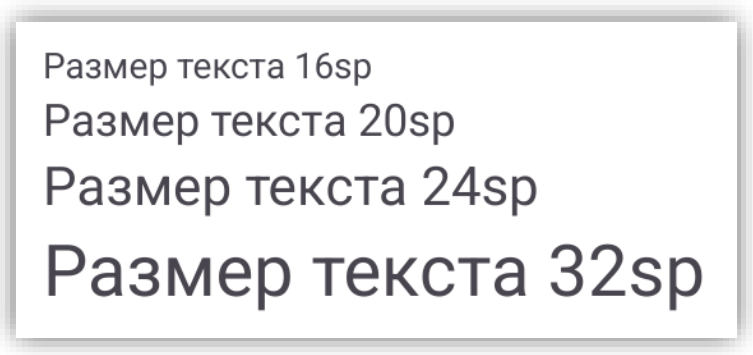
```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Привет, мир!"/>
```

Результатом этого кода будет вот такая надпись:

Никаких отступов также не предусмотрено, отступы нужно добавлять с помощью семейства атрибутов `layout_margin`.

Размер текста задаётся с помощью свойства `textSize`. Однако в отличие от остальных размеров и отступов, которые задаются в единицах `dp`, размер текста задаётся в единицах `sp`. Это позволяет системе по отдельности масштабировать элементы управления и текст: например, пользователи со слабым зрением могут сделать текст покрупнее, а все остальные элементы оставить прежнего размера.

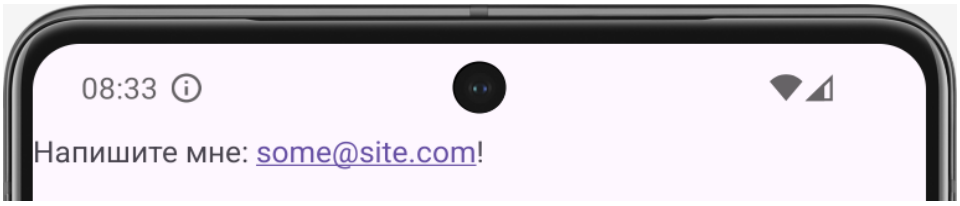
```
<TextView android:text="Размер текста 16sp" android:textSize="16sp"/>
<TextView android:text="Размер текста 20sp" android:textSize="20sp"/>
<TextView android:text="Размер текста 24sp" android:textSize="24sp"/>
<TextView android:text="Размер текста 32sp" android:textSize="32sp"/>
```



У `TextView` есть полезное свойство `autoLink`, оно говорит системе какие элементы нужно подсветить в текстовой надписи: значение `web` будет подсвечивать адреса сайтов, `email` – электронную почту, `phone` – номера телефонов, а `all` подсветит всё найденное:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Напишите мне: some@site.com!"
    android:autoLink="all"/>
```

На экране появится примерно такая надпись:

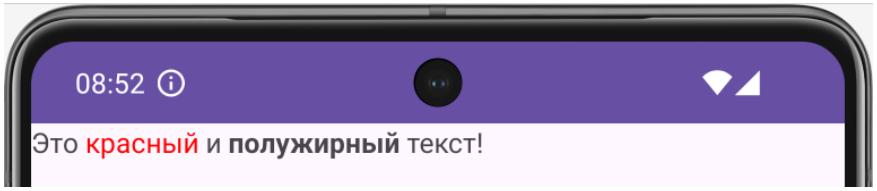


Если нажать на адрес электронной почты, будет запущена системная программа для работы с электронной почтой (если она установлена на устройстве, конечно).

Обычную строку типа `String` можно заменить на объект класса `Spannable` – такие объекты содержат строку с форматированием. Например, часть текста можно сделать красной, а другую – полужирной:

```
val s = SpannableStringBuilder("Это красный и полужирный текст!")
s.setSpan(
    ForegroundColorSpan(Color.RED),
    4, 11,
    Spannable.SPAN_EXCLUSIVE_INCLUSIVE)
s.setSpan(
    StyleSpan(Typeface.BOLD),
    14, 24,
    Spannable.SPAN_EXCLUSIVE_INCLUSIVE)
tvText.text = s
```

Здесь создаётся объект класса `SpannableStringBuilder`, и в него заносится текст, а затем с помощью метода `setSpan` последовательно применяется форматирование. Первый параметр метода – тип форматирование, в данном примере используется `ForegroundColorSpan` для изменения цвета фрагмента, и `StyleSpan` для полужирного текста. Полный список типов можно посмотреть в документации. Второй и третий параметр – с какого по какой символ (не включительно) нужно применить формат. Последний параметр – флаг, показывающий нужно ли вставлять или удалять текст, в данном случае он не используется. Созданный объект присваивается свойству `text` элемента `TextView`, результат будет примерно таком:



Поле ввода EditText

Элемент `EditText` используется для ввода текста пользователем. Выглядит этот элемент в разметке следующим образом:

```
<EditText
    android:id="@+id/myEditText"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inputType="text"/>
```

Для полей ввода чаще всего требуется указывать идентификатор `id`, ведь информацию из поля ввода нужно получать и использовать в программном коде. Для доступа к содержимому поля ввода служит поле `text`:

```
val myEditText = findViewById<EditText>(R.id.myEditText)
val s = myEditText.text
```

Однако тут возникает неожиданное осложнение: переменная `s` оказывается не строкового типа `String`, а совсем другого типа `Editable`! Это одно из последствий того, что внутри Android предоставлет программисту Java-интерфейсы, а компилятор автоматически преобразует их в Kotlin-интерфейсы, и иногда результат не совсем такой как ожидалось. Поэтому правильнее будет преобразовать содержимое свойства `text` в строку:

```
val myEditText = findViewById<EditText>(R.id.myEditText)
val s = myEditText.text.toString()
```

Содержимое поля ввода можно программно изменить. Кажется, было бы логично использовать то же свойство `text`, с ним возникает та же проблема: оно другого типа. Поэтому правильно будет вызвать функцию `setText`:

```
myEditText.setText("Новый текст")
```

Полезным свойством элемента `EditText` является `inputType`, оно описывает какие данные ожидаются в поле ввода, и не даёт вводить недопустимые символы. Вот наиболее часто используемые значения этого свойства:

- `text`, `textMultiLine` – обычный однострочный или многострочный текст
- `textCapCharacters`, `textCapWords`, `textCapSentences` – текст, система автоматически будет делать заглавной каждую букву, или первую букву каждого слова, или первую букву каждого предложения
- `date`, `datetime`, `time` – дата, или дата со временем, или только время
- `number`, `numberDecimal`, `numberSigned` – целое число, или число с дробной частью, или число со знаком
- `textPassword` – пароль, символы будут закрыты чёрными кружками
- `textUri` – гиперссылка

Опции можно комбинировать: например, если указать `numberDecimal`, то можно будет вводить число с десятичной частью, но только положительное; если указать `numberSigned`, то можно вводить отрицательное число, но только целое; а вот если указать комбинацию

`numberDecimal|numberSigned`, то становится возможным ввести и отрицательное дробное число тоже.

Свойство `inputType` полезно ещё и тем, что клавиатура для ввода будет показываться наиболее подходящая к указанному значению, например, для числовых полей это будет цифровая клавиатура.

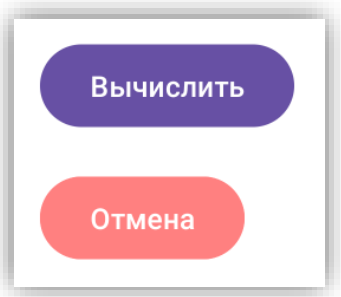
Кнопка Button

Элемент `Button` представляет собой кнопку, которую пользователь может нажать:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Вычислить"/>
```

При желании можно изменить цвет кнопки, для этого в Android предназначено свойство `backgroundTint`. Цвет можно задавать как непосредственно в HEX-формате, так и с помощью ресурсов (работа с ресурсами рассматривается в дальнейших темах курса):

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Вычислить"
    android:layout_margin="8dp"/>

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Отмена"
    android:backgroundTint="#ff8080"
    android:layout_margin="8dp"/>
```



Но самое главное в кнопке – это, конечно, возможность её нажать и выполнить при этом какое-то действие. Для этого используются слушатели.

Слушатели

Слушатель (*англ.* Listener) в Android – это функция, которая вызывается элементом управления (или каким-либо другим объектом) при возникновении события: например, для кнопки это чаще всего нажатие, для поля ввода – ввод очередного символа, и т. д.

Некоторые слушатели можно установить прямо в интерфейсе. Например, для кнопки это атрибут `onClick`:

```
<Button
    ...
    android:onClick="buttonCalcClick"/>
```

При нажатии этой кнопки будет вызван метод `buttonCalcClick`, который обязательно должен быть расположен в том классе, который соответствует данной активности, и иметь строго определённую сигнатуру:

```
fun buttonClick(view: View) {
    // Код, выполняемый при нажатии кнопки
}
```

Хотя параметр `view` в данном коде может и не использоваться, удалять его ни в коем случае нельзя, потому что при нажатии кнопки система будет искать метод с указанным названием и именно таким набором параметров. При отсутствии подходящего метода программа упадёт.

Далеко не все методы могут быть описаны в интерфейсе, многие приходится подключать непосредственно в коде программы. Для того сначала нужно получить ссылку на объект по идентификатору:

```
<Button
    android:id="@+id/myButton"
    .../>
```

Тогда в коде программы (например, в методе `onCreate`) можно написать примерно такой код:

```
val myButton = findViewById<Button>(R.id.myButton)
myButton.setOnClickListener {
    // Код, выполняемый при нажатии кнопки
}
```

Здесь уже не нужно беспокоиться о наличии подходящих параметров функции – Android Studio автоматически сформирует правильную сигнатуру. Аналогичным образом можно подключить и другие обработчики событий, все они начинаются с `setOn...` и далее конкретное действие. Некоторые специализированные обработчики подключаются более сложным образом, про это будет рассказано в дальнейших темах курса.

Слушатели, оформленные таким образом в виде лямбда-выражения, получают все те же параметры что и обычные слушатели. Например, событие при нажатии клавиши выглядит так:

```
myButton.setOnKeyListener { v, keyCode, event ->
    // Код
}
```

Здесь `v`, `keyCode` и `event` – это параметры, которые передаются в слушатель. Почему же в слушатель `setOnClickListener` ничего не передаётся? На самом деле, передаются, просто скрытым образом:

```
myButton.setOnClickListener { it ->
    // it – ссылка на кнопку, которая вызвала срабатывание слушателя
}
```

Здесь серым цветом условно «дорисован» параметр `it`: он есть, но явно никогда не прописывается. Это особенность лямбда-выражений на языке Kotlin: если параметр всего один, то его можно явно не прописывать и заменять ключевым словом `it`. В слушателях Android этот параметр чаще всего содержит ссылку на объект, который вызвал срабатывание слушателя, это полезно в тех случаях если сразу к нескольким кнопкам нужно привязать один слушатель. Например, в калькуляторе десять цифровых кнопок («0», «1», «2», … «9») и было бы слишком расточительно к каждому из них привязывать отдельный слушатель. Вместо этого можно сделать, например, что-то такое:

```
val calcListener = { it: View ->
    when ((it as Button).text.toString()) {
        "0" -> // Нажата кнопка с текстом «0»
        "1" -> // Нажата кнопка с текстом «1»
        ...
    }
}

calcButton0.setOnClickListener(calcListener)
calcButton1.setOnClickListener(calcListener)
...
calcButton9.setOnClickListener(calcListener)
```

Теперь один и тот же обработчик будет вызываться для каждой из кнопок, а внутри будет производиться анализ конкретной нажатой кнопки.

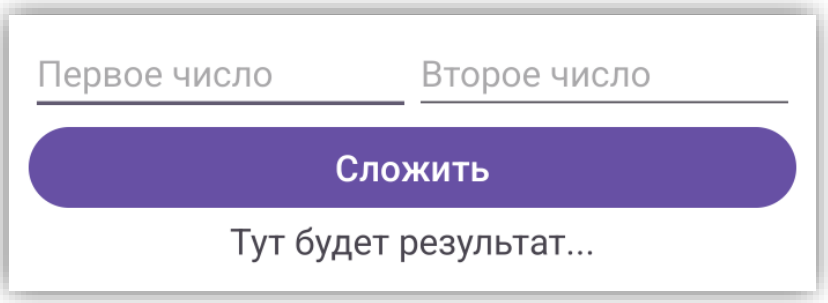
Помимо простого обработчика нажатия `setOnClickListener`, который вызывается при простом нажатии на элемент управления, можно отметить ещё два полезных обработчика:

- `setOnLongClickListener`, который вызывается при долгом нажатии на элемент
- `setOnTouchListener`, который вызывается при касании и отпускании элемента, а также перемещении пальца по поверхности. В качестве параметра он принимает параметр `motionEvent`, который показывает какое именно действие произошло:

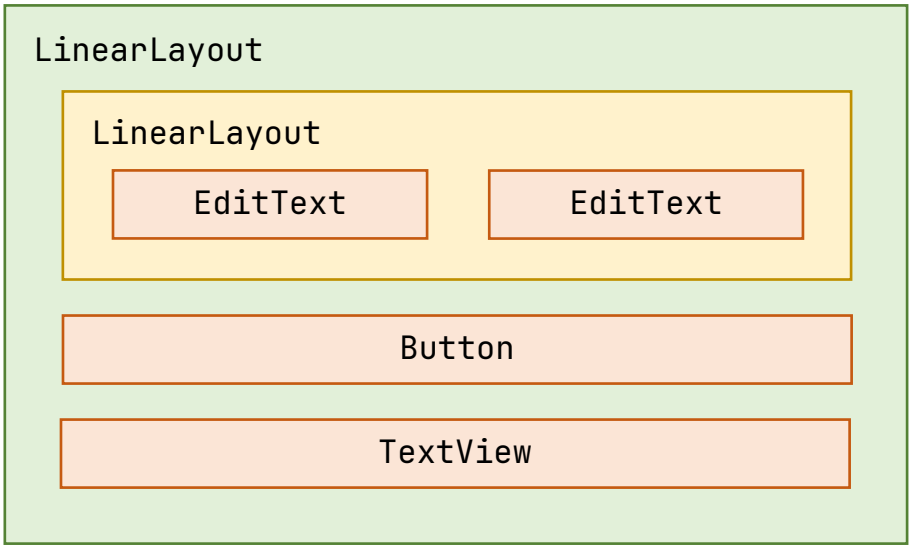
```
someElement.setOnTouchListener { view, motionEvent ->
    when (motionEvent.action) {
        MotionEvent.ACTION_DOWN -> { } // Действия при касании
        MotionEvent.ACTION_UP -> { }   // Действия при отпускании
        MotionEvent.ACTION_MOVE -> { } // Действия при перемещении
    }
    true // Обязательно указывается возвращаемое значение
}
```

Пример: простой калькулятор

В качестве примера для закрепления материала рассмотрим пример простого калькулятора: два поля ввода для чисел, и несколько кнопок для выполнения действий над ними:



Интерфейс будет состоять из общего вертикального контейнера `LinearLayout`, который включает в себя горизонтальный контейнер `LinearLayout` с полями ввода `EditText`, кнопку `Button` и текстовое поле `TextView` для вывода результата:

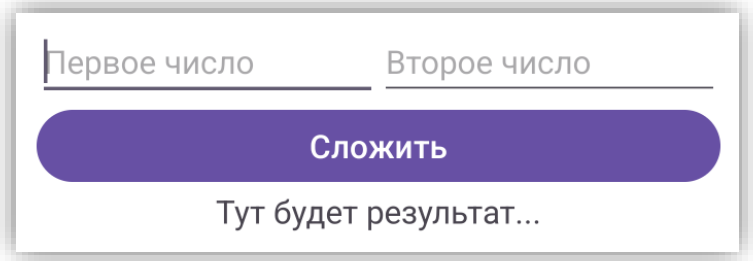


Полный XML-код с интерфейсом должен получиться примерно такой:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@color/white"
    android:padding="16dp">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

        <EditText
            android:id="@+id/etFirst"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:hint="Первое число"
            android:layout_weight="1" />
```

```
        <EditText
            android:id="@+id/etSecond"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:hint="Второе число"
            android:layout_weight="1" />

    </LinearLayout>

    <Button
        android:id="@+id/btAdd"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Сложить"
        android:textSize="18sp"/>

    <TextView
        android:id="@+id/tvResult"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Тут будет результат..."
        android:textAlignment="center"
        android:textSize="18sp"/>

</LinearLayout>
```

Итоговый интерфейс будет выглядеть примерно таким образом:



Теперь напишем программный код на Kotlin, который будет взаимодействовать с этим интерфейсом. Весь код будет помещаться к методе onCreate, в нём после создания проекта уже может быть какой-то автоматически сгенерированный код, его нужно удалить и оставить только минимально необходимый код:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Тут будет код программы
```

```
    }
}
```

Для начала получим в программе ссылки на объекты полей ввода, кнопки и текстовой надписи для результатов:

```
val etFirst = findViewById<EditText>(R.id.etFirst)
val etSecond = findViewById<EditText>(R.id.etSecond)
val btAdd = findViewById<TextView>(R.id.btAdd)
val tvResult = findViewById<TextView>(R.id.tvResult)
```

Названия переменным (и идентификаторам) можно давать любые, которые соответствуют правилам языка. Google рекомендует называть переменные с маленькой буквы, а каждое новое слово должно начинаться с заглавной буквы – это так называемая нотация lowerCamelCase. В данном случае используется также венгерская нотация, когда к каждой переменной добавляется небольшой префикс, соответствующий типу переменной, например, `et` для `EditText`, `tv` для `TextView`, `bt` для `Button` и т. д. Обычно такие нотации используются для удобства и улучшения читаемости кода, их использование является результатом личного выбора. Однако в крупных компаниях часто используется унифицированный подход к оформлению кода, и там конкретный стиль кода обязателен для использования.

Операция `findViewById` может быть затратна по времени, особенно если интерфейс сложный, содержит много элементов, контейнеров, фрагментов и т. д. Иногда это может даже ухудшить производительность программы, особенно если обращение к элементу осуществляется много раз, например, в цикле. Поэтому рекомендуется получать и сохранять ссылки на все нужные элементы при запуске программы, чтобы потом не тратить на это драгоценные мгновения. Альтернативой такому подходу может быть ленивая инициализация, когда переменная получает своё значение в момент первого использования, и в дальнейшем уже не меняется:

```
private val etFirst by lazy { findViewById<EditText>(R.id.etFirst) }
```

Обычно используется какой-то один подход – или получение ссылок вручную, или ленивая инициализация, смешивать их не очень хорошо, т. к. легко в дальнейшем запутаться.

После того как ссылки получены, напишем код для выполнения операции сложения при нажатии кнопки:

```
btAdd.setOnClickListener {
    val num1 = etFirst.text.toString().toFloatOrNull()
    val num2 = etSecond.text.toString().toFloatOrNull()
    if (num1 != null && num2 != null)
        tvResult.text = "$num1 + $num2 = ${num1 + num2}"
    else
        tvResult.text = "Неверное число!"
}
```

Здесь получаются значения из полей ввода `etFirst` и `etSecond`, и сразу преобразуются в тип `Float`. Если в поле оказалось нечисловое значение, то будет получено значение `null`. Если

оба значения числовые – в текстовое поле выводится результат операции, в противном случае выводится информация о неверном числе.

Задание

Доработайте программу-калькулятор: добавив дополнительные действия. Используйте общий слушатель событий, который выполняет действие в зависимости от нажатой кнопки. Интерфейс должен быть примерно таким: