# Chapitre 4

# Les interfaces utilisateurs avec Android (suite)

# Plan du chapitre 3

- Un second programme : IHM par programmation, par description
- Les `Layout`, les contrôles
- La gestion des événements
- Enchaîner les écrans
- `Toast` et traces
- L'internationalisation

# Smartphone != ordinateur

- Android tire partie des particularités des smartphones :
  - interface homme machine adapté (tactile, multitouch)
  - divers modes : vibreur, sonnerie, silencieux, alarme
  - notifications (d'applications, d'emails, de SMS, d'appels en instance)
  - de boussole, accéléromètre, GPS
  - divers capteurs (gyroscope, gravité, baromètre)
  - NFC, RFID
  - téléphonie (GSM) et réseau EDGE, 3G, 4G, SMS, MMS
  - une base de données intégrée (SQLite)
- En plus de ce qu'on peut avoir sur un ordinateur : navigateur, bibliothèques graphiques 2D, 3D (Open GL), applications de rendu multimédia (audio, vidéo, image) de divers formats, réseau Bluetooth et Wi-Fi, caméra

# Les IHM graphiques avec Android

- Bibliothèque propre
- Pas AWT, ni Swing, ni Java ME / LCDUI
- Décrit par fichier XML
- Ecran en Android géré par une activité

# Activité (Activity)

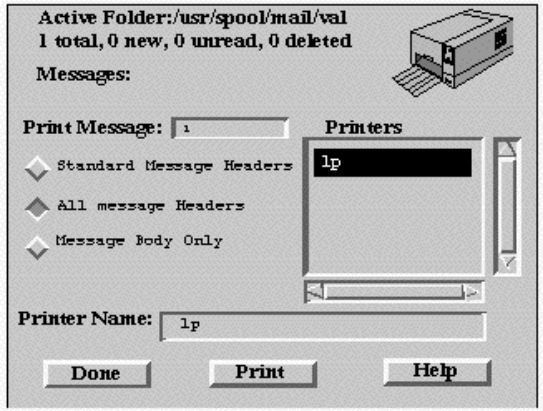
- Correspond à une seule classe Java
- Une activité gère l'affichage et l'interaction d'un écran (IHM)
- Gère les événements, lance l'affichage d'autres écrans, lance du code applicatif
- Suit un cycle de vie déterminé (cf. applets, midlets, servlets, EJB, ...)
- Utilise les `Intent` pour lancer d'autres activités

# Construction d'une IHM

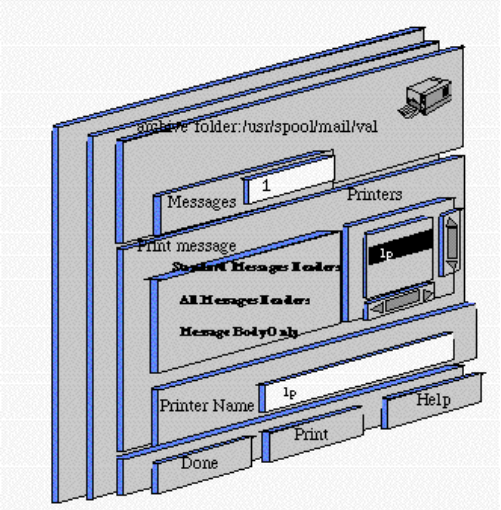
- Plutôt en XML mais
- XML ne peut pas être débogué !
- Tout ne peut pas être fait en XML

# Premier principe des IHM (1/4)

- Quand on voit ceci :

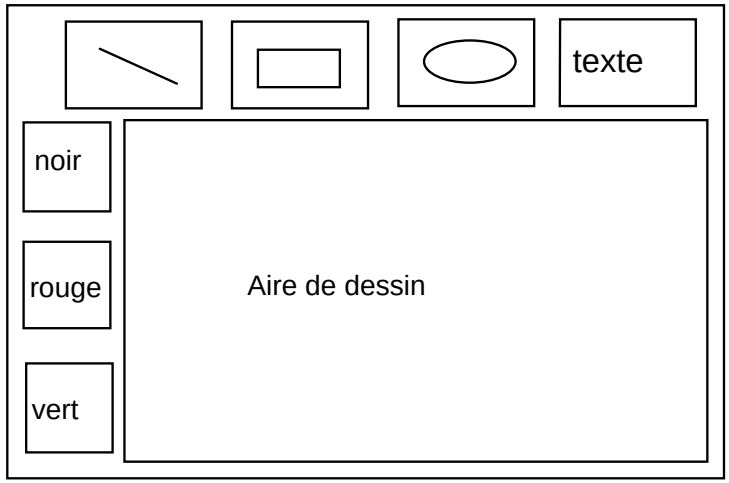



- C'est qu'on a programmé cela :

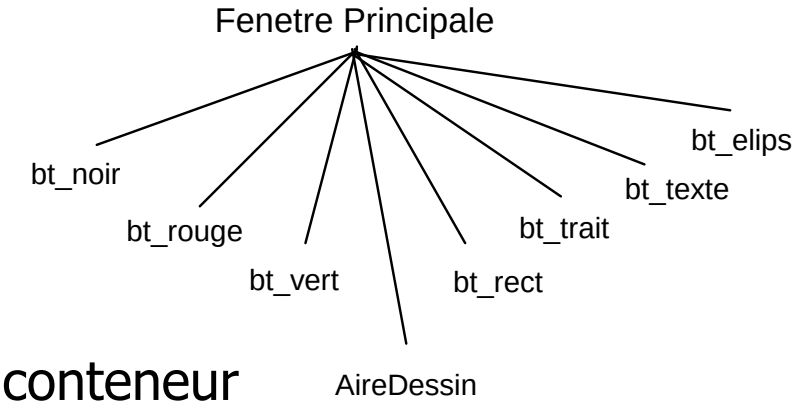


# Premier principe des IHM (2/4)

- Si on veut :



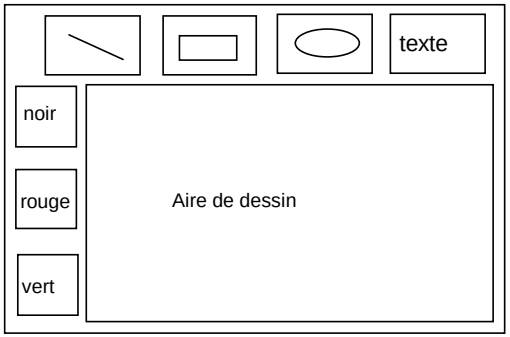

- On doit construire cela :



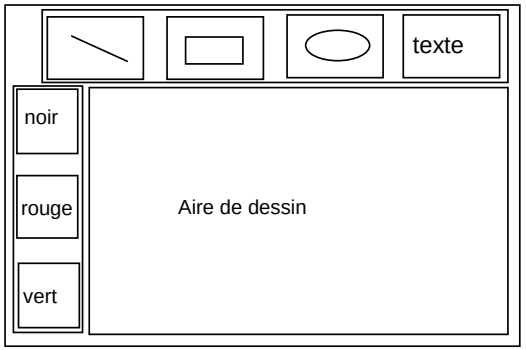

- => Fenetre principale = un conteneur

# Premier principe des IHM (3/4)

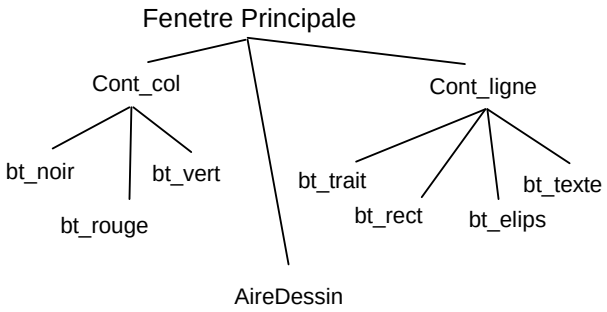
- Plus sûrement, si on veut :




on écrit plutôt :




c'est à dire :



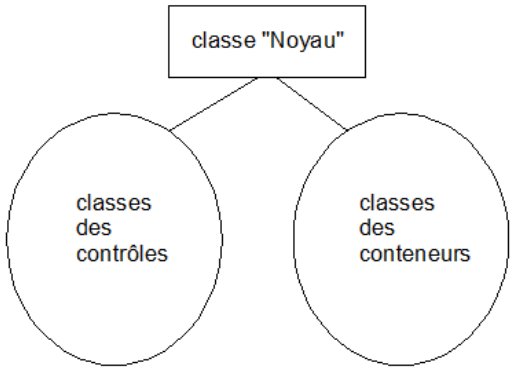
# Premier principe des IHM (4/4)

- (Premier principe) : construire une IHM, c'est mettre des composants graphiques les uns à l'intérieur des autres
- Il y a donc, dans une IHM à présenter à l'utilisateur, un arbre de composants graphiques
- Les éléments de cet arbre sont des composants graphiques (redite !)
- Etre "fils de" dans cet arbre signifie "être contenu dans"

- Voilà pour les composants graphiques ! (et le premier principe des IHM)

# Second principe des IHM (1/3)

- Les ensembles de composants graphiques sont des classes. On aura la classe des boutons, la classe des cases à cocher, etc.
- Un composant graphique particulier sera une instance particulière d'une classe. Par exemple le bouton "Quitter" et le bouton "Sauvegarder" d'une IHM seront deux instances de la classe des boutons : merci l'OO !
- Il y a une famille de conteneurs et une famille de non conteneurs
- D'où les classes de composants graphiques :

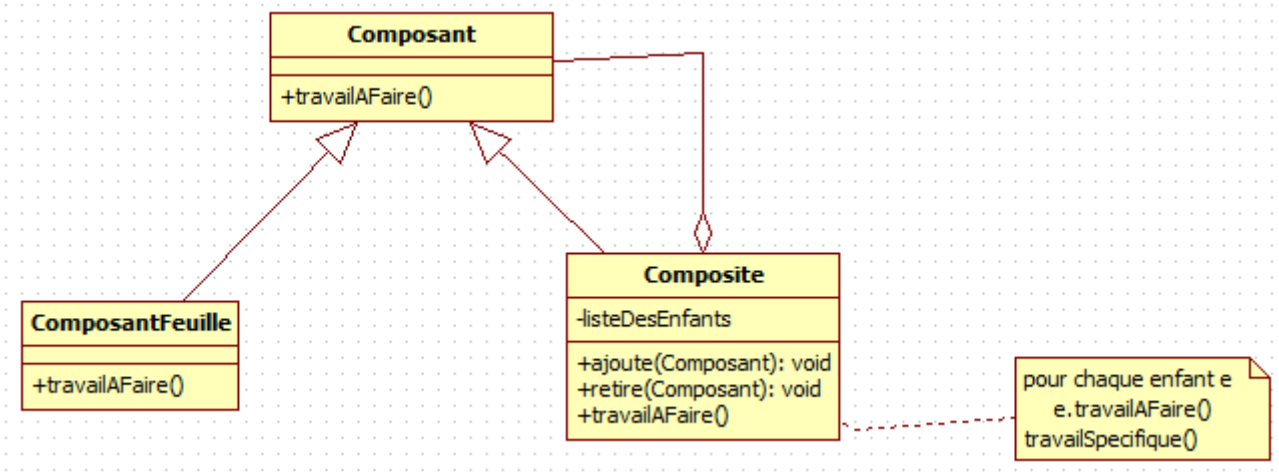



- Question : Comment sont rangées ces classes ?

- Réponse : dans un arbre d'héritage de classes : merci l'OO (bis) !

# Second principe des IHM (2/3)

- Plus précisément le point de départ de l'arborescence des classes est :




- C'est le design pattern Composite
- remarque : `travailAFaire()` est répercuté sur tout l'arbre des instances

# Second principe des IHM (3/3)

- (Second principe) : les bibliothèques pour construire des IHM sont, en Java, (et souvent !) des classes rangées par arborescence d'héritage
- Les éléments de cette arborescence sont des classes (redite !)
- Etre "fils de" dans cette arborescence signifie "hérite de"
- Voilà pour les classes (et le second principe des IHM)

# Les deux principes des IHM

- Lorsqu'on parle d'IHM, il existe deux arborescences et donc deux "principes"
- 1er principe : Une IHM est construite en mettant des composants graphiques les uns à l'intérieur des autres
- 2ième principe : les composants graphiques sont obtenus comme instances de classes. Ces classes sont rangées dans une arborescence d'héritage
- Remarque : Ces deux arbres (celui des composants graphiques et celui des classes) ont peu de choses à voir l'un l'autre
  - Le premier est l'architecture de l'interface i.e. le placement des divers composants graphiques les uns par rapport aux autres,
  - le second est un arbre d'héritage de classes donné une bonne fois par le concepteur de la bibliothèque graphique

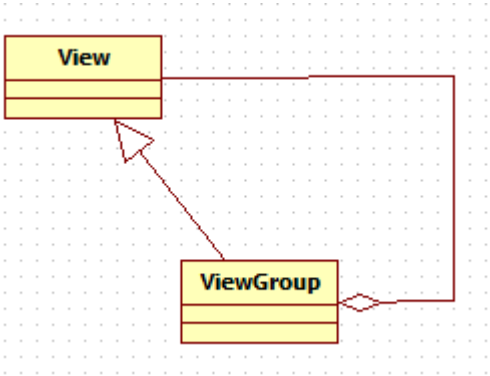
# Un composant IHM = 3 parties

- Un composant graphique a 3 parties :
  - les données qu'il représente : le modèle (model)
  - le dessin d'affichage : la vue (view)
  - ce qui prend en charge les actions de l'utilisateur sur ce composant : le controleur (controler)
- C'est l'architecture MVC
- "L'idée est de bien séparer les données, la présentation et les traitements" (*)
- Les traitements sont souvent délégués à un objet autre que le composant graphique lui même (et qui n'est pas, en général, un composant graphique) : programmation par délégation

- (*) source : `http://fr.wikipedia.org/wiki/Paradigme_MVC`

# Les fondamentaux d'Android

- La classe "Noyau" de base est la classe `android.view.View` (~ `java.awt.Component` de AWT)
- La classe de base des conteneurs est `android.view.ViewGroup` (~ `java.awt.Container` de AWT)
- On a donc :




- En Android, les conteneurs sont souvent appelés les Layout, les contrôles sont parfois appelés des widgets (window objects)

# IHM : construction procédurale vs. déclarative

- En Android on peut construire les IHM en codant en Java des instructions ...
- ... ou en décrivant l'IHM par un fichier XML
- La première solution est celle habituelle de Java (SE Swing et AWT, ME) ou d'autres domaines (Motif, Openwin de Sun, ...)
- La seconde est courante dans certains domaines (Apple, ...)

- Une nouvelle : on peut construire la plupart des IHM en glisser-déposer cf. Interface Builder de NeXT : Jean-Marie Hullot (1989) , Visual Basic, ...

# Un second programme

- On veut créer une application Android qui affiche un bouton. Lorsque l'utilisateur actionne le bouton, l'heure est affichée. Le résultat est :


- L'heure est mise à jour lorsqu'on actionne le bouton

# Code du second programme (1/2)

- On peut tout coder en Java dans l'activité :

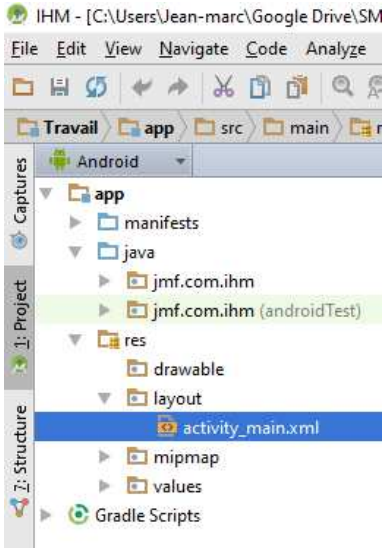
```
package android.jmf;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import java.util.Date;
public class BoutonHeureActivite extends Activity implements View.OnClickListener
{
    private Button btn;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        btn = new Button(this);
        btn.setOnClickListener(this);
        updateTime();
        setContentView(btn);
    }
    public void onClick(View view) {
        updateTime();
    }
    private void updateTime() {
        btn.setText(new Date().toString());
    }
}
```

# Code du second programme (2/2)

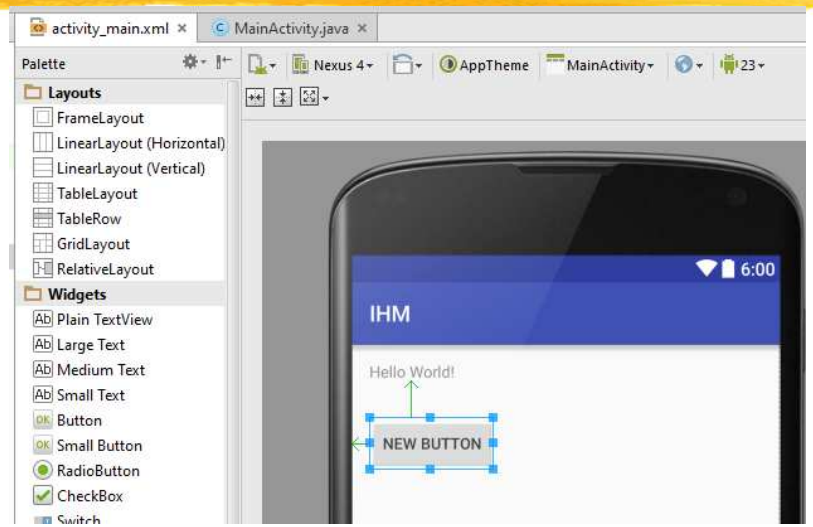
- L'activité est le listener du bouton :
  `public class BoutonHeureActivite extends Activity implements View.OnClickListener`
- Ce qui nécessite d'implémenter la méthode :
  `public void onClick(View view)` qui est lancée lorsqu'on actionne le bouton (technique des listeners cf. événement Java SE)
- Le bouton est mis dans l'activité (`setContentView(btn)`)

# Second programme : une autre solution (1/5)

- En général, pour l'IHM on utilise plutôt les fichiers XML
- Par exemple, créer une nouvelle application Android
- Ouvrir le fichier `activity_main.xml` (dans `res/layout/activity_main.xml`)
- Et on peut construire l'IHM en glisser déposer !



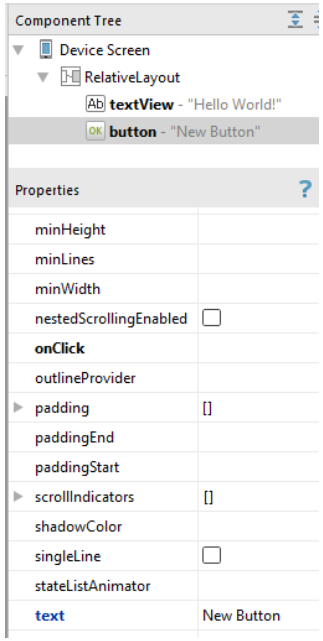
# Second programme : une autre solution (2/5)




- La fenêtre de `activity_main.xml` propose deux onglets : l'onglet Design et l'onglet Text
- L'onglet Text donne le code XML de ce fichier
- L'onglet Design (qui met parfois du temps à s'afficher lors de sa première ouverture) permet de visualiser l'IHM correspondant à ce fichier (partie droite) et de construire cette IHM en glisser-déposer avec les éléments graphiques de la partie gauche
- Par exemple, on peut sélectionner le composant `Button` et l'amener dans la partie droite
- Ce constructeur d'IHM est l'ADT Visual Designer

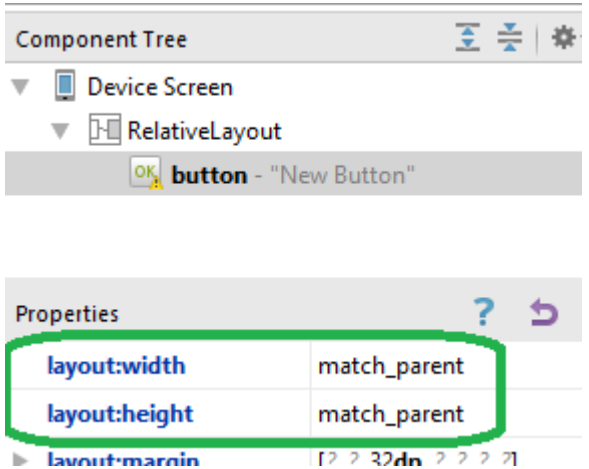
# Second programme : une autre solution (3/5)

- L'onglet Component Tree indique l'arborescence des composants graphiques de l'application
- Lorsqu'on sélectionne le composant graphique (quelle que soit la fenêtre !) apparaît l'onglet Properties : ce sont les propriétés du composant graphique



# Second programme : une autre solution (4/5)

- On peut changer la largeur et la hauteur d'un composant à l'aide des propriétés `layout:width` et `layout:height`




- Les valeurs de ces propriétés peuvent être
  - `match_parent` (anciennement nommé `fill_parent` avant l'API 8) indique que le composant graphique sera aussi grand en largeur ou hauteur que son conteneur le lui autorise
  - `wrap_content` indiquant que le composant prend seulement la taille qui lui faut en largeur ou hauteur pour s'afficher correctement

- source : `http://developer.android.com/guide/topics/ui/declaring-layout.html`

# Second programme : une autre solution (5/5)

- L'IHM est alors généré cf. les deux onglets de `activity_main.xml` :

activity_main.xml × | app × | AndroidManifest.xml ×

```
<?xml version="1.0" encoding="utf-8"?>
<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" android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp" tools:context=".MainActivity">

    <Button
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="New Button"
        android:id="@+id/button"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginTop="32dp"
        android:layout_alignParentEnd="false" />
</RelativeLayout>
```

# Texte d'un `Button`

- Les versions récentes d'Android suggère de mettre tout le texte des boutons en majuscule : les fichiers de styles (de thème) indique `android:textAllCaps="true"` (text all capitalised)
- Si on veut garder la chaine originale (en majuscule et minuscule) indiquer la propriété `android:textAllCaps="false"`
- C'est le cas pour les `Buttons`

# Correspondance attribut - méthode

- Pour positionner une propriété (= valeur d'attribut = données = ...) d'un composant graphique on peut le faire soit en XML soit par appel d'une méthode appropriée
- Dans le fichier XML, on positionne une propriété d'un composant graphique à l'aide d'une valeur d'attribut de sa balise XML
- Donc il y a une correspondance entre les noms d'attribut d'une balise XML et une méthode, pour positionner une propriété d'un composant graphique. On a, par exemple :

| XML Attributes | | |
|---|---|---|
| *Attribute Name* | *Related Method* | *Description* |
| android:alpha | setAlpha(float) | alpha property of the view, as a value between 0 (completely transparent) and 1 (completely opaque). |
| android:background | setBackgroundResource(int) | A drawable to use as the background. |
| android:clickable | setClickable(boolean) | Defines whether this view reacts to click events. |
| android:contentDescription | setContentDescription(CharSequence) | Defines text that briefly describes content of the view. |
| android:drawingCacheQuality | setDrawingCacheQuality(int) | Defines the quality of translucent drawing caches. |
| android:duplicateParentState | | When this attribute is set to true, the view gets its drawable state (focused, pressed, etc.) from its direct parent rather than from itself. |
| android:fadeScrollbars | setScrollbarFadingEnabled(boolean) | Defines whether to fade out scrollbars when they are not in use. |
| android:fadingEdgeLength | getVerticalFadingEdgeLength() | Defines the length of the fading edges. |
| android:filterTouchesWhenObscured | setFilterTouchesWhenObscured(boolean) | Specifies whether to filter touches when the view's window is obscured by |

- bibliographie :
  `http://developer.android.com/reference/android/view/View.html`

# Identifier les composants = la méthode "miracle"

- Le fichier `activity_main.xml` repère les composants par `android:id`

```
<Button
    android:id="@+id/button1"
    android:layout_width="fill_parent"
```

- Dans cet exemple il s'agit de `button1`
- Le composant est manipulé par cet identifiant dans le programme Java à l'aide de la méthode ("miracle") `findViewById(R.id.`*`nomIdentifiant`*`);`
- La valeur *`nomIdentifiant`* est celle qui apparaît dans le fichier `activity_main.xml` après `@+id/`
- Pour cet exemple ce sera `findViewById(R.id.button1);`

# Le traitement de `setContentView()`

- `setContentView()` n'est pas banale ! Elle est capable de lire un fichier xml, l'analyser, construire des objets Java (souvent des `View`) à partir de ces données, les agencer pour construire une IHM
- Ce mécanisme est dit inflate

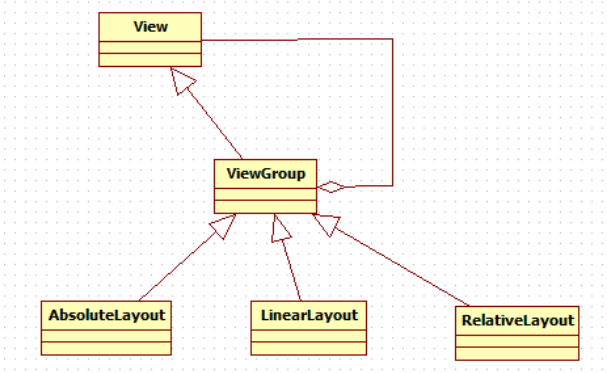
# Second programme, solution 2 : le code

- Comme toute l'IHM est faite dans `activity_main.xml`, voici le code de l'activité :

```
package android.jmf;

import java.util.Date;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class BoutonHeure2Activite extends Activity implements View.OnClickListener {
     private Button btn;
     @Override
     public void onCreate(Bundle icicle) {
          super.onCreate(icicle);
          setContentView(R.layout.activity_main);
          btn=(Button)findViewById(R.id.button1);
          btn.setOnClickListener(this);
          updateTime();
     }
     public void onClick(View view) {
          updateTime();
     }
     private void updateTime() {
          btn.setText(new Date().toString());
     }
}
```

# Les `Layout`

- Les conteneurs Android sont souvent des `XXXLayout` !
- C'est un peu différent de Java AWT ou Swing. Un Layout Android est un container et un Layout AWT à la fois
- Les principaux Layout Android sont :
  - `LinearLayout` (~ un containeur AWT géré par un FlowLayout AWT)
  - `RelativeLayout`
  - `AbsoluteLayout` (déprécié depuis l'API 3 !)
- On a donc :



# LinearLayout

- Les composants à l'intérieur d'un `LinearLayout` sont rangés les uns à la suite des autres horizontalement ou verticalement
- Principales propriétés d'un `LinearLayout` : l'orientation, le mode de remplissage (fill model)

# Orientation d'un `LinearLayout`

- L'orientation indique si le `LinearLayout` présente ces contenus sur une ligne (horizontalement) ou sur une colonne (verticalement)
- La propriété d'orientation à utiliser pour un `LinearLayout` dans le fichier XML est l'attribut `android:orientation` de la balise `LinearLayout`. Les valeurs possibles pour cette propriété sont `vertical` et `horizontal`
- La valeur `vertical` indique que les contenus seront les uns en dessous des autres, la valeur `horizontal` indique qu'ils seront les uns à la suite des autres
- L'orientation peut être modifiée à l'éxécution par la méthode `setOrientation()` lancée sur le `LinearLayout` en précisant la valeur `LinearLayout.HORIZONTAL` ou `LinearLayout.VERTICAL`

# Mode de remplissage (fill model) d'un `LinearLayout`

- Cela concerne les attributs `android:layout_width` et `android:layout_height` à positionner sur les composants graphiques contenus dans le `LinearLayout`
- Les valeurs possibles de ces propriétés peuvent être :
  - une valeur exacte de pixel (125px pour 125 pixels) : c'est fortement déconseillé
  - `wrap_content` qui indique que le composant prend la taille qu'il faut pour s'afficher correctement entièrement
  - `match_parent` (anciennement `fill_parent` avant l'API 8) indique que le composant remplit complètement la dimension indiquée du `LinearLayout`

# Le `RelativeLayout`

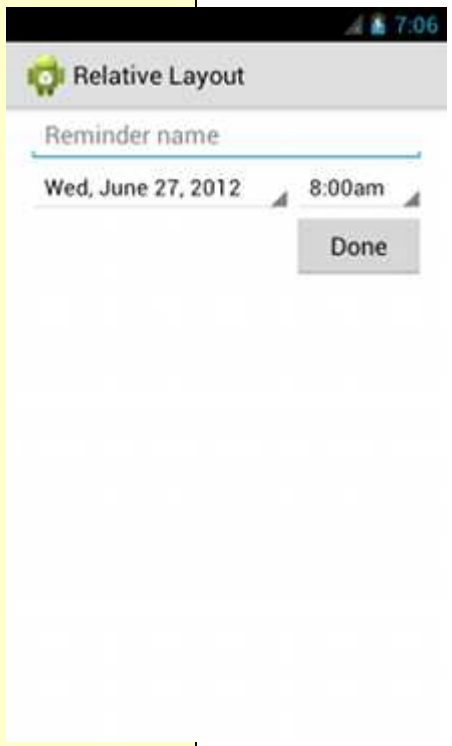
- = un conteneur qui permet de placer ses contenus les uns par rapport aux autres
- Les `Views` contenues dans le `RelativeLayout` indiquent leur positionnement à l'aide de leurs attributs (dans le fichier XML de l'IHM)
- Il ne doit pas y avoir de dépendance cyclique (bon sens)
- Les `Views` indiquent leur position par rapport à la vue parente ou leurs `Views` soeurs (en utilisant leur `id`)
- Les valeurs des attributs sont soit des `boolean`, soit l'`id` d'une autre `View`

# Attributs possibles pour une `View` dans un `RelativeLayout`

- Les `Views` dans un `RelativeLayout` peuvent utiliser les attributs :
  - `android:layout_alignParentTop` : si `true`, le haut de la `View` est calé sur le haut de la vue parente
  - `android:layout_centerVertical` : si `true`, la `View` est centrée verticalement à l'intérieur de la vue parente
  - `android:layout_below` : le haut de la vue est en dessous de la `View` indiquée (par son l'`id`)
  - `android:layout_toRightOf` : le coté gauche de la vue est à droite de la `View` indiquée (par son l'`id`)
- Il y a d'autres valeurs possibles voir à `http://developer.android.com/reference/android/widget/RelativeLayout.LayoutParams.html`

# RelativeLayout : un exemple

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp" >
    <EditText
        android:id="@+id/name"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/reminder" />
    <Spinner
        android:id="@+id/dates"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentLeft="true"
        android:layout_toLeftOf="@+id/times" />
    <Spinner
        android:id="@id/times"
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentRight="true" />
    <Button
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/times"
        android:layout_alignParentRight="true"
        android:text="@string/done" />
</RelativeLayout>
```

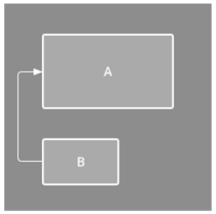


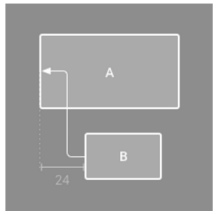
# `@+id` **vs.** `@id`

- Dans le fichier xml précédent, on utilise parfois `@+id` et parfois `@id`
- `@+id` indique qu'il faut créer, si nécessaire, une nouvelle entrée dans `R.java` (et sa classe interne `id`)
- `@id` repère simplement l'identificateur `id` et il n'y a pas de création dans `R.java`
- En fait cela fonctionne si on met toujours le `+` !

# Le `ConstraintLayout`

- nouveau Layout d'Android Studio (depuis Android Studio 2.2)
- ~ `RelativeLayout` = un conteneur qui permet de placer ses contenus les uns par rapport aux autres
- Utilise le Layout Editor d'Android Studio
- Lorsqu'on ajoute un composant par la palette, il sera placé, à l'exécution, au coordonnées (0, 0) : ce n'est pas Wysiwig !
- Pour indiquer un placement il faut lui ajouter des contraintes :
  - toute `View` doit avoir une contrainte horizontale et verticale
  - les contraintes doivent être liées à une contrainte de même type (une contraite verticale doit être liée à une autre contrainte verticale, une contraite de baseline à une autre baseline, etc.)
- Par exemple :



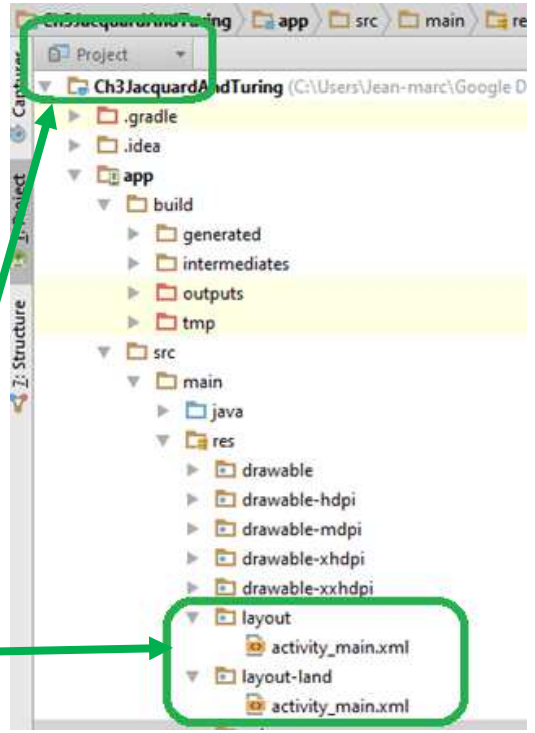





- biblio : `https://developer.android.com/training/constraint-layout/index.html#constraints-overview`

# A propos de portrait-paysage (1/2)

- Pour faire la différence entre une présentation portrait et une présentation paysage, il suffit de créer sous le répertoire `res` :
  - un répertoire `layout-land` (landscape) avec les fichiers XML d'IHM pour les présentations paysage
  - un répertoire `layout-port` (portrait) avec les fichiers XML d'IHM pour les présentations portrait
  - les fichiers par défaut sont mis dans le repertoire `layout`
- Remarque : Sélectionner l'onglet Project pour cela, pas l'onglet Android

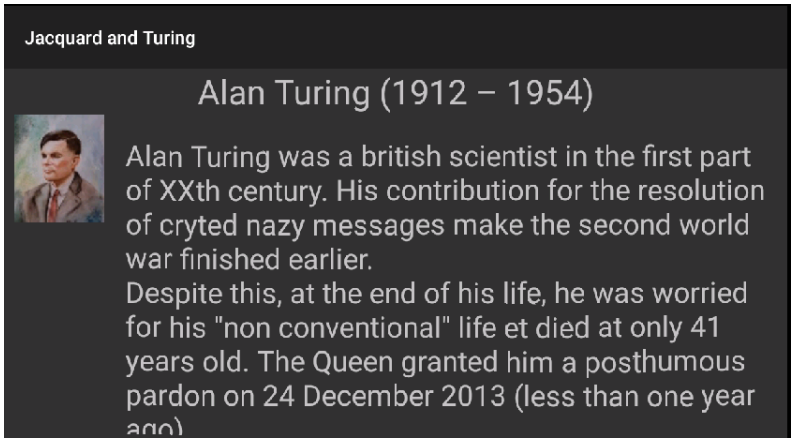


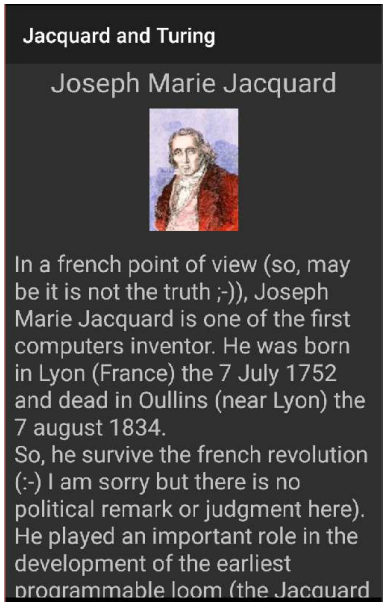
# A propos de portrait-paysage (2/2)

- Ainsi, avec la seule activité :

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

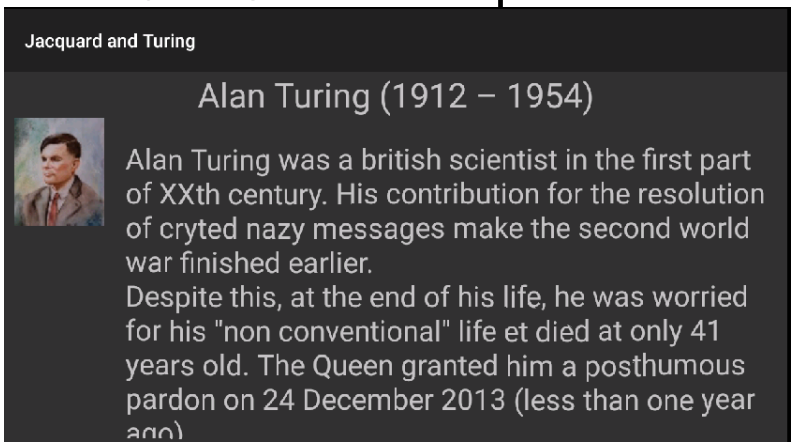
on obtient

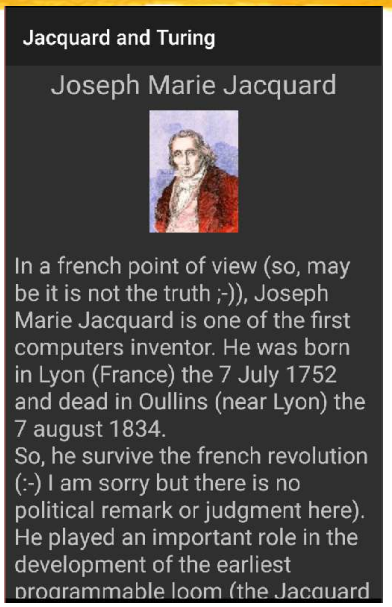


# Portrait-paysage : une démo

- demo `Ch3JacquardAndTuring`




- Et si on veut un portrait-français suivi d'un paysage-anglais (pff !)
- Il faut créer des répertoires suffixés par des extensions dans un bon ordre (`layout-fr-port` et `layout-en-land`). Voir à `http://developer.android.com/guide/topics/resources/providing-resources.html`

# Plus sur les ressources, Layouts, etc

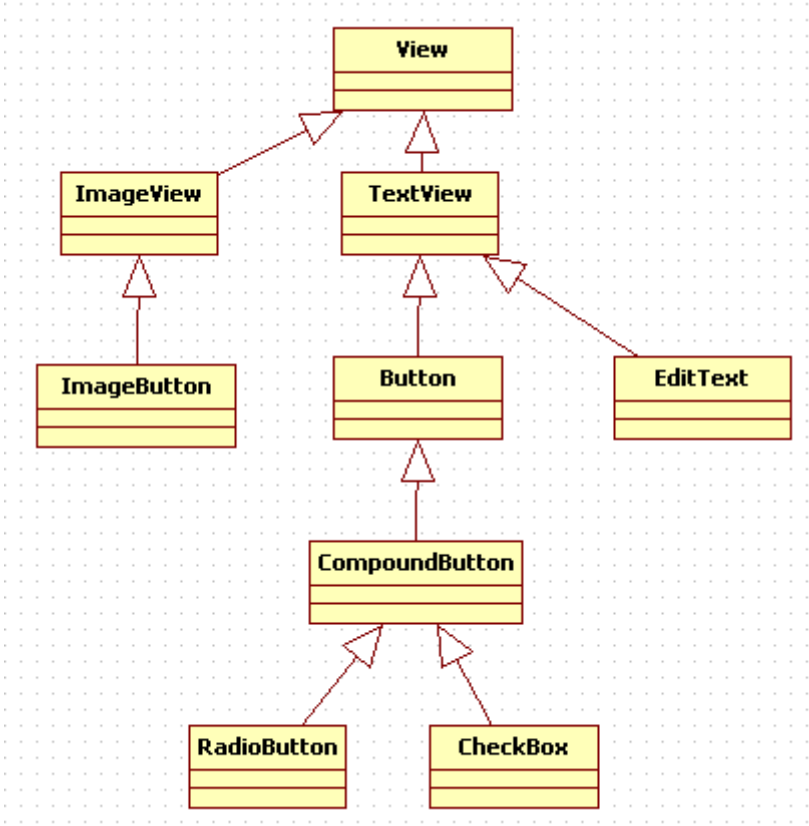
- "You can specify multiple qualifiers for a single set of resources, separated by dashes. For example, `drawable-en-rUS-land` applies to US-English devices in landscape orientation.
- The qualifiers must be in the order listed in table 2. For example:
  Wrong: `drawable-hdpi-port/`
  Correct: `drawable-port-hdpi/`"
- Bon OK, voir à
  `http://developer.android.com/guide/topics/resources/providing-resources.html`

# Les contrôles Android

- Ce sont les composants graphiques que voient l'utilisateur, avec lesquels il agit sur (contrôle) l'interface graphique
- Appelés dans certains domaines, les contrôles
- En Android ce sont (par exemple) :
  - les zones de texte non éditable (~ `Label` AWT) ou éditable (~ `TextComponent` AWT) : `TextView`
  - les boutons (~ `Button` AWT) : `Button`
  - les zones de texte éditables (~ `TextField` et `TextArea` de AWT) : `EditText`
  - les cases à cocher (~ `Checkbox` AWT) : `CheckBox` et les boutons radio `RadioButton` à regrouper dans un ensemble
- Toutes ces classes sont dans le package `android.widget` et dérivent de `android.view.View`

# Arborescence des principaux contrôles Android

- Les classes de composants non conteneurs (contrôles) sont rangées dans l'arbre d'héritage :



# TextView

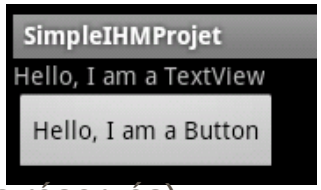
- Peut servir de zone de texte non éditable (~ `Label` AWT) et dans ce cas, sert souvent pour présenter les widgets qui suivent
- Propriétés importantes :
  - `android:text` : le texte du `TextView`
  - `android:typeface` : le type de police utilisée (monospace, ...)
  - `android:textStyle` : le style (`italic` pour l'italique, `bold_italic` pour gras et italique, ...)
  - `android:textColor` pour la couleur d'affichage du texte. Les valeurs sont en hexadécimal en unité RGB (par exemple `#FF0000` pour le rouge)

# Arborescence des composants graphiques dans une IHM

- Faire une IHM c'est mettre des composants dans des composants dans des composants ... On peut le faire par programmation en utilisant `addView(View)` d'un `ViewGroup` ou dans un fichier XML
- Par exemple le fichier :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

= un `TextView` et un `Button` dans un `LinearLayout` représente l'IHM :



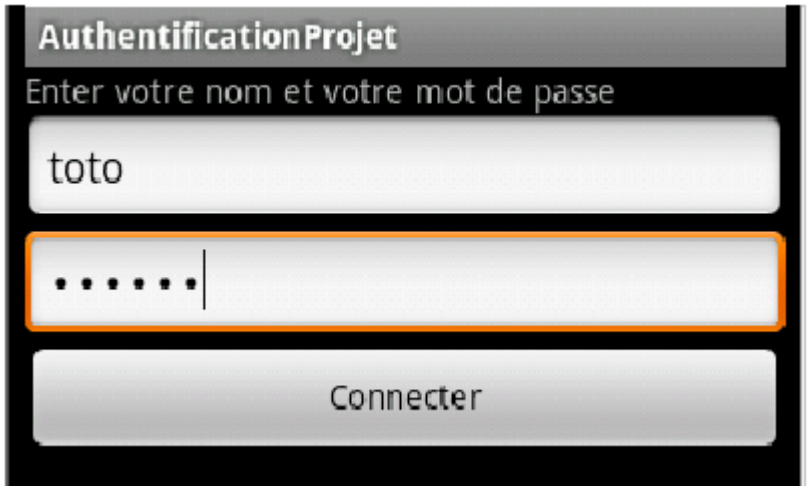
# ScrollView

- Si le contenu d'une zone de texte est trop important par rapport à l'écran, on ne voit pas tout ce contenu
- L'environnement d'exécution doit pouvoir ajouter des barres de défilement à cette zone (=scrollbars). Pour cela, on met la zone de texte dans une `ScrollView`
- Le fichier XML d'IHM contient alors :

```
<ScrollView
     android:layout_width="fill_parent"
     android:layout_height="fill_parent"
     ...
     android:paddingTop="8dp" >

     <TextView
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:padding="8dp"
          ...
     />
</ScrollView>
```

# Exercice

- Construction d'une IHM avec des composants Android
- Construire et faire afficher l'activité :




- Remarque : le troisième (si, si !) composant n'affiche pas l'écho des caractères

# Fin