Lezione 8

Programmazione Android

- Ancora sulla UI
  - Scrivere proprie View
  - Scrivere propri Layout
  - Stili e temi

Scrivere una propria View

Scrivere una propria View

- Gli elementi dell'interfaccia utente sono tutti sottoclassi di **View**
  - View, ViewGroup, Layout, Widget
  - Organizzati in un layout tramite file XML
- Per realizzare Widget custom, è sufficiente estendere View o una delle sue sottoclassi
  - **onMeasure()** – chiamato per negoziare le dimensioni
  - **onDraw()** – chiamato per disegnare l'effettiva UI
  - OnKeyDown(), onTouchEvent() & co. – chiamati per gestire l'input

Negoziare le dimensioni

- onMeasure(int widthSpec, int heightSpec)
  - I parametri passati sono i **requisiti** che il contenitore vuole investigare
  - onMeasure() **deve** fornire la risposta chiamando il metodo setMeasuredDimension(int width, int height)
    - In questo modo, si evita ogni allocazione di oggetti!
- Tipicamente, si chiama super.onMeasure() per lasciar fare alle superclasse, e poi si “aggiusta” il risultato in base alle proprie necessità
  - La superclasse chiamerà setMeasuredDimension()

Specifiche di dimensione

- Gli argomenti interi passati a onMeasure() codificano un **modo** e una **dimensione**
  - Nell'implementazione attuale, il modo è codificato dai due bit alti dell'intero, la dimensione dai rimanenti
  - Meglio però usare getMode() e getSize() di View.MeasureSpec per estrarre i valori
- Modo (costanti definite in View.MeasureSpec)
  - **UNSPECIFIED** – il contenitore non impone restrizioni
  - **EXACTLY** – il contenitore impone esattamente la dimensione data
  - **AT_MOST** – il contenitore impone un massimo

Specifiche di dimensione

- La nostra sottoclasse dovrà rispondere all'invocazione di onMeasure() specificando la sua dimensione “preferita”, nel rispetto dei vincoli
  - In qualche caso, possiamo anche ignorare i vincoli – in genere il contenitore effettuerà clipping
  - Di solito, la nostra view vorrà anche rispettare la dimensione minima indicata dai metodi
    - getSuggestedMinimumHeight() e
    - getSuggestedMinimumWidth()

Specifiche di dimensione

- L'implementazione di default (in View)

```
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec)
{
  setMeasuredDimension(
    getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec),
    getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec) );
}
```

- In pratica: per default è 100x100, ma se lo spazio è “elastico”, si allarga fino a occuparlo tutto

```
public static int getDefaultSize(int size, int measureSpec) {
    int result = size;
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize =  MeasureSpec.getSize(measureSpec);

    switch (specMode) {
    case MeasureSpec.UNSPECIFIED:
        result = size;
        break;
    case MeasureSpec.AT_MOST:
    case MeasureSpec.EXACTLY:
        result = specSize;
        break;
    }
    return result;
}
```

Disegnare il proprio aspetto

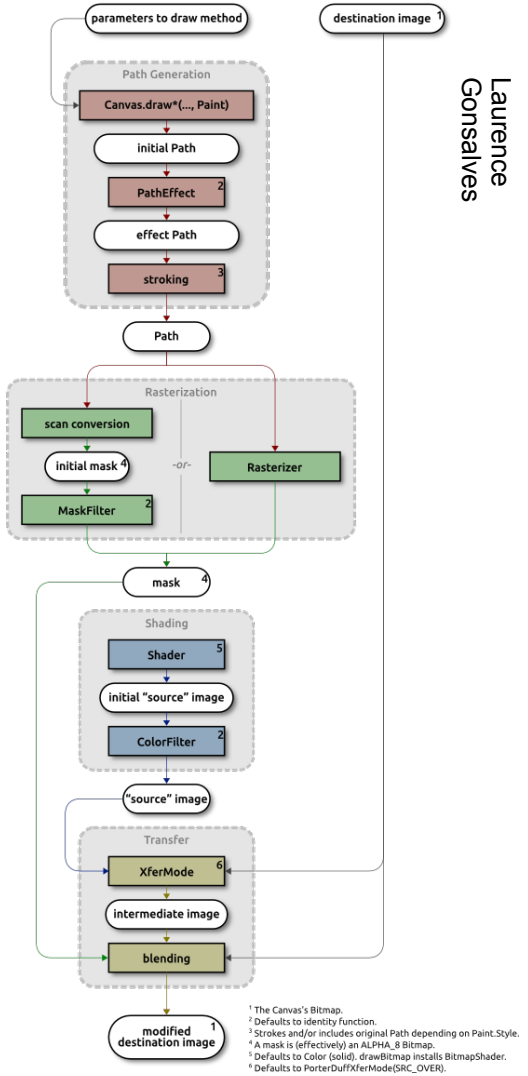
- Il metodo onDraw() viene invocato quando la view deve disegnarsi (nello spazio che è stato negoziato dalla onMeasure())
- A onDraw() viene passato un **Canvas**
  - Una superficie di disegno (eventualmente bufferizzata)
- Il nostro codice può disegnare sul canvas usando una o più **Paint**
  - "vernici" che specificano colore, trasparenza, tratteggi, font, ecc.

Canvas

- Un Canvas è una superficie di disegno virtuale
  - Non direttamente una bitmap!
- Supporta
  - Primitive di disegno varie
    - Geometriche
    - Immagini
    - Testo
  - Clipping
  - Matrici di trasformazione (rotazione, scalatura, ecc.)

La pipeline di rendering 2D

- Un Canvas è un contenitore per chiamate grafiche
- In realtà, compie molte altre operazioni prima che il disegno finisca su una Bitmap
  - **Rendering** di path, trasformazione secondo un PathEffect (arrotondamento degli angoli, tratteggio, ecc.)
  - **Rasterizzazione** (alpha-channel, anti-aliasing, filling, blurring, ecc.)
  - **Shading** (trasformazione di colori, gamma, ecc.)
  - **Trasferimento** (il disegno parziale viene combinato con i precedenti contenuti della Bitmap di destinazione, con vari operatori)



Metodi di Canvas

- **Clipping**
  - clipPath(), clipRect(), clipRegion()
- **Trasformazioni affini**
  - rotate(), scale(), translate(), skew(), concat(), setMatrix(), save(), restore()
- **Primitive di disegno**
  - drawBitmap/Picture()
  - drawArc/Circle/Oval()
  - drawColor/Paint/RGB/ARGB()
  - drawLine/Lines/Path/Vertices()
  - drawRect/RoundRect()
  - drawText/PosText/TextOnPath()
- **Informative**
  - getWidth/Height()
  - GetMaximumBitmapWidth/Height()
  - isHardwareAccelerated()
  - isOpaque()
  - getDensity()
  - getClipBounds(), getMatrix()

Varianti

- Di ogni metodo sono disponibili molte varianti, spesso con overloading e utilità varie
- Esempi:
  - void **drawRect**(float left, float top, float right, float bottom, Paint paint)
  - void **drawRect**(RectF rect, Paint paint)
  - void **drawRect**(Rect r, Paint paint)
  - void **drawRoundRect**(RectF rect, float rx, float ry, Paint paint)

Paint

- Un oggetto di classe Paint rappresenta una specifica completa del modo con cui una primitiva grafica (di Canvas) deve essere disegnata
- Proprietà più comuni
  - Colore, alpha
  - Dithering, antialiasing
  - Font, hinting, stili
  - Dimensione delle linee, tratteggio, cap, join
- Fornisce inoltre metodi per gestire la metrica del testo
  - Ascent, descent, misure in font proporzionali, ecc.

Costruttori della nostra view

- L'ultimo elemento mancante sono i **costruttori**
  - View(Context c)
    - Costruttore di base, associa la view al suo **context** (tramite il quale accedere, per esempio, alle risorse)
  - View(Context c, AttributeSet attr)
    - Costruttore chiamato quando la view viene creata a partire dalla specifica XML in un file di layout
    - Da attr si possono ottenere i valori degli **attributi**, tramite metodi getter come getAttribute<*tipo*>Value(String namespace, String attributo, <*tipo*> valoreDefault)
  - View(Context c, AttributeSet attr, int stile)
    - Costruttore che in aggiunta applica uno **stile** (identificato dal suo ID di risorsa)

Costruttori della nostra view

- L'ultimo elemento mancante sono i **costruttori**
  - View(Context c)
    - Costruttore di base, associa
      per esempio, alle risorse)
  - View(Context c, AttributeSe
    - Costruttore chiamato quand
      in un file di layout
    - Da attr si possono ottenere i valori degli **attributi**, tramite metodi getter come getAttribute<*tipo*>Value(String namespace, String attributo, <*tipo*> valoreDefault)
  - View(Context c, AttributeSet attr, int stile)
    - Costruttore che in aggiunta applica uno **stile** (identificato dal suo ID di risorsa)

Manca il costruttore vuoto View(): non si può avere una view senza il suo contesto

Esempio: CartaQuadretti

- Vogliamo realizzare una variante custom di EditText (il widget di sistema per l'editing di testi) che abbia come sfondo una carta a quadretti
- Dovremo:
  - Estendere EditText
  - Implementare i costruttori
  - Fare override di onDraw()
- E, nel contempo, cercare di far fare più lavoro possibile alla superclasse!

CartaQuadretti – intestazione

```
package it.unipi.di.sam.customviewtest;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.util.AttributeSet;
import android.widget.EditText;

public class CartaQuadretti extends EditText {

    int dimquad;
    Paint righini = new Paint();
```

Estendiamo la classe di sistema

Il tag da usare in layout.xml sarà <it.unipi.di.sam.customviewtest.CartaQuadretti>

Paint con cui disegneremo i righini

CartaQuadretti – costruttore

```
public CartaQuadretti(Context context) {
    super(context);
    init();
}

public CartaQuadretti(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
    init();
}

public CartaQuadretti(Context context, AttributeSet attrs) {
    super(context, attrs);
    init();
}
```

Un metodo privato che si occupa di inizializzare la CartaQuadretti. Potrebbe prendere context, attrs e defStyle come argomenti, ma noi lo teniamo semplice.

CartaQuadretti – init

```
private void init() {
    righini.setARGB(20, 0, 0, 140);
    dimquad = 20;
}
```

Qui impostiamo righini a un colore fisso (blu semi-trasparente).
Potremmo però rendere la nostra view configurabile, in vari modi:

- Potremmo definire un nome simbolico per il colore nelle risorse, farci passare il Context, e accedere al colore con

  ```
  Color c = context.getResources().getColor(R.color.righini);
  righini.setColor(c);
  ```

- Oppure, potremmo definire nuove risorse Stylable e includerle fra gli attributi del nostro tag XML (o anche eriditarle da un tema).

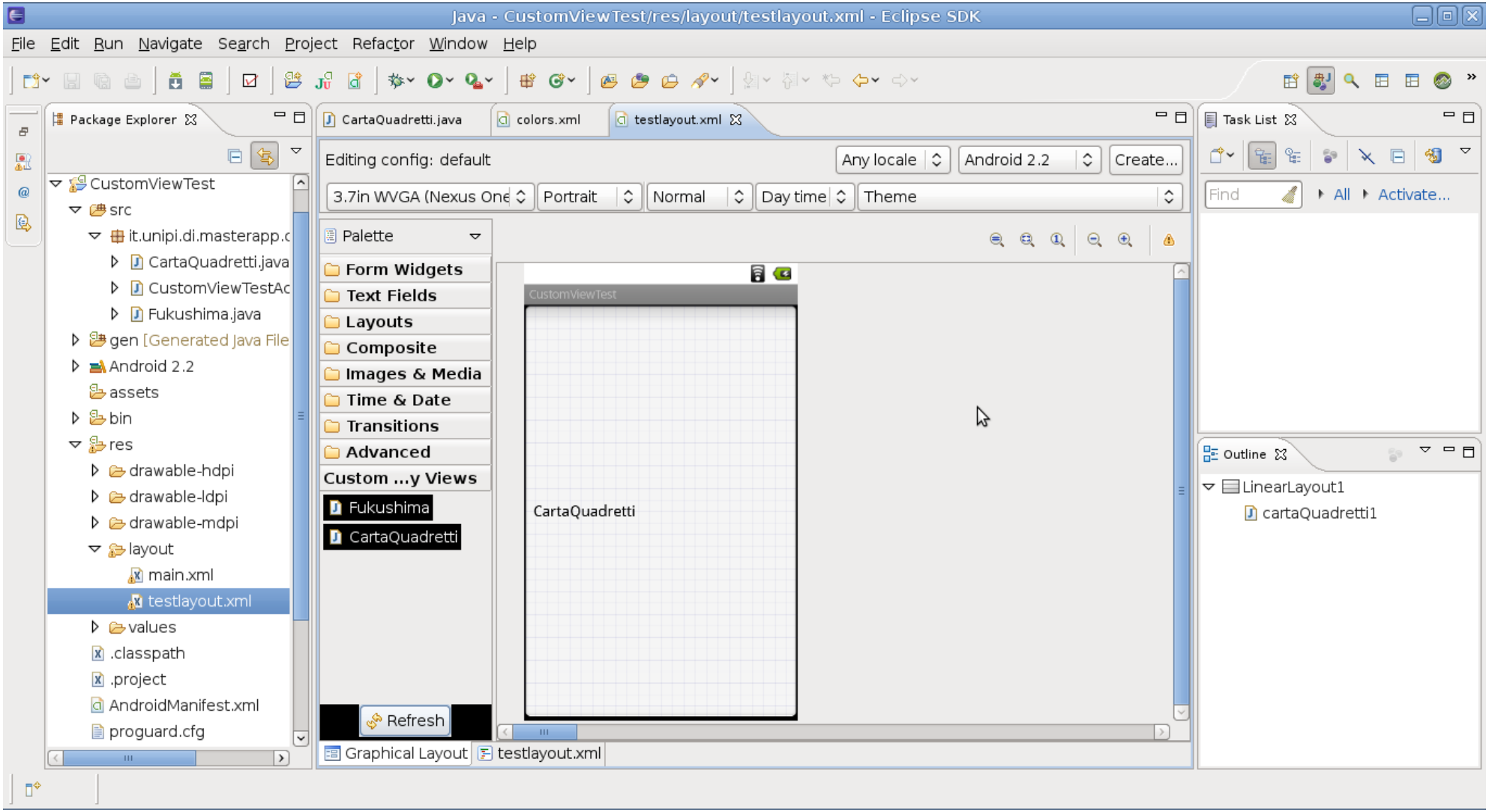
Stessa cosa per dimquad!

CartaQuadretti – disegno

```
protected void onDraw(Canvas canvas) {
    int w=getWidth(), h=getHeight();
    for (int x=0;x<w;x+=dimquad)
        canvas.drawLine(x, 0, x, h, righini);
    for (int y=0;y<h;y+=dimquad)
        canvas.drawLine(0, y, w, y, righini);
    super.onDraw(canvas);
}
```

Qui disegnamo tutto il resto (incluso il testo!)

CartaQuadretti – risultato



Ridisegno

- Una vista ha spesso delle proprietà
  - Grafica (es.: colore e dimensione dei quadretti)
  - Contenuto (es.: testo visualizzato)
- In questi casi, occorre un mezzo per ottenere il rinfresco della vista quando cambia la proprietà

```
public void setDimQuad(int n) {
    if (n>0) {
        dimquad=n;
        invalidate();
    }
}
```

Invalida la view: il sistema chiamerà la onDraw() per ridisegnare la CartaQuadretti con il nuovo dimquad.

Attributi custom in XML

- Molte custom view avranno attributi specifici
  - Come il colore e la dimensione dei quadretti nel nostro esempio
- Sempre una buona idea implementare getter/setter e consentire l'accesso da programma
- Può anche essere utile definire i valori per gli attributi nei file di layout (XML)
  - Occorre dichiarare prima quali sono gli attributi aggiunti
  - Risorse di tipo *stylable* dichiarate fra le risorse

Dichiarare stylable

- Si definiscono nodi <declare-stylable> dentro un container <resources>
  - Potrebbe essere ovunque, ma per convenzione si definiscono gli attributi custom in res/values/attrs.xml
- Ogni nodo <declare-stylable> definisce i nomi e i tipi degli attributi per una particolare custom view

```
<resources>
   <declare-styleable name="CartaQuadretti">
      <attr name="dimquad" format="integer"/>
      <attr name="righino" format="color"/>
   </declare-styleable>
</resources>
```

Riferire styleable

- Occorre **importare** le nostre nuove risorse styleable nel file di layout
  - Solitamente, si definisce un namespace per brevità

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  xmlns:cq="http://schemas.android.com/apk/res/it.unipi.di.sam.customviewtest"
  …
>
…
  <it.unipi.di.sam.customviewtest.CartaQuadretti
    cq:righino=”@color/red” cq:dimquad=”32” … />
…
</LinearLayout>
```

Recuperare i valori

- Quando il file XML di layout viene letto dal sistema, tutti gli attributi di ogni nodo vengono memorizzati in un **AttributeSet** che viene passato al costruttore di ciascuna View
  - Il costruttore può recuperare i valori degli attributi leggendo direttamente il parametro AttributeSet
  - **Però...**
    - I riferimenti a risorsa rimangono stringhe (“@color/red”), e non vengono dereferenziati automaticamente
    - Non vengono applicati automaticamente **stili** e **temi**
  - **Si usa quindi un metodo più articolato**

Recuperare i valori

```
public CartaQuadretti(Context context) {
    super(context);
    init(context,null,0);
}

public CartaQuadretti(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
    init(context,attrs,defStyle);
}

public CartaQuadretti(Context context, AttributeSet attrs) {
    super(context, attrs);
    init(context,attrs,0);
}
```

Passiamo i parametri a init()

Recuperare i valori

```
private init(Context context, AttributeSet attrs, int defStyle) {
    TypedArray a = context.getTheme().obtainStyledAttributes(
        attrs,
        R.styleable.CartaQuadretti,
        defStyle, 0);

    try {
        dimQuad = a.getInteger(R.styleable.CartaQuadretti_dimquad, 20);
        colrig = a.getColor(R.styleable.CartaQuadretti_righino,defcol);
        …
    } finally {
        a.recycle();
    }
}
```

Scrivere un proprio Layout

Ereditare da ViewGroup

- Per creare una **view composta**
  - In pratica, si costruisce un gruppo di widget pre-assemblato
  - Il costruttore della nostra view composta fa le **new** e le **.add()** necessarie per costruire tutto un sottoalbero
  - Poco interessante...
- Per creare un **layout custom**
  - Consente di disporre i figli secondo criteri personalizzati
  - Tecnica da usare quando i LayoutManager di libreria non soddisfano le necessità

Approfondiamo!

Layout custom

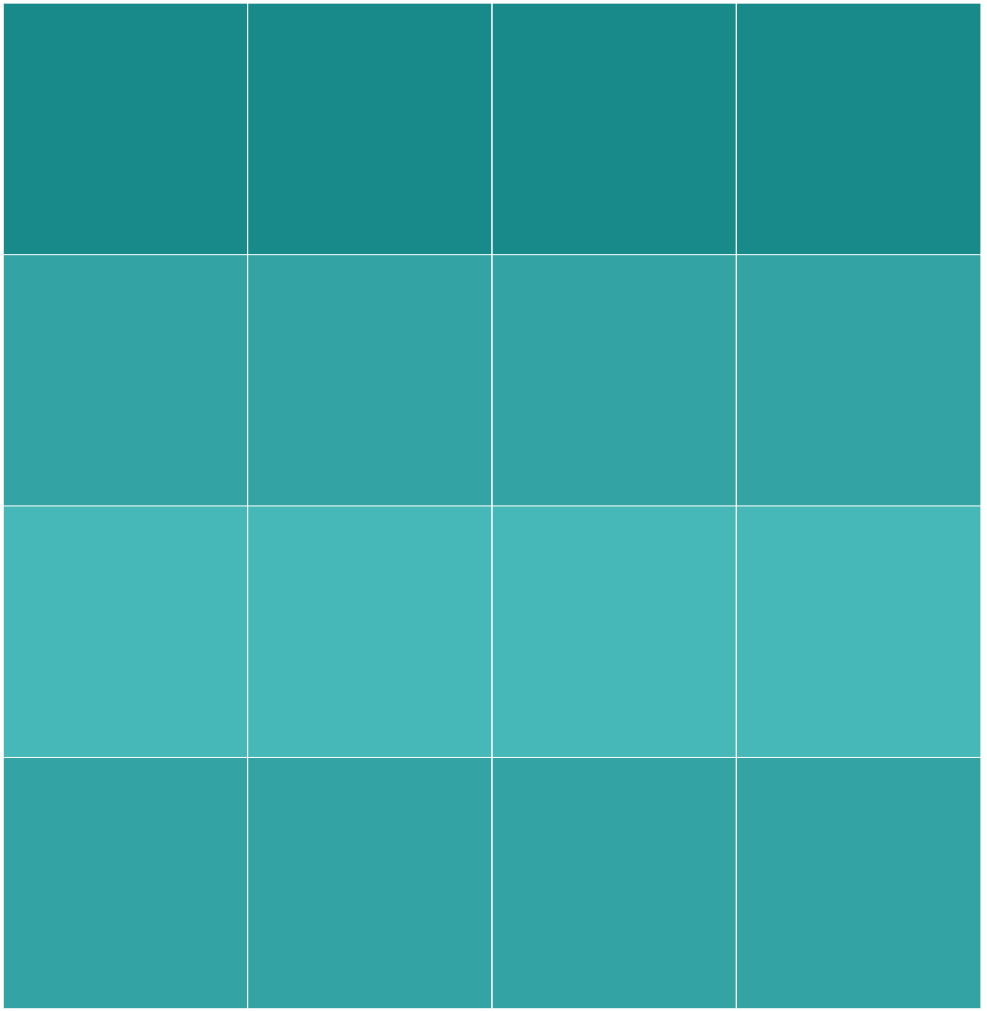
- Un layout deve fare override di un solo metodo:

  void onLayout(boolean changed,
  int left, int top, int right, int bottom)

- Molto spesso, dovrà anche fare override di onMeasure()
  - Perché le dimensioni del gruppo dipenderanno dalle dimensioni dei figli **e** dalla disposizione degli stessi

Esempio

- Vogliamo un layout che disponga i figli secondo una griglia equispaziata
  - Che so, per esempio per fare i tasti di una calcolatrice...
- Versione semplice
  - Non gestiamo View multi-cella
  - Calcoliamo noi il numero di righe e colonne in base al numero di figli



Esempio: EqLayout

```
package it.unipi.di.sam.eqlayout;

import android.content.Context;
import android.util.AttributeSet;
import android.view.View;
import android.view.ViewGroup;

public class EqLayout extends ViewGroup {

    public EqLayout(Context context) {
        super(context);
    }

    public EqLayout(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    public EqLayout(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
    }
```

Visto che un Layout è un ViewGroup, e ViewGroup è una sottoclasse di View, si applica tutto quanto abbiamo detto in precedenza sulle View custom:

- costruttori
- attributi custom e styleable

Esempio: EqLayout

```
@Override
protected void onLayout(boolean changed, int left, int top, int right, int bottom) {

    int lato = getLato();
    int w = (right-left)/lato;
    int h = (bottom-top)/lato;
    for (int i = 0; i < this.getChildCount(); i++) {
        View v = getChildAt(i);
        int x = i%lato, y = i/lato;
        v.layout(x*w, y*h, (x+1)*w, (y+1)*h);
    }
}

private int getLato() {
    return (int)Math.ceil(Math.sqrt(getChildCount()));
}
```

Quando il sistema grafico chiama il nostro **onLayout**(), ci sta chiedendo di disporre i figli in maniera tale che tutto il gruppo occupi lo spazio definito da left, top, right, bottom.

Noi semplicemente dividiamo il nostro spazio in un quadrato di celle di uguale dimensione, e disponiamo i figli chiamando su ciascuno il metodo **layout**() con le coordinate calcolate.

Esempio

- Dovremo anche implementare una onMeasure(), che dovrà passare la misurazione ai figli
  - Alcuni dei figli potrebbero essere a loro volta dei ViewGroup complessi, che hanno bisogno di stabilire le misure dei figli, ecc.
  - I figli necessitano di una onMeasure() anche per implementare correttamente la onDraw()
    - Per esempio, per centrare correttamente una label nello spazio allocato

Esempio: EqLayout

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
    int lato = getLato();
    int w = getMeasuredWidth()/lato;
    int h = getMeasuredHeight()/lato;
    int ws = MeasureSpec.makeMeasureSpec(w, MeasureSpec.EXACTLY);
    int hs = MeasureSpec.makeMeasureSpec(h, MeasureSpec.EXACTLY);
    for(int i = 0; i < this.getChildCount(); i++){
        View v = getChildAt(i);
        v.measure(ws,hs);
    }
}
```

Nel nostro esempio, informiamo i figli che vogliamo che ciascuno di loro sia *esattamente* alto e largo quanto la cella che lo conterrà – è ora di finirla con questo buonismo che conduce all'anarchia!

Esempio

- Il nostro esempio è molto basico, ma può essere facilmente esteso
- Una volta definito un custom layout, può essere usato come qualunque altro layout
  - Anche in questo caso, si può usare il nome fully-qualified come tag nel file XML di layout
  - In alternativa, si può istanziare il layout a programma con una **new**

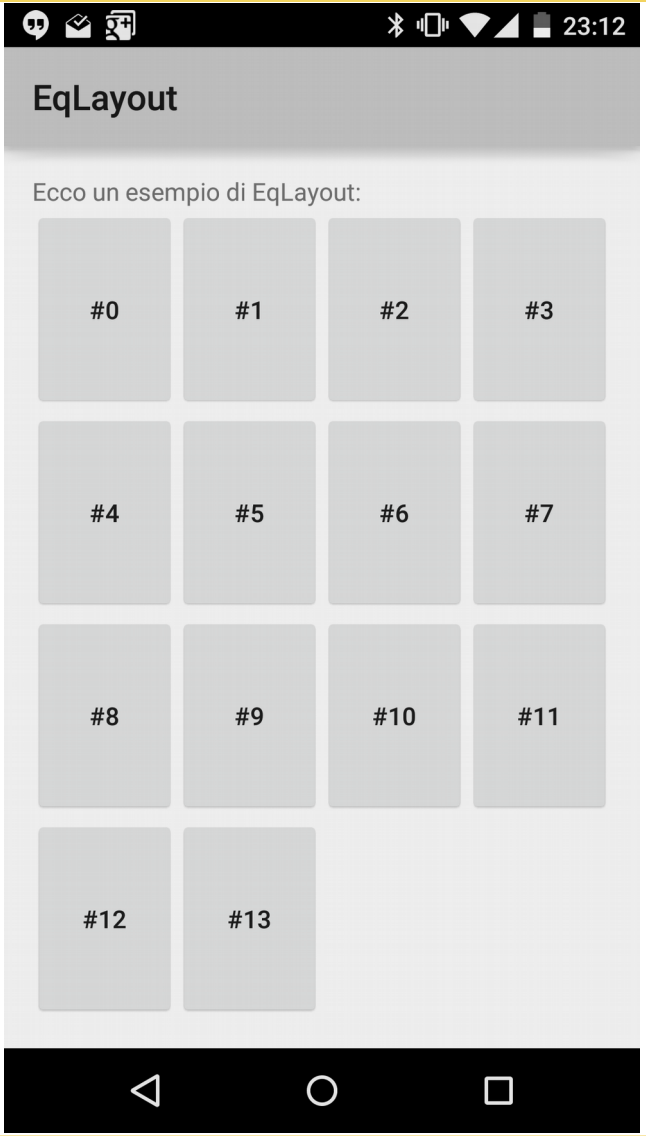
Esempio di uso di EqLayout

```
package it.unipi.di.sam.eqlayout;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Button;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        EqLayout eql = (EqLayout) findViewById(R.id.eqlayout);
        for (int i=0;i<14;i++) {
            Button b=new Button(this);
            b.setText("#"+i);
            eql.addView(b);
        }
    }
}
```



Stili e temi

Stili e temi

- Sia gli **stili** che i **temi** sono insiemi di coppie attributo-valore
  - Attributi e valori sono gli stessi che possono essere applicati alle viste in un file di layout
- Consentono di assegnare un nome a una configurazione, e poi applicarla riferendo il nome
- Differenza:
  - Uno **stile** si applica a una vista, e vale solo per quella vista
  - Un **tema** si applica a una vista, a un'Activity o a un'intera app, e vale per tutte le viste “figlie” 5.0+
  - Un **tema** definisce anche attributi che si applicano alle finestre e non alle viste
    - Es: Theme.NoTitleBar

Definire uno stile

- Esempio: /res/values/styles.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="bigbutt">
        <item name="android:textAppearance">?android:attr/textAppearanceLarge</item>
        <item name="android:textColor">@color/deepblue</item>
        <item name="android:padding">0dp</item>

    </style>

    <style name="pisan">
        <item name="android:rotation">3.97</item>
        <item name=”android:background”>@drawable/romanico</item>
    </style>
</resources>
```

Riferimento al valore di un attributo nel tema corrente

Riferimento a una risorsa

Valore letterale

Applicare uno stile

- Per applicare uno stile, è sufficiente dichiarare l'attributo *style* nel file di layout, riferendo lo stile definito
- In ogni caso, la vista applicherà solo gli attributi rilevanti dello stile
  - Altri attributi, non previsti dalla vista, vengono ignorati

```
...
  <Button    android:id="@+id/butt"
             style="@style/bigbutt"
             android:textColor="#888"
             android:text="BIG!"
  />
...
```

Applicare un tema

- Analogamente, per applicare un tema si usa l'attributo android:theme del tag <application> o <activity> in AndroidManifest.xml

```
...
  <application
    android:theme="@style/miotema"
        ...
  />
...
```

```
...
  <activity
    android:theme="@style/miotema"
        ...
  />
...
```

- Raramente si definisce un intero tema custom
  - Anche per consistenza visuale col sistema!

Eredità

- Gli stili di Android hanno qualche similarità con i CSS... ma sono realizzati assai peggio
  - Per esempio: niente selettori, si può applicare un solo stile a vista
  - Hanno una limitata forma di **ereditarietà**
    - Più debole del “cascading” dei CSS
- La combinazione di ereditarietà e selettori di risorse consente comunque di realizzare applicazioni che si “adattano” alle tante versioni di UI esistenti su Android
  - Per esempio: lo “skinning” dei diversi produttori è ottenuto modificando i temi di default del sistema

Eredità

- Ereditare da uno stile definito in altro package

```
<style name="blackpaua" parent="android:Theme.Light">
  <item name="android:textColor">#000</item>
</style>
```

- Ereditare da uno stile definito nello stesso package

```
<style name="bigbutt.black">
  <item name="android:textColor">#000</item>
</style>
```

Personalizzare gli stili

- La pratica più comune consiste nell'usare un tema di sistema, personalizzandone solo alcuni dettagli

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="MioTema" parent="android:Theme.Light">
        <item name="android:windowNoTitle">true</item>
        <item name="android:windowBackground">@color/rosso</item>
        <item name="android:listViewStyle">@style/MiaListView</item>
    </style>
    <style name="MiaListView" parent="@android:style/Widget.ListView">
        <item name="android:listSelector">@drawable/mlv_ls</item>
    </style>
</resources>
```

Quale tema scegliere?

- Android definisce *numerosissimi* stili e temi
  - **Stili**: https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/core/res/res/values/styles.xml
  - **Temi**: https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/core/res/res/values/themes.xml
- Scegliere quello “giusto” non è banale!

Qualche esempio

- Temi di default
  - **Theme** – tema base
    - **Theme.NoTitleBar**
      - **Theme.NoTitleBar.FullScreen**
      - **Theme.NoTitleBar.OverlayActionModes**
    - **Theme.WithActionBar**

Qualche esempio

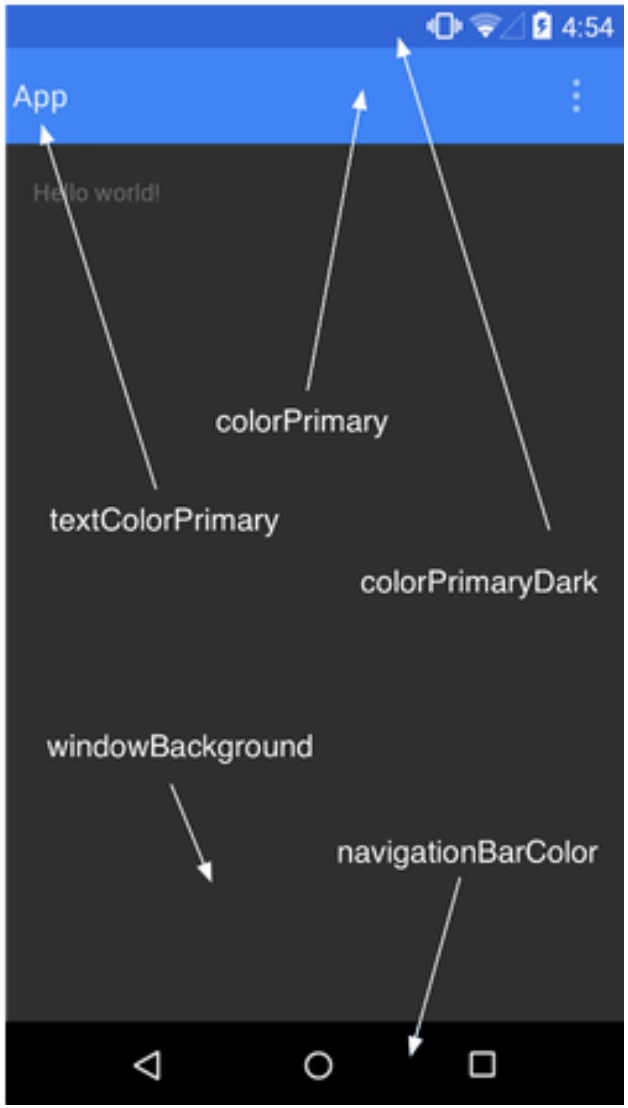
- Temi per API 10 e inferiori
  - **Theme.Light** – sfondo chiaro e testo nero
    - **Theme.Light.NoTitleBar**
      - **Theme.Light.NoTitleBar.FullScreen**
  - **Theme.Black** – sfondo nero e testo chiaro
    - **Theme.Black.NoTitleBar**
      - **Theme.Black.NoTitleBar.FullScreen**
  - **Theme.Wallpaper** – sfondo uguale alla home
    - **Theme.Wallpaper.NoTitleBar**
      - **Theme.Wallpaper.NoTitleBar.FullScreen**
  - **Theme.Translucent** – sfondo trasparente
    - ...

Qualche esempio

- Temi per API 11-20 (Holographic)
  - **Theme.Holo.Light**
  - **Theme.Holo.Dark**
- Temi per API 21 (Material Design)
  - **Theme.Material**
  - **Theme.Material.Light**

- Ovviamente, ciascuno con innumerevoli sottotemi

Stilare Material Design



- Con Android 5, lo stile “Material Design” raccomanda di usare colori pieni, personalizzati per la propria applicazione

```
<resources>
  <style name="AppTheme" parent="android:Theme.Material">
    <item name="android:colorPrimary">@color/azzurro</item>
    <item name="android:colorPrimaryDark">@color/blu</item>
    <item name="android:colorAccent">@color/giallo</item>
  </style>
</resources>
```

- Le *style guide* danno indicazioni precise sulla grafica