

# Programmation mobile avec Android

Pierre Nerzic - pierre.nerzic@univ-rennes1.fr

février-mars 2016

## Abstract

Il s'agit des transparents du cours mis sous une forme plus facilement imprimable et lisible. Ces documents ne sont pas totalement libres de droits. Ce sont des supports de cours mis à votre disposition pour vos études sous la licence *Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International*.



Version du 16/05/2017 à 08:21

## Table des matières

<b>1</b>	<b>Environnement de développement</b>	<b>14</b>
1.1	Introduction . . . . .	14
1.1.1	Android . . . . .	14
1.1.2	Définition . . . . .	14
1.1.3	Composants d'Android . . . . .	16
1.1.4	Programmation d'applications . . . . .	16
1.2	SDK Android et Android Studio . . . . .	16
1.2.1	SDK et Studio . . . . .	16
1.2.2	SDK Manager . . . . .	16
1.2.3	Choix des éléments du SDK . . . . .	16
1.2.4	Dossiers du SDK . . . . .	17
1.2.5	Android Studio . . . . .	18
1.3	Première application . . . . .	18
1.3.1	Objectif de la semaine 1 . . . . .	18
1.3.2	Assistant de création d'application . . . . .	18

1.3.3	Choix de la version . . . . .	18
1.3.4	Choix de la version . . . . .	18
1.3.5	Choix du type d'application . . . . .	18
1.3.6	Points à configurer . . . . .	22
1.3.7	Noms des packages et classes . . . . .	22
1.3.8	Résultat de l'assistant . . . . .	22
1.3.9	Fenêtre du projet . . . . .	22
1.3.10	Éditeurs spécifiques . . . . .	22
1.3.11	Exemple <code>res/values/strings.xml</code> . . . . .	22
1.3.12	Exemple <code>res/layout/main.xml</code> . . . . .	25
1.3.13	Source XML sous-jacent . . . . .	25
1.3.14	Reconstruction du projet . . . . .	25
1.3.15	Gradle . . . . .	27
1.3.16	Gradle en ligne de commande . . . . .	27
1.4	Première exécution . . . . .	27
1.4.1	Exécution de l'application . . . . .	27
1.4.2	Assistant de création d'une tablette virtuelle . . . . .	27
1.4.3	Caractéristiques d'un AVD . . . . .	27
1.4.4	Lancement d'une application . . . . .	29
1.4.5	Application sur l'AVD . . . . .	29
1.4.6	Contrôle de l'AVD . . . . .	30
1.5	Communication AVD - Android Studio . . . . .	30
1.5.1	Fenêtres Android . . . . .	30
1.5.2	Fenêtre <code>LogCat</code> . . . . .	30
1.5.3	Filtrage des messages . . . . .	30
1.5.4	Émission d'un message pour <code>LogCat</code> . . . . .	31
1.5.5	Logiciel ADB . . . . .	31
1.5.6	Mode d'emploi de ADB . . . . .	31
1.5.7	Mode d'emploi, suite . . . . .	31
1.5.8	Système de fichiers Android . . . . .	32
1.5.9	Mode d'emploi, suite . . . . .	32
1.6	Création d'un paquet installable . . . . .	32
1.6.1	Paquet . . . . .	32
1.6.2	Signature d'une application . . . . .	33

1.6.3	Création du <i>keystore</i> . . . . .	33
1.6.4	Création d'une clé . . . . .	33
1.6.5	Création du paquet . . . . .	33
1.6.6	Et voilà . . . . .	35
<b>2</b>	<b>Création d'interfaces utilisateur</b>	<b>36</b>
2.1	Interface et ressources . . . . .	36
2.1.1	Activités . . . . .	36
2.1.2	Création d'un écran . . . . .	36
2.1.3	Identifiant de ressource . . . . .	37
2.1.4	La classe R . . . . .	37
2.1.5	Rappel sur la structure d'un fichier XML . . . . .	37
2.1.6	Espaces de nommage dans un fichier XML . . . . .	38
2.1.7	Création d'une interface par programme . . . . .	38
2.1.8	Programme et ressources . . . . .	38
2.1.9	Ressources de type chaînes . . . . .	39
2.1.10	Traduction des chaînes ( <i>localisation</i> ) . . . . .	39
2.1.11	Référencement des ressources texte . . . . .	39
2.1.12	Identifiants et vues . . . . .	40
2.1.13	@id/nom ou @+id/nom ? . . . . .	40
2.1.14	Images : R.drawable.nom . . . . .	40
2.1.15	Tableau de chaînes : R.array.nom . . . . .	41
2.1.16	Autres . . . . .	41
2.2	Dispositions . . . . .	41
2.2.1	Structure d'une interface Android . . . . .	41
2.2.2	Arbre des vues . . . . .	41
2.2.3	Représentation en XML . . . . .	42
2.2.4	Paramètres de positionnement . . . . .	42
2.2.5	Paramètres généraux . . . . .	42
2.2.6	Autres paramètres géométriques . . . . .	43
2.2.7	Marges et remplissage . . . . .	43
2.2.8	Groupe de vues <code>LinearLayout</code> . . . . .	43
2.2.9	Pondération des tailles . . . . .	44
2.2.10	Exemple de poids différents . . . . .	44

2.2.11	Groupe de vues <code>TableLayout</code> . . . . .	44
2.2.12	Largeur des colonnes d'un <code>TableLayout</code> . . . . .	45
2.2.13	Groupe de vues <code>RelativeLayout</code> . . . . .	45
2.2.14	Utilisation d'un <code>RelativeLayout</code> . . . . .	45
2.2.15	Autres groupements . . . . .	46
2.3	Composants d'interface . . . . .	46
2.3.1	Vues . . . . .	46
2.3.2	<code>TextView</code> . . . . .	46
2.3.3	<code>Button</code> . . . . .	47
2.3.4	Bascules . . . . .	47
2.3.5	<code>EditText</code> . . . . .	47
2.3.6	Autres vues . . . . .	47
2.4	Styles . . . . .	48
2.4.1	Styles et thèmes . . . . .	48
2.4.2	Définir un style . . . . .	48
2.4.3	Utiliser un style . . . . .	48
2.4.4	Utiliser un thème . . . . .	48
2.4.5	C'est tout . . . . .	49

### **3 Vie d'une application 50**

3.1	Applications et activités . . . . .	50
3.1.1	Composition d'une application . . . . .	50
3.1.2	Déclaration d'une application . . . . .	50
3.1.3	Sécurité des applications . . . . .	51
3.1.4	Autorisations d'une application . . . . .	51
3.1.5	Démarrage d'une application . . . . .	51
3.1.6	Démarrage d'une activité et <code>Intents</code> . . . . .	52
3.1.7	Lancement d'une activité par programme . . . . .	52
3.1.8	Lancement d'une application Android . . . . .	52
3.1.9	Lancement d'une activité d'une autre application . . . . .	52
3.2	Applications . . . . .	53
3.2.1	Fonctionnement d'une application . . . . .	53
3.2.2	Navigation entre activités . . . . .	53
3.2.3	Lancement sans attente . . . . .	53

3.2.4	Lancement avec attente de résultat . . . . .	53
3.2.5	Lancement avec attente, suite . . . . .	55
3.2.6	Terminaison d'une activité . . . . .	55
3.2.7	Méthode <code>onActivityResult</code> . . . . .	55
3.2.8	Transport d'informations dans un <code>Intent</code> . . . . .	56
3.2.9	Extraction d'informations d'un <code>Intent</code> . . . . .	56
3.2.10	Contexte d'application . . . . .	56
3.2.11	Définition d'un contexte d'application . . . . .	56
3.2.12	Définition d'un contexte d'application, suite . . . . .	57
3.2.13	Définition d'un contexte d'application, fin . . . . .	57
3.3	Activités . . . . .	57
3.3.1	Présentation . . . . .	57
3.3.2	Cycle de vie d'une activité . . . . .	58
3.3.3	Événements de changement d'état . . . . .	58
3.3.4	Squelette d'activité . . . . .	58
3.3.5	Terminaison d'une activité . . . . .	59
3.3.6	Pause d'une activité . . . . .	59
3.3.7	Arrêt d'une activité . . . . .	59
3.3.8	Enregistrement de valeurs d'une exécution à l'autre . . . . .	59
3.3.9	Restaurer l'état au lancement . . . . .	60
3.4	Vues et activités . . . . .	60
3.4.1	Obtention des vues . . . . .	60
3.4.2	Propriétés des vues . . . . .	60
3.4.3	Actions de l'utilisateur . . . . .	61
3.4.4	Définition d'un écouteur . . . . .	61
3.4.5	Écouteur privé anonyme . . . . .	62
3.4.6	Écouteur privé . . . . .	62
3.4.7	L'activité elle-même en tant qu'écouteur . . . . .	62
3.4.8	Distinction des émetteurs . . . . .	63
3.4.9	Événements des vues courantes . . . . .	63
3.4.10	C'est fini pour aujourd'hui . . . . .	63

<b>4</b>	<b>Application liste</b>	<b>64</b>
4.1	Présentation . . . . .	64
4.1.1	Principe général . . . . .	64
4.1.2	Schéma global . . . . .	65
4.1.3	Une classe pour représenter les items . . . . .	65
4.1.4	Données initiales . . . . .	65
4.1.5	Copie dans un <code>ArrayList</code> . . . . .	66
4.1.6	Le container Java <code>ArrayList&lt;type&gt;</code> . . . . .	66
4.1.7	Données initiales dans les ressources . . . . .	67
4.1.8	Données dans les ressources, suite . . . . .	67
4.1.9	Remarques . . . . .	67
4.2	Affichage de la liste . . . . .	67
4.2.1	Activité spécialisée ou layout . . . . .	67
4.2.2	Mise en œuvre . . . . .	68
4.2.3	Layout de l'activité pour afficher une liste . . . . .	68
4.2.4	Mise en place du layout d'activité . . . . .	68
4.2.5	Layout pour un item . . . . .	69
4.2.6	Autre layouts . . . . .	69
4.2.7	Layouts prédéfinis . . . . .	69
4.3	Adaptateurs . . . . .	70
4.3.1	Relations entre la vue et les données . . . . .	70
4.3.2	Rôle d'un adaptateur . . . . .	70
4.3.3	Adaptateurs prédéfinis . . . . .	70
4.3.4	<code>ArrayAdapter&lt;Type&gt;</code> pour les listes . . . . .	70
4.3.5	Exemple d'emploi . . . . .	71
4.3.6	Affichage avec une <code>ListActivity</code> . . . . .	71
4.3.7	Exemple avec les layouts standards . . . . .	71
4.4	Adaptateur personnalisé . . . . .	72
4.4.1	Classe <code>Adapter</code> personnalisée . . . . .	72
4.4.2	Classe <code>Adapter</code> perso, suite . . . . .	72
4.4.3	Méthode <code>getView</code> personnalisée . . . . .	73
4.4.4	Méthode <code>PlanetextView.create</code> . . . . .	73
4.4.5	Layout d'item <code>res/layout/item_planete.xml</code> . . . . .	73
4.4.6	Classe personnalisée dans les ressources . . . . .	74

4.4.7	Classe <code>PlaneteView</code> pour afficher les items . . . . .	74
4.4.8	Définition de la classe <code>PlaneteView</code> . . . . .	74
4.4.9	Créer des vues à partir d'un layout XML . . . . .	75
4.4.10	Méthode <code>PlaneteView.create</code> . . . . .	75
4.4.11	Méthode <code>findViews</code> . . . . .	75
4.4.12	Pour finir, la méthode <code>PlaneteView.display</code> . . . . .	76
4.4.13	Récapitulatif . . . . .	76
4.4.14	Le résultat . . . . .	76
4.5	Actions utilisateur sur la liste . . . . .	76
4.5.1	Modification des données . . . . .	76
4.5.2	Clic sur un élément . . . . .	77
4.5.3	Clic sur un élément, suite . . . . .	77
4.5.4	Clic sur un élément, suite . . . . .	78
4.5.5	Clic sur un élément, fin . . . . .	78
4.5.6	Liste d'éléments cochables . . . . .	78
4.5.7	Liste cochable simple . . . . .	78
4.5.8	Liste à choix multiples . . . . .	79
4.5.9	Liste cochable personnalisée . . . . .	79
4.5.10	Ouf, c'est fini . . . . .	79

## 5 Ergonomie 80

5.1	Barre d'action et menus . . . . .	80
5.1.1	Barre d'action . . . . .	80
5.1.2	Réalisation d'un menu . . . . .	80
5.1.3	Spécification d'un menu . . . . .	81
5.1.4	Icônes pour les menus . . . . .	81
5.1.5	Thème pour une barre d'action . . . . .	81
5.1.6	Écouteurs pour les menus . . . . .	82
5.1.7	Réactions aux sélections d'items . . . . .	82
5.1.8	Menus en cascade . . . . .	82
5.1.9	Menus contextuels . . . . .	83
5.1.10	Associer un menu contextuel à une vue . . . . .	83
5.1.11	Callback d'affichage du menu . . . . .	84
5.1.12	Callback des items du menu . . . . .	84

5.2	Annonces et dialogues . . . . .	84
5.2.1	Annonces : <i>toasts</i> . . . . .	84
5.2.2	Annonces personnalisées . . . . .	85
5.2.3	Dialogues . . . . .	85
5.2.4	Dialogue d'alerte . . . . .	85
5.2.5	Boutons et affichage d'un dialogue d'alerte . . . . .	86
5.2.6	Autres types de dialogues d'alerte . . . . .	86
5.2.7	Dialogues personnalisés . . . . .	86
5.2.8	Création d'un dialogue . . . . .	87
5.2.9	Affichage du dialogue . . . . .	87
5.3	Fragments et activités . . . . .	88
5.3.1	Fragments . . . . .	88
5.3.2	Tablettes, smartphones... . . . .	88
5.3.3	Structure d'un fragment . . . . .	88
5.3.4	Différents types de fragments . . . . .	89
5.3.5	Cycle de vie des fragments . . . . .	89
5.3.6	ListFragment . . . . .	89
5.3.7	ListFragment, suite . . . . .	90
5.3.8	Menus de fragments . . . . .	90
5.3.9	Intégrer un fragment dans une activité . . . . .	91
5.3.10	Fragments statiques dans une activité . . . . .	91
5.3.11	FragmentManager . . . . .	91
5.3.12	Attribution d'un fragment dynamiquement . . . . .	91
5.3.13	Disposition selon la géométrie de l'écran . . . . .	92
5.3.14	Changer la disposition selon la géométrie . . . . .	92
5.3.15	Deux dispositions possibles . . . . .	93
5.3.16	Communication entre Activité et Fragments . . . . .	93
5.3.17	Interface pour un écouteur . . . . .	93
5.3.18	Écouteur du fragment . . . . .	94
5.3.19	Écouteur de l'activité . . . . .	94
5.3.20	Relation entre deux classes à méditer, partie 1 . . . . .	95
5.3.21	À méditer, partie 2 . . . . .	95
5.4	Préférences d'application . . . . .	95
5.4.1	Illustration . . . . .	95



5.4.2	Présentation . . . . .	95
5.4.3	Définition des préférences . . . . .	96
5.4.4	Explications . . . . .	96
5.4.5	Accès aux préférences . . . . .	97
5.4.6	Préférences chaînes et nombres . . . . .	97
5.4.7	Modification des préférences par programme . . . . .	97
5.4.8	Affichage des préférences . . . . .	98
5.4.9	Fragment pour les préférences . . . . .	98
5.5	Bibliothèque support . . . . .	98
5.5.1	Compatibilité des applications . . . . .	98
5.5.2	Compatibilité des versions Android . . . . .	99
5.5.3	Bibliothèque support . . . . .	99
5.5.4	Versions de l'Android Support Library . . . . .	99
5.5.5	Mode d'emploi . . . . .	99
5.5.6	Programmation . . . . .	100
5.5.7	Il est temps de faire une pause . . . . .	100
<b>6</b>	<b>Bases de données SQLite3 &lt;&gt;</b>	<b>101</b>
6.1	SQLite3 . . . . .	101
6.1.1	Stockage d'informations . . . . .	101
6.1.2	SQLite3 . . . . .	101
6.1.3	Exemples SQL . . . . .	102
6.1.4	Autres usages de SQLite3 . . . . .	102
6.1.5	Lancement de sqlite3 en shell . . . . .	102
6.1.6	Commandes internes . . . . .	103
6.2	SQLite dans une application Android . . . . .	103
6.2.1	Bases de données Android . . . . .	103
6.2.2	Classes pour travailler avec SQLite . . . . .	103
6.2.3	Étapes du travail avec une BDD . . . . .	103
6.2.4	Base ouverte dans une activité . . . . .	104
6.2.5	Patron de conception pour les requêtes . . . . .	104
6.2.6	Noms des colonnes . . . . .	104
6.2.7	Classe pour une table . . . . .	105
6.2.8	Exemples de méthodes . . . . .	105

6.2.9	Méthodes <code>SQLiteDatabase.execSQL</code> . . . . .	106
6.2.10	Méthodes spécialisées . . . . .	106
6.2.11	Méthode <code>insert</code> . . . . .	106
6.2.12	Méthodes <code>update</code> et <code>delete</code> . . . . .	107
6.2.13	Méthode <code>rawQuery</code> . . . . .	107
6.2.14	<code>rawQuery</code> pour un seul n-uplet . . . . .	107
6.2.15	Classe <code>Cursor</code> . . . . .	108
6.2.16	Exemple de requête, classe <code>TablePlanetes</code> . . . . .	108
6.2.17	Autre type de requête . . . . .	108
6.2.18	Méthodes <code>query</code> : sans aucun intérêt . . . . .	109
6.2.19	Ouverture d'une base . . . . .	109
6.2.20	Première ouverture et ouvertures suivantes . . . . .	109
6.2.21	Un <i>helper</i> pour gérer l'ouverture/création/màj . . . . .	109
6.2.22	Exemple de <i>helper</i> . . . . .	110
6.2.23	Exemple de <i>helper</i> , suite . . . . .	110
6.2.24	méthode <code>onUpgrade</code> . . . . .	111
6.2.25	méthode <code>onUpgrade</code> , suite . . . . .	111
6.2.26	Utilisation du Helper dans l'application . . . . .	111
6.2.27	Fermeture de la base . . . . .	112
6.3	<code>CursorAdapter</code> et <code>Loaders</code> . . . . .	112
6.3.1	Lien entre une BDD et un <code>ListView</code> . . . . .	112
6.3.2	Étapes à suivre . . . . .	112
6.3.3	Activité ou fragment d'affichage d'une liste . . . . .	113
6.3.4	Création d'un adaptateur de curseur . . . . .	113
6.3.5	Ouverture de la base et création d'un chargeur . . . . .	113
6.3.6	Callback <code>onCreateLoader</code> de l'activité . . . . .	114
6.3.7	classe <code>MonCursorLoader</code> . . . . .	114
6.3.8	Callback <code>onLoadFinished</code> de l'activité . . . . .	115
6.3.9	Mise à jour de la liste . . . . .	115
6.3.10	En mode compatibilité . . . . .	115
6.4	<code>ContentProviders</code> . . . . .	115
6.4.1	Présentation rapide . . . . .	115
6.4.2	C'est tout pour aujourd'hui . . . . .	115

<b>7</b>	<b>Services réseau</b>	<b>116</b>
7.1	WebServices . . . . .	116
7.1.1	Base de donnée distante . . . . .	116
7.1.2	Échange entre un serveur SQL et une application Android . . . . .	116
7.1.3	Principe général . . . . .	116
7.1.4	Exemple de script PHP Get . . . . .	117
7.1.5	Exemple de script PHP Get . . . . .	117
7.1.6	Exemple de script PHP Post . . . . .	117
7.1.7	Format JSON <i>JavaScript Object Notation</i> . . . . .	118
7.1.8	JSON en Java . . . . .	118
7.1.9	Dans l'application Android . . . . .	118
7.1.10	Affichage d'une liste . . . . .	119
7.1.11	La classe <b>RemoteDatabase</b> . . . . .	119
7.1.12	Modification d'un n-uplet . . . . .	119
7.1.13	Méthode <b>post</b> ( <b>écouteur</b> , <b>script</b> , <b>params</b> ) . . . . .	120
7.1.14	Principe de la méthode <b>post</b> . . . . .	120
7.2	<b>AsyncTasks</b> . . . . .	120
7.2.1	Présentation . . . . .	120
7.2.2	Tâches asynchrones . . . . .	121
7.2.3	Principe d'utilisation d'une <b>AsyncTask</b> . . . . .	121
7.2.4	Structure d'une <b>AsyncTask</b> . . . . .	121
7.2.5	Paramètres d'une <b>AsyncTask</b> . . . . .	122
7.2.6	Exemple de paramétrage . . . . .	122
7.2.7	Paramètres variables . . . . .	122
7.2.8	Définition d'une <b>AsyncTask</b> . . . . .	123
7.2.9	<b>AsyncTask</b> , suite . . . . .	123
7.2.10	Lancement d'une <b>AsyncTask</b> . . . . .	123
7.2.11	Schéma récapitulatif . . . . .	124
7.2.12	<b>execute</b> ne retourne rien . . . . .	124
7.2.13	Récupération du résultat d'un <b>AsyncTask</b> . . . . .	124
7.2.14	Simplification . . . . .	125
7.2.15	Recommandations . . . . .	125
7.2.16	Autres tâches asynchrones . . . . .	125
7.3	Requêtes HTTP . . . . .	126

7.3.1	Présentation . . . . .	126
7.3.2	Principe de programmation pour un GET . . . . .	126
7.3.3	Exemple de requête GET . . . . .	126
7.3.4	Encodage de paramètres pour une requête . . . . .	127
7.3.5	Principe de programmation pour un POST . . . . .	127
7.3.6	Exemple de requête POST . . . . .	127
7.3.7	Requêtes asynchrones . . . . .	128
7.3.8	Permissions pour l'application . . . . .	128
7.3.9	Voilà tout pour cette semaine . . . . .	128
<b>8</b>	<b>Cartes et Dessin 2D interactif</b>	<b>129</b>
8.1	OpenStreetMap . . . . .	129
8.1.1	Présentation . . . . .	129
8.1.2	Documentation . . . . .	129
8.1.3	Pour commencer . . . . .	131
8.1.4	Layout pour une carte OSM . . . . .	131
8.1.5	Activité pour une carte OSM . . . . .	131
8.1.6	Positionnement de la vue . . . . .	132
8.1.7	Calques . . . . .	132
8.1.8	Mise à jour de la carte . . . . .	132
8.1.9	Marqueurs . . . . .	133
8.1.10	Marqueur personnalisés . . . . .	133
8.1.11	Réaction à un clic . . . . .	133
8.1.12	Itinéraires . . . . .	134
8.1.13	Position GPS . . . . .	134
8.1.14	Mise à jour en temps réel de la position . . . . .	134
8.1.15	Positions simulées . . . . .	135
8.1.16	Clics sur la carte . . . . .	135
8.1.17	Traitement des clics . . . . .	135
8.1.18	Autorisations . . . . .	136
8.2	Dessin en 2D . . . . .	136
8.2.1	Principes . . . . .	136
8.2.2	Layout pour le dessin . . . . .	136
8.2.3	Méthode <code>onDraw</code> . . . . .	137

8.2.4	Méthodes de la classe <b>Canvas</b>	137
8.2.5	Peinture <b>Paint</b>	137
8.2.6	Quelques accesseurs de <b>Paint</b>	138
8.2.7	Motifs	138
8.2.8	Shaders	138
8.2.9	Shaders, suite et fin	139
8.2.10	Quelques remarques	139
8.2.11	« Dessinables »	139
8.2.12	Images PNG étirables 9patch	140
8.2.13	<b>Drawable</b> , suite	140
8.2.14	Variantes	140
8.2.15	Utilisation d'un <b>Drawable</b>	141
8.2.16	Enregistrer un dessin dans un fichier	141
8.2.17	Coordonnées dans un canvas	141
8.3	Interactions avec l'utilisateur	142
8.3.1	Écouteurs pour les touchers de l'écran	142
8.3.2	Modèle de gestion des actions	142
8.3.3	Automate pour gérer les actions	143
8.4	Boîtes de dialogue spécifiques	143
8.4.1	Sélecteur de couleur	143
8.4.2	Version simple	143
8.4.3	Concepts	143
8.4.4	Fragment de dialogue	144
8.4.5	Méthode <b>onCreateDialog</b>	144
8.4.6	Vue personnalisée dans le dialogue	145
8.4.7	Layout de cette vue	145
8.4.8	Utilisation du dialogue	145
8.4.9	Sélecteur de fichier	146
8.4.10	C'est la fin	147

## Semaine 1

---

### Environnement de développement

---

Le cours de cette semaine présente l'environnement de développement Android :

- Le SDK Android et Android Studio
- Création d'une application simple
- Communication avec une tablette.

### 1.1. Introduction

#### 1.1.1. Android



né en 2004,  
racheté par Google en 2005,  
publié en 2007, version 1.5,  
de nombreuses versions depuis, on en est à la 7.1.1 (janvier 2017).

#### 1.1.2. Définition

Système complet pour smartphones et tablettes

- Gestion matérielle : système d'exploitation Linux sous-jacent
- API de programmation : interfaces utilisateur, outils...
- Applications : navigateur, courrier...

Évolution et obsolescence très rapides (c'est voulu)

- Ce que vous allez apprendre sera rapidement dépassé (1 an)
  - syntaxiquement (méthodes, paramètres, classes, ressources...)
  - mais pas les concepts (principes, organisation...)
- Vous êtes condamné(e) à une autoformation permanente, mais c'est le lot des informaticiens.

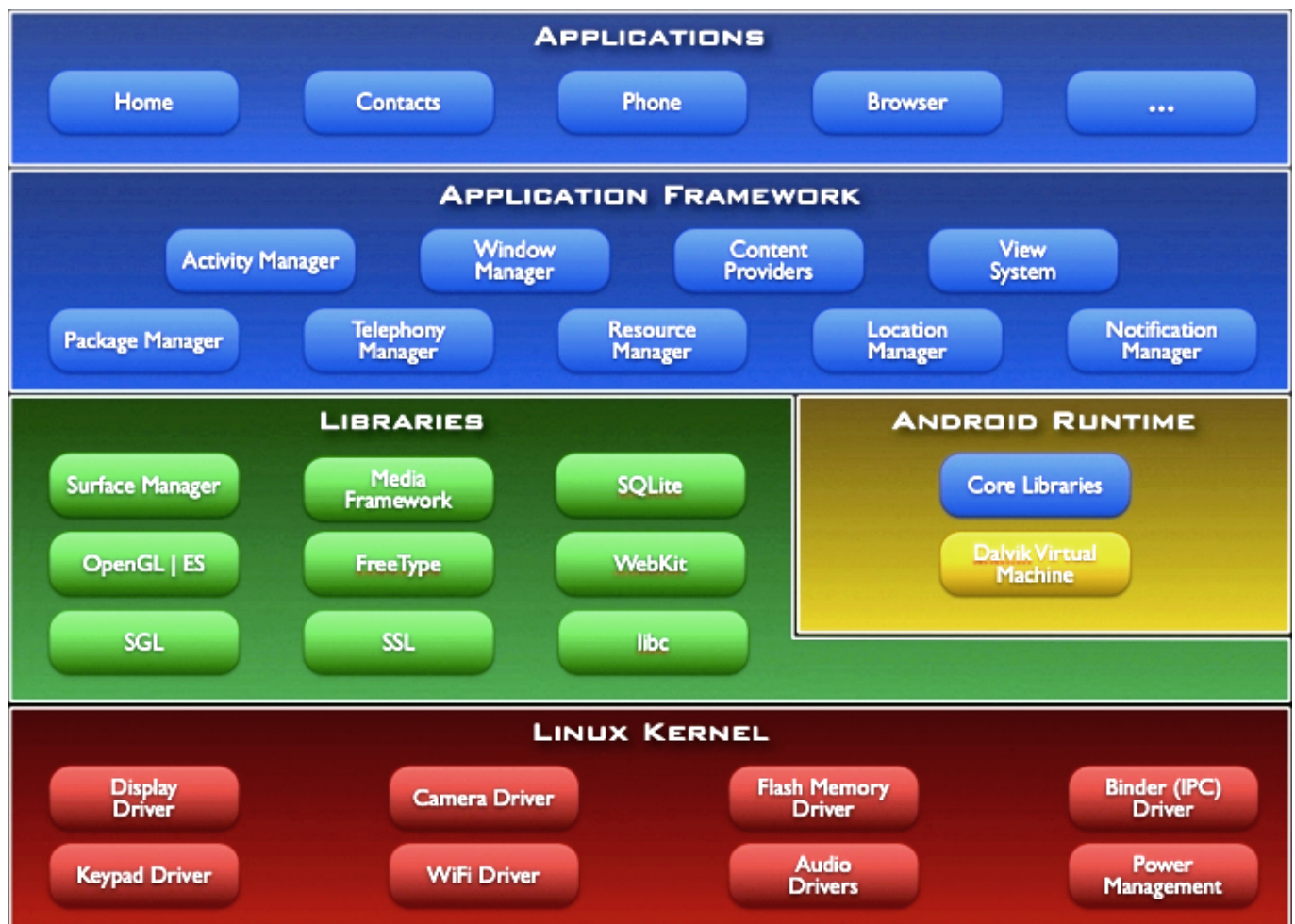


Figure 1: Constituants d'Android

### 1.1.3. Composants d'Android

Voir la figure 1, page 15.

### 1.1.4. Programmation d'applications

Une application Android est composée de :

- **Sources Java** compilés pour une machine virtuelle « *Dalvik* » (versions  $\leq 4.4$ ) ou « *ART* » depuis la version 5
- Fichiers XML appelés **ressources** : interface, textes...
- Fichiers de données supplémentaires
- **Manifeste** = description du contenu du logiciel
  - fichiers présents dans l'archive
  - demandes d'autorisations
  - signature des fichiers, durée de validité, etc.

Tout cet ensemble est géré à l'aide d'un IDE (environnement de développement) appelé *Android Studio* et d'un ensemble logiciel (bibliothèques, outils) appelé *SDK Android*.

## 1.2. SDK Android et Android Studio

### 1.2.1. SDK et Studio

Le SDK contient :

- les bibliothèques Java pour créer des logiciels
- les outils de mise en boîte des logiciels
- un émulateur de tablettes pour tester les applications *AVD*
- un outil de communication avec les vraies tablettes *ADB*

Android Studio offre :

- un éditeur de sources
- des outils de compilation et de lancement d'*AVD*

### 1.2.2. SDK Manager

Le SDK est livré avec un gestionnaire. C'est une application qui permet de choisir les composants à installer.

Voir la figure 2, page 17.

### 1.2.3. Choix des éléments du SDK

Télécharger le SDK correspondant au système d'exploitation. Ce SDK contient un gestionnaire (*SDK Manager*).

Le gestionnaire permet de choisir les versions<sup>2</sup> à installer :

---

<sup>1</sup>Certaines images de ce cours sont de <http://developer.android.com>

<sup>2</sup>versions existantes à la date de rédaction de ce cours



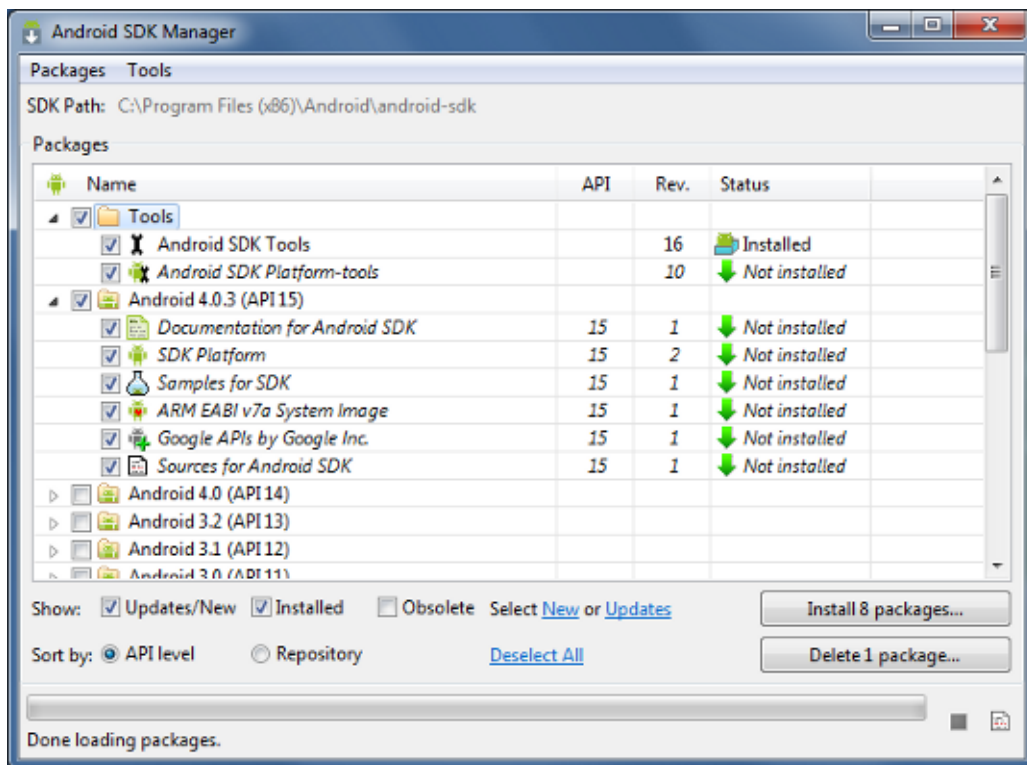


Figure 2: Gestionnaire de paquets Android

- Android 7.1.1 (API 25)
- Android 7.0 (API 24)
- Android 6 (API 23)
- Android 5.1.1 (API 22)
- Android 4.4W.2 (API 20)
- ...
- Android 1.5 (API 3)

Choisir celles qui correspondent aux tablettes qu'on vise.

#### 1.2.4. Dossiers du SDK

Le gestionnaire télécharge environ 800Mo de fichiers :

- SDK Tools : indispensable, contient le gestionnaire,
- SDK Platform-tools : indispensable, contient adb,
- SDK Platform : indispensable, contient les bibliothèques,
- System images : pour créer des AVD,
- Android Support : divers outils pour créer des applications,
- Exemples et sources.

C'est déjà installé à l'IUT, mais dans des versions antérieures, correspondant aux tablettes dont on dispose.

### 1.2.5. Android Studio

Pour finir, il faut installer Android Studio selon la procédure expliquée sur [cette page](#). Il est déjà installé à l'IUT, mais en version un peu plus ancienne.

Après cette installation, il faut indiquer l'emplacement du SDK dans Android Studio.

Une autre manière est d'installer Android Studio en premier, lui-même installant tous les composants manquants (et même un peu plus que le nécessaire).

## 1.3. Première application

### 1.3.1. Objectif de la semaine 1

Cette semaine, ce sera seulement un aperçu rapide des possibilités :

- Création d'une application « *HelloWorld* » avec un assistant,
- Tour du propriétaire,
- Exécution de l'application,
- Mise sous forme d'un paquet.

### 1.3.2. Assistant de création d'application

Android Studio contient un assistant de création d'applications :

Voir la figure 3, page 19.

### 1.3.3. Choix de la version

Chaque version d'Android, dénotée par son *API level*, ex: 25, apporte des améliorations et supprime des dispositifs obsolètes.

Toute application exige un certain niveau d'API :

- **Minimum SDK** : il faut au moins cette API car on utilise certaines classes et méthodes absentes des précédentes APIs,

Avec Eclipse, on devait aussi spécifier :

- **Target SDK** : l'application sera testée et marchera correctement jusqu'à ce niveau d'API,
- **Compile With** : c'est le niveau maximal de fonctionnalités qu'on se limite à employer. Si on fait appel à quelque chose de plus récent que ce niveau, le logiciel ne se compilera pas.

### 1.3.4. Choix de la version

Voici comment choisir le Minimum SDK :

Voir la figure 4, page 20.

### 1.3.5. Choix du type d'application

Ensuite, on choisit le type de projet. Pour un premier essai, on se limite au plus simple, *Blank Activity* :

Voir la figure 5, page 21.

The screenshot shows the 'Create New Project' window in Android Studio. The window has a title bar 'Create New Project' and a close button. Below the title bar is a header with the Android Studio logo and the text 'New Project' and 'Android Studio'. The main area is titled 'Configure your new project'. It contains four input fields: 'Application name' with the value 'MementOL', 'Company Domain' with the value 'fr.iutlan.games.mementol', 'Package name' with the value 'mementol.games.iutlan.fr.mementol' and an 'Edit' link, and 'Project location' with the value '/home/pierre/Projets/AndroidStudioProjects/MementOL' and a browse button. At the bottom, there are four buttons: 'Previous', 'Next', 'Cancel', and 'Finish'.

Create New Project

New Project  
Android Studio

Configure your new project

Application name: MementOL

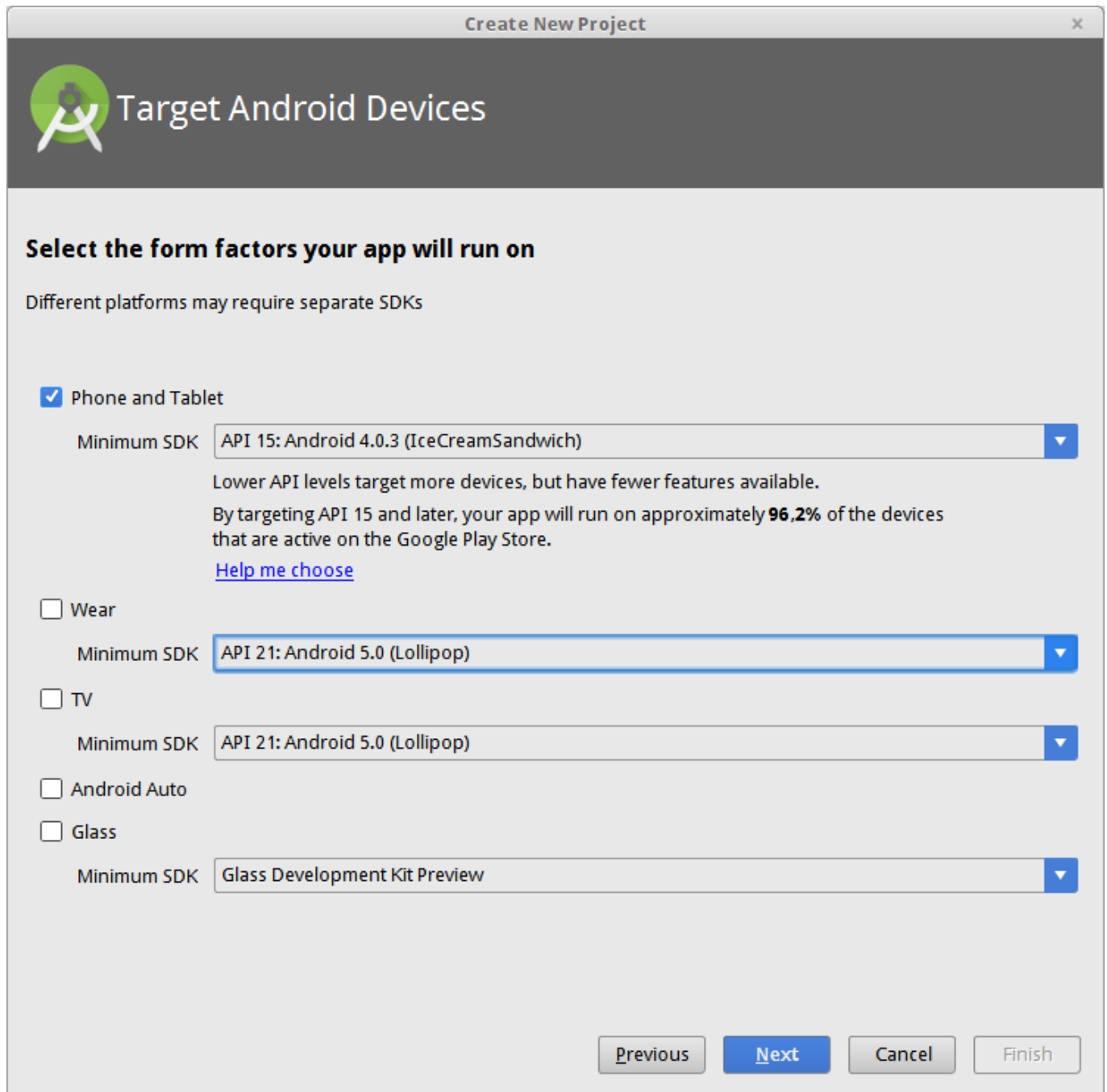
Company Domain: fr.iutlan.games.mementol

Package name: mementol.games.iutlan.fr.mementol [Edit](#)


Project location: /home/pierre/Projets/AndroidStudioProjects/MementOL

Previous Next Cancel Finish

Figure 3: Assistant de création de projet



**Create New Project**

 **Target Android Devices**

**Select the form factors your app will run on**

Different platforms may require separate SDKs

☒ **Phone and Tablet**

Minimum SDK **API 15: Android 4.0.3 (IceCreamSandwich)**

Lower API levels target more devices, but have fewer features available.  
By targeting API 15 and later, your app will run on approximately **96.2%** of the devices that are active on the Google Play Store.  
[Help me choose](#)

☐ **Wear**

Minimum SDK **API 21: Android 5.0 (Lollipop)**

☐ **TV**

Minimum SDK **API 21: Android 5.0 (Lollipop)**

☐ **Android Auto**

☐ **Glass**

Minimum SDK **Glass Development Kit Preview**

**Previous** **Next** **Cancel** **Finish**

Figure 4: Choix de la version

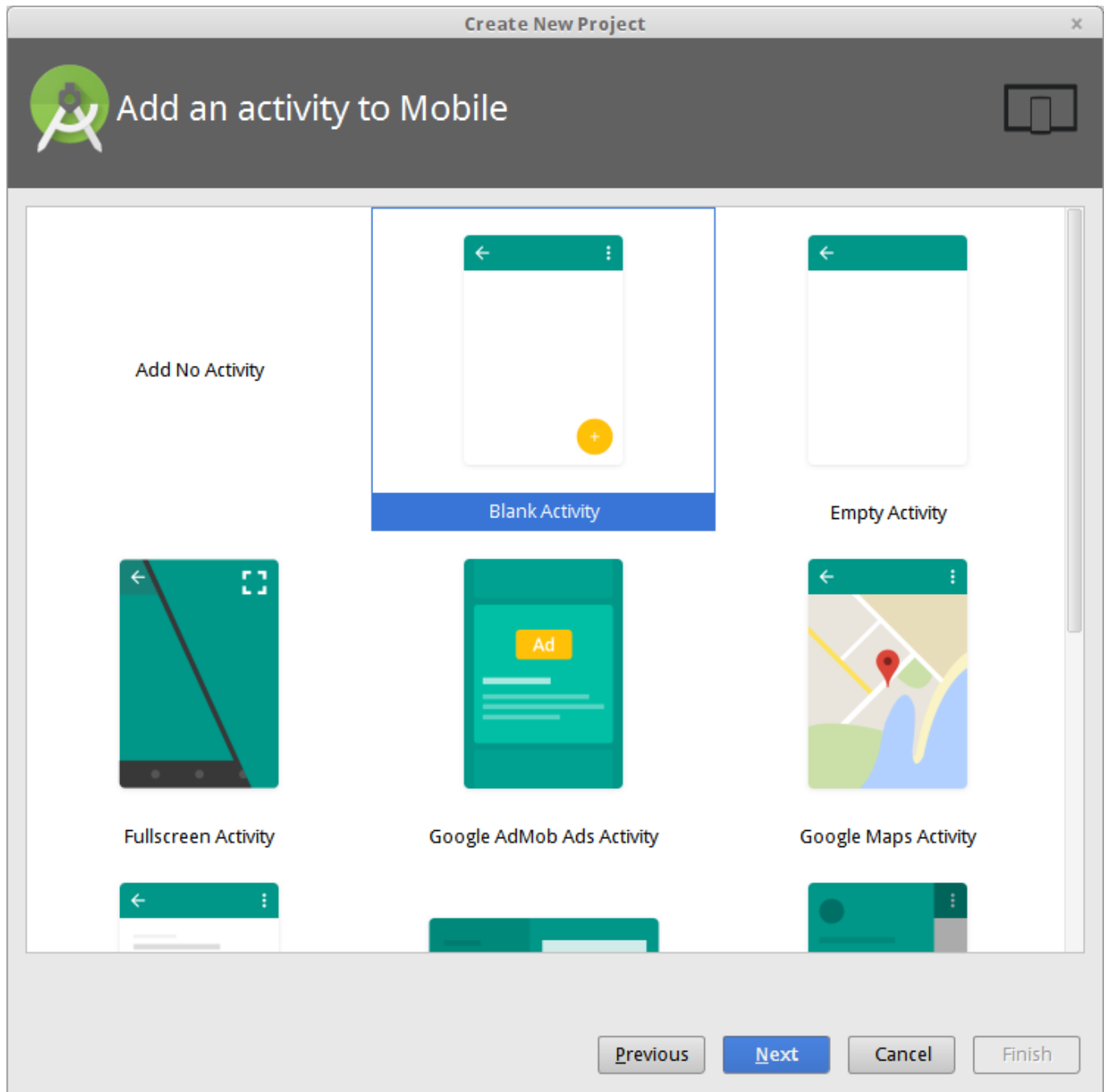


Figure 5: Choix du type d'activité

### 1.3.6. Points à configurer

L'assistant demande ensuite plusieurs informations :

- Nom de l'application, ex : `HelloWorld`,
- Nom de la classe principale : `MainActivity`,
- Nom du layout de la classe principale : `activity_main`<sup>3</sup>,
- Nom du layout du menu principal : `menu_main`.

Tout peut être renommé ultérieurement, voir `refactor/rename`.

Le package du logiciel a été défini dans le premier écran.

### 1.3.7. Noms des packages et classes

Voici où on indique ces informations :

Voir la figure 6, page 23.

### 1.3.8. Résultat de l'assistant

L'assistant a créé de nombreux éléments visibles dans la colonne de gauche de l'IDE :

- **manifests** : description et liste des classes de l'application
- **java** : les sources, rangés par paquetage,
- **res** : ressources = fichiers XML et images de l'interface, il y a des sous-dossiers :
  - **layout** : interfaces (disposition des vues sur les écrans)
  - **menu** : menus contextuels ou d'application
  - **mipmap** et **drawable** : images, icônes de l'interface
  - **values** : valeurs de configuration, textes...
- **Gradle scripts** : c'est l'outil de compilation du projet.

NB: on ne va pas chercher à comprendre ça cette semaine.

### 1.3.9. Fenêtre du projet

Voir la figure 7, page 24.

### 1.3.10. Éditeurs spécifiques

Les ressources (disposition des vues dans les interfaces, menus, images vectorielles, textes...) sont définies à l'aide de fichiers XML.

Studio fournit des éditeurs spécialisés pour ces fichiers, par exemple :

- Formulaires pour :
  - **res/values/strings.xml** : textes de l'interface.
- Éditeurs graphiques pour :
  - **res/layout/\*.xml** : disposition des contrôles sur l'interface.

---

<sup>3</sup>Je n'aime pas ce nommage inversé entre activités `TrucActivity` et layouts `activity_truc`, je préfère `truc_activity.xml`. Même remarque pour les menus, `main_menu` au lieu de `menu_main`. Ça permet d'organiser les ressources par activités, `main_activity`, `main_menu`..., et non par catégories.

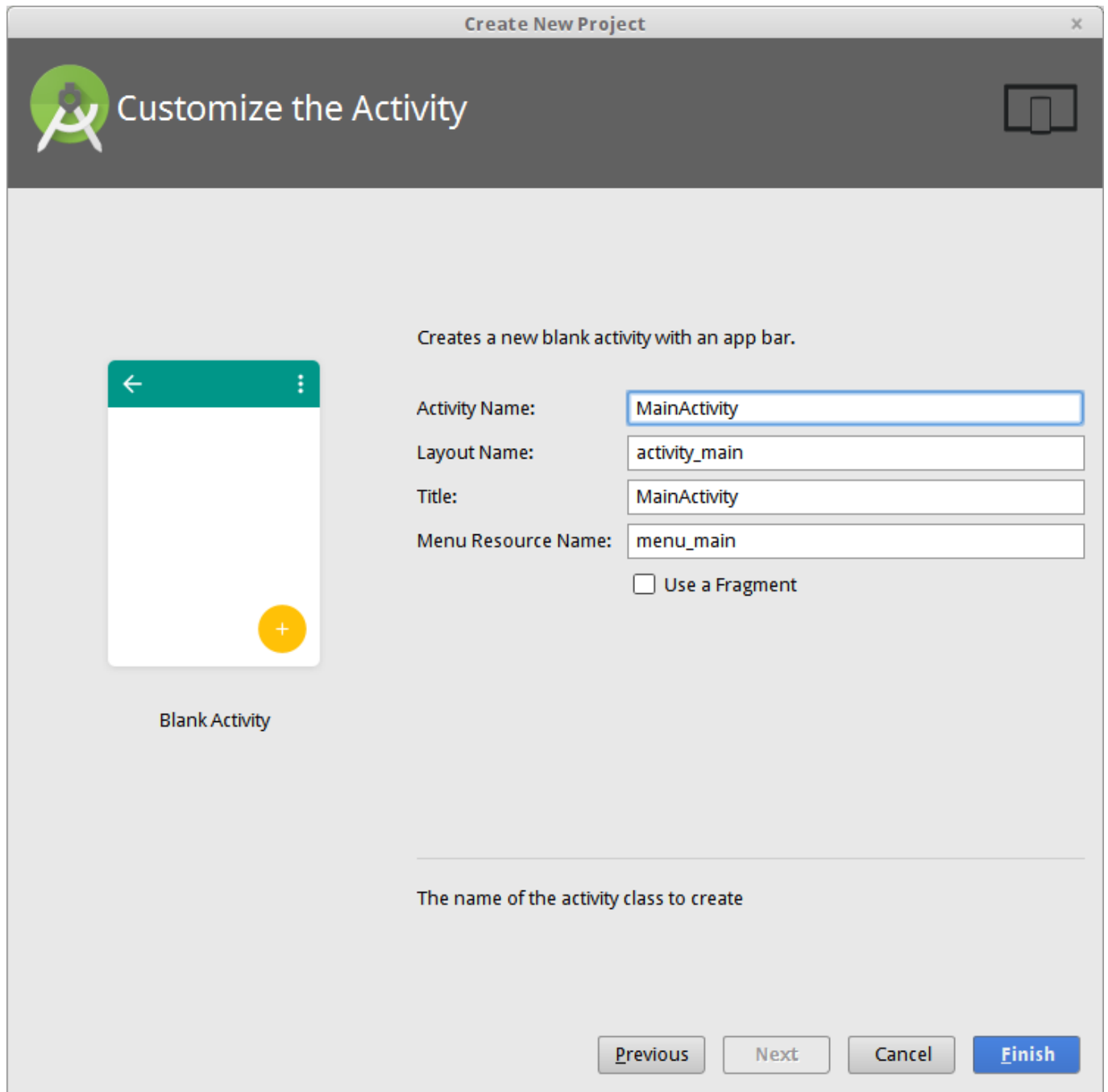


Figure 6: Choix du type d'activité

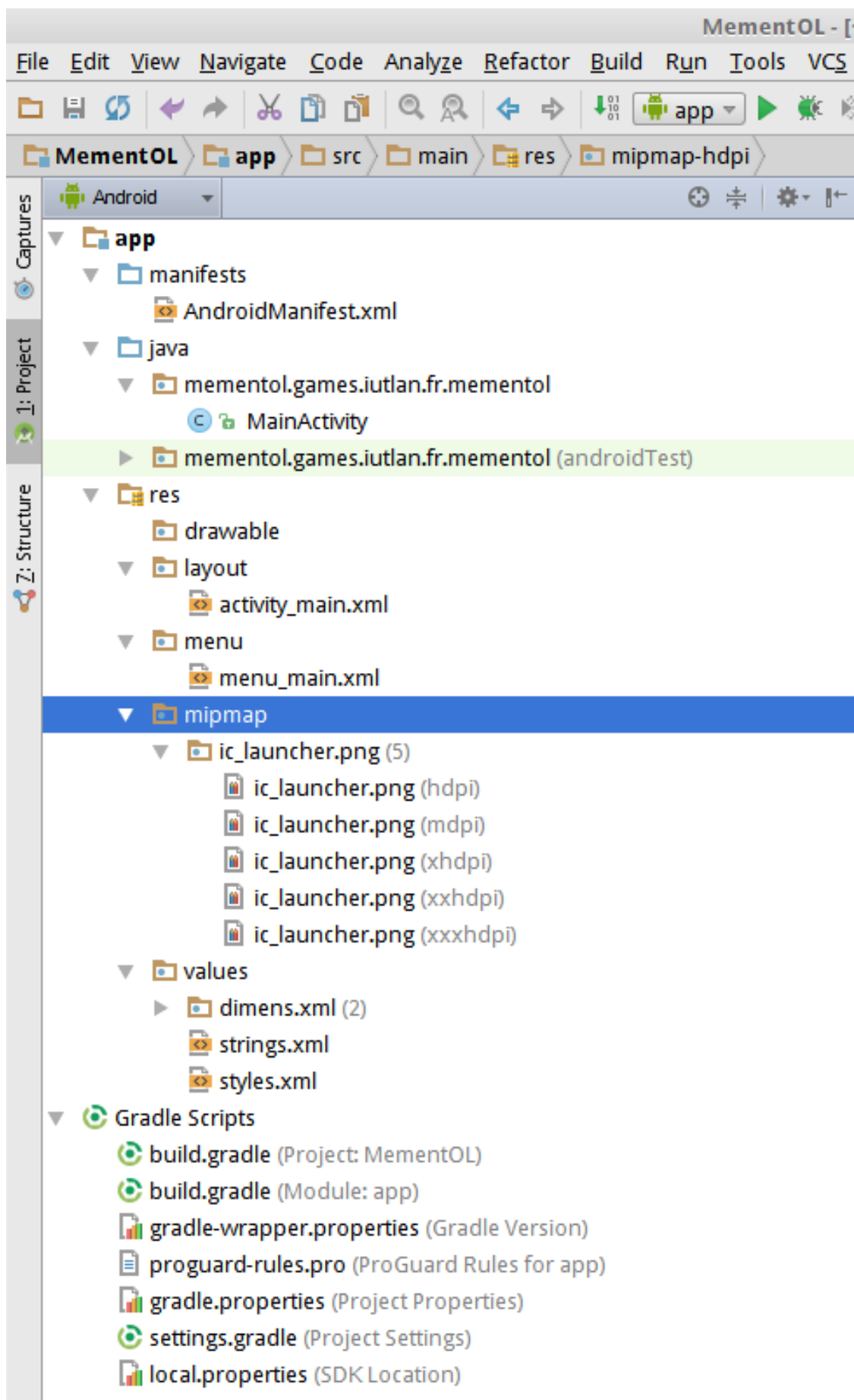


Figure 7: Éléments d'un projet Android



### 1.3.11. Exemple res/values/strings.xml

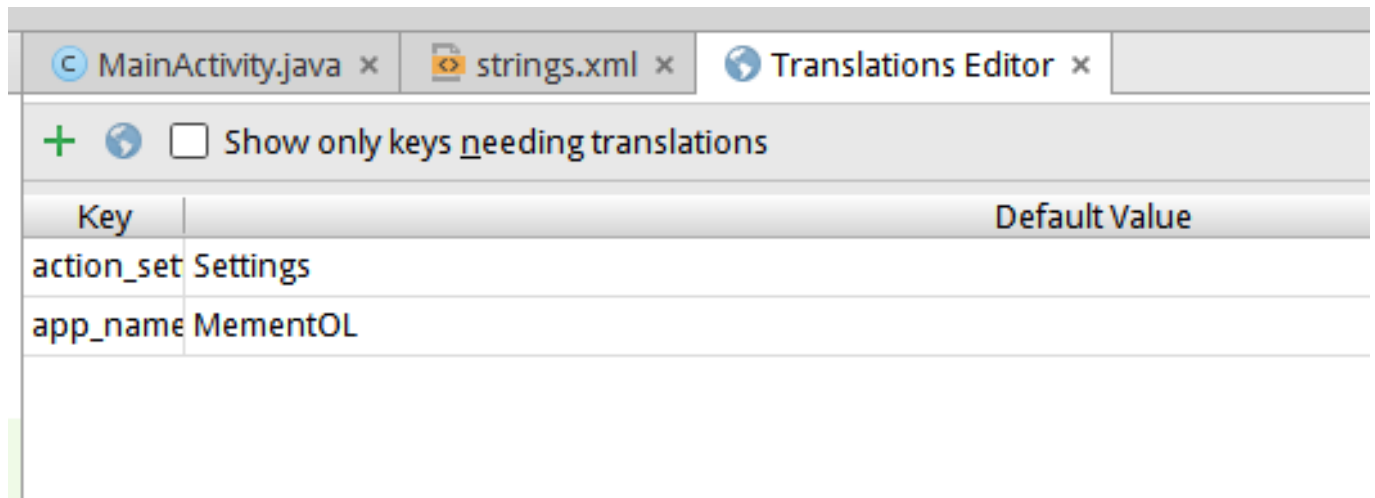


Figure 8: Éditeur du manifeste

### 1.3.12. Exemple res/layout/main.xml

Voir la figure 9, page 26.

### 1.3.13. Source XML sous-jacent

Ces éditeurs sont beaucoup plus confortables que le XML brut, mais ne permettent pas de tout faire (et plantent souvent parfois).

Dans certains cas, il faut éditer le source XML directement :



```
<RelativeLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
</RelativeLayout>
```

Vous avez remarqué le *namespace* des attributs.

### 1.3.14. Reconstruction du projet

Automatique :

- Ex: modifier le fichier res/values/strings.xml ou un source Java,
- Gradle compile automatiquement le projet.

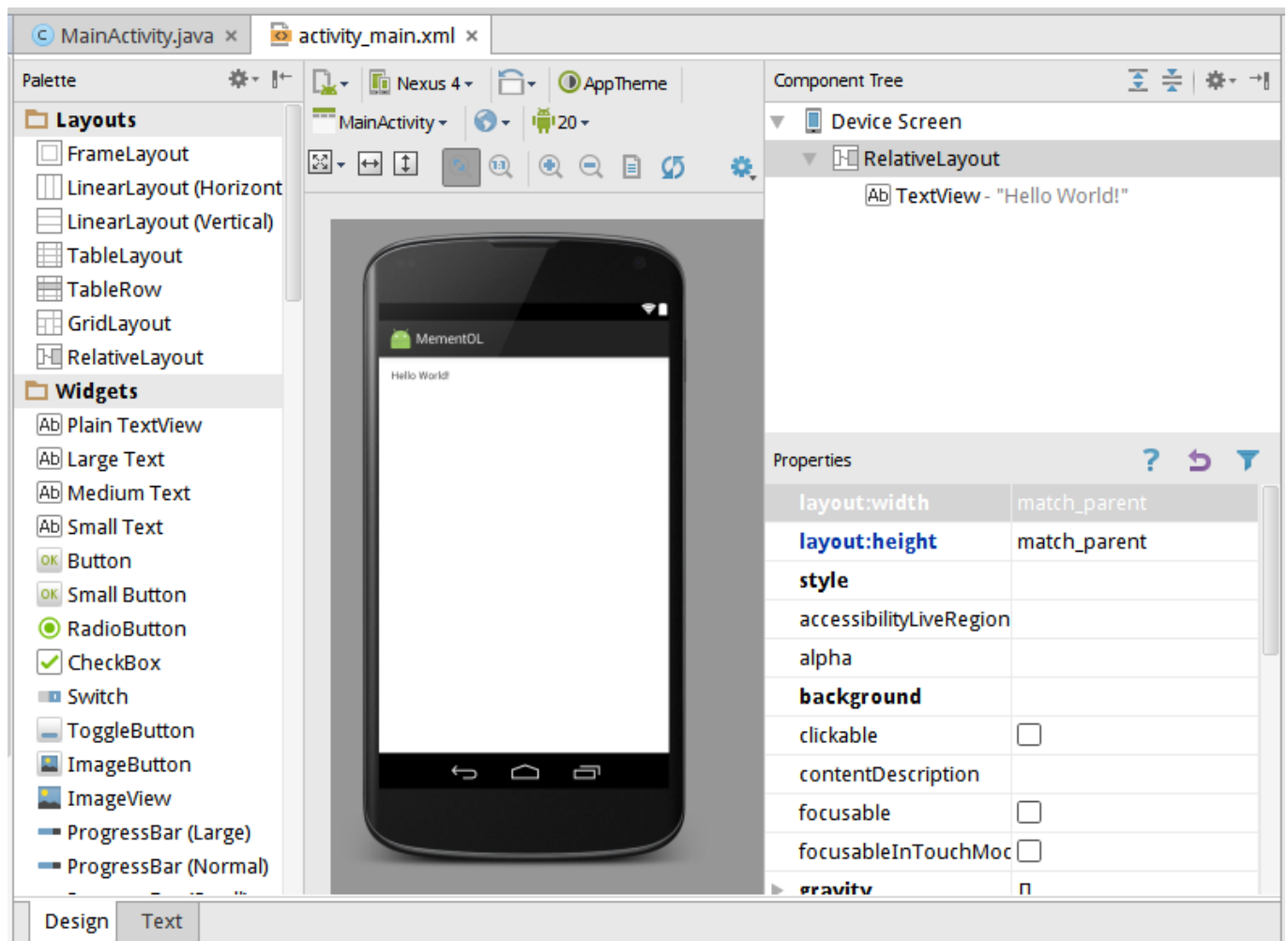


Figure 9: Éditeur graphique

Manuelle, parfois nécessaire quand on modifie certaines ressources :

- Sélectionner le projet et choisir menu **Build/Clean...**

Ces actions lancent l'exécution de Gradle.

### 1.3.15. Gradle

**Gradle** est un outil de construction de projets comme **Make** (projets C++ sur Unix), **Ant** (projets Java dans Eclipse) et **Maven**. Il se sert d'un (ou plusieurs) script `build.gradle` qui indique la structure du projet.

Un projet AndroidStudio est constitué de :

- dossier `app` qui contient `src/main` qui lui-même contient `java`, `res` et le manifeste
- dossier `build` pour les fichiers intermédiaires (`.class` et autres)
- des fichiers de configuration de gradle :
  - `local.properties` : chemin d'accès au SDK
  - `build.gradle` : structure du projet

### 1.3.16. Gradle en ligne de commande

Parmi les fichiers et dossiers, il y a également :

- un script bash `gradlew` qui appelle `gradle` proprement
- dossier `gradle` qui contient l'archive jar permettant de lancer `gradlew`

Il suffit de taper :

- `gradlew clean` : suppression du dossier `build`
- `gradlew assembleDebug` : compilation du projet, en version debug, `assembleRelease` pour la version publique
- `gradlew build` : compile et teste l'application
- `gradlew installDebug` : compilation et installation
- `gradlew tasks` : liste des cibles possibles.

## 1.4. Première exécution

### 1.4.1. Exécution de l'application

Le SDK Android permet de :

- Installer l'application sur une vraie tablette connectée par USB
- Simuler l'application sur une tablette virtuelle *AVD*

AVD = Android Virtual Device

C'est une machine virtuelle comme celles de VirtualBox et VMware, mais basée sur QEMU.

QEMU est en licence GPL, il permet d'émuler toutes sortes de CPU dont des ARM7, ceux qui font tourner la plupart des tablettes Android.

### 1.4.2. Assistant de création d'une tablette virtuelle

Voir la figure 10, page 28.

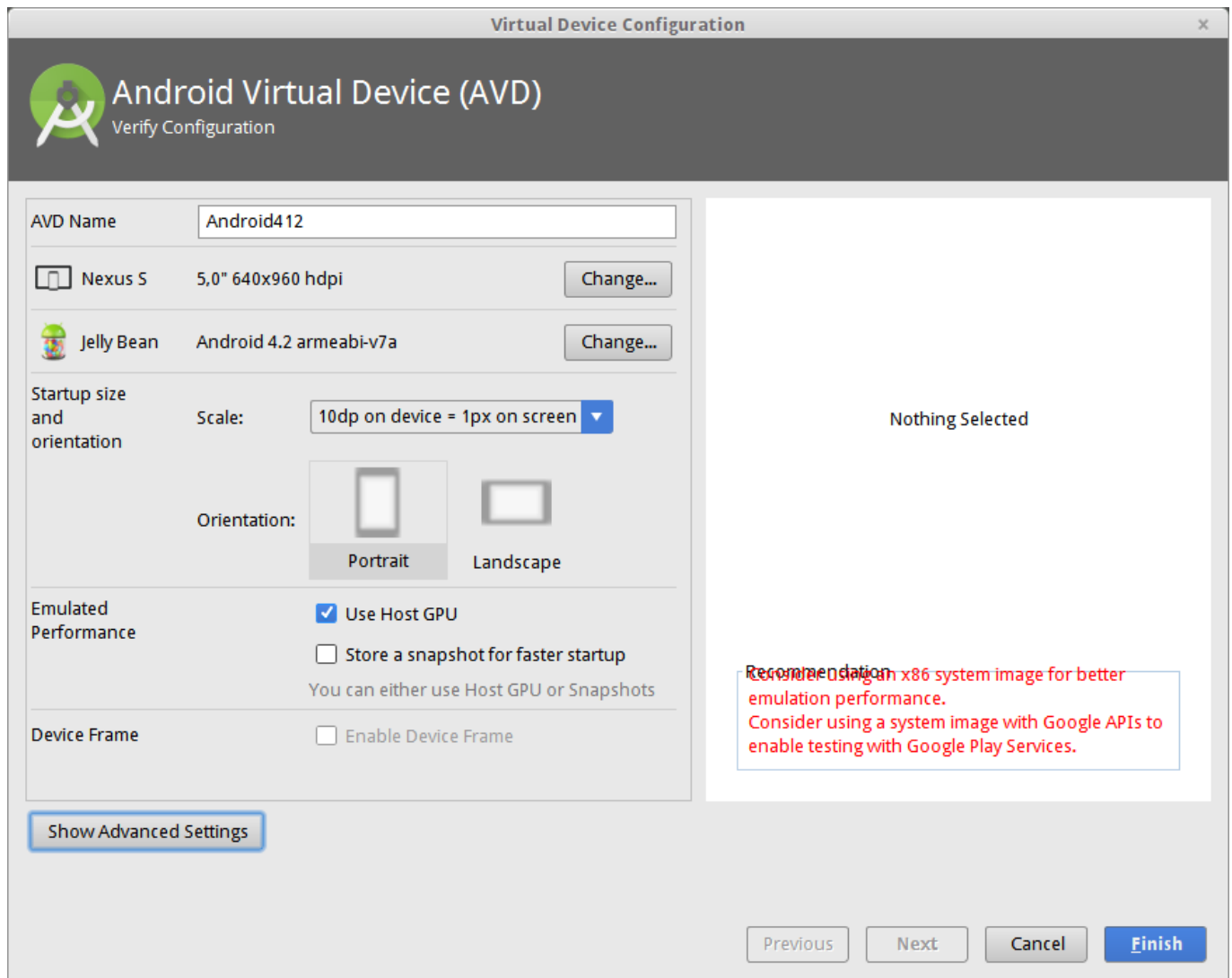


Figure 10: Création d'un AVD

### 1.4.3. Caractéristiques d'un AVD

L'assistant de création de tablette demande :

- Modèle de tablette ou téléphone à simuler,
- Version du système qu'il doit contenir,
- Orientation et densité de l'écran
- Options de simulation :
  - **Snapshot** : mémorise l'état de la machine d'un lancement à l'autre, **mais exclut Use Host GPU**,
  - **Use Host GPU** : accélère les dessins 2D et 3D à l'aide de la carte graphique du PC.
- Options avancées :
  - **RAM** : mémoire à allouer, mais est limitée par votre PC,
  - **Internal storage** : capacité de la flash interne,
  - **SD Card** : capacité de la carte SD simulée supplémentaire (optionnelle).

### 1.4.4. Lancement d'une application

Bouton vert pour exécuter, bleu pour déboguer :

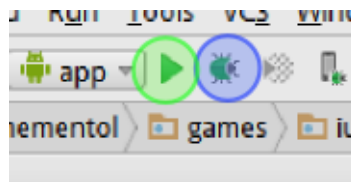


Figure 11: Barre d'outils pour lancer une application

### 1.4.5. Application sur l'AVD

L'apparence change d'une version à l'autre du SDK.

### 1.4.6. Contrôle de l'AVD

Pour simuler les boutons d'une vraie tablette :

**CTRL-M** affiche le menu de l'application

**CTRL-backspace** retour en arrière

**home** retour à l'écran d'accueil

**ctrl-left ctrl-right** rotation paysage/portrait avec les flèches du clavier.

## 1.5. Communication AVD - Android Studio

### 1.5.1. Fenêtres Android

Android Studio affiche plusieurs fenêtres utiles indiquées dans l'onglet tout en bas :

**Android Monitor** Affiche tous les messages émis par la tablette courante

**Console** Messages du compilateur et du studio

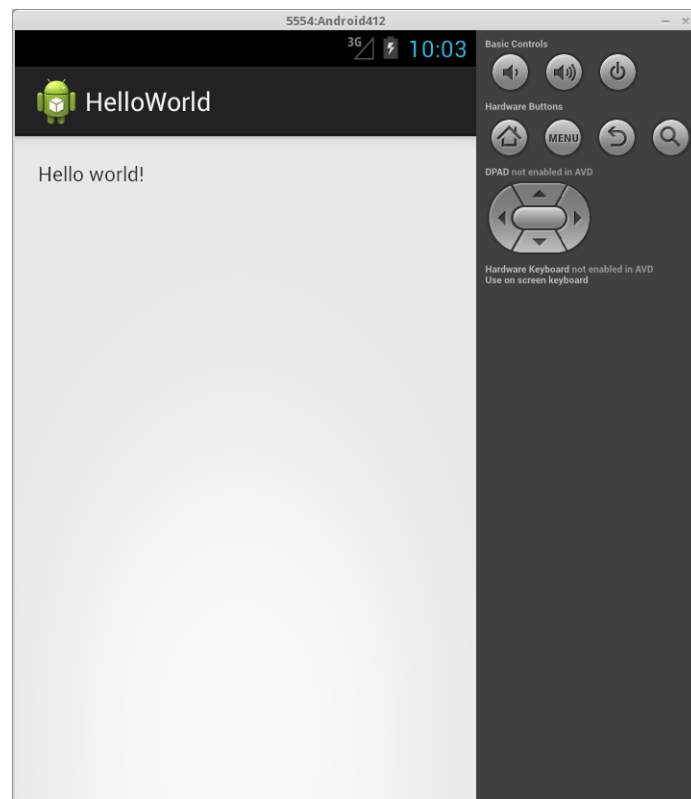


Figure 12: Résultat sur l'AVD

### 1.5.2. Fenêtre LogCat

Des messages détaillés sont affichés dans la fenêtre LogCat :

Ils sont émis par les applications : debug, infos, erreurs...

### 1.5.3. Filtrage des messages

Il est commode de définir des *filtres* pour ne pas voir la totalité des messages de toutes les applications de la tablette :

- sur le niveau de gravité : **verbose**, **debug**, **info**, **warn**, **error** et **assert**,
- sur l'étiquette *TAG* associée à chaque message,
- sur le *package* de l'application qui émet le message.

### 1.5.4. Émission d'un message pour LogCat

Une application émet un message par ces instructions :



```
import android.util.Log;
public class MainActivity extends Activity {
    public static final String TAG = "hello";
    void maMethode() {
        Log.i(TAG, "Salut !");
    }
}
```

Fonctions Log.\* :

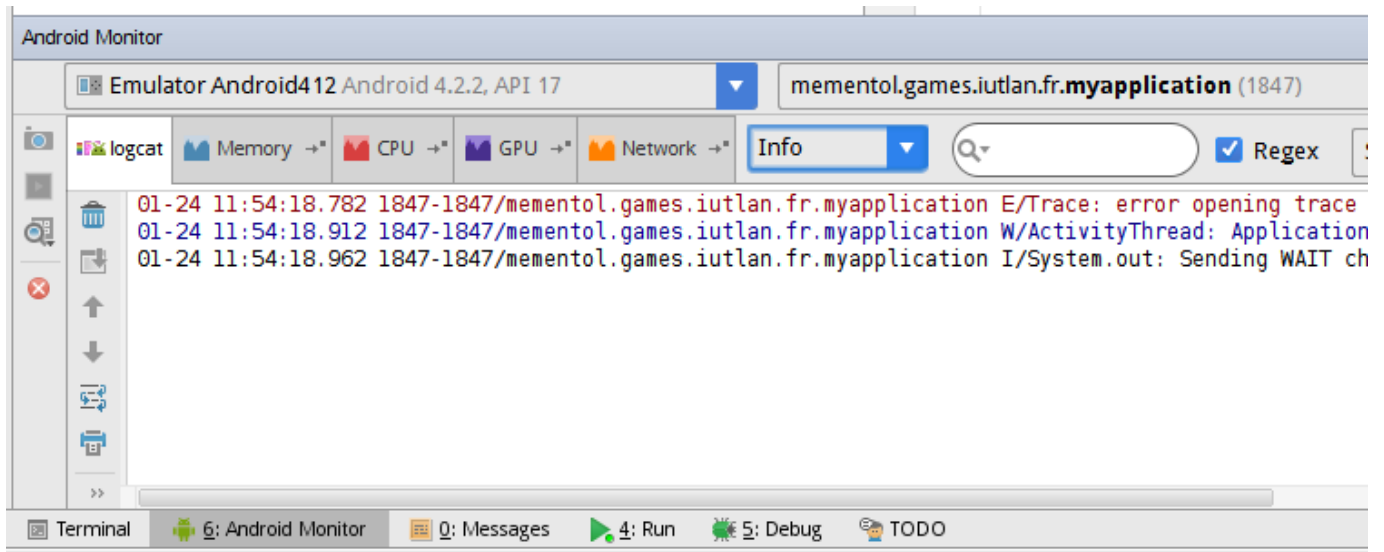


Figure 13: Fenêtre LogCat

- `Log.i(String tag, String message)` affiche une info,
- `Log.w(String tag, String message)` affiche un avertissement,
- `Log.e(String tag, String message)` affiche une erreur.

### 1.5.5. Logiciel ADB

Android Debug Bridge est une passerelle entre une tablette (réelle ou virtuelle) et votre PC

- Serveur de connexion des tablettes
- Commande de communication

ADB emprunte à FTP (transfert de fichiers) et SSH (connexion à un shell).

### 1.5.6. Mode d'emploi de ADB

En ligne de commande : `adb commande paramètres...`

- Gestion du serveur
  - `adb start-server` : démarre le serveur,
  - `adb kill-server` : arrête le serveur,
  - `adb devices` : liste les tablettes connectées.

Exemple :

```
~/CoursAndroid/$ adb devices
List of devices attached
emulator-5554    device
c1608df1b170d4f device
~/CoursAndroid/$
```

### 1.5.7. Mode d'emploi, suite

Chaque tablette (*device*) possède un *identifiant*, ex: `c1608df1b170d4f` ou `emulator-5554` qu'il faut fournir aux commandes `adb` à l'aide de l'option `-s`.

Par défaut, c'est la seule tablette active qui est concernée.

- Connexion à un shell
  - `adb -s identifiant shell commande_unix...`  
exécute la commande sur la tablette
  - `adb -s identifiant shell`  
ouvre une connexion de type shell sur la tablette.

Ce shell est un interpréteur `sh` simplifié (type *busybox*) à l'intérieur du système Unix de la tablette. Il connaît les commandes standard Unix de base : `ls`, `cd`, `cp`, `mv`, `ps`...

### 1.5.8. Système de fichiers Android

On retrouve l'architecture des dossiers Unix, avec des variantes :

- Dossiers Unix classiques : `/usr`, `/dev`, `/etc`, `/lib`, `/sbin`...
- Les volumes sont montés dans `/mnt`, par exemple `/mnt/sdcard` (mémoire flash interne) et `/mnt/extSdCard` (SDcard amovible)
- Les applications sont dans :
  - `/system/app` pour les pré-installées
  - `/data/app` pour les applications normales
- Les données des applications sont dans `/data/data/nom.du.paquetage.java`  
Ex: `/data/data/fr.iutlan.helloworld/...`

NB : il y a des restrictions d'accès sur une vraie tablette, car vous n'y êtes pas *root* ... enfin en principe.

### 1.5.9. Mode d'emploi, suite

- Pour échanger des fichiers avec une tablette :
  - `adb push nom_du_fichier_local /nom/complet/dest`  
envoi du fichier local sur la tablette
  - `adb pull /nom/complet/fichier`  
récupère ce fichier de la tablette
- Pour gérer les logiciels installés :
  - `adb install paquet.apk`
  - `adb uninstall nom.du.paquetage.java`
- Pour archiver les données de logiciels :
  - `adb backup -f fichier_local nom.du.paquetage.java ...`  
enregistre les données du/des logiciels dans le fichier local
  - `adb restore fichier_local`  
restaure les données du/des logiciels d'après le fichier.

## 1.6. Création d'un paquet installable

### 1.6.1. Paquet

Un paquet Android est un fichier `.apk`. C'est une archive signée (authenticifiée) contenant les binaires, ressources compressées et autres fichiers de données.

La création est relativement simple avec Studio :



1. Menu contextuel du projet Build..., choisir **Generate Signed APK**,
2. Signer le paquet à l'aide d'une *clé privée*,
3. Définir l'emplacement du fichier **.apk**.

Le résultat est un fichier **.apk** dans le dossier spécifié.

### 1.6.2. Signature d'une application

Lors de la mise au point, Studio génère une clé qui ne permet pas d'installer l'application ailleurs. Pour distribuer une application, il faut une *clé privée*.

Les clés sont stockées dans un *keystore* = trousseau de clés. Il faut le créer la première fois. C'est un fichier crypté, protégé par un mot de passe, à ranger soigneusement.

Ensuite créer une *clé privée* :

- **alias** = nom de la clé, mot de passe de la clé
- informations personnelles complètes : prénom, nom, organisation, adresse, etc.

Les mots de passe du trousseau et de la clé seront demandés à chaque création d'un **.apk**.

### 1.6.3. Création du *keystore*

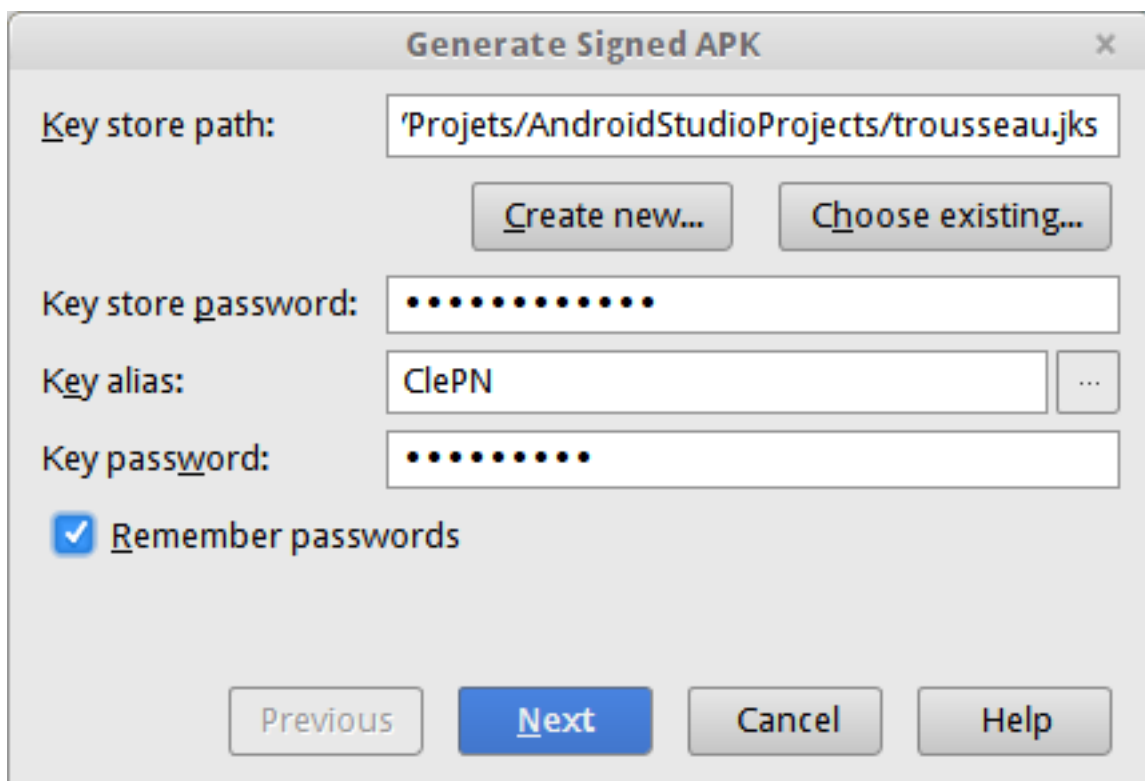


Figure 14: Création d'un trousseau de clés

### 1.6.4. Création d'une clé

Voir la figure 15, page 34.

**New Key Store**

Key store path: /home/pierre/Projets/AndroidStudioProjects/trousseau.jks ...

Password: ..... Confirm: .....

**Key**

Alias: ClePN

Password: ..... Confirm: .....

Validity (years): 25

**Certificate**

First and Last Name: Pierre Nerzic

Organizational Unit: Département Informatique

Organization: IUT de Lannion

City or Locality: Lannion

State or Province: Côtes d'Armor 22

Country Code (XX): FR

OK Cancel

Figure 15: Création d'une clé

### 1.6.5. Création du paquet

Ensuite, Studio demande où placer le .apk :

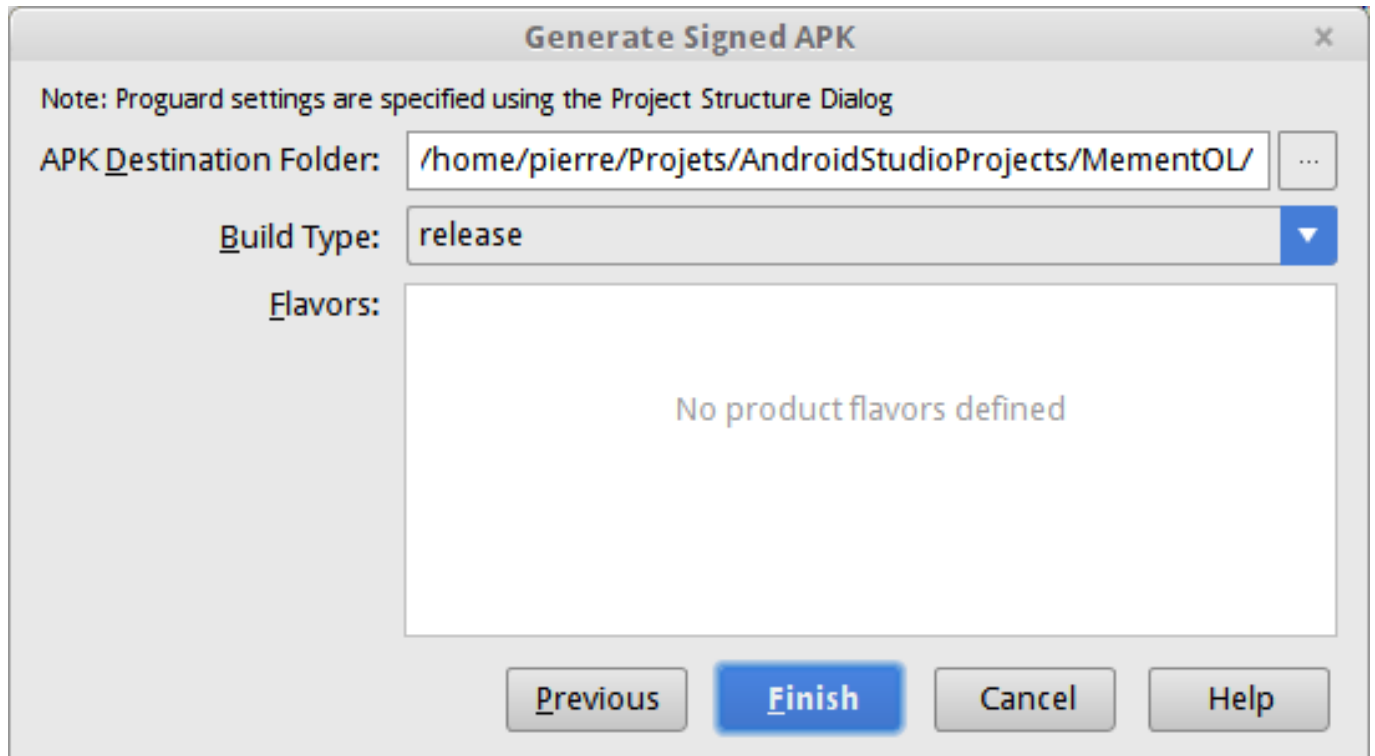


Figure 16: Création du paquet

### 1.6.6. Et voilà

C'est fini pour cette semaine, rendez-vous la semaine prochaine pour un cours sur les interfaces Android.

## Semaine 2

---

### Création d'interfaces utilisateur

---

Le cours de cette semaine explique la création d'interfaces utilisateur :

- Relations entre un source Java et des ressources
- Layouts et vues
- Styles

On ne s'intéresse qu'à la mise en page. L'activité des interfaces sera étudiée la semaine prochaine.

NB: les textes [fuchsia](#) sont des liens cliquables.

## 2.1. Interface et ressources

### 2.1.1. Activités

L'interface utilisateur d'une application Android est composée d'écrans. Un écran correspond à une *activité*, ex :

- afficher une liste d'items
- éditer un item à l'aide d'un formulaire.

Les dialogues et les *pop-up* ne sont pas des activités, ils se superposent temporairement à l'écran d'une activité.

Android permet de naviguer d'une activité à l'autre, ex :

- une action de l'utilisateur, bouton, menu ou l'application fait aller sur l'écran suivant
- le bouton **back** ramène sur l'écran précédent.

### 2.1.2. Création d'un écran

Chaque écran est géré par une instance d'une sous-classe perso de [Activity](#). Sa méthode `onCreate` définit, entre autres, ce qui doit être affiché sur l'écran :

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

`@Override` signifie que `onCreate` surcharge cette méthode de la superclasse et il faut aussi l'appeler sur `super`

### 2.1.3. Identifiant de ressource

La méthode `setContentView` spécifie l'identifiant de l'interface à afficher dans l'écran : `R.layout.main`. C'est un entier, identifiant d'une *disposition de vues* : un *layout*.

Le SDK Android (aapt) construit automatiquement une classe statique appelée `R`. Elle ne contient que des constantes entières :

```
package fr.iutlan.helloworld;
public final class R {
    public static final class id {
        public static final int texte=0x7f080000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
}
```

### 2.1.4. La classe R

Cette classe `R` est générée automatiquement par ce que vous mettez dans le dossier `res` : dispositions, identifiants, chaînes... Certaines de ces ressources sont des fichiers XML, d'autres sont des images PNG.

Par exemple, `res/values/strings.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Exemple</string>
    <string name="message">Bonjour !</string>
</resources>
```

### 2.1.5. Rappel sur la structure d'un fichier XML

Un [fichier XML](#) : nœuds racine, éléments, attributs, valeurs, texte.

```
<?xml version="1.0" encoding="utf-8"?>
<racine>
    <!-- commentaire -->
    <element attribut1="valeur1"
              attribut2="valeur2">
        <feuille1 attribut3="valeur3"/>
        <feuille2>texte</feuille2>
    </element>
    texte en vrac
</racine>
```

Voir le cours [XML](#).

### 2.1.6. Espaces de nommage dans un fichier XML

Dans le cas d'Android, il y a un grand nombre d'éléments et d'attributs normalisés. Pour les distinguer, ils ont été regroupés dans le *namespace* `android`. Dans la norme XML, le namespace par défaut n'est jamais appliqué aux attributs, donc il faut mettre le préfixe sur chacun d'eux.

Vous pouvez lire [cette page](#) et [celle-ci](#) sur les *namespaces*.

```
<menu xmlns:android=
    "http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_settings"
        android:orderInCategory="100"
        android:showAsAction="never"
        android:title="Configuration"/>
</menu>
```

### 2.1.7. Création d'une interface par programme

Il est possible de créer une interface par programme, mais c'est assez compliqué :

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Context ctx = getApplicationContext();
    TextView tv = new TextView(ctx);
    tv.setText("Demat !");
    RelativeLayout rl = new RelativeLayout(ctx);
    LayoutParams lp = new LayoutParams();
    lp.width = LayoutParams.MATCH_PARENT;
    lp.height = LayoutParams.MATCH_PARENT;
    rl.addView(tv, lp);
    setContentView(rl);
}
```

### 2.1.8. Programme et ressources

Il est donc préférable de stocker l'interface dans un fichier `res/layout/main.xml` :

```
<RelativeLayout ...>
    <TextView android:text="Demat !" ... />
</RelativeLayout>
```

qui est référencé par son identifiant `R.layout.nom_du_fichier` (ici c'est `main`) dans le programme Java :

```
protected void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    setContentView(R.layout.main);
}
```

### 2.1.9. Ressources de type chaînes

Dans `res/values/strings.xml`, on place les chaînes de l'application, au lieu de les mettre en constantes dans le source :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">HelloWorld</string>
    <string name="main_menu">Menu principal</string>
    <string name="action_settings">Configuration</string>
    <string name="bonjour">Demat !</string>
</resources>
```

Intérêt : pouvoir traduire une application sans la recompiler.

### 2.1.10. Traduction des chaînes (*localisation*)

Lorsque les textes sont définis dans `res/values/strings.xml`, il suffit de faire des copies du dossier `values`, en `values-us`, `values-fr`, `values-de`, etc. et de traduire les textes en gardant les attributs `name`. Voici par exemple `res/values-de/strings.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">HelloWorld</string>
    <string name="main_menu">Hauptmenü</string>
    <string name="action_settings">Einstellungen</string>
    <string name="bonjour">Guten Tag</string>
</resources>
```

Le système android ira chercher automatiquement le bon texte en fonction des paramètres linguistiques configurés par l'utilisateur.

### 2.1.11. Référencement des ressources texte

Voici comment affecter une ressource chaîne à une vue en Java :

```
TextView tv = new TextView(ctx);
tv.setText(R.string.bonjour);
```

`R.string.bonjour` désigne le texte de `<string name="bonjour">...` dans le fichier `res/values*/strings.xml`.

Voici comment spécifier un titre de label dans un `layout.xml` :

```
<RelativeLayout>
    <TextView android:text="@string/bonjour" />
</RelativeLayout>
```

`@string/nom` est une référence à une ressource, la chaîne de `res/values/strings.xml` ayant ce nom.

### 2.1.12. Identifiants et vues

La méthode `setContentView` fait afficher le formulaire défini par l'identifiant `R.layout` indiqué. Lorsque l'application veut manipuler l'une de ses vues, elle doit faire utiliser `R.id.symbole`, ex :

```
TextView tv = (TextView) findViewById(R.id.message);
```

NB: remarquez la conversion de type, `findViewById` retourne une `View`, superclasse de `TextView`. avec la définition suivante dans `res/layout/main.xml` :

```
<RelativeLayout>
    <TextView android:id="@+id/message"
        android:text="@string/bonjour" />
</RelativeLayout>
```

La notation `@id/nom` définit un identifiant pour le `TextView`.

### 2.1.13. @id/nom ou @+id/nom ?

Il y a les deux notations :

**@id/nom** pour référencer un identifiant déjà défini (ailleurs)

**@+id/nom** pour définir (créer) cet identifiant

Exemple, le `Button` `btn` désigne le `TextView` `titre` :

```
<RelativeLayout xmlns:android="..." ... >
    <TextView ...
        android:id="@+id/titre"
        android:text="@string/titre" />
    <Button ...
        android:id="@+id/btn"
        android:layout_below="@id/titre"
        android:text="@string/ok" />
</RelativeLayout>
```

### 2.1.14. Images : R.drawable.nom

De la même façon, les images PNG placées dans `res/drawable` et `res/mipmaps-*` sont référençables :

```
<ImageView
    android:src="@drawable/velo"
    android:contentDescription="@string/mon_velo" />
```

La notation `@drawable/nom` référence l'image portant ce nom dans l'un des dossiers.

NB: les dossiers `res/mipmaps-*` contiennent la même image à des définitions différentes, pour correspondre à différents téléphones et tablettes. Ex: `mipmap-hdpi` contient des icônes en 72x72 pixels.



### 2.1.15. Tableau de chaînes : `R.array.nom`

Voici un extrait du fichier `res/values/arrays.xml` :

```
<resources>
  <string-array name="planetes">
    <item>Mercure</item>
    <item>Venus</item>
    <item>Terre</item>
    <item>Mars</item>
  </string-array>
</resources>
```

Dans le programme Java, il est possible de faire :

```
Resources res = getResources();
String[] planetes = res.getStringArray(R.array.planetes);
```

### 2.1.16. Autres

D'autres notations existent :

- `@style/nom` pour des définitions de `res/style`
- `@menu/nom` pour des définitions de `res/menu`

Certaines notations, `@package:type/nom` font référence à des données prédéfinies, comme :

- `@android:style/TextAppearance.Large`
- `@android:color/black`

Il y a aussi une notation en `?type/nom` pour référencer la valeur de l'attribut `nom`, ex : `?android:attr/textColorSecondary`.

## 2.2. Dispositions

### 2.2.1. Structure d'une interface Android

Un écran Android de type formulaire est généralement composé de plusieurs vues. Entre autres :

**TextView**, **ImageView** titre, image

**EditText** texte à saisir

**Button**, **CheckBox** bouton à cliquer, case à cocher

Ces vues sont alignées à l'aide de groupes sous-classes de **ViewGroup**, éventuellement imbriqués :

**LinearLayout** positionne ses vues en ligne ou colonne

**RelativeLayout** positionne ses vues l'une par rapport à l'autre

**TableLayout** positionne ses vues sous forme d'un tableau

### 2.2.2. Arbre des vues

Les groupes et vues forment un arbre :

Voir la figure 17, page 42.

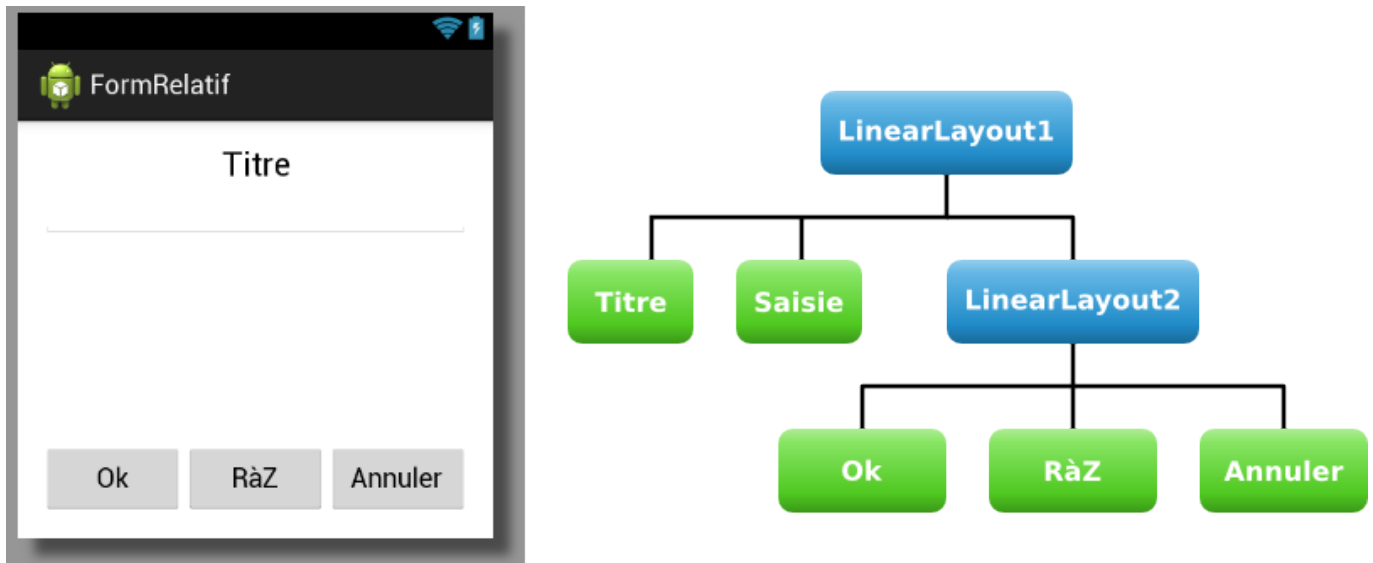


Figure 17: Arbre de vues

### 2.2.3. Représentation en XML

Cet arbre s'écrit en XML :

```
<LinearLayout android:id="@+id/groupe1" ...>
  <TextView android:id="@+id/titre" .../>
  <EditText android:id="@+id/saisie" .../>
  <LinearLayout android:id="@+id/groupe2" ...>
    <Button android:id="@+id/ok" .../>
    <Button android:id="@+id/raz" .../>
    <Button android:id="@+id/annuler" .../>
  </LinearLayout>
</LinearLayout>
```

### 2.2.4. Paramètres de positionnement

La plupart des groupes utilisent des *paramètres de placement* sous forme d'attributs XML. Par exemple, telle vue à droite de telle autre, telle vue la plus grande possible, telle autre la plus petite.

Ces paramètres sont de deux sortes :

- ceux qui sont demandés pour toutes les vues, par exemple `android:layout_width`, `android:layout_height` et `android:layout_weight`
- ceux qui sont demandés par le groupe englobant et qui en sont spécifiques, comme `android:layout_alignParentBottom`, `android:layout_centerInParent`...

### 2.2.5. Paramètres généraux

Toutes les vues doivent spécifier ces deux attributs :

**`android:layout_width`** largeur de la vue

**`android:layout_height`** hauteur de la vue

Ils peuvent valoir :

- "wrap\_content" : la vue est la plus petite possible
- "match\_parent" : la vue est la plus grande possible
- "valeurdp" : une taille fixe, ex : "100dp" mais c'est peu recommandé

Les **dp** sont une unité de taille indépendante de l'écran. 100dp font 100 pixels sur un écran de 100 dpi (100 *dots per inch*) tandis qu'ils font 200 pixels sur un écran 200dpi. Ça fait la même taille apparente.

### 2.2.6. Autres paramètres géométriques

Il est possible de modifier l'espacement des vues :

**Padding** espace entre le texte et les bords, géré par chaque vue

**Margin** espace autour des bords, géré par les groupes

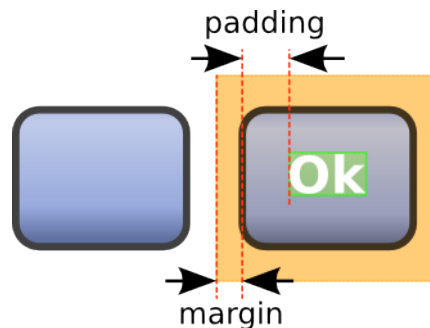


Figure 18: Bords et marges

### 2.2.7. Marges et remplissage

On peut définir les marges et les remplissages séparément sur chaque bord (Top, Bottom, Left, Right), ou identiquement sur tous : 

```
<Button
    android:layout_margin="10dp"
    android:layout_marginTop="15dp"
    android:padding="10dp"
    android:paddingLeft="20dp" />
```

### 2.2.8. Groupe de vues LinearLayout

Il range ses vues soit horizontalement, soit verticalement 

```
<LinearLayout android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <Button android:text="Ok" android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Button android:text="Annuler" android:layout_weight="1"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</LinearLayout>
```

Il faut seulement définir l'attribut `android:orientation` à "horizontal" ou "vertical". Lire la [doc Android](#).

### 2.2.9. Pondération des tailles

Une façon intéressante de spécifier les tailles des vues dans un `LinearLayout` consiste à leur affecter un *poids* avec l'attribut `android:layout_weight`.

- Un `layout_weight` égal à 0 rend la vue la plus petite possible
- Un `layout_weight` non nul donne une taille correspondant au rapport entre ce poids et la somme des poids des autres vues

Pour cela, il faut aussi fixer la taille de ces vues (ex : `android:layout_width`) soit à "wrap\_content", soit à "0dp". Si la taille vaut "wrap\_content", alors le poids agit seulement sur l'espace supplémentaire alloué aux vues. Mettre "0dp" pour que ça agisse sur la taille entière.

### 2.2.10. Exemple de poids différents

Voici 4 `LinearLayout` horizontaux de 3 boutons ayant des poids égaux à leurs titres. En 3<sup>e</sup> ligne, les boutons ont une largeur de 0dp



Figure 19: Influence des poids sur la largeur

### 2.2.11. Groupe de vues `TableLayout`

C'est une variante du `LinearLayout` : les vues sont rangées en lignes de colonnes bien tabulées. Il faut construire une structure XML comme celle-ci. Voir sa [doc Android](#).

```
<TableLayout ...>
  <TableRow>
    <item 1.1 .../>
    <item 1.2 .../>
  </TableRow>
  <TableRow>
    <item 2.1 .../>
    <item 2.2 .../>
  </TableRow>
</TableLayout>
```

NB : les `<TableRow>` n'ont aucun attribut.

### 2.2.12. Largeur des colonnes d'un `TableLayout`

Ne pas spécifier `android:layout_width` dans les vues d'un `TableLayout`, car c'est obligatoirement toute la largeur du tableau. Seul la balise `<TableLayout>` exige cet attribut.

Deux propriétés intéressantes permettent de rendre certaines colonnes étirables. Fournir les numéros (première = 0).

- `android:stretchColumns` : numéros des colonnes étirables
- `android:shrinkColumns` : numéros des colonnes reductibles

```
<TableLayout
  android:stretchColumns="1,2"
  android:shrinkColumns="0,3"
  android:layout_width="match_parent"
  android:layout_height="wrap_content" >
```

### 2.2.13. Groupe de vues `RelativeLayout`

C'est le plus complexe à utiliser mais il donne de bons résultats. Il permet de spécifier la position relative de chaque vue à l'aide de *paramètres* complexes : ([LayoutParams](#))

- Tel bord aligné sur le bord du parent ou centré dans son parent :
  - `android:layout_alignParentTop`, `android:layout_centerVertical...`
- Tel bord aligné sur le bord opposé d'une autre vue :
  - `android:layout_toRightOf`, `android:layout_above`, `android:layout_below...`
- Tel bord aligné sur le même bord d'une autre vue :
  - `android:layout_alignLeft`, `android:layout_alignTop...`

### 2.2.14. Utilisation d'un `RelativeLayout`

Pour bien utiliser un [RelativeLayout](#), il faut commencer par définir les vues qui ne dépendent que des bords du Layout : celles qui sont collées aux bords ou centrées.

```
<TextView android:id="@+id/titre"  
    android:layout_alignParentTop="true"  
    android:layout_alignParentRight="true"  
    android:layout_alignParentLeft="true" .../>
```

Puis créer les vues qui dépendent des vues précédentes.

```
<EditText android:layout_below="@id/titre"  
    android:layout_alignParentRight="true"  
    android:layout_alignParentLeft="true" .../>
```

Et ainsi de suite.

### 2.2.15. Autres groupements

Ce sont les sous-classes de [ViewGroup](#) également présentées dans [cette page](#). Impossible de faire l'inventaire dans ce cours. C'est à vous d'aller explorer en fonction de vos besoins.

## 2.3. Composants d'interface

### 2.3.1. Vues

Android propose un grand nombre de vues, à découvrir en TP :

- Textes : titres, saisies
- Boutons, cases à cocher
- Curseurs

Beaucoup ont des variantes. Ex: saisie de texte = n° de téléphone ou adresse ou texte avec suggestion ou ...

Consulter la doc en ligne de toutes ces vues. On les trouve dans le package [android.widget](#).

À noter que les vues évoluent avec les versions d'Android, certaines changent, d'autres disparaissent.

### 2.3.2. TextView

Le plus simple, il affiche un texte statique, comme un titre. Son libellé est dans l'attribut `android:text`.

```
<TextView  
    android:id="@+id/tvtitre"  
    android:text="@string/titre"  
    ... />
```

On peut le changer dynamiquement :



```
TextView tvTitre = (TextView) findViewById(R.id.tvtitre);  
tvTitre.setText("blablabla");
```

### 2.3.3. Button

L'une des vues les plus utiles est le **Button** :

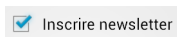


```
<Button  
    android:id="@+id/btn_ok"  
    android:text="@string/ok"  
    ... />
```

- En général, on définit un identifiant pour chaque vue active, ici : `android:id="@+id/btn_ok"`
- Son titre est dans l'attribut `android:text`.
- Voir la semaine prochaine pour son activité : réaction à un clic.

### 2.3.4. Bascules

Les **CheckBox** sont des cases à cocher :



```
<CheckBox  
    android:id="@+id/cbx_abonnement_nl"  
    android:text="@string/abonnement_newsletter"  
    ... />
```

Les **ToggleButton** sont une variante :



. On peut définir le texte actif et le texte inactif avec `android:textOn` et `android:textOff`.

### 2.3.5. EditText

Un **EditText** permet de saisir un texte  :

```
<EditText  
    android:id="@+id/email_address"  
    android:inputType="textEmailAddress"  
    ... />
```

L'attribut `android:inputType` spécifie le type de texte : adresse, téléphone, etc. Ça définit le clavier qui est proposé pour la saisie.

Lire [la référence Android](#) pour connaître toutes les possibilités.

### 2.3.6. Autres vues

On reviendra sur toutes ces vues les prochaines semaines, pour préciser les attributs utiles pour une application. D'autres vues pourront aussi être employées à l'occasion.

## 2.4. Styles

### 2.4.1. Styles et thèmes

Un *style* permet de modifier l'apparence d'une vue :

- Police de caractères et tailles pour les textes
- Couleurs, images...
- Géométrie par défaut des vues : taille, espacement, remplissage...

Un *thème* est un style appliqué à toute une activité ou application.

Consulter la [documentation Android](#).

### 2.4.2. Définir un style

Il faut créer un fichier XML dans `res/value` :



```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="Elegant"
        parent="@android:style/TextAppearance.Medium">
    <item name="android:textColor">#010101</item>
    <item name="android:typeface">serif</item>
  </style>
</resources>
```

L'attribut `name` identifie le style, et `parent` le rattache à un autre pour héritage des propriétés non définies ici. Voir [les styles](#) et [les thèmes](#) prédéfinis.

### 2.4.3. Utiliser un style

Il suffit de le préciser dans la définition de la vue :



```
<TextView
  style="@style/Elegant"
  android:text="@string/titre" />
```

### 2.4.4. Utiliser un thème

Un thème est simplement un style appliqué partout dans l'application. Cela se spécifie dans le fichier `AndroidManifest.xml` :



```
<application
  android:theme="@style/Elegant"
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  ...>
  ...
</application>
```

Attention, si votre style n'est pas complet, vous aurez une erreur.



### **2.4.5. C'est tout**

C'est fini pour cette semaine, rendez-vous la semaine prochaine pour un cours sur les écouteurs et les activités.

## Semaine 3

---

### Vie d'une application

---

Le cours de cette semaine concerne la vie d'une application :

- Applications et activités, manifeste : [bibliographie](#)
- Cycles de vie : [voir cette page](#)
- Vues, événements et écouteurs : [voir ce lien](#) et [celui-ci](#)

### 3.1. Applications et activités

#### 3.1.1. Composition d'une application

Une application est composée de plusieurs *activités*. Chacune gère un écran d'interaction avec l'utilisateur et est définie par une classe Java.

Une application complexe peut aussi contenir :

- des *services* : ce sont des processus qui tournent en arrière-plan,
- des *fournisseurs de contenu* : ils représentent une sorte de base de données, voir la semaine 5,
- des *récepteurs d'annonces* : pour gérer des événements globaux envoyés par le système à toutes les applications.

#### 3.1.2. Déclaration d'une application

Le fichier `AndroidManifest.xml` déclare les éléments d'une application, avec un '.' devant le nom des activités

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
  <application android:icon="@drawable/app_icon.png" ...>
    <activity android:name=".MainActivity"
      ... />
    <activity android:name=".EditActivity"
      ... />
    ...
  </application>
</manifest>
```

`<application>` est la seule branche sous la racine `<manifest>` et ses filles sont des `<activity>`.

### 3.1.3. Sécurité des applications

Chaque application est associée à un UID (compte utilisateur Unix) unique dans le système. Ce compte les protège les unes des autres. Cet UID peut être défini dans le fichier AndroidManifest.xml :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...
    android:sharedUserId="fr.iutlan.demos">
    ...
</manifest>
```

Définir l'attribut `android:sharedUserId` avec une chaîne identique à une autre application, et signer les deux applications avec le même certificat, permet à l'une d'accéder à l'autre.

### 3.1.4. Autorisations d'une application

Une application doit déclarer les autorisations dont elle a besoin : accès à internet, caméra, carnet d'adresse, GPS, etc.

Cela se fait en rajoutant des éléments dans le manifeste :

```
<manifest ... >
    <uses-permission
        android:name="android.permission.INTERNET" />
    ...
</manifest>
```

Consulter [cette page](#) pour la liste des permissions existantes.

### 3.1.5. Démarrage d'une application

L'une des activités est marquée comme démarrable de l'extérieur :

```
<activity android:name=".MainActivity" ...>
    <intent-filter>
        <action android:name=
            "android.intent.action.MAIN" />
        <category android:name=
            "android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Un `<intent-filter>` déclare les conditions de démarrage d'une activité, ici il dit que c'est l'activité principale.

### 3.1.6. Démarrage d'une activité et Intents

Les activités sont démarrées à l'aide d'**Intents**. Un **Intent** contient une demande destinée à une activité, par exemple, composer un numéro de téléphone ou lancer l'application.

- *action* : spécifie ce que l'**Intent** demande. Il y en a de [très nombreuses](#) :
  - **VIEW** pour afficher quelque chose, **EDIT** pour modifier une information, **SEARCH**...
- *données* : selon l'action, ça peut être un numéro de téléphone, l'identifiant d'une information...
- *catégorie* : information supplémentaire sur l'action, par exemple, ...**LAUNCHER** pour lancer une application.

Une application a la possibilité de lancer certaines activités d'une autre application, celles qui ont un **intent-filter**.

### 3.1.7. Lancement d'une activité par programme

Soit une application contenant deux activités : **Activ1** et **Activ2**. La première lance la seconde par :



```
Intent intent = new Intent(this, Activ2.class);
startActivity(intent);
```

L'instruction **startActivity** démarre **Activ2**. Celle-ci se met devant **Activ1** qui se met alors en sommeil.

Ce bout de code est employé par exemple lorsqu'un bouton, un menu, etc. est cliqué. Seule contrainte : que ces deux activités soient déclarées dans **AndroidManifest.xml**.

### 3.1.8. Lancement d'une application Android

Il n'est pas possible de montrer toutes les possibilités, mais par exemple, voici comment ouvrir le navigateur sur un URL spécifique :



```
String url =
    "https://perso.univ-rennes1.fr/pierre.nerzic/Android";
intent = new Intent(Intent.ACTION_VIEW, Uri.parse(url));
startActivity(intent);
```

L'action **VIEW** avec un *URI* (généralisation d'un *URL*) est interprétée par Android, cela fait ouvrir automatiquement le navigateur.

### 3.1.9. Lancement d'une activité d'une autre application

Soit une seconde application dans le package **fr.iutlan.appli2**. Une activité peut la lancer ainsi :



```
intent = new Intent(Intent.ACTION_MAIN);
intent.addCategory(Intent.CATEGORY_LAUNCHER);
intent.setClassName(
    "fr.iutlan.appli2",
```

```
"fr.iutlan.appli2.MainActivity");  
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);  
startActivity(intent);
```

Cela consiste à créer un `Intent` d'action `MAIN` et de catégorie `LAUNCHER` pour la classe `MainActivity` de l'autre application.

## 3.2. Applications

### 3.2.1. Fonctionnement d'une application

Au début, le système Android lance l'activité qui est marquée action=`MAIN` et catégorie=`LAUNCHER` dans `AndroidManifest.xml`.

Ensuite, d'autres activités peuvent être démarrées. Chacune se met « devant » les autres comme sur une pile. Deux cas sont possibles :

- La précédente activité se termine, on ne revient pas dedans.  
Par exemple, une activité où on tape son login et son mot de passe lance l'activité principale et se termine.
- La précédente activité attend la fin de la nouvelle car elle lui demande un résultat en retour.  
Exemple : une activité de type liste d'items lance une activité pour éditer un item quand on clique longuement dessus, mais attend la fin de l'édition pour rafraîchir la liste.

### 3.2.2. Navigation entre activités

Voici un schéma illustrant les possibilités de navigation parmi plusieurs activités.

Voir la figure 20, page 54.

### 3.2.3. Lancement sans attente

Rappel, pour lancer `Activ2` à partir de `Activ1` :



```
Intent intent = new Intent(this, Activ2.class);  
startActivity(intent);
```

On peut demander la terminaison de `this` après lancement de `Activ2` ainsi :



```
Intent intent = new Intent(this, Activ2.class);  
startActivity(intent);  
finish();
```

`finish()` fait terminer l'activité courante. L'utilisateur ne pourra pas faire back dessus, car elle disparaît de la pile.

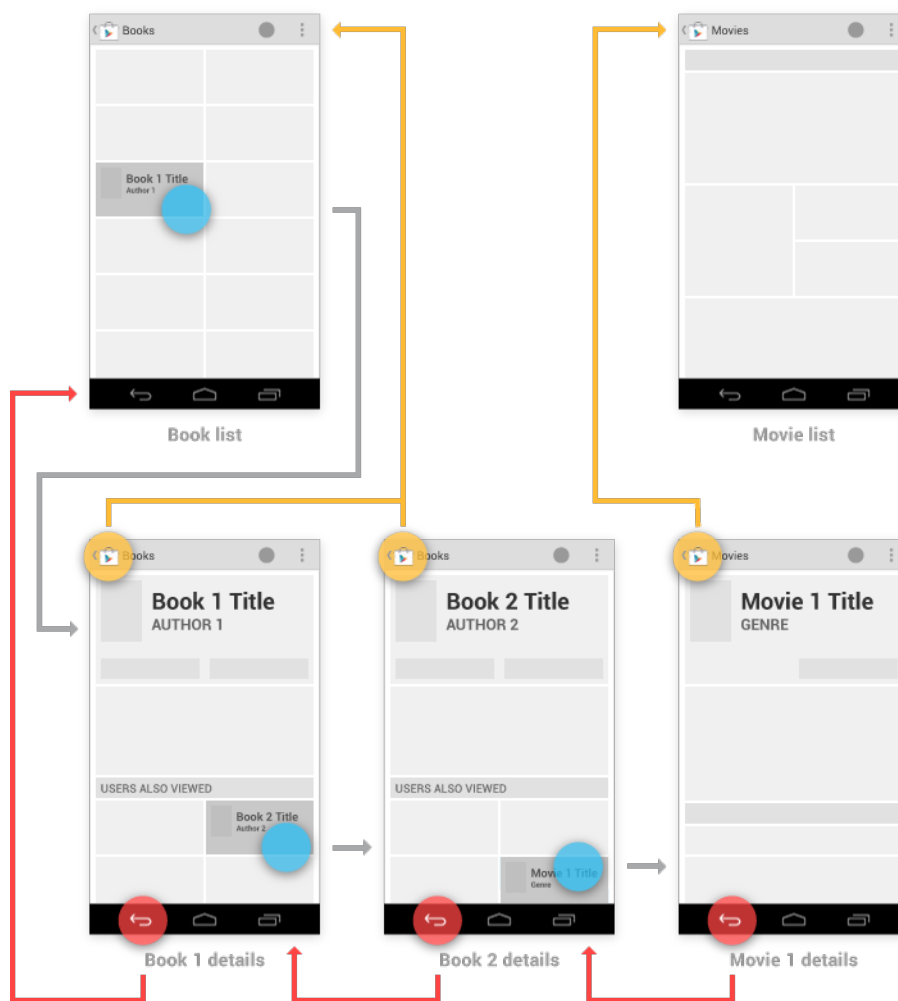



Figure 20: Navigation parmi les activités d'une application

### 3.2.4. Lancement avec attente de résultat

Le lancement d'une activité avec attente de résultat est plus complexe. Il faut définir un *code d'appel* `requestCode` fourni au lancement. 

```
private static final int APPEL_ACTIV2 = 1;
Intent intent = new Intent(this, Activ2.class);
startActivityForResult(intent, APPEL_ACTIV2);
```

Ce code identifie l'activité lancée, afin de savoir plus tard que c'est d'elle qu'on revient. Par exemple, on pourrait lancer au choix plusieurs activités : édition, copie, suppression d'informations. Il faut pouvoir les distinguer au retour.

Consulter [cette page](#).

### 3.2.5. Lancement avec attente, suite

Ensuite, il faut définir une méthode *callback* qui est appelée lorsqu'on revient dans notre activité : 

```
@Override
protected void onActivityResult(
    int requestCode, int resultCode, Intent data)
{
    // uti a fait back
    if (resultCode == Activity.RESULT_CANCELED) return;
    // selon le code d'appel
    switch (requestCode) {
        case APPEL_ACTIV2: // on revient de Activ2
            ...
    }
}
```

### 3.2.6. Terminaison d'une activité

L'activité lancée par la première peut se terminer pour deux raisons :

- Volontairement, en appelant la méthode `finish()` : 

```
setResult(RESULT_OK);
finish();
```

- À cause du bouton « back » du téléphone, son action revient à faire ceci : 

```
setResult(RESULT_CANCELED);
finish();
```

Dans ces deux cas, on revient dans l'activité appelante (sauf si elle-même avait fait `finish()`).

### 3.2.7. Méthode `onActivityResult`

Quand on revient dans l'activité appelante, Android lui fait exécuter cette méthode :

```
onActivityResult(int requestCode, int resultCode, Intent data)
```

- `requestCode` est le code d'appel de `startActivityForResult`
- `resultCode` vaut soit `RESULT_CANCELED` soit `RESULT_OK`, voir le transparent précédent
- `data` est fourni par l'activité appelée et qui vient de se terminer.

Ces deux dernières viennent d'un appel à `setResult(resultCode, data)`

### 3.2.8. Transport d'informations dans un `Intent`

Les `Intent` servent aussi à transporter des informations d'une activité à l'autre : les *extras*.

Voici comment placer des données dans un `Intent` :



```
Intent intent =  
    new Intent(this, DeleteInfoActivity.class);  
intent.putExtra("idInfo", idInfo);  
intent.putExtra("hiddencopy", hiddencopy);  
startActivity(intent);
```

`putExtra(nom, valeur)` rajoute un couple (nom, valeur) dans l'intent. La valeur doit être *sérialisable* : nombres, chaînes et structures simples.

### 3.2.9. Extraction d'informations d'un `Intent`

Ces instructions récupèrent les données d'un `Intent` :



```
Intent intent = getIntent();  
Integer idInfo = intent.getIntExtra("idInfo", -1);  
bool hidden = intent.getBooleanExtra("hiddencopy", false);
```

- `getIntent()` retourne l'`Intent` qui a démarré cette activité.
- `getTypeExtra(nom, valeur par défaut)` retourne la valeur de ce nom si elle en fait partie, la valeur par défaut sinon.

### 3.2.10. Contexte d'application

Pour finir sur les applications, il faut savoir qu'il y a un objet global vivant pendant tout le fonctionnement d'une application : le contexte d'application. Voici comment le récupérer :



```
Application context = this.getApplicationContext();
```

Par défaut, c'est un objet neutre ne contenant que des informations Android.

Il est possible de le sous-classer afin de stocker des variables globales de l'application.



### 3.2.11. Définition d'un contexte d'application

Pour commencer, dériver une sous-classe de Application :



```
public class MonApplication extends Application
{
    // variable globale de l'application
    public int varglob;

    // initialisation du contexte
    @Override
    public void onCreate()
    {
        super.onCreate();
        varglob = 3;
    }
}
```

### 3.2.12. Définition d'un contexte d'application, suite

Ensuite, la déclarer dans AndroidManifest.xml, dans l'attribut android:name de l'élément <application>, mettre un point devant :

```
<manifest xmlns:android="..." ...>
    <application android:name=".MonApplication"
        android:icon="@drawable/icon"
        android:label="@string/app_name">
        ...
    </application>
</manifest>
```

### 3.2.13. Définition d'un contexte d'application, fin

Enfin, l'utiliser dans n'importe laquelle des activités :



```
// récupérer le contexte d'application
MonApplication context =
    (MonApplication) this.getApplicationContext();

// utiliser la variable globale
... context.varglob ...
```

Remarquez la conversion de type.

Il est recommandé de définir des *setters* et *getters*. D'autre part, attention aux variables globales : ne les utiliser qu'à bon escient.

## 3.3. Activités

### 3.3.1. Présentation

Voyons maintenant comment fonctionnent les activités.

- Démarrage (à cause d'un `Intent`)
- Apparition/masquage sur écran
- Terminaison

Une activité se trouve dans l'un de ces états :

- active (*resumed*) : elle est sur le devant, l'utilisateur peut jouer avec,
- en pause (*paused*) : partiellement cachée et inactive, car une autre activité est venue devant,
- stoppée (*stopped*) : totalement invisible et inactive, ses variables sont préservées mais elle ne tourne plus.

### 3.3.2. Cycle de vie d'une activité

Ce diagramme résume les changements d'états d'une activité :

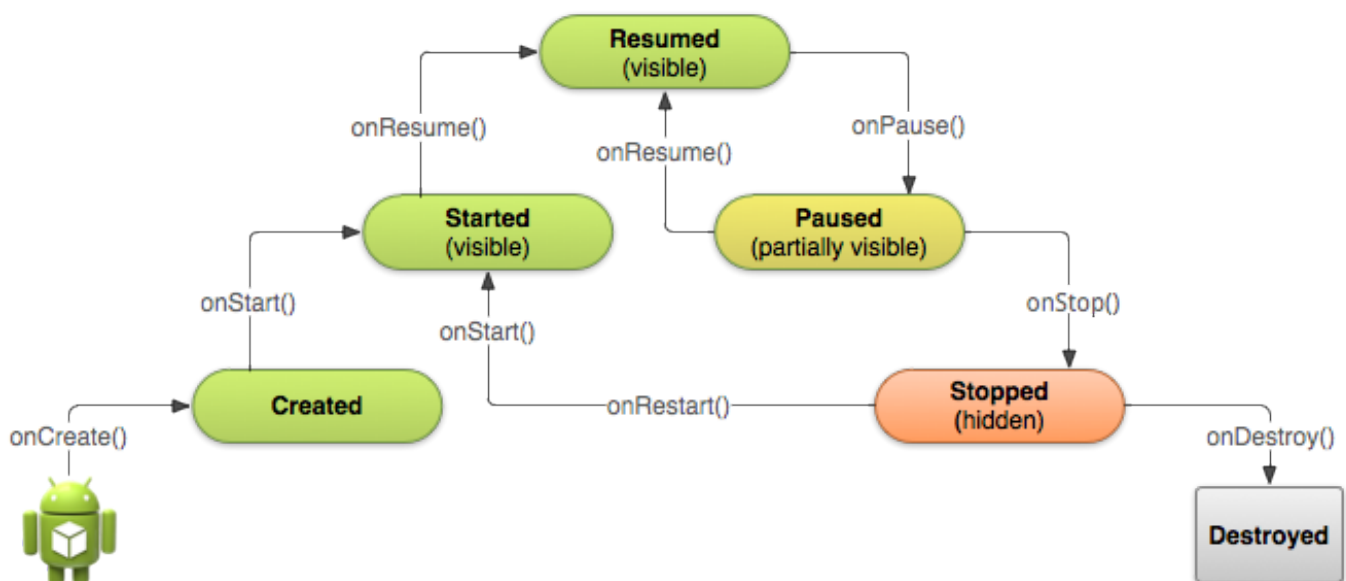


Figure 21: Cycle de vie

### 3.3.3. Événements de changement d'état

La classe `Activity` reçoit des événements de la part du système Android, ça appelle des fonctions appelées *callbacks*.

Exemples :

**onCreate** Un `Intent` arrive dans l'application, il déclenche la création d'une activité, dont l'interface.

**onPause** Le système prévient l'activité qu'une autre activité est passée devant, il faut enregistrer les informations au cas où l'utilisateur ne revienne pas.

### 3.3.4. Squelette d'activité



```
public class EditActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // met en place les vues de cette activité
        setContentView(R.layout.edit_activity);
    }
}
```

`@Override` signifie que cette méthode remplace celle héritée de la superclasse. Il faut quand même l'appeler sur `super` en premier.

### 3.3.5. Terminaison d'une activité

Voici la prise en compte de la terminaison définitive d'une activité, avec la fermeture d'une base de données :



```
@Override
public void onDestroy() {
    super.onDestroy();

    // fermer la base
    db.close();
}
```

En fait, il se peut que cette méthode ne soit jamais appelée. Voir `onStop` plutôt.

### 3.3.6. Pause d'une activité

Cela arrive quand une nouvelle activité passe devant, exemple : un appel téléphonique. Il faut libérer les ressources qui consomment de l'énergie (animations, GPS...).



```
@Override public void onPause() {
    super.onPause();
    // arrêter les animations sur l'écran
    ...
}
@Override public void onResume() {
    super.onResume();
    // démarrer les animations
    ...
}
```

### 3.3.7. Arrêt d'une activité


Cela se produit quand l'utilisateur change d'application dans le sélecteur d'applications, ou qu'il change d'activité dans votre application. Cette activité n'est plus visible et doit enregistrer ses données.

Il y a deux méthodes concernées :

- `protected void onStop()` : l'application est arrêtée, libérer les ressources,
- `protected void onStart()` : l'application démarre, allouer les ressources.

Il faut comprendre que les utilisateurs peuvent changer d'application à tout moment. La votre doit être capable de résister à ça.


### 3.3.8. Enregistrement de valeurs d'une exécution à l'autre

Il est possible de sauver des informations d'un lancement à l'autre de l'application (certains cas comme la rotation de l'écran ou une interruption par une autre activité), dans un `Bundle`. C'est un container de données quelconques, sous forme de couples ("nom", valeur). 

```
static final String ETAT_SCORE = "ScoreJoueur"; // nom
private int mScoreJoueur = 0; // valeur

@Override
public void onSaveInstanceState(Bundle etat) {
    // enregistrer l'état courant
    etat.putInt(ETAT_SCORE, mScoreJoueur);
    super.onSaveInstanceState(etat);
}
```

### 3.3.9. Restaurer l'état au lancement

La méthode `onRestoreInstanceState` reçoit un paramètre de type `Bundle` (comme la méthode `onCreate`, mais dans cette dernière, il peut être `null`). Il contient l'état précédemment sauvé. 

```
@Override
protected void onRestoreInstanceState(Bundle etat) {
    super.onRestoreInstanceState(etat);
    // restaurer l'état précédent
    mScoreJoueur = etat.getInt(ETAT_SCORE);
}
```

Ces deux méthodes sont appelées automatiquement (sorte d'écouteurs), sauf si l'utilisateur tue l'application. Cela permet de reprendre l'activité là où elle en était.

## 3.4. Vues et activités

### 3.4.1. Obtention des vues

La méthode `setContentView` charge une disposition sur l'écran. Ensuite l'activité peut avoir besoin d'accéder aux vues, par exemple lire la chaîne saisie dans un texte. Pour cela, il faut obtenir l'objet

Java correspondant.



```
EditText nom = (EditText) findViewById(R.id.edt_nom);
```

Cette méthode cherche la vue qui possède cet identifiant dans le layout de l'activité. Si cette vue n'existe pas (mauvais identifiant, ou pas créée), la fonction retourne `null`.

Un mauvais identifiant peut être la raison d'un bug.

### 3.4.2. Propriétés des vues

La plupart des vues ont des *setters* et *getters* Java pour leurs propriétés XML. Par exemple `TextView`.

En XML :



```
<TextView android:id="@+id/titre"  
    android:lines="2"  
    android:text="@string/debut" />
```

En Java :



```
TextView tvTitre = (TextView) findViewById(R.id.titre);  
tvTitre.setLines(2);  
tvTitre.setText(R.string.debut);
```

Consulter leur documentation pour les propriétés, qui sont extrêmement nombreuses.

### 3.4.3. Actions de l'utilisateur

Prenons l'exemple de ce `Button`. Lorsque l'utilisateur appuie dessus, cela déclenche un *événement* « `onClick` », et appelle automatiquement la méthode `Valider` de l'activité.



```
<Button  
    android:id="@+id/btn_valider"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/valider"  
    android:onClick="Valider" />
```

Il faut définir la méthode `Valider` dans l'activité :




```
public void Valider(View btn) {  
    ...  
}
```

### 3.4.4. Définition d'un écouteur

Il y a une autre manière de définir une réponse à un clic : un écouteur (*listener*). C'est une instance de classe qui possède la méthode `public void onClick(View v)` ainsi que spécifié par l'interface `View.OnClickListener`.


Cela peut être :

- une classe privée anonyme,
- une classe privée ou public dans l'activité,
- l'activité elle-même.

Dans tous les cas, on fournit cette instance en paramètre à la méthode `setOnClickListener` du bouton : 

```
btn.setOnClickListener(ecouteur);
```


### 3.4.5. Écouteur privé anonyme

Il s'agit d'une classe qui est définie à la volée, lors de l'appel à `setOnClickListener`. Elle ne contient qu'une seule méthode. 

```
Button btn = (Button) findViewById(R.id.btn_valider);
btn.setOnClickListener(new View.OnClickListener() {
    public void onClick(View btn) {
        // faire quelque chose
    }
});
```

Employer la syntaxe `MonActivity.this` pour manipuler les variables et méthodes de l'activité sous-jacente.


### 3.4.6. Écouteur privé

Cela consiste à définir une classe privée dans l'activité ; cette classe implémente l'interface `OnClickListener` ; et à en fournir une instance en tant qu'écouteur. 

```
private class EcBtnValider implements OnClickListener {
    public void onClick(View btn) {
        // faire quelque chose
    }
};

public void onCreate(...) {
    ...
    Button btn=(Button)findViewById(R.id.btn_valider);
    btn.setOnClickListener(new EcBtnValider());
}
```


### 3.4.7. L'activité elle-même en tant qu'écouteur

Il suffit de mentionner `this` comme écouteur et d'indiquer qu'elle implémente l'interface `OnClickListener`. 

```
public class EditActivity extends Activity
    implements OnClickListener {
    public void onCreate(...) {
        ...
        Button btn=(Button)findViewById(R.id.btn_valider);
        btn.setOnClickListener(this);
    }
    public void onClick(View btn) {
        // faire quelque chose
    }
}
```

Ici, par contre, tous les boutons appelleront la même méthode.

### 3.4.8. Distinction des émetteurs

Dans le cas où le même écouteur est employé pour plusieurs vues, il faut les distinguer en se basant sur leur identifiant obtenu avec `getId()` : 

```
public void onClick(View v) {
    switch (v.getId()) {
        case R.id.btn_valider:
            ...
            break;
        case R.id.btn_effacer:
            ...
            break;
    }
}
```

### 3.4.9. Événements des vues courantes

Vous devrez étudier la documentation. Voici quelques exemples :

- Button : `onClick` lorsqu'on appuie sur le bouton, voir [sa doc](#)
- Spinner : `OnItemSelected` quand on choisit un élément, voir [sa doc](#)
- RatingBar : `OnRatingBarChange` quand on modifie la note, voir [sa doc](#)
- etc.

Heureusement, dans le cas de formulaires, les actions sont majoritairement basées sur des boutons.

### 3.4.10. C'est fini pour aujourd'hui

C'est assez pour cette semaine, rendez-vous la semaine prochaine pour un cours sur les applications de gestion de données (listes d'items).

Plus tard, nous verrons comment Android raffine la notion d'activité, en la séparant en *fragments*.

## Semaine 4

---

### Application liste

---

Durant les prochaines semaines, nous allons nous intéresser aux applications de gestion d'une liste d'items.

- Stockage d'une liste
- Affichage d'une liste, adaptateurs
- Consultation et édition d'un item

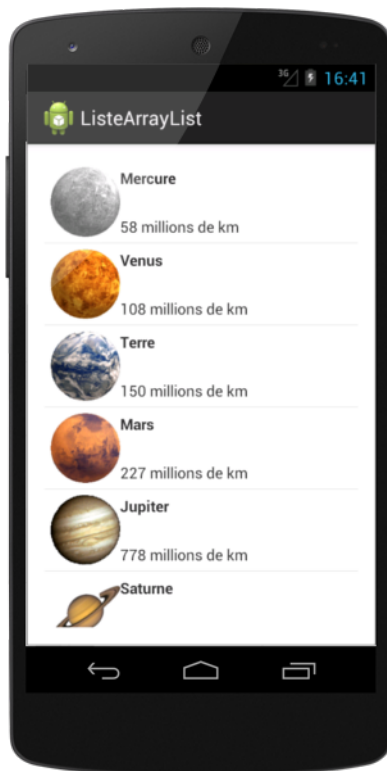


Figure 22: Liste d'items

## 4.1. Présentation

### 4.1.1. Principe général

On veut programmer une application pour afficher et éditer une liste d'items.

- Cette semaine, la liste est stockée dans un tableau dynamique appelé `ArrayList` ; en semaine 6, ça sera dans une base de données SQL locale ou sur un serveur distant.



- L'écran est occupé par un `ListView`. C'est une vue spécialisée dans l'affichage de listes quelconques.

Consulter [cette documentation](#) sur les `ListView`.

#### 4.1.2. Schéma global

L'intermédiaire entre la liste et la vue est géré par un *adaptateur*, objet qui sait comment afficher un item dans le `ListView`.

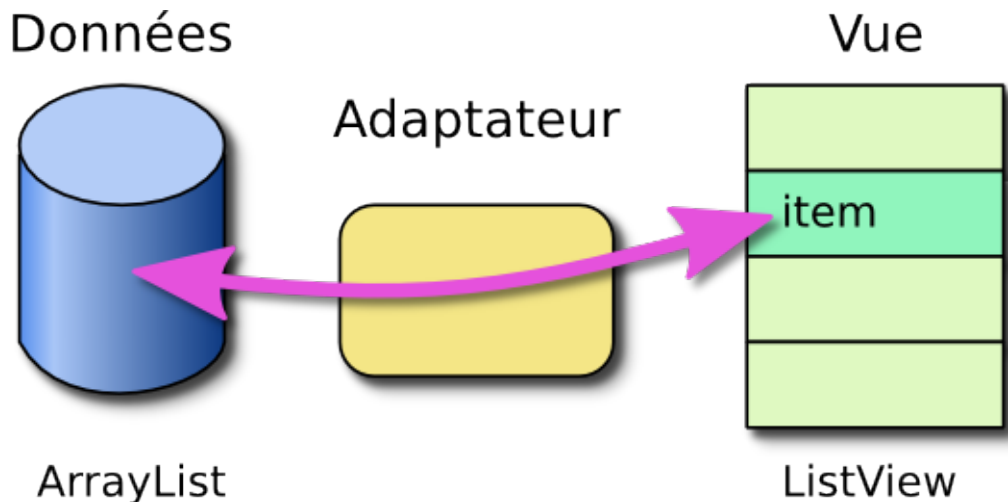


Figure 23: Vue, adaptateur et données

#### 4.1.3. Une classe pour représenter les items

Pour commencer, une classe pour représenter les items :



```
public class Planete {
    public String mNom;
    public int mDistance;

    Planete(String nom, int distance) {
        mNom = nom;           // nom de la planète
        mDistance = distance; // distance au soleil en Gm
    }

    public String toString() {
        return mNom;
    }
};
```

#### 4.1.4. Données initiales

Deux solutions pour initialiser la liste avec des items prédéfinis :

- Un tableau dans les ressources, voir page 67,

- Un tableau Java comme ceci :



```
final Planete[] initdata = {  
    new Planete("Mercure", 58),  
    new Planete("Vénus", 108),  
    new Planete("Terre", 150),  
    ...  
};
```

`final` signifie constant, sa valeur ne changera plus.

#### 4.1.5. Copie dans un ArrayList

L'étape suivante consiste à recopier les valeurs initiales dans un tableau dynamique de type `ArrayList<Planete>` :



```
protected ArrayList<Planete> mliste;  
  
void onCreate(...)  
{  
    ...  
    // création du tableau dynamique  
    mliste = new ArrayList<Planete>();  
    // boucle améliorée Java7  
    for (Planete planete: initdata) {  
        mliste.add(planete);  
    }  
}
```

#### 4.1.6. Le container Java ArrayList<type>

C'est un type de données générique, c'est à dire paramétré par le type des éléments mis entre `<...>` ; ce type doit être un objet.

```
import java.util.ArrayList;  
ArrayList<TYPE> liste = new ArrayList<TYPE>();
```

Quelques méthodes utiles :

- `liste.size()` : retourne le nombre d'éléments présents,
- `liste.clear()` : supprime tous les éléments,
- `liste.add(elem)` : ajoute cet élément à la liste,
- `liste.remove(elem ou indice)` : retire cet élément
- `liste.get(indice)` : retourne l'élément présent à cet indice,
- `liste.contains(elem)` : true si elle contient cet élément,
- `liste.indexOf(elem)` : indice de l'élément, s'il y est.

#### 4.1.7. Données initiales dans les ressources

On crée deux tableaux dans le fichier `res/values/arrays.xml` :



```
<resources>
    <string-array name="noms">
        <item>Mercure</item>
        <item>Venus</item>
        ...
    </string-array>
    <integer-array name="distances">
        <item>58</item>
        <item>108</item>
        ...
    </integer-array>
</resources>
```

#### 4.1.8. Données dans les ressources, suite

Ensuite, on récupère ces tableaux pour remplir le `ArrayList` :



```
// accès aux ressources
Resources res = getResources();
final String[] noms = res.getStringArray(R.array.noms);
final int[] distances = res.getIntArray(R.array.distances);

// recopie dans le ArrayList
mListe = new ArrayList<Planete>();
for (int i=0; i<noms.length; ++i) {
    mListe.add(new Planete(noms[i], distances[i]));
}
```

#### 4.1.9. Remarques

Cette semaine, les données sont représentées dans un tableau. Dans les exemples précédents, c'est une variable membre de l'activité. Pour faire mieux que cela, il faut définir une **Application** comme en semaine 3 et mettre ce tableau ainsi que son initialisation dedans. Ainsi, le tableau devient disponible dans toutes les activités de l'application. Voir le TP4.

En semaine 6, nous verrons comment utiliser une base de données SQL locale ou un **WebService**, ce qui résoud proprement le problème.

### 4.2. Affichage de la liste

#### 4.2.1. Activité spécialisée ou layout

Deux possibilités :

- employer la classe `ListActivity`,

- employer la classe `Activity` de base.

Ces deux possibilités sont très similaires : un *layout* contenant un `ListView` pour l'activité, un *layout* pour les items de la liste et un adaptateur pour accéder aux données.

La `ListActivity` prépare un peu plus de choses pour gérer les sélections d'items, tandis qu'avec une simple `Activity`, c'est à nous de tout faire, voir page 76. Par exemple, si on rajoute un `TextView` particulier, on peut avoir un message « La liste est vide ».


#### 4.2.2. Mise en œuvre

Que ce soit avec une `ListActivity` ou avec une `Activity` de base, deux choses sont à faire :

1. Créer un layout pour l'écran ; il doit contenir un `ListView` identifié par `@android:id/list`,
2. Créer un layout pour un item ; il doit contenir un `TextView` identifié par `@android:id/text1`,

Consulter [la documentation](#).

#### 4.2.3. Layout de l'activité pour afficher une liste

Voici d'abord le layout d'écran. J'ai rajouté le `TextView` qui affiche « Liste vide ». Notez les identifiants spéciaux. 

```
<LinearLayout xmlns:android="..."
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <ListView android:id="@android:id/list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
    <TextView android:id="@android:id/empty"
        android:text="Liste vide"
        ... />
</LinearLayout>
```

On peut rajouter d'autres vues : boutons...

#### 4.2.4. Mise en place du layout d'activité

Classiquement : 

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    // appeler la méthode surchargée dans la superclasse
    super.onCreate(savedInstanceState);

    // mettre en place le layout contenant le ListView
    setContentView(R.layout.main);

    // initialisation de la liste
```

```
mListe = new ArrayList<Planete>();  
...
```

#### 4.2.5. Layout pour un item

Ensuite, le layout `res/layout/item.xml` pour afficher un item. L'identifiant du `TextView` devient `android.R.id.text1` en Java. 

```
<LinearLayout xmlns:android="..."  
    android:orientation="vertical"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content">  
    <TextView android:id="@android:id/text1"  
        android:textStyle="bold"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"/>  
</LinearLayout>
```

#### 4.2.6. Autre layouts

Il est possible de créer des dispositions plus complexes pour les items mais alors il faudra programmer un adaptateur spécifique.



Figure 24: Layout complexe

```
<RelativeLayout xmlns:android="..." ...>  
    <ImageView android:id="@+id/item_planete_image" .../>  
    <TextView android:id="@+id/item_planete_nom" .../>  
    <TextView android:id="@+id/item_planete_distance" .../>  
</RelativeLayout>
```

Voir les adaptateurs personnalisés, page [72](#).

#### 4.2.7. Layouts prédéfinis

Android définit des layouts pour des éléments de listes simples :

- `android.R.layout.simple_list_item_1`  
C'est un layout qui affiche un seul `TextView`. Son identifiant est `android.R.id.text1`,
- `android.R.layout.simple_list_item_2`  
C'est un layout qui affiche deux `TextView` : un titre en grand et un sous-titre. Ses identifiants sont `android.R.id.text1` et `android.R.id.text2`.

Il suffit de les fournir à l'adaptateur. Il n'y a pas besoin de créer des fichiers XML, ni pour l'écran, ni pour les items.

## 4.3. Adaptateurs

### 4.3.1. Relations entre la vue et les données

Un `ListView` affiche les items à l'aide d'un *adaptateur* (*adapter*).

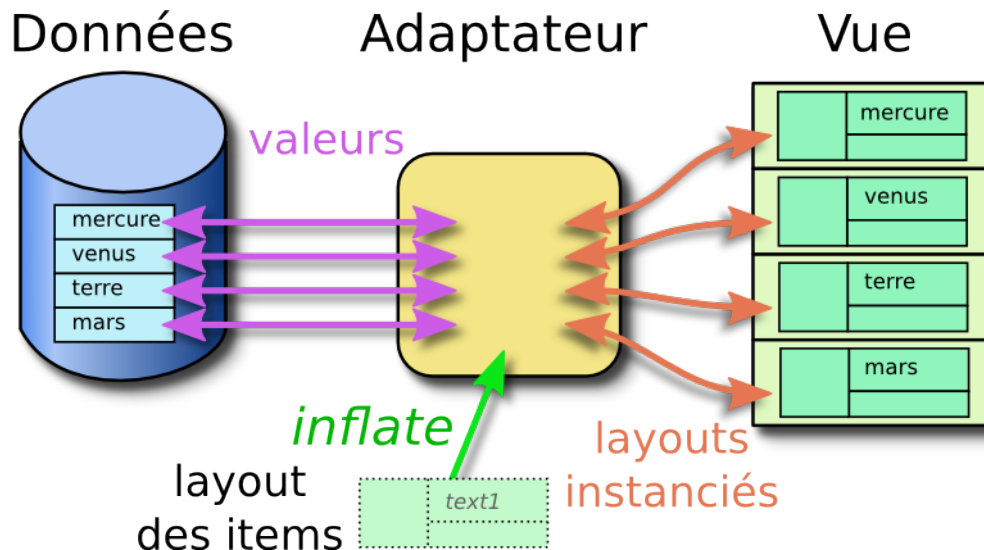


Figure 25: Adaptateur entre les données et la vue

### 4.3.2. Rôle d'un adaptateur

L'adaptateur répond à la question que pose le `ListView` : « *que dois-je afficher à tel endroit dans la liste ?* ». Il va chercher les données et instancie le layout d'item avec les valeurs.

C'est une classe qui :

- accède aux données à l'aide de méthodes telles que `getItem(int position)`, `getCount()`, `isEmpty()` quelque soit le type de stockage des éléments : tableau, BDD...
- crée les vues d'affichage des items : `getView(...)` à l'aide du layout des items. Cela consiste à instancier le layout — on dit *expanser* le layout, *inflate* en anglais.

### 4.3.3. Adaptateurs prédéfinis

Android propose quelques classes d'adaptateurs prédéfinis, dont :

- `ArrayAdapter` pour un tableau simple (liste dynamique),
- `SimpleCursorAdapter` pour accéder à une base de données, qu'on verra dans deux semaines.

En général, dans une application innovante, il faut définir son propre adaptateur, voir page 72, mais commençons par un `ArrayAdapter` standard.

### 4.3.4. `ArrayAdapter<Type>` pour les listes

Il permet d'afficher les données d'un `ArrayList`, mais il est limité à une seule chaîne par item, par exemple le nom d'une planète, fournie par sa méthode `toString()`. Son constructeur :

```
ArrayAdapter(Context context, int item_layout_id, int textview_id, List<T> données)
```

**context** c'est l'activité qui crée cet adaptateur, mettre **this**  
**item\_layout\_id** identifiant du layout des items, p. ex. `android.R.layout.simple_list_item_1`  
ou `R.layout.item_planete`  
**textview\_id** identifiant du `TextView` dans ce layout, p. ex. `android.R.id.text1` ou  
`R.id.item_planete_nom`  
**données** c'est la liste contenant les données (`List` est une surclasse de `ArrayList`)


#### 4.3.5. Exemple d'emploi

Suite de la méthode `onCreate` de l'activité, on fournit la `ArrayList<Planete> mListe` au constructeur d'adaptateur : 

```
// créer un adaptateur standard pour mListe
ArrayAdapter<Planete> adapter =
    new ArrayAdapter<Planete>(this,
        R.layout.item_planete,
        R.id.item_planete_nom,
        mListe);
// associer la liste affichée et l'adaptateur
ListView lv = (ListView) findViewById(android.R.id.list);
lv.setAdapter(adapter);
```

La classe `Planete` doit avoir une méthode `toString()`, cf page 65. Cet adaptateur n'affiche que le nom de la planète, rien d'autre.

#### 4.3.6. Affichage avec une `ListActivity`

Si l'activité est une `ListActivity`, la fin est peu plus simple : 

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);

    mListe = new ArrayList<Planete>();
    ...

    ArrayAdapter<Planete> adapter = new ArrayAdapter...

    // association liste - adaptateur
    setListAdapter(adapter);
}
```

#### 4.3.7. Exemple avec les layouts standards

Avec les layouts d'items standards Android, cela donne : 

```
// créer un adaptateur standard pour mListe
ArrayAdapter<Planete> adapter =
    new ArrayAdapter<Planete>(this,
        android.R.layout.simple_list_item_1,
        android.R.id.text1,
        mListe);

// associer la liste affichée et l'adaptateur
setListAdapter(adapter);
```

Le style d’affichage est minimaliste, seulement la liste des noms. On ne peut pas afficher deux informations avec un `ArrayAdapter`.

## 4.4. Adaptateur personnalisé

### 4.4.1. Classe Adapter personnalisée

Parce que `ArrayAdapter` n’affiche qu’un seul texte, nous allons définir notre propre adaptateur : `PlaneteAdapter`.

Il faut le faire hériter de `ArrayAdapter<Planete>` pour ne pas tout reprogrammer :



```
public class PlaneteAdapter extends ArrayAdapter<Planete>
{
    public PlaneteAdapter(Context context,
        List<Planete> planetes)
    {
        super(context, 0, planetes);
    }
}
```

Source biblio : <http://www.bignerdranch.com/blog/customizing-android-listview-rows-subclassing>

### 4.4.2. Classe Adapter perso, suite

Sa principale méthode est `getView` qui crée les vues pour le `ListView`. Elle retourne une disposition, p. ex. un `RelativeLayout` contenant des `TextView` et `ImageView`.



```
public
    View getView(int position, View recup, ViewGroup parent);
```

- `position` est le numéro, dans le `ListView`, de l’item à afficher.
- `recup` est une ancienne vue devenue invisible dans le `ListView`. C’est une technique pour diminuer les allocations mémoire, on récupère une vue inutile au lieu d’en allouer une nouvelle. NB: elle s’appelle `convertView` dans les docs.
- `parent` : le `ListView` auquel sera rattaché cette vue.



#### 4.4.3. Méthode `getView` personnalisée

Voici la surcharge de cette méthode :



```
@Override
public View
    getView(int position, View recap, ViewGroup parent)
{
    // créer ou récupérer un PlaneteView
    PlaneteView vueItem = (PlaneteView) recap;
    if (vueItem == null)
        vueItem = PlaneteView.create(parent); // <==(!!)

    // afficher les valeurs
    vueItem.display(super.getItem(position));
    return vueItem;
}
```

#### 4.4.4. Méthode `PlaneteView.create`

Cette méthode crée une instance de `PlaneteView`. C'est un groupe de vues qui affiche un seul item des données.

- Le `PlaneteAdapter` crée des `PlaneteView` à la demande du `ListView`,
- Un `PlaneteView` est une sorte de `RelativeLayout` contenant des `TextView` et `ImageView`



Figure 26:

- Cette disposition est définie par un fichier layout XML `res/layout/item_planete.xml`.

L'ensemble des données est affiché par plusieurs instances de `PlaneteView` dans le `ListView`.

#### 4.4.5. Layout d'item `res/layout/item_planete.xml`

C'est subtil : on va remplacer la racine du layout des items, un `RelativeLayout` par une classe personnalisée :



```
<?xml version="1.0" encoding="utf-8"?>
<fr.iutlan.planetes.PlaneteView
    xmlns:android="..."
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

Et cette classe `PlaneteView` hérite de `RelativeLayout` :



```
package fr.iutlan.planetes;  
public class PlaneteView extends RelativeLayout  
{  
    ...  
}
```

#### 4.4.6. Classe personnalisée dans les ressources


Android permet d'utiliser les classes de notre application à l'intérieur d'un layout. Il suffit de les préfixer par le package. 

```
<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="..."  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content">  
    <fr.iutlan.customviews.MaVuePerso  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"/>  
</LinearLayout>
```

La classe MaVuePerso doit hériter de View et implémenter certaines méthodes.

#### 4.4.7. Classe PlaneteView pour afficher les items


Cette classe a pour but de gérer les vues dans lesquelles il y a les informations des planètes : nom, distance, image.

On la met à la place du RelativeLayout dans res/layout/item\_planete.xml : 

```
<?xml version="1.0" encoding="utf-8"?>  
<fr.iutlan.planetes.PlaneteView .../>  
    <ImageView android:id="@+id/item_planete_image" .../>  
    <TextView android:id="@+id/item_planete_nom" .../>  
    <TextView android:id="@+id/item_planete_distance" .../>  
</fr.iutlan.planetes.PlaneteView>
```

Les propriétés de placement restent les mêmes.

#### 4.4.8. Définition de la classe PlaneteView

Le constructeur de PlaneteView est nécessaire, mais quasi-vide : 


```
public class PlaneteView extends RelativeLayout  
{  
    public PlaneteView(Context context, ...) {  
        super(context, attrs);  
    }  
}
```

Tout se passe dans la méthode de classe PlaneteView.create appelée par l'adaptateur. Rappel de la page 73 :

```
// créer ou récupérer un PlaneteView
PlaneteView vueItem = (PlaneteView) recup;
if (vueItem == null) vueItem = PlaneteView.create(parent);
...
```

Cette méthode `create` génère les vues du layout `item.xml`.

#### 4.4.9. Créer des vues à partir d'un layout XML


La génération de vues pour afficher les items repose sur un mécanisme appelé `LayoutInflater` qui fabrique des vues Android à partir d'un layout XML : 

```
LayoutInflater li = LayoutInflater.from(context);
View vueItem = li.inflate(R.layout.item_planete, parent);
```

On lui fournit l'identifiant du layout, p. ex. celui des items. Elle crée les vues spécifiées dans `res/layout/item_planete.xml`.

- `context` est l'activité qui affiche toutes ces vues,
- `parent` est la vue qui doit contenir ces vues, `null` si aucune.


#### 4.4.10. Méthode `PlaneteView.create`

La méthode de classe `PlaneteView.create` expande le layout des items à l'aide d'un `LayoutInflater` : 

```
public static PlaneteView create(ViewGroup parent)
{
    LayoutInflater li =
        LayoutInflater.from(parent.getContext());
    PlaneteView itemView = (PlaneteView)
        li.inflate(R.layout.item_planete, parent, false);
    itemView.findViews();
    return itemView;
}
```

`static` signifie qu'on appelle cette méthode sur la classe elle-même et non pas sur une instance. C'est une *méthode de classe*.

#### 4.4.11. Méthode `findViews`

Cette méthode a pour but de récupérer les objets Java des `TextView` et `ImageView` de l'item. Elle les recherche avec leurs propriétés `android:id`. 

```
private void findViews()
{
    tvNom = (TextView) findViewById(R.id.item_planete_nom);
    tvDistance = (TextView)
```

```
        findViewById(R.id.item_planete_distance);  
        ivImage = (ImageView)  
            findViewById(R.id.item_planete_image);  
    }
```

Ces trois variables sont des membres d'instance du `PlaneteView`.

#### 4.4.12. Pour finir, la méthode `PlaneteView.display`

Son rôle est d'afficher les informations d'une planète dans les `TextView` et `ImageView` de l'item. 📌

```
public void display(final Planete planete)  
{  
    tvNom.setText(planete.getNom());  
    tvDistance.setText(  
        Integer.toString(planete.getDistance())  
        + " millions de km");  
    ivImage.setImageResource(planete.getIdImage());  
}
```

Elle utilise les *getters* de la classe `Planete` : `getNom...`

#### 4.4.13. Récapitulatif

Voici la séquence qui amène à l'affichage d'un item dans la liste :

1. Le `ListView` appelle la méthode `getView(position, ...)` de l'adaptateur, `position` est le n° de l'élément concerné,
2. L'adaptateur appelle éventuellement `PlaneteView.create` :
  - a. `PlaneteView.create` fait instancier `item.xml` = une sous-classe de `RelativeLayout` appelée `PlaneteView`.
  - b. Cela crée les vues `nom`, `distance` et `image` dont `PlaneteView.findViews` récupère les objets Java.
3. L'adaptateur appelle la méthode `display` du `PlaneteView` avec les données à afficher.
  - a. `PlaneteView.display` appelle `setText` des vues pour afficher les valeurs.

#### 4.4.14. Le résultat

Voir la figure 27, page 77.

### 4.5. Actions utilisateur sur la liste

#### 4.5.1. Modification des données

Les modifications sur les données doivent se faire par les méthodes `add`, `insert`, `remove` et `clear` de l'adaptateur. Voir [la doc](#).

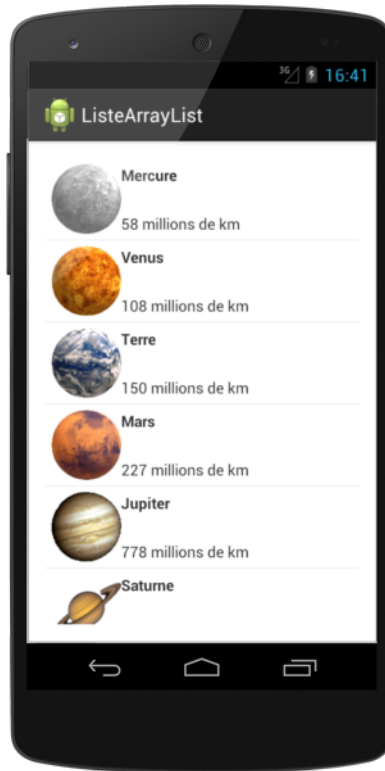


Figure 27: Liste d'items

Si ce n'est pas possible, par exemple parce qu'on a changé d'activité et modifié les données sans adaptateur, alors au retour, par exemple dans `onActivityResult`, il faut prévenir l'adaptateur par la méthode suivante :

```
adapter.notifyDataSetChanged();
```

#### 4.5.2. Clic sur un élément

Voyons le traitement des sélections utilisateur sur une liste. La classe `ListActivity` définit déjà un écouteur pour les clics. Il suffit de le surcharger :

```
@Override
public void onItemClick (
    ListView l, View v, int position, long id)
{
    // gérer un clic sur l'item identifié par id
    ...
}
```

Par exemple, créer un `Intent` pour afficher ou éditer l'item. Ne pas oublier d'appeler `adapter.notifyDataSetChanged();` au retour.

#### 4.5.3. Clic sur un élément, suite

Si votre activité est une simple `Activity` (parce qu'il y a autre chose qu'une liste, ou plusieurs listes), alors c'est plus complexe :

- Votre activité doit implémenter l'interface `AdapterView.OnItemClickListener`,
- Vous devez définir `this` en tant qu'écouteur du `ListView`,
- Votre activité doit surcharger la méthode `onItemClick`.

#### 4.5.4. Clic sur un élément, suite



```
public class MainActivity extends Activity
    implements OnItemClickListener
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        // appeler la méthode surchargée dans la superclasse
        super.onCreate(savedInstanceState);

        // mettre en place le layout contenant le ListView
        setContentView(R.layout.main);
        ListView lv=(ListView)findViewById(android.R.id.list);
        lv.setOnItemClickListener(this);
    }
}
```

#### 4.5.5. Clic sur un élément, fin

Et voici sa méthode `onItemClick` :



```
@Override
public void onItemClick(
    AdapterView<?> parent, View v, int position, long id)
{
    // gérer un clic sur l'item identifié par id
    ...
}
```

Il existe aussi la méthode boolean `onItemLongClick` ayant les mêmes paramètres, installée par `setOnItemLongClickListener`.

#### 4.5.6. Liste d'éléments cochables

Android offre des listes cochables comme celles-ci :

Voir la figure 28, page 79.

Le style de la case à cocher dépend du choix unique ou multiple.






Mercure		Mercure	<input type="checkbox"/>
Vénus		Vénus	<input type="checkbox"/>
Terre		Terre	<input checked="" type="checkbox"/>
Mars		Mars	<input checked="" type="checkbox"/>
Jupiter		Jupiter	<input type="checkbox"/>

Figure 28: Éléments cochables

#### 4.5.7. Liste cochable simple

Android propose un layout prédéfini pour items cochables :



```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    ...
    setListAdapter(
        new ArrayAdapter<Planete>(this
            android.R.layout.simple_list_item_checked,
            android.R.id.text1, mListe));

    ListView lv=(ListView)findViewById(android.R.id.list);
    lv.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
}
```

#### 4.5.8. Liste à choix multiples

Toujours avec des listes prédéfinies, c'est une simple variante :

- mettre `simple_list_item_multiple_choice` à la place de `simple_list_item_checked`,
- mettre `ListView.CHOICE_MODE_MULTIPLE` au lieu de `ListView.CHOICE_MODE_SINGLE`.

La méthode `onListItemClick` est appelée sur chaque élément cliqué.

#### 4.5.9. Liste cochable personnalisée

Si on veut un layout personnalisé comme `PlaneteView`, il faut que sa classe implémente l'interface `Checkable` càd 3 méthodes :

- `public boolean isChecked()` indique si l'item est coché
- `public void setChecked(boolean etat)` doit changer *l'état interne* de l'item
- `public void toggle()` doit inverser *l'état interne* de l'item

Il faut rajouter un booléen dans chaque item, celui que j'ai appelé *état interne*.

D'autre part, dans le layout d'item, il faut employer un `CheckedTextView` même vide, plutôt qu'un `CheckBox` qui ne réagit pas aux clics (bug Android).

#### **4.5.10. Ouf, c'est fini**

C'est tout pour cette semaine. La semaine prochaine nous parlerons des menus, dialogues et fragments.



## Semaine 5

---

### Ergonomie

---

Le cours de cette semaine concerne l'ergonomie d'une application Android.

- Menus et barre d'action
- Popup-up : messages et dialogues
- Activités et fragments
- Préférences (pour info)
- Bibliothèque support (pour info)

### 5.1. Barre d'action et menus

#### 5.1.1. Barre d'action

La barre d'action contient l'icône d'application (1), quelques items de menu (2) et un bouton pour avoir les autres menus (3).



Figure 29: Barre d'action

#### 5.1.2. Réalisation d'un menu

Avant Android 3.0 (API 11), les actions d'une application étaient lancées avec un bouton de menu, mécanique. Depuis, elles sont déclenchées par la barre d'action. C'est presque la même chose.

Le principe général : un menu est une liste d'items qui apparaît soit quand on appuie sur le bouton menu, soit sur la barre d'action. Certains de ces items sont présents en permanence dans la barre d'action. La sélection d'un item déclenche une *callback*.

Docs Android sur la [barre d'action](#) et sur [les menus](#)

Il faut définir :

- un fichier `res/menu/nom_du_menu.xml`,
- des thèmes pour afficher soit la barre d'action, soit des menus,
- deux *callbacks* pour gérer les menus : création et activation.

### 5.1.3. Spécification d'un menu

Créer `res/menu/nom_du_menu.xml` :



```
<menu xmlns:android="..." >
    <item android:id="@+id/menu_creer"
        android:icon="@drawable/ic_menu_creer"
        android:showAsAction="ifRoom"
        android:title="@string/menu_creer"/>
    <item android:id="@+id/menu_chercher" ... />
    ...
</menu>
```

L'attribut `showAsAction` vaut "always", "ifRoom" ou "never" selon la visibilité qu'on souhaite dans la barre d'action. Cet attribut est à modifier en `app:showAsAction` si on utilise la bibliothèque support (appcompat).

### 5.1.4. Icônes pour les menus

Android distribue gratuitement un grand jeu d'icônes pour les menus, dans les deux styles : HoloDark et HoloLight.



Figure 30: Icônes de menus

Consulter la page [Downloads](#) pour des téléchargements gratuits de toutes sortes de modèles et feuilles de styles.

Téléchargez [Action Bar Icon Pack](#)

pour améliorer vos applications.

### 5.1.5. Thème pour une barre d'action

Les thèmes permettent d'afficher soit une barre d'action, soit un menu ancien style. Ils sont définis dans trois dossiers :

- `res/values/styles.xml`
- `res/values-v11/styles.xml`
- `res/values-v14/styles.xml`

En résumé, il faut indiquer que votre application gère les barres d'action. Voici par exemple pour la V11 :



```
<resources>
    <style name="AppBaseTheme"
        parent="android:Theme.Holo.Light" />
</resources>
```

Utiliser l'assistant pour avoir les thèmes adéquats.

### 5.1.6. Écouteurs pour les menus

Il faut programmer deux méthodes. L'une affiche le menu, l'autre réagit quand l'utilisateur sélectionne un item. Voici la première :

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

Cette méthode rajoute les items du menu défini dans le XML.

Un `MenuInflater` est un lecteur/traducteur de fichier XML en vues ; sa méthode `inflate` crée les vues.

### 5.1.7. Réactions aux sélections d'items

Voici la seconde *callback*, c'est un aiguillage selon l'item choisi :

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menu_creer:
            ...
            return true;
        case R.id.menu_chercher:
            ...
            return true;
        ...
        default: return super.onOptionsItemSelected(item);
    }
}
```

### 5.1.8. Menus en cascade

Définir deux niveaux quand la barre d'action est trop petite :

```
<menu xmlns:android="..." >
    <item android:id="@+id/menu_item1" ... />
    <item android:id="@+id/menu_item2" ... />
    <item android:id="@+id/menu_more"
        android:icon="@drawable/ic_action_overflow"
        android:showAsAction="always"
        android:title="@string/menu_more">
        <menu>
            <item android:id="@+id/menu_item3" ... />
            <item android:id="@+id/menu_item4" ... />
        </menu>
    </item>
</menu>
```

### 5.1.9. Menus contextuels

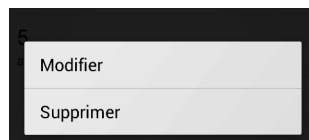


Figure 31: MenuContextuel

Ces menus apparaissent lors un clic long sur un élément de liste. Le principe est le même que pour les menus normaux :

- Attribuer un écouteur à l'événement `onCreateContextMenu`. Cet événement correspond à un clic long et au lieu d'appeler la callback du clic long, ça fait apparaître le menu.
- Définir la callback de l'écouteur : elle expose un layout de menu.
- Définir la callback des items du menu.

### 5.1.10. Associer un menu contextuel à une vue

Cela se passe par exemple dans la méthode `onCreate` d'une activité :

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    ListView lv = (ListView)findViewById(android.R.id.list);
    registerForContextMenu(lv);
    ...
}
```

Au lieu de `registerForContextMenu(lv)`; on peut aussi faire :

```
lv.setOnCreateContextMenuListener(this);
```

### 5.1.11. Callback d’affichage du menu

Quand l’utilisateur fait un clic long, cela déclenche cette méthode :

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
    ContextMenuInfo menuInfo)
{
    super.onCreateContextMenu(menu, v, menuInfo);
    getMenuInflater().inflate(R.menu.main_context, menu);
}
```

Son rôle est d’expanser (*inflate*) le menu `res/menu/main_context`.

### 5.1.12. Callback des items du menu

Pour finir, si l’utilisateur choisit un item du menu :

```
public boolean onContextItemSelected(MenuItem item) {
    AdapterContextMenuInfo info = (ACMI...) item.getMenuInfo();
    switch (item.getItemId()) {
        case R.id.editer:
            onMenuEditer(info.id); // identifiant de l'élément
            return true;
        case R.id.supprimer:
            onMenuSupprimer(info.id);
            return true;
    }
    return false;
}
```

L’objet `AdapterContextMenuInfo info` permet d’avoir l’identifiant de ce qui est sélectionné, qui a fait apparaître le menu contextuel.

## 5.2. Annonces et dialogues

### 5.2.1. Annonces : *toasts*

Un « *toast* » est un message apparaissant en bas d’écran pendant un instant, par exemple pour confirmer la réalisation d’une action. Un *toast* n’affiche aucun bouton et n’est pas actif.

Voici comment l’afficher avec une ressource chaîne :

```
Toast.makeText(getApplicationContext(),
    R.string.item_supprime, Toast.LENGTH_SHORT).show();
```

La durée d’affichage peut être allongée avec `LENGTH_LONG`.

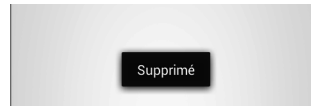


Figure 32: Toast

### 5.2.2. Annonces personnalisées

Il est possible de personnaliser une annonce. Il faut seulement définir un layout dans `res/layout/toast_perso.xml`. La racine de ce layout doit avoir un identifiant, ex : `toast_perso_id` qui est mentionné dans la création :

```
// expanser le layout du toast
LayoutInflater inflater = getLayoutInflater();
View layout = inflater.inflate(R.layout.toast_perso,
    (ViewGroup) findViewById(R.id.toast_perso_id));
// créer le toast et l'afficher
Toast toast = new Toast(getApplicationContext());
toast.setDuration(Toast.LENGTH_LONG);
toast.setView(layout);
toast.show();
```

### 5.2.3. Dialogues

Un dialogue est une petite fenêtre qui apparaît au dessus d'un écran pour afficher ou demander quelque chose d'urgent à l'utilisateur, par exemple une confirmation.

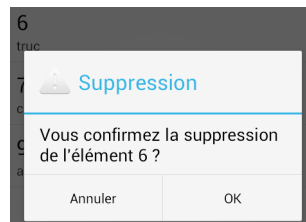


Figure 33: Dialogue d'alerte

Il existe plusieurs sortes de dialogues :

- Dialogues d'alerte
- Dialogues généraux

### 5.2.4. Dialogue d'alerte

Un dialogue d'alerte `AlertDialog` affiche un texte et un à trois boutons au choix : ok, annuler, oui, non, aide. . .

Un dialogue d'alerte est construit à l'aide d'une classe nommée `AlertDialog.Builder`. Le principe est de créer un *builder* et c'est lui qui crée le dialogue. Voici le début :

```
Builder confirm = new AlertDialog.Builder(this);
confirm.setTitle("Suppression");
confirm.setIcon(android.R.drawable.ic_dialog_alert);
confirm.setMessage("Vous confirmez la suppression ?");
```

Ensuite, on rajoute les boutons et leurs écouteurs.

### 5.2.5. Boutons et affichage d'un dialogue d'alerte

Le *builder* permet de rajouter toutes sortes de boutons : oui/non, ok/annuler... Cela se fait avec des fonctions comme celle-ci. On peut associer un écouteur (anonyme privé ou ...) ou aucun. 

```
// rajouter un bouton "oui" qui supprime vraiment
confirm.setPositiveButton(android.R.string.yes,
    new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int idBtn) {
            SupprimerElement(idElement);
        }
    });
// rajouter un bouton "non" qui ne fait rien
confirm.setNegativeButton(android.R.string.no, null);
// affichage du dialogue
confirm.show();
```

### 5.2.6. Autres types de dialogues d'alerte

Dans un dialogue d'alerte, au lieu de boutons, il est possible d'afficher une [liste de propositions prédéfinies](#). Pour cela :

- Définir une ressource de type tableau de chaînes `res/values/arrays.xml` : 

```
<resources>
    <string-array name="notes">
        <item>Nul</item>
        <item>Ça le fait</item>
        <item>Trop cool</item>
    </string-array>
</resources>
```

- Appeler la méthode `confirm.setItems(R.array.notes, écouteur)`. L'écouteur est le même que pour un clic. Il reçoit le numéro du choix en 2<sup>e</sup> paramètre `idBtn`.

Dans ce cas, ne pas appeler `confirm.setMessage` car ils sont exclusifs. C'est soit une liste, soit un message.

### 5.2.7. Dialogues personnalisés

Lorsqu'il faut demander une information plus complexe à l'utilisateur, mais sans que ça nécessite une activité à part entière, il faut faire appel à un [dialogue personnalisé](#).

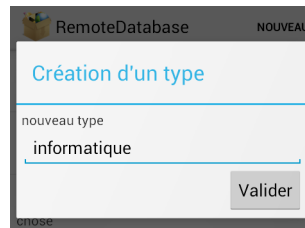


Figure 34: Dialogue perso

### 5.2.8. Création d'un dialogue

Il faut définir le layout du dialogue incluant tous les textes, sauf le titre, et au moins un bouton pour valider, sachant qu'on peut fermer le dialogue avec le bouton `back`.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="..." ...>
    <TextView android:id="@+id/dialog_titre" .../>
    <EditText android:id="@+id/dialog_libelle" .../>
    <Button android:id="@+id/dialog_btn_valider" ... />
</LinearLayout>
```

Ensuite cela ressemble à ce qu'on fait dans `onCreate` d'une activité : `setContentView` avec le layout et des `setOnClickListener` pour attribuer une action aux boutons.

### 5.2.9. Affichage du dialogue



```
// créer le dialogue
final Dialog dialog = new Dialog(this);
dialog.setContentView(R.layout.edit_dialog);
dialog.setTitle("Création d'un type");
// bouton valider
Button btnValider =
    (Button) dialog.findViewById(R.id.dialog_btn_valider);
btnValider.setOnClickListener(new OnClickListener() {
    @Override public void onClick(View v) {
        ... // récupérer et traiter les infos
        dialog.dismiss(); // fermer le dialogue
    }
});
// afficher le dialogue
dialog.show();
```



## 5.3. Fragments et activités

### 5.3.1. Fragments

Depuis Android 4, les dialogues doivent être gérés par des instances de `DialogFragment` qui sont des sortes de *fragments*, voir [cette page](#). Cela va plus loin que les dialogues. Toutes les parties des interfaces d'une application sont susceptibles de devenir des *fragments* :

- liste d'items
- affichage des infos d'un item
- édition d'un item

Un *fragment* est une sorte de mini-activité. Dans le cas d'un dialogue, elle gère l'affichage et la vie du dialogue. Dans le cas d'une liste, elle gère l'affichage et les sélections des éléments.

### 5.3.2. Tablettes, smartphones...

Une interface devient plus souple avec des fragments. Selon la taille d'écran, on peut afficher une liste et les détails, ou séparer les deux.

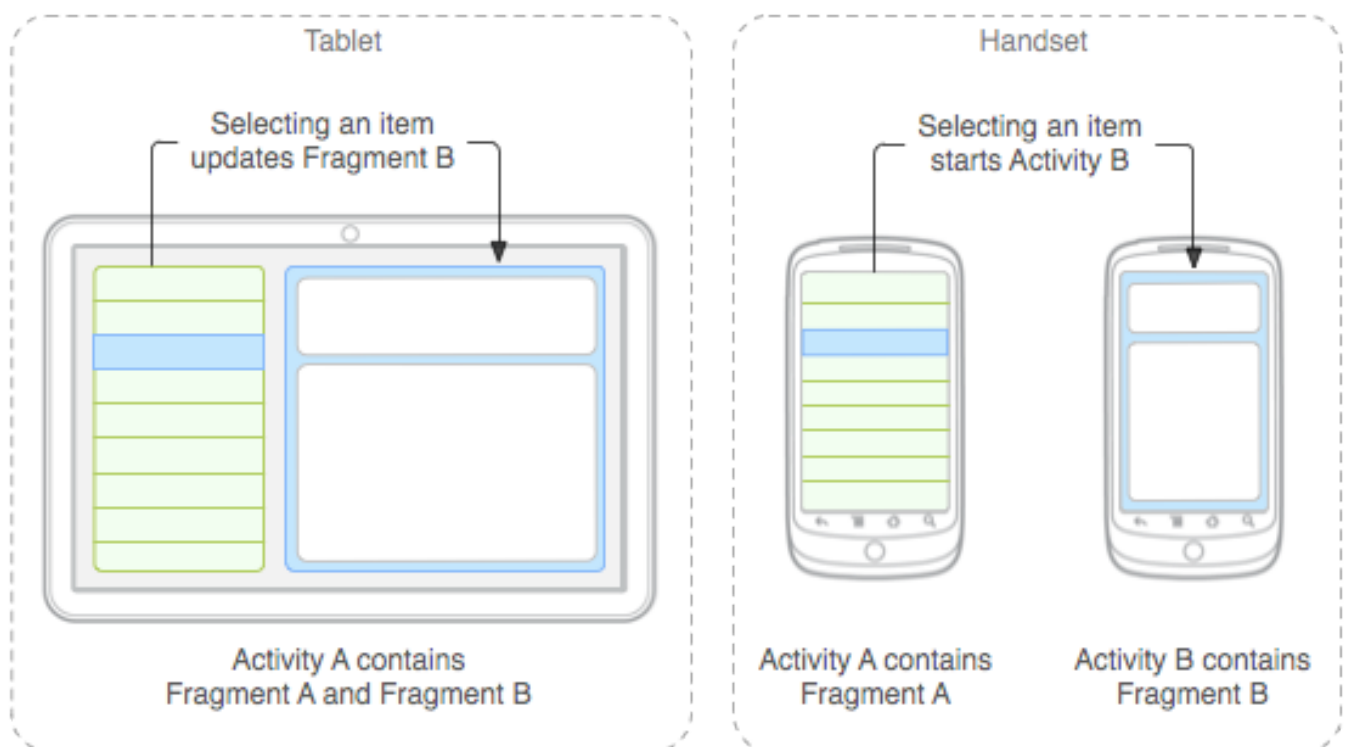



Figure 35: Différentes apparences

### 5.3.3. Structure d'un fragment

Un fragment est une activité très simplifiée. C'est seulement un arbre de vue défini par un layout, et des écouteurs. Un fragment minimal est : 

```
public class InfosFragment extends Fragment
{
    public InfosFragment() {}

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

### 5.3.4. Différents types de fragments

Il existe différents types de fragments, voici quelques uns :

- `ListFragment` pour afficher une liste d'items, comme le ferait une `ListActivity`.
- `DialogFragment` pour afficher un fragment dans une fenêtre flottante au dessus d'une activité.
- `PreferenceFragment` pour gérer les préférences.

En commun : il faut surcharger la méthode `onCreateView` qui définit leur contenu.

### 5.3.5. Cycle de vie des fragments

Les fragments ont un cycle de vie similaire à celui des activités, avec quelques méthodes de plus correspondant à leur intégration dans une activité.

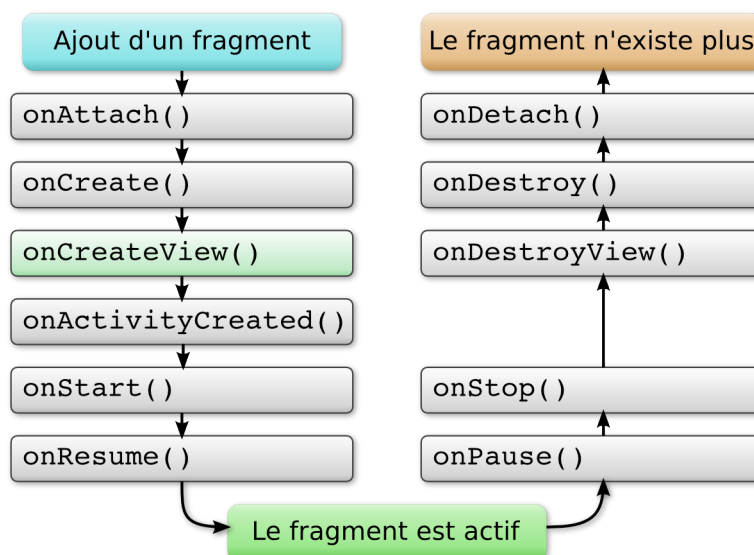


Figure 36: Cycle de vie d'un fragment

### 5.3.6. ListFragment


Par exemple, voici l'attribution d'un layout standard pour la liste :



```
@Override
public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle savedInstanceState)
{
    // liste des éléments venant de l'application
    FragmentApplication app = (FragmentApplication)
        getActivity().getApplicationContext();
    listeItems = app.getListe();


    // layout du fragment
    setHasOptionsMenu(true);    // le fragment a un menu
    return inflater.inflate(android.R.layout.list_content,
        container, false);
}
```

### 5.3.7. ListFragment, suite

Voici la suite, le remplissage de la liste et l'attribution d'un écouteur pour les clics sur les éléments : 

```
@Override
public void onActivityCreated(Bundle savedInstanceState)
{
    super.onActivityCreated(savedInstanceState);
    // adaptateur standard pour la liste
    ArrayAdapter<Item> adapter = new ArrayAdapter<Item>(
        getActivity(), android.R.layout.simple_list_item_1,
        listeItems);
    setListAdapter(adapter);
    // attribuer un écouteur pour les clics sur les items
    ListView lv = getListView();
    lv.setOnItemClickListener(this);
}
```

### 5.3.8. Menus de fragments

Un fragment peut définir un menu. Ses éléments sont intégrés à la barre d'action de l'activité. Seule la méthode de création du menu diffère, l'*inflater* arrive en paramètre : 

```
@Override
public void onCreateOptionsMenu(
    Menu menu, MenuInflater menuInflater)
{
    super.onCreateOptionsMenu(menu, menuInflater);
    menuInflater.inflate(R.menu.edit_fragment, menu);
}
```

NB: dans la méthode `onCreateView` du fragment, il faut rajouter `setHasOptionsMenu(true);`

### 5.3.9. Intégrer un fragment dans une activité

De lui-même, un fragment n'est pas capable de s'afficher. Il ne peut apparaître que dans le cadre d'une activité, comme une sorte de vue interne. On peut le faire de deux manières :

- statiquement : les fragments à afficher sont prévus dans le layout de l'activité. C'est le plus simple à faire et à comprendre.
- dynamiquement : les fragments sont ajoutés, enlevés ou remplacés en cours de route selon les besoins.

### 5.3.10. Fragments statiques dans une activité


Dans ce cas, c'est le layout de l'activité qui inclut les fragments, p. ex. `res/layout-land/main_activity.xml`. Ils ne peuvent pas être modifiés ultérieurement. 

```
<LinearLayout ... android:orientation="horizontal" ... >
  <fragment
    android:id="@+id/frag_liste"
    android:name="fr.iutlan.fragments.ListeFragment"
    ... />
  <fragment
    android:id="@+id/frag_infos"
    android:name="fr.iutlan.fragments.InfosFragment"
    ... />
</LinearLayout>
```

Chaque fragment doit avoir un identifiant et un nom complet.

### 5.3.11. FragmentManager


Pour définir des fragments dynamiquement, on fait appel au **FragmentManager** de l'activité. Il gère l'affichage des fragments. L'ajout et la suppression de fragments se fait à l'aide de *transactions*. C'est simplement l'association entre un « réceptacle » (un **FrameLayout** vide) et un fragment.

Soit un layout contenant deux **FrameLayout** vides : 

```
<LinearLayout xmlns:android="..."
  android:orientation="horizontal" ... >
  <FrameLayout android:id="@+id/frag_liste" ... />
  <FrameLayout android:id="@+id/frag_infos" ... />
</LinearLayout>
```

On peut dynamiquement attribuer un fragment à chacun.

### 5.3.12. Attribution d'un fragment dynamiquement

En trois temps : obtention du manager, création d'une transaction et attribution des fragments aux « réceptacles ». 

```
// gestionnaire
FragmentManager manager = getFragmentManager();

// transaction
FragmentManager trans = manager.beginTransaction();

// mettre les fragments dans les réceptacles
trans.add(R.id.frag_liste, new ListeFragment());
trans.add(R.id.frag_infos, new InfosFragment());
trans.commit();
```

Les `FrameLayout` sont remplacés par les fragments.

### 5.3.13. Disposition selon la géométrie de l'écran

Le plus intéressant est de faire apparaître les fragments en fonction de la taille et l'orientation de l'écran (application « liste + infos »).

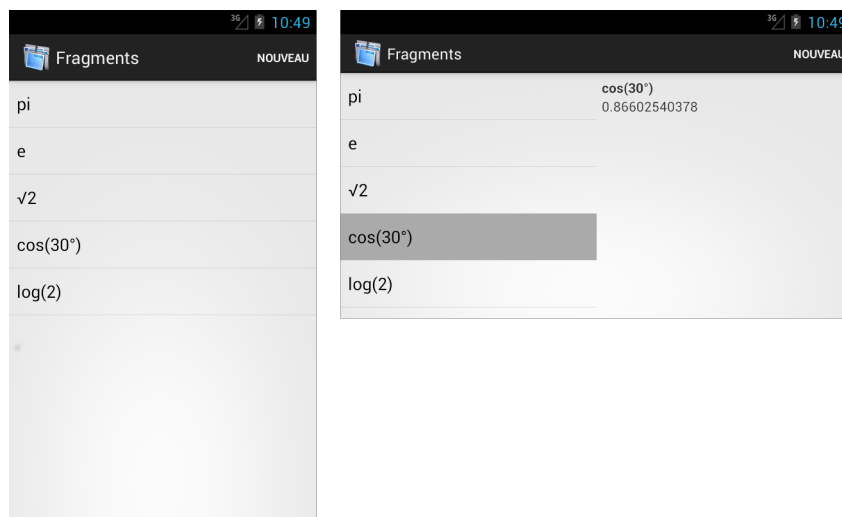


Figure 37: Un ou deux fragments affichés

### 5.3.14. Changer la disposition selon la géométrie

Pour cela, il suffit de définir deux layouts (définition statique) :

- `res/layout-port/main_activity.xml` en portrait :

```
<LinearLayout xmlns:android="..."
    android:orientation="horizontal" ... >
    <fragment android:id="@+id/frag_liste" ... />
</LinearLayout>
```

- `res/layout-land/main_activity.xml` en paysage :

```
<LinearLayout xmlns:android="..."
    android:orientation="horizontal" ... >
    <fragment android:id="@+id/frag_liste" ... />
    <fragment android:id="@+id/frag_infos" ... />
</LinearLayout>
```

### 5.3.15. Deux dispositions possibles

Lorsque la tablette est verticale, le layout de `layout-port` est affiché et lorsqu'elle est horizontale, c'est celui de `layout-land`.

L'activité peut alors faire un test pour savoir si le fragment `frag_infos` est affiché :



```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_activity);
    FragmentManager manager = getFragmentManager();
    InfosFragment frag_infos = (InfosFragment)
        manager.findFragmentById(R.id.frag_infos);
    if (frag_infos != null) {
        // le fragment des informations est présent
        ...
    }
}
```

### 5.3.16. Communication entre Activité et Fragments

Lorsque l'utilisateur clique sur un élément de la liste du fragment `frag_liste`, cela doit afficher ses informations :

- dans le fragment `frag_infos` s'il est présent,
- ou lancer une activité d'affichage séparée si le fragment n'est pas présent (layout vertical).

Cela implique plusieurs petites choses :

- L'écouteur des clics sur la liste est le fragment `frag_liste`. Il doit transmettre l'item cliqué à l'activité.
- L'activité doit déterminer si le fragment `frag_infos` est affiché :
  - s'il est visible, elle lui transmet l'item cliqué
  - sinon, elle lance une activité spécifique, `InfosActivity`.

Voici les étapes.

### 5.3.17. Interface pour un écouteur

D'abord la classe `ListeFragment` : définir une interface pour gérer les sélections d'items et un écouteur :



```
public interface OnItemSelectedListener {
    public void onItemSelected(Item item);
}

private OnItemSelectedListener listener;
```

Ce sera l'activité principale qui sera cet écouteur, grâce à :



```
@Override public void onAttach(Activity activity)
{
    super.onAttach(activity);
    listener = (OnItemSelectedListener) activity;
}
```

### 5.3.18. Écouteur du fragment

Toujours dans la classe ListeFragment, voici la *callback* pour les sélections dans la liste :



```
@Override
public void onItemClick(AdapterView<?> parent, View view,
    int position, long id)
{
    Item item = listeItems.get((int)id);
    listener.onItemSelected(item);
}
```

Elle va chercher l'item sélectionné et le fournit à l'écouteur, c'est à dire à l'activité principale.

### 5.3.19. Écouteur de l'activité

Voici maintenant l'écouteur de l'activité principale :



```
@Override public void onItemSelected(Item item)
{
    FragmentManager manager = getFragmentManager();
    InfosFragment frag_infos = (InfosFragment)
        manager.findFragmentById(R.id.frag_infos);
    if (frgInfos != null && frgInfos.isVisible()) {
        // le fragment est présent, alors lui fournir l'item
        frgInfos.setItem(item);
    } else {
        // lancer InfosActivity pour afficher l'item
        Intent intent = new Intent(this, InfosActivity.class);
        intent.putExtra("item", item);
        startActivity(intent);
    }
}
```

### 5.3.20. Relation entre deux classes à méditer, partie 1

Une classe « active » capable d'avertir un écouteur d'un événement. Elle déclare une interface que doit implémenter l'écouteur.

```
public class Classe1 {
    public interface OnEvenementListener {
        public void onEvenement(int param);
    }
    private OnEvenementListener ecouteur = null;
    public void setOnEvenementListener(
        OnEvenementListener objet) {
        ecouteur = objet;
    }
    private void traitementInterne() {
        ...
        if (ecouteur!=null) ecouteur.onEvenement(argument);
    }
}
```

### 5.3.21. À méditer, partie 2

Une 2<sup>e</sup> classe en tant qu'écouteur des événements d'un objet de Classe1, elle implémente l'interface et se déclare auprès de l'objet.

```
public class Classe2 implements Classe1.OnEvenementListener
{
    private Classe1 objet1;

    public Classe2() {
        ...
        objet1.setOnEvenementListener(this);
    }

    public void onEvenement(int param) {
        ...
    }
}
```

## 5.4. Préférences d'application

### 5.4.1. Illustration

Les préférences mémorisent des choix de l'utilisateur entre deux exécutions de l'application.

### 5.4.2. Présentation

Il y a deux concepts mis en jeu :



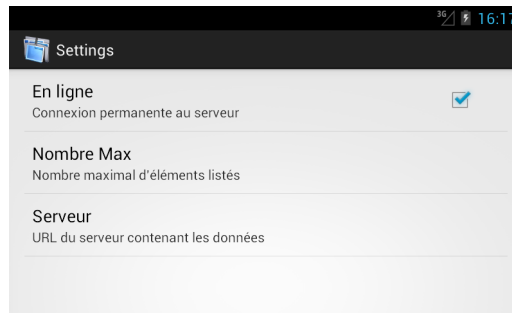


Figure 38: Préférences de l'application

- Une activité pour afficher et modifier les préférences.
- Une sorte de base de données qui stocke les préférences,
  - booléens,
  - nombres : entiers, réels...
  - chaînes et ensembles de chaînes.

Chaque préférence possède un *identifiant*. C'est une chaîne comme "prefs\_nbmax". La base de données stocke une liste de couples (*identifiant*, *valeur*).

Voir la [documentation Android](#)

### 5.4.3. Définition des préférences

D'abord, construire le fichier `res/xml/preferences.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="...">
    <CheckBoxPreference android:key="prefs_online"
        android:title="En ligne"
        android:summary="Connexion permanente au serveur"
        android:defaultValue="true" />
    <EditTextPreference android:key="prefs_nbmax"
        android:title="Nombre Max"
        android:summary="Nombre maximal d'éléments listés"
        android:inputType="number"
        android:numeric="integer"
        android:defaultValue="100" />
    ...
</PreferenceScreen>
```

### 5.4.4. Explications

Ce fichier xml définit à la fois :


- Les préférences :
  - l'identifiant : `android:key`
  - le titre résumé : `android:title`
  - le sous-titre détaillé : `android:summary`
  - la valeur initiale : `android:defaultValue`

- La mise en page. C'est une sorte de layout contenant des cases à cocher, des zones de saisie... Il est possible de créer des pages de préférences en cascade comme par exemple, les préférences système.

Consulter [la doc](#) pour connaître tous les types de préférences.

NB: le résumé n'affiche malheureusement pas la valeur courante. Consulter [stackoverflow](#) pour une proposition.

### 5.4.5. Accès aux préférences

Les préférences sont gérées par une classe statique appelée `PreferenceManager`. On doit lui demander une instance de `SharedPreferences` qui représente la base et qui possède des *getters* pour chaque type de données. 

```
// récupérer la base de données des préférences
SharedPreferences prefs = PreferenceManager
    .getDefaultSharedPreferences(getBaseContext());


// récupérer une préférence booléenne
boolean online = prefs.getBoolean("prefs_online", true);
```

Les *getters* ont deux paramètres : l'identifiant de la préférence et la valeur par défaut.

### 5.4.6. Préférences chaînes et nombres

Pour les chaînes, c'est `getString(identifiant, défaut)`.


```
String hostname = prefs.getString("prefs_hostname", "localhost");
```

Pour les entiers, il y a bug important (février 2015). La méthode `getInt` plante. Voir [stackoverflow](#) pour une solution. Sinon, il faut passer par une conversion de chaîne en entier : 

```
int nbmax = prefs.getInt("prefs_nbmax", 99);    // PLANTE
int nbmax =
    Integer.parseInt(prefs.getString("prefs_nbmax", "99"));
```

### 5.4.7. Modification des préférences par programme

Il est possible de modifier des préférences par programme, dans la base `SharedPreferences`, à l'aide d'un objet appelé *editor* qui possède des *setters*. Les modifications font partie d'une transaction comme avec une base de données.

Voici un exemple : 

```
// début d'une transaction
SharedPreferences.Editor editor = prefs.edit();
// modifications
editor.putBoolean("prefs_online", false);
```

```
editor.putInt("prefs_nbmax", 20);  
// fin de la transaction  
editor.commit();
```

### 5.4.8. Affichage des préférences

Il faut créer une activité toute simple :

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

Le layout `prefs_activity.xml` contient seulement un fragment :

```
<fragment xmlns:android="..."  
    android:id="@+id/frag_prefs"  
    android:name="LE.PACKAGE.COMPLET.PrefsFragment"  
    ... />
```

Mettre le nom du package complet devant le nom du fragment.

### 5.4.9. Fragment pour les préférences

Le fragment `PrefsFragment` hérite de `PreferenceFragment` :

```
public class PrefsFragment extends PreferenceFragment {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        // charger les préférences  
        addPreferencesFromResource(R.xml.preferences);  
        // mettre à jour les valeurs par défaut  
        PreferenceManager.setDefaultValues(  
            getActivity(), R.xml.preferences, false);  
    }  
}
```

C'est tout. Le reste est géré automatiquement par Android.

## 5.5. Bibliothèque support

### 5.5.1. Compatibilité des applications

Android est un système destiné à de très nombreux types de tablettes, téléphones, lunettes, montres et autres. D'autre part, il évolue pour offrir de nouvelles possibilités. Cela pose deux types de problèmes :

- Compatibilité des matériels,
- Compatibilité des versions d'Android.

Sur le premier aspect, chaque constructeur est censé faire en sorte que son appareil réagisse conformément aux spécifications de Google. Ce n'est pas toujours le cas quand les spécifications sont trop vagues. Certains créent leur propre API, par exemple Samsung pour la caméra.

### 5.5.2. Compatibilité des versions Android

Concernant l'évolution d'Android (deux versions du SDK par an, dont une majeure), un utilisateur qui ne change pas de téléphone à ce rythme est rapidement confronté à l'impossibilité d'utiliser des applications récentes.

Normalement, les téléphones devraient être mis à jour régulièrement, mais ce n'est quasiment jamais le cas.

Dans une application, le manifeste déclare la version nécessaire :

```
<uses-sdk android:minSdkVersion="17"  
          android:targetSdkVersion="25" />
```

Avec ce manifeste, si la tablette n'est pas au moins en API niveau 17, l'application ne sera pas installée. L'application est garantie pour bien fonctionner jusqu'à l'API 25 incluse.

### 5.5.3. Bibliothèque support

Pour créer des applications fonctionnant sur de vieux téléphones et tablettes, Google propose une solution depuis 2011 : une API alternative, « *Android Support Library* ». Ce sont des classes similaires à celles de l'API normale, mais qui sont programmées pour fonctionner partout, quel que soit la version du système installé.

C'est une approche intéressante qui compense l'absence de mise à jour des tablettes : au lieu de mettre à jour les appareils, Google met à jour la bibliothèque pour que les dispositifs les plus récents d'Android (ex: ActionBar, Fragments, etc.) fonctionnent sur les plus anciens appareils.

### 5.5.4. Versions de l'Android Support Library

Il en existe plusieurs variantes, selon l'ancienneté qu'on vise. Le principe est celui de l'attribut `minSdkVersion`, la version de la bibliothèque : `v4`, `v7` ou `v11` désigne le niveau minimal exigé pour le matériel qu'on vise.

- `v4` : c'est la plus grosse API, elle permet de faire tourner une application sur tous les appareils depuis Android 1.6. Par exemple, elle définit la classe `Fragment` utilisable sur ces téléphones. Elle contient même des classes qui ne sont pas dans l'API normale, telles que `ViewPager`.
- `v7-appcompat` : pour les tablettes depuis Android 2.1. Par exemple, elle définit l'ActionBar. Elle s'appuie sur la `v4`.
- Il y en a d'autres, plus spécifiques, `v8`, `v13`, `v17`.

### 5.5.5. Mode d'emploi

La première chose à faire est de définir le niveau de SDK minimal nécessaire dans le manifeste. Actuellement, début 2017, c'est 9 :

```
<uses-sdk android:minSdkVersion="9" .../>
```

Ensuite, il faut modifier le script `build.gradle` :

```
dependencies {  
    compile fileTree(dir: 'libs', include: ['*.jar'])  
    compile 'com.android.support:support-compat:25.1.1'  
    compile 'com.android.support:support-core-utils:+'  
}
```

On rajoute les éléments nécessaires. C'est assez compliqué. Il y a à la fois le **nom** : `support-compat`, `support-core-ui`, `support-core-utils`, `support-fragment`, `appcompat-v7...` et un numéro de version complet, ou seulement `+` pour la plus récente.

### 5.5.6. Programmation

Enfin, il suffit de faire appel à ces classes pour travailler. Elles sont par exemple dans le package `android.support.v4`.

```
import android.support.v4.app.FragmentActivity;  
  
public class MainActivity extends FragmentActivity  
...
```

Il y a de très nombreuses classes et paquets, c'est difficile à comprendre quand on débute.

Le problème, c'est qu'elles ont parfois le même nom que les classes normales d'Android, ex: `Fragment`, `FragmentActivity` et parfois pas du tout, ex: `AppCompatActivity`. Il y a aussi des conflits inextricables entre ces classes, ex: `LoaderManager` (voir la semaine prochaine).

### 5.5.7. Il est temps de faire une pause

C'est fini pour cette semaine, rendez-vous la semaine prochaine pour un cours sur les adaptateurs de bases de données et les WebServices.

## Semaine 6

---

### Bases de données SQLite3 <>

---

Après avoir représenté une liste d'items sous forme d'un tableau en semaine 4, nous allons la stocker dans un SGBD SQL.



Figure 39: Logo de SQLite3

- SQLite3
- Requêtes et curseurs
- ContentProviders

### 6.1. SQLite3

#### 6.1.1. Stockage d'informations

Il n'est pas pertinent d'enregistrer des informations dans un tableau stocké en mémoire vive, c'est à dire sur un support volatile. Android contient un SGBD SQL appelé SQLite3, parfait pour stocker des informations durables.

Exemple en ligne de commande :



```
bash$ sqlite3
sqlite> CREATE TABLE Planetes (
    _id INTEGER PRIMARY KEY AUTOINCREMENT,
    nom TEXT NOT NULL,
    distance REAL);
sqlite> INSERT INTO Planetes VALUES (1, "Sedna", 12925.26);
sqlite> SELECT * FROM Planetes;
1|Sedna|12925.26
sqlite>
```

#### 6.1.2. SQLite3

SQLite3 est un vrai SGBD relationnel SQL, mais simplifié pour tenir sur une tablette.

Ce qui lui manque :

- Aucune gestion des utilisateurs (autorisations), pas de sécurité.
- Pas de réglages pour améliorer les performances car
- Peu de types de données, ex: date = entier ou chaîne, un seul type d'entiers...

SQLite3 fonctionne sans serveur. Il stocke ses données dans un seul fichier. Ce fichier est portable, c'est à dire copiable sur n'importe quelle autre machine.

### 6.1.3. Exemples SQL

Toutes les requêtes SQL que vous connaissez fonctionnent, p. ex. :



```
SELECT COUNT(*) FROM Planetes WHERE nom LIKE 'Ter%';  
SELECT * FROM Planete WHERE distance>200.0 ORDER BY distance;  
SELECT AVG(distance) AS moyenne FROM Planetes;  
DELETE FROM Planetes WHERE distance IS NULL;  
ALTER TABLE Planetes ADD COLUMN date INTEGER;  
UPDATE Planetes SET distance=12775.66 WHERE _id=1;  
DROP TABLE Planetes;
```

Jointures, groupements, requêtes imbriquées, transactions, index, triggers... tout cela est possible.  
Consulter [la documentation](#).

### 6.1.4. Autres usages de SQLite3

Ce SGBD est utilisé dans de nombreuses applications ailleurs que dans Android, p. ex. dans Firefox pour stocker les marque-pages, l'historique de navigation, etc.

SQLite3 fournit aussi une API pour différents langages de programmation : C, Python, PHP, Java... On peut exécuter des requêtes SQL en appelant des fonctions.

Android propose cette API aux programmeurs pour stocker des informations structurées, plutôt que bricoler avec des fichiers. C'est assez facile à utiliser une fois le cadre mis en place.

### 6.1.5. Lancement de sqlite3 en shell

En ligne de commande, `sqlite3` ouvre une session SQL. Il n'y a pas de connexion à un serveur, donc pas de mot de passe.

On peut fournir un paramètre, c'est le nom d'un fichier qui contient la base. Si on ne fournit pas ce nom, alors la base n'est qu'en mémoire, perdue quand on quitte.

```
bash$ sqlite3 planetes.db  
sqlite>
```

Cette commande est dans le dossier du SDK, sous-dossier `platform-tools` (Linux et Windows). Elle n'est pas forcément incluse dans le système Linux de la tablette.

### 6.1.6. Commandes internes

Le shell de SQLite3 possède des commandes internes, p. ex. :

- .help** affiche la liste des commandes internes
- .tables** affiche la liste des tables
- .indices *table*** affiche la liste des index de la table si elle est fournie, toutes sinon
- .schema *table*** affiche la structure de la table (toutes si pas de nom fourni)
- .dump *table*** affiche le contenu de la table ou de toute la base si la table est omise
- .headers** mettre on ou off pour écrire les noms des colonnes en tête de tous les **select**
- .exit** sort du shell `sqlite3`, CTRL-D le fait aussi.

## 6.2. SQLite dans une application Android

### 6.2.1. Bases de données Android

Chaque application peut créer une base de données. C'est un fichier `*.db` placé dans le dossier `/data/data/PAQUETAGE/databases/NOM_BDD`

Remarque : le chemin exact peut varier selon les tablettes et AVD.

Vous pourrez échanger ce fichier avec le PC (`adb push` ou `pull`). Consulter [cette page](#) pour des détails sur la marche à suivre.

Dans une application Android, ces fichiers sont manipulés à l'aide de classes de l'API.

NB: ce cours commence par une grande simplification (l'ouverture d'une BDD). Lisez la totalité pour savoir comment on procède en réalité.

### 6.2.2. Classes pour travailler avec SQLite

Il faut connaître au moins deux classes :

- **SQLiteDatabase** : elle représente une BDD. Ses méthodes permettent d'exécuter une requête, par exemple :
  - `void execSQL(String sql)` pour `CREATE`, `ALTER`, `DROP`... qui ne retournent pas de données.
  - `Cursor rawQuery(String sql, ...)` pour des `SELECT` qui retournent des n-uplets.
  - D'autres méthodes pour des requêtes spécialisées.
- **Cursor** : représente un n-uplet. Il y a des méthodes pour récupérer les colonnes.

Voyons les détails.

### 6.2.3. Étapes du travail avec une BDD

Voici les étapes du travail avec une BDD en Java :

1. Ouverture de la base, création du fichier si besoin :

```
SQLiteDatabase bdd;  
bdd = SQLiteDatabase.openOrCreateDatabase(...);
```
2. Exécution de requêtes :
  1. Obtention d'un « curseur » sur le résultat des `select`

```
Cursor cursor = bdd.rawQuery(requete, ...);
```



2. Parcours des n-uplets du curseur à l'aide d'une boucle `for`
3. Fermeture du curseur (indispensable, prévoir les exceptions, voir plus loin)  
`cursor.close()`
3. Fermeture de la base (indispensable, prévoir les exceptions) :  
`bdd.close()`

#### 6.2.4. Base ouverte dans une activité

Il est préférable d'ouvrir la base pour toute la vie de l'activité dans `onCreate`, et la fermer dans la méthode `onDestroy` :

```
class MonActivite extends Activity
{
    private SQLiteDatabase bdd;

    void onCreate(...) {
        ...
        bdd = SQLiteDatabase.openOrCreateDatabase(...);
    }

    void onDestroy() {
        ...
        bdd.close();
    }
}
```

#### 6.2.5. Patron de conception pour les requêtes

Il est très conseillé de définir une classe associée à chaque table (et même chaque jointure). Ça permet de faire évoluer le logiciel assez facilement. Cette classe regroupe toutes les requêtes SQL la concernant : création, suppression, mise à jour, parcours, insertions... sous forme de méthodes de classe. La base de données est passée en premier paramètre de toutes les méthodes.

Cette démarche s'inspire du patron de conception [Active Record](#) qui représente une table par une classe. Ses instances sont les n-uplets de la table. Les attributs de la table sont gérés par des accesseurs et mutateurs. Des méthodes comme `find` permettent de récupérer des n-uplets et d'autres méthodes permettent la mise à jour de la base : `new`, `save` et `delete`.

Voir [ActiveAndroid](#) et [Realm](#) pour une implantation Android.

#### 6.2.6. Noms des colonnes

Dans ce cours, on va mettre en œuvre une simplification de ce patron de conception.

On définit une classe ne contenant que des méthodes statiques.

```
public class TablePlanetes
{
    // méthodes statiques pour gérer les données
}
```

```
public static create(...) ...  
  
public static drop(...) ...  
  
public static insert(...) ...  
  
...  
}
```

### 6.2.7. Classe pour une table

Par exemple, pour créer et supprimer la table :



```
public static void create(SQLiteDatabase bdd)  
{  
    bdd.execSQL( "CREATE TABLE Planetes (" +  
                  "_id INTEGER PRIMARY KEY AUTOINCREMENT," +  
                  "nom TEXT NOT NULL," +  
                  "distance REAL)");  
}  
  
public static void drop(SQLiteDatabase bdd)  
{  
    bdd.execSQL("DROP TABLE IF EXISTS Planetes");  
}
```

Gare à la syntaxe, ça doit créer du SQL correct !

### 6.2.8. Exemples de méthodes

Voici des méthodes pour créer ou supprimer des n-uplets :




```
public static void insert(SQLiteDatabase bdd, Planete pl)  
{  
    bdd.execSQL(  
        "INSERT INTO Planetes VALUES (null, ?, ?)",  
        new Object[] { pl.getNom(), pl.getDistance() });  
}  
  
public static void delete(SQLiteDatabase bdd, long id)  
{  
    bdd.execSQL(  
        "DELETE FROM Planetes WHERE _id=?",  
        new Object[] {id});  
}
```

On va aussi rajouter des méthodes de consultation des données mais avant, voyons la méthode `execSQL`.

### 6.2.9. Méthodes SQLiteDatabase.execSQL

Cette méthode exécute une requête SQL qui ne retourne pas d'informations : CREATE, INSERT... Elle a deux variantes :

- `void execSQL (String sql)` : on doit juste fournir la requête sous forme d'une chaîne. C'est une requête constante. Ne pas mettre de ; à la fin. Par exemple : 

```
bdd.execSQL("DROP TABLE Planetes");
```

- `void execSQL (String sql, Object[] bindArgs)` : c'est pour le même type de requête mais contenant des jokers ? à affecter avec les objets fournis en paramètre. 

```
bdd.execSQL("DELETE FROM Planetes WHERE _id BETWEEN ? AND ?",  
    new Object[] { 3, 8 });
```

### 6.2.10. Méthodes spécialisées

Android propose des méthodes spécifiques pour insérer, modifier, supprimer des n-uplets :

- `int insert(String table, null, ContentValues valeurs)`
- `int update(String table, ContentValues valeurs, String whereClause, String[] whereArgs)`
- `int delete(String table, String whereClause, String[] whereArgs)`


La différence avec `execSQL`, c'est qu'elles demandent un tableau de `String`. Il faut donc convertir toutes les données en chaînes.

### 6.2.11. Méthode insert

`insert(table, null, valeurs)` effectue une requête du type :

```
INSERT INTO table (col1, col2...) VALUES (val1, val2...);
```

Elle retourne l'identifiant du nouveau n-uplet. Ses paramètres sont :

- `table` : fournir le nom de la table
- `null` sauf si vous voulez faire un `INSERT INTO table`; sans fournir aucune valeur (valeurs toutes par défaut)
- `valeurs` : c'est une structure du type `ContentValues` qui associe des noms et des valeurs quelconques : 

```
ContentValues valeurs = new ContentValues();  
valeurs.putNull("_id");  
valeurs.put("nom", "Sedna");  
int id = bdd.insert("Planetes", null, valeurs);
```

### 6.2.12. Méthodes update et delete

`update(table, valeurs, whereClause, whereArgs)` fait `UPDATE table SET col1=val1, col2=val2 WHERE ...`; et `delete(table, whereClause, whereArgs)` effectue `DELETE FROM table WHERE ...`; Elles retournent le nombre de n-uplets altérés. Les paramètres sont :

- `table` : fournir le nom de la table
- `valeurs` : ce sont les couples (colonne, valeur à mettre)
- `whereClause` : une condition contenant des jokers ?
- `whereArgs` : chaînes à mettre à la place des ?



```
ContentValues valeurs = new ContentValues();
valeurs.put("nom", "Sedna");
bdd.update("Planetes", valeurs, "_id=?", new String[] { "4" });
bdd.delete("Planetes", "_id BETWEEN ? AND ?", new String[] {"5", "8"});
```

### 6.2.13. Méthode rawQuery

Cette méthode, `rawQuery` permet d'exécuter des requêtes de type `SELECT`. Elle retourne un objet Java de type `Cursor` qui permet de parcourir les n-uplets un à un :



```
Cursor cursor = bdd.rawQuery("SELECT * FROM table WHERE...");
if (cursor != null) {
    try {
        if (cursor.moveToFirst()) {           // obligatoire
            while (!cursor.isAfterLast()) {    // test de fin
                ...utiliser le curseur...
                cursor.moveToNext();          // n-uplet suivant
            }
        }
    } finally {
        cursor.close();
    }
}
```

### 6.2.14. rawQuery pour un seul n-uplet

S'il n'y a qu'un seul n-uplet dans la réponse, il n'est pas nécessaire de faire une boucle, mais il faut quand même initialiser le curseur par `moveToFirst` et le fermer à la fin :



```
Cursor cursor = bdd.rawQuery("SELECT MAX(distance) FROM Planetes");
if (cursor != null) {
    try {
        if (cursor.moveToFirst()) {           // obligatoire
            float distmax = cursor.getFloat(0);
        }
    } finally {
        cursor.close();                       // obligatoire
    }
}
```

Le `finally` garantit la fermeture du curseur, même en cas d'exception et de `return` dans le `try`.

### 6.2.15. Classe `Cursor`

La classe `Cursor` propose deux types de méthodes :

- celles qui permettent de parcourir les n-uplets :
  - `getCount()` : retourne le nombre de n-uplets,
  - `getColumnCount()` : retourne le nombre de colonnes,
  - `moveToFirst()` : place le curseur sur le premier n-uplet,
  - `isAfterLast()` : retourne vrai si le parcours est fini,
  - `moveToNext()` : passe au n-uplet suivant.
- celles qui permettent d'obtenir la valeur de la colonne n°*nc* allant de 0 à `getColumnCount()-1` du n-uplet courant :
  - `getColumnName(nc)` : retourne le nom de la colonne *nc*,
  - `isNull(nc)` : vrai si la colonne *nc* est nulle,
  - `getInt(nc)`, `getLong(nc)`, `getFloat(nc)`, `getString(nc)`, etc. : valeur de la colonne *nc*.

### 6.2.16. Exemple de requête, classe `TablePlanetes`



```
public static String getNom(SQLiteDatabase bdd, long id)
{
    Cursor cursor = bdd.rawQuery(
        "SELECT nom FROM Planetes WHERE _id=?",
        new String[] {Long.toString(id)});
    if (cursor == null) return null;
    try {
        if (cursor.moveToFirst() && !cursor.isNull(0)) {
            return cursor.getString(0);
        } else
            return null;
    } finally {
        cursor.close();
    }
}
```

### 6.2.17. Autre type de requête

Cette autre méthode retourne non pas une valeur, mais directement un curseur. Elle est utilisée pour afficher tous les éléments de la table dans une liste, voir page 114.



```
public static Cursor getAll(SQLiteDatabase bdd)
{
    return bdd.rawQuery("SELECT * FROM Planetes", null);
}
```

Attention, votre application doit prendre soin de fermer ce curseur dès qu'il ne sert plus, ou alors de le fournir à un objet (ex: un adaptateur) qui sait le fermer automatiquement.

### 6.2.18. Méthodes query : sans aucun intérêt

Android propose également des méthodes « pratiques » pour effectuer des requêtes, telles que :

```
query(String table, String[] columns, String selection,  
      String[] selectionArgs, String groupBy, String having,  
      String orderBy, String limit)
```

mais je ne vois pas l'intérêt de recoder en Java ce qui se fait parfaitement en SQL, sans compter les risques d'erreur si on permute involontairement les paramètres de ces méthodes.

### 6.2.19. Ouverture d'une base

Revenons vers les aspects gestion interne de la base de données. L'ouverture d'une base se fait ainsi :



```
String path = this.getFilesDir().getPath().concat("/test.db");  
SQLiteDatabase bdd =  
    SQLiteDatabase.openOrCreateDatabase(path, null);
```

NB: cela ne crée pas les tables, seulement le fichier qui contient la base.

Il faut fournir le chemin d'accès à la base. Mais en faisant ainsi, la base est créée dans /data/data/\*package\*/files et non pas .../databases. Voir page 109 pour la véritable façon de faire.

### 6.2.20. Première ouverture et ouvertures suivantes

Ensuite, après avoir ouvert la base, si c'est la première fois, il faut créer les tables. Cependant, ça cause une erreur de créer une table qui existe déjà et il serait coûteux de tester l'existence des tables.

Une possibilité consiste à rajouter IF NOT EXISTS à la requête de création. Par exemple :



```
bdd.execSQL(  
    "CREATE TABLE IF NOT EXISTS Planetes ("      +  
        "_id INTEGER PRIMARY KEY AUTOINCREMENT,"  +  
        "nom TEXT NOT NULL)");
```

Un autre problème, c'est la mise à jour de l'application. Qu'allez-vous proposer à vos clients si vous changez le schéma de la base entre la V1 et la V2, la V2 et la V3... ?

### 6.2.21. Un *helper* pour gérer l'ouverture/création/màj

Android propose la classe supplémentaire `SQLiteOpenHelper` qui facilite la gestion des bases de données. Il faut programmer une dérivation de cette classe en surchargeant deux méthodes :

- `onCreate(bdd)` : cette méthode est appelée quand la base de données n'existe pas encore. Son rôle est de créer les tables. C'est là que vous mettez les `CREATE TABLE...`

- `onUpgrade(bdd, int oldV, int newV)` : cette méthode est appelée quand la version de l'application est supérieure à celle de la base. Son rôle peut être de faire des `ALTER TABLE...`, `UPDATE...`

Les méthodes `getReadableDatabase` et `getWritableDatabase` de `SQLiteOpenHelper` ouvrent la base et appellent automatiquement `onCreate` et `onUpgrade` si nécessaire.

### 6.2.22. Exemple de *helper*



```
public class MySQLiteHelper extends SQLiteOpenHelper
{
    // nom du fichier contenant la base de données
    private static final String DB_NAME = "test.db";

    // version du schéma de la base de données
    private static final int DB_VERSION = 1;

    // constructeur du helper = ouvre et crée/màj la base
    public MySQLiteHelper(Context context)
    {
        super(context, DB_NAME, null, DB_VERSION);
    }

    ...
}
```

### 6.2.23. Exemple de *helper*, suite



```
@Override
public void onCreate(SQLiteDatabase bdd)
{
    // création avec la méthode de la classe TablePlanetes
    TablePlanetes.create(bdd);
}

@Override
public void onUpgrade(SQLiteDatabase bdd,
    int oldVersion, int newVersion)
{
    // suppression de toutes les données !
    TablePlanetes.drop(bdd);
    // re-création de la base
    onCreate(bdd);
}
}
```

### 6.2.24. méthode *onUpgrade*

```
public void onUpgrade(SQLiteDatabase bdd,  
                      int oldVersion, int newVersion)
```

Dans le cas d'une application sérieuse, on ne détruit pas toutes les données utilisateur quand on change le schéma. C'est à vous de déterminer les modifications minimales qui permettent de transformer les données présentes, de leur version actuelle *oldVersion* à la version *newVersion*.

Il est indiqué de procéder par étapes :

- passer de la version *oldVersion* à la *oldVersion+1*
- passer de la version *oldVersion+1* à la *oldVersion+2*
- ainsi de suite, jusqu'à arriver à la *newVersion*.

### 6.2.25. méthode *onUpgrade*, suite

Cela donne quelque chose comme ça :



```
@Override  
public void onUpgrade(..., int oldVersion, int newVersion){  
    while (oldVersion < newVersion) {  
        switch (oldVersion) {  
            case 1: // amener la base de la V1 à la V2  
                bdd.execSQL("ALTER TABLE Planetes ADD COLUMN taille REAL");  
                break;  
            case 2: // amener la base de la V2 à la V3  
                ...  
                break;  
        }  
        oldVersion++;  
    }  
}
```

### 6.2.26. Utilisation du Helper dans l'application

Avec un *helper*, cela devient très simple d'ouvrir une base, en consultation seule ou en modification :



```
public class MainActivity extends Activity {  
    private MySQLiteHelper helper;  
    private SQLiteDatabase bdd;  
  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        ...  
        helper = new MySQLiteHelper(this);  
        bdd = helper.getReadableDatabase();  
    }  
}
```



```
        /* ou bien */  
        bdd = helper.getWritableDatabase();  
        ...  
    }  
}
```

### 6.2.27. Fermeture de la base

A la terminaison de l'application, c'est le *helper* qu'il faut fermer, et c'est lui qui ferme la base :

```
@Override  
protected void onDestroy()  
{  
    super.onDestroy();  
    ...  
    helper.close();  
    ...  
}
```

Ainsi, la base reste ouverte pendant toute la vie de l'activité. Elle est représentée par la variable `bdd`, transmise à tous les appels à la classe `TablePlanetes`.

## 6.3. CursorAdapter et Loaders

### 6.3.1. Lien entre une BDD et un ListView

On revient vers l'application qui affiche une liste. Cette fois, la liste doit être le résultat d'une requête `SELECT`. Comment faire ?

Les choses sont devenues relativement complexes depuis Android 3. Afin d'éviter que l'application se bloque lors du calcul de la requête et voir le message « l'application ne répond pas », Android emploie un mécanisme appelé *chargeur*, *loader* en anglais.

Le principe est de rendre le calcul de la requête SQL *asynchrone*, désynchronisé de l'interface. On lance la requête `SELECT` et en même temps, on affiche une liste vide. Lorsque la requête sera finie, la liste sera mise à jour, mais en attendant, l'interface ne reste pas bloquée.


### 6.3.2. Étapes à suivre

- Méthode `onCreate` de l'activité qui affiche la liste :
  1. Définir un adaptateur de curseur pour la liste
  2. Ouvrir la base de données
  3. Créer un chargeur de curseur et l'associer à `this`
  4. Démarrer le chargeur
- Définir la classe `MonCursorLoader`, sous-classe de `CursorLoader` qui effectue la requête SQL
- Définir trois *callbacks* :
  - `onCreateLoader` : retourne un `MonCursorLoader`

- `onLoadFinished` et `onLoaderReset` : reçoivent un curseur à jour et mettent à jour l'adaptateur

Voici le détail.


### 6.3.3. Activité ou fragment d'affichage d'une liste

Cette activité hérite de `ListActivity` (ou `ListFragment`) et elle implémente les méthodes d'un « chargeur de curseur » : 

```
public class MainActivity extends ListActivity
    implements LoaderManager.LoaderCallbacks<Cursor>
{
    private MySQLiteHelper helper;
    private SQLiteDatabase bdd;
    private SimpleCursorAdapter adapter;


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

### 6.3.4. Création d'un adaptateur de curseur

Ça ressemble à l'adaptateur d'un tableau, mais on fournit deux listes : les noms des colonnes et les identifiants des vues dans lesquelles il faut mettre les valeurs. 

```
// créer un adaptateur curseur-liste
adapter = new SimpleCursorAdapter(this,
    // layout des éléments de la liste
    android.R.layout.simple_list_item_2,
    // le curseur sera chargé par le loader
    null,
    // noms des colonnes à afficher
    new String[]{"nom", "distance"},
    // identifiants des TextView qui affichent les colonnes
    new int[]{android.R.id.text1, android.R.id.text2},
    0); // options, toujours mettre 0
setListAdapter(adapter);
```

### 6.3.5. Ouverture de la base et création d'un chargeur

Ensuite, toujours dans la méthode `onCreate` de l'activité, on ouvre la base, ici en consultation car cette activité ne modifie pas les données, puis on crée un chargeur associé à `this`. 

```
// identifiant du chargeur (utile s'il y en a plusieurs)
private static final int LOADER_LISTE_PLANETES = 1;
// ouvrir la base de données SQLite
helper = new MySQLiteHelper(this);
bdd = helper.getReadableDatabase();
// crée et démarre un chargeur pour cette liste
getLoaderManager().initLoader(LOADER_LISTE_PLANETES, null, this);
```

Cette dernière instruction exige de définir trois *callbacks* dans l'activité : `onCreateLoader`, `onLoadFinished` et `onLoaderReset`. Voyons d'abord la première.

### 6.3.6. Callback `onCreateLoader` de l'activité

Toujours dans la classe d'activité qui affiche la liste :

```
@Override
public Loader<Cursor> onCreateLoader(int loaderID,
                                     Bundle bundle)
{
    return new MonCursorLoader(getApplicationContext(), bdd);
}
```

Cette callback fait instancier un `MonCursorLoader` qui est une sous-classe de `CursorLoader`, définie dans notre application, voir le transparent suivant. Son rôle est de lancer la requête qui retourne le curseur contenant les données à afficher.


### 6.3.7. classe `MonCursorLoader`

```
static private class MonCursorLoader extends CursorLoader
{
    private SQLiteDatabase bdd;

    public MonCursorLoader(Context context, SQLiteDatabase bdd) {
        super(context);
        this.bdd = bdd;
    }
    @Override
    protected Cursor onLoadInBackground() {
        return TablePlanetes.getAll(bdd);
    }
}
```

Voir page 108 pour la méthode `getAll`, elle fait seulement `return bdd.rawQuery("SELECT * FROM Planetes", null);`


### 6.3.8. Callback `onLoadFinished` de l'activité

Pour finir, la callback qui est appelée lorsque les données sont devenues disponibles ; elle met à jour l'adaptateur, ce qui affiche les n-uplets dans la liste. L'autre callback est appelée si le chargeur doit être supprimé. On met donc toujours ceci : 

```
@Override
public void onLoadFinished(Loader<Cursor> loader,
    Cursor cursor) {
    adapter.changeCursor(cursor);
}

@Override
public void onLoaderReset(Loader<Cursor> loader) {
    adapter.changeCursor(null);
}
```

### 6.3.9. Mise à jour de la liste

Quand il faut mettre à jour la liste, si les données ont changé, il faut relancer le chargeur de curseur et non pas l'adaptateur. Cela se fait de la manière suivante : 

```
// le chargeur doit recommencer son travail
getLoaderManager().restartLoader(LOADER_LISTE_PLANETES, null, this);
```

### 6.3.10. En mode compatibilité

Si vous utilisez la bibliothèque support *appcompat-v7*, le type du `LoaderManager` est spécifique. Il faut que votre activité hérite de `AppCompatActivity`, et le `LoaderManager` s'obtient par `getSupportLoaderManager()`.

## 6.4. ContentProviders

### 6.4.1. Présentation rapide

Les *Fournisseurs de contenu* sont des sortes de tables de données disponibles d'une application à l'autre et accessibles à l'aide d'un URI (généralisation d'un URL). Un exemple est le carnet d'adresse de votre téléphone. D'autres applications que la téléphonie peuvent y avoir accès.

Un `ContentProvider` possède différentes méthodes ressemblant à celles des bases de données :

- `query` : retourne un *Cursor* comme le fait un `SELECT`,
- `insert`, `update`, `delete` : modifient les données,
- D'autres méthodes permettent de consulter le type MIME des données.

Comme c'est très compliqué à mettre en œuvre et que ça ressemble à une simple table SQL sans jointure, on n'en parlera pas plus ici.

### 6.4.2. C'est tout pour aujourd'hui

C'est fini pour cette semaine, rendez-vous la semaine prochaine pour un cours sur les WebServices.

## Semaine 7

---

### Services réseau

---

Le cours de cette semaine explique comment écrire une application Android utilisant un WebService, c'est à dire une base de données sur un serveur distant. Cela repose sur quelques concepts importants à connaître : les tâches asynchrones et les requêtes réseau.

## 7.1. WebServices

### 7.1.1. Base de donnée distante

On arrive au plus intéressant de tout ce cours, faire en sorte qu'une application Android stocke ses données sur un serveur distant. Pour commencer, révisez vos cours de Web Design, PHP, PDO...

### 7.1.2. Échange entre un serveur SQL et une application Android

Soit un serveur HTTP connecté à une base de données ([PostgreSQL](#) en TP). Ce serveur possède des scripts PHP qui vont répondre aux demandes de l'application Android à l'aide d'au moins [deux types d'échanges HTTP](#)<sup>4</sup> :

- Les **SELECT** vont être traitées par des **GET**,
- Les **INSERT**, **UPDATE**, **DELETE**... vont être envoyés par des **POST**.

Chaque requête sera associée à un script spécifique : `get_planete.php`, `get_all_planetes.php`, `insert_planete.php`, `delete_planete.php`...

La création et la destruction des tables ne seront pas possibles car gérées par l'administrateur sur le serveur.

### 7.1.3. Principe général

Soit la requête `SELECT * FROM Planetes WHERE _id=3`. On va envoyer l'identifiant 3 sur le réseau et c'est un script PHP qui va effectuer la requête. Il y a un script par sorte de requête, donc chacun sait exactement quels paramètres il va recevoir.


1. L'application construit une requête HTTP, p. ex. de type **GET**
  - URL = `http://serveur/script?paramètres`
  - paramètres = conditions du select, p. ex. `identifiant=3`.
2. L'application (cliente) envoie cette requête au serveur puis attend la réponse,
3. Le script PHP exécute la requête puis retourne le résultat encodé en *JSON* à l'application,
4. L'application décode le résultat et l'affiche.

Les autres requêtes suivent le même principe client-serveur.

---

<sup>4</sup>En fait, un vrai WebService Restful est plus complexe, voir [wikipedia](#)

#### 7.1.4. Exemple de script PHP Get

Voici `get_planete.php` qui retourne l'un des n-uplets : 


```
// connexion au serveur SQL par PDO
$db = new PDO("pgsql:host=$hostname;dbname=$dbname",
    $login, $passwd);

// paramètres de la requête (TODO: tester la présence)
$id = $_GET['_id'];

// requête SQL
$query = $db->prepare("SELECT * FROM Planetes WHERE _id=?");
$query->execute(array($id));

// encodage JSON de la réponse
echo json_encode($query->fetch(PDO::FETCH_NUM));
```

#### 7.1.5. Exemple de script PHP Get

Voici `get_all_planetes.php` qui retourne tous les n-uplets : 


```
// connexion au serveur SQL par PDO
$db = new PDO("pgsql:host=$hostname;dbname=$dbname",
    $login, $passwd);

// requête SQL
$query = $db->prepare("SELECT * FROM Planetes ORDER BY _id");
$query->execute();

// encodage JSON de toutes les réponses
echo json_encode($query->fetchAll(PDO::FETCH_NUM));
```

Ce script est appelé pour afficher la liste dans `MainActivity`.

#### 7.1.6. Exemple de script PHP Post

Voici le script `update_planete.php`. Il est lancé par un POST. 

```
// connexion au serveur SQL par PDO
$db = new PDO("pgsql:host=$hostname;dbname=$dbname",
    $login, $passwd);

// paramètres de la requête (TODO: tester la présence)
$id = $_POST['_id'];
$nom = $_POST['nom'];

// préparation et exécution
```

```
$query = $db->prepare("UPDATE Planetes SET nom=? WHERE _id=?");  
$query->execute(array($nom, $id));
```

NB: il n'est pas complet (autres champs) et il manque toutes les vérifications de sécurité (XSS, ...).

### 7.1.7. Format JSON *JavaScript Object Notation*

C'est un format concurrent de XML pour transporter des tableaux et des objets à travers le réseau. Ils sont écrits sous forme d'un texte.

Par exemple la liste des n-uplets présents dans la table Planetes :

```
[[1,"Mercure",58],[2,"Venus",108],[3,"Terre",150],...
```

En PHP, c'est très simple :

```
// encodage : tableau -> jsondata  
$jsondata = json_encode($tableau);  
// décodage : jsondata -> tableau  
$tableau = json_decode($jsondata);
```

Le tableau peut venir d'un `fetchAll(PDO::FETCH_NUM)`.

### 7.1.8. JSON en Java

Décoder le JSON en Java est plus compliqué. Il faut employer une instance de `JSONArray`. Elle possède des *setters* et des *getters* pour chaque type de données.

```
// encodage : tableau -> jsondata  
int[] tableau = new int[] { 2, -7, 5 };  
JSONArray ja = new JSONArray();  
for (int v: tableau) ja.put(v);  
String jsondata = ja.toString();  
// décodage : jsondata -> tableau  
JSONArray ja = new JSONArray(jsondata);  
final int nb = ja.length();  
int[] tableau = new int[nb];  
for (int i=0; i<nb; i++) tableau[i] = ja.getInt(i);
```

C'est à adapter aux données à échanger : entiers, chaînes...

### 7.1.9. Dans l'application Android

Tout le problème est de construire une requête HTTP, d'attendre la réponse, de la décoder et de l'afficher.

Pour commencer, il faut que l'application soit autorisée à accéder à internet. Rajouter cette ligne dans le manifeste :


```
<uses-permission android:name="android.permission.INTERNET"/>
```

Ensuite, il faut transformer tout ce qui est requête SQL :

- Affichage des données : changer le chargeur de curseur,
- Modifications des données.

Voyons cela dans l'ordre.

### 7.1.10. Affichage d'une liste

Il suffit de reprogrammer la méthode `getAll` de la classe `TablePlanetes`, voir cours précédent et aussi `get_all_planetes.php` page 117 : 

```
public static Cursor getAll(RemoteDatabase bdd)
{
    // requête Get à l'aide de la classe RemoteDatabase
    String jsondata = bdd.get("get_all_planetes.php", null);
    // décoder les n-uplets et en faire un curseur
    return bdd.cursorFromJSON(jsondata,
        new String[] { "_id", "nom", "distance" });
}
```

J'ai retiré les tests d'erreur et traitements d'exceptions.

### 7.1.11. La classe RemoteDatabase

C'est une classe que je vous propose. Elle fait un peu tout : le café, les croissants... C'est elle qui organise la communication avec le serveur, avec des méthodes comme :

- `get("script.php", params)` appelle le script PHP par un `GET` en lui passant les paramètres indiqués et retourne un *String* contenant la réponse du serveur sous forme de données JSON.
- `post(...)` est très similaire.
- `cursorFromJSON(jsondata, noms_des_colonnes)` construit un curseur avec la réponse JSON du serveur. On est obligé de fournir les noms des colonnes car ils ne sont pas présents dans les données JSON.

Cette classe est assez complexe, et tout ce qui suit maintenant explique les détails.

### 7.1.12. Modification d'un n-uplet

Voici par exemple une requête POST pour modifier un n-uplet : 

```
public static void update(RemoteDatabase bdd, Planete pl,
    RemoteDatabaseListener listener)
{
    // paramètres de la requête
    ContentValues params = new ContentValues();
    params.put("_id", pl.getId());
    params.put("nom", pl.getNom());
}
```



```
// requête Post asynchrone
bdd.post(listener, "update_planete.php", params);
}
```

Elle appelle le script `update_planete.php` , voir page 117.

### 7.1.13. Méthode `post(écouteur, script, params)`

La méthode `post(écouteur, script, params)` appelle un script PHP en lui fournissant des paramètres. Par exemple, c'est le script `update_type.php` avec les paramètres `_id` et `libelle`.

Elle a une particularité : cette méthode est *asynchrone*. C'est à dire qu'elle lance un échange réseau en arrière-plan, et n'attend pas qu'il se termine. C'est obligatoire, sinon Android affiche une erreur : *l'application ne répond pas*, dialogue « ANR ».

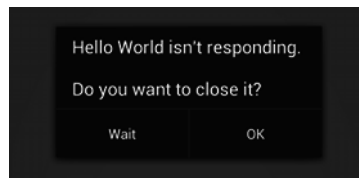


Figure 40: Dialogue ANR

### 7.1.14. Principe de la méthode `post`

Le script PHP appelé peut durer un certain temps : connexion réseau et travail du serveur. Pour ne pas bloquer l'application, la méthode `post` doit travailler en arrière-plan, comme avec le shell d'Unix, mais pouvoir prévenir l'application quand elle a fini.

Le principe pour cela est de créer une `AsyncTask`. Elle gère une action qui est exécutée dans un autre *thread* que celui de l'interface.

Du coup, il faut un écouteur à prévenir quand l'action est terminée. C'est le premier paramètre passé à la méthode `post`. Par exemple, c'est l'activité d'affichage de liste qui peut alors mettre à jour la liste affichée.

La méthode `get` est exactement pareille, elle aussi gère un écouteur à prévenir quand les données sont reçues du serveur.

## 7.2. AsyncTask

### 7.2.1. Présentation

Une activité Android repose sur une classe, ex `MainActivity` qui possède différentes méthodes comme `onCreate`, les écouteurs des vues, des menus et des chargeurs.

Ces fonctions sont exécutées par un seul processus léger, un *thread* appelé « Main thread ». Il dort la plupart du temps, et ce sont les événements qui le réveillent.

Ce *thread* ne doit jamais travailler plus de quelques fractions de secondes sinon l'interface paraît bloquée et Android peut même décider que l'application est morte (*App Not Responding*).

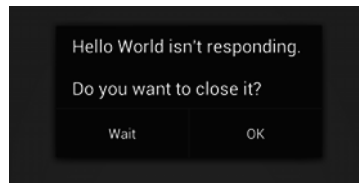


Figure 41: Application bloquée

### 7.2.2. Tâches asynchrones

Pourtant dans certains cas, une *callback* peut durer longtemps :

- gros calcul
- requête SQL un peu complexe
- requête réseau

La solution passe par une séparation des *threads*, par exemple à l'aide d'une tâche asynchrone [AsyncTask](#). C'est un autre *thread*, indépendant de l'interface utilisateur, comme un *job* Unix.

Lancer un **AsyncTask** ressemble à faire `commande &` en shell.

L'interface utilisateur peut être mise à jour de temps en temps par la **AsyncTask**. Il est également possible de récupérer des résultats à la fin de l'**AsyncTask**.

### 7.2.3. Principe d'utilisation d'une AsyncTask

Ce qui est mauvais :

1. Android appelle la *callback* de l'activité, ex: `onClick`
2. La *callback* a besoin de 20 secondes pour faire son travail,
3. Mais au bout de 5 secondes, Android propose de tuer l'application.

Ce qui est correct :

1. Android appelle la *callback* de l'activité,
2. La *callback* crée une **AsyncTask** puis sort immédiatement,
3. Le *thread* de l'**AsyncTask** travaille pendant 20 secondes,
4. Pendant ce temps, l'interface est vide, mais reste réactive,
5. L'**AsyncTask** affiche les résultats sur l'interface ou appelle un écouteur.

### 7.2.4. Structure d'une AsyncTask

Une tâche asynchrone est définie par plusieurs méthodes :

**Constructeur** permet de passer des paramètres à la tâche.

**onPreExecute** Initialisation effectuée par le *thread* principal, p. ex. elle initialise une barre d'avancement (**ProgressBar**).

**doInBackground** C'est le corps du traitement. Cette méthode est lancée dans son propre *thread*. Elle peut durer autant qu'on veut.

**onProgressUpdate** Cette méthode permet de mettre à jour l'interface, p. ex. la barre d'avancement. Pour ça, **doInBackground** doit appeler **publishProgress**.

**onPostExecute** Elle est appelée quand l'**AsyncTask** a fini, par exemple pour masquer la barre d'avancement et mettre à jour les données sur l'interface.

### 7.2.5. Paramètres d'une AsyncTask

Ce qui est difficile à comprendre, c'est que `AsyncTask` est une classe générique (comme `ArrayList`). Elle est paramétrée par trois types de données :

```
AsyncTask<Params, Progress, Result>
```

- *Params* est le type des paramètres de `doInBackground`,
- *Progress* est le type des paramètres de `onProgressUpdate`,
- *Result* est le type du paramètre de `onPostExecute` qui est aussi le type du résultat de `doInBackground`.

NB: ça ne peut être que des classes, donc `Integer` et non pas `int`, et `Void` au lieu de `void` (dans ce dernier cas, faire `return null;`).

### 7.2.6. Exemple de paramétrage

Soit une `AsyncTask` qui doit interroger un serveur météo pour savoir quel temps il va faire. Elle va retourner un réel indiquant de 0 à 1 s'il va pleuvoir. La tâche reçoit un `String` en paramètre (l'URL du serveur), publie régulièrement le pourcentage d'avancement (un entier) et retourne un `Float`. Cela donne cette instantiation du modèle générique :

```
class MyTask extends AsyncTask<String, Integer, Float>
```

et ses méthodes sont paramétrées ainsi :

```
Float doInBackground(String urlserveur)  
void onProgressUpdate(Integer pourcentage)  
void onPostExecute(Float pluie)
```

### 7.2.7. Paramètres variables

Alors en fait, c'est encore plus complexe, car `doInBackground` reçoit non pas un seul, mais un nombre quelconque de paramètres tous du même type. La syntaxe Java utilise la notation « ... » pour signifier qu'en fait, c'est un tableau de paramètres.

```
Float doInBackground(String... urlserveur)
```

Ça veut dire qu'on peut appeler la même méthode de toutes ces manières, le nombre de paramètres est variable :

```
doInBackground();  
doInBackground("www.meteo.fr");  
doInBackground("www.meteo.fr", "www.weather.fr", "www.bericht.fr");
```

Le paramètre `urlserveur` est équivalent à un `String[]` qui contiendra les paramètres.

### 7.2.8. Définition d'une AsyncTask

Il faut dériver et instancier la classe générique. Pour l'exemple, j'ai défini un constructeur qui permet de spécifier une `ProgressBar` à mettre à jour pendant le travail.

Par exemple :



```
private class PrevisionPluie
    extends AsyncTask<String, Integer, Float>
{
    // ProgressBar à mettre à jour
    private ProgressBar mBarre;

    // constructeur, fournir la ProgressBar concernée
    PrevisionPluie(ProgressBar barre) {
        this.mBarre = barre;
    }
}
```

### 7.2.9. AsyncTask, suite

Voici la suite avec la tâche de fond et l'avancement :



```
protected Float doInBackground(String... urlserveur) {
    float pluie = 0.0f;
    int nbre = urlserveur.length;
    for (int i=0; i<nbre; i++) {
        ... interrogation de urlserveur[i] ...
        // faire appeler onProgressUpdate avec le %
        publishProgress((int)(i*100.0f/nbre));
    }
    // ça va appeler onPostExecute(pluie)
    return pluie;
}

protected void onProgressUpdate(Integer... progress) {
    mBarre.setProgress( progress[0] );
}
```

### 7.2.10. Lancement d'une AsyncTask

C'est très simple, on crée une instance de cet `AsyncTask` et on appelle sa méthode `execute`. Ses paramètres sont directement fournis à `doInBackground` :



```
ProgressBar mProgressBar =
    (ProgressBar) findViewById(R.id.pourcent);
new PrevisionPluie(mProgressBar)
    .execute("www.meteo.fr", "www.weather.fr", "www.bericht.fr");
```

`execute` va créer un *thread* séparé pour effectuer `doInBackground`, mais les autres méthodes du `AsyncTask` restent dans le *thread* principal.

### 7.2.11. Schéma récapitulatif

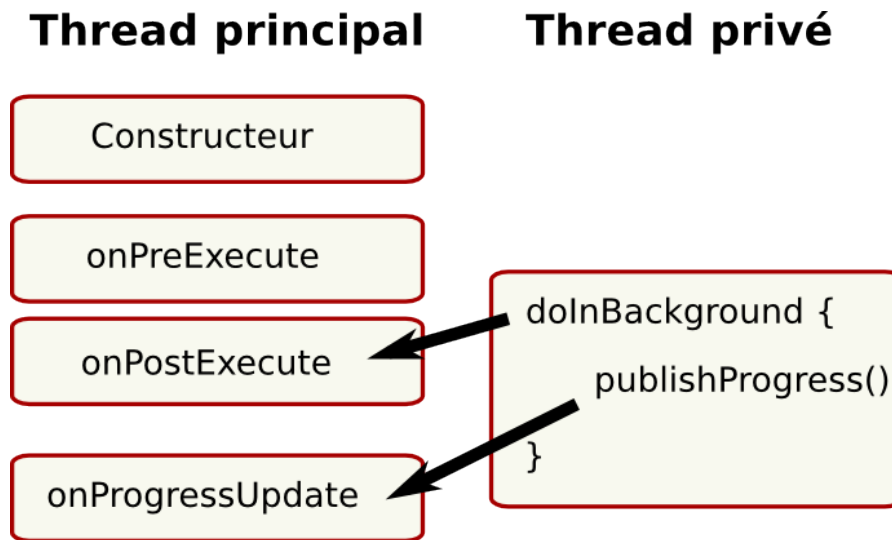


Figure 42: Méthodes d'un AsyncTask

### 7.2.12. execute ne retourne rien

En revanche, il manque quelque chose pour récupérer le résultat une fois le travail terminé. Pourquoi n'est-il pas possible de faire ceci ?


```
float pluie =
    new PrevisionPluie(mProgressBar).execute("www.meteo.fr");
```

Ce n'est pas possible car :

1. `execute` retourne void, donc rien,
2. l'exécution de `doInBackground` n'est pas dans le même *thread*, or un *thread* ne peut pas faire `return` dans un autre,
3. `execute` prend du temps et c'est justement ça qu'on veut pas.

Solutions : définir le *thread* appelant en tant qu'écouteur de cet `AsyncTask` ou faire les traitements du résultat dans la méthode `onPostExecute`.

### 7.2.13. Récupération du résultat d'un AsyncTask

Pour recevoir le résultat d'un `AsyncTask`, il faut généralement mettre en place un écouteur qui est déclenché dans la méthode `onPostExecute`. Exemple : 

```
public interface PrevisionPluieListener {
    public void onPrevisionPluieConnue(Float pluie);
}
// écouteur = l'activité qui lance l'AsyncTask
private PrevisionPluieListener ecouteur;
// appelée quand c'est fini, réveille l'écouteur
```

```
protected void onPostExecute(Float pluie) {  
    ecouteur.onPrevisionPluieConnue(pluie);  
}
```

L'écouteur est fourni en paramètre du constructeur, par exemple :

```
new PrevisionPluie(this, ...).execute(...);
```

#### 7.2.14. Simplification

On peut simplifier un peu s'il n'y a pas besoin de `ProgressBar` et si le résultat est directement utilisé dans `onPostExecute` :



```
private class PrevisionPluie  
    extends AsyncTask<String, Void, Float> {  
  
    protected Float doInBackground(String... urlserveur) {  
        float pluie = 0.0f;  
        // interrogation des serveurs  
        ...  
        return pluie;  
    }  
    protected void onPostExecute(Float pluie) {  
        // utiliser pluie, ex: l'afficher dans un TextView  
        ...  
    }  
}
```

#### 7.2.15. Recommandations

Il faut faire extrêmement attention à :

- ne pas bloquer le *thread* principal dans une *callback* plus de quelques fractions de secondes,
- ne pas manipuler une vue ailleurs que dans le *thread* principal.

Ce dernier point est très difficile à respecter dans certains cas. Si on crée un *thread*, il ne doit jamais accéder aux vues de l'interface. Un *thread* n'a donc aucun moyen direct d'interagir avec l'utilisateur. Si vous tentez quand même, l'exception qui se produit est :

*Only the original thread that created a view hierarchy can touch its views*

Les solutions dépassent largement le cadre de ce cours et passent par exemple par la méthode [Activity.runOnUiThread](#)

#### 7.2.16. Autres tâches asynchrones

Il existe une autre manière de lancer une tâche asynchrone :



```
Handler handler = new Handler();
final Runnable tache = new Runnable() {
    @Override
    public void run() {
        ... faire quelque chose ...
        // optionnel : relancer cette tâche dans 5 secondes
        handler.postDelayed(this, 5000);
    }
};
// lancer la tâche tout de suite
handler.post(tache);
```

Le handler gère le lancement immédiat (`post`) ou retardé (`postDelayed`) de la tâche. Elle peut elle-même se relancer.

## 7.3. Requêtes HTTP

### 7.3.1. Présentation

Voici quelques explications sur la manière de faire une requête HTTP d'une tablette vers un serveur. Android propose plusieurs mécanismes :

- un client HTTP Apache `DefaultHttpClient` bien pratique, mais il est obsolète depuis l'API 22,
- une classe appelée `URLConnection` maintenant recommandée,
- une API appelée `Volley` un peu trop complexe pour ce cours.

Vous savez que le protocole HTTP a plusieurs « méthodes », dont `GET`, `POST`, `PUT` et `DELETE` qui sont employées pour gérer un `WebService`. On va voir les deux premières.

### 7.3.2. Principe de programmation pour un GET

Voici les étapes :

1. Créer une instance de `URL` qui indique l'url de la page voulue, avec ses paramètres,
2. Créer une instance de `URLConnection` en appelant `openConnection()` sur l'`URL`,
3. (optionnel) Configurer la requête : agent, authentification, type mime, session, cookies...
4. Lire la réponse avec `getInputStream()`, intercepter les exceptions `IOException` s'il y a un problème,
5. Déconnecter afin de libérer la connexion.

Noter que le serveur peut mettre du temps à répondre, il faut donc placer cela dans une `AsyncTask`.

### 7.3.3. Exemple de requête GET



```
URL url = new URL("http://SERVEUR/get_avis.php?_id=2");
URLConnection connexion =
```

```
(URLConnection) url.openConnection();
connexion.setReadTimeout(10000);
connexion.setRequestProperty("User-Agent", "Mozilla/5.0");
try {
    InputStream reponse = connexion.getInputStream();
    int code = connexion.getResponseCode();

    ... utiliser new BufferedInputStream(reponse) ...
} catch (IOException e) {
    ... mauvais URL, pb réseau ou serveur inactif ...
} finally {
    connexion.disconnect();
}
```

#### 7.3.4. Encodage de paramètres pour une requête

Les paramètres d'une requête GET ou POST doivent être encodés (cf [wikipedia](#)). Les couples (*nom1, val1*), (*nom2, val2*) deviennent ?*nom1=val1&nom2=val2*. Dedans, les espaces sont remplacés par + et les caractères bizarres par leur code UTF8, ex: é devient %C3%A9.

Utiliser la méthode `URLEncoder.encode(chaine, charset)` :



```
String params =
    "?libelle=" + URLEncoder.encode(libelle, "UTF-8") +
    "&auteur=" + URLEncoder.encode(auteur, "UTF-8");
```

Voir le TP7 pour une implantation plus polyvalente (boucle sur un `ContentValues`).

#### 7.3.5. Principe de programmation pour un POST

Un POST est un peu plus complexe car il faut encoder un corps de requête. Le début est similaire à une requête GET, mais ensuite :

1. Configurer en mode POST
2. Fournir un contenu avec `getOutputStream()`,
3. (optionnel) Lire la réponse avec `getInputStream()`,
4. Déconnecter afin de libérer la connexion.

Le contenu est à placer dans le flux désigné par `getOutputStream()`, mais avant :

- soit on connaît la taille du contenu dès le début :
  - appeler `setFixedLengthStreamingMode(taille)`;
- soit on ne la connaît pas (ex: streaming) :
  - appeler `setChunkedStreamingMode(0)`;

#### 7.3.6. Exemple de requête POST





```
URL url = new URL("http://SERVEUR/insert_avis.php");
URLConnection connexion = (HUC..) url.openConnection();
try {
    connexion.setDoOutput(true);
    connexion.setRequestMethod("POST");
    String params = "libelle=ok&note=3.5&...";
    connexion.setFixedLengthStreamingMode(params.length());
    DataOutputStream contenu =
        new DataOutputStream(connexion.getOutputStream());
    contenu.writeBytes(params);
    contenu.close();
    ... éventuellement utiliser getInputStream ...
} finally {
    connexion.disconnect();
}
```

### 7.3.7. Requêtes asynchrones

Comme le serveur peut répondre avec beaucoup de retard, il faut employer une sous-classe d'`AsyncTask`. Par exemple ceci :

- Constructeur : on lui fournit l'URL à contacter ainsi que tous les paramètres nécessaires, ils sont simplement mémorisés dans la classe,
- `String doInBackground()` : ouvre la connexion, construit et lance la requête, retourne la réponse du serveur et ferme la connexion,
- `void onPostExecute(String reponse)` : traite la réponse du serveur ou réveille un écouteur.

### 7.3.8. Permissions pour l'application

Pour finir, il faut rajouter ceci dans le manifeste au même niveau que l'application :



```
<uses-permission android:name="android.permission.INTERNET"/>
```

Sans cela, les connexions réseau seront systématiquement refusées.

### 7.3.9. Voilà tout pour cette semaine

C'est fini pour cette semaine, rendez-vous la semaine prochaine pour un cours sur les cartes et sur le dessin en 2D.

## Semaine 8

---

### Cartes et Dessin 2D interactif

---

Le cours de cette semaine concerne le dessin de cartes et de figures 2D et les interactions avec l'utilisateur.

- OpenStreetMap, la carte mondiale libre
- CustomView et Canvas
- Un exemple de boîte de dialogue utile

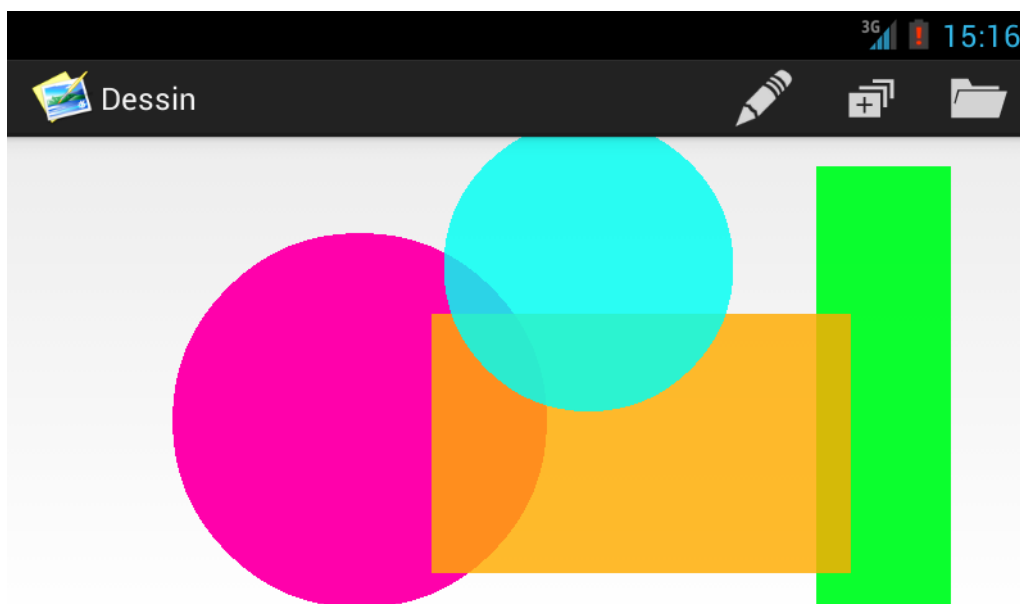


Figure 43: Application de dessin

## 8.1. OpenStreetMap

### 8.1.1. Présentation

Au contraire de Google Maps, OSM est vraiment libre et OpenSource, et il se programme extrêmement facilement.

### 8.1.2. Documentation

Nous allons utiliser deux librairies :

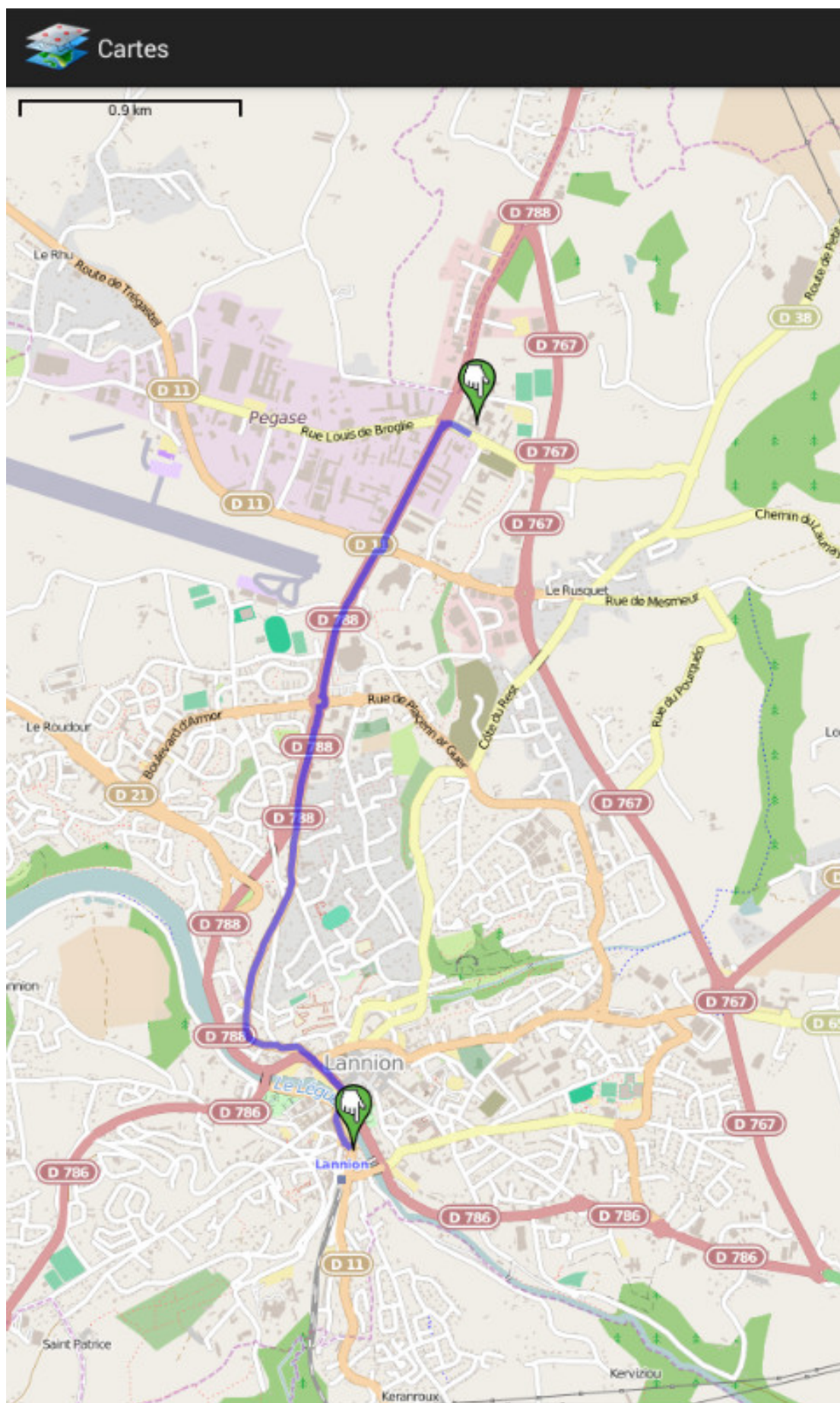


Figure 44: Google Maps

- [OSMdroid](#) : c'est la librairie de base, super mal documentée. Attention à ne pas confondre avec un site de piraterie.
- [OSMbonusPack](#), un ajout remarquable à cette base. Son auteur s'appelle Mathieu Kergall. Il a ajouté de très nombreuses fonctionnalités permettant entre autres d'utiliser OpenStreetMap pour gérer des itinéraires comme les GPS de voiture et aussi afficher des fichiers KML venant de Google Earth.

Lire [cette suite de tutoriels](#) pour découvrir les possibilités de osmbonuspack.

### 8.1.3. Pour commencer

Il faut d'abord installer plusieurs archives jar :

- [OSMbonusPack](#). Il est indiqué comment inclure cette librairie et ses dépendances dans votre projet AndroidStudio. Voir le TP8 partie 2 pour voir comment faire sans connexion réseau.
- [OSMdroid](#). C'est la librairie de base pour avoir des cartes OSM.
- GSON : c'est une librairie pour lire et écrire du JSON,
- OkHTTP et OKio : deux librairies pour générer des requêtes HTTP.

L'inclusion de librairies est à la fois simple et compliqué. La complexité vient de l'intégration des librairies et de leurs dépendances dans un serveur central, « maven ».

### 8.1.4. Layout pour une carte OSM

Ce n'est pas un fragment, mais une vue personnalisée :



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="..."
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <org.osmdroid.views.MapView
        android:id="@+id/map"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tilesource="Mapnik"/>
</LinearLayout>
```

Vous pouvez rajouter ce que vous voulez autour.

### 8.1.5. Activité pour une carte OSM

Voici la méthode onCreate minimale :



```
private MapView mMap;

@Override
protected void onCreate(Bundle savedInstanceState)
{
    // mise en place de l'interface
```

```
super.onCreate(savedInstanceState);
setContentView(R.layout.main_activity);

// rajouter les contrôles utilisateur
mMap = (MapView) findViewById(R.id.map);
mMap.setMultiTouchControls(true);
mMap.setBuiltInZoomControls(true);
}
```

### 8.1.6. Positionnement de la vue

Pour modifier la vue initiale de la carte, il faut faire appel au `IMapController` associé à la carte : 

```
// récupérer le gestionnaire de carte (= caméra)
IMapController mapController = mMap.getController();

// définir la vue initiale
mapController.setZoom(14);
mapController.setCenter(new GeoPoint(48.745, -3.455));
```


Un `GeoPoint` est un couple (latitude, longitude) représentant un point sur Terre. Il y a aussi l'altitude si on veut. C'est équivalent à un `LatLng` de GoogleMaps.

### 8.1.7. Calques

Les ajouts sur la carte sont faits sur des *overlays*. Ce sont comme des calques. Pour ajouter quelque chose, il faut créer un `Overlay`, lui rajouter des éléments et insérer cet overlay sur la carte.


Il existe différents types d'overlays, p. ex. :

- `ScaleBarOverlay` : rajoute une échelle
- `ItemizedIconOverlay` : rajoute des marqueurs
- `RoadOverlay`, `Polyline` : rajoute des lignes

Par exemple, pour rajouter un indicateur d'échelle de la carte : 

```
// ajouter l'échelle des distances
ScaleBarOverlay echelle = new ScaleBarOverlay(mMap);
mMap.getOverlays().add(echelle);
```

### 8.1.8. Mise à jour de la carte

Chaque fois qu'on rajoute quelque chose sur la carte, il est recommandé de rafraîchir la vue : 

```
// redessiner la carte
mMap.invalidate();
```

Ça marche sans cela dans la plupart des cas, mais y penser s'il y a un problème.

### 8.1.9. Marqueurs

Un marqueur est représenté par un Marker :



```
Marker mrkIUT = new Marker(mMap);
GeoPoint gpIUT = new GeoPoint(48.75792, -3.4520072);
mrkIUT.setPosition(gpIUT);
mrkIUT.setSnippet("Département INFO, IUT de Lannion");
mrkIUT.setAlpha(0.75f);
mrkIUT.setAnchor(Marker.ANCHOR_CENTER, Marker.ANCHOR_BOTTOM);
mMap.getOverlays().add(mrkIUT);
```

- `snippet` est une description succincte du marqueur,
- `alpha` est la transparence : 1.0=opaque, 0.0=invisible,
- `anchor` désigne le *hot point* de l'image, le pixel à aligner avec la position.

### 8.1.10. Marqueur personnalisés

Pour changer l'image par défaut (une main dans une poire), il vous suffit de placer une image png dans `res/drawable`. Puis charger cette image et l'attribuer au marqueur :



```
Drawable fleche = getResources().getDrawable(R.drawable.fleche);
mrkIUT.setIcon(fleche);
mrkIUT.setAnchor(Marker.ANCHOR_RIGHT, Marker.ANCHOR_BOTTOM);
```

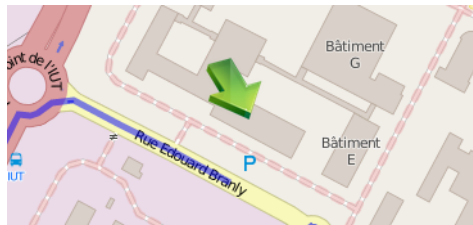


Figure 45: Marqueur personnalisé

### 8.1.11. Réaction à un clic

On peut définir un écouteur pour les clics sur le marqueur :




```
mrkIUT.setOnMarkerClickListener(new OnMarkerClickListener() {
    @Override
    public boolean onMarkerClick(Marker marker, MapView map)
    {
        Toast.makeText(MainActivity.this,
            marker.getSnippet(),
            Toast.LENGTH_LONG).show();
        return false;
    }
});
```

Ici, je fais afficher le *snippet* du marqueur dans un *Toast*.



### 8.1.12. Itinéraires

Il est très facile de dessiner un itinéraire sur OSM. On donne le `GeoPoint` de départ et celui d'arrivée dans une liste, éventuellement des étapes intermédiaires : 

```
RoadManager manager = new OSRMRoadManager(this);
ArrayList<GeoPoint> etapes = new ArrayList<>();
etapes.add(gpGare);
etapes.add(gpIUT);
Road route = manager.getRoad(etapes);
if (road.mStatus != Road.STATUS_OK) Log.e(TAG, "pb serveur");
Polyline ligne =
    RoadManager.buildRoadOverlay(route, Color.BLUE, 4.0f);
mMap.getOverlays().add(0, ligne);
```

Seul problème : faire cela dans un `AsyncTask` ! (voir le [TP8](#))

### 8.1.13. Position GPS

Un dernier problème : comment lire les coordonnées fournies par le récepteur GPS ? Il faut faire appel au `LocationManager`. Ses méthodes retournent les coordonnées géographiques. 

```
LocationManager locationManager =
    (LocationManager) getSystemService(LOCATION_SERVICE);
Location position =
    locationManager.getLastKnownLocation(
        locationManager.GPS_PROVIDER);
if (position != null) {
    mapController.setCenter(new GeoPoint(position));
}
```


NB: ça ne marche qu'en plein air (réception GPS). Consulter aussi [cette page](#) à propos de l'utilisation du GPS et des réseaux.

### 8.1.14. Mise à jour en temps réel de la position

Si on veut suivre et afficher les mouvements : 

```
locationManager.requestLocationUpdates(
    locationManager.GPS_PROVIDER, 0, 0, this);
```

On peut utiliser la localisation par Wifi, mettre `NETWORK_PROVIDER`.

Le dernier paramètre est un écouteur, ici `this`. Il doit implémenter les méthodes de l'interface `LocationListener` dont : 

```
public void onLocationChanged(Location position)
{
    // déplacer le marqueur de l'utilisateur
    mrkUti.setPosition(new GeoPoint(position));
    // redessiner la carte
    mMap.invalidate();
}
```


### 8.1.15. Positions simulées

Pour tester une application basée sur le GPS sans se déplacer physiquement, il y a moyen d'envoyer de fausses positions avec Android Studio.

Il faut afficher la fenêtre **Android Device Monitor** par le menu **Tools**, item **Android**. Dans l'onglet **Emulator**, il y a un panneau pour définir la position de l'AVD, soit fixe, soit à l'aide d'un fichier GPX provenant d'un récepteur GPS de randonnée par exemple.

Cette fenêtre est également accessible avec le bouton ... en bas du panneau des outils de l'AVD.

### 8.1.16. Clics sur la carte

C'est le seul point un peu complexe. Il faut sous-classer la classe **Overlay** afin de récupérer les touches de l'écran. On doit seulement intercepter les clics longs pour ne pas gêner les mouvements sur la carte. Voici le début : 

```
public class LongPressMapOverlay extends Overlay
{
    @Override
    protected void draw(Canvas c, MapView m, boolean shadow)
    {}
}
```

Pour installer ce mécanisme, il faut rajouter ceci dans **onCreate** :

```
mMap.getOverlays().add(new LongPressMapOverlay());
```

### 8.1.17. Traitement des clics

Le cœur de la classe traite les clics longs en convertissant les coordonnées du clic en coordonnées géographiques : 


```
@Override
public boolean onLongPress(MotionEvent event, MapView map)
{
    if (event.getAction() == MotionEvent.ACTION_DOWN) {
        Projection projection = map.getProjection();
        GeoPoint position = (GeoPoint) projection.fromPixels(
            (int)event.getX(), (int)event.getY());
        // utiliser position ...
    }
}
```



```
}  
    return true;  
}
```

Par exemple, elle crée ou déplace un marqueur.


### 8.1.18. Autorisations

Pour finir, Il faut autoriser plusieurs choses dans le *Manifeste* : accès au GPS et au réseau, et écriture sur la carte mémoire : 

```
<uses-permission  
    android:name="android.permission.ACCESS_COARSE_LOCATION"/>  
<uses-permission  
    android:name="android.permission.ACCESS_FINE_LOCATION"/>  
<uses-permission  
    android:name="android.permission.ACCESS_WIFI_STATE" />  
<uses-permission  
    android:name="android.permission.ACCESS_NETWORK_STATE" />  
<uses-permission  
    android:name="android.permission.INTERNET" />  
<uses-permission  
    android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```


## 8.2. Dessin en 2D

### 8.2.1. Principes

Une application de dessin 2D définit une sous-classe de `View` et surcharge sa méthode `onDraw`, c'est elle qui est appelée pour dessiner la vue. Voici le squelette minimal : 

```
package fr.iutlan.dessin;  
public class DessinView extends View {  
    Paint mPeinture;  
    public DessinView(Context context, AttributeSet attrs) {  
        super(context, attrs);  
        mPeinture = new Paint();  
        mPeinture.setColor(Color.BLUE);  
    }  
    public void onDraw(Canvas canvas) {  
        canvas.drawCircle(100, 100, 50, mPeinture);  
    }  
}
```

### 8.2.2. Layout pour le dessin

Pour voir `DessinView`, il faut l'inclure dans un layout : 

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="..."
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <fr.iutlan.dessin.DessinView
        android:id="@+id/dessin"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

Il faut mettre le package et le nom de la classe en tant que balise XML. Ne pas oublier les attributs de taille.

### 8.2.3. Méthode `onDraw`

La méthode `onDraw(Canvas canvas)` doit effectuer tous les tracés. Cette méthode doit être rapide. Également, elle ne doit faire aucun `new`. Il faut donc créer tous les objets nécessaires auparavant, par exemple dans le constructeur de la vue.

Son paramètre `canvas` représente la zone de dessin. Attention, ce n'est pas un bitmap. Un `canvas` ne possède pas de pixels ; c'est le bitmap associé à la vue qui les possède. Voici comment on associe un `canvas` à un bitmap :

```
Bitmap bm =
    Bitmap.createBitmap(100, 100, Bitmap.Config.ARGB_8888);
Canvas canvas = new Canvas(bm);
```

C'est déjà fait pour le `canvas` fourni à la méthode `onDraw`. On obtient le bitmap de la vue avec `getDrawingCache()`.

### 8.2.4. Méthodes de la classe `Canvas`

La classe `Canvas` possède de nombreuses méthodes de dessin :

- `drawColor(int color)` : efface le `canvas` avec la couleur indiquée. Cette couleur est un code 32 bits retourné par la classe statique `Color` :
  - `Color.BLACK`, `Color.RED`... : couleurs prédéfinies,
  - `Color.rgb(int r, int v, int b)` : convertit des composantes RVB 0..255 en un code de couleur.
- `drawLine (float x1, float y1, float x2, float y2, Paint peinture)` : trace une ligne entre (x1,y1) et (x2,y2) avec la peinture
- `drawCircle (float cx, float cy, float rayon, Paint paint)` dessine un cercle.
- etc.

### 8.2.5. Peinture `Paint`

Cette classe permet de représenter les modes de dessin : couleurs de tracé, de remplissage, polices, lissage... C'est extrêmement riche. Voici un exemple d'utilisation :

```
mPeinture = new Paint(Paint.ANTI_ALIAS_FLAG);
mPeinture.setColor(Color.rgb(128, 255, 32));
mPeinture.setAlpha(192);
mPeinture.setStyle(Paint.Style.STROKE);
mPeinture.setStrokeWidth(10);
```


Il est préférable de créer les peintures dans le constructeur de la vue ou une autre méthode, mais surtout pas dans la méthode `onDraw`.

### 8.2.6. Quelques accesseurs de Paint

Parmi la liste de [ce qui existe](#), on peut citer :

- `setColor(Color)`, `setARGB(int a, int r, int v, int b)`, `setAlpha(int a)` : définissent la couleur et la transparence de la peinture,
- `setStyle(Paint.Style style)` : indique ce qu'il faut dessiner pour une forme telle qu'un rectangle ou un cercle :
  - `Paint.Style.STROKE` uniquement le contour
  - `Paint.Style.FILL` uniquement l'intérieur
  - `Paint.Style.FILL_AND_STROKE` contour et intérieur
- `setStrokeWidth(float pixels)` définit la largeur du contour.

### 8.2.7. Motifs

Il est possible de créer une peinture basée sur un motif. On part d'une image `motif.png` dans le dossier `res/drawable` qu'on emploie comme ceci : 

```
Bitmap bmMotif = BitmapFactory.decodeResource(
    context.getResources(), R.drawable.motif);
BitmapShader shaderMotif = new BitmapShader(bmMotif,
    Shader.TileMode.REPEAT, Shader.TileMode.REPEAT);
mPaintMotif = new Paint(Paint.ANTI_ALIAS_FLAG);
mPaintMotif.setShader(shaderMotif);
mPaintMotif.setStyle(Paint.Style.FILL_AND_STROKE);
```

Cette peinture fait appel à un *Shader*. C'est une classe permettant d'appliquer des effets progressifs, tels qu'un dégradé ou un motif comme ici (`BitmapShader`).

### 8.2.8. Shaders

Voici la réalisation d'un dégradé horizontal basé sur 3 couleurs : 

```
final int[] couleurs = new int[] {
    Color.rgb(128, 255, 32),      // vert pomme
    Color.rgb(255, 128, 32),      // orange
    Color.rgb(0, 0, 255)          // bleu
};
final float[] positions = new float[] { 0.0f, 0.5f, 1.0f };
```

```
Shader shader = new LinearGradient(0, 0, 100, 0,  
    couleurs, positions, Shader.TileMode.CLAMP);  
mPaintDegrade = new Paint(Paint.ANTI_ALIAS_FLAG);  
mPaintDegrade.setShader(shader);
```



Figure 46: Dégradé horizontal

### 8.2.9. Shaders, suite et fin

Le dégradé précédent est basé sur trois couleurs situées aux extrémités et au centre du rectangle. On fournit donc deux tableaux, l'un pour les couleurs et l'autre pour les positions des couleurs relativement au dégradé, de 0.0 à 1.0.

Le dégradé possède une dimension, 100 pixels de large. Si la figure à dessiner est plus large, la couleur sera maintenue constante avec l'option `CLAMP`. D'autres options permettent de faire un effet miroir, `MIRROR`, ou redémarrer au début `REPEAT`.

[Cette page](#) présente les shaders et filtres d'une manière extrêmement intéressante. Comme vous verrez, il y a un grand nombre de possibilités.

### 8.2.10. Quelques remarques

Lorsqu'il faut redessiner la vue, appelez `invalidate`. Si la demande de réaffichage est faite dans un autre *thread*, alors il doit appeler `postInvalidate`.

La technique montrée dans ce cours convient aux dessins relativement statiques, mais pas à un jeu par exemple. Pour mieux animer le dessin, il est recommandé de sous-classer `SurfaceView` plutôt que `View`. Les dessins sont alors faits dans un thread séparé et déclenchés par des événements.

### 8.2.11. « Dessinables »

Les canvas servent à dessiner des figures géométriques, rectangles, lignes, etc, mais aussi des `Drawable`, c'est à dire des « choses dessinables » telles que des images bitmap ou des formes quelconques. Il existe beaucoup de sous-classes de [Drawable](#).

Un `Drawable` est créé :

- par une image PNG ou JPG dans `res/drawable...`



```
Bitmap bm = BitmapFactory  
    .decodeResource(getResources(), R.drawable.image);  
Drawable d = new BitmapDrawable(getResources(), bm);
```

Android a défini une norme pour des images PNG étirables, les « 9patch ».

### 8.2.12. Images PNG étirables 9patch

Il s'agit d'images PNG nommées en `.9.png` qui peuvent être dessinées de différentes tailles. À gauche, l'image d'origine et à droite, 3 exemplaires étirés.

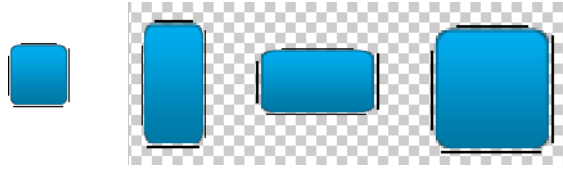



Figure 47: Image étirable

Une image « 9patch » est bordée sur ses 4 côtés par des lignes noires qui spécifient les zones étirables en haut et à gauche, et les zones qui peuvent être occupées par du texte à droite et en bas.

Il faut utiliser l'outil `draw9patch` pour les éditer. Ça demande un peu de savoir-faire.

### 8.2.13. Drawable, suite

- Un drawable peut également provenir d'une forme vectorielle dans un fichier XML. Ex : `res/drawable/carre.xml` : 

```
<?xml version="1.0" encoding="UTF-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <stroke android:width="4dp" android:color="#F000" />
    <gradient android:angle="90"
        android:startColor="#FFBB"
        android:endColor="#F77B" />
    <corners android:radius="16dp" />
</shape>
```



Figure 48: Dessin vectoriel XML

### 8.2.14. Variantes

Android permet de créer des « dessinables » à variantes par exemple pour des boutons personnalisés.



```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="...">


    <item android:drawable="@drawable/button_pressed"
        android:state_pressed="true" />
```

```
<item android:drawable="@drawable/button_checked"
      android:state_checked="true" />
<item android:drawable="@drawable/button_default" />

</selector>
```

L'une ou l'autre des images sera choisie en fonction de l'état du bouton, enfoncé, relâché, inactif.

### 8.2.15. Utilisation d'un Drawable


Ces objets dessinable peuvent être employés dans un canvas. Puisque ce sont des objets vectoriels, il faut définir les coordonnées des coins haut-gauche et bas-droit, ce qui permet d'étirer la figure. Les tailles qui sont indiquées dans le xml sont pourtant absolues. 

```
Drawable drw = getResources().getDrawable(R.drawable.carre);
drw.setBounds(x1, y1, x2, y2); // coins
drw.draw(canvas);
```

Remarquez le petit piège de la dernière instruction, on passe le canvas en paramètre à la méthode draw du drawable.

NB: la première instruction est à placer dans le constructeur de la vue, afin de ne pas ralentir la fonction de dessin.

### 8.2.16. Enregistrer un dessin dans un fichier

C'est très facile. Il suffit de récupérer le bitmap associé à la vue, puis de le compresser en PNG. 

```
public void save(String filename)
{
    Bitmap bitmap = getDrawingCache();
    try {
        FileOutputStream out = new FileOutputStream(filename);
        bitmap.compress(Bitmap.CompressFormat.PNG, 90, out);
        out.close();
    } catch (Exception e) {
        ...
    }
}
```

### 8.2.17. Coordonnées dans un canvas

Un dernier mot sur les canvas. Il y a tout un mécanisme permettant de modifier les coordonnées dans un canvas :

- déplacer l'origine avec `translate(dx,dy)` : toutes les coordonnées fournies ultérieurement seront additionnées à (dx,dy)
- multiplier les coordonnées par sx,sy avec `scale(sx,sy)`


- pivoter les coordonnées autour de (px,py) d'un angle  $a^\circ$  avec `rotate(a, px, py)`

En fait, il y a un mécanisme de transformations matricielles 2D appliquées aux coordonnées, ainsi qu'une pile permettant de sauver la transformation actuelle ou la restituer.

- `save()` : enregistre la matrice actuelle
- `restore()` : restitue la matrice avec celle qui avait été sauvée

## 8.3. Interactions avec l'utilisateur


### 8.3.1. Écouteurs pour les touches de l'écran

Il existe beaucoup d'écouteurs pour les actions de l'utilisateur sur une zone de dessin. Parmi elles, on doit connaître `onTouchEvent`. Son paramètre indique la nature de l'action (toucher, mouvement...) ainsi que les coordonnées. 

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    float x = event.getX();
    float y = event.getY();

    switch (event.getAction()) {
        case MotionEvent.ACTION_MOVE:
            ...
            break;
    }
    return true;
}
```

### 8.3.2. Modèle de gestion des actions

Souvent il faut distinguer le premier toucher (ex: création d'une figure) des mouvements suivants (ex: taille de la figure). 

```
switch (event.getAction()) {
    case MotionEvent.ACTION_DOWN:
        figure = Figure.creer(typefigure, color);
        figure.setReference(x, y);
        figures.add(figure);
        break;
    case MotionEvent.ACTION_MOVE:
        if (figures.size() < 1) return true;
        figure = figures.getLast();
        figure.setCoin(x,y);
        break;
}
invalidate();
```

### 8.3.3. Automate pour gérer les actions

L'algo précédent peut se représenter à l'aide d'un automate de Mealy à deux états : repos et en cours d'édition d'une figure. Les changements d'états sont déclenchés par les actions utilisateur et effectuent un traitement.

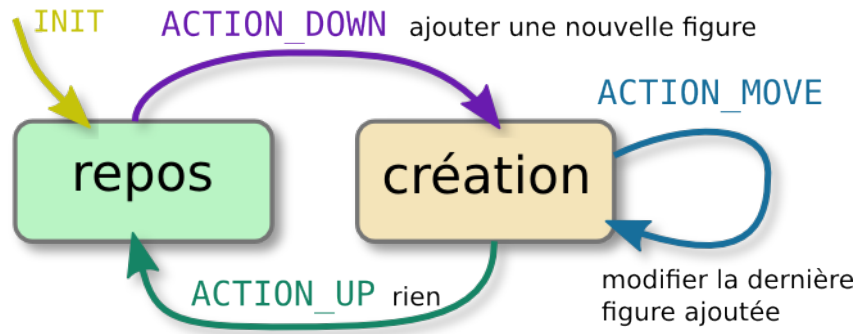


Figure 49: Automate

## 8.4. Boîtes de dialogue spécifiques

### 8.4.1. Sélecteur de couleur

Android ne propose pas de sélecteur de couleur, alors il faut le construire soi-même.



Figure 50: Sélecteur de couleur

### 8.4.2. Version simple

En TP, on va construire une version simplifiée afin de comprendre le principe :

### 8.4.3. Concepts

Plusieurs concepts interviennent dans ce sélecteur de couleur :

- La fenêtre dérive de `DialogFragment`, elle affiche un dialogue de type `AlertDialog` avec des boutons Ok et Annuler,



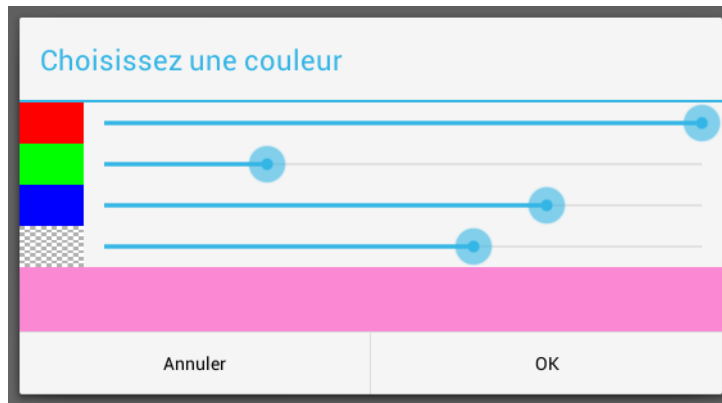


Figure 51: Sélecteur de couleur simple

- Cet `AlertDialog` contient une vue personnalisée contenant des `SeekBar` pour régler les composantes de couleur,
- Les `SeekBar` du layout ont des *callbacks* qui mettent à jour la couleur choisie en temps réel,
- Le bouton Valider du `AlertDialog` déclenche un écouteur dans l'activité qui a appelé le sélecteur.

#### 8.4.4. Fragment de dialogue

Le fragment de dialogue doit définir plusieurs choses :

- C'est une sous-classe de `DialogFragment`

```
public class ColorPickerDialog extends DialogFragment
```

- Il définit une interface pour un écouteur qu'il appellera à la fin :

```
public interface OnColorChangedListener {  
    void colorChanged(int color);  
}
```

- Une méthode `onCreateDialog` retourne un `AlertDialog` pour bénéficier des boutons *ok* et *annuler*. Le bouton *ok* est associé à une *callback* qui active l'écouteur en lui fournissant la couleur.


#### 8.4.5. Méthode `onCreateDialog`



```
public Dialog onCreateDialog(Bundle savedInstanceState) {  
    Context ctx = getActivity();  
    Builder builder = new AlertDialog.Builder(ctx);  
    builder.setTitle("Choisissez une couleur");  
    final ColorPickerView cpv = new ColorPickerView(ctx);  
    builder.setView(cpv);  
    builder.setPositiveButton(android.R.string.yes,  
        new DialogInterface.OnClickListener() {  
            public void onClick(DialogInterface dialog, int btn) {  
                // prévenir l'écouteur  
            }  
        })  
}
```

```
        mListener.colorChanged(cpv.getColor());
    }
});
builder.setNegativeButton(android.R.string.no, null);
return builder.create();
}
```

#### 8.4.6. Vue personnalisée dans le dialogue

Voici la définition de la classe ColorPickerView qui est à l'intérieur du dialogue d'alerte. Elle gère quatre curseurs et une couleur : 

```
private static class ColorPickerView extends LinearLayout {
    // couleur définie par les curseurs
    private int mColor;
    // constructeur
    ColorPickerView(Context context) {
        // constructeur de la superclasse
        super(context);
        // mettre en place le layout
        inflate(getContext(), R.layout.colorpickerdialog, this);
        ...
    }
}
```


#### 8.4.7. Layout de cette vue

Le layout colorpickerdialog.xml contient quatre SeekBar, rouge, vert, bleu et alpha. Ils ont une callback comme celle-ci : 

```
SeekBar sbRouge = (SeekBar) findViewById(R.id.sbRouge);
sbRouge.setOnSeekBarChangeListener(
    new OnSeekBarChangeListener() {
        public void onProgressChanged(SearchBar seekBar,
            int progress, boolean fromUser) {
            mColor = Color.argb(
                Color.alpha(mColor), progress,
                Color.green(mColor), Color.blue(mColor));
        }
    });
```

Elle change seulement la composante rouge de la variable mColor. Il y a les mêmes choses pour le vert, le bleu et la transparence.

#### 8.4.8. Utilisation du dialogue

Pour finir, voici comment on affiche ce dialogue, par exemple dans un menu : 

```
new ColorPickerDialog(  
    new ColorPickerDialog.OnColorChangeListener() {  
        @Override  
        public void colorChanged(int color) {  
            // utiliser la couleur ....  
        }  
    }  
).show(getFragmentManager(), "fragment_colorpicker");
```

Cela consiste à définir un écouteur qui reçoit la nouvelle couleur du sélecteur. L'écouteur peut la transmettre à la classe qui dessine une nouvelle figure.

#### 8.4.9. Sélecteur de fichier

Dans le même genre mais nettement trop complexe, il y a le sélecteur de fichiers pour enregistrer un dessin.

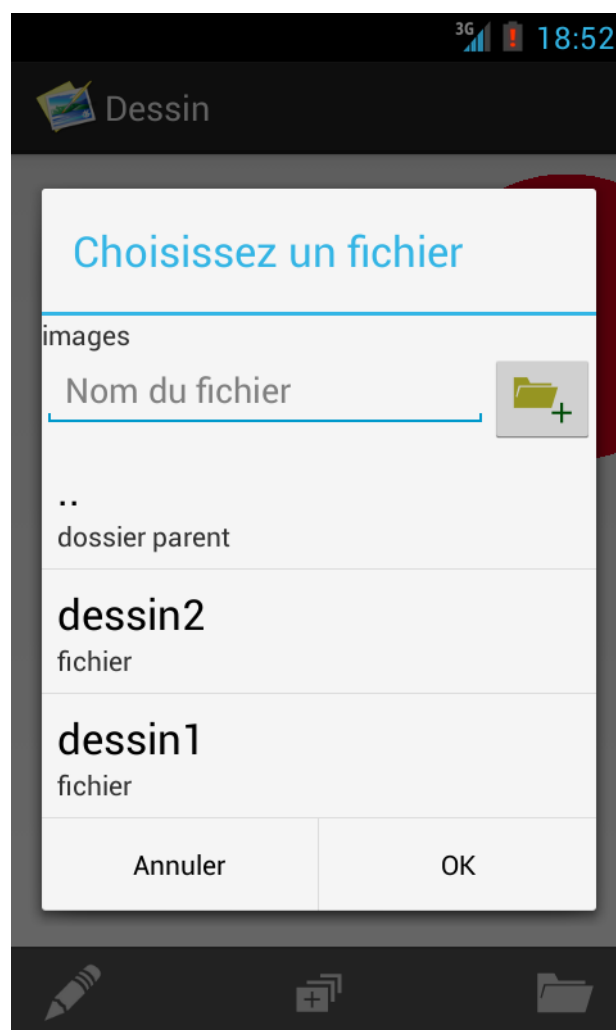


Figure 52: Sélecteur de fichier

#### **8.4.10. C'est la fin**

C'est fini, nous avons étudié tout ce qu'il était raisonnable de faire en 8 semaines.