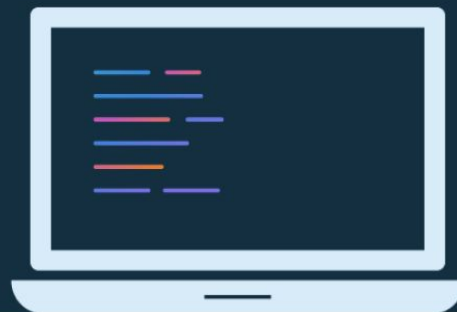




# Lezione 2.2: Activity & layouts



[Blame the machine - Duran duran](#)



# Questa lezione

## Lezione 2.2: Activities & Layouts

- Le Activity
- I Layout in Android
  - ViewGroup(FrameLayout, LinearLayout, ConstraintLayout)
- Data binding
- Gestione gli eventi
- Summary

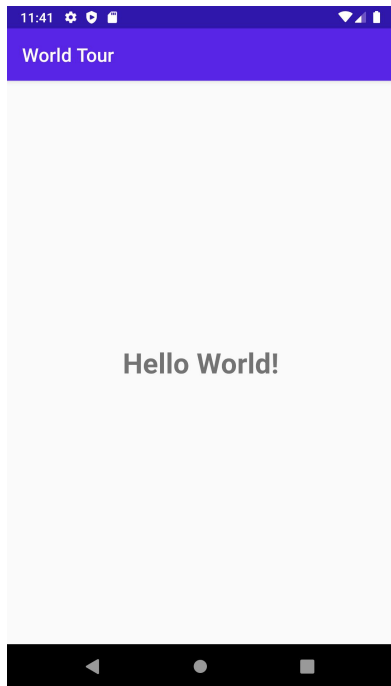
# Questa lezione

Un'app per funzionare ha bisogno almeno di tre cose principali:

1. un'Activity che funga da entry point per l'applicazione
2. un Layout che contenga la definizione degli elementi grafici
3. un file Manifest che descriva il progetto di app

# 1) Le Activity

# Cos'è un'Activity?



- Un'Activity rappresenta una singola cosa che può essere fatta nell'app, un singolo modo per eseguire un obiettivo dell'utente
- Un'app è composta di una o più Activity

# MainActivity.kt

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

# MainActivity.kt

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

**Nota:** le Activity ereditano dalla classe `android.app.AppCompatActivity`, che è figlia di `android.app.FragmentActivity`, che a sua volta è figlia di `android.app.Activity`. Viene usata questa classe perché supporta elementi moderni per l'UI, e può comunque girare su versioni precedenti di Android ("compat").

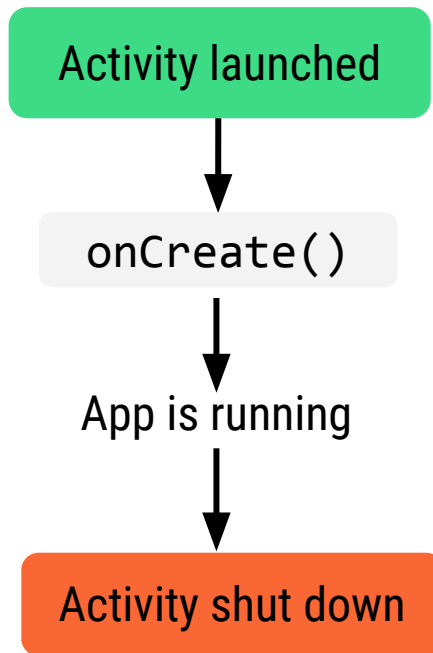
# MainActivity.kt

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

- Serve per inizializzare l'Activity, eseguita quando il runtime effettua la creazione
- Altri metodi che un'Activity può implementare (che vedremo più avanti)
  - onPause (per gestire il caso in cui l'app viene messa in pausa)
  - onStop (quando l'Activity viene fermata)



# Come viene eseguita un'Activity



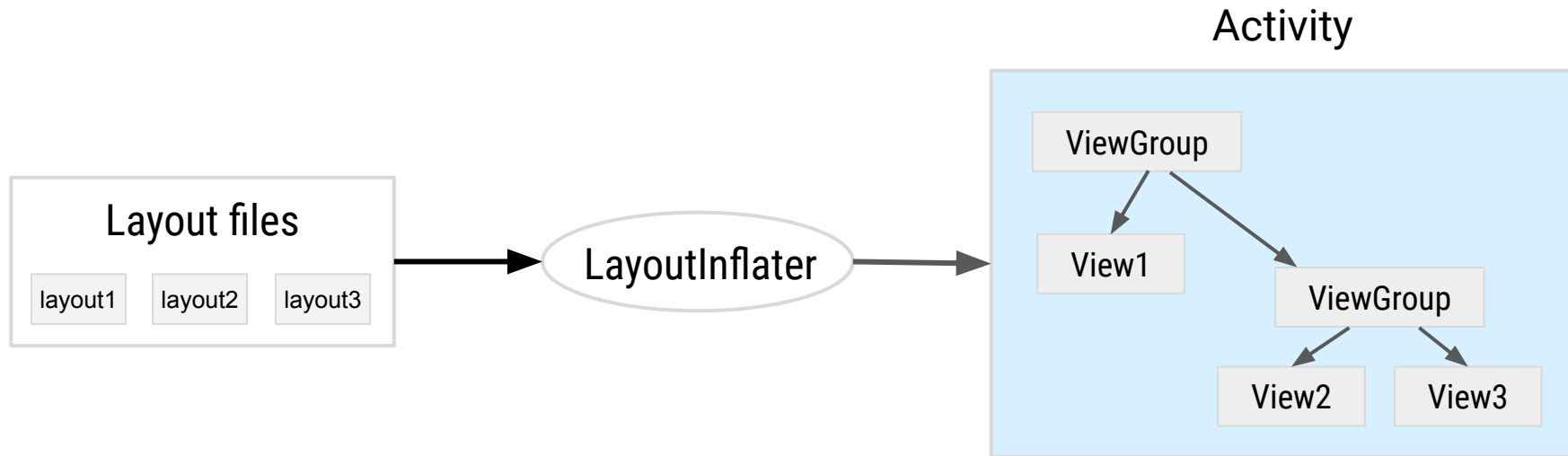
Nota: l'immagine rappresenta una semplificazione del reale ciclo di vita di un'app, che verrà discusso più avanti.

# MainActivity.kt

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        ① super.onCreate(savedInstanceState)  
           setContentView(R.layout.activity_main) ②  
    }  
}
```

Dopo aver (1) chiamato il metodo omonimo della superclasse, (2) in `onCreate` si deve effettuare il “*layout inflation*”, ossia specificare quale file utilizzare per il layout della UI tramite il metodo `setContentView`

# Layout inflation/binding



### 3) Dichiarare l'entry point dell'app

Quando un'app viene eseguita, il runtime cerca l'entry point all'interno del file manifest, in cui occorre definire, tra le informazioni, anche le proprietà di ciascuna Activity

Se l'activity è l'entry point, dovrà dichiarare questo nel tag `<action>`

```
<activity android:name=".MainActivity">
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
```

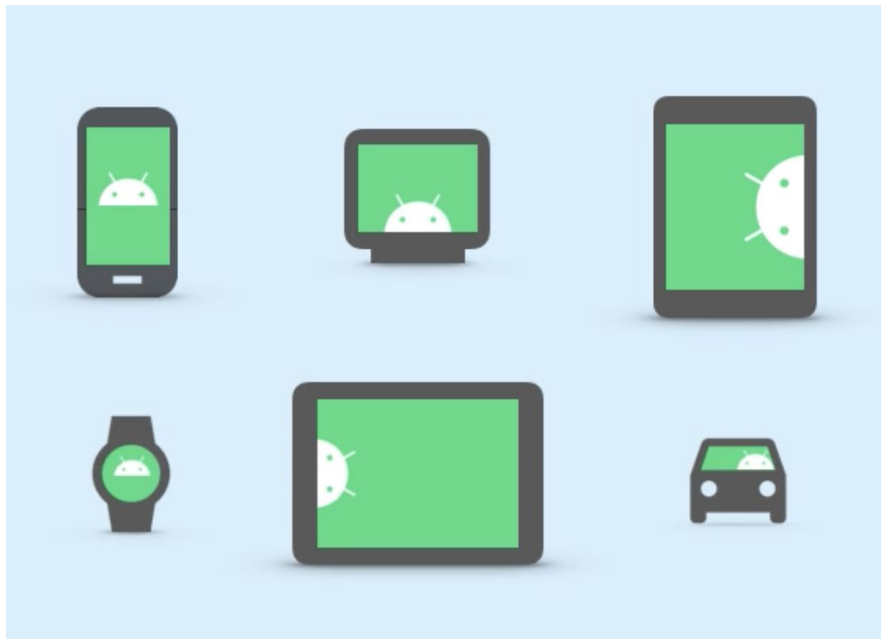
## 2) Layout

# View

- Le View sono gli elementi base per costruire le interfacce utente in Android
  - Racchiuse da un'area rettangolare sullo schermo
  - Responsabili per l'aspetto grafico e la gestione degli eventi (click, focus, drag, ...)
  - Esempi: TextView, ImageView, Button
- Possono essere raggruppate per costruire interfacce più complesse

# Android devices

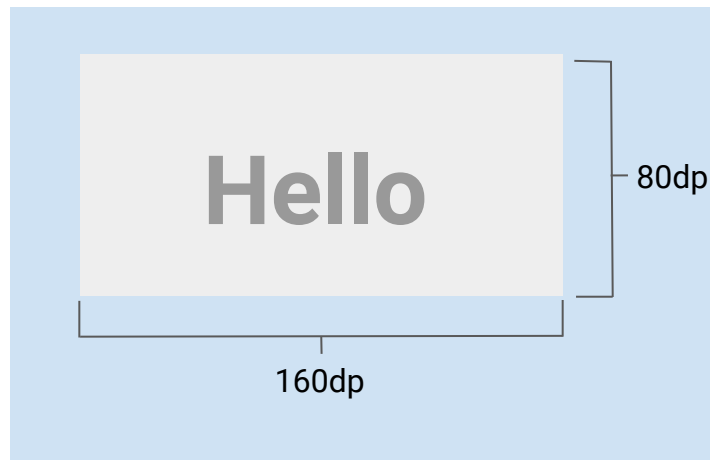
- I dispositivi Android hanno diverse forme e dimensioni
- A seconda del device, gli schermi hanno risoluzioni e densità crescenti
- Gli sviluppatori hanno bisogno di poter specificare le dimensioni di layout in modo consistente tra dispositivi diversi



# Density-independent pixels (dp)

Usiamo dp quando specifichiamo le dimensioni (width o height) dei layout

- I density-independent pixels (dp) tengono conto della densità dello schermo
- Le view in Android sono misurate in density-independent pixels.
- $dp = \frac{\text{width in pixel} * 160}{\text{densità dello schermo}}$

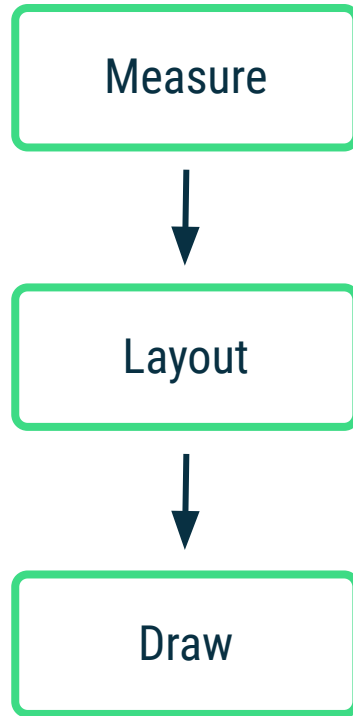




# Screen-density buckets

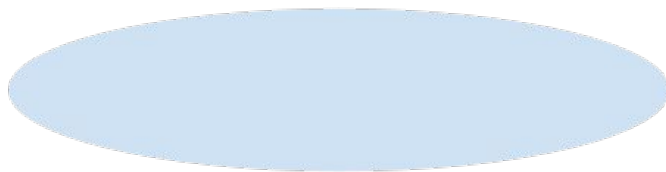
Descrittore di densità	Descrizione	DPI (stimati)
ldpi (poco usato)	Low density	~120dpi
mdpi (densità baseline)	Medium density	~160dpi
hdpi	High density	~240dpi
xhdpi	Extra-high density	~320dpi
xxhdpi	Extra-extra-high density	~480dpi
xxxhdpi	Extra-extra-extra-high density	~640dpi

# Android View rendering cycle



# Drawing region

Quello che  
vediamo:

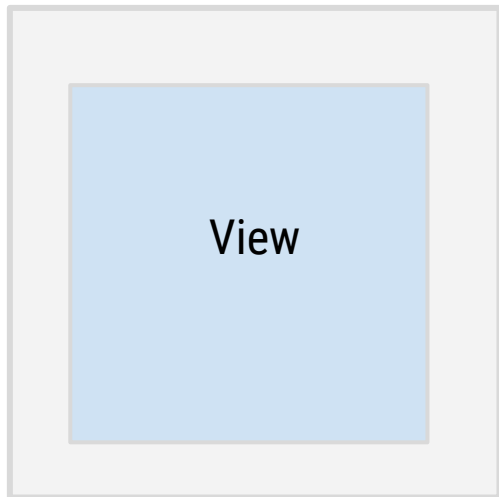


Come viene  
disegnato:

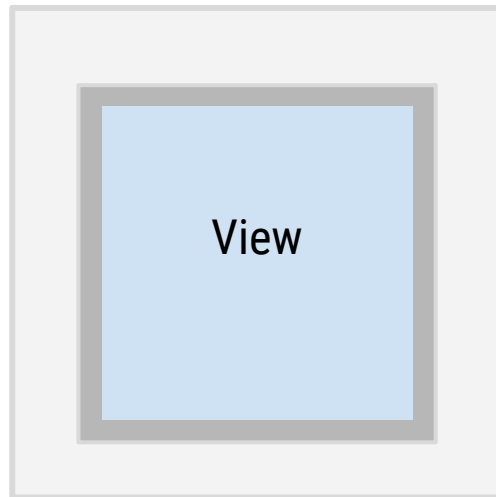


# Margini e padding delle View

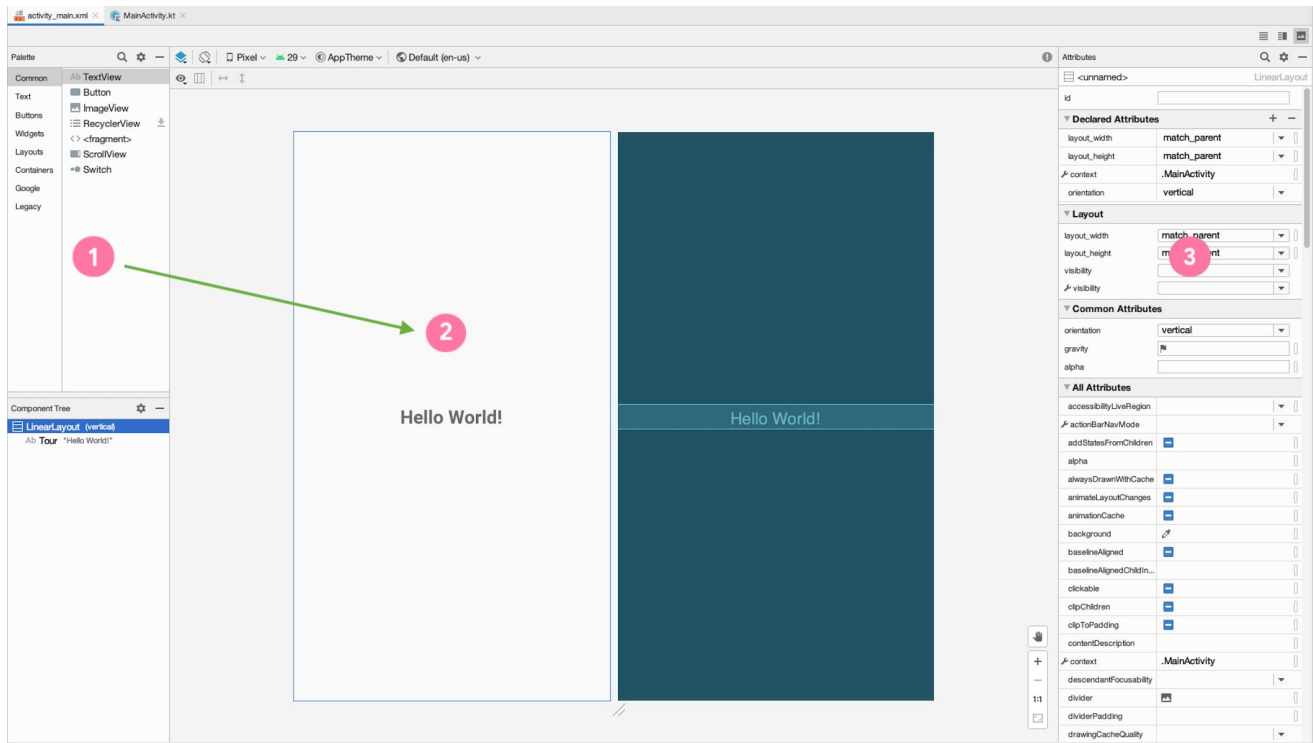
View con margini



View con margini e padding



# Come disegnare un layout: Layout Editor



# File di Layouts

- Un file di layout è un documento XML che consente di descrivere quali View vengono utilizzate e le loro proprietà
- Ciascuna View presente nel file di layout corrisponde ad una classe Kotlin che controlla come quella View funziona

# Esempio: XML per una TextView

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Hello World!"/>
```

Hello World!

# Esempi di proprietà di una View

- wrap\_content (dimensione sufficiente per mostrare l'intero contenuto)

```
android:layout_width="wrap_content"
```

- match\_parent (usa le dimensioni della View genitore)

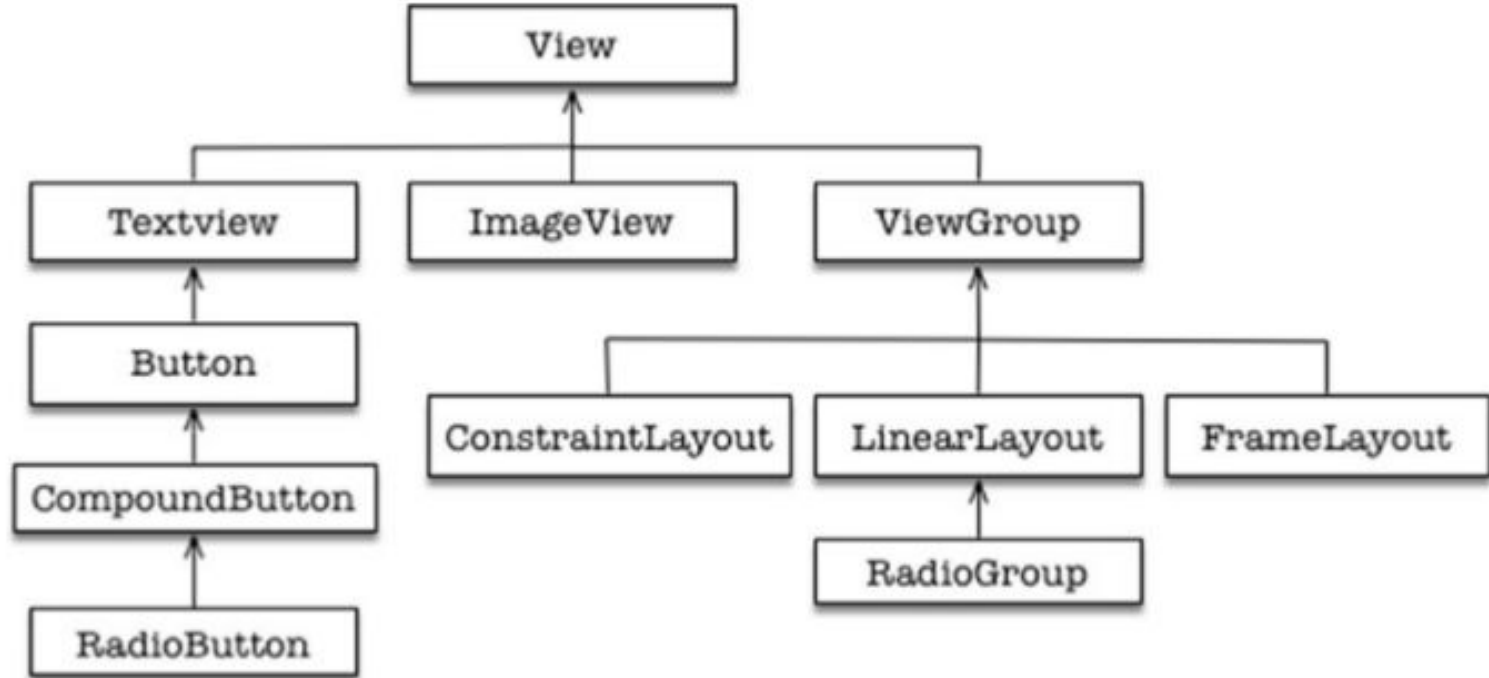
```
android:layout_width="match_parent"
```

- Valore fisso (espresso in dp)

```
android:layout_width="48dp"
```



# Tipologie di View



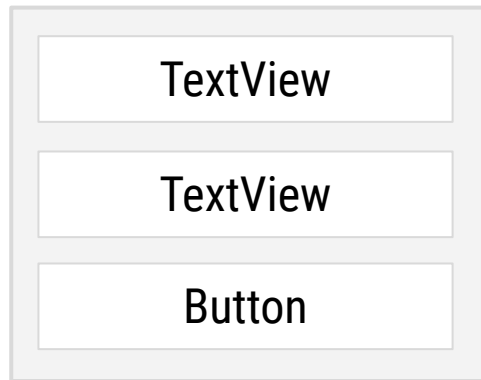
# ViewGroups

Una `ViewGroup` è un contenitore che determina come sono disposte le view

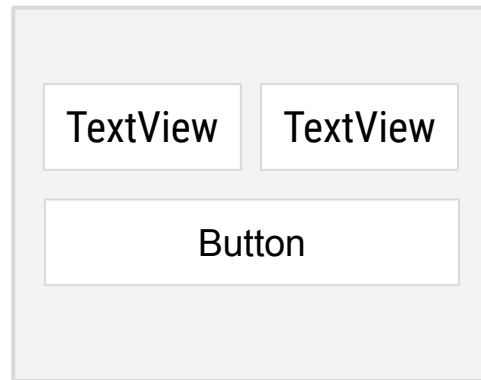
FrameLayout



LinearLayout



ConstraintLayout



Il ViewGroup rappresenta il genitore (parent) e le view contenute sono le figlie.

# FrameLayout: esempio

Un `FrameLayout` generalmente contiene una singola `View` figlia.

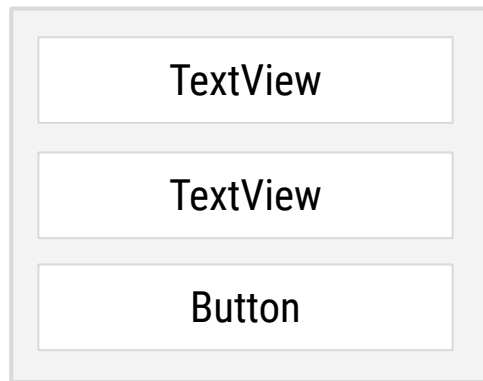
```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Hello World!"/>
</FrameLayout>
```



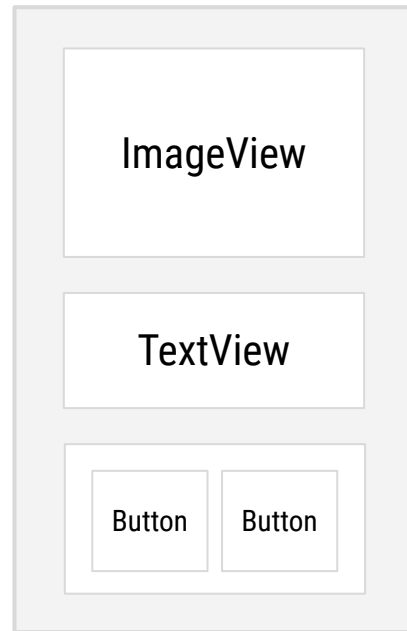
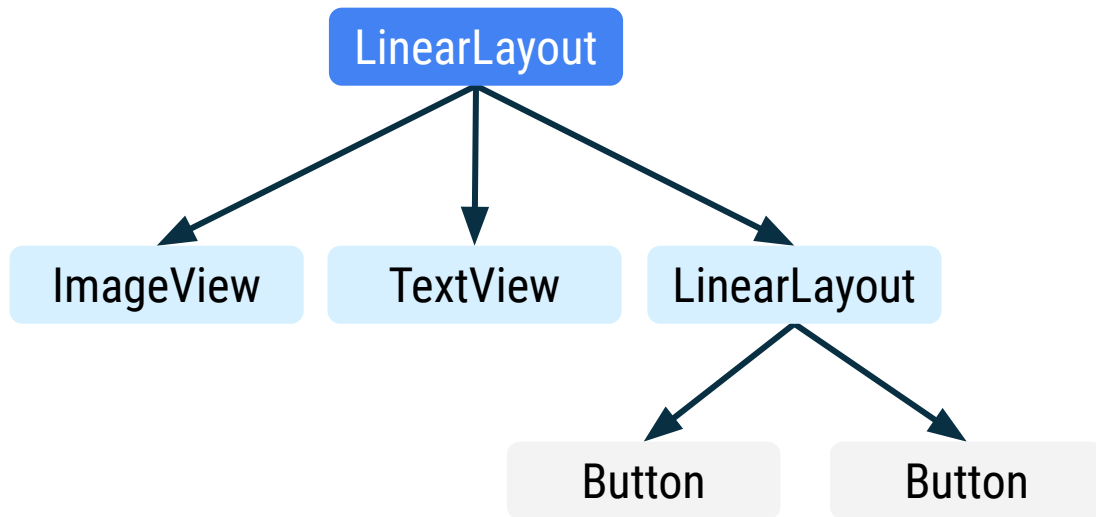
# LinearLayout: esempio

- Allinea le view figlie in una riga o in una colonna
- Setta `android:orientation` al valore `horizontal` o `vertical`

```
<LinearLayout  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:orientation="vertical">  
    <TextView ... />  
    <TextView ... />  
    <Button ... />  
</LinearLayout>
```



# Gerarchia delle view



# ConstraintLayout

# L'annidamento dei layout è costoso

- Annidare diverse ViewGroups tra loro può aiutare ad organizzare il layout, ma talvolta richiede molta computazione
- Le view possono infatti subire più processi di misurazione, e ciò può causare un rallentamento della UI e mancanza di reattività

L'utilizzo del ConstraintLayout contribuisce ad alleviare questi problemi

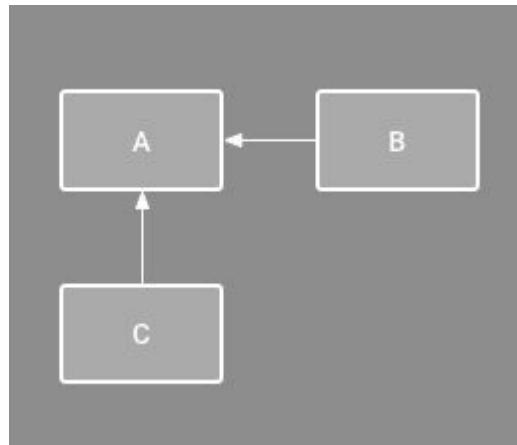
# Cos'è il ConstraintLayout?

- Il layout raccomandato per Android di default
- Risolve le problematiche di efficienza legate a troppi livelli di annidamento di Viewgroup, consentendo comunque di costruire layout complessi
- Posizione e dimensione delle view contenute nel layout vengono definite tramite un insieme di vincoli (constraint)



# Cos'è un constraint?

Una restrizione o limitazione delle proprietà di una View che il layout cerca di rispettare



Ad esempio: B è vincolato ad essere alla destra di A, C è vincolato ad essere al di sotto di A

# Constraint di posizionamento relativo

È possibile definire un vincolo relativo al *parent container* (cioè il suo contenitore)

**Formato:** `layout_constraint<SourceConstraint>_to<TargetConstraint>Of`

Esempio per una `TextView`:

```
app:layout_constraintTop_toTopOf="parent"
```

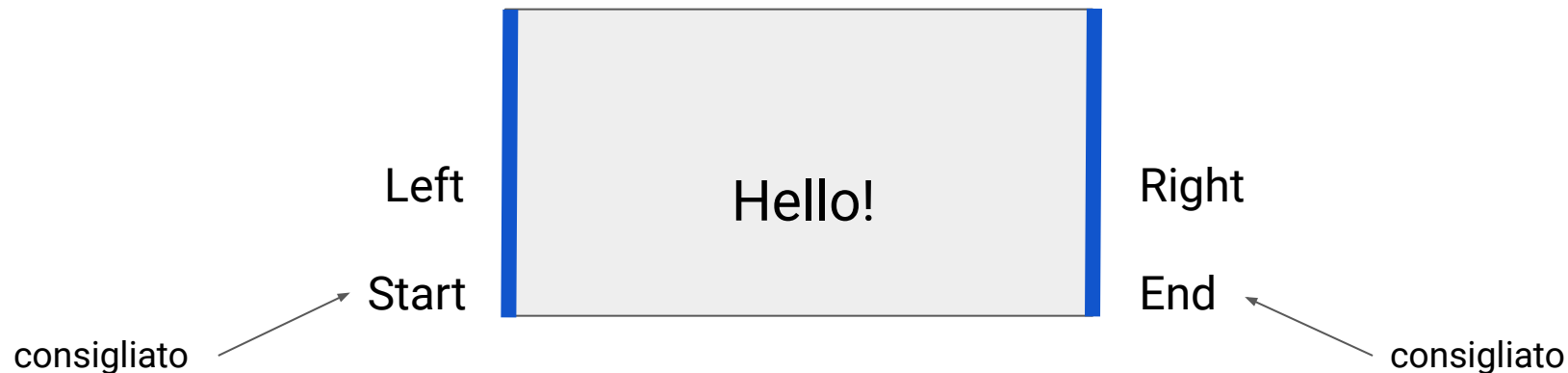
```
app:layout_constraintLeft_toLeftOf="parent"
```



# Constraint di posizionamento relativo



# Constraint di posizionamento relativo

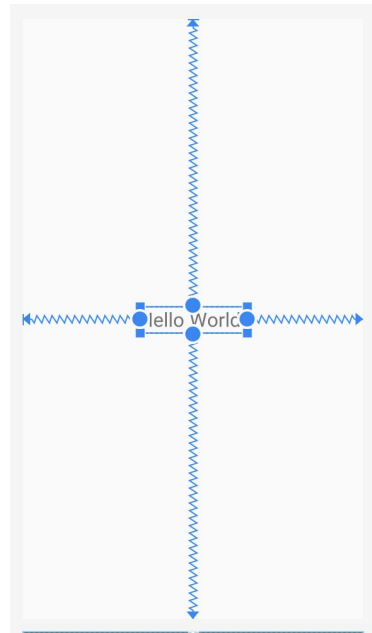


# Esempio di ConstraintLayout

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

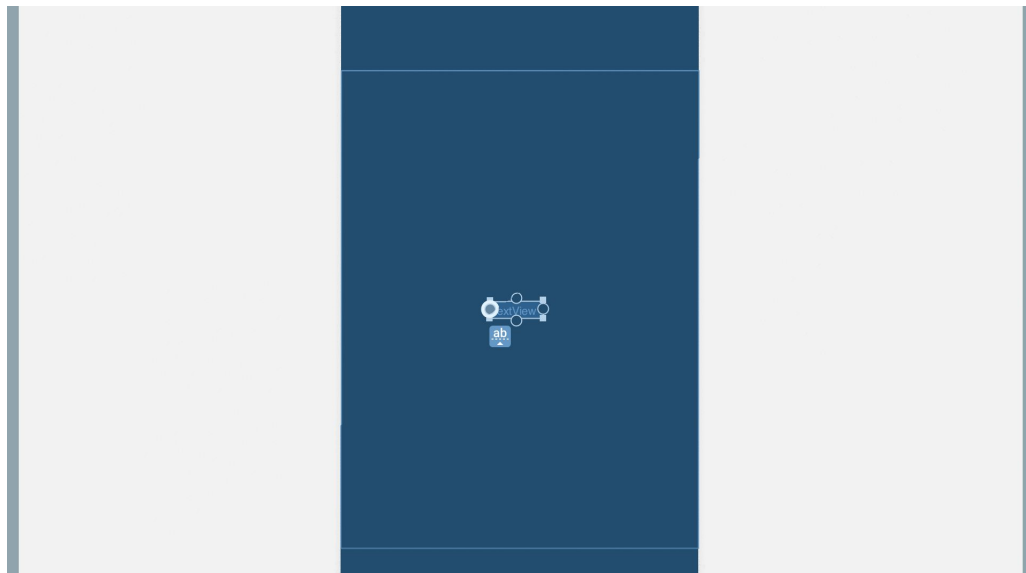
    <TextView
        ...
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

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



# Layout Editor in Android Studio

È sufficiente fare click & drag per aggiungere un vincolo ad una View.



# Constraint Widget nel Layout Editor



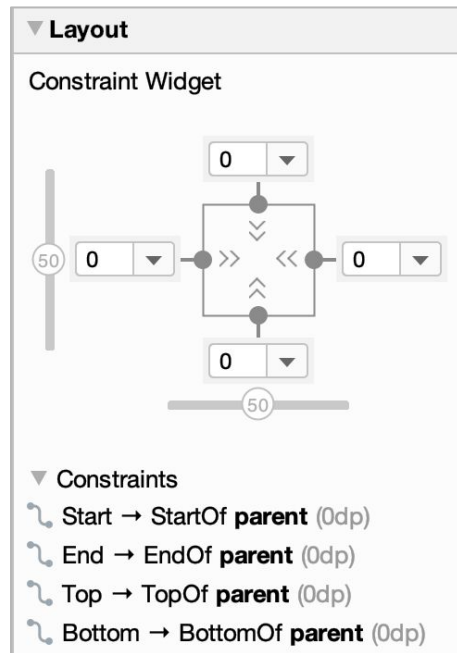
Fisso



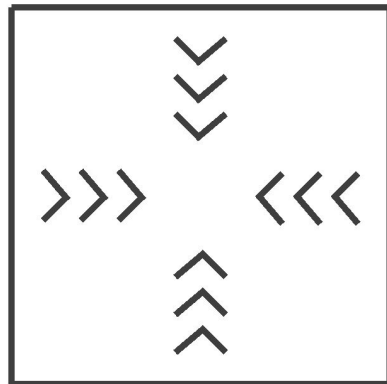
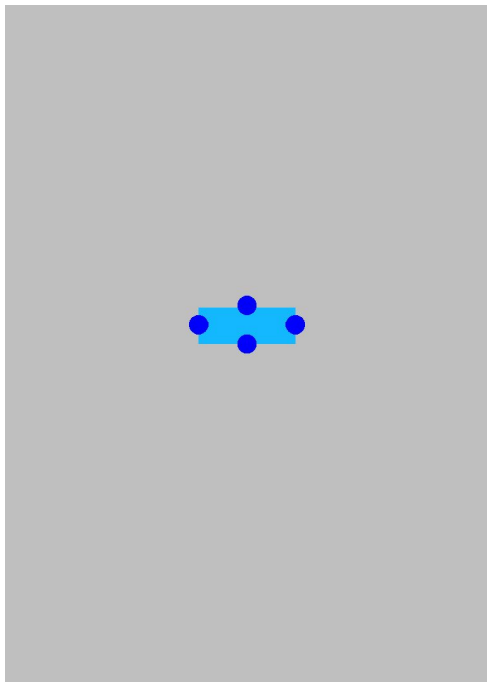
Wrap content



Match constraints



# Wrap content per larghezza/altezza

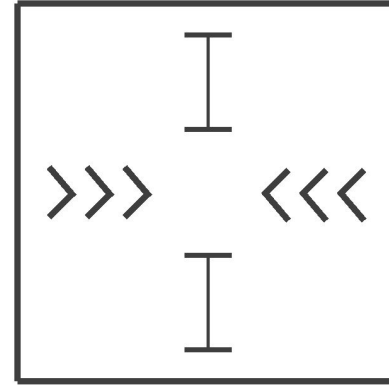
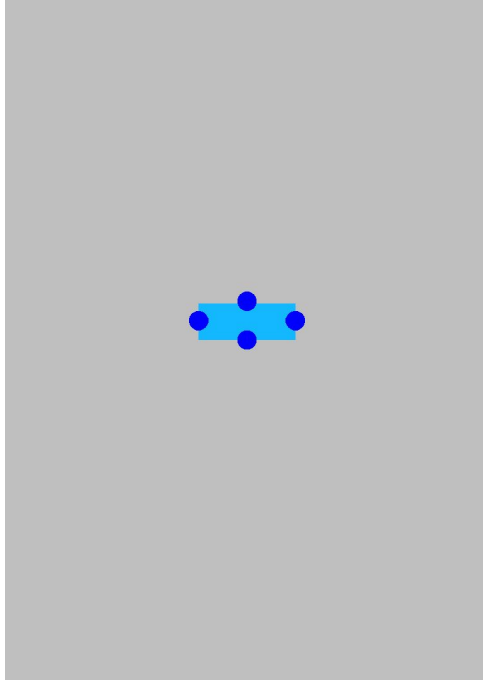


layout\_width      wrap\_content

layout\_height    wrap\_content

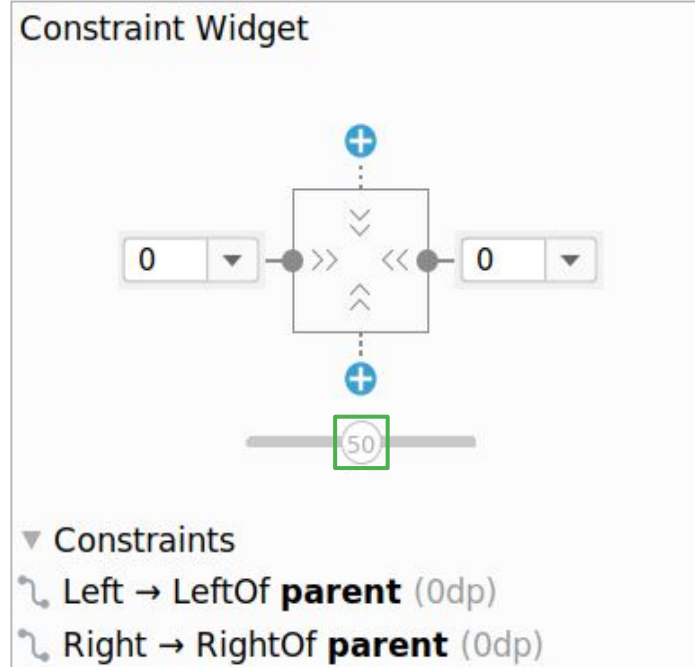
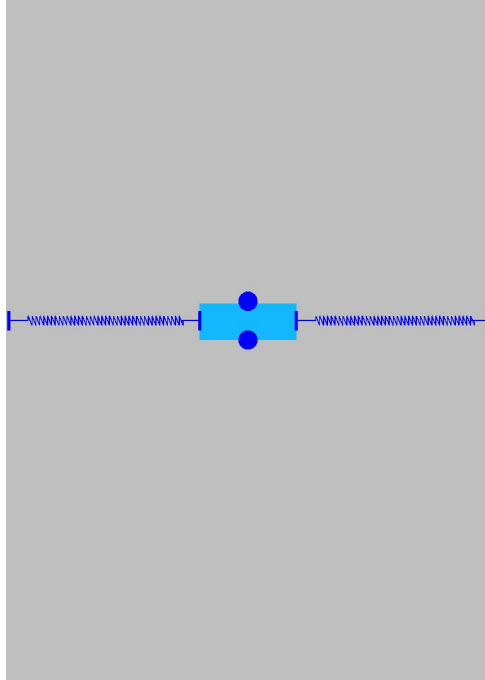


# Wrap content per larghezza, altezza fissa



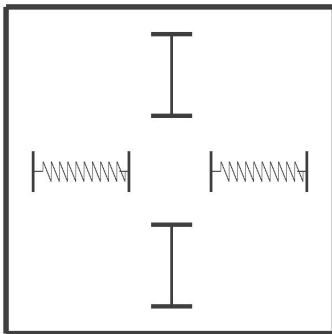
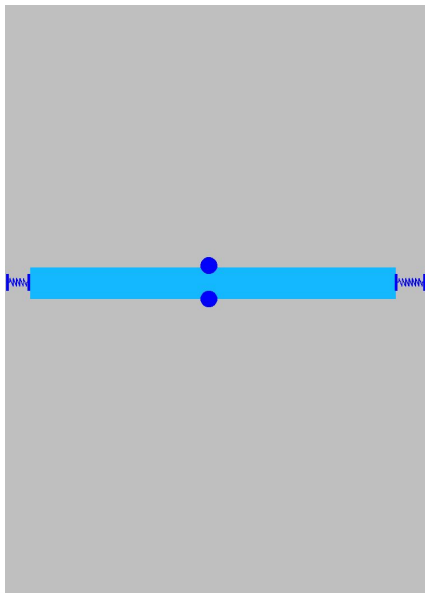
`layout_width`    `wrap_content`  
`layout_height`   `48dp`

# Centrare una view orizzontalmente



# Uso del match\_constraint

Non si può usare `match_parent` su una child view (come nel `LinearLayout`), ma `match_constraint`



`layout_width`     `0dp(match_constraint)`

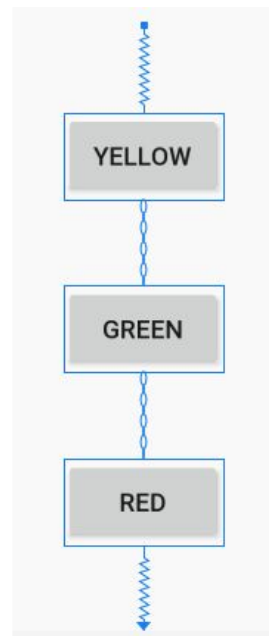
`layout_height`   `48dp`

# Chains

- Un insieme di view che sono linkate l'una all'altra con constraint bidirezionali
- Consentono di posizionare una view in relazione alle altre
- Possono essere linkate orizzontalmente o verticalmente
- Forniscono molte delle funzionalità di un LinearLayout

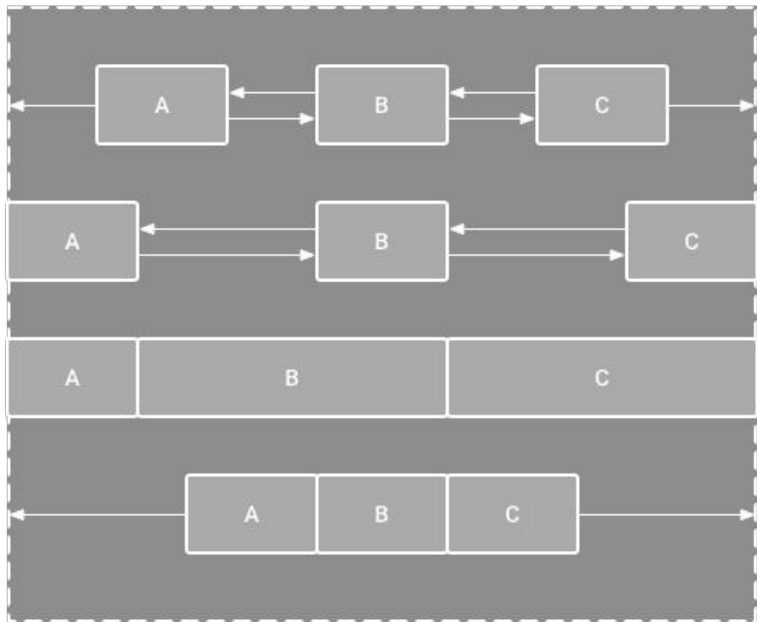
# Creare una Chain nel Layout Editor

1. Selezionare gli oggetti che vogliamo nella chain
2. Right-click e selezioniamo **Chains**.
3. Creiamo una chain orizzontale o verticale.



# Chain styles

Esistono diversi stili per le chain, che organizzano lo spazio tra le views:



Spread Chain

Spread Inside Chain

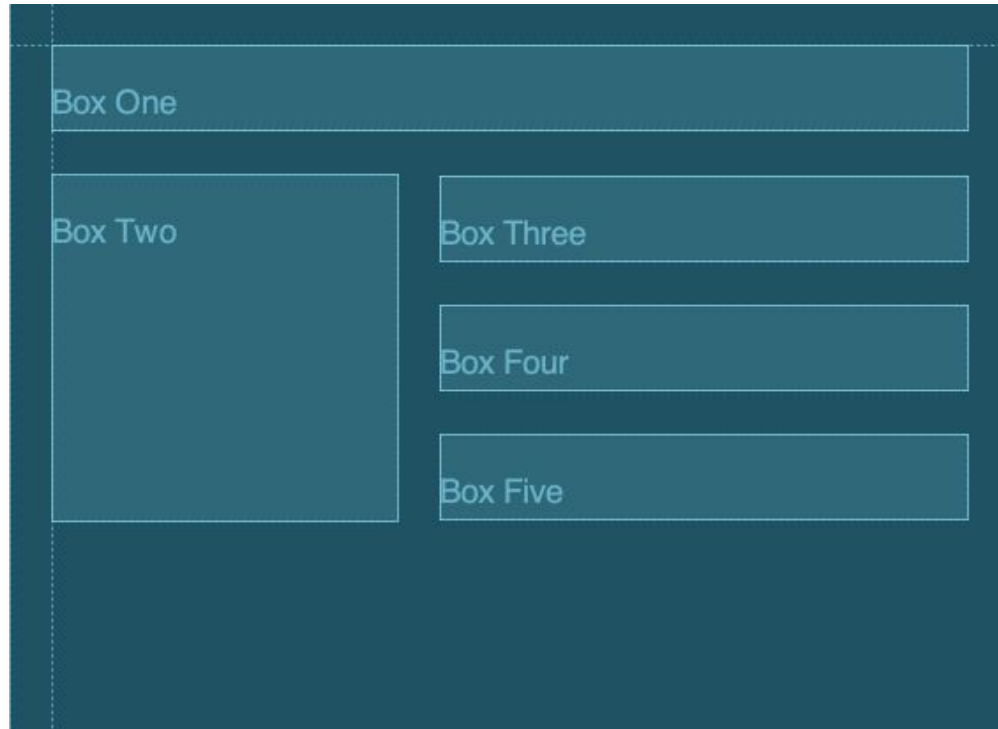
Weighted Chain

Packed Chain

# Elementi utili: Guidelines

- Consentono di posizionare più views su una guida
- Verticali o orizzontali
- Supportano lo sviluppo collaborativo con i team dei grafici
- Non vengono mostrate sul device

# Guidelines in Android Studio





# Example Guideline

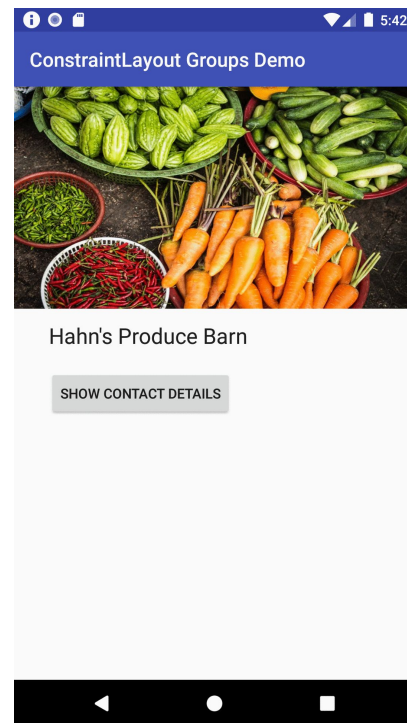
```
<ConstraintLayout>
    <androidx.constraintlayout.widget.Guideline
        android:id="@+id/start_guideline"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        app:layout_constraintGuide_begin="16dp" />
    <TextView ...
        app:layout_constraintStart_toEndOf="@id/start_guideline" />
</ConstraintLayout>
```

# Creare Guidelines

- `layout_constraintGuide_begin`
- `layout_constraintGuide_end`
- `layout_constraintGuide_percent`

# Elementi utili: Groups

- Controllano la visibilità di un insieme di widgets
- La visibilità del gruppo può essere attivata o disattivata nel codice.



# Esempio di group

```
<androidx.constraintlayout.widget.Group  
    android:id="@+id/group"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    app:constraint_referenced_ids="locationLabel,locationDetails"/>
```

# Codice per controllare il group

```
override fun onClick(v: View?) {  
    if (group.visibility == View.GONE) {  
        group.visibility = View.VISIBLE  
        button.setText(R.string.hide_details)  
    } else {  
        group.visibility = View.GONE  
        button.setText(R.string.show_details)  
    }  
}
```

# Data binding

# Ottenere un riferimento ad una view

Per manipolare una View è necessario prima ottenere un riferimento ad essa. Questa operazione si può effettuare in vari modi.

Il meccanismo più semplice è utilizzare il metodo `findViewById(ID)`

- Specificando come parametro il resource ID della risorsa che stiamo cercando
- Il metodo ritorna un riferimento alla risorsa se questa esiste, altrimenti causa un crash a runtime se la risorsa non è nel layout corrente

# Ottenere un riferimento ad una view

Esempio:

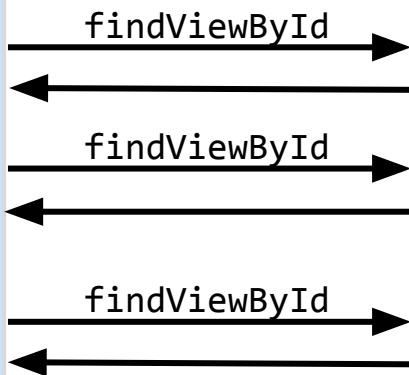
MainActivity.kt

```
val name = findViewById(...)
val age = findViewById(...)
val loc = findViewById(...)

name.text = ...
age.text = ...
loc.text = ...
```

activity\_main.xml

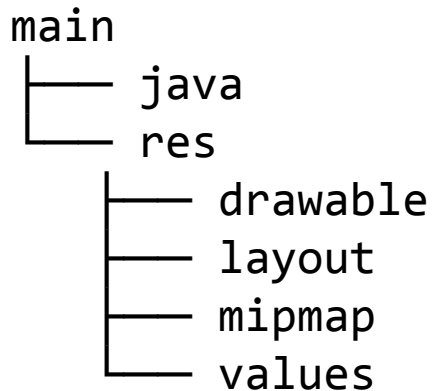
```
<ConstraintLayout ... >
  <TextView
    android:id="@+id/name"/>
  <TextView
    android:id="@+id/age"/>
  <TextView
    android:id="@+id/loc"/>
</ConstraintLayout>
```





# Directory per le risorse

Nuove risorse possono essere aggiunte all'app includendole nelle sottodirectory della cartella `res`



# Resource IDs

- Ogni risorsa ha un *resource ID* che possiamo usare per accedervi
- Quando diamo nomi alle risorse, la convenzione è di usare caratteri minuscoli con underscore (ad esempio, `activity_main.xml`).
- Android autogenera una classe `R.java` contenente riferimenti a tutte le risorse dell'app
- I singoli elementi sono referenziati in modo gerarchico in base alle directory:  
`R.<resource_type>.<resource_name>`

Esempi:

<code>R.drawable.ic_launcher</code>	<code>(res/drawable/ic_launcher.xml)</code>
<code>R.layout.activity_main</code>	<code>(res/layout/activity_main.xml)</code>

# Resource IDs per le view

Anche delle singole view possono avere un resource ID. E' sufficiente aggiungere l'attributo `android:id` alla View in XML per assegnarle un nome, tramite la sintassi `@+id/name`

```
<TextView
    android:id="@+id/helloTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"/>
```

All'interno dell'app, ora è possibile riferirsi a questa TextView con `R.id.helloTextView`

# Modificare una View dinamicamente

Dentro `MainActivity.kt`:

Ottieni un riferimento alla View nella gerarchia delle views tramite il resource ID:

```
val resultTextView: TextView = findViewById(R.id.textview)
```

Modifica le proprietà o chiama i metodi dell'istanza della View (alcune view hanno delle proprietà specifiche, come il testo per una TextView o un Button)

```
resultTextView.text = "Goodbye!"
```

# Approccio data binding

Un approccio più efficiente e meno soggetto ad errori è quello del data binding, che ci consente un accesso diretto a tutti gli elementi del layout

MainActivity.kt

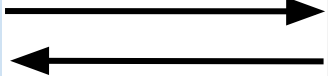
```
Val binding:ActivityMainBinding
```

```
binding.name.text = ...
```

```
binding.age.text = ...
```

```
binding.loc.text = ...
```

initialize binding



activity\_main.xml

```
<layout>
  <ConstraintLayout ... >
    <TextView
      android:id="@+id/name"/>
    <TextView
      android:id="@+id/age"/>
    <TextView
      android:id="@+id/loc"/>
  </ConstraintLayout>
</layout>
```

# Modificare il file build.gradle

```
android {  
    ...  
    buildFeatures {  
        dataBinding true  
    }  
}
```

# Aggiungere un tag layout

**<layout>**

```
<androidx.constraintlayout.widget.ConstraintLayout>  
    <TextView ... android:id="@+id/username" />  
    <EditText ... android:id="@+id/password" />  
</androidx.constraintlayout.widget.ConstraintLayout>
```

**</layout>**

# Layout inflation con data binding

E' sufficiente sostituire questo:

```
setContentView(R.layout.activity_main)
```

con questo:

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

```
binding.username = "Melissa"
```



# Data binding: variabili di layout

```
<layout>
  <data>
    <variable name="name" type="String"/>
  </data>
  <androidx.constraintlayout.widget.ConstraintLayout>
    <TextView
      android:id="@+id/textView"
      android:text="@{name}" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

In MainActivity.kt:

```
binding.name = "John"
```

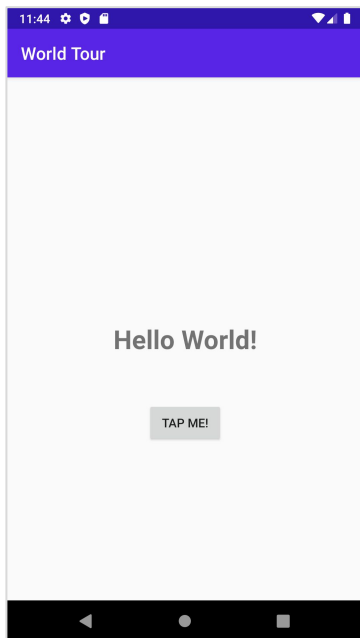
# Data binding: espressioni di layout

```
<layout>
  <data>
    <variable name="name" type="String"/>
  </data>

  <androidx.constraintlayout.widget.ConstraintLayout>
    <TextView
      android:id="@+id/textView"
      android:text="@{name.toUpperCase()}" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

# Gestione degli eventi

# Definire il comportamento di un app



Modificare l'Activity in modo che l'app risponda all'input dell'utente (ad esempio la pressione di un bottone)

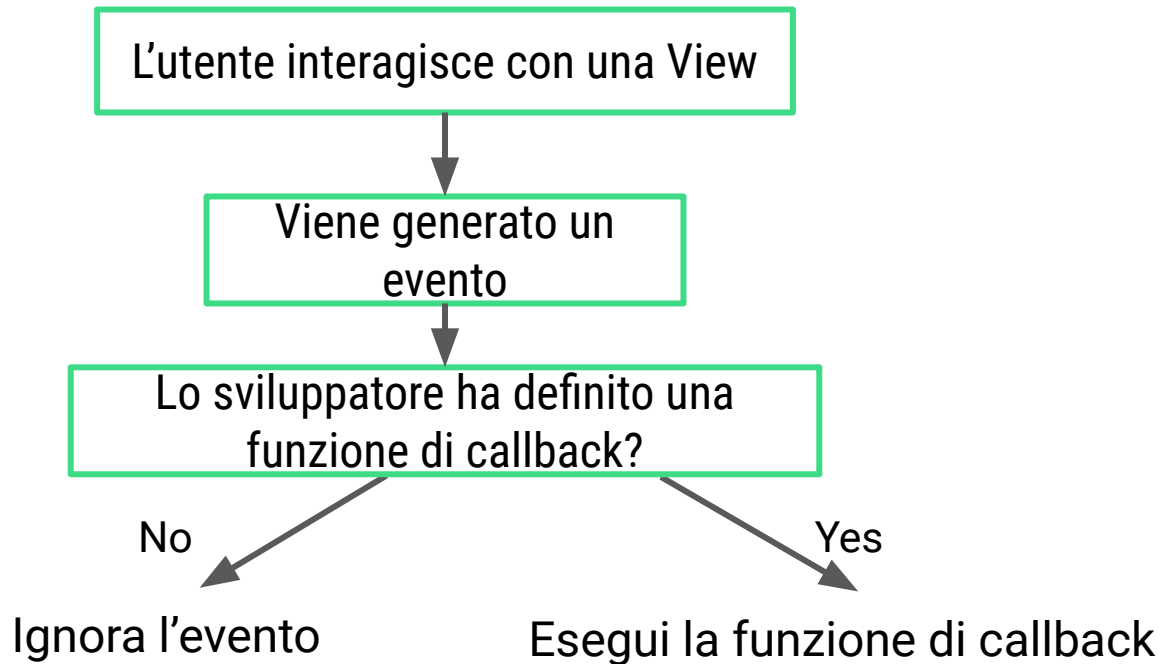
# Gestione degli eventi

Il più semplice evento è il click su un bottone.

- L'approccio più semplice per legare l'esecuzione di una funzione a questo evento è di usare l'attributo `android:onClick` nella view, settandolo al nome di quella funzione
- es: funzione che incrementa un contatore ad ogni click su un bottone

Questo meccanismo è però limitato al solo evento di click, ma esistono molti altri tipi di eventi, che possono essere gestiti tramite dei Listeners

# Configurare un listener per eventi specifici



# View.OnClickListener

View.OnClickListener è una interfaccia che descrive la funzione di callback che viene invocata quando avviene un click su una View.

Contiene solo un metodo astratto `onClick(v: View) : Unit`

Definire un listener per una View (ad esempio un Button) significa: (1) implementare questa interfaccia, (2) crearne un'istanza e (3) passarla al metodo `setOnClickListener` della View

# View.OnClickListener

```
class MainActivity : AppCompatActivity(){  
    override fun onCreate(savedInstanceState: Bundle?) {  
        ...  
        val button: Button = findViewById(R.id.button)  
        val myListener = MyButtonListener(applicationContext)  
        button.setOnClickListener(myListener)  
    }  
}  
  
class MyButtonListener(val c: Context): View.OnClickListener{  
    override fun onClick(v: View){ ... }  
}
```

2

1

3



# View.OnClickListener: soluzione più sintetica

L'idea è di sfruttare il concetto di **object** in Kotlin per creare un'istanza senza definire la classe.

L'uso della object declaration è utile per creare classi che ammettono un'unica istanza, mentre companion object serve per definire una porzione statica di una classe

```
object Singleton{  
  
    var variableName = "I am Var"  
  
    fun printVarName(){  
  
        println(variableName)  
    }  
}
```

# View.OnClickListener: soluzione più sintetica

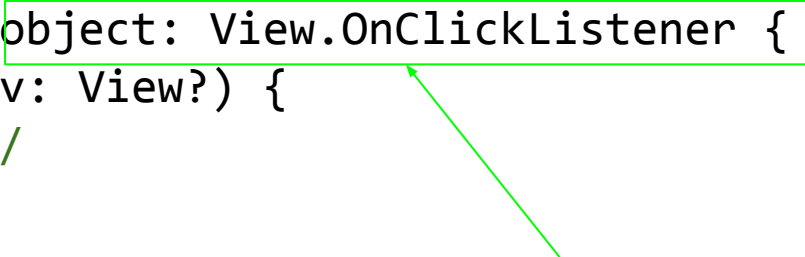
Esiste in Kotlin anche il concetto di object expression, che serve per istanziare una classe senza prima definirla (si parla anche di anonymous class instance).

Questo approccio è utile per evitare di dover definire classi molto semplici, che vengono istanziate comunque una sola volta.

```
val dayRates = object {  
    var standard: Int = 30 * standardDays  
    var festivity: Int = 50 * festivityDays  
    var special: Int = 100 * specialDays  
}
```

# View.OnClickListener: soluzione più sintetica

```
class MainActivity : AppCompatActivity(){  
    override fun onCreate(savedInstanceState: Bundle?) {  
        ...  
        val button: Button = findViewById(R.id.button)  
        button.setOnClickListener(object: View.OnClickListener {  
            override fun onClick(v: View?) {  
                /* do something */  
            }  
        })  
    }  
}
```



Una object expression, che crea una singola istanza di un oggetto (l'implementazione dell'interfaccia)

# SAM (single abstract method)

Converte una funzione nell'implementazione di un'interfaccia

**Formato:** `InterfaceName { lambda body }`

```
val runnable = Runnable { println("Hi there") }
```

È equivalente a:

```
val runnable = (object: Runnable {  
    override fun run() {  
        println("Hi there")  
    }  
})
```

# View.OnClickListener come SAM

Un modo più conciso di dichiarare un click listener

```
val button: Button = findViewById(R.id.button)
```

Se l'ultimo parametro è una funzione, può andare fuori dalle ():

```
button.setOnClickListener() {v -> /* do something */}
```

Se l'unico parametro è una funzione, posso omettere le ():

```
button.setOnClickListener { v-> /* do something */}
```

Se non si utilizza il parametro, allora si può omettere

```
button.setOnClickListener {/* do something */}
```

# Late initialization in Kotlin

```
class Student(val id: String) {  
  
    lateinit var records: HashSet<Any>  
  
    init {  
        // retrieve records given an id  
    }  
}
```

# Lateinit: esempio in Activity

```
class MainActivity : AppCompatActivity() {  
  
    lateinit var result: TextView  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        ...  
        result = findViewById(R.id.result_text_view)  
    }  
}
```



# Visualizzare liste con RecyclerView



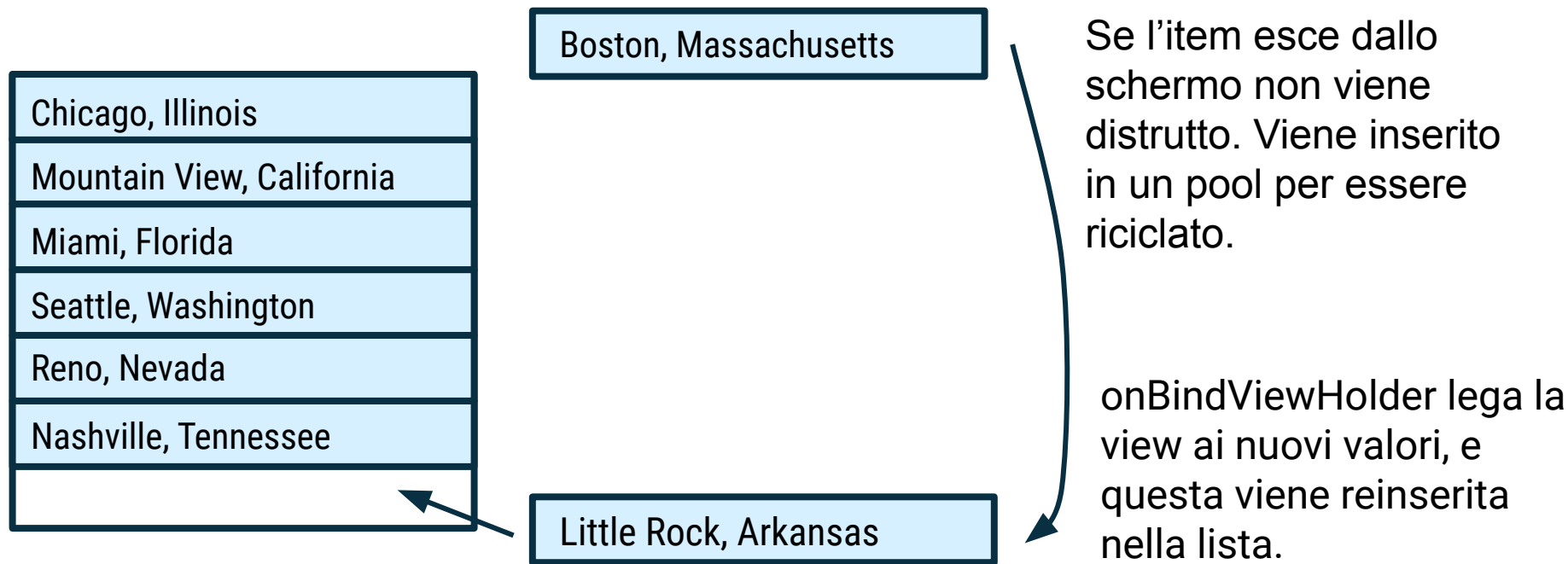
# RecyclerView

- Widget per visualizzare liste contenenti dati
- "Ricicla" (riutilizza) degli oggetti view per rendere lo scrolling più performante
- Si può specificare un layout per ogni singolo item
- Supporta animazioni e transizioni

# RecyclerView.Adapter

- Fornisce dati e layout che il RecyclerView mostra
- Un Adapter personalizzato estende `RecyclerView.Adapter` e sovrascrivere queste tre funzioni:
  - `getItemCount`
  - `onCreateViewHolder`
  - `onBindViewHolder`

# Riciclare view nel RecyclerView



# Aggiungere RecyclerView al layout

```
<androidx.recyclerview.widget.RecyclerView  
    android:id="@+id/rv"  
    android:scrollbars="vertical"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"/>
```

# Crea un layout per il list item

res/layout/item\_view.xml

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <TextView
        android:id="@+id/number"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</FrameLayout>
```

# Crea un list adapter

```
class MyAdapter(val data: List<Int>) : RecyclerView.Adapter<MyAdapter.MyViewHolder>() {  
    class MyViewHolder(val row: View) : RecyclerView.ViewHolder(row) {  
        val textView = row.findViewById<TextView>(R.id.number)  
    }  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {  
        val layout = LayoutInflater.from(parent.context).inflate(R.layout.item_view,  
            parent, false)  
        return MyViewHolder(layout)  
    }  
  
    override fun onBindViewHolder(holder: MyViewHolder, position: Int) {  
        holder.textView.text = data.get(position).toString()  
    }  
  
    override fun getItemCount(): Int = data.size  
}
```

# Configura l'adapter nel RecyclerView

In MainActivity.kt:

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    val rv: RecyclerView = findViewById(R.id.rv)  
    rv.layoutManager = LinearLayoutManager(this)  
  
    rv.adapter = MyAdapter(IntRange(0, 100).toList())  
}
```

# Summary



# Summary

In Lesson 5, you learned how to:

- Specify lengths in dp for your layout
- Work with screen densities for different Android devices
- Render Views to the screen of your app
- Layout views within a ConstraintLayout using constraints
- Simplify getting View references from layout with data binding

# Learn more

- [Pixel density on Android](#)
- [Spacing](#)
- [Device metrics](#)
- [Type scale](#)
- [Build a Responsive UI with ConstraintLayout](#)
- [Data Binding Library](#)
- [Create dynamic lists with RecyclerView](#)

# Pathway

Practice what you've learned by completing the pathway:

[Lesson 5: Layouts](#)

