



View e Viewgroup

Programmazione Mobile

A.A. 2021/22

M.O. Spata

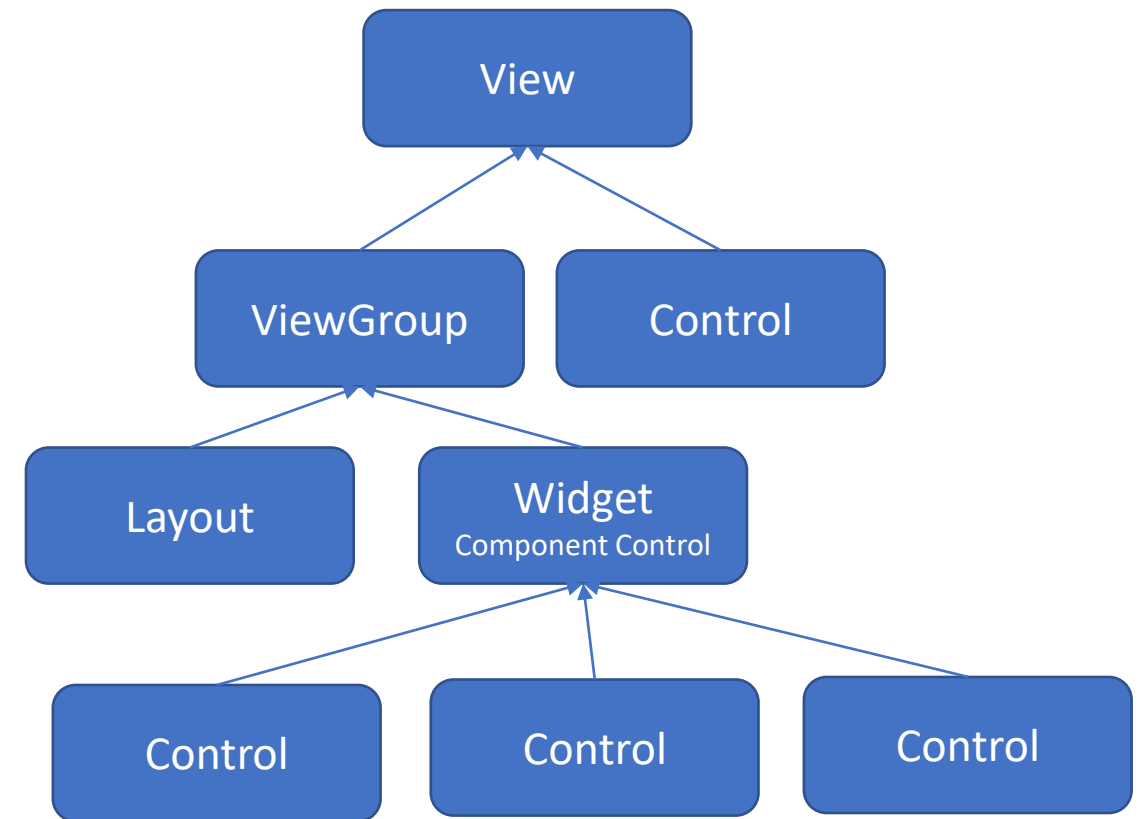


View e ViewGroup

- Nuova terminologia introdotta da Android:
 - **View**: Classi per la rappresentazione degli elementi visuali dell'interfaccia. Tutte le classi per i controlli ed i layout sono derivati dalla classe View.
 - **ViewGroups**: le classi ViewGroup sono estensioni della classe View che possono contenere più oggetti View "figli". Sono queste le classi da utilizzare per la creazione di controlli custom fatti da più oggetti view interconnessi o per gestire il layout di più view (Es. LinearLayour un Layout Manager).
 - **Activities**: sono state più volte definite; permettono di rappresentare e gestire le finestre o le schermate da presentare all'utente.

View e ViewGroup

- Android fornisce nativamente diversi controlli, widgets e layout manager; per la maggior parte delle applicazioni sarà opportuno estendere e modificare i controlli standard per introdurre e fornire nuove funzionalità.



Classi View di controllo

- **TextView**: si tratta della etichetta (label) standard. Ovvero etichetta di testo read-only; supporta il multilinea, la visualizzazione con formattazione, a capo automatico.
- **EditText**: casella di testo editabile. Supporta il multiline, il testo a capo, il testo di suggerimento (hint).
- **ListView**: si tratta di una ViewGroup che crea e gestisce una lista verticale di Views mostrandole come una riga della lista. La più semplice ListView mostra il valore stringa di ogni oggetto di un array attraverso una TextView per ognuno di essi.
- **Spinner**: un controllo composto che mostra una TextView con associata una ListView che permette di selezionare un elemento da una lista da mostrare in una TextView. Alla TextView è associato un Button che quando premuto mostra una dialog per la selezione.
- **Button**: un pulsante standard.
- **CheckBox**: un pulsante a due stati rappresentato da un box checked o unchecked

Layouts

- I layouts (layout managers) sono estensioni della classe ViewGroup usate per posizionare controlli (views) figli nella Interfaccia Utente. I layout possono essere arbitrariamente annidati il che permette di creare interfacce piuttosto complesse come combinazione di layout.
- Nativamente l'SDK di Android fornisce diversi layout. Sta allo sviluppatore cercare la "giusta" combinazione di layout per rendere l'interfaccia facile da capire e da utilizzare.

Tipi di Layouts

- **FrameLayout:** è il layout più semplice che semplicemente posiziona ogni view figlia nell'angolo in alto a sinistra. Con l'aggiunta di più view figlie verrà creata una pila di view (una sull'altra) dove ogni view oscura quella inserita in precedenza.
- **LinearLayout:** permette di allineare ogni view figlia in verticale o in orizzontale. Un layout verticale consisterà una colonna di Views, in maniera analoga un layout orizzontale consisterà di una riga di Views. Per tale tipo di Layout è possibile specificare per ogni View figlia la dimensione, ovvero lo spazio da occupare fra quello disponibile.
- **RelativeLayout:** permette di definire la posizione di ogni View relativamente alle altre Views e ai bordi del display.
- **TableLayout:** permette di disporre le Views su una griglia di righe e colonne. Sono ammessi span di righe e colonne.
- **Gallery:** permette di mostrare una singola riga di elementi in una lista scrollabile orizzontalmente.

Definizione di un Layout

- La via preferita per definire ed implementare layout è quella di utilizzare dei file di risorsa esterni XML. Un file XML che definisce un layout contiene un unico nodo Radice che definisce appunto il primo livello di layout e che può contenere al suo interno tutte le view e i layout annidati necessari per la realizzazione della schermata desiderata.



```
1<?xml version="1.0" encoding="utf-8"?>
2<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3    android:orientation="vertical"
4    android:layout_width="fill_parent"
5    android:layout_height="fill_parent"
6    >
7<TextView
8    android:layout_width="fill_parent"
9    android:layout_height="wrap_content"
10    android:text="@string/hello"
11    />
12</LinearLayout>
13
```

Implementazioni di Layout

- Come abbiamo già visto è possibile implementare Layout anche attraverso il codice:

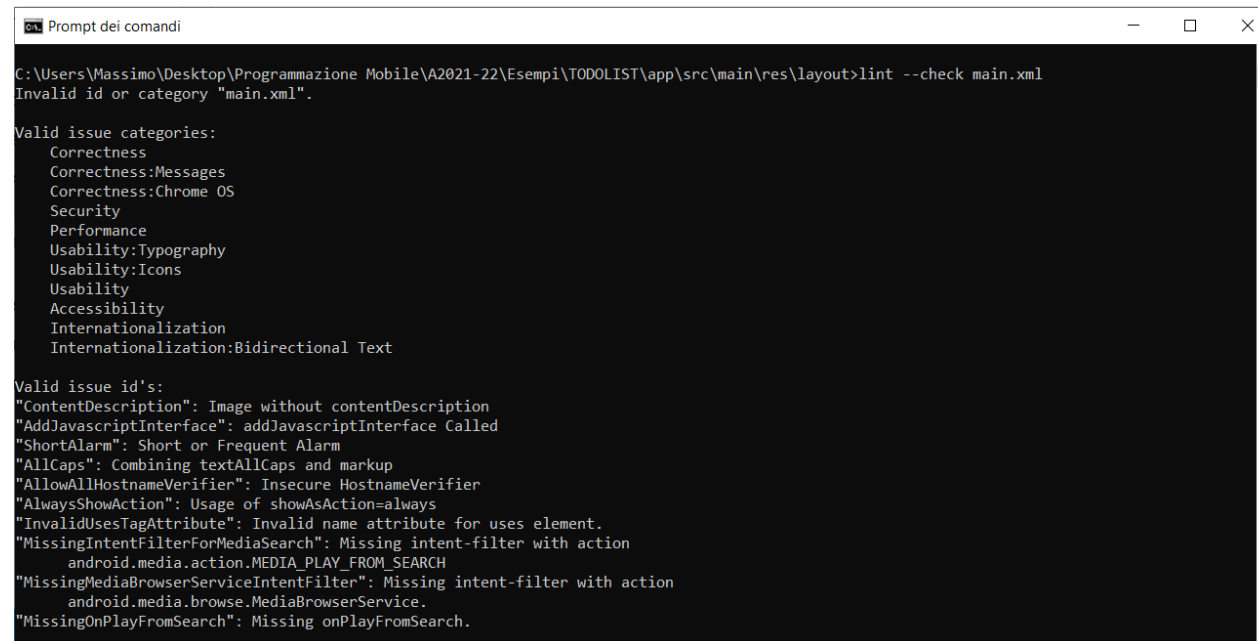
```
LinearLayout.LayoutParams lp;  
lp = new LinearLayout.LayoutParams(LayoutParams.FILL_PARENT,  
LayoutParams.FILL_PARENT);  
LinearLayout.LayoutParams textViewLP;  
textViewLP = new LinearLayout.LayoutParams(LayoutParams.FILL_PARENT,  
LayoutParams.WRAP_CONTENT);  
LinearLayout ll = new LinearLayout(this);  
ll.setOrientation(LinearLayout.VERTICAL);  
TextView myTextView = new TextView(this);  
myTextView.setText("Hello World, HelloWorld");  
ll.addView(myTextView, textViewLP);  
this.addView(ll, lp);
```


Alcune semplici regole per creare App veloci...

- La renderizzazione dei vari layouts all'interno di una applicazione è un'operazione computazionalmente costosa; la creazione e renderizzazione di layout e view annidate può avere effetti negativi sulla reattività dell'applicazione. Seguono alcuni consigli per creare layout efficienti:
 - Evitare annidamenti non necessari: Fare sempre il check sull'esistenza di Layout Ridondanti, specie se il layout definitivo è stato ottenuto attraverso modifiche significative di una versione precedente.
 - Non usare troppe View: Ogni view aggiuntiva in un layout richiede tempo di renderizzazione. Un layout dovrebbe includere al più 80 View.
 - Evitare annidamenti "profondi": limitare il livello di annidamento; anche se non ci sono limiti hardware è buona norma fermarsi a 10 livelli.

lint

- Esiste un tool denominato lint fornito con l'SDK che consente di analizzare il layout il cui nome di risorsa viene fornito in input e restituisce dopo un'opportuna analisi raccomandazioni per eventuali aggiustamenti e/o miglioramenti.
- Per usarlo basta aggiungere alla variabile di ambiente PATH il seguente percorso:
 - C:\Users\nome_utente\AppData\Local\Android\Sdk\tools\bin
- La sintassi è la seguente:
 - `lint [flags] <project directory>`



```
Prompt dei comandi

C:\Users\Massimo\Desktop\Programmazione Mobile\A2021-22\Esempi\TODOLIST\app\src\main\res\layout>lint --check main.xml
Invalid id or category "main.xml".

Valid issue categories:
  Correctness
  Correctness:Messages
  Correctness:Chrome_OS
  Security
  Performance
  Usability:Typography
  Usability:Icons
  Usability
  Accessibility
  Internationalization
  Internationalization:Bidirectional Text

Valid issue id's:
"ContentDescription": Image without contentDescription
"AddJavascriptInterface": addJavascriptInterface Called
"ShortAlarm": Short or Frequent Alarm
"AllCaps": Combining textAllCaps and markup
"AllowAllHostnameVerifier": Insecure HostnameVerifier
"AlwaysShowAction": Usage of showAsAction=always
"InvalidUsesTagAttribute": Invalid name attribute for uses element.
"MissingIntentFilterForMediaSearch": Missing intent-filter with action
    android.media.action.MEDIA_PLAY_FROM_SEARCH
"MissingMediaBrowserServiceIntentFilter": Missing intent-filter with action
    android.media.browse.MediaBrowserService.
"MissingOnPlayFromSearch": Missing onPlayFromSearch.
```

View Custom

- Android permette di creare nuove view attraverso il subclassing di view esistenti oppure permettendo l'implementazione da zero di nuovi controlli.
- Gli approcci possibili sono:
 - Modificare o Estendere l'aspetto e/o il comportamento di un controllo esistente se questo fornisce già la funzionalità di base che servono per l'applicazione. In particolare, ridefinendo l'event handlers **onDraw**, richiamando comunque lo stesso metodo della classe di base, è possibile customizzare la vista senza reimplementare le funzionalità di base.
 - Combinare Views per creare controlli monolitici e riutilizzabili che sfruttano le funzionalità di diverse viste connesse.
 - Creare interamente un nuovo controllo quando serve una interfaccia completamente nuova e differente dalle altre nell'apparenze e nel comportamento

Compromesso tra estetica ed usabilità

- Quando si progetta una interfaccia utente è importante che vi sia un buon compromesso fra estetica ed usabilità.
- La possibilità di creare nuovi controlli customizzati potrebbe indurre alla tentazione di creare tutti i controlli from scratch.
- Tuttavia ci sono diversi buoni motivi per resistere:
 - Le View standard saranno sicuramente familiari agli utenti di altre applicazioni Android, e saranno automaticamente aggiornate con la distribuzione di nuove release della piattaforma.

Modifica di View esistenti

- Come mostreremo nell'esempio successivo, per creare una nuova View a partire da un controllo esistente bisogna creare una nuova classe che lo estende.
- Per modificare l'aspetto e/o il comportamento del controllo ridefinire le funzioni di gestione eventi (event handlers) associate con le caratteristiche da modificare

Esempio

- L'Esempio seguente mostra come ridefinire il metodo onDraw per modificare l'aspetto del controllo e la funzione onKeyDown per permette di customizzare l'azione associata alla pressione di un tasto:

```
// ** Extending Views ***** //
```

```
public class MyTextView extends TextView {  
  
    public MyTextView (Context context, AttributeSet ats, int defStyle) {  
        super(context, ats, defStyle);  
    }  
  
    public MyTextView (Context context) {  
        super(context);  
    }  
  
    public MyTextView (Context context, AttributeSet attrs) {  
        super(context, attrs);  
    }  
  
    @Override  
    public void onDraw(Canvas canvas) {  
        [ ... Draw things on the canvas under the text ... ]  
  
        // Render the text as usual using the TextView base class.  
        super.onDraw(canvas);  
  
        [ ... Draw things on the canvas over the text ... ]  
    }  
  
    @Override  
    public boolean onKeyDown(int keyCode, KeyEvent keyEvent) {  
        [ ... Perform some special processing ... ]  
        [ ... based on a particular key press ... ]  
  
        // Use the existing functionality implemented by  
        // the base class to respond to a key press event.  
        return super.onKeyDown(keyCode, keyEvent);  
    }  
}
```

Esercizio

- Creare una nuova Classe `TodoListItemView` che estende `TextView`.
Includere un blocco per l'overriding del metodo `onDraw`, ed implementare un costruttore che chiama un nuovo metodo `init`.

1.

```
package com.paad.todolist;
import android.content.Context;
import android.content.res.Resources;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.util.AttributeSet;
import android.widget.TextView;

public class TodoListItemView extends TextView {
    public TodoListItemView (Context context, AttributeSet ats, int ds) {
        super(context, ats, ds);
        init();
    }
    public TodoListItemView (Context context) {
        super(context);
        init();
    }
    public TodoListItemView (Context context, AttributeSet attrs) {
        super(context, attrs);
        init();
    }
    private void init () {
    }
    @Override
    public void onDraw (Canvas canvas) {
        // Use the base TextView to render the text.
        super.onDraw(canvas);
    }
}
```


2.

Creare un nuovo file di risorsa colors.xml dentro la cartella res/values. Creare nuovi valori di colore, margini, linee, e colori di testo.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="notepad_paper">#FFFFFF99</color>
    <color name="notepad_lines">#FF0000FF</color>
    <color name="notepad_margin">#90FF0000</color>
    <color name="notepad_text" >#AA0000FF</color>
</resources>
```

3.

Creare un nuovo file di risorse dimension.xml ed aggiungi un nuovo valore per il margin ed il width

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<resources>
```

```
    <dimen name= "notepad_margin" >30dp</dimen>
```

```
</resources>
```

4.

Con le risorse definite, siamo pronti a customizzare la grafica del `TodoListItemView`. Creiamo una nuova variabile istanza per conservare l'oggetto `Paint` che useremo per impostare il background ed il margin. Inoltre creiamo le variabili per il valore del colore dei margini e della larghezza. Con il metodo `init` settiamone i valori per prendere le istanze delle risorse create nei precedenti due passi, e creiamo l'oggetto `Paint`.

```
private Paint marginPaint;
private Paint linePaint;
private int paperColor;
private float margin;
private void init() {
    // Get a reference to our resource table.
    Resources myResources = getResources();
    // Create the paint brushes we will use in the onDraw method.
    marginPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    marginPaint.setColor(myResources.getColor(R.color.notepad_margin));
    linePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    linePaint.setColor(myResources.getColor(R.color.notepad_lines));
    // Get the paper background color and the margin width.
    paperColor = myResources.getColor(R.color.notepad_paper);
    margin = myResources.getDimension(R.dimen.notepad_margin);
}
```

5.

To draw the paper, override `onDraw` and draw the image using the `Paint` objects you created in Step 4. Once you've drawn the paper image, call the superclass's `onDraw` method and let it draw the text as usual.

```
@Override
public void onDraw(Canvas canvas) {
    // Color as paper canvas.
    drawColor(paperColor);
    // Draw ruled lines canvas.
    drawLine(0, 0, getMeasuredHeight(), (linePaint));
    canvas.drawLine(0, getMeasuredHeight(), getMeasuredWidth(),
        getMeasuredHeight(), linePaint());
    // Draw margin
    canvas.drawLine(margin, 0, margin, getMeasuredHeight(), marginPaint);
    //Move the text across from the margin
    canvas.save();
    canvas.translate(margin, 0);
    // Use the TextView to render the text.
    super.onDraw(canvas);
    canvas.restore();
}
```

6.

Ciò completa l'implementazione di `TodoListItemView`. Per utilizzarlo nell'attività dell'elenco di cose da fare è necessario includerlo in un nuovo layout e passare quel layout al costruttore dell'adattatore array. Inizia creando una nuova risorsa `todolist_item.xml` nella cartella `res/layout`. Specifica come viene visualizzato ciascuno degli elementi dell'elenco della to do list. Per questo esempio il tuo layout deve essere composto solo dalla nuova `TodoListItemView`, impostata per riempire l'intera area disponibile.

```
<?xml version= "1.0" encoding="utf-8"?>
<com.paad.todolist.TODOListItemView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height=" fill_parent"
    android:padding="10dp"
    android:scrollbars="vertical"
    android:textColor="@color/notepad_text"
    android:fadingEdge="vertical"
/>
```

7.

Ora apri la classe Attività ToDoList. Il passaggio finale consiste nel modificare i parametri passati a ArrayAdapter in onCreate. Sostituisci il riferimento ad android.R.layout.simple_list_item_1 con un riferimento al nuovo layout R.layout.todolist_item creato nel passaggio 6.

```
final ArrayList<String> todoItems = new ArrayList<String>();  
int resID = R.layout.todolist_item;  
final ArrayAdapter<String> aa = new ArrayAdapter<String>(this, resID, todoItem);  
myListView.setAdapter(aa);
```

Controlli Composti

- Quando si crea un controllo composto si deve definire il layout, l'aspetto e l'interazione fra le view che esso contiene.
- Per farlo si può estendere una classe ViewGroup (di solito un Layout) che viene scelta in base alla disposizione che i controlli dovrebbero avere nella configurazione finale.

Esempio

- Il seguente blocco di codice mostra la definizione di un file XML che permette di creare un controllo che contiene al suo interno una casella di testo (EditText) ed un semplice Pulsante (Button)

```
public class MyCompoundView extends LinearLayout {  
    public MyCompoundView(Context context) {  
        super(context);  
    }  
  
    public MyCompoundView(Context context, AttributeSet attrs) {  
        super(context, attrs);  
    }  
}  
  
<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical"  
    android:layout_width="fill_parent"  
    android:layout_height="fill_parent">  
    <EditText  
        android:id="@+id/editText"  
        android:layout_width="fill_parent"  
        android:layout_height="wrap_content"  
    />  
    <Button  
        android:id="@+id/clearButton"  
        android:layout_width="fill_parent"  
        android:layout_height="wrap_content"  
        android:text="Clear"  
    />  
</LinearLayout>
```


Esempio

- Per renderizzare questo controllo bisogna creare una classe come estensione della viewgroup scelta, in questo caso LinearLayout quindi ridefinire opportunamente il costruttore ed utilizzare l'oggetto LayoutInflater ed il metodo Inflate come mostrato di seguito:

```
public class ClearableEditText extends LinearLayout {  
  
    EditText editText;  
    Button clearButton;  
  
    public ClearableEditText(Context context) {  
        super(context);  
  
        // Inflate the view from the layout resource.  
        String infService = Context.LAYOUT_INFLATER_SERVICE;  
        LayoutInflater li;  
        li = (LayoutInflater)getContext().getSystemService(infService);  
        li.inflate(R.layout.clearable_edit_text, this, true);  
  
        // Get references to the child controls.  
        editText = (EditText)findViewById(R.id.editText);  
        clearButton = (Button)findViewById(R.id.clearButton);  
  
        // Hook up the functionality  
        hookupButton();  
    }  
}
```

Esempio

- Lo stesso controllo può essere definito senza l'ausilio del file XML costruendo il layout direttamente da codice come di seguito mostrato:

```
public ClearableEditText(Context context) {  
    super(context);  
  
    // Set orientation of layout to vertical  
    setOrientation(LinearLayout.VERTICAL);  
  
    // Create the child controls.  
    editText = new EditText(getContext());  
    clearButton = new Button(getContext());  
    clearButton.setText("Clear");  
  
    // Lay them out in the compound control.  
    int lHeight = LayoutParams.WRAP_CONTENT;  
    int lWidth = LayoutParams.FILL_PARENT;  
  
    addView(editText, new LinearLayout.LayoutParams(lWidth, lHeight));  
    addView(clearButton, new LinearLayout.LayoutParams(lWidth, lHeight));  
  
    // Hook up the functionality  
    hookupButton();  
}  
  
private void hookupButton() {  
    clearButton.setOnClickListener(new Button.OnClickListener() {  
        public void onClick(View v) {  
            editText.setText("");  
        }  
    });  
}
```

View custom

- La creazione di controlli e Views nuove permette di definire e personalizzare al massimo il look and feel e la modalità di interazione di un'applicazione.
- Per creare un nuovo controllo è necessario estendere una delle classi *View* o *SurfaceView*.
 - La classe *View* fornisce un oggetto *Canvas* (con una serie di metodi draw) ed una classe *Paint*. Tali entità possono essere utilizzati per creare l'interfaccia visuale (grafica 2D). Ovviamente è possibile ridefinire le funzioni di gestione evento per fornire l'interattività.
 - La classe *SurfaceView* fornisce un oggetto *Surface* e la possibilità di disegnare oggetti attraverso un thread in background e che fa uso di *OpenGL* per la grafica 3D.

View custom

- Per creare una nuova interfaccia utente 2D estendiamo la classe View: quest'ultima (myView) fornisce un quadrato di dimensione 100x100 pixel.
- Per cambiare la dimensione del controllo e mostrare una interfaccia un po' più complessa è necessario ridefinire i metodi onMeasure e onDraw.
 - Con onMeasure la nuova View calcolerà l'altezza e la larghezza che impiegherà rispetto ad un insieme di condizioni limite definite.
 - Il metodo onDraw permetterà di disegnare sul Canvas.

Esempio

- Segue la ridefinizione di View per una nuova ipotetica MyView

```
public class MyView extends View {
    // Constructor required for in-code creation
    public MyView(Context context) {
        super(context);
    }
    // Constructor required for inflation from resource file
    public MyView (Context context, AttributeSet ats, int defStyleAttr) {
        super(context, ats, defStyleAttr );
    }
    //Constructor required for inflation from resource file
    public MyView (Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    @Override
    protected void onMeasure(int wMeasureSpec, int hMeasureSpec) {
        int measuredHeight = measureHeight(hMeasureSpec);
        int measuredWidth = measureWidth(wMeasureSpec);
        // MUST make this call to setMeasuredDimension
        // or you will cause a runtime exception when the control is laid out.
        setMeasuredDimension(measuredHeight, measuredWidth);
    }
    private int measureHeight(int measureSpec) {
        int specMode = MeasureSpec.getMode(measureSpec);
        int specSize = MeasureSpec.getSize(measureSpec);

        [ ... Calculate the view height ... ]
        return specSize;
    }
    private int measureWidth(int measureSpec) {
        int specMode = MeasureSpec.getMode(measureSpec);
        int specSize = MeasureSpec.getSize(measureSpec);

        [ ... Calculate the view width ... ]
        return specSize;
    }
    @Override
    protected void onDraw(Canvas canvas) {
        [ ... Draw your visual interface ... ]
    }
}
```