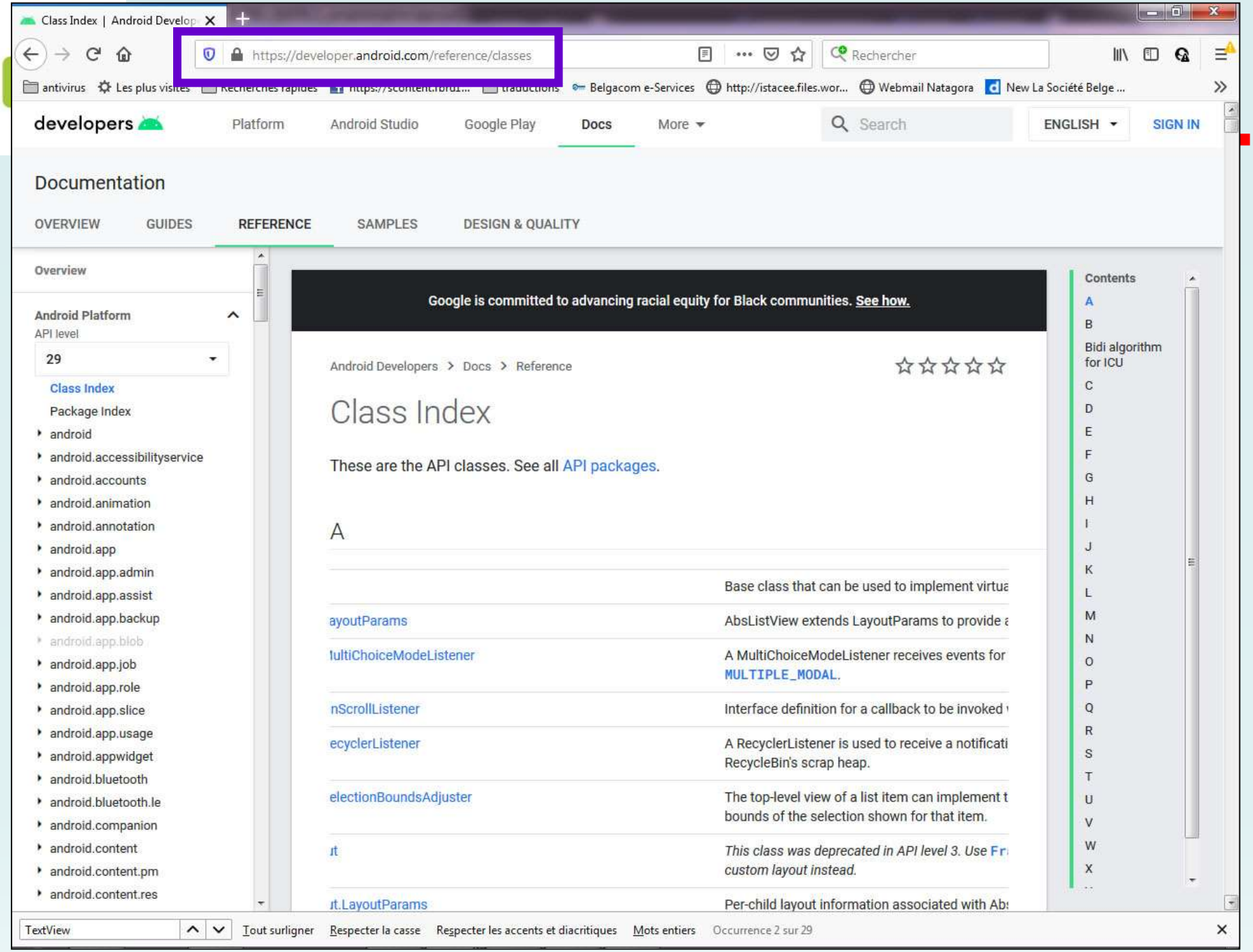
Une petite introduction à Google Android
2. Les bases de la programmation

Claude Vilvens / Christophe Charlet

2023

Agenda


Class Index | Android Develop
https://developer.android.com/reference/classes
Rechercher
antivirus
Les plus visités
Belgacom e-Services
http://istacee.files.wor...
Webmail Natagora
New La Société Belge ...
developers
Platform
Android Studio
Google Play
Docs
More
Search
ENGLISH
SIGN IN
Documentation
OVERVIEW
GUIDES
REFERENCE
SAMPLES
DESIGN & QUALITY
Overview
Android Platform
API level
29
Class Index
Package Index
android
android.accessibilityservice
android.accounts
android.animation
android.annotation
android.app
android.app.admin
android.app.assist
android.app.backup
android.app.blob
android.app.job
android.app.role
android.app.slice
android.app.usage
android.appwidget
android.bluetooth
android.bluetooth.le
android.companion
android.content
android.content.pm
android.content.res
Google is committed to advancing racial equity for Black communities. See how.
Android Developers > Docs > Reference
Class Index
These are the API classes. See all API packages.
A
Base class that can be used to implement virtua
ayoutParams
AbsListView extends LayoutParams to provide a
ultiChoiceModeListener
A MultiChoiceModeListener receives events for
MULTIPLE_MODAL.
nScrollListener
Interface definition for a callback to be invoked
ecyclerListener
A RecyclerListener is used to receive a notificati
RecycleBin's scrap heap.
electionBoundsAdjuster
The top-level view of a list item can implement t
bounds of the selection shown for that item.
t
This class was deprecated in API level 3. Use Fr
custom layout instead.
t.LayoutParams
Per-child layout information associated with Abs
Contents
A
B
Bidi algorithm for ICU
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
TextView
Tout surligner
Respecter la casse
Respecter les accents et diacritiques
Mots entiers
Occurrence 2 sur 29

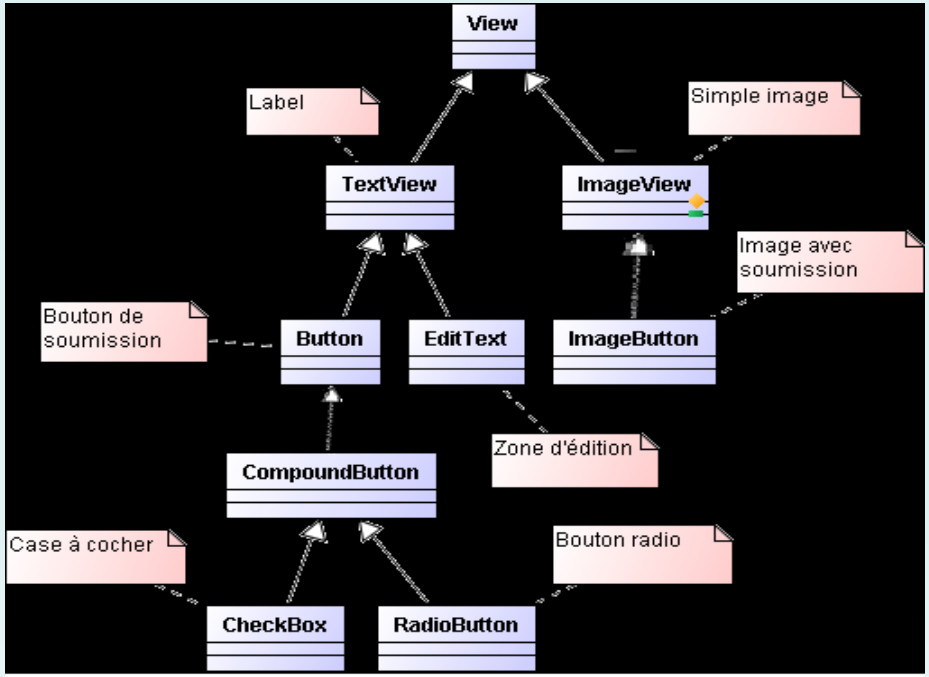
12. Les

Rappel: le FindViewById crée un objet vue
à partir de ressources (fichier XML généré par Android Studio)

1. View et ViewGroup

Tous les composants graphiques d'Android héritent de la classe **View** (package android.view). Cette dernière sert de base aux sous-classes appelées **widgets**, qui sont des éléments de l'interface graphique (GUI) prêts à être utilisés comme les champs de texte, les boutons ou cases à cocher, etc.

Composants usuels:



12. Les interfaces graphiques (2/20)

La classe **ViewGroup** sert de base aux sous-classes appelées **Layouts**, qui offrent différentes sortes de structure de page comme linéaire, tabulaire et relative.

Ce sont des composants graphiques !

L'interface utilisateur se définit donc en utilisant un arbre d'objets **View** et **ViewGroup** : les composants s'imbriquent les uns dans les autres selon une **arborescence hiérarchique**. Les objets de héritant de **View** sont les feuilles de l'arbre, ceux héritant de **ViewGroup** sont les branches.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res
/android"    android:orientation="vertical"
android:layout_width="wrap_content"
android:layout_height="wrap_content" >

<Button android:id="@+id/bouton_ok" android:text="OK"
android:layout_height="wrap_content" … ></Button>
<Button android:layout_width="wrap_content" …
id:text="Annuler" android:id="@+id/bouton_annuler"></Button>

</LinearLayout>
```

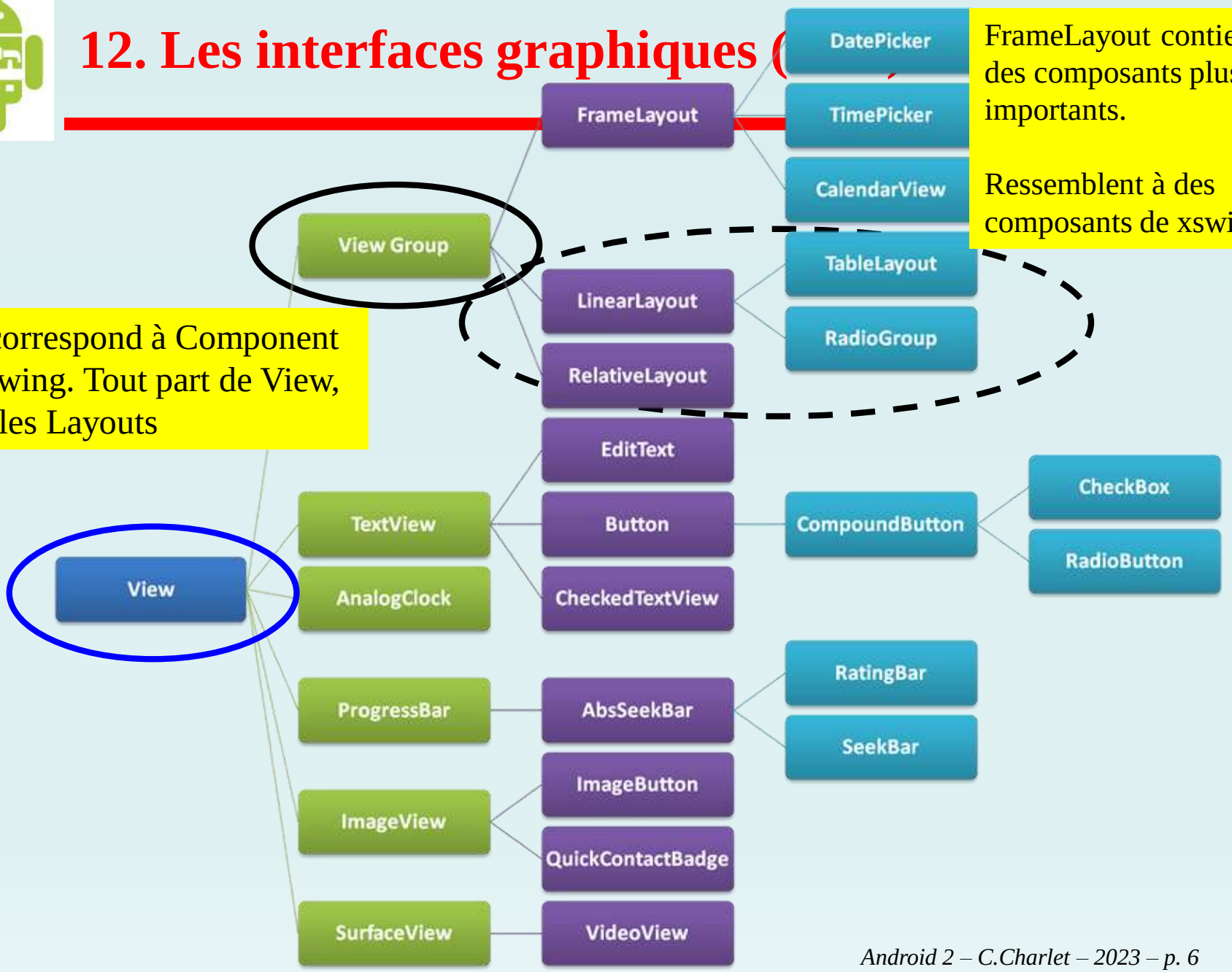
Chaque ViewGroup doit demander à chacun de ses enfants de s'afficher (ceux-ci peuvent demander une taille et une position à leur parent, mais c'est ce dernier qui détient la décision finale concernant la position et la taille de chacun de ses enfants).

12. Les interfaces graphiques (

FrameLayout contient des composants plus importants.

Ressemblent à des composants de xswing

View correspond à Component dans Swing. Tout part de View, même les Layouts



12. Les interfaces graphiques (4/20)

2. Déclaration et propriétés des composants d'un GUI

Comme nous l'avons déjà vu, bien qu'il soit théoriquement possible de définir un GUI par programmation Java pure, il est pratique d'utiliser pour cela des **déclarations XML** du type :

<element attribut1="valeur" attribut2="valeur"> ... </element>

```
<Button android:layout_width="wrap_content" … android:text="Annuler" …
android:id="@+id/bouton_annuler"></Button>
```

Chaque Vue Android peut disposer d'un identifiant défini grâce à l'attribut android:id. La convention consiste à utiliser le format **@+id/nom_unique** où nom_unique représente le nom de l'objet à utiliser dans le code Java. En fait :

- **@** : indique au système que la ressource pointée par l'identifiant n'est pas définie dans le fichier Java courant, mais dans un fichier externe; Ressource externe
- **+** : indiquer à Android que l'identifiant peut être ajouté à la liste des identifiants de l'application si cela n'est pas déjà fait;
- **id/** : combiné avec le "@", ce préfixe oblige Android à regarder dans la liste des identifiant déjà déclarés (par +) : l'id doit être unique;
- **nom_unique**: nom (unique) de l'identifiant.

12. Les interfaces graphiques (5/20)

Chaque type de vue possède des attributs spécifiques, et d'autres communs à tous les widgets (mais aussi aux layouts).

Parmi les propriétés communes, **layout_width** et **layout_height** sont obligatoires et existent pour toutes les vues. Ces deux propriétés peuvent prendre trois types de valeur :

♦ taille fixe (50px),

♦ **fill_parent** (le composant prend automatiquement la même taille que son conteneur parent après le placement des autres widgets) et

♦ **wrap_content** (le composant prend la taille de son contenu et la vue ne prend que la place qui lui est juste nécessaire).

A noter que fill_parent est à présent remplacé dans les versions récentes par **match_parent**.

La classe de base View comporte à elle seule **une armée d'attributs possibles**. Citons :

♦ **android:clickable** || setClickable(boolean) : l'élément est réactif à un clic de souris;

♦ **android:minHeight** / **android:minWidth**: hauteur/largeur minimale du composant;

♦ **android:onClic** : spécification de la méthode à appeler automatiquement lors d'un clic de la souris sur le composant.

Mais nous allons y revenir …

La taille de l'écran peut être variable suivant le smartphone utilisé par l'application
=> Problème pour les composants graphiques (point de vue de la position)

Parmi les propriétés propres aux widgets, on trouve aussi **layout_x** et **layout_y** :

```
<TextView
        android:id="@+id/widget33" …
        android:background="@color/yellow"
        android:text="Bonjour jeune patricien !"
        android:textColor="@color/black"
        android:layout_x="66dp"
        android:layout_y="14dp" />
```

C'est pas le dpi (dot per inch)!

L'unité classique des tailles des composants est le **dp** ou **dip** (**d**ensity **i**ndependent **p**ixel) : elle se base sur la densité de pixels d'un écran pour définir la taille du composant considéré et **correspond plus ou moins à 1 pixel sur un écran de 160 dpi** (**d**ot **p**er **i**nch = pixel par pouce).

Ecran de référence: 160 dpi (HTC)

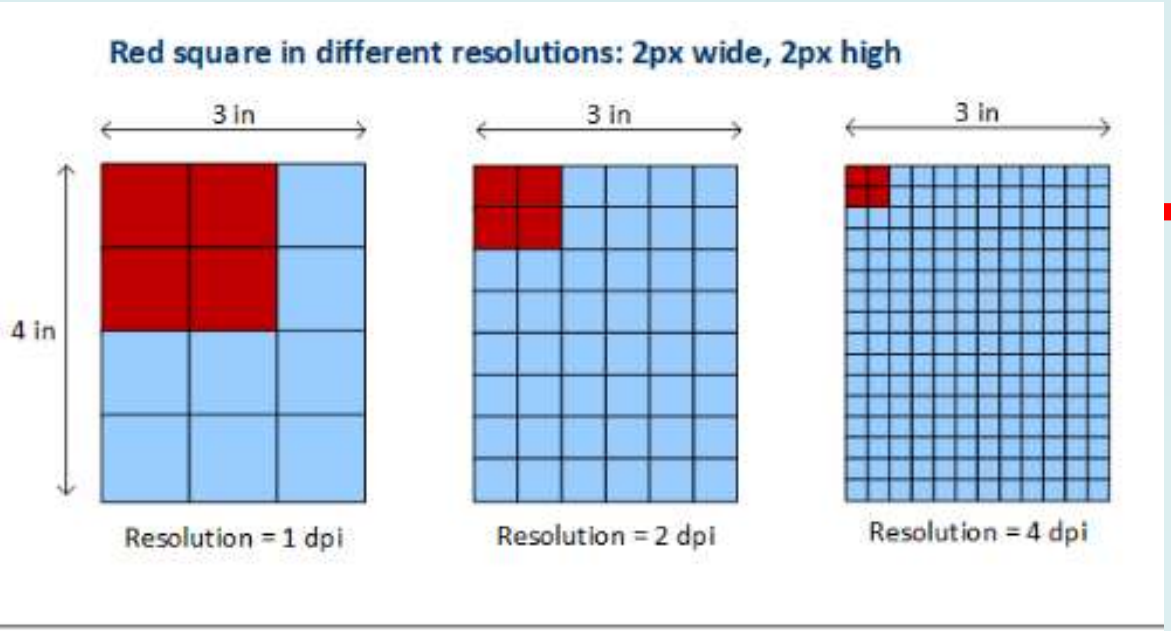
Si px désigne la taille en pixels et dpi la taille d'une image, alors :

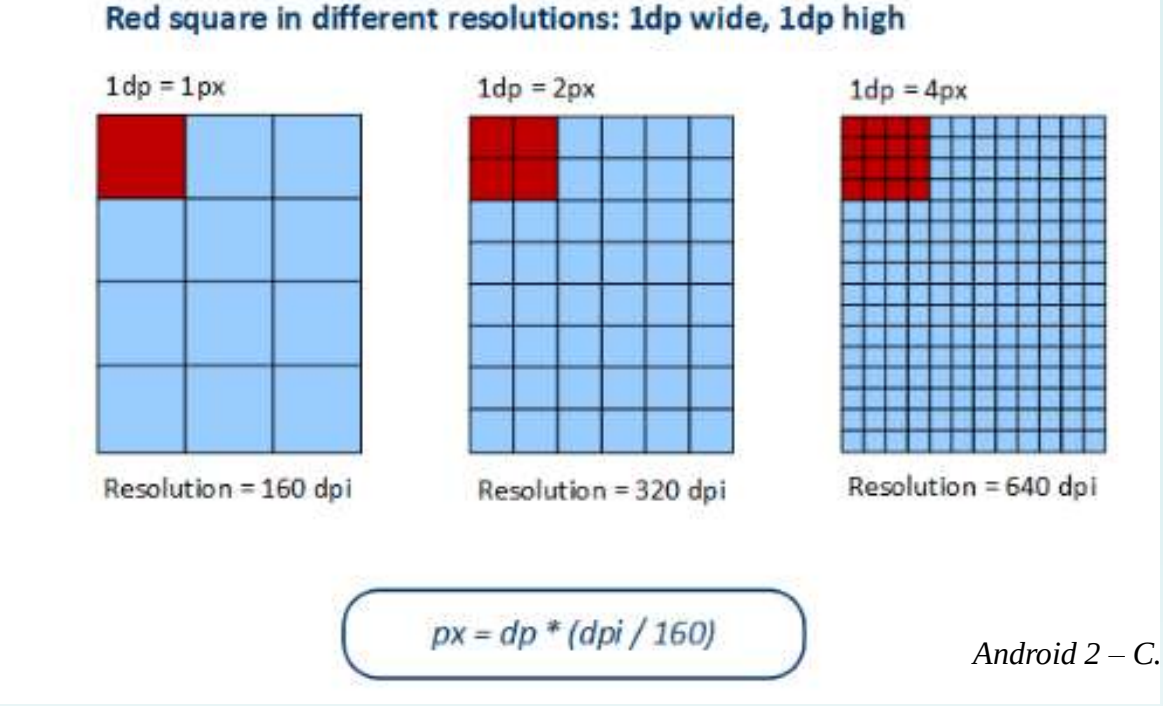
dp = px / (dpi/160) = px * 160 / dpi ou px = dp * (dpi / 160)

On parle encore d'"**unité virtuelle**" car elle n'est pas mesurable directement, mais plutôt **concrétisée sur un matériel de densité donnée**.

(un smartphone iPad de 9.7 pouces (1024×769) est 132 dp et un UTC de 4.3 pouces (800x480) est 216 dp)

L'intérêt : **en fonction de la densité de l'écran sur lequel s'exécute effectivement l'application**, le système appliquera les transformations nécessaires pour augmenter ou diminuer la taille effective d'1 dp.


Red square in different resolutions: 2px wide, 2px high
3 in
3 in
3 in
4 in
Resolution = 1 dpi
Resolution = 2 dpi
Resolution = 4 dpi


Red square in different resolutions: 1dp wide, 1dp high
1dp = 1px
1dp = 2px
1dp = 4px
Resolution = 160 dpi
Resolution = 320 dpi
Resolution = 640 dpi
px = dp * (dpi / 160)

12. Les interfaces graphiques (7/20)

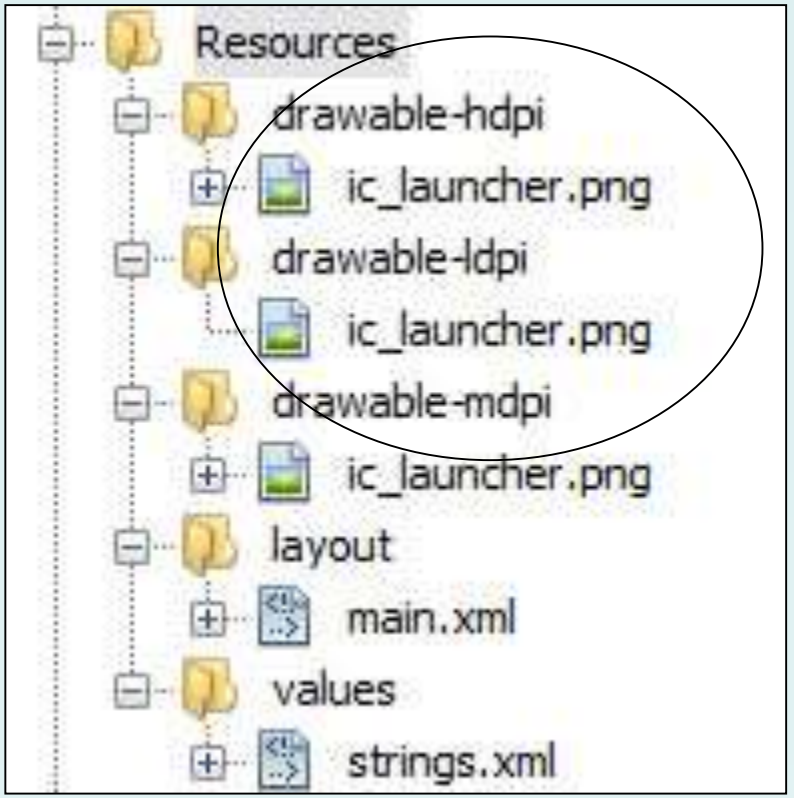
Jusqu'à Android 3.2 ;-), une possibilité envisageable est de choisir pour l'écran l'une des 3 densités prédéfinies **ldpi** (120 dpi), **mdpi** (160 dpi) et **hdpi** (240 dpi), éventuellement complétées par xhdpi (320 dpi) ou xxhdpi (480 dpi).

(un moniteur de 17 pouces (1280×1024) est 96 dpi
et un moniteur de 19 pouces est 87 dpi)

Une fois cette densité choisie, elle sera appliquée partout, quelle que soit la véritable densité du téléphone.

Un designer pourra donc créer une image, et au pire, voire cette image avec des pixels doublés, ou divisés par deux.

Bien sûr, le designer peut fournir une version pour chacune des densités et ainsi être sûr qu'un pixel d'un bitmap correspondra toujours à un pixel à l'écran, quel que soit le terminal.



On s'adapte à la densité de résolution (Low, Medium, High)

12. Les interfaces graphiques (8/20)

3. Les composants spécifiques

Bien sûr, chaque composant graphique, matérialisé par une classe héritée de View (package android.widget), possède ses propres caractéristiques :

1) **TextView** : "label", le composant graphique le plus simple.

♦ android:autoLink || setAutoLinkMask(int) : contrôle si le texte correspond à un lien quelconque (adresse email ou URL) qui, le cas échéant, est converti en élément cliquable;
♦ android:autoText || setKeyListener(KeyListener) : contrôle si le champ doit fournir une correction automatique de l'orthographe – il existe de nombreuses autres utilisations d'un KeyListener (nombres, majuscules, etc)
♦ android:editable : permet de définir cet élément comme champ de saisie.
♦ android:hint || setHint(int) : propose un texte par défaut lorsque le champ est vierge.
♦ android:maxLength || setMaxLength(int) : définit la longueur maximale du texte en terme de nombre de caractères.

2) **Button** : le "bouton" classique, hérité de TextView

12. Les interfaces graphiques (9/20)

3) **ImageView** et **ImageButton** : équivalent en image de TextView et Button. L'attribut android:src permettant de préciser l'image utilisée (une ressource graphique, un "drawable").

4) **EditText** : classe dérivée de TextView, cette vue est l'habituel champ de saisie. On s'en doute, une multitude d'attributs permet de contrôler la saisie comme android:password.

****A partir d'Android 1.5 : la plupart des modes de saisie ont été regroupés dans un seul attribut **android:inputType** dont la valeur est composée d'une *classe* et de *modificateurs*, séparés par le caractère "|" : la classe décrit généralement ce que l'utilisateur est autorisé à saisir et détermine l'ensemble de base des touches disponibles sur le clavier logiciel.

** **classes possibles** : text (par défaut), number, phone, date, time, etc.

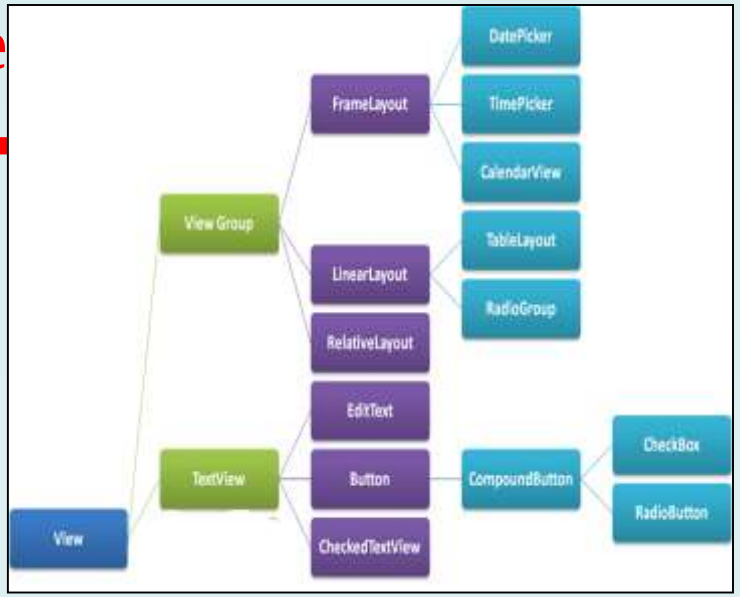
Afin de préciser ce que l'utilisateur pourra saisir, on peut ensuite enrichir les classes par un ou plusieurs modificateurs (exemple : **text|textCapCharacters** précise que la saisie attendue est de type texte et que ce dernier sera entièrement transformé en lettres majuscules).

5) **CheckBox** et **RadioButton** : les habituelles classe sœurs, dérivées de CompoundButton ;

la classe RadioGroup peut guetter un changement d'état avec
public void **setOnCheckedChangeListener** (RadioGroup.OnCheckedChangeListener listener)

12. Les interfaces graphique

4. Les gestionnaires de mise en page



Analogues à ceux que l'on connaît en AWT/Swing, **mais ce sont des composants graphiques** :

♦ **LinearLayout** : toutes les vues sont placées les unes à la suite des autres - ce layout aligne tous ses enfants verticalement ou horizontalement.

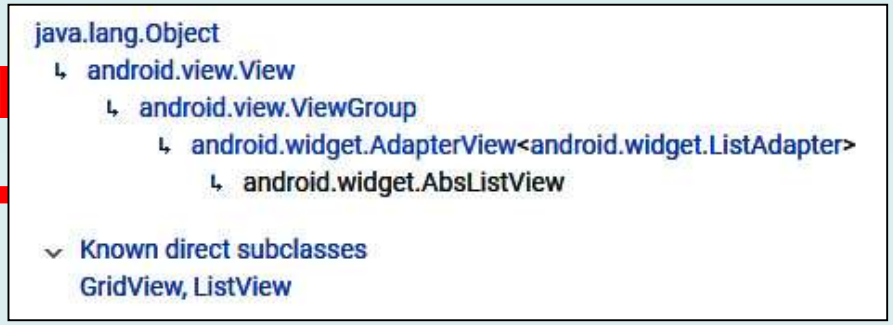
♦ **TableLayout** : le tableau en lignes-colonnes; chaque rangée est représentée par un TableRow qui peut contenir 0 à n View.

♦**RelativeLayout** : les vues sont positionnées les unes par rapport aux autres; une série d'attributs est disponible pour chaque composant (View/ViewGroup) afin de *spécifier la relation avec les composants voisins* (tel que **android:layout_below** pour spécifier l'id du composant au-dessus, **android:layout_above** pour spécifier l'ID du composant en-dessous, etc).

♦**FrameLayout** : permet de réserver un espace à l'écran pour afficher une vue avec "décorations" diverses, comme une date ou un temps;

♦ **AbsoluteLayout** : on a compris, mais c'est rarement une bonne idée.

12. Les interfaces grapl



On peut ajouter comme *outil de disposition* en tant que GUI d'une activité:

Equivalent à la JTable

♦ **GridView** : les éléments sont vus comme placés sur une grille "scrollable" dont le maillage sera l'unité; chaque élément va alors recevoir une dimension représentée par une quantité de colonne et de rangées qu'elle va occuper sur cette grille (android:layout_rowCount et android:columnCount), les recouvrements étant spécifiés au moyen de android:layout_columnSpan et android:layout_rowSpan;

Equivalent à la JList

♦ **ListView** : affiche une liste d'éléments scrollable.

♦ **ScrollView** : permet le défilement d'un nombre important de vues.

Un certain nombre d'attributs sont incontournables pour un composant classique :

♦ **alignement vertical ou horizontal** : android:orientation="vertical" ou "horizontal";

♦ **gravité**, c'est-à-dire importance relative de chaque composant : andoid:layout_weight (en %)

♦ **positionnement relatif** : android:layout_alignParentTop, android:layout_alignParentBottom, android:layout_alignParentLeft, android:layout_alignParentRight, android:layout_centerHorizontal, android:layout_centerVertical, android:layout_centerInParent.

Exemple


5554:Galaxy_gio
23:08
MainActivity
Bonjour jeune patricien !
Toi aussi tu veux du data mining ?
Login ?
Mot de passe ?
ANOVA ?
Tests de conformité ?
Chi carré ?
Ok !!!!
Basic Controls
Hardware Buttons
MENU
DPAD not enabled in AVD
Hardware Keyboard not enabled in AVD
Use on screen keyboard

Exemple

```
<?xml version="1.0" encoding="utf-8"?>
<AbsoluteLayout
        android:id="@+id/widget32"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        xmlns:android="http://schemas.android.com/apk/res/android">
<TextView
        android:id="@+id/widget33" …
        android:background="@color/yellow"
        android:text="Bonjour jeune patricien !"
        …
<ImageView
        android:id="@+id/widget43"…
        android:src="@drawable/loi_normale"
        android:layout_x="60dp"
        android:layout_y="277dp" />
</AbsoluteLayout>
```

identifiants "automatiques" : à changer

Idéalement, les noms des id devraient être modifiés (pas la valeur par defaut)

Il existe plusieurs manières de gérer les événements graphiques

12. Les interfaces graphiques (12/20)

5. La gestion des événements graphiques

3 boutons …

main.xml

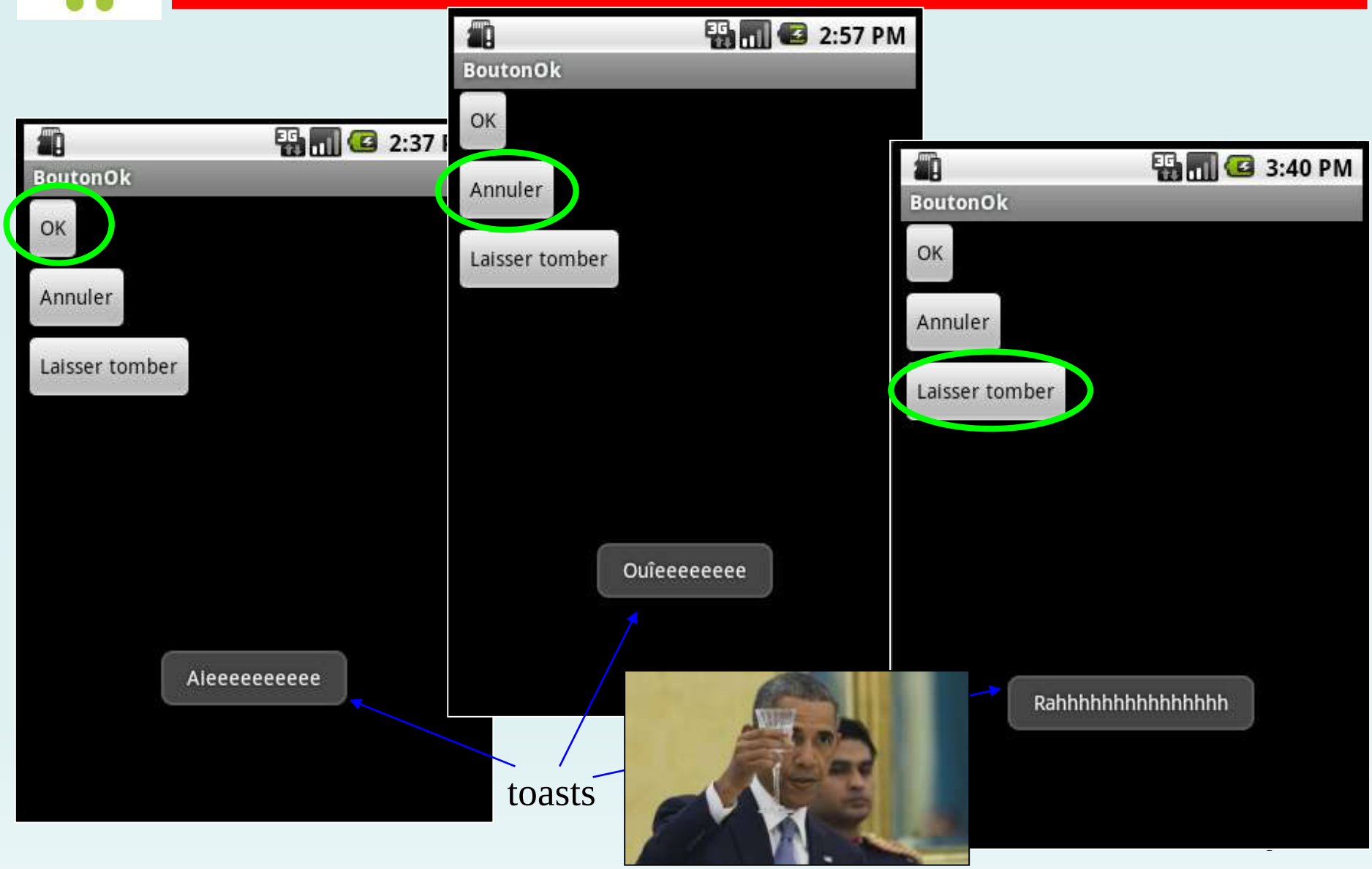
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"    android:layout_width="wrap_content"
android:layout_height="wrap_content" >

<Button android:id="@+id/bouton_ok" android:text="OK"
android:layout_height="wrap_content"
android:layout_width="wrap_content"></B
<Button android:layout_width="wrap_cont
android:layout_height="wrap_content" andr
android:id="@+id/bouton_annuler"></Bu
<Button android:layout_width="wrap_cont
android:layout_height="wrap_content"
android:id="@+id/bouton_assez"
android:text="'Laisser tomber'"></Button>

</LinearLayout>
```

R.java

```
package …;
public final class R
{  …
   public static final class id {
      public static final int bouton_ok=0x7f04000d;
      public static final int bouton_annuler=0x7f040003;
      public static final int bouton_assez=0x7f04000c; …
   }
   public static final class layout {
      public static final int main=0x7f020000;
   }
   public static final class string {
      public static final int app_name=0x7f030000;
   }
}
```

Juste 3 boutons

12. Les interfaces graphiques (13/20)



12. Les interfaces graphiques (14/20)

a) gestion au moyen d'un attribut

main.xml

```
<?xml version="1.0" encoding="utf-8"?> …
<Button android:id="@+id/bouton_ok" android:text="OK"
android:layout_height="wrap_content" android:layout_width="wrap_content"
android:onClick="methode_ok">
</Button> …
```

Il doit y avoir une méthode dans l'activité associée à cet interface graphique : « methode_ok » . Mélange entre la partie graphique et programmation !

```
package basics.gui;

import android.app.Activity; …

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

12. Les interfaces graphiques (15/20)

```
public void methode_ok(View v)
    {
            String msg = "Aieeeeeeeeee";
            Toast.makeText(this,msg,Toast.LENGTH_LONG).show();
    }
}
```

this: (paramètre contexte) référence sur l'objet principale sur lequel on est

la classe **Toast** (package android.widget) étant une simple vue destinée à délivrer un petit message à l'utilisateur.

Sa méthode polymorphe

public static Toast **makeText** (Context context, int resId, int duration)
public static Toast **makeText** (Context context, CharSequence text, int duration)

construit un Toast standard avec une simple TextView ou chaîne de caractères

pendant un temps long ou court (configurable) fixé par :

public static final int LENGTH_LONG
public static final int LENGTH_SHORT

12. Les interfaces graphiques (16/20)

b) gestion au moyen d'un listener explicite

```
package basics.gui;
import android.app.Activity; …
public class OkHandler extends Activity implements OnClickListener
{
  public void onCreate(Bundle savedInstanceState)
  {
      super.onCreate(savedInstanceState); setContentView(R.layout.main);
     Button b = (Button) this.findViewById(R.id.bouton_annuler);
     b.setOnClickListener(this);
  }…
  public void onClick(View v)
  {      if (v == this.findViewById(R.id.bouton_annuler))
         {          String msg = "Ouîeeeeeeee";
                    Toast.makeText(this,msg,Toast.LENGTH_LONG).show();
         }
   }
}
```

Idem java classique: ActionListener et ActionPerform , on a simplement changer les noms

Enregistrement sur le composant qu'on veut surveiller => il faut donc la référence

FindViewById => Classe R => sous classe Id => identifiant du bouton = > dans le fichier XML (propriétés du bouton) => on instancie un objet de la classe button (fille de View) => on a la référence du bouton

12. Les interfaces graphiques (17/20)

c) gestion au moyen d'un listener implicite

On utilise un setOnClickListerner, on crée un onClickListener juste au moment où on en a besoin

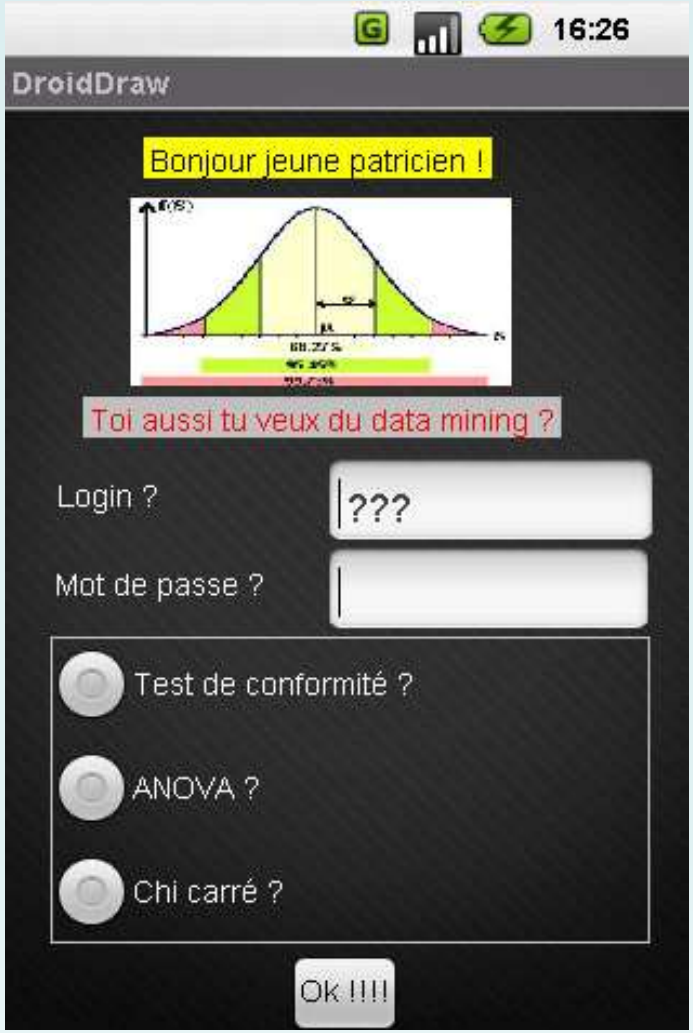
On instancie une classe (=View) dans laquelle se trouve l'interface onClickListener qui doit juste définir la seule méthode onClick => Classe qui n'a pas de nom => nom$x.class OkHandler$x.class. Une fois la classe définie, on l'instancie

```
package basics.gui;

import android.app.Activity;

public class OkHandler
{
  public void onCreate(B
  { …
      Button bfin = (Button) this.findViewById(R.id.bouton_assez);
      bfin.setOnClickListener(new View.OnClickListener()
       {
         public void onClick(View v)
         { // Toast.makeText(this,"Rahhhhhhhhhhhhhh", ..).show();
           // this is refering to the View.OnClickListener instead of the Activity
           Toast.makeText(OkHandler.this,"Rahhhhhhhhhhhhhh",
                 Toast.LENGTH_LONG).show();
         }
       });
  } …}
```

Et il faut vraiment confectionner un tel GUI à la main ?

12. Les interfaces graphiques (18/20)

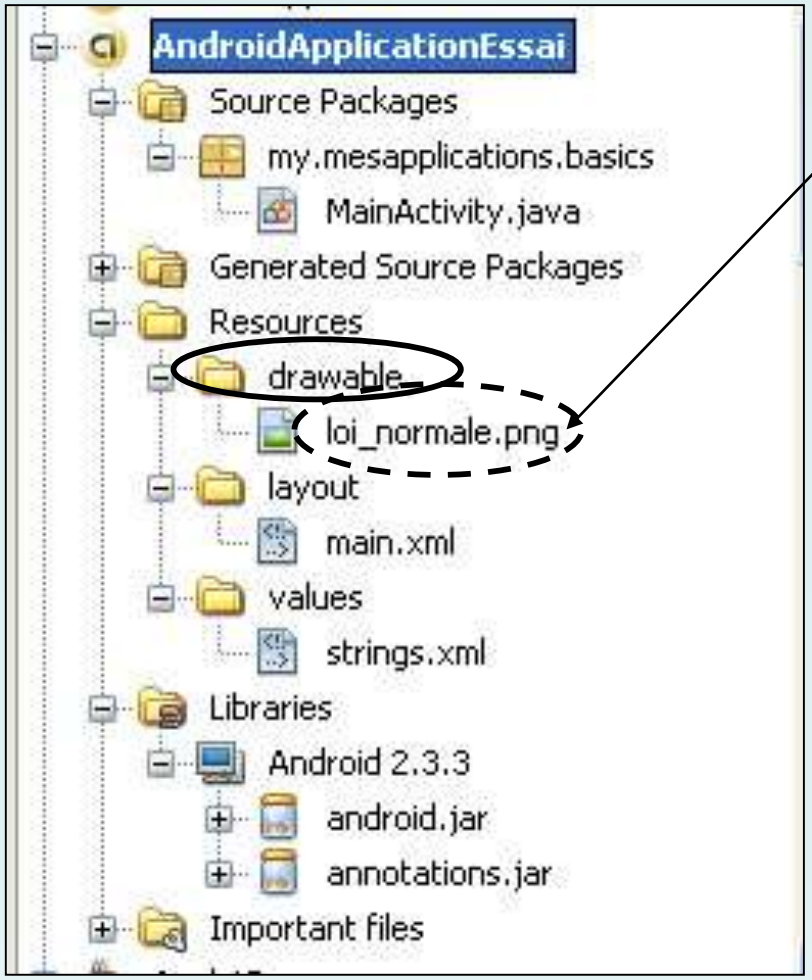
6. Intégrer des images

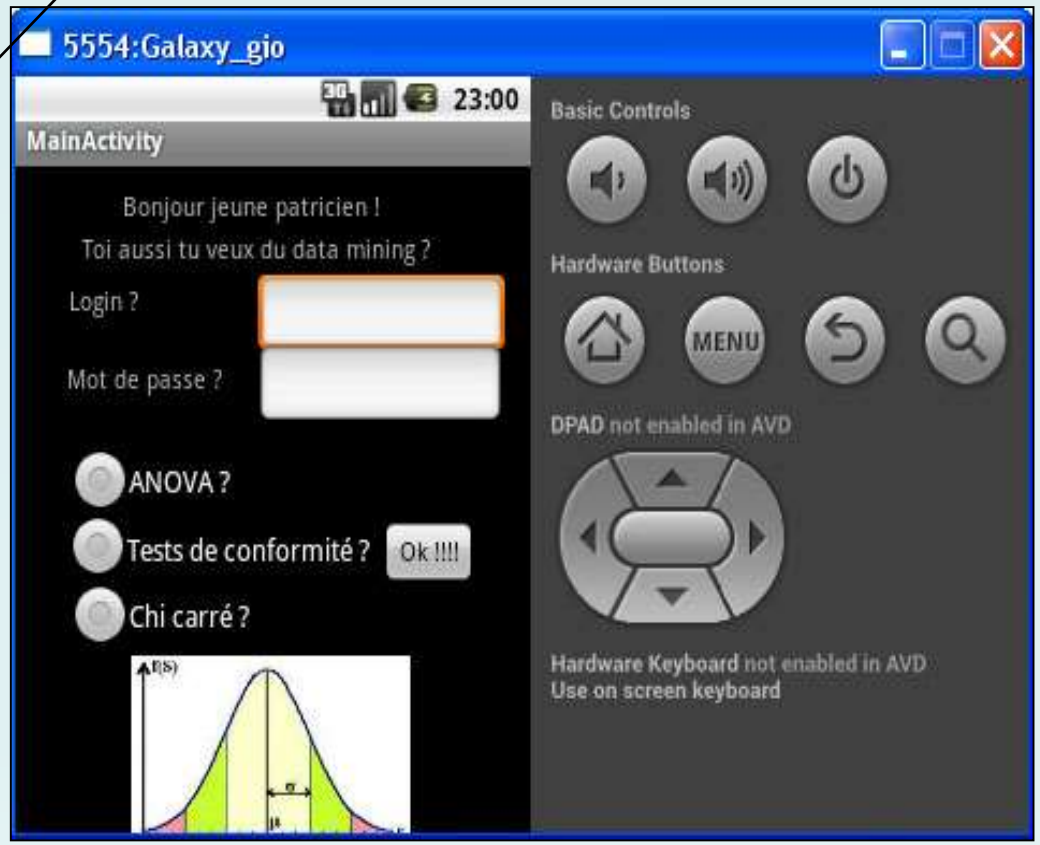



```
<?xml version="1.0" encoding="utf-8"?>
<AbsoluteLayout
        android:id="@+id/widget32"
…
<TextView
        android:id="@+id/widget33" …
        android:background="@color/yellow"
        android:text="Bonjour jeune patricien !"
        android:textColor="@color/black" …
…
<ImageView
        android:id="@+id/widget43"…
        android:src="@drawable/loi_normale"
        android:layout_x="60dp"
        android:layout_y="277dp" />
</AbsoluteLayout>
```

12. Les interfaces graphiques (19/20)

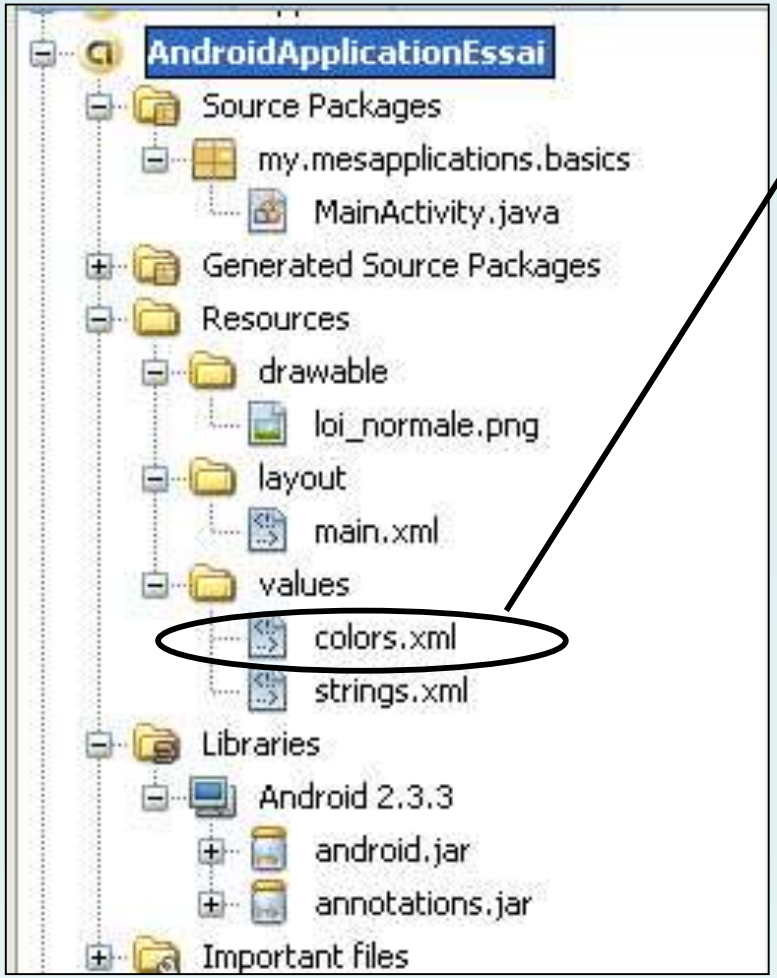
nom de fichier à la Unix !





12. Les interfaces graphiques (20/20)

7. Utiliser des couleurs

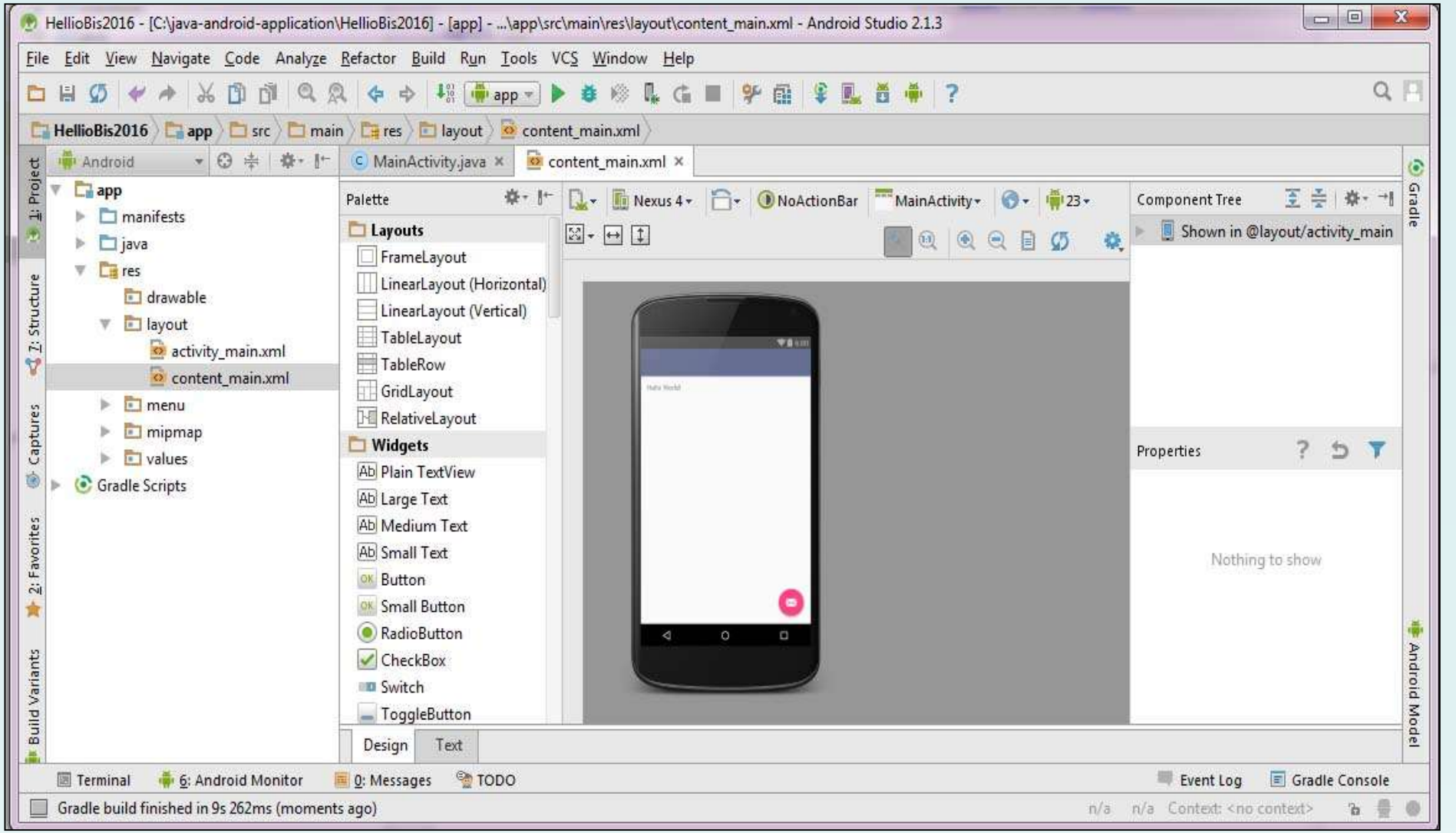


colors.xml

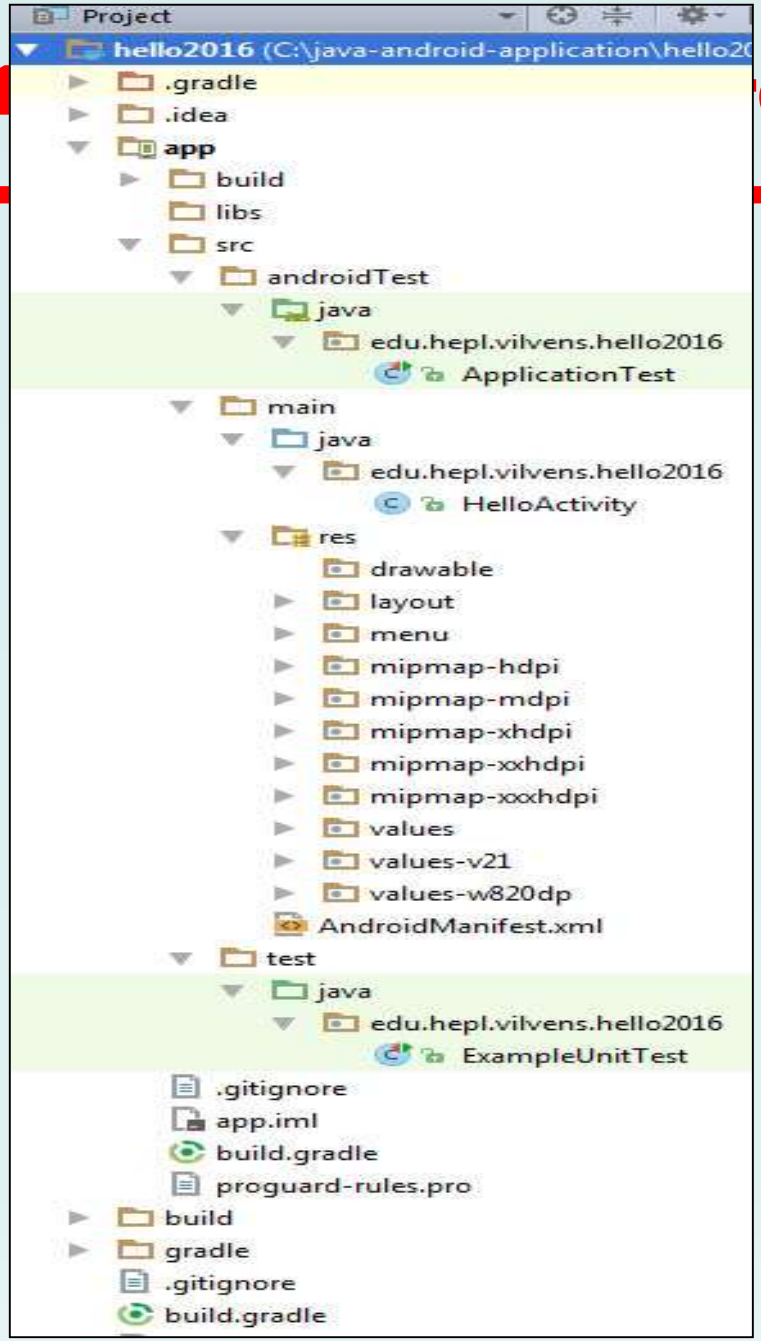
```
<?xml version="1.0" encoding="UTF-8"?>
<resources>
        <color name="black">#000</color>
        <color name="yellow">#0f0</color>
</resources>
```

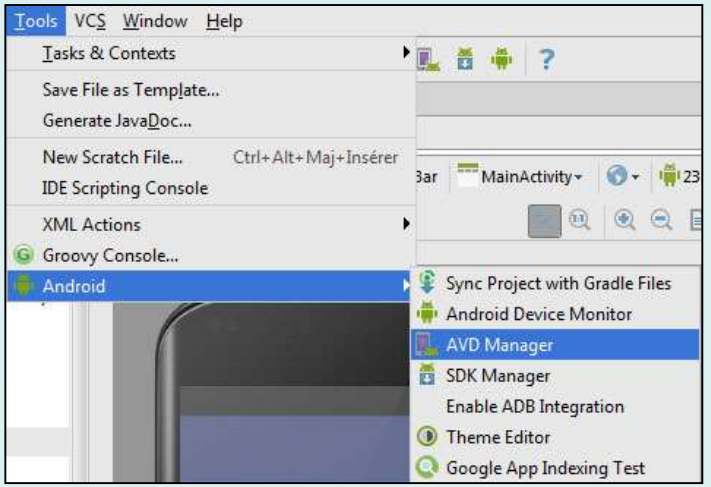
13. Design UI avec Android Studio (1/5)

suite



1 oid Studio (2/5)



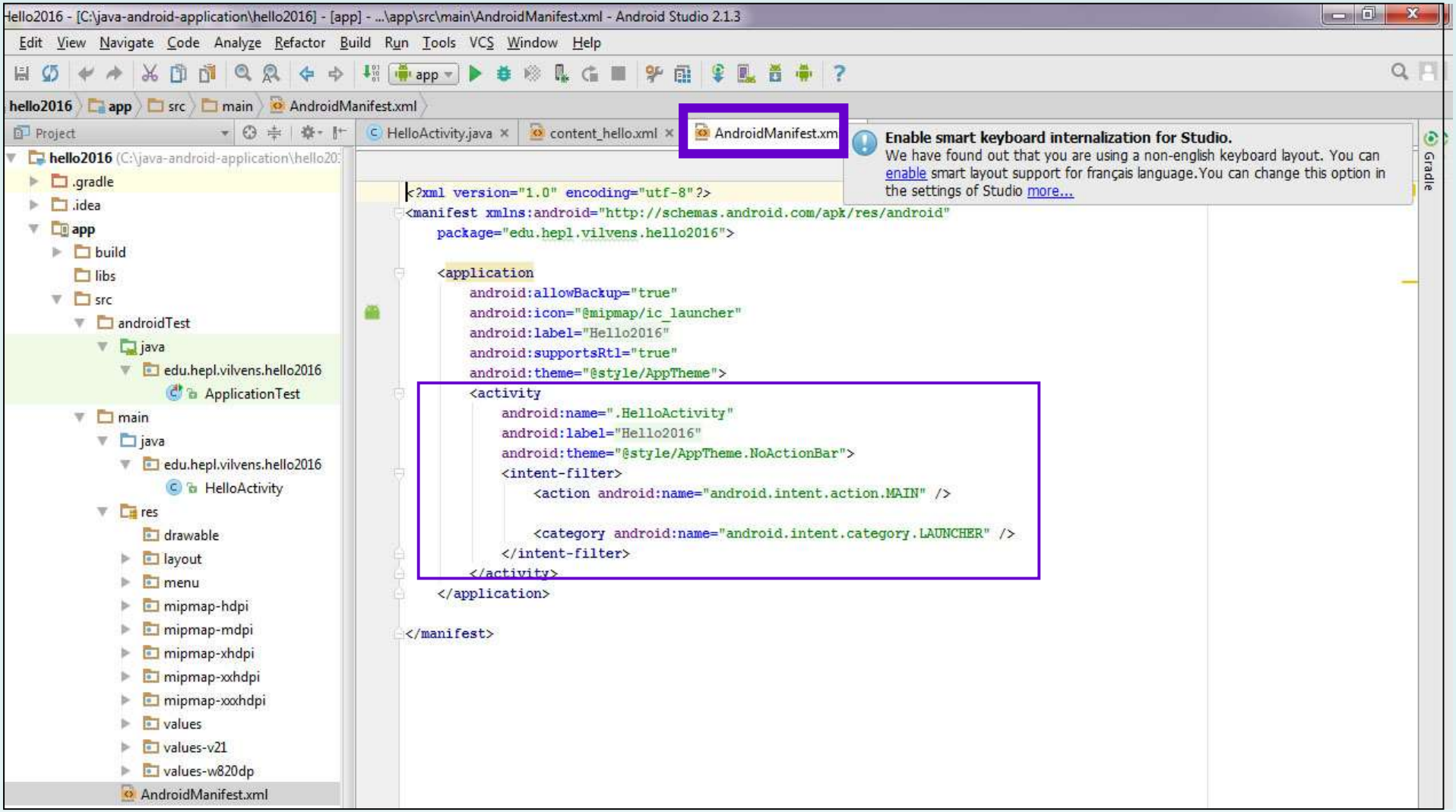


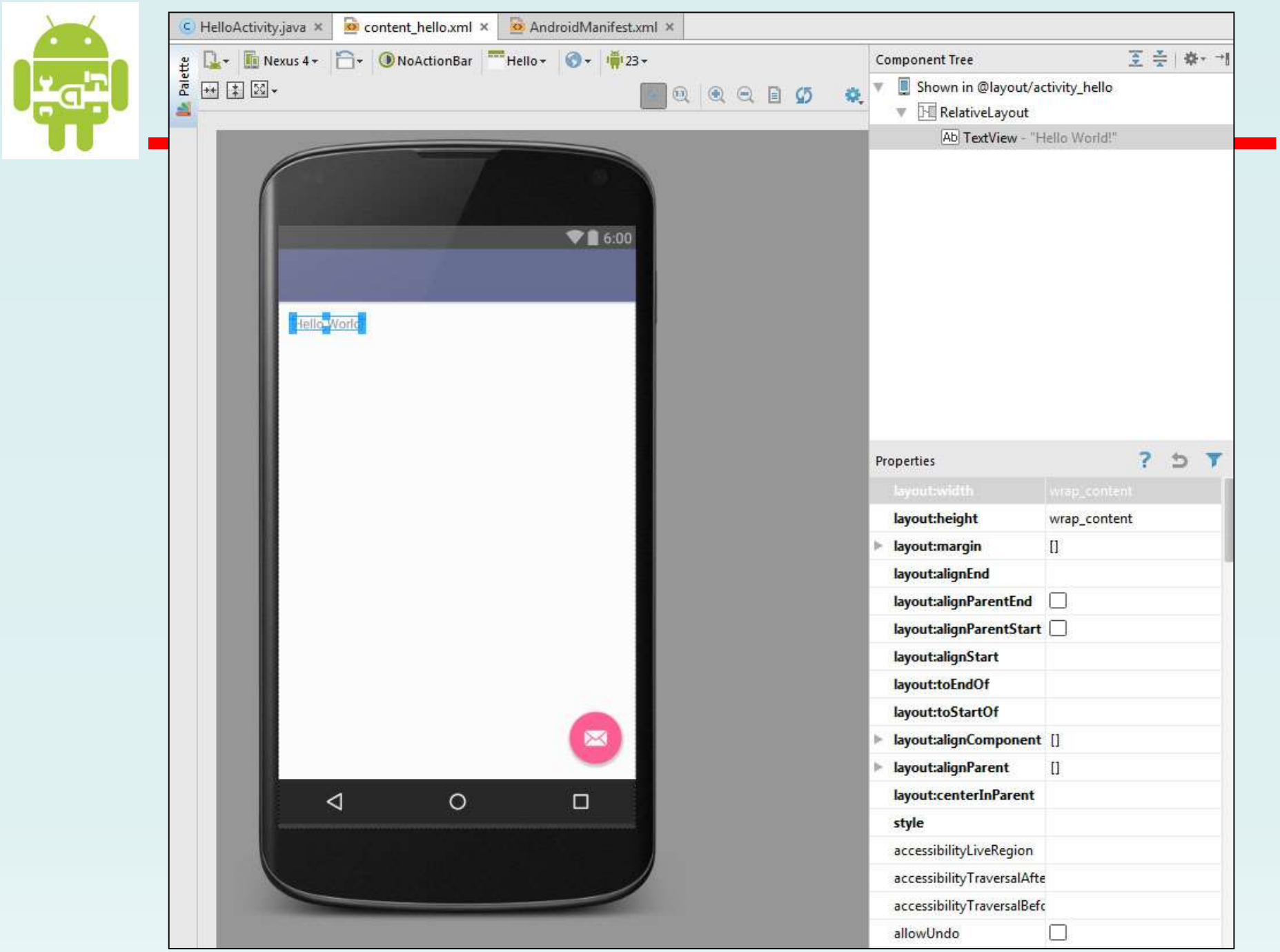
1

HelloActivity.java × | content_hello.xml × | AndroidManifest.xml ×

```
package edu.hepl.vilvens.hello2016;

import ...

public class HelloActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
        fab.setOnClickListener((view) → {
                Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
                        .setAction("Action", null).show();
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_hello, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
}
```

13. Design UI avec Android Studio (4/5)




HelloActivity.java
content_hello.xml
AndroidManifest.xml
Palette
Nexus 4
NoActionBar
Hello
23
Component Tree
Shown in @layout/activity_hello
RelativeLayout
TextView - "Hello World!"
6:00
Hello World
Properties
layout:width
wrap_content
layout:height
wrap_content
layout:margin
[]
layout:alignEnd
layout:alignParentEnd
layout:alignParentStart
layout:alignStart
layout:toEndOf
layout:toStartOf
layout:alignComponent
[]
layout:alignParent
[]
layout:centerInParent
style
accessibilityLiveRegion
accessibilityTraversalAfte
accessibilityTraversalBefc
allowUndo

14. Les menus (1/10)

1. Le menu classique

Tout utilisateur d'Android est habitué à faire apparaître un menu au sein d'une application par sollicitation de la touche "menu" de son smartphone.

Les deux questions qui se posent donc à nous sont

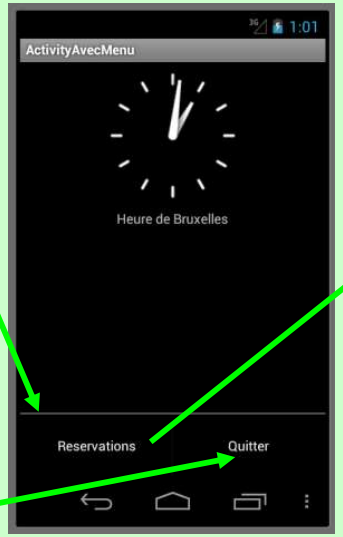
- comment définir et faire apparaître un menu ?
- comment gérer le choix d'un item de menu ?

En fait, le menu doit d'abord être défini au sein d'un fichier XML à l'allure suivante :

14. Les menus (2/10)

menu_reservation.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
<item android:id="@+id/reservation"
    android:title="Reservations">
    <!--android:icon="@drawable/option"-->
    <menu android:id="@+id/sousmenu">
        <item android:id="@+id/vol"
            android:title="Un vol"   />

        <item android:id="@+id/chambre"
            android:title="Une chambre"  />
    </menu>
</item>
<item android:id="@+id/quitter"
    android:title="Quitter" />
    <!--android:icon="@drawable/quit"-->
</menu>
```



Ce fichier XML doit se trouver dans un répertoire "menu" du répertoire res.

14. Les menus (3/10)

On passe par R, pas de FindViewById mais des MenuInflater (méthode: getMenuInflater() fournit l'objet)

MenuActivity.java

```
package my.applications.divers;

public class MenuActivity extends Activity
{
    @Override
    public void onCrea
    {
        super.onCreate(savedInstanc
        setContentView(R.layout.ma
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu)
    {
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.menu_reservation, menu);
        return true;
     }
}
```

désigner la ressource menu dans le fichier R (typiquement ici "R.menu.menu_reservation")

l'objet Menu à remplir ("gonfler") avec les items du fichier XML

Crée le menu à partir du fichier menu ressource au moyen de la méthode :

public void **inflate** (int menuRes, Menu menu)

fournit l'objet inflater dédié au contexte qu'est l'activité

14. Les menus (4/10)

Reste à r menu la, il suffit de

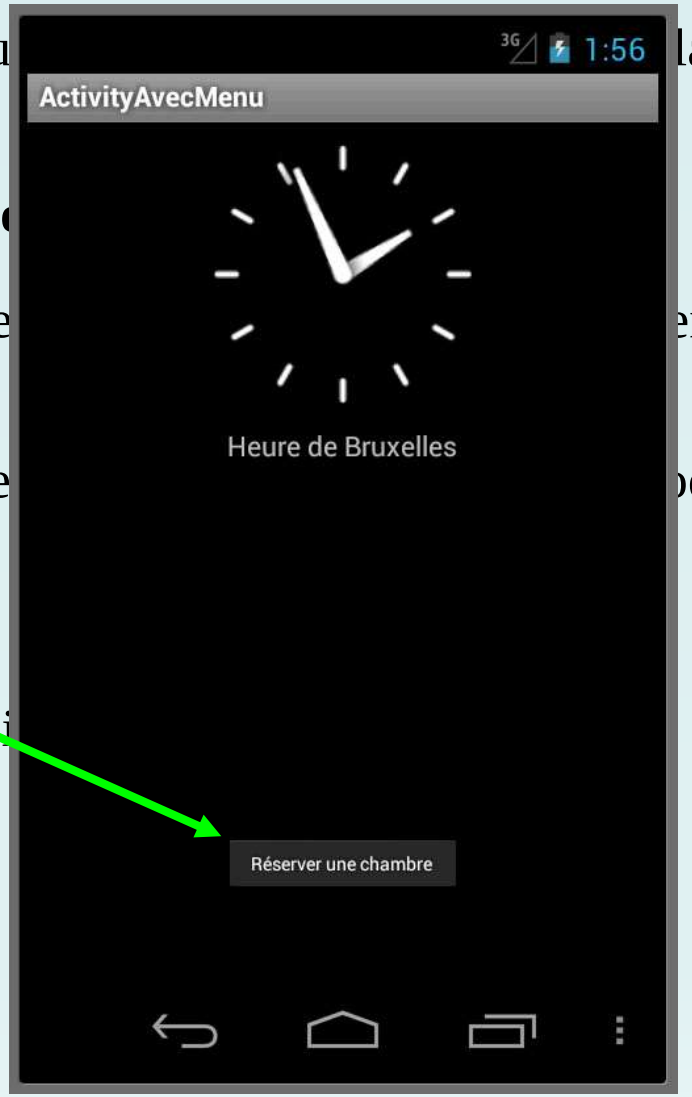
pu elect

qui est a le me en paramètr

Il ne rest uel e ode de MenuIte

p

et d'encl ssoci



14. Les menus (5/10)

MenuActivity.java

```
package my.applications.divers;
public class MenuActivity extends Activity
{   …
    @Override
    public boolean onOptionsItemSelected(MenuItem item)
    {
        switch (item.getItemId())
       {  case R.id.reservation: Toast.makeText(MenuActivity.this, "Reservation",
                        Toast.LENGTH_SHORT).show(); return true;
          case R.id.vol: Toast.makeText(MenuActivity.this, "Réserver un vol",
                                Toast.LENGTH_SHORT).show(); return true;
          case R.id.chambre: Toast.makeText(MenuActivity.this, "Réserver une
                                chambre", Toast.LENGTH_SHORT).show(); return true;
         case R.id.quitter: finish(); return true;
        }
        return false;
    }
} }
```

14. Les menus (6/10)

2. La barre des actions

[bref]

Une tendance actuelle est de plus en plus de remplacer le menu classique par une "**Action Bar**", soit l'équivalent du menu sous forme d'une liste d'icônes plus ou moins discrètes placées dans la coin supérieur droit de l'interface graphique.

menu avec sous menus ➔
un menu plus "linéaire"

♦ icône, petite image png placée dans le répertoire res\drawable ;

♦ attribut android:showAsAction expliquant si l'icône doit apparaître si il y a assez de place ("ifRoom") ou toujours ("always").

menu_reservation.xml

```
<menu …>
<item android:id="@+id/voyage"
    android:title="Reservations"
    android:showAsAction="ifRoom"
    android:icon="@drawable/sun" >
</item>
<item android:id="@+id/vol"
    android:title="Un vol"
    android:showAsAction="ifRoom"
    android:icon="@drawable/globe" />
<item android:id="@+id/chambre"
    android:title="Une chambre"
    android:showAsAction="ifRoom"
    android:icon="@drawable/flag" />
<item android:id="@+id/quitter"
    android:title="Quitter"
    android:showAsAction="ifRoom"
    android:icon="@drawable/cloud" />
</menu>
```

14. Les menus (7/10)

MenuActivity.java

```
package my.applications.divers;
public class MenuActivity extends Activity
{    …
    @Override
    public boolean onOptionsItemSelected(MenuItem item)
    {
        switch (item.getItemId())
        {
            case R.id.voyage: Toast.makeText(MenuActivity.this, "Reserver un voyage",
                        Toast.LENGTH_SHORT).show(); return true;
            case R.id.vol: Toast.makeText(MenuActivity.this, "Réserver un vol",
                        Toast.LENGTH_SHORT).show(); return true;
            case R.id.chambre: Toast.makeText(MenuActivity.this, "Réserver une
                        chambre", Toast.LENGTH_SHORT).show(); return true;
          case R.id.quitter: finish(); return true;
        }
        return false;}
}
```

Bref, pas de grand changement - mais le résultat ?

14. Les menus (8/10)

☹ Les icônes apparaissent, mais dans un menu conventionnel !

? En fait, les barres d'action n'existent que depuis la version Honeycomb d'Android (donc 3.0 = API 11).

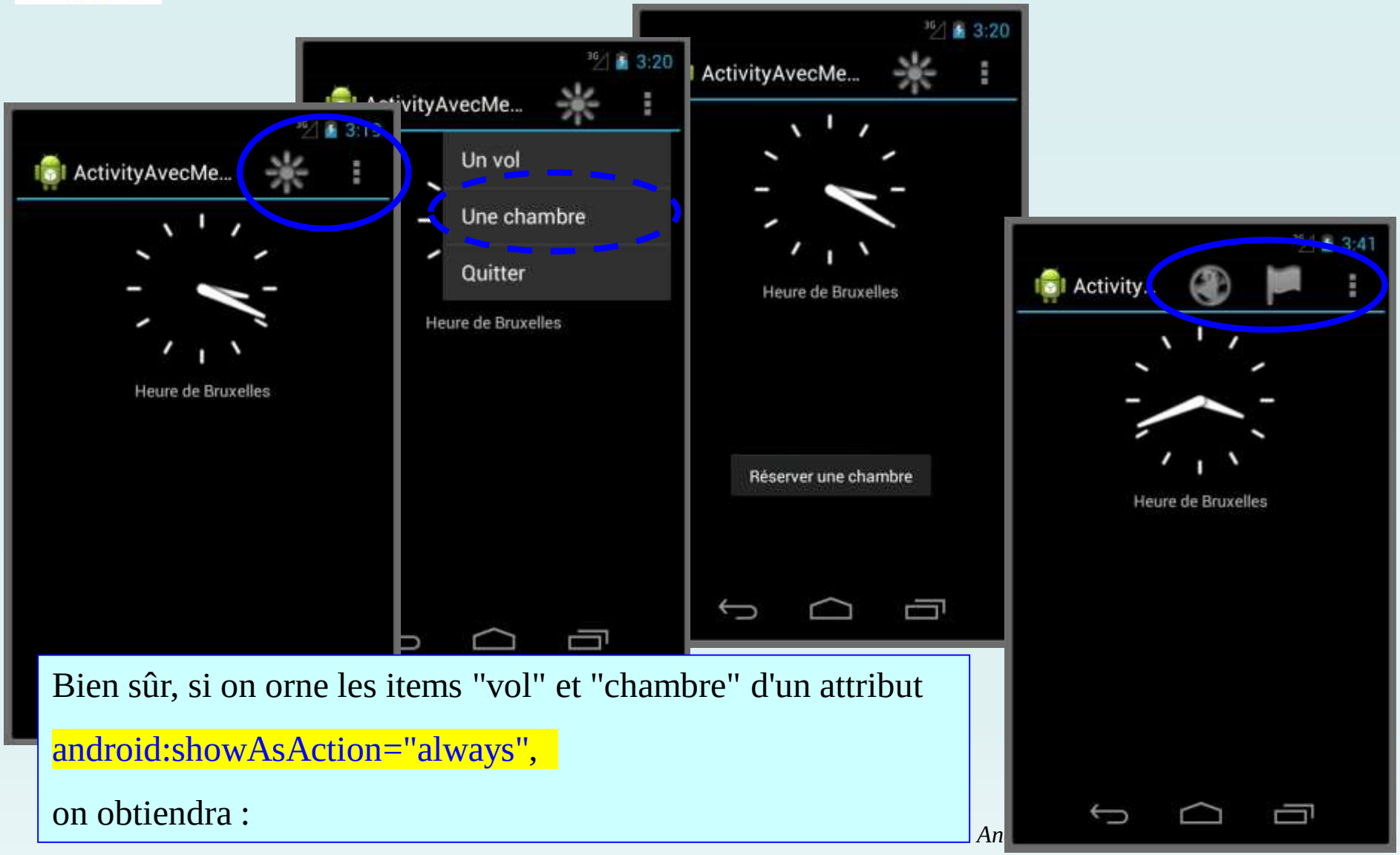
Et donc, comme souvent dans les applications Android, il faut écrire dans le fichier **AndroidManifest** le fait que l'on travaille au moins dans cette version, soit quelque chose de ce genre :

14. Les menus (9/10)

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="applics.basics"
    android:versionCode="1"
    android:versionName="1.0">
  <application android:icon="@drawable/icon"
               android:label="@string/app_name">
    <activity android:name=". MenuActivity"
          android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
  <uses-sdk android:minSdkVersion="4" android:targetSdkVersion="11"/>
</manifest>
```

14. Les menus (10/10)



Bien sûr, si on orne les items "vol" et "chambre" d'un attribut android:showAsAction="always",

on obtiendra :

An

15. Une application GUI simple (1/10)

Une simplissime application Android destinée aux médecins effectuant leurs consultations et souhaitant garder une trace de leurs contacts avec leurs patients

La structure du GUI est assez simple mais met en action différents layouts et contrôles :

15. Une application GUI simple (2/10)

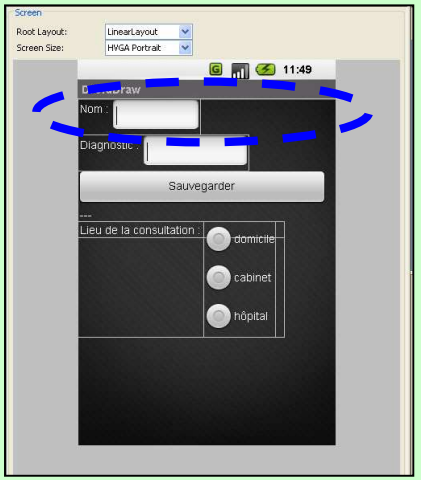
après éventuellement des rectifications manuelles dans le fichier XML ;-) !

➔

15. Une application GUI simple (3/10)

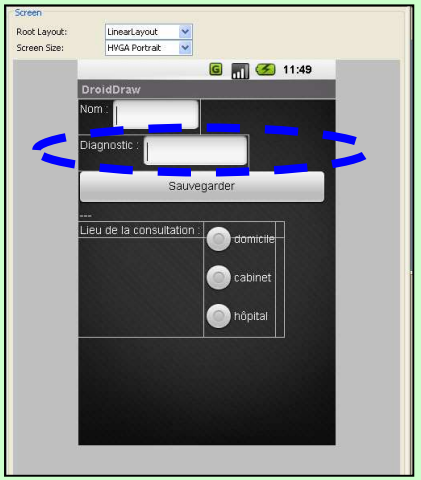
main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
        android:id="@+id/widget32" …
        android:orientation="vertical"
        xmlns:android="http://schemas.android.com/apk/res/android">
<LinearLayout
        android:id="@+id/widget33"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
<TextView
        android:id="@+id/widget35"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Nom :" />
<EditText
        android:id="@+id/nom"
        android:layout_width="110dp"
        android:layout_height="wrap_content"
        android:textSize="18sp" />
</LinearLayout>
```



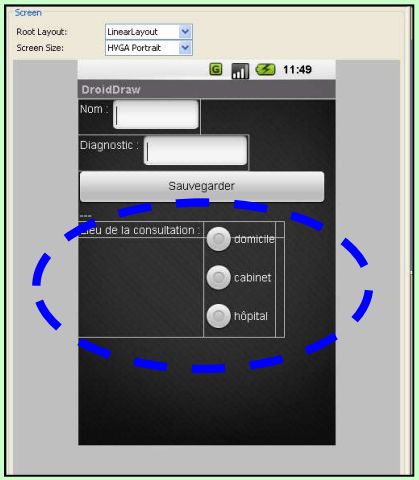
15. Une application GUI simple (4/10)

```
<LinearLayout
        android:id="@+id/widget43"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
<TextView
        android:id="@+id/widget44"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Diagnostic :" />
<EditText
        android:id="@+id/diagnostic"
        android:layout_width="132dp"
        android:layout_height="wrap_content"
        android:textSize="18sp" />
</LinearLayout>
<Button
        android:id="@+id/sauver"
        android:layout_width="309dp"
        android:layout_height="wrap_content"
        android:text="Sauvegarder" />
```



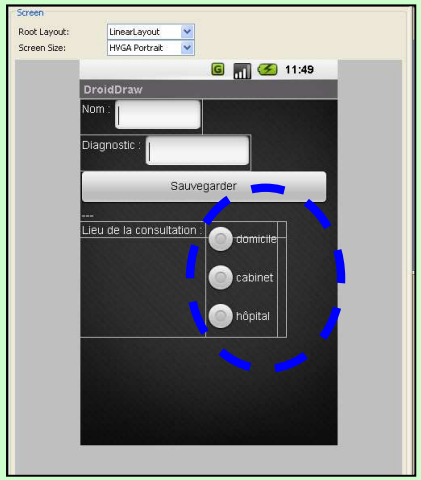
15. Une application GUI simple (5/10)

```
<TextView
        android:id="@+id/indicateur"
        android:layout_width="303dp"
        android:layout_height="wrap_content"
        android:text="---" />
<TableLayout
        android:id="@+id/widget45"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:stretchColumns="1"
        android:shrinkColumns="1">
<TableRow
        android:id="@+id/widget91"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">
<TextView …      android:text="Lieu de la consultation :" />
<RadioGroup    … android:id="@+id/endroit" … />
```



15. Une application GUI simple (6/10)

```
<RadioGroup
        android:id="@+id/endroit"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical">
<RadioButton
        android:id="@+id/domicile"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="domicile" />
<RadioButton
        android:id="@+id/cabinet" … />
<RadioButton … />
</RadioGroup>

</TableRow>
</TableLayout>
</LinearLayout>
```



15. Une application GUI simple (7/10)

R.java

```
package my.mesapplications.medecin;
public final class R
{  …
   public static final class id {
      public static final int consultations=0x7f04000e;
      public static final int diagnostic=0x7f040005;
      …
      public static final int indicateur=0x7f04000d;
      public static final int nom=0x7f040003;
      public static final int sauver=0x7f04000c; …
   }
   public static final class layout {
      public static final int main=0x7f020000;
   }
   public static final class string {
      public static final int app_name=0x7f030000;
   }
}
```

15. Une application GUI simple (8/10)

Consultation.java

```
package my.mesapplications.medecin;
public class Consultation
{
    private String nomPatient;
    private String diagnostic;
    public Consultation()
    {
        nomPatient = null; diagnostic = null;
    }
    public String getNomPatient() { return nomPatient; }
    public void setNomPatient(String nomPatient) { this.nomPatient = nomPatient; }
     …
    @Override
    public String toString()
    {
        return getNomPatient() + getDiagnostic();
    }
}
```

15. Une application GUI simple (9/10)

MainActivity.java

```
package my.mesapplications.medecin;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
…
public class MainActivity extends Activity
{
   Consultation laConsultation = new Consultation();

   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);

      Button sauvegarde = (Button) this.findViewById(R.id.sauver);
      sauvegarde.setOnClickListener(onSauvegarde);
   }
```

Le traitement de l'événement sera assuré par un listener à la volée style "Android-preferred" :

15. Une application GUI simple (10/10)

```
private View.OnClickListener onSauvegarde = new View.OnClickListener()
    {
        public void onClick(View view)
        {
            EditText nom = (EditText) findViewById(R.id.nom);
            EditText diagnostic = (EditText) findViewById(R.id.diagnostic);

            laConsultation.setNomPatient(nom.getText().toString());
            laConsultation.setDiagnostic(diagnostic.getText().toString());

            TextView sauvetageFait = (TextView) findViewById(R.id.indicateur);
            sauvetageFait.setText("Sauvetage effectué !!"); //plutôt "à effectuer" ;-)
        }
     };
}
```

Bien sûr, créer un objet Consultation ne présente guère d'intérêt : il faudrait au moins mémoriser les consultations et les afficher dans une liste, et même aussi rendre cette liste persistante.

Rappel: pour Swing : M +UI component = contrôleur / vue regroupés => 2 couches
Android sépare bien les 3 couches.

10. Les listes et les adapters (1/7)

Idée : ajouter une boîte de liste qui contiendra les consultations.

Un tel widget est classiquement une **ListView**. (Equivalent à Jlist)

Ce genre de composant est en fait géré selon le **framework MVC**, tout comme ses homologues JList et JComboBox de Swing. Et même "mieux" puisque les trois composants MVC existent de manière distincte.

Pour programmer l'ajout dynamique d'éléments à une liste, il nous faudra :

♦ évidemment une instance de **ListView**, dont les caractéristiques visuelles sont toujours définies dans main.xml et qui sera la **Vue**;

♦ un container mémoire du type **List**<?> qui contiendra les données à afficher et/ou à sélectionner : bien sûr, il sera le **M**odèle;

♦ un objet "***adapter***", en fait un Contrôleur, responsable de la synchronisation entre les données et leur vue.

Adapter en Android est différent de Adapter en Java Classique. En Java, Adapter implémente un WindowAdapterqui implémente un WindowListener (7 méthodes à redéfinir)

Un observer = un objet qui doit être averti des modifications dans les données

16. Les listes et les adapters (2/7)

Un tel ***adapter*** implémente l'interface android.widget.**Adapter**, qui déclare des méthodes comme :
public abstract Object **getItem** (int position)
ou public abstract void **registerDataSetObserver** (DataSetObserver observer)

Un DataSetObserver est un objet averti des modifications dans les données

Adaptée aux ListViews

La classe usuelle utilisée comme adapter est android.widget.**ArrayAdapter**<T>,

qui représente une version d'adapter plus particulièrement adaptée aux widgets de type ListView (elle implémente d'ailleurs l'interface android.widget.ListAdapter).

Un constructeur classique est

public **ArrayAdapter** (Context context, int textViewResourceId, List<T> objects)

où le 1er paramètre est habituellement l'activité hôte, le 3ème désigne le modèle et le 2ème vaut classiquement

simple_list_item_1

ce qui signifie que les données du modèle, qui seront transformées par défaut par l'adapter en chaînes de caractères (par appel de toString()), seront utilisées pour initialiser des TextView qui seront en définitive ajoutées à la ListView.

16. Les listes et les adapters (3/7)

MainActivity.java

```
package my.mesapplications.medecin;
…
import java.util.ArrayList; …
public class MainActivity extends Activity
{
   List<Consultation> modeleConsultations = new ArrayList<Consultation>();
   ArrayAdapter<Consultation> controleurConsultations  = null;

   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);setContentView(R.layout.main);
      controleurConsultations = new ArrayAdapter<Consultation>
            (this, android.R.layout.simple_list_item_1, modeleConsultations );
      ListView vueConsultations = (ListView) findViewById(R.id.consultations);
      vueConsultations.setAdapter(controleurConsultations);
      Button sauvegarde = (Button) this.findViewById(R.id.sauver);
      sauvegarde.setOnClickListener(onSauvegarde);
   }
```

Le contrôleur a son modèle

La vue a son contrôleur

16. Les listes et les adapters (4/7)

```
private View.OnClickListener onSauvegarde = new View.OnClickListener()
    {
     public void onClick(View view)
     {
        EditText nom = (EditText) findViewById(R.id.nom);
        EditText diagnostic = (EditText) findViewById(R.id.diagnostic);

        Consultation laConsultation = new Consultation();
        laConsultation.setNomPatient(nom.getText().toString());
        laConsultation.setDiagnostic(diagnostic.getText().toString());
        controleurConsultations.add(laConsultation);
        controleurConsultations.notifyDataSetChanged();
        TextView sauvetageFait = (TextView) findViewById(R.id
        sauvetageFait.setText("Sauvetage effectué !!");
     }
    };
}
```

On notifie les observers **idéalement** après que tous les changements aient été effectués => fonctionnement à l'économie

signale aux observers que les data ont été modifiées et que toutes les vues intéressées doivent se mettre à jour

16. Les listes et les adapters (5/7)

Nous avons donc ajouté une ListView dans notre GUI

MAIS se pose alors un problème : notre liste et nos composants graphiques doivent occuper la place maximale sans occulter les autres contrôles graphiques. Pour obtenir cela, on peut utiliser un **ScrollLayout** pour les composants autres que la ListView (qui, elle, utilise un scroll natif) ou une **ScrollView**.

main.xml

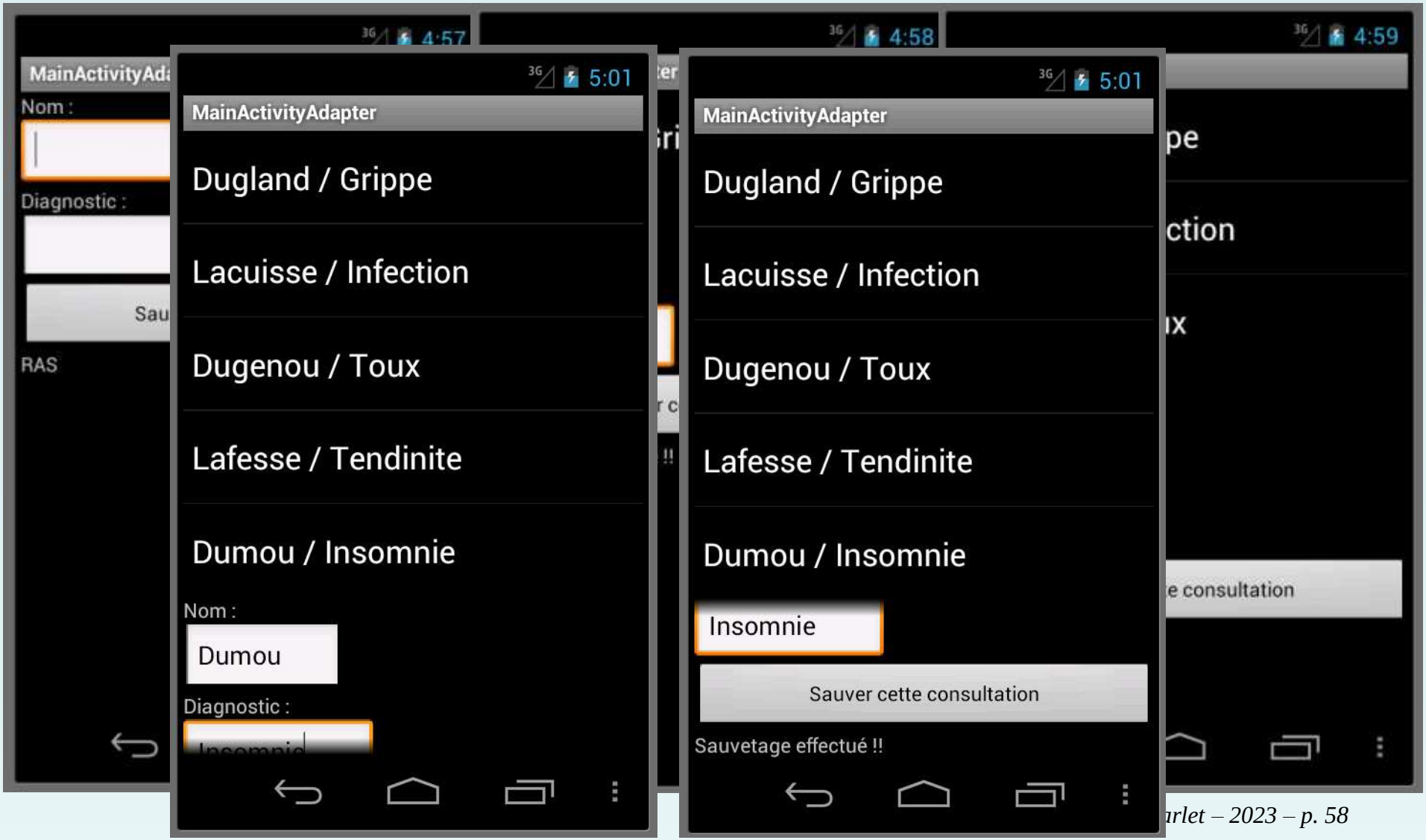
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/gui"
    …
    tools:context=".MainActivity" >

    <ListView
            android:id="@+id/consultations"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_alignParentTop="true" />
```

namespace XML – l'attribut "context" désigne à quelle activité associer l'élément XML racine du layout

16. Les listes et les adapters (6/7)

```
    <ScrollView
            android:id="@+id/widget32"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            xmlns:android="http://schemas.android.com/apk/res/android">
      <LinearLayout …>
        <TextView … android:text="Nom :" />
        <EditText android:id="@+id/nom" …/>
        <TextView … android:text="Diagnostic :" />
        <EditText android:id="@+id/diagnostic" … />
        <Button android:id="@+id/sauver" ….. android:text="Sauver cette
            consultation"/>
        <TextView android:id="@+id/indicateur"… android:text="RAS"/>
       </LinearLayout>
    </ScrollView>
</LinearLayout>
```

16. Les listes et les adapters (7/7)



17. Les communications réseaux (1/10)

Les communications réseaux sous Android nous sont bien connues ! En effet, **elles utilisent *les classes sockets de la programmation réseau habituelle sous Java***, comme la classe Socket par exemple

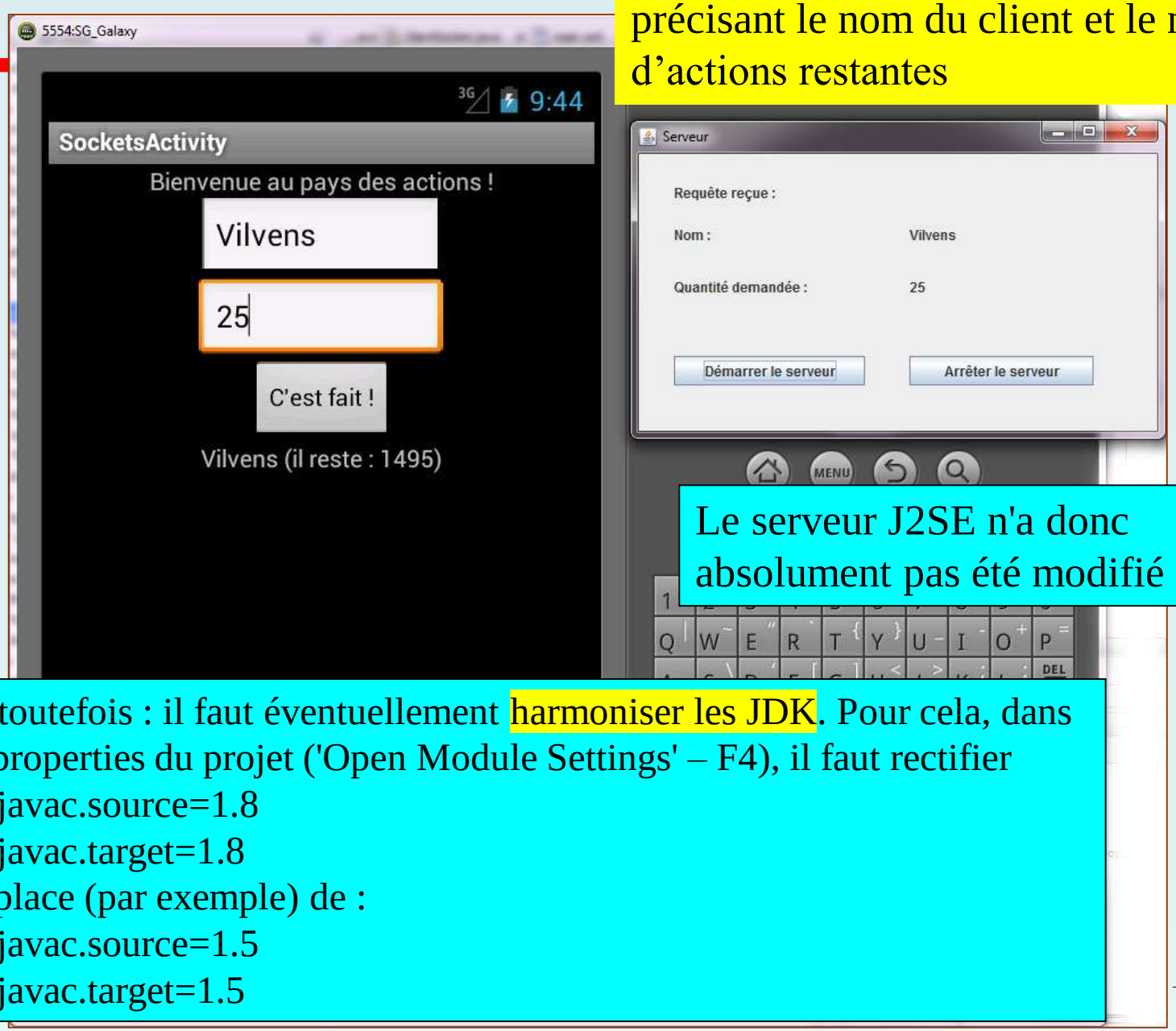
Une petite condition pour que notre application puisque accéder au réseau : il faut **ajouter dans le fichier manifeste une clause autorisant cet accès** :

AndoidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="applics.basics"
            …
  <application android:icon="@drawable/icon" android:label="@string/app_name">
    <activity android:name=".HelloBonjour"
            android:label="@string/app_name">
                    …
    </activity>
  </application>
            …
   <uses-permission android:name="android.permission.INTERNET"></uses-permission>
</manifest>
```

17. Les communications

Le client vend des actions en se connectant au serveur qui répond en précisant le nom du client et le nbr d'actions restantes



Le serveur J2SE n'a donc absolument pas été modifié ☺ !

Attention toutefois : il faut éventuellement harmoniser les JDK. Pour cela, dans le fichier properties du projet ('Open Module Settings' – F4), il faut rectifier

javac.source=1.8
javac.target=1.8

en lieu et place (par exemple) de :

javac.source=1.5
javac.target=1.5

17. Les communications réseaux (3/10)

main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout …>
<TextView … android:text="Bienvenue au pays des actions !"
        android:layout_gravity="center_horizontal" />
<EditText android:id="@+id/EditTextUser" …/>
<EditText android:id="@+id/EditTextCombien" …. />
<Button android:id="@+id/ButtonEnvoi" … Android:text="C&apos;est fait !" />
<TextView
        android:id="@+id/TextViewReponse" …
        android:text="Voyons voir ..."
        android:layout_gravity="center_horizontal" />
<ImageView
        android:id="@+id/IVOk"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:visibility="invisible"
        android:layout_gravity="center_horizontal" />
</LinearLayout>
```

Une image apparaitra lorsque la connexion aura été effectuée.

17. Les communications réseaux (4/10)

SocketsActvity.java

```
package mesapplications.network;
import java.io.*; …
public class SocketsActvity extends Activity implements OnClickListener
{
        Button BLogin;
        EditText ETLogin; EditText ETCombien;
        TextView TVReponse;
        @Override
        public void onCreate(Bundle savedInstanceState)
        {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);
            BLogin = (Button)this.findViewById(R.id.ButtonEnvoi);
            BLogin.setOnClickListener(this);
            ETLogin = (EditText)this.findViewById(R.id.EditTextUser);
            ETCombien = (EditText)this.findViewById(R.id.EditTextCombien);
            TVReponse = (TextView)this.findViewById(R.id.TextViewReponse);
        }
```

17. Les communications réseaux (5/10)

```
@Override
public void onClick(View arg0)
{
        String nom = (String) ETLogin.getText().toString();
        contacteServeur(nom);
        BLogin.setText("C'est fait !");
}

private void contacteServeur(String n)
{
        Socket cliSock=null;
        DataInputStream dis = null; DataOutputStream dos = null;
        try
        {
           cliSock = new Socket(InetAddress.getByName("192.168.1.5"),
                   50000);
        }
        catch (UnknownHostException e) …
```

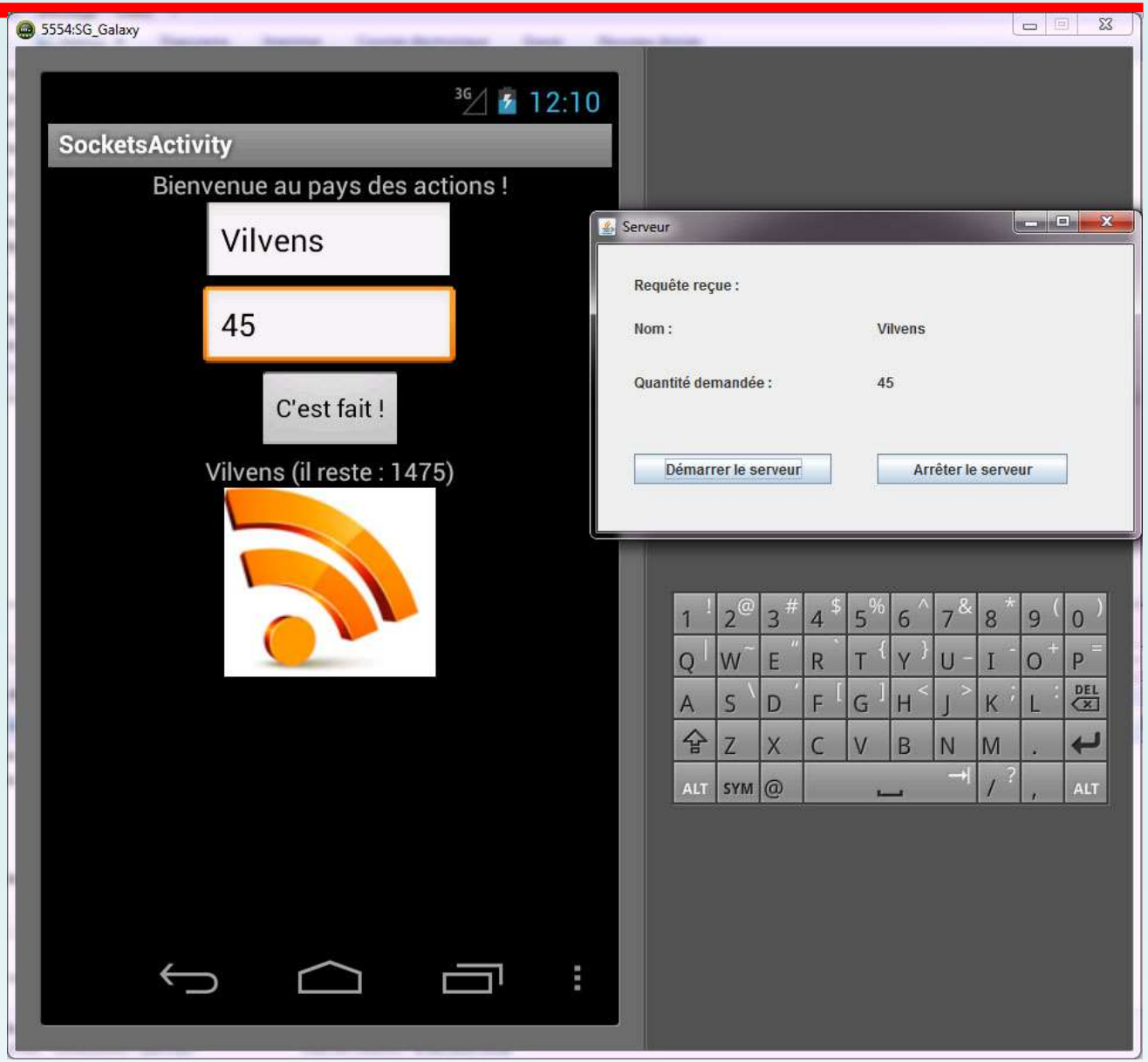
attention au localhost (thread nécessaire)

17. Les communications réseaux (6/10)

```
try
{  dis = new DataInputStream(cliSock.getInputStream());
   dos = new DataOutputStream(cliSock.getOutputStream());
}
catch (IOException e)      {  }
String outNom = n; int outQuantite = q;
String reponse = null;  int inQuantiteRestante = 0;
if (!outNom.equals("") && outQuantite>0 && dos!=null && dis!=null)
{          try
           {    dos.writeUTF(outNom);dos.writeInt(outQuantite);
                reponse = dis.readUTF(); inQuantiteRestante = dis.readInt();
                TVReponse.setText(reponse + " (il reste : " +
                                                inQuantiteRestante+ ")");
           }
           catch (IOException e) { }
           finally { try  { dis.close();dos.close(); }
                     catch (IOException e)  { e.printStackTrace(); }}
} } }
```

17. Les communications réseaux (7/10)

On peut même imaginer de faire apparaître une image illustrant le fait que la transaction réseau a réussi et qui n'apparaîtrait qu'après la réception de la réponse du serveur :



17. Les communications réseaux (8/10)

♦ ajouter l'image en question (reseau.png) dans le répertoire drawable du répertoire res de notre projet :

♦ modifier le layout avec une ImageView, rendue provisoirement invisible :

main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout …>
…
<ImageView
        android:id="@+id/IVOk"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:visibility="invisible"
        android:layout_gravity="center_horizontal" />
</LinearLayout>
```

♦ ajouter au code de la méthode contacteServeur() les instructions d'affichage :

17. Les communications réseaux (9/10)

SocketsActvity.java

```
package mesapplications.network;
import java.io.*; …
public class SocketsActvity extends Activity implements OnClickListener
{
        EditText ETLogin; EditText ETCombien;TextView TVReponse;
        @Override
        public void onCreate(Bundle savedInstanceState) { … }

        @Override
        public void onClick(View arg0)
        {
                …
                contacteServeur(nom, quantite);
                BLogin.setText("C'est fait !");
        }

        private void contacteServeur(String n, int q)
        { … ➔
```

17. Les communications réseaux (10/10)

```
        private void contacteServeur(String n, int q)
        {
                                                    ') && outQuantite>0 && dos!=null && dis!=null)
                Resources res =this.getResources();
                Drawable draw = res.getDrawable(R.drawable.reseau);
                ImageView img = (ImageView) this.findViewById(R.id.IVOk);
                img.setImageDrawable(draw);
                img.setVisibility(View.VISIBLE);
                        …
                }
        }
}
```

res (Objet Ressources) :
contient toutes les ressources
« mapper » dans le fichier .arsc

On recherche l'image dans les ressources
L'extension du fichier n'est pas précisée.

On lie l'image à l'imageView
et on la rend visible

Img n'est que l'ImageView = l'endroit où l'mage doit être affichée

18. Threads asynchrones et GUIs (1/9)

Notre pratique des applications installées nous rend bien conscients que l'on a intérêt à **threader les opérations réseau** afin de ne pas bloquer l'interface graphique.

➔ on décharge le thread principal (appelé **UI Thread**) de la surveillance du réseau (par exemple), car celui-ci exécute déjà le code de l'Activity, gère les interactions avec l'utilisateur et régit l'affichage.

classe "*helper*" dédiée à ce genre de travail : **AsyncTask**

= classe fournie pour faciliter la manipulation de threads au comportement stéréotypé.

= permet de **créer une tâche qui va s'effectuer en arrière-plan et qui sera capable de communiquer sur son état d'avancement**.

La classe est paramétrable :

AsyncTask<Params, Progress, Result>

type des paramètres envoyés à la tâche

type du résultat envoyé par la tâche

le type de l'information envoyée par le thread au UI thread pour que ce dernier puisse rendre compte de l'état d'avancement;

L'un ou l'autre de ces paramètres peut être Void si il n'a pas d'usage.

18. Threads asynchrones et GUIs (2/9)

Le cycle de vie d'un Asynctask comporte **4 états**, associés à **4 méthodes**.

1) Concrètement, un tel objet doit toujours implémenter la méthode **doInBackground**(). **C'est là que sera réalisé le traitement lourd de manière asynchrone dans un thread implicite séparé**. C'est là aussi que peut être appelée la méthode **publishProgress**() qui permet la mise à jour de la progression.

2) Les méthodes **onPreExecute**() (appelée avant le traitement - exemple typique : afficher une barre de progression),

3) **onProgressUpdate**() (appelée en réponse à publishProgress() pour afficher la progression par le UI thread (on ne doit donc pas appeler la méthode onProgressUpdate() directement) et

4) **onPostExecute**() (appelée après le traitement)

sont optionnelles.

Ces trois méthodes s'exécutent depuis l'UI Thread : elles doivent en effet pouvoir modifier l'interface. Il faudra donc éviter d'y programmer des traitements lourds.

Supposons que nous voulions réaliser l'exemple suivant :

18. Threads asynchrones et GUIs (3/9)

18. Threads asynchrones et GUIs (4/9)

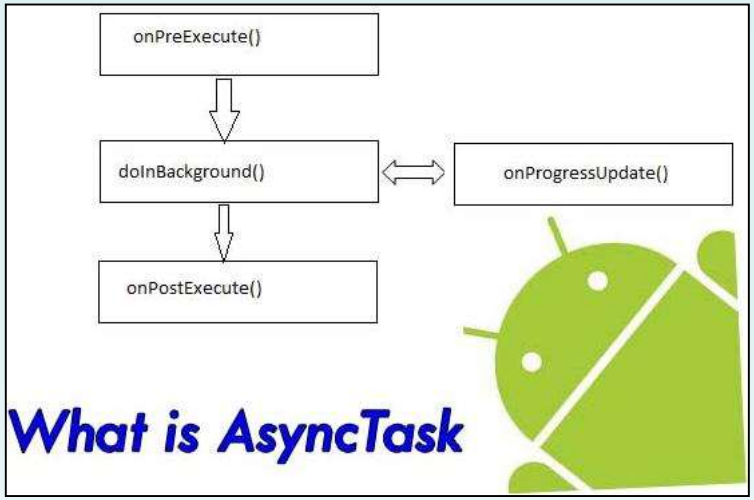
Nous allons donc logiquement utilisé le widget **ProgressBar** (package android.widget) dont une méthode utilisée ici sera :

public synchronized void **setProgress** (int progress)

pour mettre à jour la progression affichée.

Notre activité sera alors simple à programmer : **elle utilisera une classe privée imbriquée dérivée d'Asynctask dont elle fera démarrer une instance avec la méthode**

public final AsyncTask<Params, Progress, Result> **execute** (Params... params)



18. Threads asynchrones et GUIs (5/9)

L'interface graphique :

main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=http://schemas.android.com/apk/res/android …>

   <Button … android:id="@+id/lancer" … android:text="Lancer la tâche" />
   <TextView … android:text="Progression de la tâche asynchrone:" />

   <ProgressBar
      android:id="@+id/barre_progression"
      style="?android:attr/progressBarStyleHorizontal"
      android:layout_width="fill_parent"
      android:layout_margin="10dp"
      android:layout_height="wrap_content" />

   <TextView … android:text="Et à part ça ? Quoi de neuf ?" …/>
   <EditText …    android:id="@+id/EditTextUser" … />
   <Button android:id="@+id/ButtonEnvoi"… />
</LinearLayout>
```

Pointe un thème dans la libraire du SDK

18. Threads asynchrones et GUIs (6/9)

ActivityAvecAsynctask.java

```
package my.mesapplications.async;

import android.app.Activity;
import android.os.AsyncTask;
…

public class ActivityAvecAsynctask extends Activity
{
        private ProgressBar barre;
        private Button boutonDemarre;

        @Override
        public void onCreate(Bundle savedInstanceState)
        {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.main);

                barre = (ProgressBar) findViewById(R.id.barre_progression);
                boutonDemarre = (Button) findViewById(R.id.lancer);
```

18. Threads asynchrones et GUIs (7/9)

```
                    boutonDemarre.setOnClickListener(new OnClickListener()
                      {         @Override
                                public void onClick(View arg0)
                                {
                                  TraitementLourd calcul=new TraitementLourd();
                                  calcul.execute();
                                }});
            }
/* ----------------------------------------------------------------------------------*/
      private class TraitementLourd extends AsyncTask<Void, Integer, Void>
      {
         @Override
         protected void onPreExecute()
         {
           super.onPreExecute();
           Toast.makeText(getApplicationContext(), "Début traitement asynchrone",
                       Toast.LENGTH_LONG).show();
         }
```

18. Threads asynchrones et GUIs (8/9)

```
@Override
protected void onProgressUpdate(Integer... values)
{
  super.onProgressUpdate(values);
  barre.setProgress(values[0]);
}
@Override
protected void doInBackground(Void... arg0)
{
  int progress;
  for (progress=0;progress<=100;progress++)
  {
    for (int i=0; i<1000000; i++){} // ex: résolution d'un grand système d'équations ;-)
    publishProgress(progress);
    progress++;
  }
  return null;
}
```

liste à arguments variables du C ;-) ?

Partie qui prend du temps
On récupère l'information (progression)

18. Threads asynchrones et GUIs (9/9)

```
        @Override
        protected void onPostExecute(Void result)
        {
          Toast.makeText(getApplicationContext(), "Traitement terminé !",
                    Toast.LENGTH_LONG).show();
        }
    }
}
```

On aura aussi remarqué l'utilisation

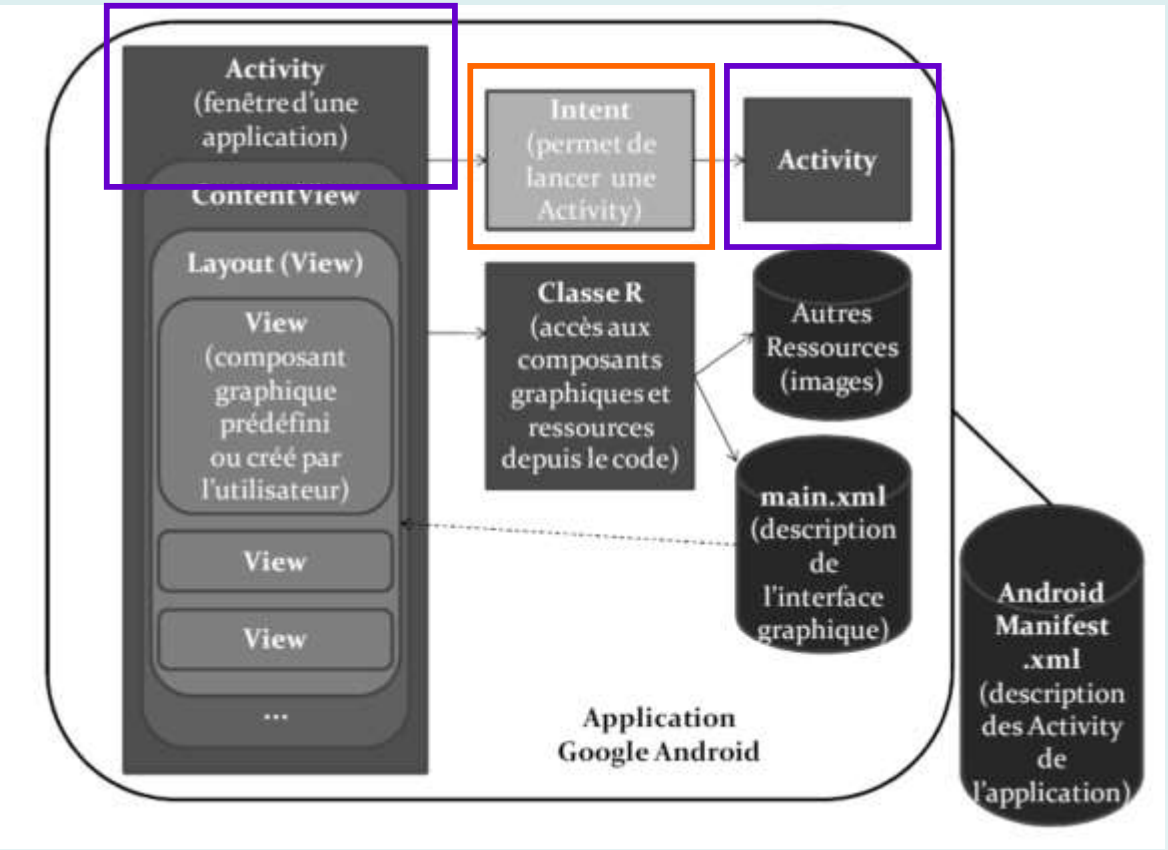
♦de la classe **Void**, qui permet de manipuler une référence quelconque

♦ainsi que celle d'une liste de paramètres indéterminés spécifiés par l'**ellipse** "…" (ce qui a le désavantage de ne pas permettre le contrôle des types de ces paramètres).

19. Les intents (1/19)

1. Application, activités et intents

Une application Android apparaît classiquement comme structurée selon ce schéma :



Autrement dit, une application Android est loin d'être monolithique :

elle est formée d'activités (et/ou de services) qui dialoguent entre eux au moyen de messages appelés "intents".

19. Les intents (2/19)

2. Un message asynchrone

La politique de sécurité d'Android est modélisée selon le principe de la **sandbox** bien connue en Java : les applications sont presque totalement séparées les unes des autres, garantissant ainsi qu'une application ne peut mettre en péril l'ensemble du système.

"***Presque totalement***" : parce qu'il faut tout de même bien de temps en temps qu'un composant applicatif communique avec un autre : c'est là qu'interviennent les **intent**s.

> Un **intent** ("**intention**") est un message asynchrone spécifiant une action et contenant l'URI (Uniform Ressource Identifier) du composant à manipuler ou de l'action à initier.

On peut donc encore voir un intent comme **un ensemble de données qui peut être passé à un autre composant applicatif** (de la même application ou non) que l'on fera démarrer.

19. Les intents (3/19)

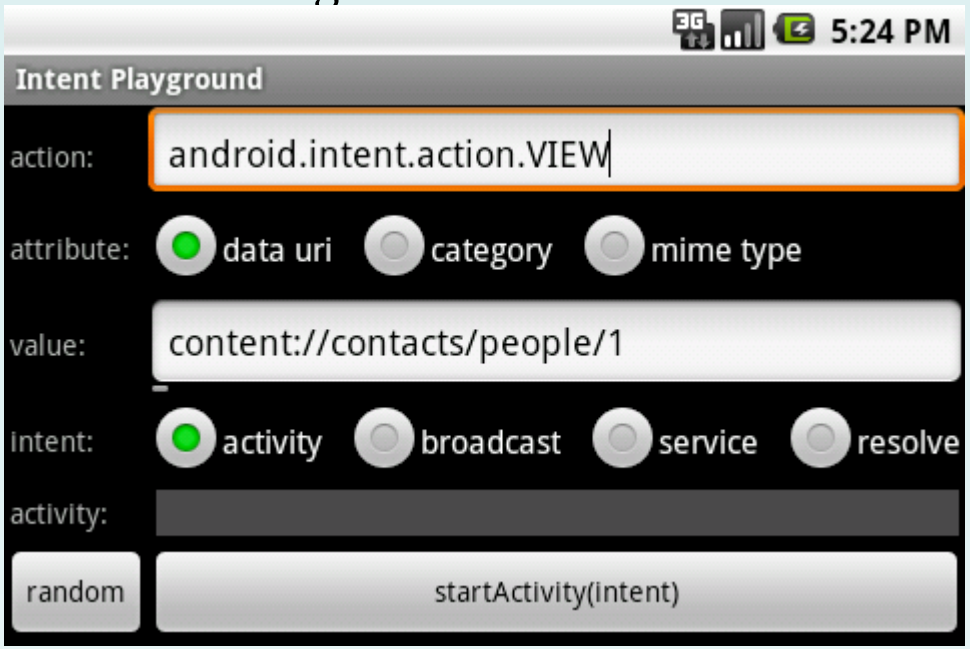
Si on reçoit une URL Webkit => on lance le browser
Si on reçoit un DIAL (n° tel) l'activity-manager analyse et lance la numérotation

Un tel "message" peut être envoyé de deux façons :

♦ soit en ciblant un composant précis d'une application (on parle alors de mode ***explicite***),

♦ soit en laissant le système déléguer le traitement (ou non) de cette demande au composant le plus approprié (on parle alors de mode ***implicite***) : c'est l'objet associé de par sa nature à l'action qui recevra le message.

Un système de filtres permet à chaque application de filtrer et de gérer uniquement les **Intents** qui sont pertinents pour celle-ci.
Une application peut ainsi être dans un état d'inactivité tout en restant à l'écoute des intents circulant dans le système.



19. Les intents (4/19)

2. Démarrage explicite

Du point de vue programmation, un intent est une Instance de la classe **Intent** (package android.content). Parmi les divers constructeurs possibles, celui qui permet d'envoyer un intent à un composant précis est :

public **Intent** (Context packageContext, **Class<?>** cls)

donc "n'importe quoi" ;-)

♦ **Context** est une classe abstraite dont l'implémentation est fournie par Android (typiquement désigné par le champ this de la classe appelante);

♦ **Class** est évidemment **l'activité que l'on veut faire démarrer au sein de la même application**.

Donc, si une application est composée de plusieurs écrans qui s'enchaînent les uns à la suite des autres en fonction des actions de l'utilisateur,

le passage de l'un à l'autre se fera par Intent : **startActivity** (**Intent** intent)

chaque écran est représenté par une activité, définissant son interface utilisateur et sa logique, et donc chaque activité de l'application nécessitera l'emploi d'un Intent pour être démarrée.

19. Les intents (5/19)

Donc, dans la version la plus simple :

```
Intent msgDemarrage = new Intent(ActiviteAppellante.this, ActiviteAppelee.class);
startActivity(msgDemarrage);
```

Un exemple classique d'utilisation de ce procédé est celui d'une application composée de plusieurs écrans qui s'enchaînent les uns à la suite des autres en fonction de l'action de l'utilisateur : ***chaque écran est associé à une activité avec son interface utilisateur et sa logique tandis que le passage de l'un à l'autre se fera par Intent***.

*Un Intent peut **contenir la référence d'un objet** à condition que celui-ci implémente l'interface **Parcelable** (package android.os).* *(Parcelable = mis sous forme d'un colis)*

On peut alors ajouter des informations à un **Intent** au moyen de la méthode polymorphe :

public Intent **putExtra** (String name, int value)
public Intent **putExtra** (String name, String value)
…
le premier paramètre étant une **clé** et le deuxième la **valeur** associée à la clé selon le principe d'une hashtable.

19. Les intents (6/19)

Encore mieux : on peut ajouter toute une hashtable à un Intent avec

public Intent **putExtras** (Bundle extras)

où un objet **Bundle** (package android.os) implémente un mécanisme clé-valeur comme celui des hashtable, avec des méthodes :

public void **putInt** (String key, int value)
public void **putFloat** (String key, float value)
public void **putString** (String key, String value) …

Les méthodes **getExtras**(), **getIntExtra**(String key), **getStringExtra**(String key), … permettront évidemment de récupérer ces données.

On l'a dit, l'objet Intent étant ainsi créé et configuré, une activité cible sera démarrée de manière inchangée par **startActivity** (Intent intent).

19. Les intents (7/19)

On se souviendra qu'un système de filtres permet à chaque application de filtrer et de gérer uniquement les Intents qui l'intéressent. Ces filtres sont spécifiés dans le fichier **AndroidManifest.xml** au moyen d'une balise <intent-filter> au sein de la balise <activity> - par exemple :

AndroidManifest.xml

```
<activity android:name="…"
          android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.intent.action.DEFAULT" />
    </intent-filter>
</activity>
```

Il est clairement fait allusion à une constante :

public static final String **ACTION_DEFAULT**

qui est en fait un synonyme de

public static final String **ACTION_VIEW** = "android.intent.action.**VIEW**"

dont la signification est de provoquer l'affichage de l'interface de l'activité (donc de visualiser les données à l'utilisateur de l'application).

19. Les intents (8/19)

On pourra comparer avec :

AndroidManifest.xml

```
<activity android:name=".HelloBonjour" android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
</activity>
```

où l'on utilise les constantes :

- public static final String **ACTION_MAIN** = "android.intent.action.**MAIN**"

qui spécifie quelle activité est le point d'entrée de l'application (donc, pas d'action véritable);

- public static final String **CATEGORY_LAUNCHER** = "android.intent.category.**LAUNCHER**"

En fait, pour qu'une application puisse être démarrée depuis l'écran Applications d'Android, une de ses activités doit déclarer un tel "intent filter".

19. Les intents (9/19)

4. Communication entre deux activités : mode d'emploi

1) A → B : simple envoi d'un intent

Dans l'activité A :

- ♦ créer le Bundle
- ♦ créer l'Intent
- ♦ appeler putExtras() du Bundle
- ♦ appeler **startActivity**(…)

Ne pas oublier de déclarer la deuxième activité dans AndroidManifest.xml !

```
<activity android:name=".AAA"
          android:label="@string/app_name">
    <intent-filter> …</intent-filter>
</activity>
<activity android:name=".BBB"></activity>
```

Dans l'activité B :

♦ dans la méthode
public void **onCreate**(Bundle savedInstanceState) : (Dans **onResume**: si on redémarre)

♦ récupérer l'Intent qui a démarré l'activité :

public Intent **getIntent** ()

♦ en extraire le Bundle : public Bundle **getExtras**()
puis en extraire les infos contenues.

19. Les intents (10/19)

2) A ←→ B : envoi d'un intent avec attente d'une réponse

Dans l'activité A (→) :

créer le Bundle - créer l'Intent - putExtras du Bundle **MAIS** utiliser :

public void **startActivityForResult** (Intent intent, int requestCode)
(le code >=0 peut servir à la cible à identifier la source)

Dans l'activité B :

dans la méthode public void **onCreate**(Bundle savedInstanceState) :

- ♦ récupérer l'Intent : getIntent ()
- ♦ en extraire le Bundle : getExtras ()
- ♦ après analyse, envoyer un code de résultat avec

public final void **setResult** (int resultCode)

Dans l'activité A (←) : ajouter la méthode

protected void **onActivityResult** (int requestCode, int resultCode, Intent data)

dans laquelle on récupère le code renvoyé par l'activité B quand elle s'est terminée.

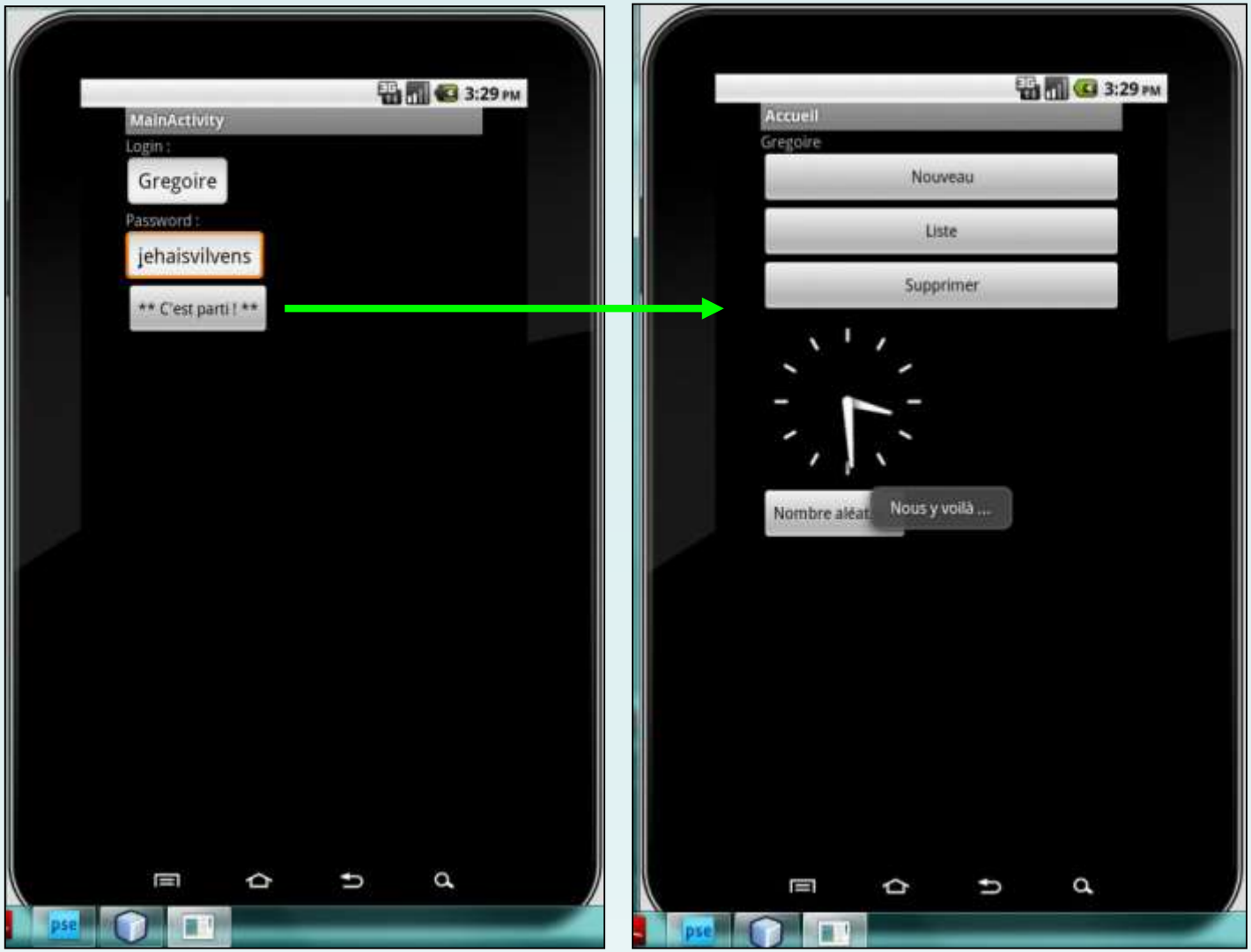
19. Les intents (11/19)

RandomActivities

5. Dialogue entre deux activités par intent

Projet = une première activité **MainActivity** demande l'introduction d'un login.

Un appui sur un bouton permet d'envoyer par intent ce login à une deuxième activité **ContactActivity** (celle-ci s'adressera ensuite à un service **RandomService** qui émet un nombre aléatoire dans un toast – voir plus loin).



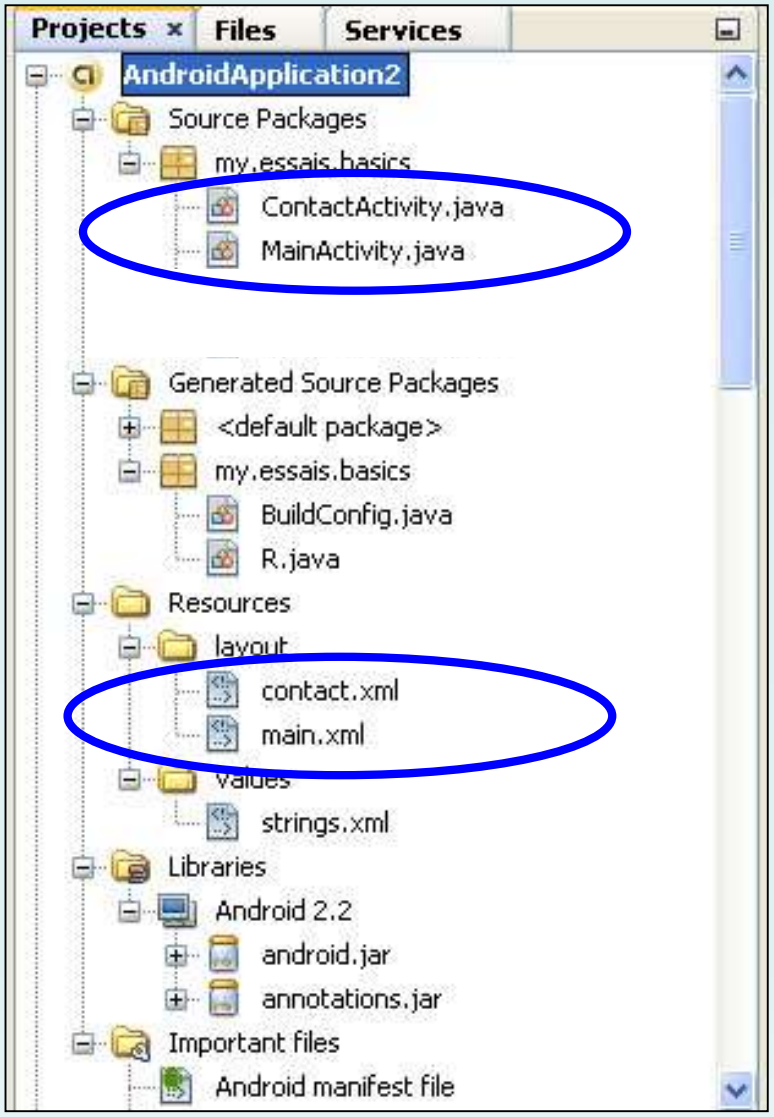
19. Les intents (12/19)

Le fichier manifeste déclare essentiellement les composants de l'application, pour l'instant les deux activités :

AndoidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="my.essais.basics" android:versionCode="1"
    android:versionName="1.0">
   <application android:label="@string/app_name" >
      <activity android:name="MainActivity" android:label="@string/app_name">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
      <activity android:name="ContactActivity"
            android:label="@string/suite_name">
         <intent-filter>
            <action android:name="android.intent.action.DEFAULT" />
            <category android:name="android.intent.category.DEFAULT" />
         </intent-filter>
      </activity>
```

19. Les intents (13/19)



Les fichiers de ressources (main.xml, contact.xml et strings.xml) ne présentent aucun intérêt particulier, si ce n'est que contact.xml déclare la méthode de traitement de l'appui sur le bouton "Nombre aléatoire" :

main.xml

```
…
<Button
        android:id="@+id/boutonNA"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Nombre aléatoire"
        android:onClick="onButtonClick" />
…
```

19. Les intents (14/19)

MainActivity.java

```
package my.essais.basics; …
public class MainActivity extends Activity
{  private EditText ZENom; private Button BLogin;
   @Override
   public void onCreate(Bundle savedInstanceState)
   {  super.onCreate(savedInstanceState);setContentView(R.layout.main);
      this.ZENom = (EditText)this.findViewById(R.id.editNom);
      BLogin = (Button)this.findViewById(R.id.boutonLogin);
      BLogin.setOnClickListener (new View.OnClickListener()
          {
              public void onClick(View v)
              {    BLogin.setText("** C'est parti ! **");
                   Intent suite = new Intent(MainActivity.this, ContactActivity.class);
                   suite.putExtra("login", ZENom.getText().toString());
                   startActivity(suite);
               }
          });
    }
```

On instancie un objet «EditText »

On récupère le contenu: getText() on convertit ensuite en chaîne de caractères avec toString()

19. Les intents (15/19)

ContactActivity.java

```
package my.essais.basics;

import android.app.Activity;
import android.content.Intent;

public class ContactActivity extends Activity
{
    private TextView login;

    @Override
    public void onCreate(Bundle icicle)
    {
        super.onCreate(icicle);
        setContentView(R.layout.contact);
        login = (TextView)this.findViewById(R.id. labelLogin );
        login.setText(this.getIntent().getExtras().get("login").toString());
    }
```

Principe des Hashables: clé

19. Les intents (16/19)

```
    @Override
    public void onStart()
    {
       super.onStart();
       login = (TextView)this.findViewById(R.id. labelLogin);
       login.setText(this.getIntent().getExtras().get("login").toString());

       Toast.makeText(this, "Nous y voilà ...", Toast.LENGTH_LONG).show();
    }

   public void onButtonClick(View v)
   {
      Button BNombre = (Button)this.findViewById(R.id.boutonNA);
      BNombre.setText("On a appuyé");
   }
}
```

(voire **onResume()** si standbye. Si on appelle plusieurs fois l'activité, on ne sait pas dans quel état est l'activité quand on revient dedans)

19. Les intents (17/19)

A suivre …

19. Les intents (18/19)

6. Démarrage implicite

Si il s'agit de **faire démarrer une autre application connue du système**, on utilisera plutôt le constructeur :

public **Intent** (String action, Uri uri)

où l'objet **URI** (du package java.net) sera probablement créé au moyen de la méthode statique de cette classe :

public static Uri **parse** (String uriString)

Le système se chargera de **résoudre l'Intent en proposant le composant de l'application le plus approprié**. Ce mécanisme permet d'éviter les dépendances vers des applications puisque **l'association entre l'application et le composant nécessaire se fait au moment de l'exécution et non de la compilation**.

exemples : ouvrir une page web ou composer un numéro pour téléphoner

19. Les intents (19/19)

On écrira donc quelque chose du genre :

```
String page = "http://www.google.be";
Intent inWeb = new Intent(Intent.ACTION_VIEW, Uri.parse(page));
startActivity(inWeb);
```

Bien sûr, on aura remarqué qu'on ne désigne plus ainsi un composant applicatif bien précis : vu la constante ACTION_VIEW, on demande simplement à Android de visualiser ce qui correspond à l'URI

➔en clair, Android va tenter de chercher une application s'étant définie comme capable de répondre à l'action ACTION_VIEW, donc probablement le browser inclus de base dans le téléphone.

L'appel est donc bien implicite : **l'application à ouvrir est déterminé selon la fonctionnalité demandée**. Autre exemple :

```
String numero_a_appeller = "tel:0485682465";
Intent inTel = new Intent(Intent.ACTION_DIAL,
Uri.parse(numero_a_appeller)); startActivity(inTel);
```

By now …

… that's all folks ;-) !

Mais il y a encore à faire …