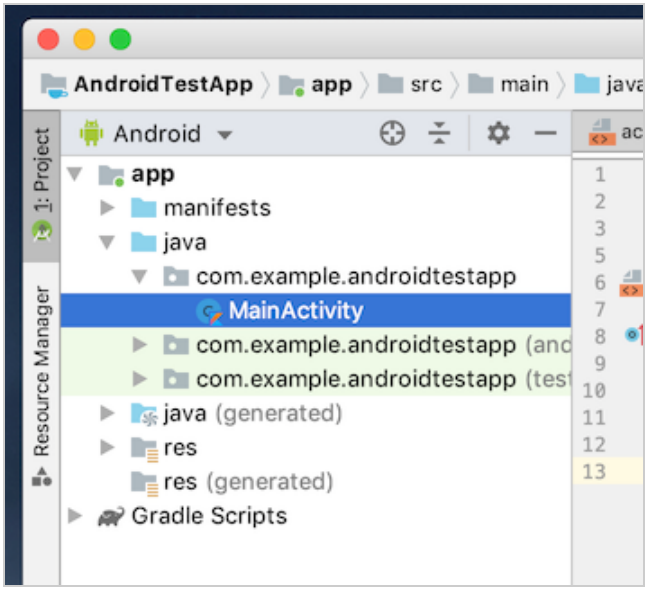
Activityの初期表示

ここではまずプロジェクト作成時に同時に生成されたMainActivityを用いて、画面表示の基本的な所をおさえていきます。

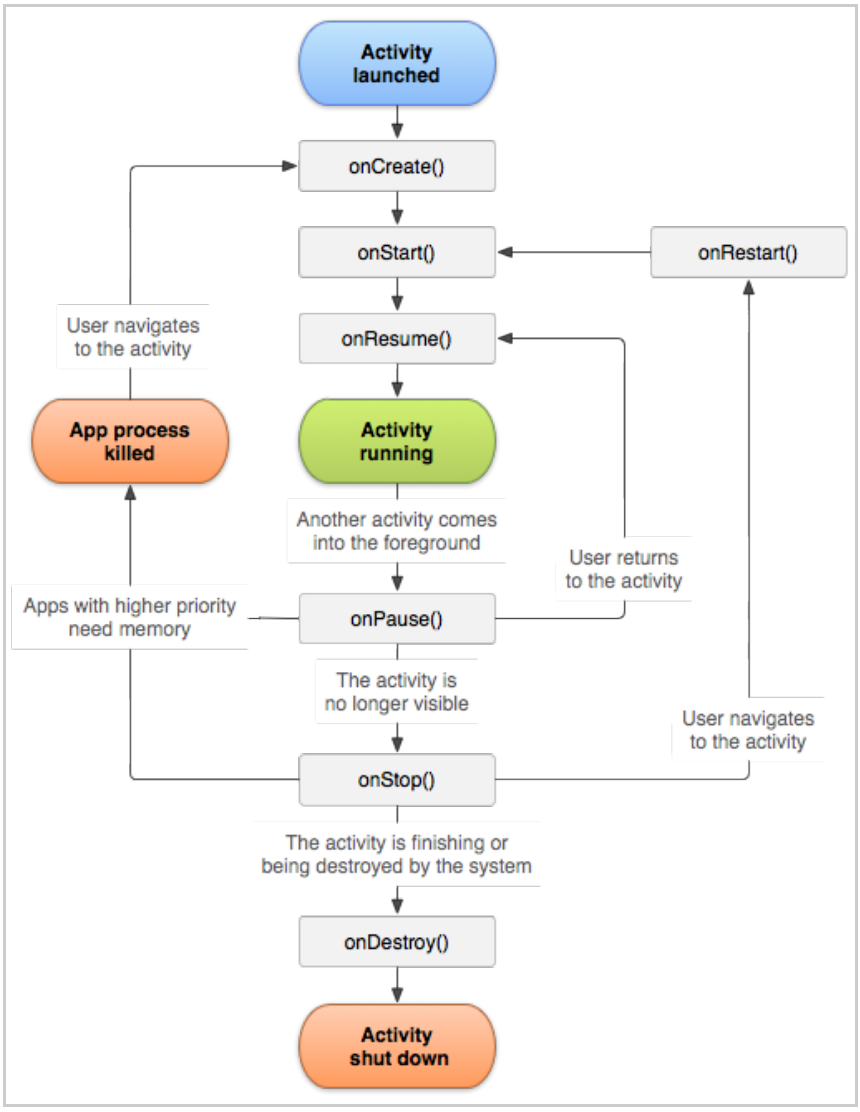
まず、MainActivityのclass内にはあらかじめ以下のメソッドが記述されているはずです。

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

MainActivityが開かれていない場合は、左のファイル一覧から選択して開いてください。



Activityにはライフサイクルがあり、以下のように生成から破棄までの一連の流れに応じたコールバックメソッドが呼ばれます。



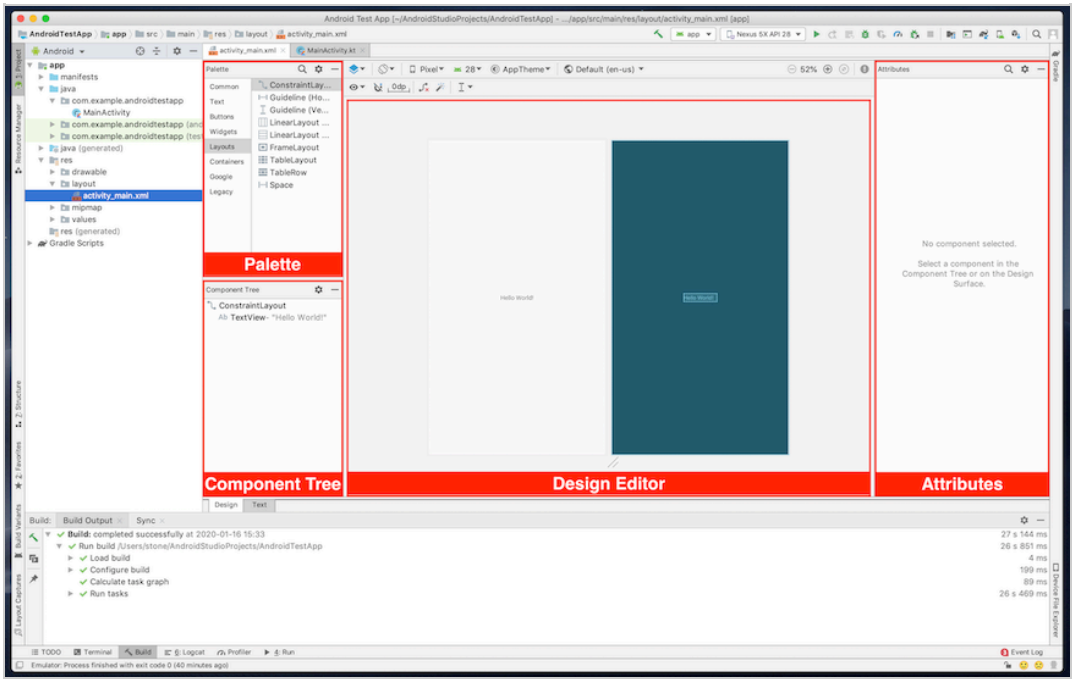
onCreate内で setContentView メソッドを呼ぶことで、レイアウトリソースをActivityにセットしています。画面表示時に配置するUIパーツはこのようにしてAcitivityで読み込みます。

今度はそのレイアウトリソースを見てみましょう。

レイアウトリソースの形式

画面のレイアウトを決めるリソースファイルは基本的にxmlファイルで作成します。
実際にxmlファイルに記述することでレイアウトを作成することもできますが、今回はより直感的にレイアウトを編集でき、視覚的にもわかりやすいLayout Editorで作成していきます。

左のファイル一覧からactivuty_main.xmlを開いてください。下のようなLayout Editorが表示されます。

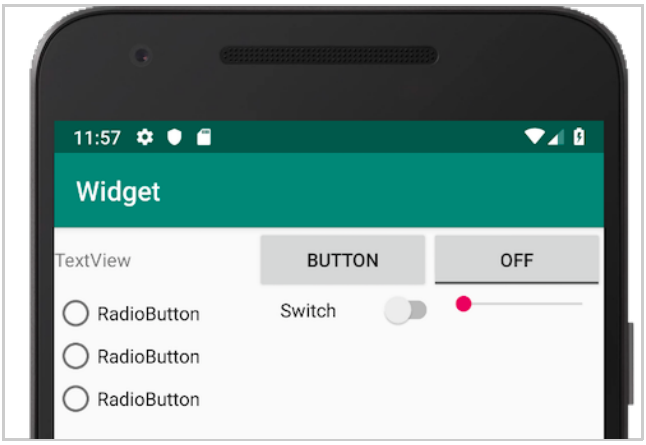


- **Palette**
  ここからレイアウトに追加する要素を選択します。
- **Component Tree**
  現在配置されている要素や要素の階層の確認・編集が可能です。
- **Design Editor**
  実際のレイアウトが表示され、ここでも配置されている要素の確認・編集ができます。
- **Attributes**
  BやCで要素を選択した際に、その要素の属性が一覧で表示され、ここで要素に表示する文字や要素の背景、サイズ等の細かな設定を行うことができます。
  またID属性を設定することにより、その要素をソースコード側が参照できるようになります（そうするとソースコード側から要素の設定ができるようになります）。

レイアウトを構成する要素

画面を構成する要素は大まかに**Widget**と**Layout**の二種類あります。

- **Widget**
  テキストラベルやボタン、入力フィールドなど、各UI部品を指します。
  Widgetは後述のLayout内にしか配置できません。

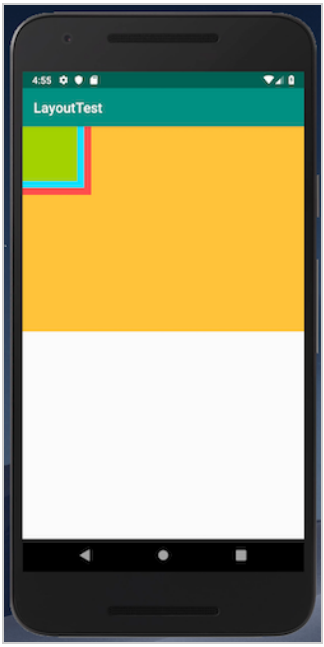


- **Layout**
  WidgetやLayoutを配置するための枠を指します。
  Layoutの種類により、各Widgetの配置方法が異なります。
  Layoutの中にはWidgetおよび子Layoutを入れることができます。

Layoutの種類

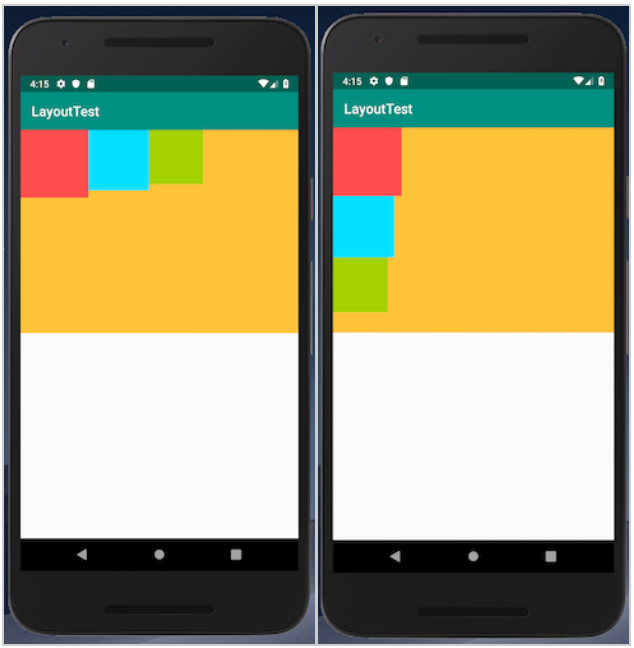
Layoutの種類はいくつかあり、レイアウト方法により使い分けたり、入れ子にして使ったりします。

- FrameLayout



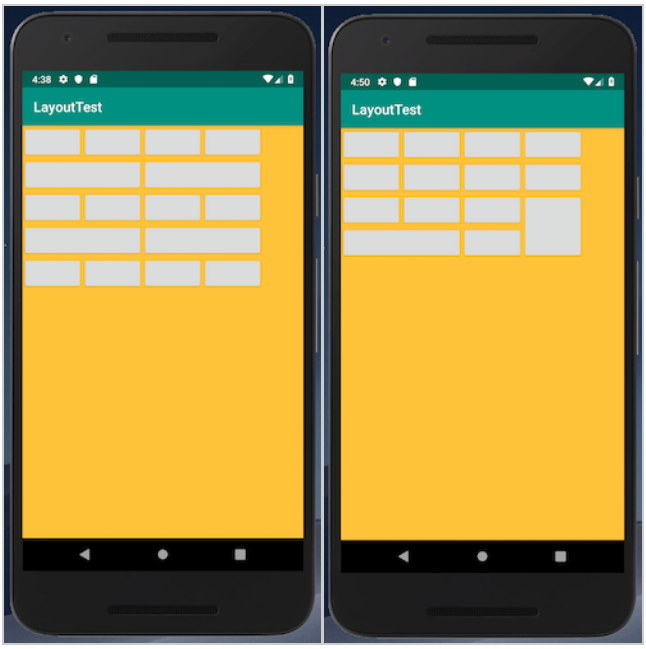
最もシンプルなレイアウトで、要素を左上に配置します。複数要素を子要素とした場合、重なって表示されます。

- LinearLayout



基本と言ってもいいレイアウトで、中に入れた要素を並列に配置します（FrameLayoutとは違い要素は重なりません）。縦横どちらかに並べることができます。

- TableLayout / GridLayout



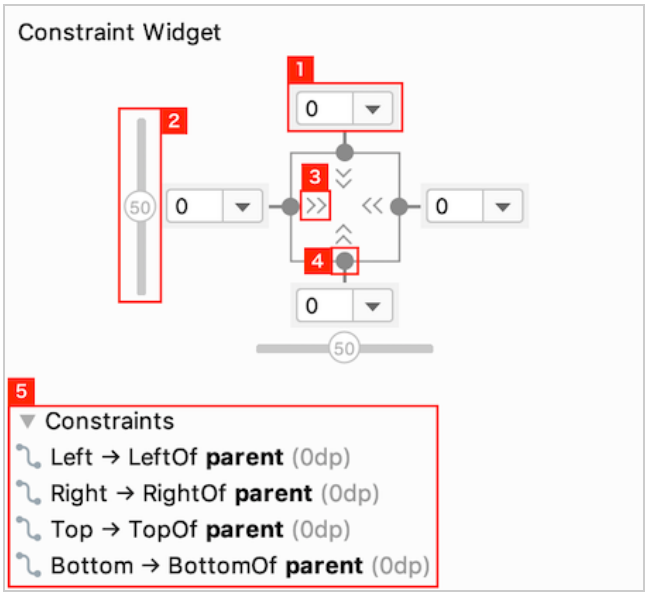
要素を格子状に並べることができます。TableLayoutは列を、GridLayoutは列・行をまたいで配置することができます。
電卓のようなUIを作りたい時便利です。

- RelativeLayout
  相対レイアウトです。基準となる要素を指定し、その要素に対して相対的に配置します。
- ConstraintLayout
  制約を付加することにより要素を配置します。RelativeLayoutに似ていますが、要素は自分で設定せず、自身に対して一番近いものからの相対配置となります。

実際にレイアウトしてみる

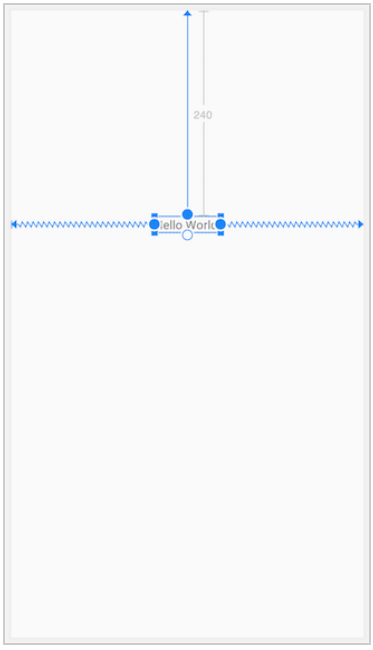
それでは実際に画面のレイアウトを組んでいきましょう。
今回のアプリでは、要素の階層の一番上にデフォルトで配置されている**ConstraintLayout**でレイアウトを組んでいきます。

ConstraintLayoutは制約を用いて要素を配置していくLayoutでしたが、では実際にはどのような形で制約をつけるのでしょうか。
デフォルトで配置されているTextViewをクリックして選択した状態で、右側の**Attributes**を見てみてください。
以下のような情報が見られると思いますが、これが今まさにこのTextViewに付けられている制約の情報です。

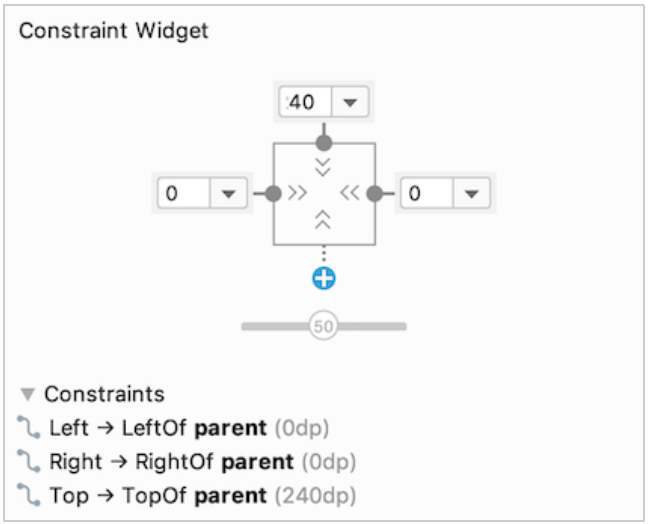


- １・・・**Margin**（制約の距離（単位はdp））
- ２・・・**Bias**（位置の移動（単位は%））
- ３・・・要素の縦/横の長さの決定方法の変更（クリックする度に以下の順で切り替わります）
  Wrap Content（文字列等の要素内の情報に対して適切な長さになります）
  Fixed（現在表示されている長さに固定します）
  Match Constraint（他の制約に従って長さを決定します）
- ４・・・**Delete Constraint**（制約の削除）
- ５・・・上記の上下左右の制約がどのLayout/Widgetに対して付けられているか
  ※Start = 左端、End = 右端、Top = 上端、Bottom = 下端
  例えば『Start → StartOf **parent** (0dp)』は、
  『（TextViewの）Startがparent（親Layout）のStartから0dp離れた位置に付く』
  という意味になります。

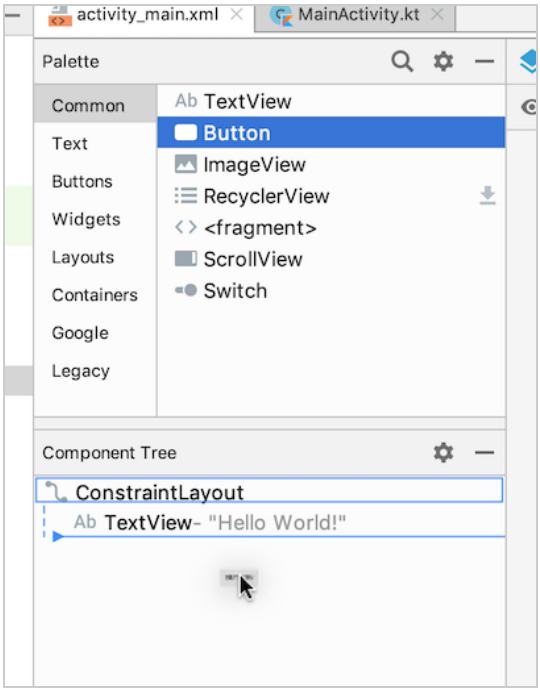
実際に１～４の値を弄ってみると直感的に理解しやすいと思います。
試しにTextViewを以下のような配置に変更してみましょう。



制約は以下のようになります。Bottomの制約を削除するのがポイントです。

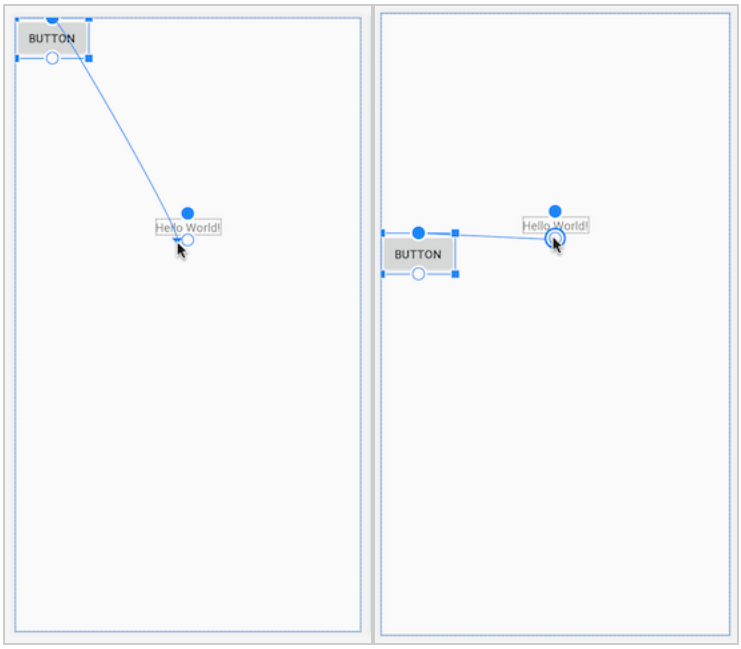


次にボタンを配置してみましょう。
左上のPaletteからその下のComponent Treeまで**Button**をドラッグ&ドロップし、TextViewの下に配置します。



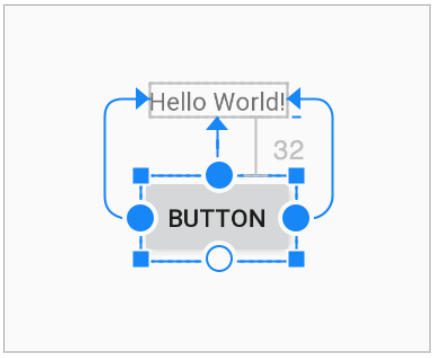
配置されたばかりのButtonにはまだ制約が付いておらず、Component Treeに赤いエラーマークが出てしまっているので、Buttonに新しく制約を付与しましょう。
以下のような配置を目指します。

まずはTextViewとの間が32dpという制約をつけてみます。
Buttonを選択状態にすると中央のDesign Editor上のButtonの上下左右に丸いCreate Constraintマークが表示されるので、ButtonのTopのマークをドラッグしてTextViewのBottomのマークに繋げてみましょう。
Buttonが移動して、0dpとして制約が付与されます。



あとは先ほどと同様に右のAttributesから数値を32に変更します。

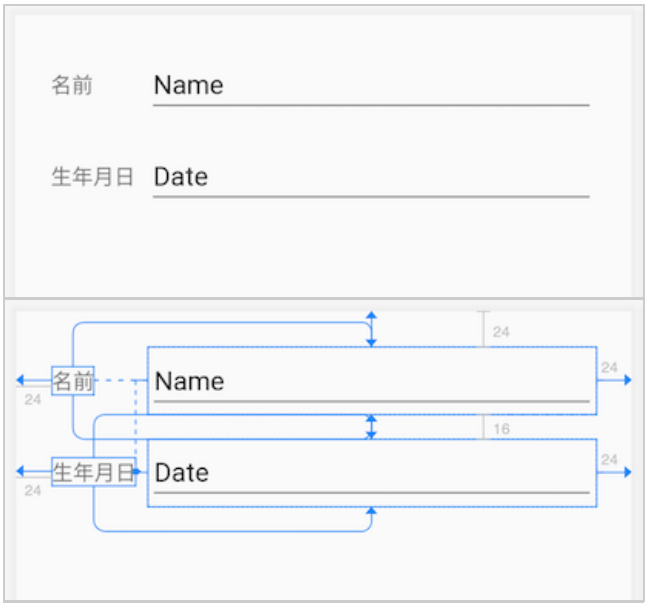
次に、TextViewと中心を合わせます。
Buttonの左右それぞれのConstraintを、TextViewの同じく左右それぞれに繋げてみましょう（距離は0dpのままで大丈夫です）。
そうすると、自然とTextViewの中心に合うようになります。
以下の画像のようになったでしょうか？



この様に、Constraint Layoutでは要素と要素の間に様々な制約を付与することによって、柔軟なレイアウトが可能となっています。

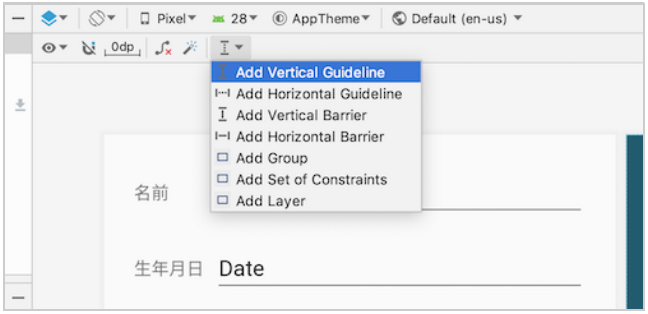
GuidelineとBarrier

Constraint Layoutで以下のようなレイアウトを組んだ時を考えてみます。
（『Name』と『Date』は、左端が揃うようにそれぞれ『生年月日』の右側に対して同じ8dpという制約をつけています）

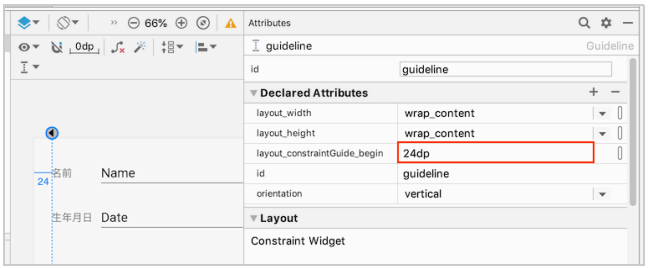


『名前』と『生年月日』の左側に同じ制約が付けられています。
この制約の付け方の場合、片方の数値を変更したらもう片方の制約も編集しなくてはならず、柔軟性に欠けてしまいます。

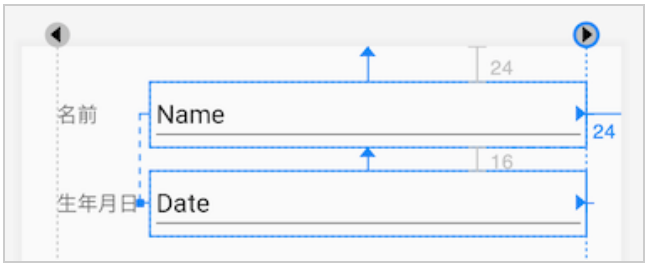
そういった共通の基準がある場合は、その基準を**Guideline**に持たせましょう。
以下のプルダウンからVertical（縦）またはHorizontal（横）のGuidelineを追加することができます。



Vertical Guidelineを追加して、左端からの距離を先ほどの共通している制約と同じ24dpに設定し、『名前』と『生年月日』の左側の制約をGuidelineに対して0dpで付け直すことで、見た目は変わらずに同じ制約を統一化することができました。
（Guidelineの値は以下の場所から変更できます。その下のConstraint Widgetではないので注意！）



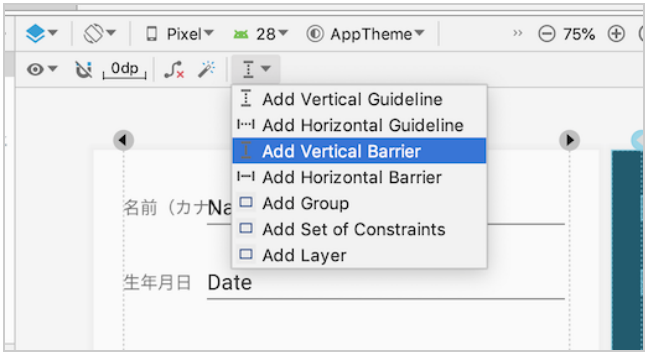
同じように『Name』と『Date』の右側にもGuidelineを設けた方が良いでしょう。
丸で囲まれた『◀』マークをクリックすると『▶』『%』とマークが変化し、向きの変更やパーセントでの指定を選択できるので、状況に応じて使い分けましょう。



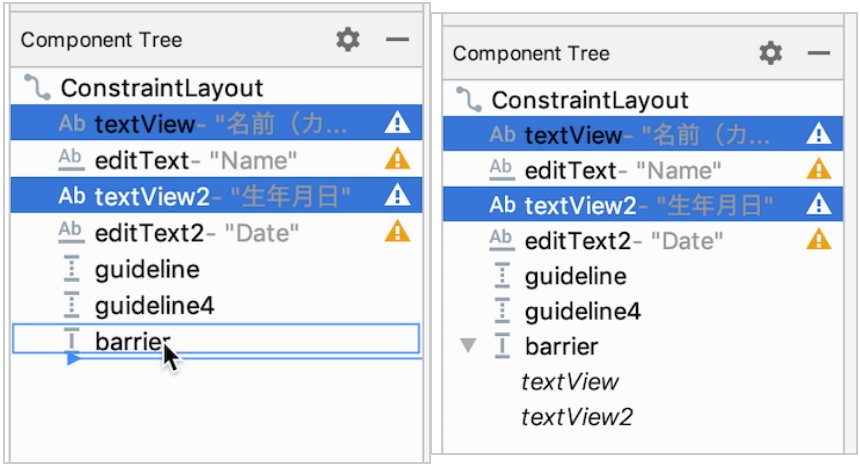
次に、『名前』の文字列を変更した時のことを考えてみましょう。
『名前（カナ）』に変更してみると、右側の『Name』に被ってしまいました。

これは『Name』の左側の制約が『生年月日』の右側に対して付いてしまっているからです。
しかし、だからと言って『名前（カナ）』の右側に合わせてしまうと、また別の修正が行われた時に同じような不都合が起きる可能性があり、一時的な解決にしかなりません。

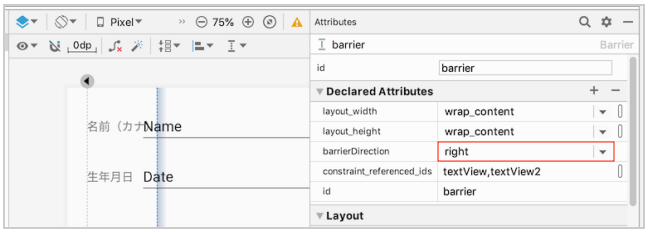
このような場合には**Barrier**を用いて、左側にある要素の表示幅を常に確保するようにしましょう。
BarrierはGuidelineと同じプルダウンから追加でき、また縦と横の2種類があるのも同じです。
今回は縦向きのBarrierを使います。



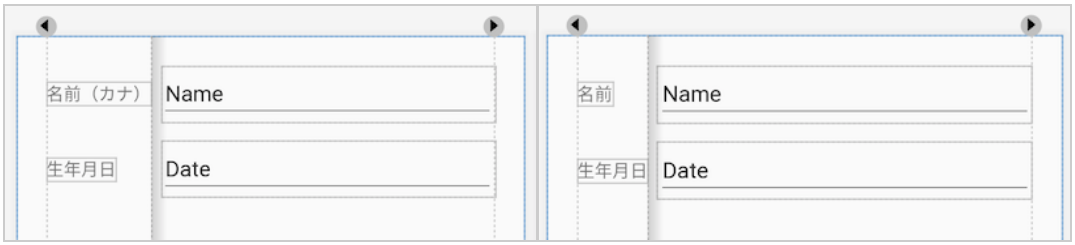
Barrierを追加したら、次に表示幅を確保したい要素をComponent Tree上でBarrierにドラッグ&ドロップします。



今追加した二つのtextView『名前（カナ）』と『生年月日』はそれより右側の要素と被らないようにしたいので、barrierのAttributesから**barrierDirection**をrightに設定します。



そうするとBarrier内の要素群の一番右端にBarrierが配置されるので、『Name』と『Date』の左端がBarrierの右端にくっつくように制約を付け直すと文字の被りが起こらなくなります。



なお、既に存在する制約の対象先だけを変更したい場合は、Attributesの**layout_constraint~**の項目で編集が可能です。
（下の画像は『Name』と『Date』のStart（左側）の制約の対象を、textView2のEnd（右側）からbarrierに変更しているところです）