



# Lezione 11



# Programmazione Android



- Ancora sulla UI (ma poi basta!)
  - WebView – una vista tuttofare
  - Drawable
  - Notifiche all'utente
    - Toast
    - Notifications
    - Dialog
  - Fragment
    - DialogFragment



# Dialoghi



- Un **Dialog** è parte dell'interfaccia utente di una Activity, e sempre collegato al suo contesto
- Android mette a disposizione alcuni tipi di dialoghi già pronti:
  - **AlertDialog** – informazioni e richieste di decisione
  - **ProgressDialog** – stato di avanzamento di un task
  - **DatePicker, TimePicker** – input di date e orari
- È sempre possibile creare dialoghi custom
  - Come al solito, basta definirsi un layout
  - È utile ereditare lo stile di sistema per i dialoghi!

Deprecato  
da API 26



# Interazione tra Dialog e Activity



- Mentre un toast “galleggia” su un'activity e non interferisce con essa, e una notifica su status bar è completamente fuori da ogni activity, un Dialog interagisce con la sua Activity
  - Il Dialog è in effetti implementato come una sotto-activity, in cui gran parte dello schermo è trasparente
  - Però è sempre parte della nostra UI: per esempio, il tasto menu attiva il menu che era attivo per l'activity che ha lanciato il dialogo



# Interazione tra Dialog e Activity

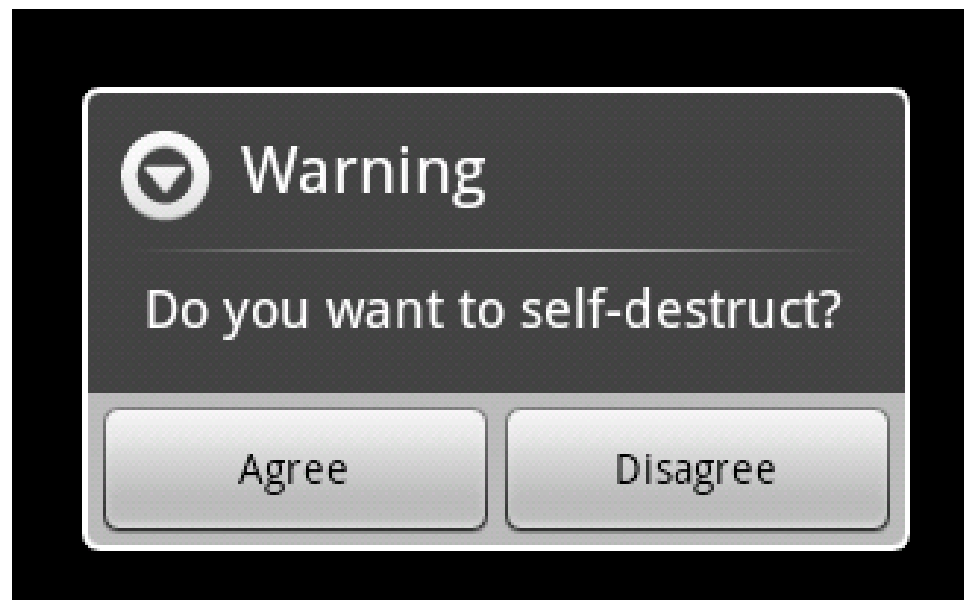


- L'activity chiede al sistema di aprire un suo dialogo (passando un int) con **showDialog(id)**
- Se è la prima volta che viene richiesto il particolare dialogo, il sistema chiama **onCreateDialog(id)** dell'activity, che costruisce e restituisce un Dialog
  - Il sistema “conserva” il Dialog restituito per usi futuri
- Prima di aprire il dialog, il sistema chiama **onPrepareDialog(id, dialog)** dell'Activity
  - Qui si possono cambiare i contenuti del dialog
- Come al solito, lo scopo è evitare le **new!**

# Creazione di un Dialog



- Il tipo più semplice (e più generale) è l'**AlertDialog**
- Comprende, opzionalmente:
  - Un titolo
  - Un'icona
  - Un testo
  - Da uno a tre pulsanti
    - positive, negative, neutral
  - Una lista di item



# Creazione di un Dialog

- La creazione effettiva viene svolta da un **DialogBuilder**

```
protected Dialog onCreateDialog(int id) {  
    switch (id) {  
        case DIALOG_YES_NO_MESSAGE:  
            return new AlertDialog.Builder(this)  
                .setIcon(R.drawable.alert_dialog_icon)  
                .setTitle(R.string.alert_dialog_two_buttons_title)  
                .setPositiveButton(R.string.alert_dialog_ok, new DialogInterface.OnClickListener() {  
                    public void onClick(DialogInterface dialog, int whichButton) {  
                        /* cose da fare se l'utente ha cliccato OK */  
                    }  
                })  
                .setNegativeButton(R.string.alert_dialog_cancel, new  
DialogInterface.OnClickListener() {  
                    public void onClick(DialogInterface dialog, int whichButton) {  
                        /* cose da fare se l'utente ha cliccato Cancel */  
                    }  
                })  
                .create();  
    }  
}
```

Method  
chaining

Il Context

Method  
chaining



# Creazione di un Dialog



- I metodi del Builder:
  - **setCancelable()** - se l'utente può chiudere il Dialog con ←
  - **setIcon(), setMessage(), setTitle()** - contenuti del messaggio
  - **setPositiveButton(), setNegativeButton(), setNeutralButton()** - label dei pulsanti
  - **setAdapter(), setCursor(), setItems(), setSingleChoiceItems(), setMultiChoiceItems()** - imposta una lista di scelte
    - Analoghi a quello che abbiamo visto per le ListView
  - **setOnCancelListener(), setOnItemSelectedListener(), setOnKeyListener()** - listener per i vari eventi
  - **create() / show()** - crea / crea e visualizza immediatamente il Dialog





# Implementazione del Builder



```
public Builder(Context context, int theme) {  
    P = new AlertController.AlertParams(new ContextThemeWrapper(  
        context, resolveDialogTheme(context, theme)));  
    mTheme = theme;  
}
```

```
public Builder setIcon(int iconId) {  
    P.mIconId = iconId;  
    return this;  
}
```

Da *android.app.AlertDialog.Builder*  
versione 5.1.1 r1

```
public Builder setPositiveButton(int textId, final OnClickListener listener) {  
    P.mPositiveButtonText = P.mContext.getText(textId);  
    P.mPositiveButtonListener = listener;  
    return this;  
}
```



# Implementazione del Builder



```
public AlertDialog create() {  
    final AlertDialog dialog = new AlertDialog(P.mContext, mTheme, false);  
    P.apply(dialog.mAlert);  
    dialog.setCancelable(P.mCancelable);  
    if (P.mCancelable) {  
        dialog.setCanceledOnTouchOutside(true);  
    }  
    dialog.setOnCancelListener(P.mOnCancelListener);  
    dialog.setOnDismissListener(P.mOnDismissListener);  
    if (P.mOnKeyListener != null) {  
        dialog.setOnKeyListener(P.mOnKeyListener);  
    }  
    return dialog;  
}
```

```
public AlertDialog show() {  
    final AlertDialog dialog = create();  
    dialog.show();  
    return dialog;  
}
```

# Dialog custom

- Il principio generale è simile, ma creare il layout del Dialog in `onCreateDialog()` diventa responsabilità del programmatore
  - Non usiamo `AlertDialog.Builder`
  - Si istanzia direttamente un `Dialog` e si imposta il suo layout con `setContentView()`
  - **In alternativa**, si possono aggiungere “pezzi” custom a un `AlertDialog` con `setView()`
- Da Android 3.0+ è consigliato usare un **DialogFragment** (che vediamo fra breve)



# Dialog custom

**case** *DIALOG\_TEXT\_ENTRY*:

```
LayoutInflater factory = LayoutInflater.from(this);
```

```
final View textEntryView = factory.inflate(R.layout.alert_dialog_text_entry, null);
```

```
return new AlertDialog.Builder(AlertDialogSamples.this)
```

```
    .setIcon(R.drawable.alert_dialog_icon)
```

```
    .setTitle(R.string.alert_dialog_text_entry)
```

```
    .setView(textEntryView)
```

```
    .setPositiveButton(R.string.alert_dialog_ok, new DialogInterface.OnClickListener() {
```

```
        public void onClick(DialogInterface dialog, int whichButton) {
```

```
            /* cose da fare se l'utente ha cliccato OK */
```

```
        }
```

```
    })
```

```
    .setNegativeButton(R.string.alert_dialog_cancel, new DialogInterface.OnClickListener() {
```

```
        public void onClick(DialogInterface dialog, int whichButton) {
```

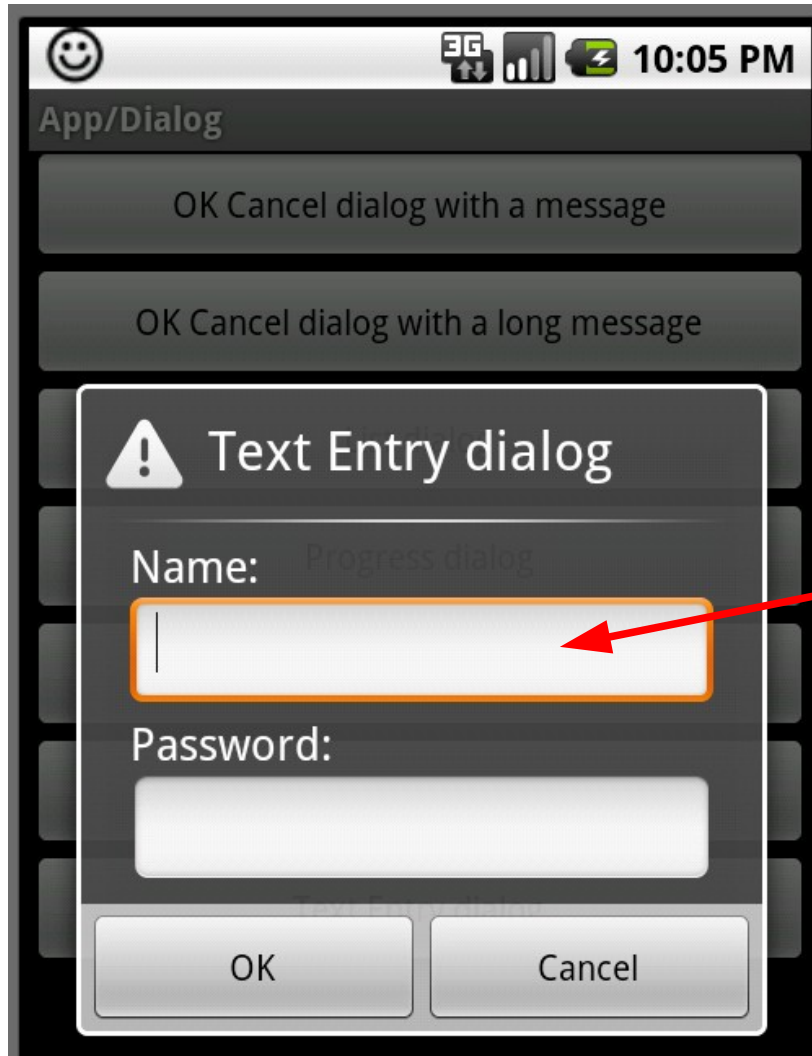
```
            /* cose da fare se l'utente ha cliccato Cancel */
```

```
        }
```

```
    })
```

```
    .create();
```

# Dialog custom



```

<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:orientation="vertical">

<TextView
android:id="@+id/username_view"
android:layout_height="wrap_content"
android:layout_width="wrap_content"
android:layout_marginLeft="20dip"
android:layout_marginRight="20dip"
android:text="@string/alert_dialog_username"
android:gravity="left"
android:textAppearance="?android:attr/textAppearanceMedium" />

<EditText
android:id="@+id/username_edit"
android:layout_height="wrap_content"
android:layout_width="fill_parent"
android:layout_marginLeft="20dip"
android:layout_marginRight="20dip"
android:scrollHorizontally="true"
android:autoText="false"
android:capitalize="none"
android:gravity="fill_horizontal"
android:textAppearance="?android:attr/textAppearanceMedium" />

<TextView
android:id="@+id/password_view"
android:layout_height="wrap_content"
android:layout_width="wrap_content"
android:layout_marginLeft="20dip"
android:layout_marginRight="20dip"
android:text="@string/alert_dialog_password"
android:gravity="left"
android:textAppearance="?android:attr/textAppearanceMedium" />

<EditText
android:id="@+id/password_edit"
android:layout_height="wrap_content"
android:layout_width="fill_parent"
android:layout_marginLeft="20dip"
android:layout_marginRight="20dip"
android:scrollHorizontally="true"
android:autoText="false"
android:capitalize="none"
android:gravity="fill_horizontal"
android:password="true"
android:textAppearance="?android:attr/textAppearanceMedium" />

</LinearLayout>

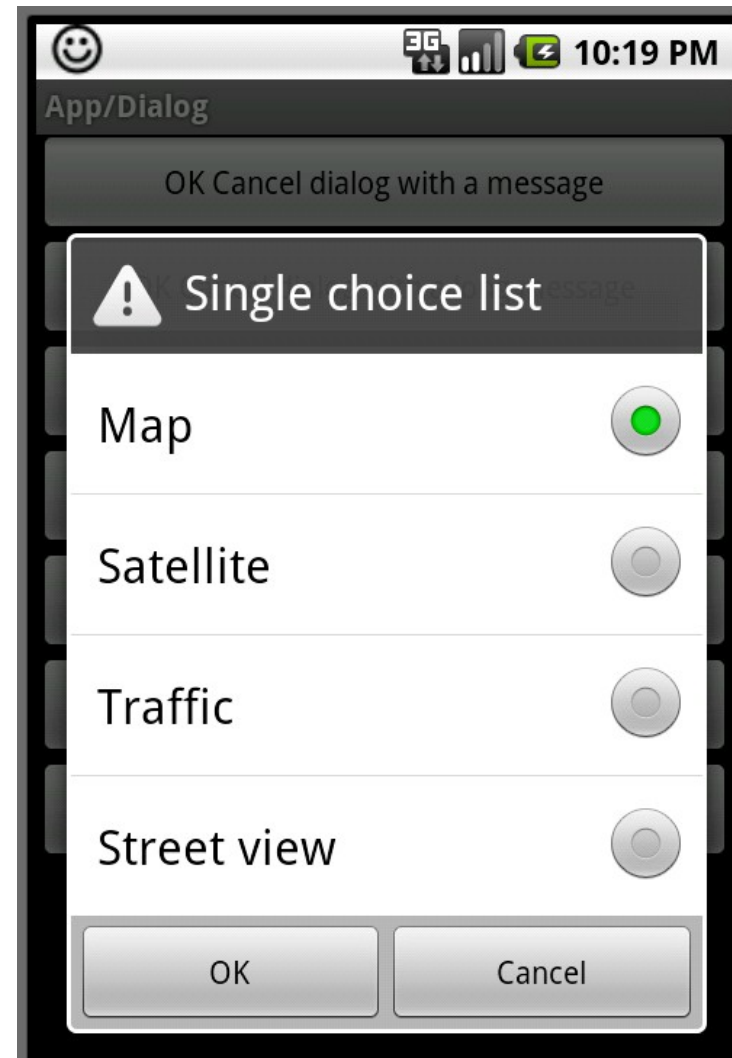
```

# Dialog single-choice



**case** `DIALOG_SINGLE_CHOICE`:

```
return new AlertDialog.Builder(AlertDialogSamples.this)
    .setIcon(R.drawable.alert_dialog_icon)
    .setTitle(R.string.alert_dialog_single_choice)
    .setSingleChoiceItems(R.array.select_dialog_items2, 0,
        new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
                /* cosa fare se l'utente ha selezionato un item */
            }
        })
    .setPositiveButton(R.string.alert_dialog_ok,
        new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
                /* cosa fare se l'utente preme OK */
            }
        })
    .setNegativeButton(R.string.alert_dialog_cancel,
        new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int whichButton) {
                /* cosa fare se l'utente preme Cancel */
            }
        })
    .create();
```

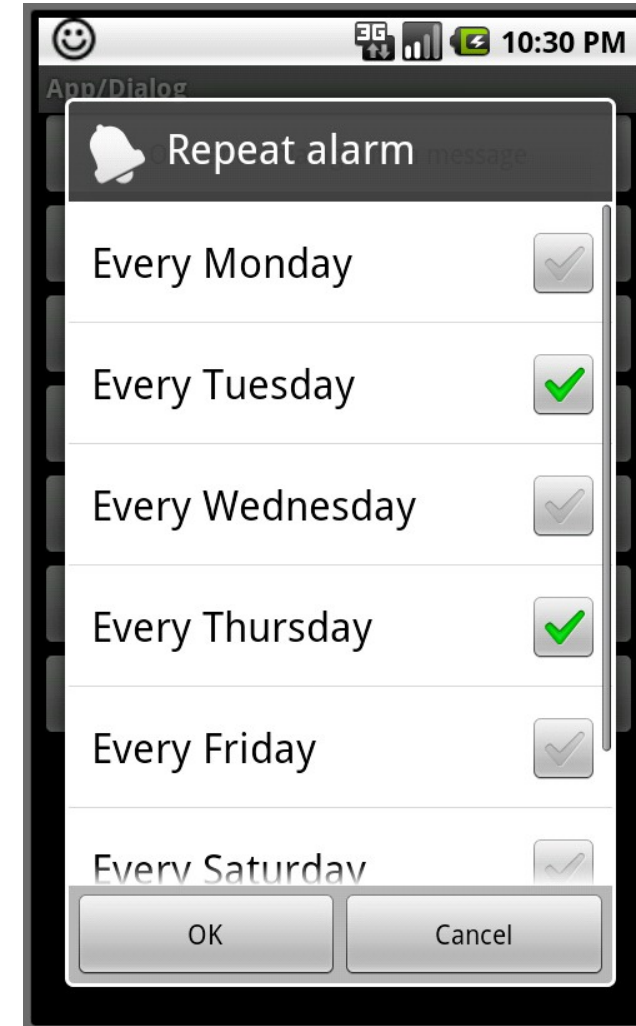


# Dialog multi-choice



**case** `DIALOG_MULTIPLE_CHOICE`:

```
return new AlertDialog.Builder(AlertDialogSamples.this)
    .setIcon(R.drawable.ic_popup_reminder)
    .setTitle(R.string.alert_dialog_multi_choice)
    .setMultiChoiceItems(R.array.select_dialog_items3,
        new boolean[]{false, true, false, true, false, false, false},
        new DialogInterface.OnMultiChoiceClickListener() {
            public void onClick(DialogInterface dialog,
                int whichButton, boolean isChecked) {
                /* cosa fare se l'utente seleziona un item */
            }
        })
    .setPositiveButton(R.string.alert_dialog_ok,
        new DialogInterface.OnClickListener() {
            public void onClick( ... ) { ... }
        })
    .setNegativeButton(R.string.alert_dialog_cancel,
        new DialogInterface.OnClickListener() {
            public void onClick( ... ) { ... }
        })
    .create();
```





# Dialog Progress (sconsigliato Lollipop+)



- Si usa la classe ProgressDialog
- Simile all'AlertDialog, ma in più:
  - `setProgressStyle()`
    - istogramma orizzontale o spinner circolare
  - `incrementProgressBy()`, `incrementSecondaryProgressBy()`
  - `setProgress()`, `setSecondaryProgress()`, `setIndeterminate()`
- Ha senso in genere solo in presenza di più thread
  - **MAI** svolgere compiti lunghi nel thread dell'UI!





# Dialog DatePicker e TimePicker



- Per fortuna, queste sono facili
  - Perché poco configurabili!

@Override

```
protected Dialog onCreateDialog(int id) {  
    switch (id) {  
        case TIME_DIALOG_ID:  
            return new TimePickerDialog(this, mTimeSetListener, mHour, mMinute, false);  
  
        case DATE_DIALOG_ID:  
            return new DatePickerDialog(this, mDateSetListener, mYear, mMonth, mDay);  
    }  
    return null;  
}
```

24 ore o  
AM/PM

Si può invocare `updateTime()` o `updateDate()` nell'`onPrepareDialog()` per assicurarsi che i dialog siano inizializzati con la data o ora corrente (o desiderata).



# Dialog

## DatePicker e TimePicker



- I listener vengono chiamati quando l'utente conferma la data/ora
  - void onDateSet(DatePicker view, int anno, int mese, int giorno)
  - void onTimeSet(TimePicker view, int ora, int minuto)
- Nota: il TimePickerDialog non consente di scegliere i secondi!

DatePickerDialog usa il widget DatePicker  
TimePickerDialog usa il widget TimePicker

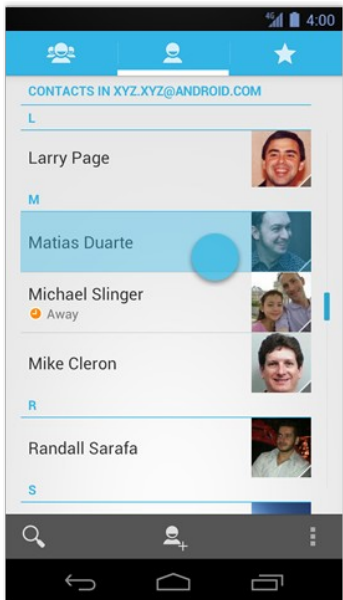
Non è detto che usare un dialog separato sia sempre la cosa migliore... si possono usare i widget direttamente nella activity



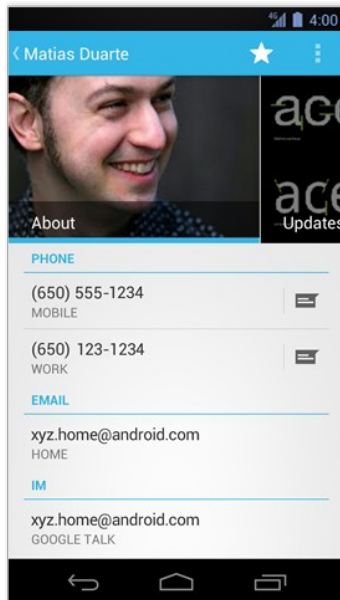
# I Fragment

# I Fragment

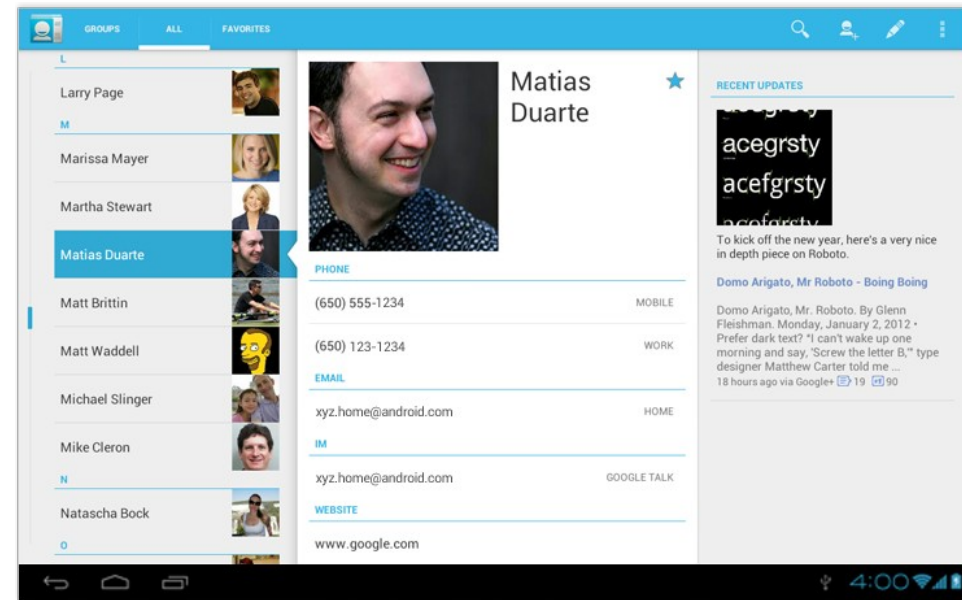
- I Fragment sono porzioni **riciclabili** di UI 3.0+
- Possono essere composti in vari modi
  - Specialmente per adattarsi a schermi di varie dimensioni
  - In particolare, possono essere aggiunti alle activity



List view



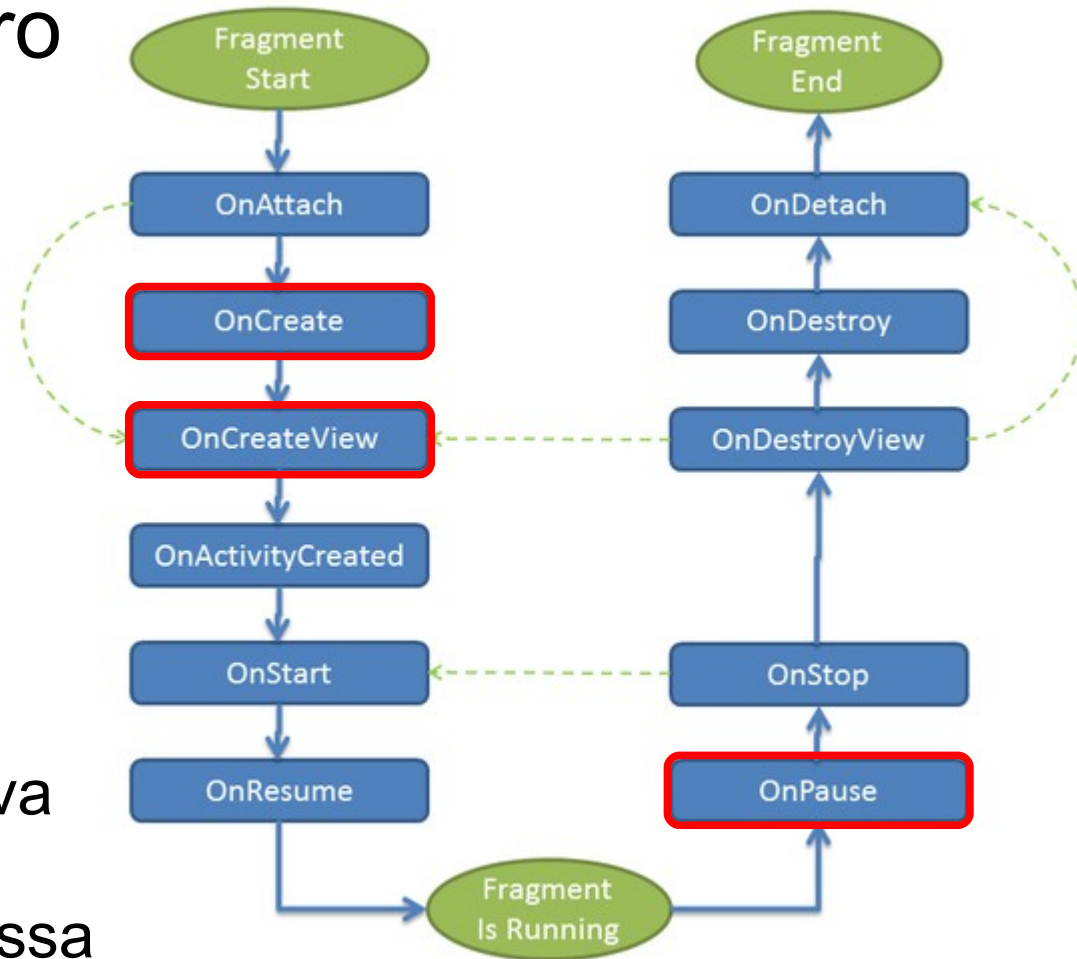
Detail view



# Ciclo di vita



- I Fragment hanno un loro ciclo di vita
  - Distinto da quello delle Activity
    - Possono essere inseriti o rimossi da una Activity in qualunque momento
  - Ma integrato con esso
    - Quando l'Activity nasce, va in pausa, o muore, i suoi Fragment seguono la stessa sorte



da <http://docs.xamarin.com/>

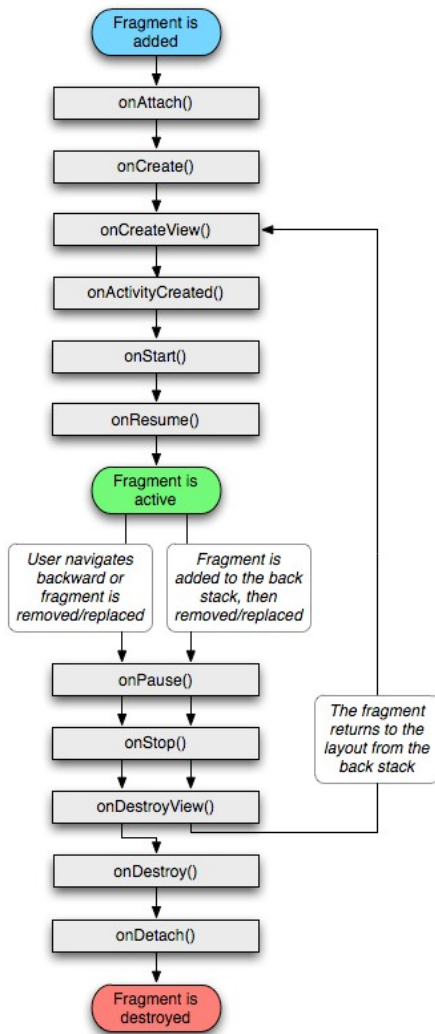


# I metodi del ciclo di vita



Lifecycle Method	Activity State
<b>onInflate()</b>	<b>Called when the Fragment is being created as part of a view layout.</b> This may be called immediately after the Fragment is created declaratively from an XML layout file. The Fragment is not associated with its Activity yet, but the Activity, Bundle, and AttributeSet from the view hierarchy are passed in as parameters. This method is best used for parsing the AttributeSet and for saving the attributes that might be used later by the Fragment.
<b>onAttach()</b>	<b>Called after the Fragment is associated with the Activity.</b> This is the first method to be run when the Fragment is ready to be used. In general, Fragments should not implement a constructor or override the default constructor. Any components that are required for the Fragment should be initialized in this method.
<b>onCreate()</b>	<b>Called by the Activity to create the Fragment.</b> When this method is called, the view hierarchy of the hosting Activity may not be completely instantiated, so the Fragment should not rely on any parts of the Activity's view hierarchy until later on in the Fragment's lifecycle. For example, do not use this method to perform any tweaks or adjustments to the UI of the application. This is the earliest time at which the Fragment may begin gathering the data that it needs. The Fragment is running in the UI thread at this point, so avoid any lengthy processing, or perform that processing on a background thread. This method may be skipped if <code>SetRetainInstance(true)</code> is called. This alternative will be described in more detail below.
<b>onCreateView()</b>	<b>Creates the view for the Fragment.</b> This method is called once the Activity's <code>onCreate()</code> method is complete. At this point, it is safe to interact with the view hierarchy of the Activity. This method should return the view that will be used by the Fragment.
<b>onActivityCreated()</b>	<b>Called after Activity.onCreate has been completed by the hosting Activity.</b> Final tweaks to the user interface should be performed at this time.
<b>onStart()</b>	<b>Called after the containing Activity has been resumed.</b> This makes the Fragment visible to the user. In many cases, the Fragment will contain code that would otherwise be in the <code>onStart()</code> method of an Activity.
<b>onResume()</b>	<b>This is the last method called before the user can interact with the Fragment.</b> An example of the kind of code that should be performed in this method would be enabling features of a device that the user may interact with, such as the camera that the location services. Services such as these can cause excessive battery drain, though, and an application should minimize their use in order to preserve battery life.

# I metodi del ciclo di vita



- I metodi “distruttori” sono, ovviamente, duali
- I Fragment hanno anche un loro salvataggio dello stato
  - Del tutto analogo a quello che conosciamo
    - onSaveInstanceState(), bundle ri-passato alla creazione ecc.
- Possono anche aggiungere propri item al menu dell'activity a cui sono “attached”



# Creare Fragment in XML



- I Fragment vengono inseriti in un layout XML con il tag `<fragment>`:

```
<fragment  
  class=".TestFragment"  
  android:id="@+id/titles"  
  android:layout_weight="1"  
  android:layout_width="0px"  
  android:layout_height="match_parent"  
>
```

- O possono essere manipolati dinamicamente
  - Tramite la classe **FragmentManager**





# Sottoclassi di Fragment



- La struttura dei Fragment ricalca le Activity
  - Soltanto, sono un “frame” e non a tutto schermo
  - Create le vostre sottoclassi di Fragment
- Fra l'altro, sono fornite sotto-classi specializzate
  - DialogFragment → implementare Dialog con Fragment
  - ListFragment → analogo a ListActivity, (Cursor ecc.)
  - PreferenceFragment → analogo a PreferenceActivity
  - WebViewFragment → frammento per WebView
  - MapFragment → mappe di Google Maps

# Creare una sottoclasse

```
public class ArticleFragment extends Fragment {  
    @Override  
    public View onCreateView(LayoutInflater inflater, ViewGroup container,  
        Bundle savedInstanceState) {  
        return inflater.inflate(R.layout.article_view, container, false);  
    }  
}
```

- L'unico metodo che è necessario sovrascrivere è **onCreateView**
- Deve restituire la View corrispondente al Fragment
  - Verrà visualizzata come parte dell'activity
  - In un **container** (ViewGroup) nel layout dell'activity

Restituisce la View  
ottenuta dal layout  
XML indicato



# Esempio




Un pattern tipico:  
Master/Detail

Create New Project

Customize the Activity

Creates a new master/detail flow, allowing users to view a collection of objects as well as details for each object. This flow is presented using two columns on tablet-size screens and one column on handsets and smaller screens. This template creates two activities, a master fragment, and a detail fragment.



Object Kind:

Object Kind Plural:

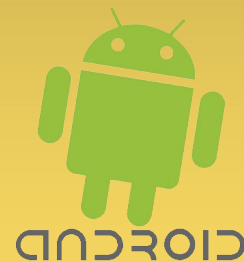
Title:

Master/Detail Flow

Other examples are 'Person', 'Book', etc.

Previous Next Cancel Finish

# Esempio



- Ho un TitlesFragment che estende ListFragment
  - Estrae i “titoli” da un Cursor, mostra la ListView, reagisce al click
- Un booleano, mDualPane, indica il modo
  - “telefono/portrait” (activity separate per titolo e dettagli)
  - “tablet/landscape” (una activity con due fragment)
- Ho un DetailsFragment che estende Fragment
  - Ha un suo layout
    - Che qui non ci interessa

# Esempio



- In TitlesFragment, intercetto l'onClick sulla lista

```
@Override  
public void onItemClick(ListView l, View v, int pos, long id)  
{  
    showDetails(pos);  
}
```

- La showDetails() dovrà distinguere i due casi
  - Se ho entrambi i Fragment, devo evidenziare la scelta del titolo e aggiornare il Fragment dei dettagli
  - Altrimenti, devo far partire l'Activity dei dettagli



# Esempio



```
void showDetails(int index) {
    if (mDualPane) {
        getListView().setItemChecked(index, true);
        DetailsFragment details = (DetailsFragment)
            getFragmentManager().findFragmentById(R.id.details);
        if (details == null || details.getShownIndex() != index) {
            details = DetailsFragment.newInstance(index);
            FragmentTransaction ft = getFragmentManager().beginTransaction();
            if (index == 0) { ft.replace(R.id.details, details); }
            else { ft.replace(R.id.a_item, details); }
            ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
            ft.commit();
        }
    } else {
        Intent intent = new Intent();
        intent.setClass(getActivity(), DetailsActivity.class);
        intent.putExtra("index", index);
        startActivity(intent);
    }
}
```



# Stato dei frammenti

- Va ricordato che abbiamo a che fare con **istanze** di Fragment – ciascuna col suo stato
- Un frammento può (in linea di principio) essere “attaccato” in Activity diverse, in momenti diversi
- Per accedere alla Activity corrente, possiamo usare `getActivity()`
- **Importante** distinguere stato del frammento e stato dell'activity!

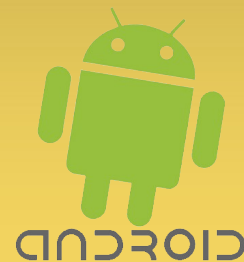
# Transazioni e Transizioni



- Quando si aggiungono e rimuovono Fragment dinamicamente a una Activity, è necessario usare le **transazioni**
  - Garantiscono che le operazioni vengano fatte in modo atomico
    - es.: non si dispatchano eventi a Fragment in corso di rimozione
- Quando si aggiungono e rimuovono Fragment dinamicamente a una Activity, è carino usare le **transizioni**
  - Si occupano di rendere esteticamente gradevole l'operazione di sostituzione
    - es.: cross-fade o slide di un nuovo pannello



# Transazioni e transizioni



- Le transazioni si occupano anche di inserire le variazioni di UI ottenute rimpiazzando i frammenti nello **stack del task**
  - Premendo back, l'utente torna alla configurazione precedente dei fragment all'interno della stessa Activity, invece di andare all'Activity precedente
- Più operazioni sui fragment possono essere effettuate con la stessa transazione
  - Es: rimpiazzare due fragment con contenuti correlati in una sola operazione

# Transazioni e transizioni



```
FragmentManager fm = getFragmentManager()
```

```
FragmentTransaction trans = fm.beginTransaction();
```

```
ArticleFragment art = new ArticleFragment(...);
```

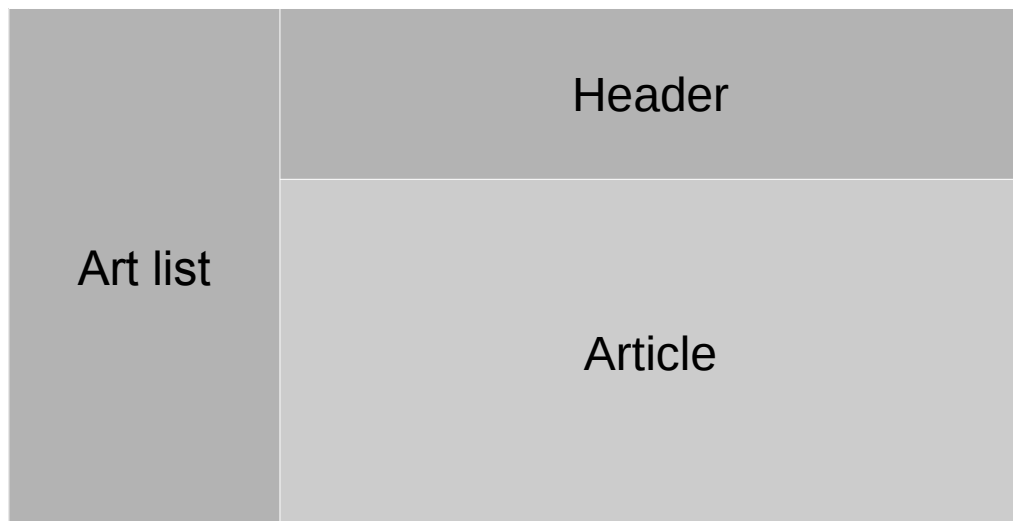
```
HeaderFragment head = new HeaderFragment(...);
```

```
trans.add(R.id.artcontainer, art);
```

```
trans.add(R.id.headcontainer, head);
```

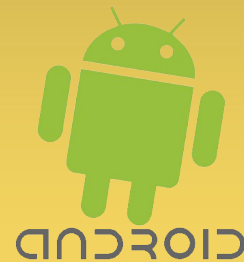
```
// trans.addToBackStack(...)
```

```
trans.commit();
```





# Transazioni e transizioni



- E' possibile associare una transizione grafica alla transazione
  - Transizioni predefinite del sistema
  - Animazioni custom
- L'animazione viene anche usata per il rollback (tasto back)
- Le animazioni sono definite come risorse del sistema
  - File XML in res/anim
- Non le vediamo ancora; ci basta sapere che hanno un ID di risorsa

# Transazioni e transizioni

- Le *transizioni* sono associate all'esecuzione di *operazioni* all'interno della *transazioni*.
  - add() - un fragment viene aggiunto all'activity
  - hide() - un fragment esistente viene reso invisibile
  - show() - un fragment invisibile viene reso visibile
  - remove() - un fragment viene rimosso dall'activity
  - replace() - un fragment viene sostituito con un altro
- Sono tutti metodi di FragmentTransaction
  - Quando si fa la commit(), si *chiede* al sistema di fare le operazioni corrispondenti, animandole in parallelo

# Transazioni e transizioni

- Per specificare la transizione da usare, si chiamano altri metodi della transazione
  - **setTransition(int style)** – una l'animazione standard
    - TRANSIT\_NONE
    - TRANSIT\_FRAGMENT\_OPEN
    - TRANSIT\_FRAGMENT\_CLOSE
  - **setCustomAnimations(int in, int out)** – Indica risorse di tipo animazione da utilizzare per l'ingresso e l'uscita
  - **setCustomAnimations(int in, int out, int stackin, int stackout)** – specifica animazioni diverse per l'in/out semplice e l'in/out da stack



# Esempio



```
FragmentManager fm = getFragmentManager()  
FragmentTransaction trans = fm.beginTransaction();
```

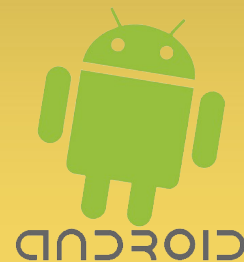
```
trans.setCustomAnimations(R.anim.slide_in_left, R.anim.slide_out_right);
```

```
ArticleFragment art = new ArticleFragment(...);  
HeadFragment head = new HeaderFragment(...);
```

```
trans.add(R.id.artcontainer, art);  
trans.add(R.id.headcontainer, head);
```

```
trans.commit();
```

# Esempio



```
FragmentManager fm = getFragmentManager()  
FragmentTransaction trans = fm.beginTransaction();
```

```
trans.setCustomAnimations(R.anim.slide_in_left, R.anim.slide_out_right);
```

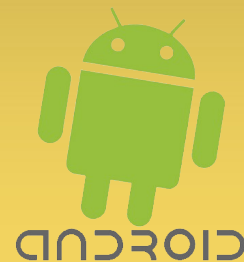
```
<?xml version="1.0" encoding="utf-8"?>  
Arti <set>  
Hea <objectAnimator  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:propertyName="x"  
    android:valueType="floatType"  
    android:valueFrom="-1280"  
    android:valueTo="0"  
    android:duration="500"/>  
tran </set>
```



# Dialog e Fragment

- Come abbiamo visto, **Activity**, **Fragment** e **Dialog** hanno ciascuno il proprio ciclo di vita, distinto da (ma integrato con) quello degli altri
- Coordinare questi cicli di vita può essere complicato
- Per questo motivo, da Android 3.0 in poi è raccomandato l'uso di **DialogFragment** al posto di **Dialog**
  - Versione retrocompatibile fino a API v4 fornita come al solito in libcompat





# DialogFragment

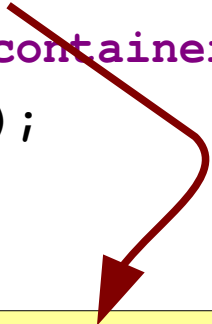
- DialogFragment è una sottoclasse di Fragment specializzata per mostrare come proprio contenuto un Dialog
- La maggior parte delle interazioni col Dialog avviene attraverso metodi di DialogFragment
  - show(), dismiss(), setStyle(), setShowsDialog(), ...
- È anche possibile accedere al Dialog contenuto nel Fragment
  - getDialog()



# DialogFragment esempio



```
public class MyDF extends DialogFragment {  
    @Override  
    public View onCreateView(LayoutInflater inflater, ViewGroup container,  
                             Bundle savedInstanceState) {  
        View v = inflater.inflate(R.layout.df, container, false);  
        getDialog().setTitle("Esempio DF");  
        return v;  
    }  
}
```



```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent" >  
  
    <TextView  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:layout_centerInParent="true"  
        android:padding="10dp"  
        android:text="@string/welcome" />  
  
</RelativeLayout>
```



# DialogFragment esempio



- Poi, nella main activity...

```
public class MyActivity extends FragmentActivity {  
    ...  
    FragmentManager fm = getSupportFragmentManager();  
    ...  
    MyDF dfrag = new MyDF();  
    ...  
    dfrag.show(fm, "MyDF TAG");  
    ...  
}
```

- MyDF può implementare i metodi per il ciclo di vita dei fragment e quelli per il controllo del Dialog
  - onCreate(), onAttach(), onStart(), onStop(), onDetach(), ecc.
  - onCancel(), onDismiss(), ecc.