

Programmation mobile avec Android

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

février-mars 2021

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 25/01/2021 à 08:57

Table des matières

1	Environnement de développement	18
1.1	Introduction	18
1.1.1	Qu'est-ce qu'Android ?	18
1.1.2	Historique	18
1.1.3	Remarque sur les versions d'API	20
1.1.4	Distribution des versions	20
1.1.5	Remarques diverses	20
1.1.6	Programmation d'applications	21
1.1.7	Applications natives	21
1.1.8	Kotlin	21
1.1.9	Exemple : objet pouvant être null	21
1.1.10	Pas de Kotlin pour ce cours	22
1.2	SDK Android et Android Studio	22
1.2.1	SDK et Android Studio	22
1.2.2	Android Studio	23

1.2.3	SDK Manager	23
1.2.4	Choix des éléments du SDK	23
1.2.5	Dossiers du SDK	23
1.3	Première application	25
1.3.1	Objectif de la semaine 1	25
1.3.2	Assistant de création d'application	25
1.3.3	Modèle d'application	26
1.3.4	Nom, package et version	26
1.3.5	Résultat de l'assistant	26
1.3.6	Fenêtre du projet	28
1.3.7	Éditeurs spécifiques	28
1.3.8	Exemple <code>res/values/strings.xml</code>	28
1.3.9	Exemple <code>res/layout/main.xml</code>	28
1.3.10	Source XML sous-jacent	28
1.3.11	Reconstruction du projet	31
1.3.12	Gradle	31
1.3.13	Structure d'un projet AndroidStudio	31
1.3.14	Utilisation de bibliothèques	31
1.3.15	Mises à jour	32
1.4	Première exécution	33
1.4.1	Exécution de l'application	33
1.4.2	Assistant de création d'une tablette virtuelle	33
1.4.3	Caractéristiques d'un AVD	33
1.4.4	Lancement d'une application	35
1.4.5	Application sur l'AVD	35
1.5	Communication AVD - Android Studio	35
1.5.1	Fenêtres Android	35
1.5.2	Fenêtre <code>Logcat</code>	35
1.5.3	Filtrage des messages	35
1.5.4	Émission d'un message vers <code>LogCat</code>	35
1.5.5	Logiciel ADB	37
1.5.6	Mode d'emploi de ADB	37
1.5.7	Système de fichiers Android	38
1.6	Création d'un paquet installable	38

1.6.1	Paquet	38
1.6.2	Signature d'une application	39
1.6.3	Création du <i>keystore</i>	39
1.6.4	Création d'une clé	39
1.6.5	Création du paquet	39
1.6.6	Et voilà	41
2	Création d'interfaces utilisateur	42
2.1	Présentation rapide des concepts	42
2.1.1	Composition d'une application	42
2.1.2	Structure d'une interface utilisateur	42
2.1.3	Création d'une interface	43
2.1.4	Création d'un écran	43
2.2	Ressources	43
2.2.1	Définition	43
2.2.2	Identifiant de ressource	43
2.2.3	Génération de la classe R	44
2.2.4	La classe R	44
2.2.5	Rappel sur la structure d'un fichier XML	44
2.2.6	Espaces de nommage dans un fichier XML	45
2.2.7	Ressources de type chaînes	45
2.2.8	Traduction des chaînes (<i>localisation</i>)	45
2.2.9	Emploi des ressources texte dans un programme	46
2.2.10	Emploi des ressources texte dans une interface	46
2.2.11	Images : <code>R.drawable.nom</code>	47
2.2.12	Tableau de chaînes : <code>R.array.nom</code>	47
2.2.13	Autres	47
2.3	Mise en page (<i>layouts</i>)	48
2.3.1	Structure d'une interface Android	48
2.3.2	Arbre des vues	48
2.3.3	Création d'une interface par programme	48
2.3.4	Ressources de type <i>layout</i>	49
2.3.5	Identifiants et vues	49
2.3.6	@id/nom ou @+id/nom ?	49

2.3.7	Paramètres de positionnement	50
2.3.8	Paramètres obligatoires	50
2.3.9	Autres paramètres géométriques	51
2.3.10	Marges et remplissage	51
2.3.11	Groupe de vues <code>LinearLayout</code>	51
2.3.12	Pondération des tailles	52
2.3.13	Exemple de poids différents	52
2.3.14	Groupe de vues <code>TableLayout</code>	52
2.3.15	Largeur des colonnes d'un <code>TableLayout</code>	53
2.3.16	Groupe de vues <code>RelativeLayout</code>	53
2.3.17	Utilisation d'un <code>RelativeLayout</code>	53
2.3.18	Autres groupements	54
2.4	Composants d'interface	54
2.4.1	Vues	54
2.4.2	<code>TextView</code>	54
2.4.3	<code>Button</code>	55
2.4.4	Bascules	55
2.4.5	<code>EditText</code>	55
2.4.6	Autres vues	55
2.4.7	C'est tout	56

3 Vie d'une application 57

3.1	Applications et activités	57
3.1.1	Composition d'une application	57
3.1.2	Déclaration d'une application	57
3.1.3	Démarrage d'une application	58
3.1.4	Démarrage d'une activité et <code>Intents</code>	58
3.1.5	Lancement d'une activité par programme	58
3.1.6	Lancement d'une application Android	58
3.1.7	Lancement d'une activité d'une autre application	59
3.1.8	Autorisations d'une application	59
3.1.9	Sécurité des applications (pour info)	59
3.2	Applications	60
3.2.1	Fonctionnement d'une application	60

3.2.2	Navigation entre activités	60
3.2.3	Lancement avec ou sans retour	60
3.2.4	Lancement avec attente de résultat	60
3.2.5	Terminaison d'une activité	62
3.2.6	Méthode <code>onActivityResult</code>	62
3.2.7	Transport d'informations dans un <code>Intent</code>	63
3.2.8	Extraction d'informations d'un <code>Intent</code>	63
3.2.9	Contexte d'application	63
3.2.10	Définition d'un contexte d'application	63
3.2.11	Définition d'un contexte d'application, fin	64
3.3	Activités	64
3.3.1	Présentation	64
3.3.2	Cycle de vie d'une activité	65
3.3.3	Événements de changement d'état	65
3.3.4	Squelette d'activité	65
3.3.5	Terminaison d'une activité	66
3.3.6	Pause d'une activité	66
3.3.7	Arrêt d'une activité	66
3.3.8	Enregistrement de valeurs d'une exécution à l'autre	67
3.3.9	Restaurer l'état au lancement	67
3.4	Vues et activités	67
3.4.1	Obtention des vues	67
3.4.2	Propriétés des vues	68
3.4.3	Actions de l'utilisateur	68
3.4.4	Définition d'un écouteur	69
3.4.5	Écouteur privé anonyme	69
3.4.6	Écouteur privé	69
3.4.7	L'activité elle-même en tant qu'écouteur	70
3.4.8	Distinction des émetteurs	70
3.4.9	Événements des vues courantes	70
3.4.10	C'est fini pour aujourd'hui	71

4	Application liste	72
4.1	Présentation	72
4.1.1	Principe général	72
4.1.2	Schéma global	73
4.1.3	Une classe pour représenter les items	73
4.1.4	Données initiales	74
4.1.5	Copie dans un <code>ArrayList</code>	74
4.1.6	Rappels sur le container <code>ArrayList<type></code>	74
4.1.7	Données initiales dans les ressources	75
4.1.8	Remarques	75
4.2	Affichage de la liste	76
4.2.1	Activité	76
4.2.2	Mise en œuvre	76
4.2.3	Layout de l'activité pour afficher une liste	76
4.2.4	Mise en place du layout d'activité	76
4.3	Adaptateurs	77
4.3.1	Relations entre la vue et les données	77
4.3.2	Rôle d'un adaptateur	77
4.3.3	Adaptateurs prédéfinis	78
4.3.4	<code>ArrayAdapter<Type></code> pour les listes	78
4.3.5	Exemple d'emploi	78
4.3.6	Affichage avec une <code>ListActivity</code>	78
4.3.7	Layout pour un item	79
4.3.8	Autre layouts	79
4.3.9	Layouts prédéfinis	80
4.3.10	Exemple avec les layouts prédéfinis	80
4.4	Adaptateur personnalisé	80
4.4.1	Classe Adapter personnalisée	80
4.4.2	Réutilisation d'une vue	81
4.4.3	Méthode <code>getView</code> personnalisée	81
4.4.4	Méthode <code>PlaneteView.create</code>	81
4.4.5	Layout d'item <code>res/layout/item.xml</code>	82
4.4.6	Classe personnalisée dans les ressources	82
4.4.7	Classe <code>PlaneteView</code> pour afficher les items	82

4.4.8	Définition de la classe <code>PlaneteView</code>	83
4.4.9	Créer des vues à partir d'un layout XML	83
4.4.10	Méthode <code>PlaneteView.create</code>	83
4.4.11	Méthode <code>findViews</code>	84
4.4.12	Pour finir, la méthode <code>PlaneteView.setItem</code>	84
4.4.13	Récapitulatif	84
4.4.14	Le résultat	85
4.5	Actions utilisateur sur la liste	85
4.5.1	Modification des données	85
4.5.2	Clic sur un élément	86
4.5.3	Clic sur un élément, fin	86
4.5.4	Liste d'éléments cochables	87
4.5.5	Liste cochable simple	87
4.5.6	Liste à choix multiples	88
4.5.7	Liste cochable personnalisée	88
4.6	<code>RecyclerView</code>	88
4.6.1	Présentation	88
4.6.2	Principes	88
4.6.3	Méthode <code>onCreate</code> de l'activité	89
4.6.4	Disposition des éléments	89
4.6.5	Mise en grille	89
4.6.6	Séparateur entre items	90
4.6.7	Adaptateur de <code>RecyclerView</code>	90
4.6.8	Affichage d'un item <code>PlaneteView</code>	91
4.7	Réponses aux sélections d'items	92
4.7.1	Clics sur un <code>RecyclerView</code>	92
4.7.2	Afficher la sélection	92
4.7.3	Ouf, c'est fini	93

5 Ergonomie 94

5.1	Barre d'action et menus	94
5.1.1	Barre d'action	94
5.1.2	Réalisation d'un menu	94
5.1.3	Spécification d'un menu	95

5.1.4	Icônes pour les menus	95
5.1.5	Écouteurs pour les menus	95
5.1.6	Réactions aux sélections d'items	96
5.1.7	Menus en cascade	96
5.1.8	Menus contextuels	97
5.1.9	Associer un menu contextuel à une vue	97
5.1.10	Callback d'affichage du menu	97
5.1.11	Callback des items du menu	98
5.1.12	Menu contextuel pour un RecyclerView	98
5.1.13	Parenthèse sur les <i>lambda</i> Java	99
5.1.14	Parenthèse sur les <i>lambda</i> Java, fin	99
5.1.15	Menu contextuel pour un RecyclerView , fin	99
5.2	Annonces et dialogues	100
5.2.1	Annonces : <i>toasts</i>	100
5.2.2	Annonces personnalisées	100
5.2.3	Dialogues	101
5.2.4	Dialogue d'alerte	101
5.2.5	Boutons et affichage d'un dialogue d'alerte	101
5.2.6	Autres types de dialogues d'alerte	102
5.2.7	Dialogues personnalisés	102
5.2.8	Création d'un dialogue	102
5.2.9	Affichage du dialogue	103
5.3	Fragments et activités	103
5.3.1	Fragments	103
5.3.2	Tablettes, smartphones...	103
5.3.3	Structure d'un fragment	104
5.3.4	Différents types de fragments	104
5.3.5	Cycle de vie des fragments	105
5.3.6	ListFragment	105
5.3.7	Menus de fragments	106
5.3.8	Intégrer un fragment dans une activité	106
5.3.9	Fragments statiques dans une activité	106
5.3.10	FragmentManager	107
5.3.11	Attribution d'un fragment dynamiquement	107

5.3.12	Disposition selon la géométrie de l'écran	107
5.3.13	Changer la disposition selon la géométrie	108
5.3.14	Deux dispositions possibles	108
5.3.15	Communication entre Activité et Fragments	109
5.3.16	Interface pour un écouteur	109
5.3.17	Écouteur du fragment	109
5.3.18	Écouteur de l'activité	110
5.3.19	Relation entre deux classes à méditer, partie 1	110
5.3.20	À méditer, partie 2	111
5.4	Préférences d'application	111
5.4.1	Illustration	111
5.4.2	Présentation	111
5.4.3	Définition des préférences	112
5.4.4	Explications	112
5.4.5	Accès aux préférences	112
5.4.6	Préférences chaînes et nombres	113
5.4.7	Modification des préférences par programme	113
5.4.8	Affichage des préférences	113
5.4.9	Fragment pour les préférences	114
5.5	Bibliothèque support	114
5.5.1	Compatibilité des applications	114
5.5.2	Compatibilité des versions Android	114
5.5.3	Bibliothèque support	115
5.5.4	Anciennes versions de l'Android Support Library	115
5.5.5	Une seule pour les gouverner toutes	115
5.5.6	Une seule pour les gouverner toutes, fin	115
5.5.7	Mode d'emploi	116
5.5.8	Programmation	116
5.5.9	Il est temps de faire une pause	116
6	Realm	117
6.1	Plugin Lombok	117
6.1.1	Présentation	117
6.1.2	Exemple	117

6.1.3	Placement des annotations	118
6.1.4	Nommage des champs	118
6.1.5	Installation du plugin	118
6.2	Realm	119
6.2.1	Définition de Realm	119
6.2.2	Realm contre les autres ORM	119
6.2.3	Configuration d'un projet Android avec Realm	119
6.2.4	Initialisation d'un Realm par l'application	119
6.2.5	Ouverture d'un Realm dans chaque activité	120
6.2.6	Fermeture du Realm	120
6.2.7	Autres modes d'ouverture du Realm	121
6.3	Modèles de données Realm	121
6.3.1	Définir une table	121
6.3.2	Table Realm et Lombok	121
6.3.3	Types des colonnes	122
6.3.4	Empêcher les valeurs <code>null</code>	122
6.3.5	Définir une clé primaire	122
6.3.6	Définir une relation simple	122
6.3.7	Relation multiple	123
6.3.8	Migration des données	123
6.4	Création de n-uplets	123
6.4.1	Résumé	123
6.4.2	Création de n-uplets par <code>createObject</code>	123
6.4.3	Création de n-uplets par <code>new</code>	124
6.4.4	Modification d'un n-uplet	124
6.4.5	Suppression de n-uplets	124
6.5	Requêtes sur la base	125
6.5.1	Résumé	125
6.5.2	Sélections	125
6.5.3	Conditions simples	125
6.5.4	Nommage des colonnes	126
6.5.5	Librairie <i>RealmFieldNamesHelper</i>	126
6.5.6	Conjonctions de conditions	126
6.5.7	Disjonctions de conditions	127

6.5.8	Négations	127
6.5.9	Classement des données	127
6.5.10	Agrégation des résultats	128
6.5.11	Jointures 1-1	128
6.5.12	Jointures 1-N	128
6.5.13	Jointures inverses	129
6.5.14	Jointures inverses, explications	129
6.5.15	Jointures inverses, exemple	129
6.5.16	Suppression par une requête	129
6.6	Requêtes et adaptateurs de listes	130
6.6.1	Adaptateur Realm pour un <code>RecyclerView</code>	130
6.6.2	Adaptateur Realm, fin	131
6.6.3	Mise en place de la liste	131
6.6.4	Réponses aux clics sur la liste	131
6.6.5	C'est la fin	131

7 Dessin 2D interactif 132

7.1	Dessin en 2D	132
7.1.1	But	132
7.1.2	Principes	132
7.1.3	Layout pour le dessin	133
7.1.4	Méthode <code>onDraw</code>	133
7.1.5	Méthodes de la classe <code>Canvas</code>	134
7.1.6	Peinture <code>Paint</code>	134
7.1.7	Quelques accesseurs de <code>Paint</code>	134
7.1.8	Motifs	134
7.1.9	Shaders	135
7.1.10	Quelques remarques	135
7.1.11	« Dessinables »	136
7.1.12	Images PNG étirables 9patch	136
7.1.13	Variantes	137
7.1.14	Utilisation d'un <code>Drawable</code>	137
7.1.15	Enregistrer un dessin dans un fichier	137
7.1.16	Coordonnées dans un canvas	138

7.2	Interactions avec l'utilisateur	138
7.2.1	Écouteurs pour les touchers de l'écran	138
7.2.2	Modèle de gestion des actions	138
7.2.3	Automate pour gérer les actions	139
7.2.4	Programmation d'un automate	139
7.3	Boîtes de dialogue spécifiques	140
7.3.1	Sélecteur de couleur	140
7.3.2	Version simple	140
7.3.3	Concepts	141
7.3.4	Fragment de dialogue	141
7.3.5	Méthode <code>onCreateDialog</code>	141
7.3.6	Vue personnalisée dans le dialogue	142
7.3.7	Layout de cette vue	142
7.3.8	Écouteurs	143
7.3.9	Utilisation du dialogue	143
7.3.10	Sélecteur de fichier	143
8	Test logiciel	145
8.1	Introduction	145
8.1.1	Principe de base	145
8.1.2	Limitations	145
8.1.3	Précision des nombres	146
8.1.4	Généralisation des tests	146
8.2	Tests unitaires	146
8.2.1	Programmation des tests unitaires	146
8.2.2	Exécution des tests unitaires	147
8.2.3	JUnit4	147
8.2.4	Explications	148
8.2.5	<code>import static</code> en Java	148
8.2.6	Assertions JUnit	148
8.2.7	Affichage d'un message d'erreur	149
8.2.8	Vérification des exceptions	149
8.2.9	Vérification de la durée d'exécution	149
8.3	Assertions complexes avec Hamcrest	150

8.3.1	Assertions Hamcrest	150
8.3.2	Catalogue des correspondants Hamcrest	150
8.3.3	Importation de Hamcrest	151
8.4	Patron Arrange-Act-Assert	152
8.4.1	Organisation des tests	152
8.4.2	Préparation des données	152
8.4.3	Préparation des données avant chaque test	152
8.4.4	Préparation des données avant l'ensemble des tests	153
8.4.5	Clôture de tests	153
8.4.6	Vérification des assertions	154
8.4.7	Tests paramétrés	154
8.4.8	Exemple de test paramétré	154
8.4.9	Fourniture des paramètres	154
8.4.10	Importation de JUnitParams	155
8.5	Tests d'intégration	155
8.5.1	Introduction	155
8.5.2	Interface à la place d'une classe	155
8.5.3	Simulation d'une interface avec Mockito	156
8.5.4	Apprentissage de résultats	156
8.5.5	Apprentissage généralisé	156
8.5.6	<i>Matchers</i> pour Mockito	156
8.5.7	Autre syntaxe	157
8.5.8	Simulation pour une autre classe	157
8.5.9	Surveillance d'une classe	158
8.5.10	Surveillance d'une activité Android	158
8.5.11	Espionnage et simulation	158
8.5.12	Liaison des vues à l'activité	158
8.5.13	Test d'appel	159
8.5.14	Installation de Mockito	159
8.6	Tests sur AVD	160
8.6.1	Définition	160
8.6.2	Correspondants de vues	160
8.6.3	<i>ViewMatchers</i> d'Expresso	160
8.6.4	<i>ViewActions</i> d'Expresso	160

8.6.5	Tests sur des listes	161
8.6.6	Test sur un spinner	161
8.6.7	Installation de Espresso	161
8.6.8	Classe de test	161
8.6.9	Manipulations directes de l'activité	162
8.6.10	C'est la fin du cours et du module	162
9	Capteurs	163
9.1	Réalité augmentée	163
9.1.1	Définition	163
9.1.2	Applications	163
9.1.3	Principes	163
9.1.4	Réalité augmentée dans Android	164
9.2	Permissions Android	164
9.2.1	Concepts	164
9.2.2	Permissions dans le manifeste	164
9.2.3	Raffinement de certaines permissions	164
9.2.4	Demandes de permissions à la volée	165
9.2.5	Test d'une autorisation	165
9.2.6	Demande d'une autorisation	165
9.2.7	Préférences d'application	165
9.2.8	Dialogue de demande de droits	166
9.2.9	Affichage du dialogue	166
9.2.10	Justification des droits	166
9.3	Capteurs de position	167
9.3.1	Présentation	167
9.3.2	Utilisation dans Android	167
9.3.3	Récupération de la position	167
9.3.4	Abonnement aux changements de position	168
9.3.5	Événements de position	168
9.3.6	Remarques	168
9.4	Caméra	169
9.4.1	Présentation	169
9.4.2	Vue <code>SurfaceView</code>	169

9.4.3	Fonctionnement du <code>SurfaceView</code>	169
9.4.4	Événements d'un <code>SurfaceHolder</code>	169
9.4.5	Écouteur <code>surfaceCreated</code>	170
9.4.6	Écouteur <code>surfaceCreated</code> , fin	170
9.4.7	Écouteur <code>surfaceChanged</code>	170
9.4.8	Choix de la prévisualisation	170
9.4.9	Suite de <code>surfaceChanged</code>	171
9.4.10	Orientation de la caméra	171
9.4.11	Orientation de l'écran	171
9.4.12	Fin de <code>surfaceChanged</code>	172
9.4.13	Écouteur <code>onResume</code>	172
9.4.14	Écouteur <code>onPause</code>	172
9.4.15	Écouteur <code>surfaceDestroyed</code>	173
9.4.16	Organisation logicielle	173
9.5	Capteurs d'orientation	173
9.5.1	Présentation	173
9.5.2	Angles d'Euler	174
9.5.3	Matrice de rotation	174
9.5.4	Accès au gestionnaire	174
9.5.5	Accès aux capteurs	175
9.5.6	Abonnement aux mesures	175
9.5.7	Réception des mesures	175
9.5.8	Atténuation des oscillations	176
9.5.9	Orientation avec <code>TYPE_ROTATION_VECTOR</code>	176
9.5.10	Orientation sans <code>TYPE_ROTATION_VECTOR</code>	176
9.5.11	Orientation sans <code>TYPE_ROTATION_VECTOR</code> , fin	177
9.5.12	Orientation avec <code>TYPE_ORIENTATION</code>	177
9.6	Réalité augmentée	177
9.6.1	Objectif	177
9.6.2	Assemblage	177
9.6.3	Transformation des coordonnées	178
9.6.4	Transformation des coordonnées, fin	178
9.6.5	Dessin du POI	178

10	Dessin 2D interactif et Cartes	179
10.1	AsyncTasks	179
10.1.1	Présentation	179
10.1.2	Tâches asynchrones	179
10.1.3	Principe d'utilisation d'une AsyncTask	180
10.1.4	Structure d'une AsyncTask	180
10.1.5	Autres méthodes d'une AsyncTask	180
10.1.6	Paramètres d'une AsyncTask	180
10.1.7	Exemple de paramétrage	180
10.1.8	Paramètres variables	181
10.1.9	Définition d'une AsyncTask	181
10.1.10	Lancement d'une AsyncTask	182
10.1.11	Schéma récapitulatif	182
10.1.12	<code>execute</code> ne retourne rien	182
10.1.13	Récupération du résultat d'un AsyncTask	183
10.1.14	Simplification	183
10.1.15	Fuite de mémoire	183
10.1.16	Recommandations	184
10.1.17	Autres tâches asynchrones	184
10.2	OpenStreetMap	184
10.2.1	Présentation	184
10.2.2	Documentation	184
10.2.3	Pour commencer	186
10.2.4	Layout pour une carte OSM	186
10.2.5	Activité pour une carte OSM	186
10.2.6	Positionnement de la vue	187
10.2.7	Calques	187
10.2.8	Mise à jour de la carte	187
10.2.9	Marqueurs	187
10.2.10	Marqueur personnalisés	188
10.2.11	Réaction à un clic	188
10.2.12	Itinéraires	188
10.2.13	Position GPS	189
10.2.14	Mise à jour en temps réel de la position	189

10.2.15	Positions simulées	189
10.2.16	Clics sur la carte	189
10.2.17	Traitement des clics	190
10.2.18	Autorisations	190



Figure 1: Robot Android

Semaine 1

Environnement de développement

Cette matière présente la programmation d'applications natives sur Android.

Il y aura 8 semaines de cours, chacune comptant 1h CM et 2h TP.

Cette semaine nous allons découvrir 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. Qu'est-ce qu'Android ?

Android est une surcouche au dessus d'un système Linux : Voir la figure 2, page 19.

([URL de l'image originale](#))

1.1.2. Historique

- Né en 2004, racheté par Google en 2005, version 1.5 publiée en 2007
- De nombreuses versions depuis. On en est à la version 11 (sept. 2020) et l'API 30. La version 11 est le numéro pour le grand public, et les versions d'API sont pour les développeurs. Exemples :
 - 4.1 JellyBean = API 16,
 - 6.x Marshmallow = API 23,
 - 9.x Pie = API 28

Une API (*Application Programming Interface*) est un ensemble de bibliothèques de classes pour programmer des applications. Son numéro de version donne un indice de ses possibilités.

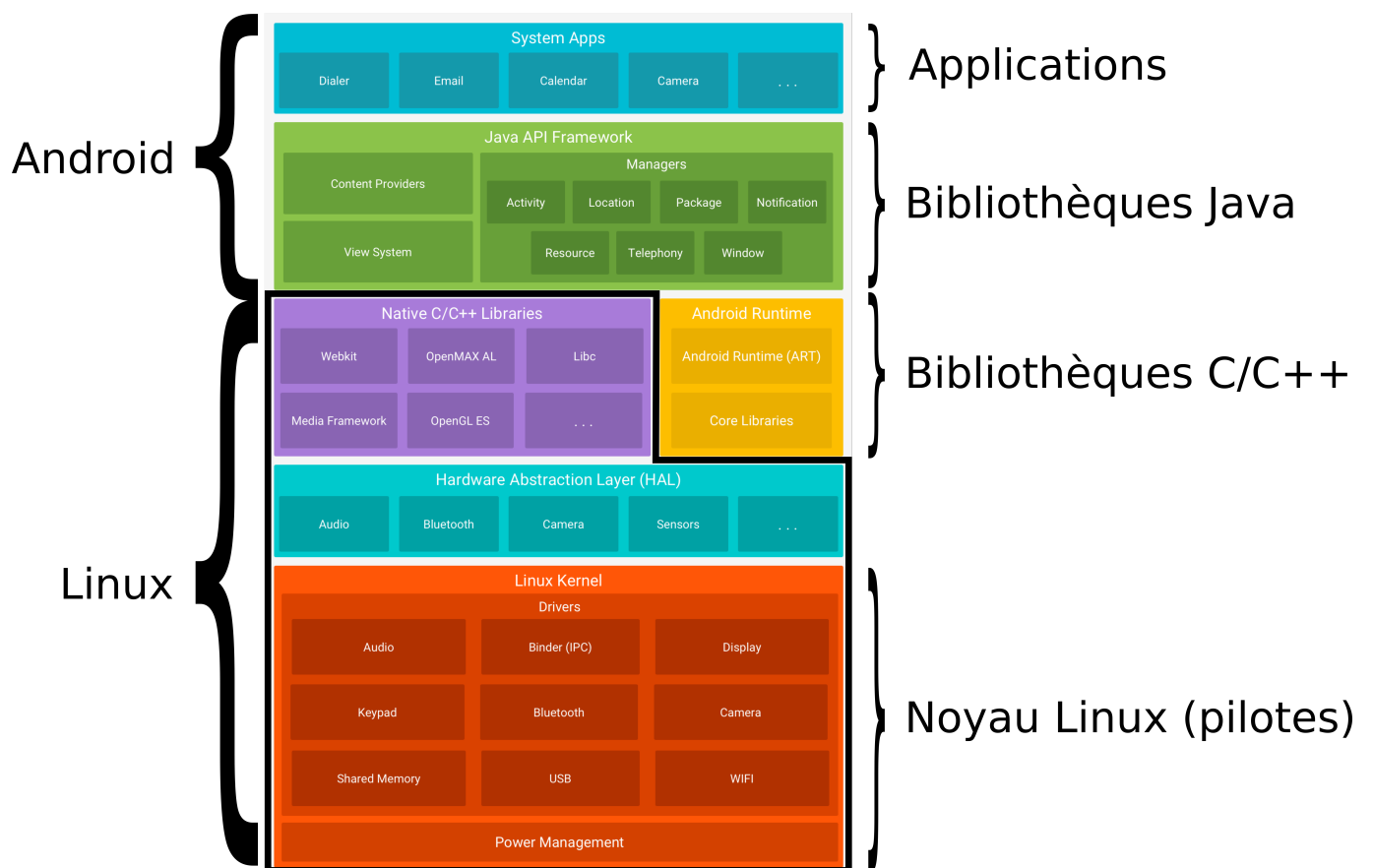


Figure 2: Constituants d'Android

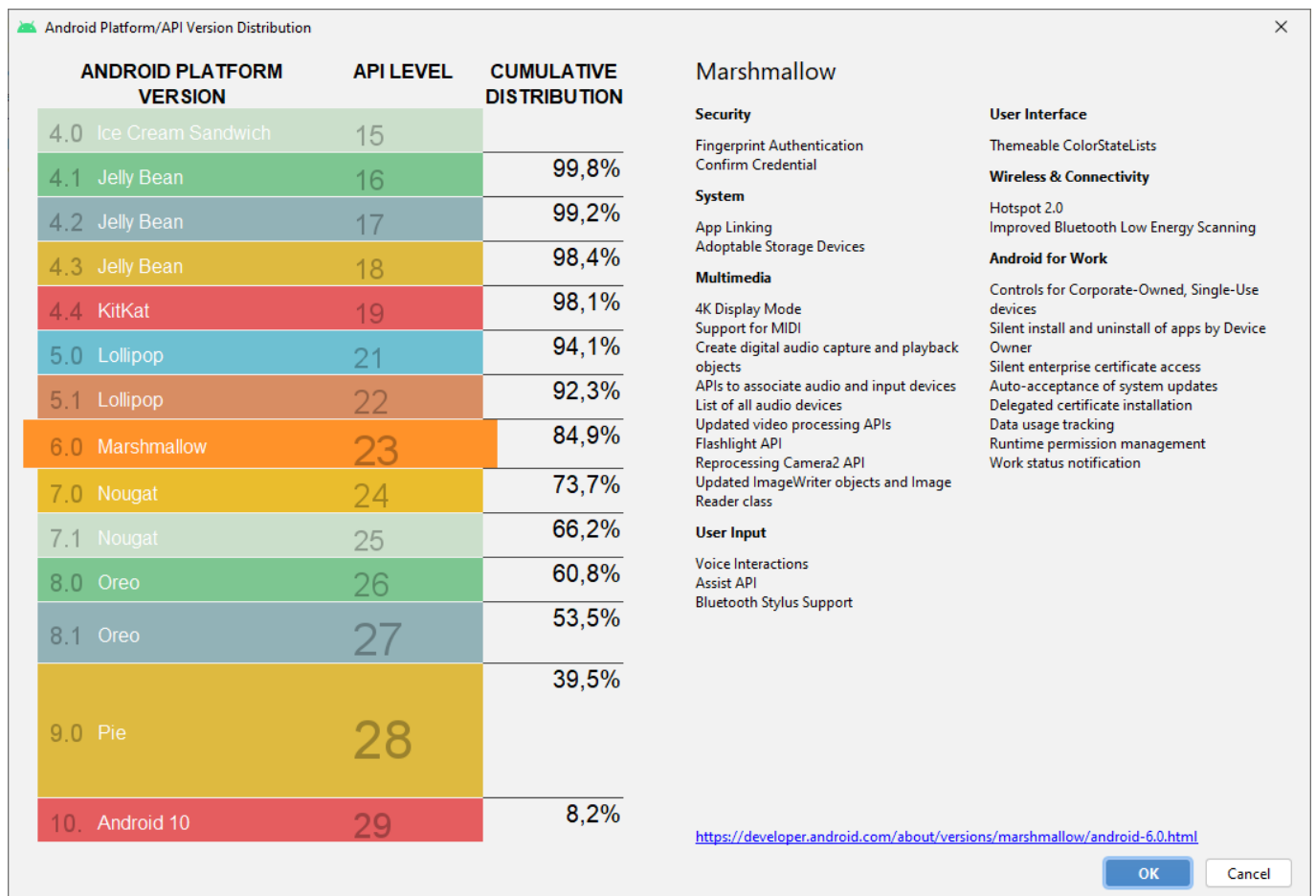


Figure 3: Distribution d'Android

1.1.3. Remarque sur les versions d'API

Chaque API apporte des fonctionnalités supplémentaires. Il y a compatibilité ascendante. Certaines fonctionnalités deviennent *dépréciées* au fil du temps, mais restent généralement disponibles.

On souhaite toujours programmer avec la dernière API (fonctions plus complètes et modernes), mais les utilisateurs ont souvent des smartphones plus anciens, qui n'ont pas cette API.

Or Android ne propose **aucune** mise à jour majeure. Les smartphones restent toute leur vie avec l'API qu'ils ont à la naissance.

Les développeurs doivent donc choisir une API qui correspond à la majorité des smartphones existant sur le marché.

1.1.4. Distribution des versions

Voici la proportion des API en janvier 2021 : figure 3

1.1.5. Remarques diverses

Évolution et obsolescence voulues et très rapides

- Suivre les modes et envies du marché, réaliser des profits

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

1.1.6. Programmation d'applications

Actuellement, les applications sont :

- « **natives** », c'est à dire programmées en Java, C++, Kotlin, compilées et fournies avec leurs données sous la forme d'une archive Jar (fichier *APK*). C'est ce qu'on étudiera ici.
- « **web app** », c'est une application pour navigateur internet, développée en HTML5, CSS3, JavaScript, dans un cadre logiciel (*framework*) tel que Node.js, Angular ou React.
- « **hybrides** », elles sont développées dans un framework comme Ionic, Flutter, React Native... Ces frameworks font abstraction des particularités du système : la même application peut tourner à l'identique sur différentes plateformes (Android, iOS, Windows, Linux...).

La charge d'apprentissage est la même.

1.1.7. Applications natives

Une application native Android est composée de :

- **Sources Java** (ou **Kotlin**) compilés pour une machine virtuelle appelée « *ART* », amélioration de l'ancienne machine « *Dalvik* » (versions ≤ 4.4).
- Fichiers appelés **ressources** :
 - format XML : interface, textes...
 - format PNG : icônes, images...
- **Manifeste** = description du contenu du logiciel
 - version minimale du smartphone,
 - fichiers présents dans l'archive avec leur signature,
 - demandes d'autorisations, durée de validité, etc.

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

1.1.8. Kotlin

C'est un langage de programmation « symbiotique » de Java :

- une classe Kotlin est compilée dans le même code machine que Java,
- une classe Kotlin peut utiliser les classes Java et réciproquement.
- On peut mélanger des sources Java et Kotlin dans une même application.

Kotlin est promu par Google parce qu'il permet de développer des programmes plus sains. Par exemple, Kotlin oblige à vérifier chaque appel de méthode sur des variables objets pouvant valoir `null`, ce qui évite les `NullPointerException`.

1.1.9. Exemple : objet pouvant être null

En Java :

```
String getNomComplet(Personne p) {  
    return p.getPrenom()+" "+p.getNom();  
}  
  
private Personne p1 = getPersonne(); // peut retourner null  
System...println(getNomComplet(p1)); // NullPointerException
```

En Kotlin :

```
fun getNomComplet(p: Personne): String {  
    return p.prenom+" "+p.nom  
}  
  
var p1: Personne = getPersonne() // retour null interdit  
println(getNomComplet(p1)) // ne se compile pas
```

En Java amélioré :

```
import androidx.annotation.NonNull;  
import androidx.annotation.Nullable;  
  
void getNomComplet(@NonNull Personne p) {  
    return p.getPrenom()+" "+p.getNom();  
}  
  
private @Nullable Personne p1 = getPersonne();  
System...println(getNomComplet(p1)); // ne se compile pas
```

NB: les `import` dépendent des bibliothèques utilisées, ici c'est `androidx` comme en TP.

Cependant, il faut y penser. Kotlin vérifie systématiquement de nombreuses choses (initialisations, etc.).

1.1.10. Pas de Kotlin pour ce cours

Kotlin ne remplace pas une analyse sérieuse et une programmation rigoureuse. Kotlin permet seulement de ne pas se faire piéger avec des bugs grossiers.

Nous ne travaillerons pas avec Kotlin car ce langage nécessite un apprentissage. Sa syntaxe est particulièrement abrégée, ex : définition implicite des variables membres à partir du constructeur, définition et appel implicites des setters/getters, liaison entre vues et variables membres d'une classe interface graphique, utilisation des *lambda*, etc. L'ensemble n'est pas toujours très lisible.

Celles et ceux qui voudront faire du Kotlin le pourront, mais sous leur seule responsabilité.

1.2. SDK Android et Android Studio

1.2.1. SDK et Android Studio

Le *Software Development Kit* (SDK) contient :

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

Le logiciel Android Studio offre :

- un éditeur de sources et de ressources
- des outils de compilation : *gradle*
- des outils de test et de mise au point

1.2.2. Android Studio

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

Pour le SDK, vous avez le choix, soit de l'installer automatiquement avec Studio, soit de faire une installation personnalisée. En général, vous pouvez choisir ce que vous voulez ajouter au SDK (version des librairies, versions des émulateurs de smartphones), à l'aide du *SDK Manager*.

NB: dans la suite, certaines copies écran sont hors d'âge, mais je ne peux pas les refaire à chaque variante de Studio.

1.2.3. SDK Manager

C'est le gestionnaire du SDK, une application qui permet de choisir les composants à installer et mettre à jour.

Voir la figure 4, page 24.

1.2.4. Choix des éléments du SDK

Le gestionnaire permet de choisir les versions à installer, ex. :

- Android 11 (API 30)
- ...
- Android 7.0 (API 24)
- ...

Choisir celles qui correspondent aux tablettes qu'on vise, mais tout n'est pas à installer : il faut cocher **Show Package Details**, puis choisir élément par élément. Seuls ceux-là sont indispensables :

- **Android SDK Platform**
- **Intel x86 Atom_64 System Image**

Le reste est facultatif (Google APIs, sources, exemples et docs).

1.2.5. Dossiers du SDK

Le gestionnaire installe les éléments dans plusieurs sous-dossiers :

- **SDK Tools** : indispensable, contient le gestionnaire,
- **SDK Platform-tools** : indispensable, contient **adb**,
- **SDK Platform** : indispensable, contient les librairies,

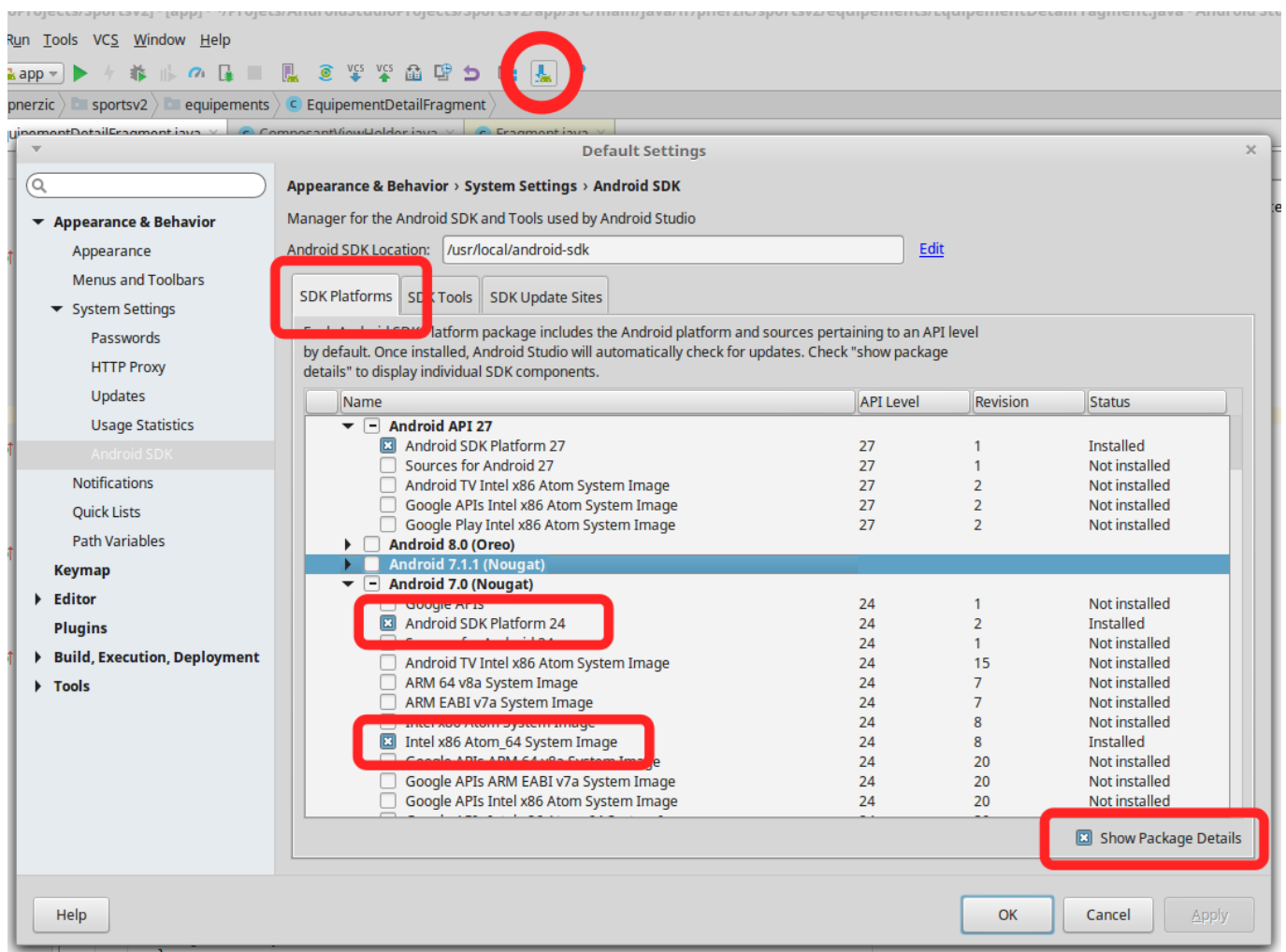


Figure 4: Gestionnaire de paquets Android

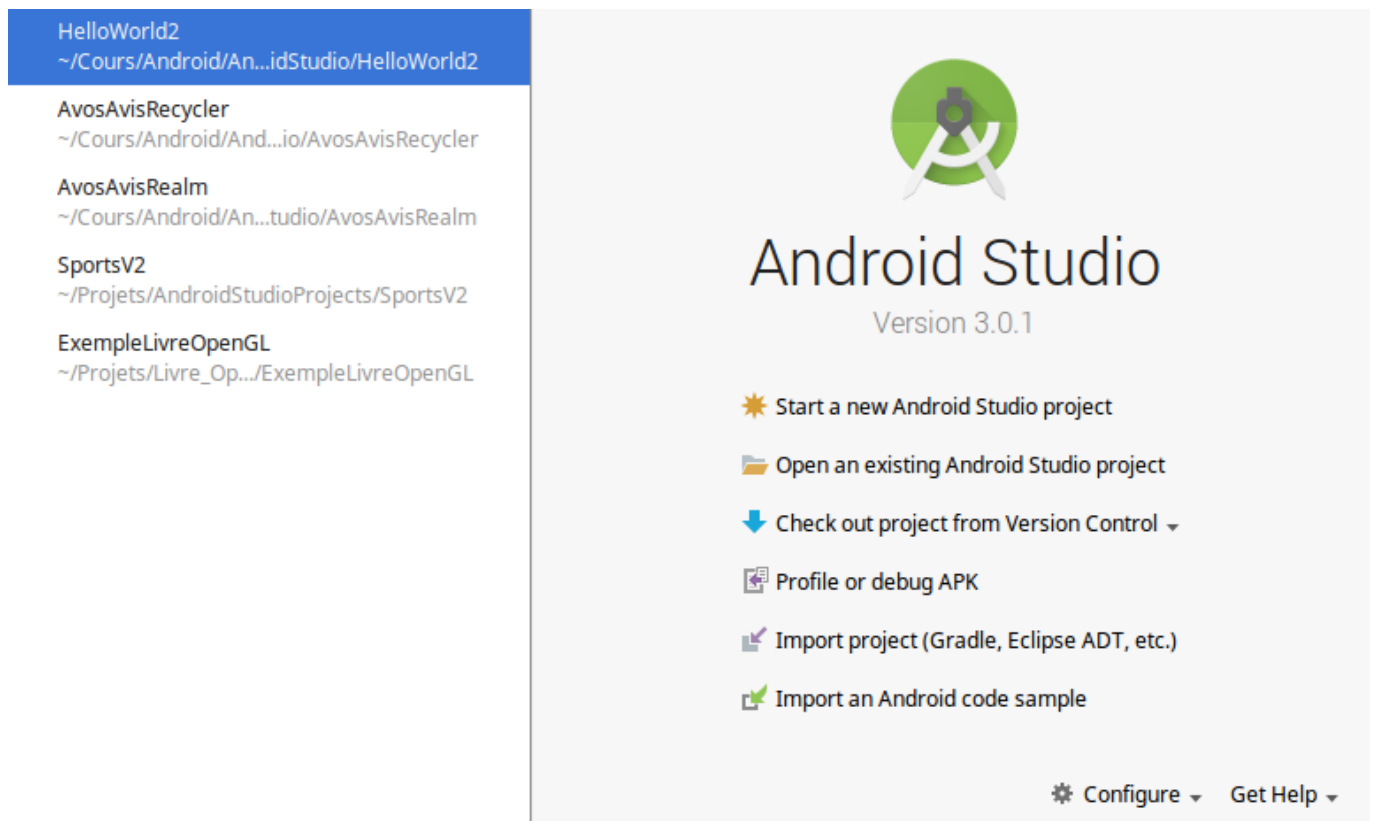


Figure 5: Assistant de création de projet

- **System images** : pour créer des AVD,
- **Android Support** : divers outils pour créer des applications,
- Exemples et sources.

C'est déjà prêt à l'IUT, dans des versions antérieures correspondant à la date de préparation des machines. Il faut savoir qu'il y a pas loin d'une mise à jour par semaine, et donc tout évolue constamment.

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 « *Hello World* » 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 : figure 5

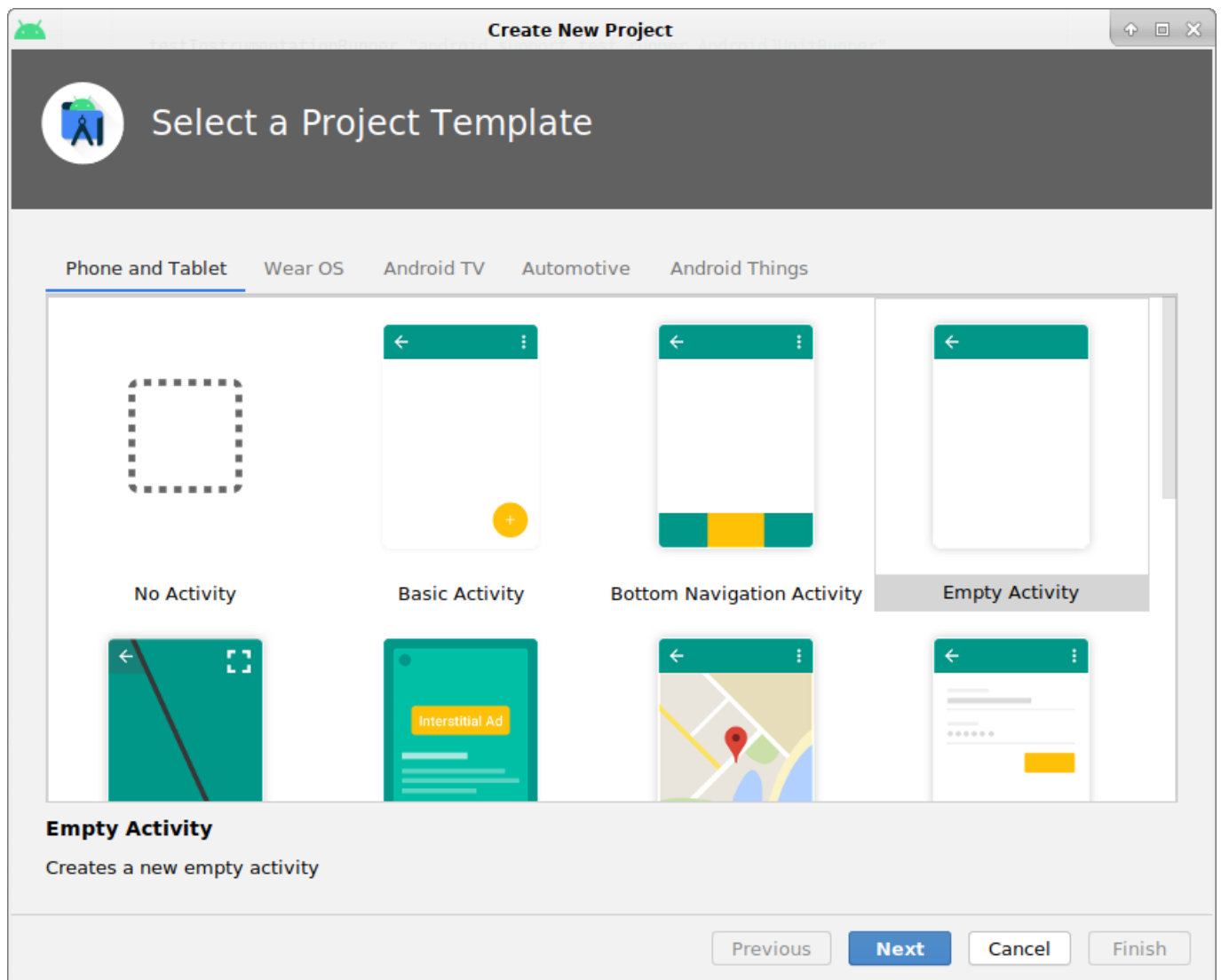


Figure 6: Choix du type d'activité

1.3.3. Modèle d'application

Android Studio propose plusieurs projets de base pour développer le nôtre. Voir la figure 6, page 26. En général, on part de celui appelé *Empty Activity*. Il faut bien connaître les autres si on veut les choisir.

1.3.4. Nom, package et version

Dans le second écran, l'assistant demande le nom du projet, son package, son emplacement, et le niveau minimal de l'API. Voir la figure 7, page 27.

1.3.5. 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

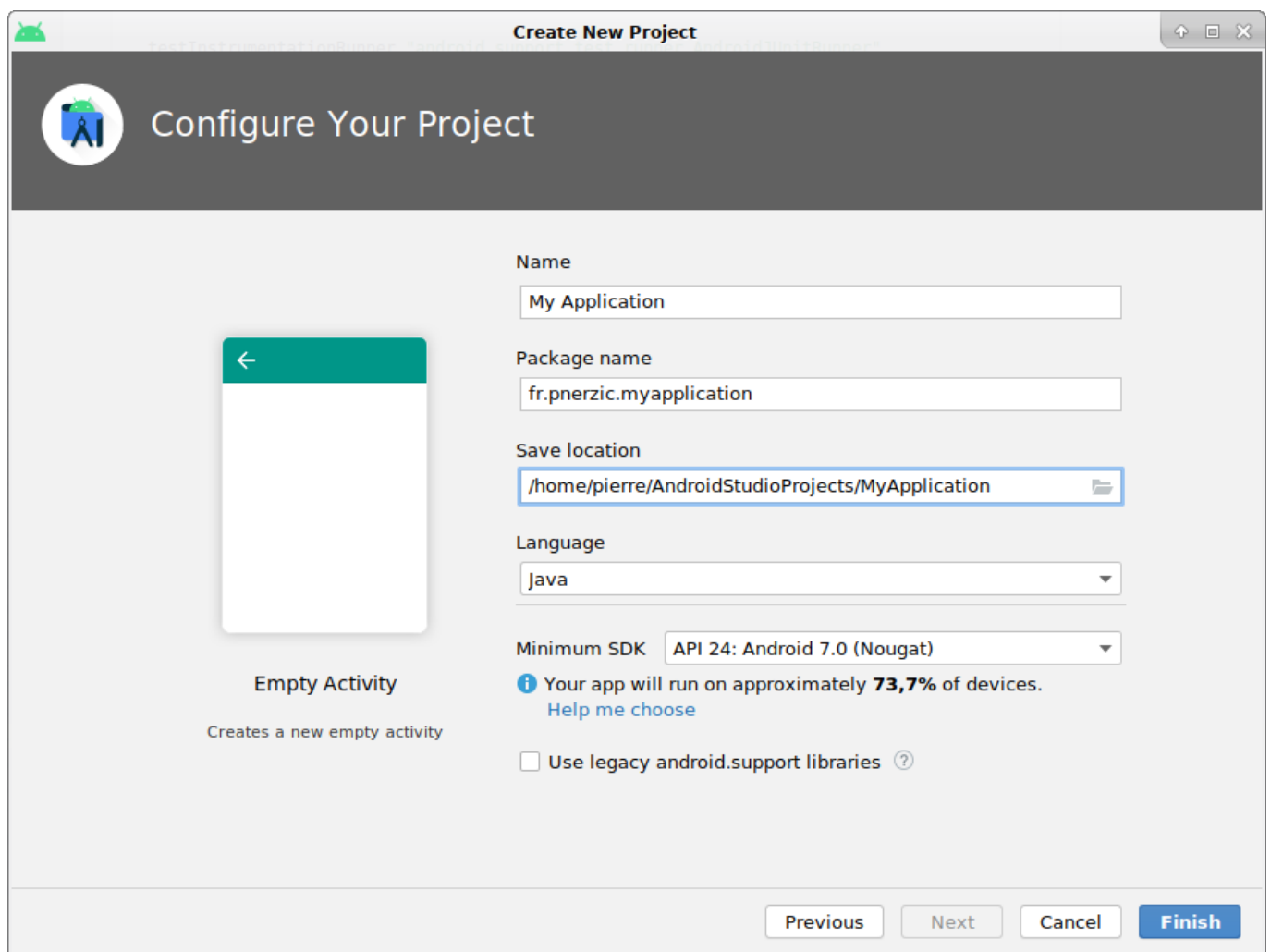


Figure 7: Choix de la version

- `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: ne pas chercher à tout comprendre cette semaine.

1.3.6. Fenêtre du projet

Voir la figure 8, page 29.

1.3.7. 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.

1.3.8. Exemple `res/values/strings.xml`

Voir la figure 9, page 30.

1.3.9. Exemple `res/layout/main.xml`

Voir la figure 10, page 30.

1.3.10. Source XML sous-jacent

Ces éditeurs sont beaucoup plus confortables que le XML brut, mais ne permettent pas de tout faire (widgets custom et plantages).

Assez souvent, 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>
```

Notez le *namespace* des éléments et le préfixe de chaque attribut.

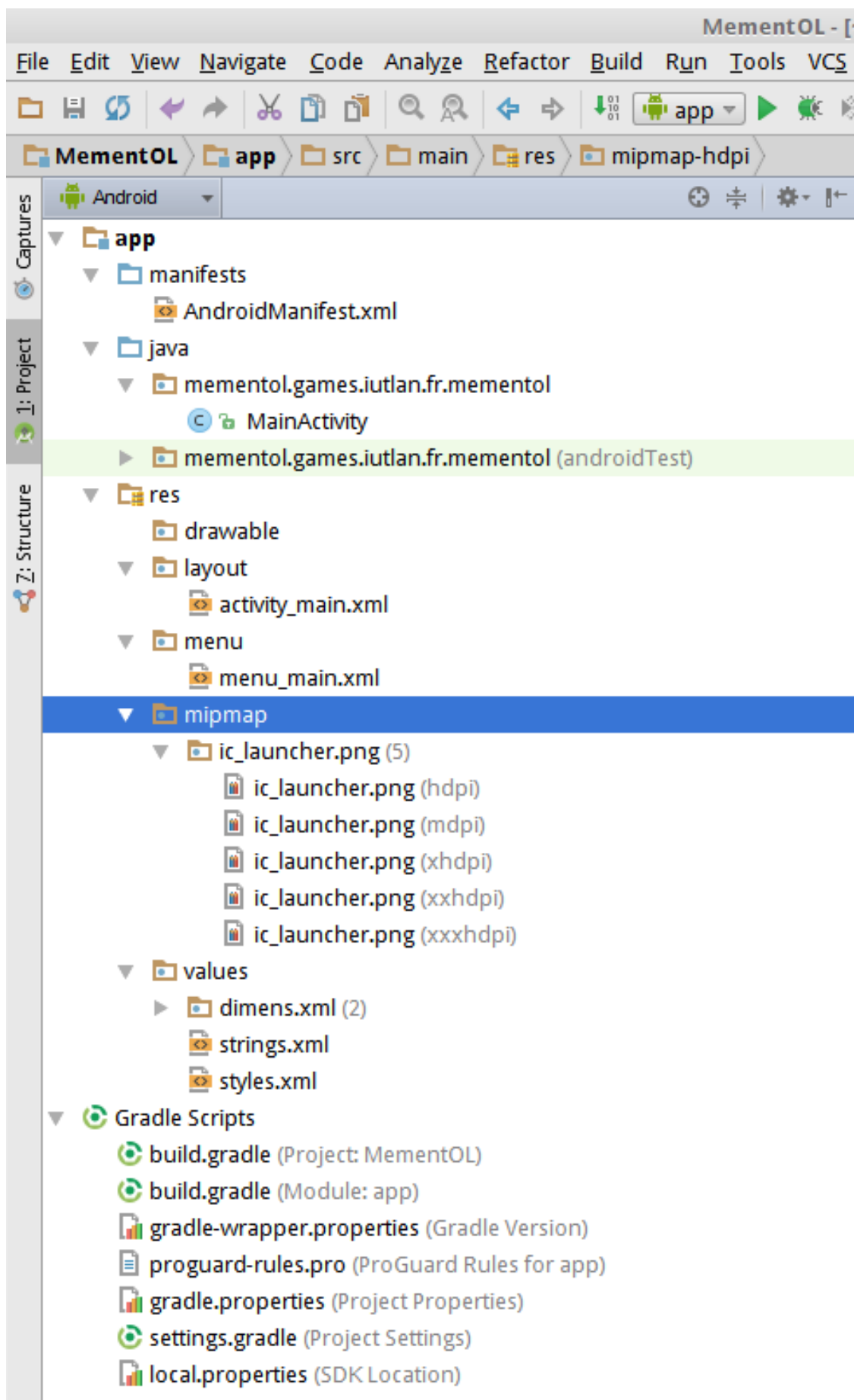


Figure 8: Éléments d'un projet Android

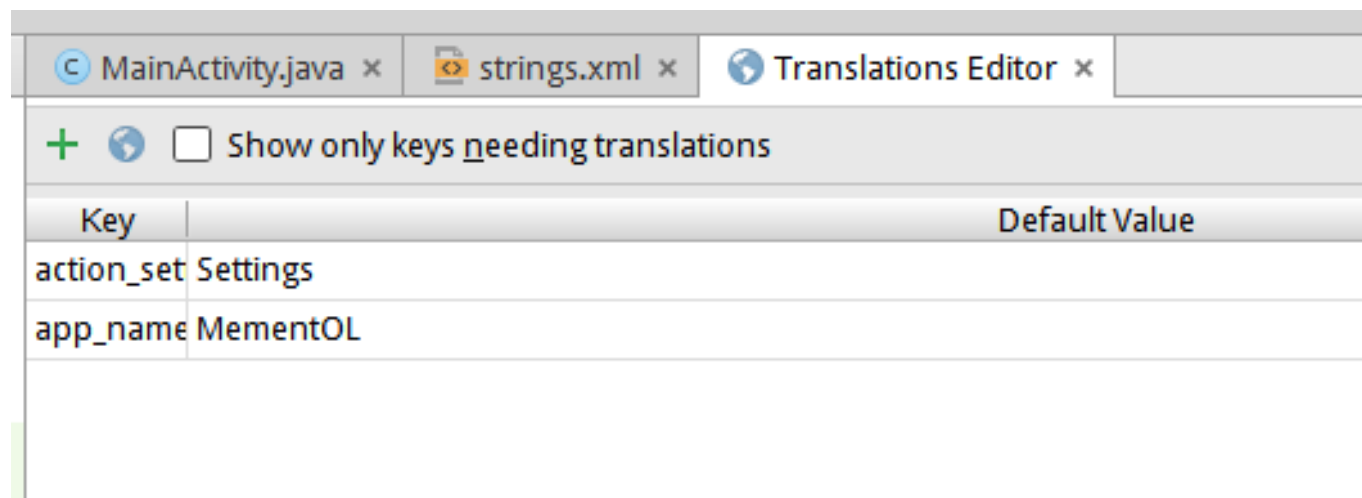


Figure 9: Éditeur du manifeste

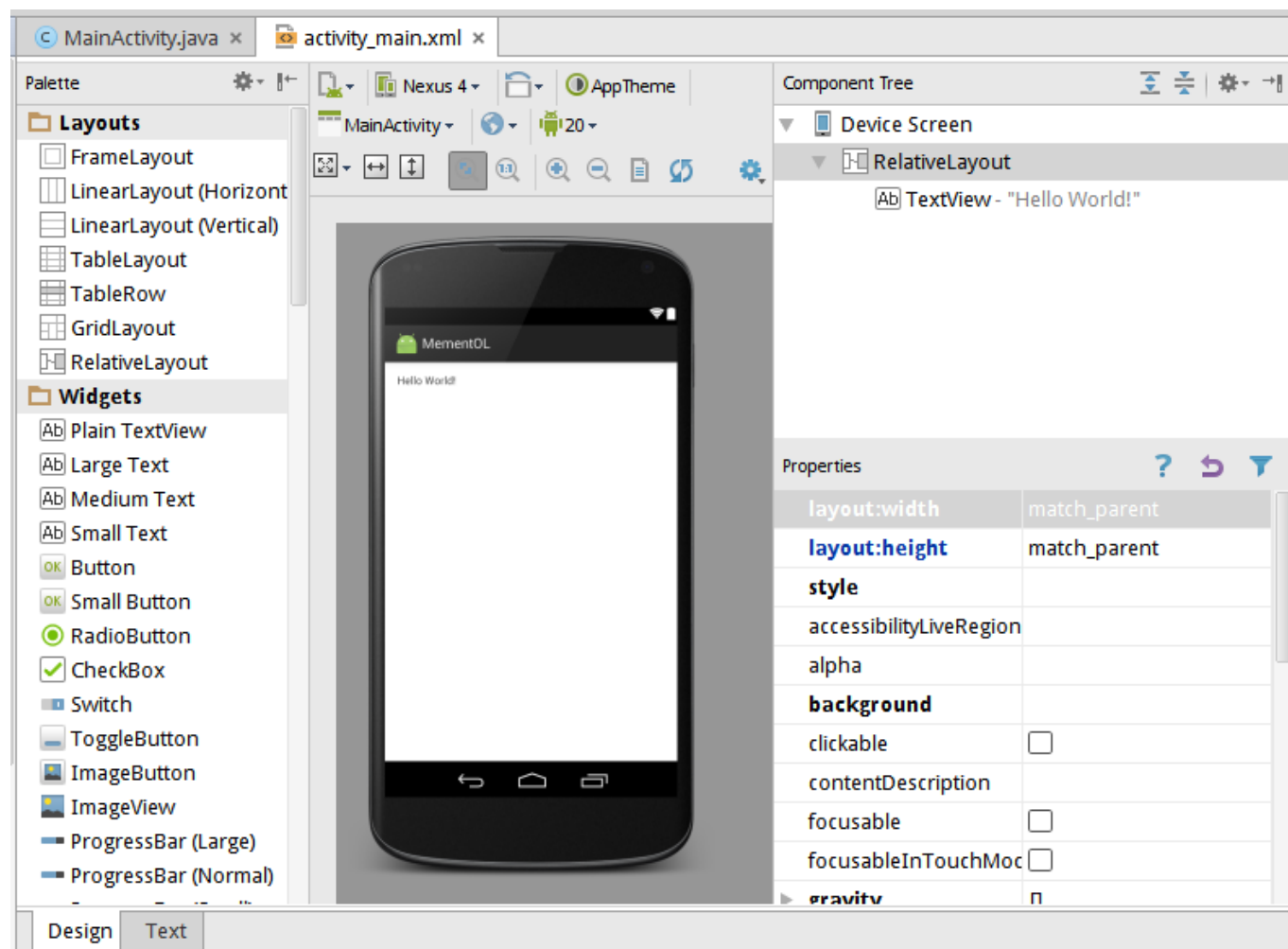


Figure 10: Éditeur graphique

1.3.11. Reconstruction du projet

Chaque modification d'une source ou d'une ressource fait reconstruire le projet (compilation des sources, transformation des XML et autres). C'est automatique.

Dans de rares circonstances, mauvaise mise à jour des sources (partages réseau ou gestionnaire de version) :

- il peut être nécessaire de reconstruire manuellement. Il suffit de sélectionner le menu **Build/Rebuild Project**,
- il faut parfois nettoyer le projet. Sélectionner le menu **Build/Clean Project**.

Ces actions lancent l'exécution de *Gradle*.

1.3.12. Gradle

Gradle est un outil de construction de projets comme **Make** (projets C++ sur Unix), **Ant** (projets Java dans Eclipse) et **Maven**.

De même que **make** se sert d'un fichier **Makefile**, Gradle se sert de fichiers nommés **build.gradle** pour construire le projet.

C'est assez compliqué car AndroidStudio fait une distinction entre le projet global et l'application. Donc il y a deux **build.gradle** :

- un script **build.gradle** dans le dossier racine du projet. Il indique quelles sont les dépendances générales (noms des dépôts Maven contenant les librairies utilisées).
- un dossier **app** contenant l'application du projet.
- un script **build.gradle** dans le dossier **app** pour compiler l'application.

1.3.13. Structure d'un projet AndroidStudio

Un projet AndroidStudio est constitué ainsi :

```
.
+-- app/
|   +-- build/                FICHIERS COMPILÉS
|   +-- build.gradle          SPÉCIF. COMPILATION
|   |-- src/
|       +-- androidTest/      TESTS UNITAIRES ANDROID
|       +-- main/
|           | +-- AndroidManifest.xml  DESCR. DE L'APPLICATION
|           | +-- java/                SOURCES
|           |-- res/                  RESSOURCES (ICONES...)
|           |-- test/                 TESTS UNITAIRES JUNIT
+-- build/                      FICHIERS TEMPORAIRES
+-- build.gradle                SPÉCIF. PROJET
+-- gradle/                     FICHIERS DE GRADLE
```

1.3.14. Utilisation de bibliothèques

Certains projets font appel à des bibliothèques externes. On les spécifie dans le **build.gradle** du dossier **app**, dans la zone **dependencies** :

```
dependencies {  
    // support  
    implementation 'androidx.appcompat:appcompat:1.2.0'  
  
    // annotations  
    annotationProcessor 'org.projectlombok:lombok:1.18.16'  
    implementation 'androidx.annotation:annotation:1.1.0'  
  
    // fuites mémoire  
    debugImplementation 'com.squareup.leakcanary:leakcanary-android:2.5'  
}
```

Les bibliothèques indiquées sont automatiquement téléchargées.

Il y a des cas plus complexes. Par exemple, [Realm](#) (une base de données distribuée), voir sa page [prerequisites](#) :

- dans le `build.gradle` à la racine du projet, mettre :

```
dependencies {  
    classpath 'io.realm:realm-gradle-plugin:10.0.1'  
}
```

- dans le `build.gradle` du dossier `app`, ajouter :

```
apply plugin: 'realm-android'
```

1.3.15. Mises à jour

Google propose souvent des mises à jour : l'IDE, le SDK et Gradle. Et aussi les `build.gradle` des projets.

Par exemple, ce `build.gradle` emploie une ancienne version de Realm, il faudrait mettre 10.0.1 à la place de 7.0.8 :

```
buildscript {  
    repositories { ... }  
    dependencies {  
        classpath 'com.android.tools.build:gradle:4.1.1'  
        classpath 'io.realm:realm-gradle-plugin:7.0.8' // <-!  
    }  
}
```

Android Studio affiche un avertissement s'il y a une mise à jour sur les serveurs (*repositories*). Il faut alors éditer les numéros de version manuellement, puis reconstruire le projet (*sync now* ou *try again*).

On doit parfois compléter le `build.gradle` du dossier `app`.

```
apply plugin: 'com.android.application'  
android {  
    compileSdkVersion 30  
    buildToolsVersion "30.0.3"          <<== PAS PAR DEFAULT
```



```
compileOptions {  
    // pour les lambda  
    sourceCompatibility JavaVersion.VERSION_1_8  
    targetCompatibility JavaVersion.VERSION_1_8  
}  
...  
}  
dependencies {  
    implementation 'androidx.appcompat:appcompat:1.2.0'  
    ...  
}
```

1.4. Première exécution

1.4.1. Exécution de l'application

L'application est prévue pour tourner sur un appareil (smartphone ou tablette) réel ou simulé (virtuel).

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 11, page 34.

1.4.3. Caractéristiques d'un AVD

L'assistant de création de tablette demande :

- Modèle de tablette ou téléphone à simuler,
- Version du système Android,
- 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).

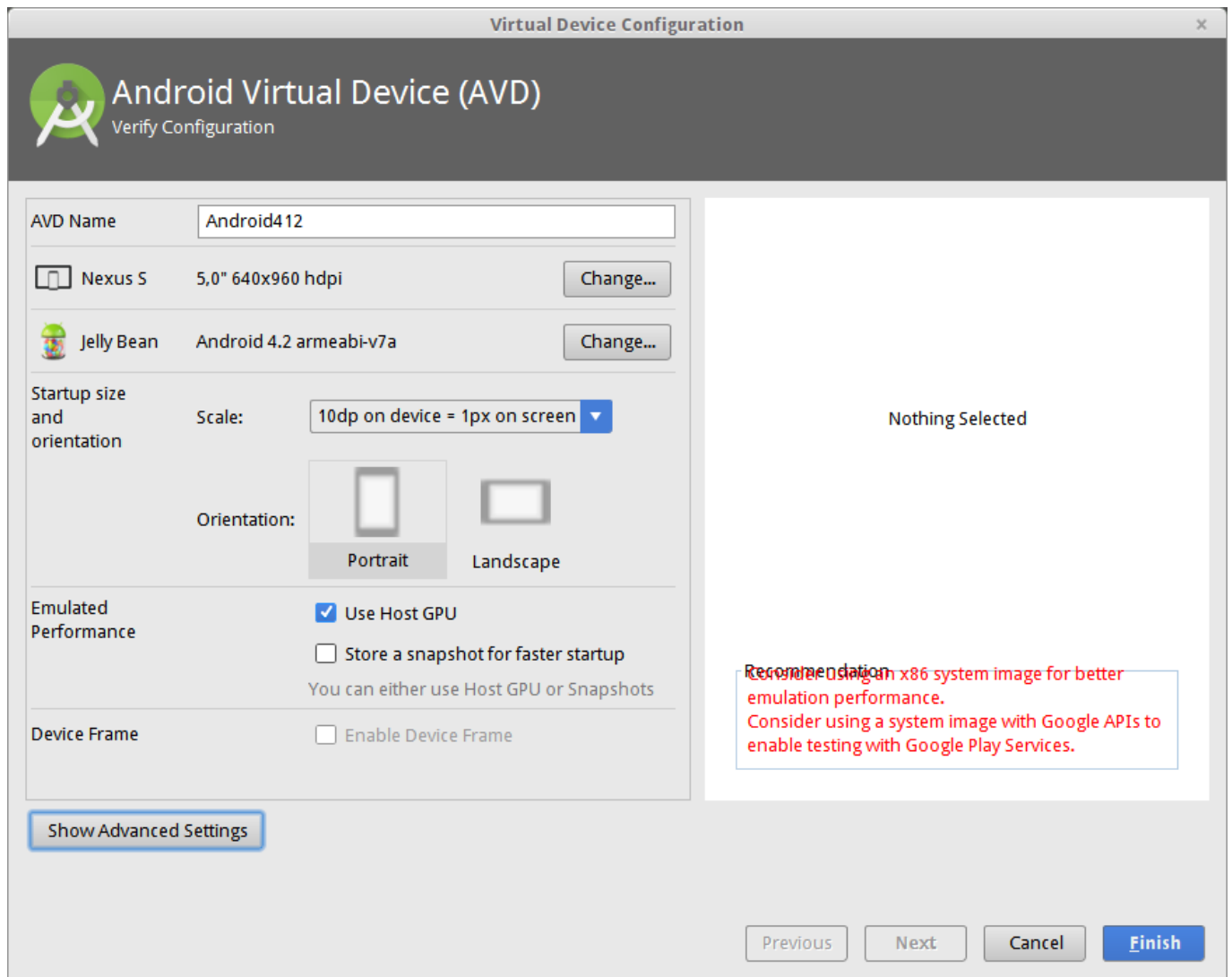


Figure 11: Création d'un AVD

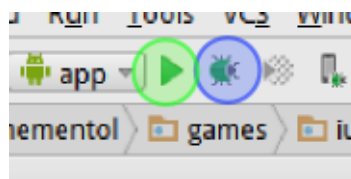


Figure 12: Barre d'outils pour lancer une application

1.4.4. Lancement d'une application

Bouton vert pour exécuter, bleu pour déboguer : Voir la figure 12, page 34.

NB: les icônes changent selon la version d'AndroidStudio.

1.4.5. Application sur l'AVD

Voir la figure 13, page 36.

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

1.5. Communication AVD - Android Studio

1.5.1. Fenêtres Android

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

Logcat Affiche tous les messages émis par la tablette courante

Messages Messages du compilateur et du studio

Terminal Shell unix permettant de lancer des commandes dans le dossier du projet.

1.5.2. Fenêtre Logcat

Des messages détaillés sont affichés dans la fenêtre LogCat : Voir la figure 14, page 37.

Ils sont émis par les applications : debug, infos, erreurs... comme syslog sur Unix : date, heure, gravité, source (code de l'émetteur) et message.

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 vers LogCat

Une application émet un message par ces instructions :



```
import android.util.Log;

public class MainActivity extends Activity {
    public static final String TAG = "monappli";

    void maMethode() {
        Log.i(TAG, "appel de maMethode()");
    }
}
```

Fonctions Log.* :

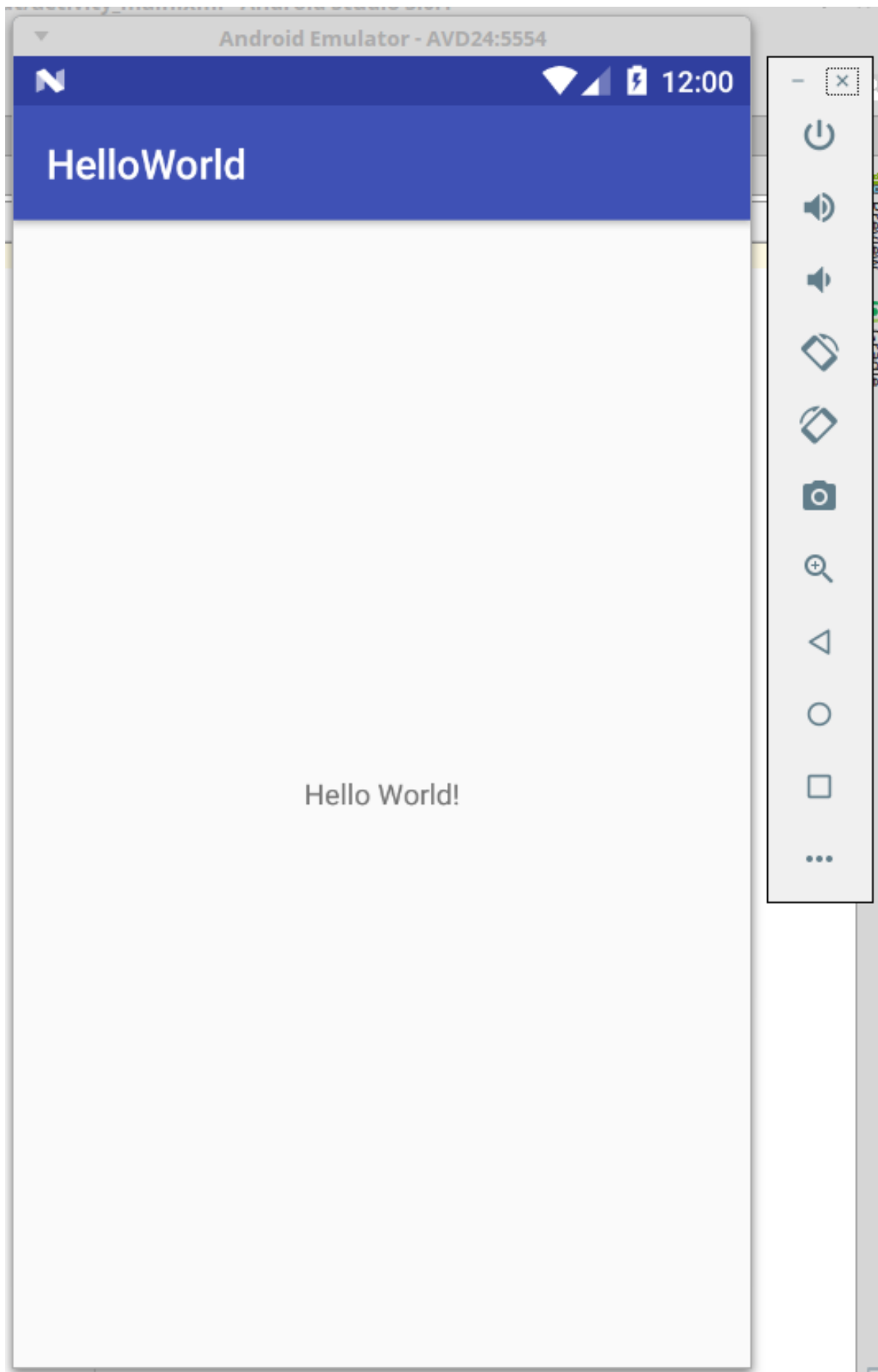


Figure 13: Résultat sur l'AVD

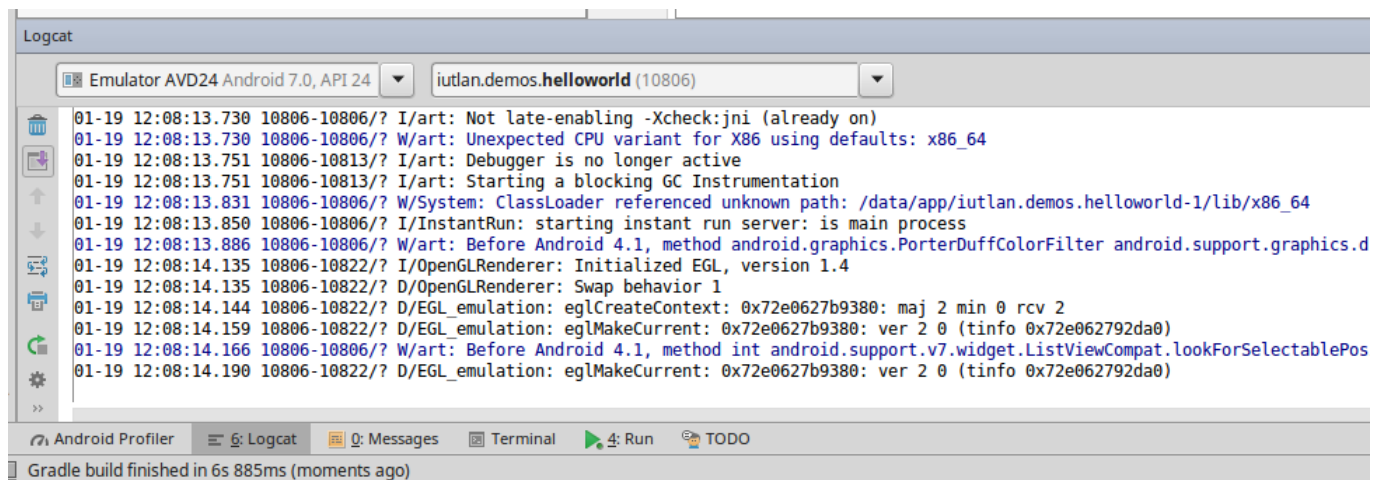


Figure 14: Fenêtre LogCat

- `Log.i(String tag, String message)` affiche une info,
- `Log.w(String tag, String message)` affiche une alerte,
- `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/$
```

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`...

1.5.7. 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.

- 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.

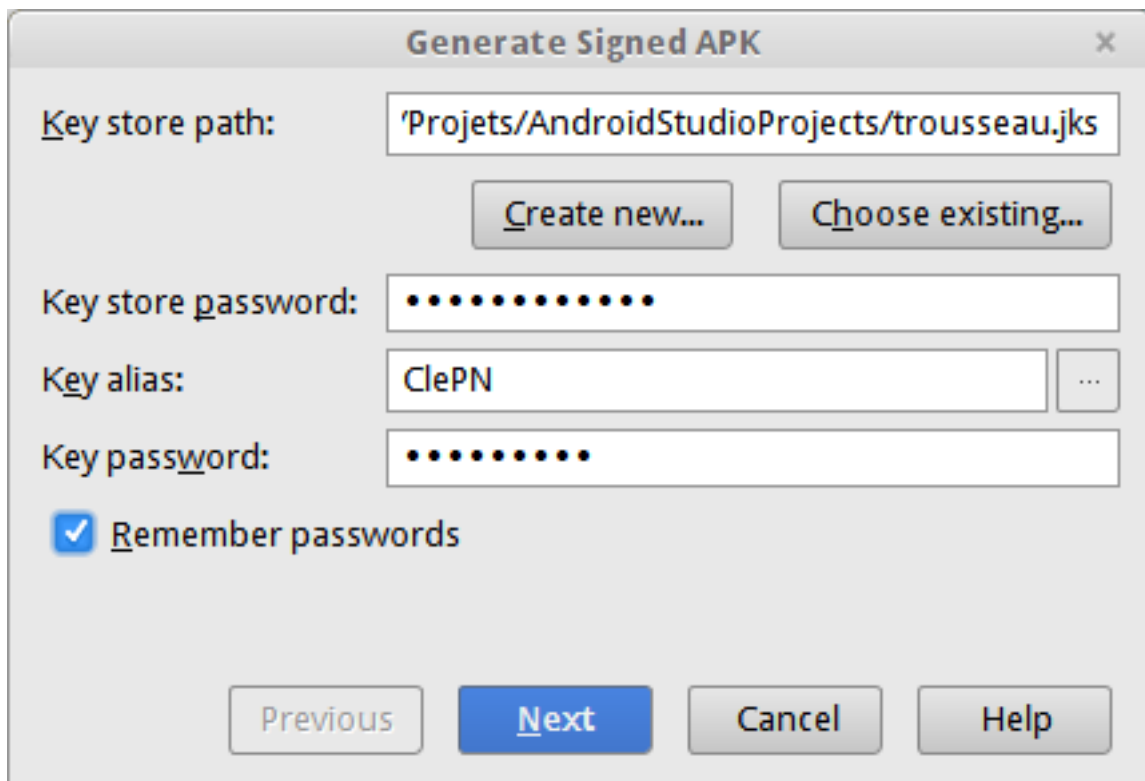


Figure 15: Création d'un trousseau de clés

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`. **Ne les perdez pas.**

1.6.3. Création du *keystore*

figure 15

1.6.4. Création d'une clé

Voir la figure 16, page 40.

1.6.5. Création du paquet

Ensuite, Studio demande où placer le `.apk` :

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 16: Création d'une clé

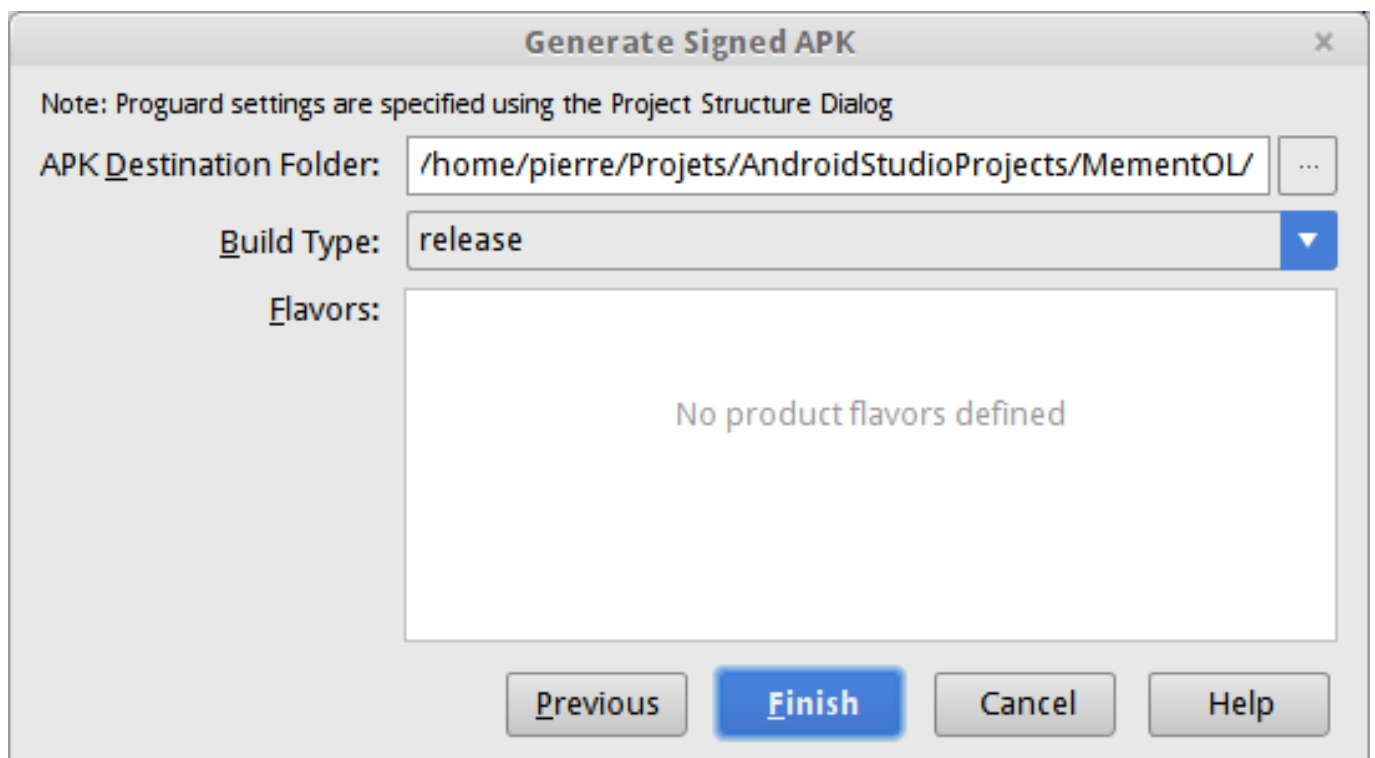


Figure 17: Création du paquet

Voir la figure 17, page 41.

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 :

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

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 vers des compléments d'information.

On va commencer par une présentation très rapide des concepts, puis revenir en détails.

2.1. Présentation rapide des concepts

2.1.1. Composition d'une application

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

- afficher des informations
- éditer des informations

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. Structure d'une interface utilisateur

L'interface d'une activité est composée de *vues* :

- vues élémentaires : boutons, zones de texte, cases à cocher...
- vues de groupement qui permettent l'alignement des autres vues : lignes, tableaux, onglets, panneaux à défilement...

Chaque vue d'une interface est gérée par un objet Java, comme en Java classique, avec AWT, Swing ou JavaFX.

Il y a une hiérarchie de classes dont la racine est [View](#). Elle a une multitude de sous-classes, dont par exemple [TextView](#), elle-même ayant des sous-classes, par exemple [Button](#).

Les propriétés des objets sont généralement visibles à l'écran : titre, taille, position, etc.

2.1.3. Création d'une interface

Ces objets d'interface pourraient être créés manuellement, voir plus loin, mais :

- c'est très complexe, car il y a une multitude de propriétés à définir,
- ça ne permet pas de *localiser*, c'est à dire adapter une application à chaque pays (sens de lecture de droite à gauche)

Alors, on préfère définir l'interface par l'intermédiaire d'un fichier XML qui décrit les vues à créer. Il est lu automatiquement par le système Android lors du lancement de l'activité et transformé en autant d'objets Java qu'il faut.

Chaque objet Java est retrouvé grâce à un « identifiant de ressource ».

2.1.4. Création d'un écran

Chaque écran est géré par une instance d'une sous-classe de `Activity` que vous programmez. Il faut au moins surcharger la méthode `onCreate` selon 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);
    }
}
```

C'est l'appel `setContentView(...)` qui met en place l'interface. Son paramètre est un *identifiant de ressource*, c'à d'une disposition de vues d'interface. C'est ce qu'on va étudier maintenant.

2.2. Ressources

2.2.1. Définition

Les ressources sont tout ce qui n'est pas programme dans une application. Dans Android, ce sont les textes, messages, icônes, images, sons, interfaces, styles, etc.

C'est une bonne séparation, car cela permet d'adapter une application facilement pour tous les pays, cultures et langues. On n'a pas à bidouiller dans le code source et recompiler chaque fois. C'est le même code compilé, mais avec des ressources spécifiques.

Le programmeur prévoit simplement des variantes linguistiques des ressources qu'il souhaite permettre de traduire. Ce sont des sous-dossier, ex: `values-fr`, `values-en`, `values-jp`, etc et il n'y a qu'à modifier des fichiers XML.

2.2.2. Identifiant de ressource

Le problème est alors de faire le lien entre les ressources et les programmes : par un identifiant.

Par exemple, la méthode `setContentView` demande l'identifiant de l'interface à afficher dans l'écran : `R.layout.main`.

Cet identifiant est un entier qui est généré automatiquement par le SDK Android. Comme il va y avoir de très nombreux identifiants dans une application :

- chaque vue possède un identifiant (si on veut)
- chaque image, icône possède un identifiant
- chaque texte, message possède un identifiant
- chaque style, thème, etc. etc.

Ils ont tous été regroupés dans une classe spéciale appelée **R**.

2.2.3. Génération de la classe R

Le SDK Android (**aapt**) construit automatiquement cette classe statique appelée **R**. Elle ne contient que des constantes entières groupées par catégories : **id**, **layout**, **menu**... :

```
public final class R {  
    public static final class string {  
        public static final int app_name=0x7f080000;  
        public static final int message=0x7f080001;  
    }  
    public static final class layout {  
        public static final int main=0x7f030000;  
    }  
    public static final class menu {  
        public static final int main_menu=0x7f050000;  
        public static final int context_menu=0x7f050001;  
    }  
    ...  
}
```

2.2.4. La classe R

Cette classe **R** est générée automatiquement (dans le dossier **generated**) par ce que vous mettez dans le dossier **res** : interfaces, menus, images, chaînes... Certaines de ces ressources sont des fichiers XML, d'autres sont des images PNG.

Par exemple, le fichier **res/values/strings.xml** :

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

Cela rajoute automatiquement deux entiers dans **R.string** : **app_name** et **message**.

2.2.5. Rappel sur la structure d'un fichier XML

Un **fichier XML** : éléments (racine et sous-éléments), attributs, texte et namespaces.

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

Rappel : dans la norme XML, le namespace par défaut n'est jamais appliqué aux attributs, donc il faut mettre le préfixe sur ceux qui sont concernés. Voir le cours [XML](#).

2.2.6. 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*.

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>
```

2.2.7. 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.

2.2.8. 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.

2.2.9. Emploi des ressources texte dans un programme

Dans un programme Java, on peut très facilement placer un texte dans une vue de l'interface :

```
TextView tv = ... // ... voir plus loin pour les vues
tv.setText(R.string.bonjour);
```

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

Cela fonctionne car `TextView.setText()` a deux versions :

- `void setText(String text)` : on peut fournir une chaîne quelconque
- `void setText(int idText)` : on doit fournir un identifiant de ressource chaîne, donc forcément l'un des textes du fichier `res/values/strings.xml`

Par contre, si on veut récupérer l'une des chaînes des ressources pour l'utiliser dans le programme, c'est un peu plus compliqué :

```
String message = getResources().getString(R.string.bonjour);
```

`getResources()` est une méthode de la classe `Activity` (héritée de la classe abstraite `Context`) qui retourne une représentation de toutes les ressources du dossier `res`. Chacune de ces ressources, selon son type, peut être récupérée avec son identifiant.

2.2.10. Emploi des ressources texte dans une interface

Maintenant, dans un fichier de ressources décrivant une interface, on peut également employer des ressources texte :

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

- Le titre du `TextView` sera pris dans le fichier de ressource des chaînes,
- par contre, le titre du `Button` sera une chaîne fixe *hard coded*, non traduisible, donc Android Studio mettra un avertissement.

`@string/nom` est une référence à la chaîne du fichier `res/values*/strings.xml` ayant ce nom.

2.2.11. 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.

2.2.12. 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);
```

2.2.13. 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`.

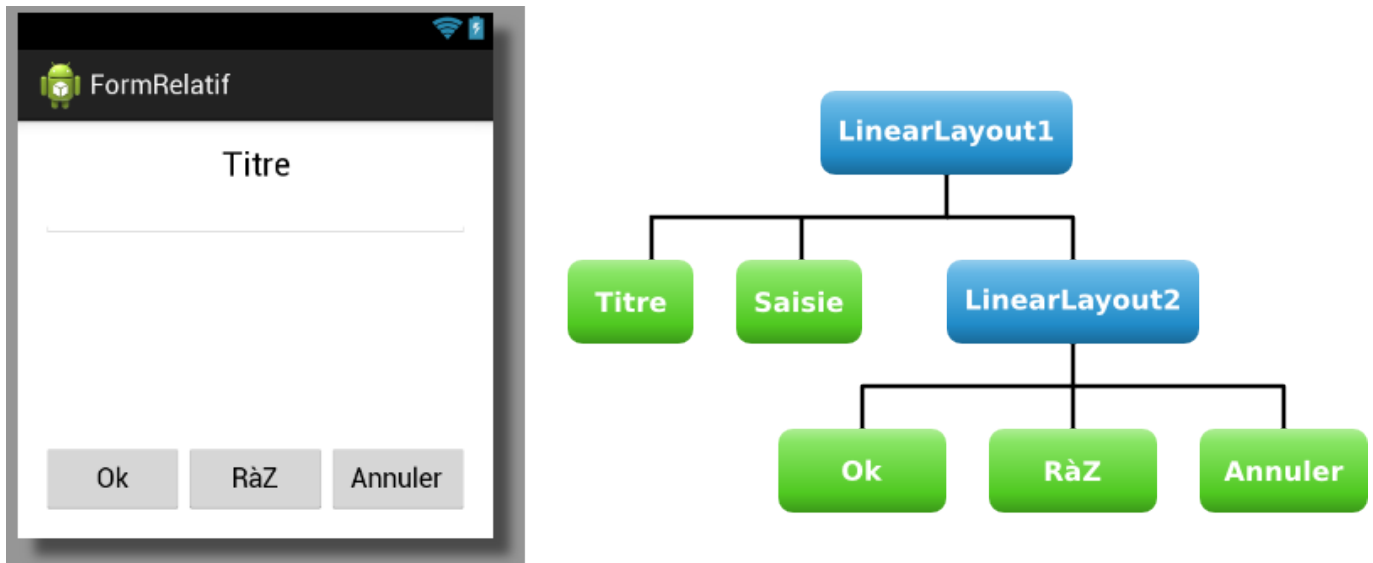


Figure 18: Arbres de vues

2.3. Mise en page (*layouts*)

2.3.1. 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 en colonne
- `RelativeLayout`, `ConstraintLayout` : positionnent leurs vues l'une par rapport à l'autre
- `TableLayout` : positionne ses vues sous forme d'un tableau

2.3.2. Arbre des vues

Les groupes et vues forment un **arbre** :

figure 18

2.3.3. Création d'une interface par programme

Il est possible de créer une interface par programme, comme avec JavaFX et Swing, mais c'est assez compliqué :

```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    TextView tv = new TextView(this);  
    tv.setText(R.string.bonjour);  
    LinearLayout rl = new LinearLayout(this);
```



```
LayoutParams lp = new LayoutParams();  
lp.width = LayoutParams.MATCH_PARENT;  
lp.height = LayoutParams.MATCH_PARENT;  
rl.addView(tv, lp);  
setContentView(rl);  
}
```

2.3.4. Ressources de type *layout*

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

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

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

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

La méthode `setContentView` fait afficher le *layout* indiqué.

2.3.5. Identifiants et vues

Lorsque l'application veut manipuler l'une de ses vues, elle doit utiliser `R.id.symbole`, ex :

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

avec la définition suivante dans `res/layout/main.xml` :

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

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

2.3.6. `@id/nom` ou `@+id/nom` ?

Dans les fichiers `layout.xml`, il y a deux notations à ne pas confondre :

`@+id/nom` pour définir (créer) un identifiant

`@id/nom` pour référencer un identifiant déjà défini ailleurs

Exemple, le Button `btn` se place sous 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>
```

2.3.7. Paramètres de positionnement

La plupart des groupes utilisent des *paramètres de taille et 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 obligatoires pour toutes les vues : `android:layout_width` et `android:layout_height`,
- ceux qui sont demandés par le groupe englobant et qui en sont spécifiques, comme `android:layout_weight`, `android:layout_alignParentBottom`, `android:layout_centerInParent`.

2.3.8. Paramètres obligatoires

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 prend la place minimale
- `"match_parent"` : la vue occupe tout l'espace restant
- `"valeurdp"` : une taille fixe, ex : `"100dp"` mais c'est peu recommandé, sauf `0dp` pour un cas particulier, voir plus loin

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.

Par exemple, trois boutons dans un `LinearLayout` horizontal :

Bouton	<code>layout_width</code>	<code>layout_height</code>
OK1	<code>wrap_content</code>	<code>wrap_content</code>
OK2	<code>wrap_content</code>	<code>match_parent</code>
OK3	<code>match_parent</code>	<code>wrap_content</code>

Voir la figure 19, page 51.

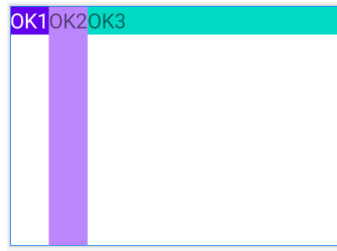


Figure 19: Arbre de vues

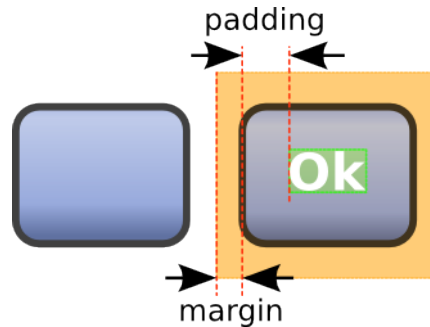


Figure 20: Bords et marges

2.3.9. 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 20

2.3.10. 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" />
```

2.3.11. 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_width="wrap_content"
```



Figure 21: Influence des poids sur la largeur

```
        android:layout_height="wrap_content"/>
    <Button android:text="Annuler"
        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](#).

2.3.12. 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.

2.3.13. Exemple de poids différents

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

figure 21

2.3.14. Groupe de vues `TableLayout`

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

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

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

2.3.15. 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 réductibles

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

2.3.16. 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...`

2.3.17. 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.

2.3.18. 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.

En TP, nous étudierons le [ConstraintLayout](#), présenté sur [cette page](#).

2.4. Composants d'interface

2.4.1. Vues

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

- Textes : titres, chaînes à saisir
- Boutons, cases à cocher...
- Curseurs : pourcentages, barres de défilement...

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.

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

2.4.3. Button

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

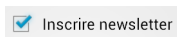


```
<Button  
    android:id="@+id/btn_ok"  
    android:text="@string/ok"  
    ... />
```

- 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.

2.4.4. 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`.

2.4.5. 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.

2.4.6. Autres vues

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

2.4.7. 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 d'une ou 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,
- 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 de classe des

```
<?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 le seul élément sous la racine `<manifest>` et ses filles sont des `<activity>`.

3.1.3. 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.

3.1.4. 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`.

3.1.5. 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`.

3.1.6. 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 :



```
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.

3.1.7. 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.

3.1.8. 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" />
    ...
    <application .../>
</manifest>
```

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

NB: les premières activités que vous créerez n'auront besoin d'aucune permission.

3.1.9. Sécurité des applications (pour info)

Chaque application est associée à un UID (compte utilisateur Unix) unique dans le système. Ce compte les protège les unes des autres. Il peut être défini dans le fichier AndroidManifest.xml sous forme d'un nom de package :

```
<?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.

3.2. 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 (Google) illustrant les possibilités de navigation parmi plusieurs activités.

Voir la figure 22, page 61.

3.2.3. Lancement avec ou sans retour

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.

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);
```

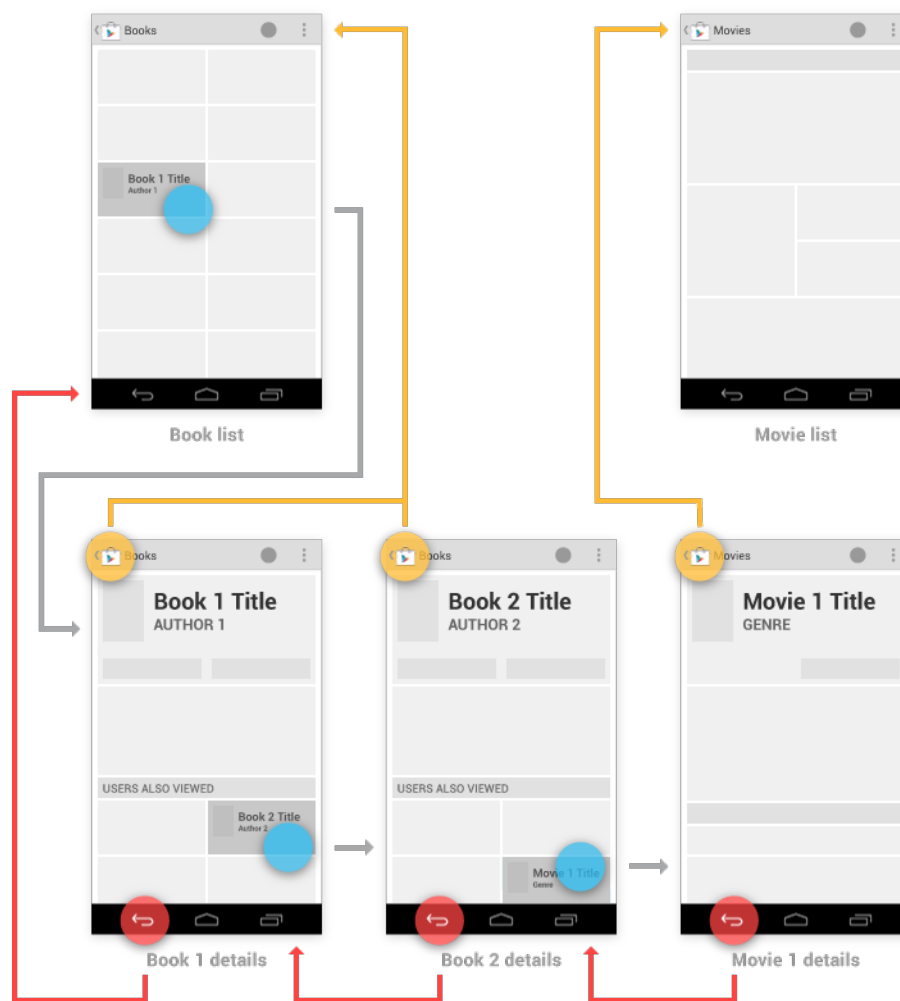


Figure 22: Navigation parmi les activités d'une application

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](#).

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
            ...
    }
}
```

3.2.5. 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()`).

3.2.6. 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)`

3.2.7. 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.

3.2.8. 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.

Il est très recommandé de placer les chaînes dans des constantes, dans la classe appelée :



```
public static final String EXTRA_IDINFO = "idInfo";  
public static final String EXTRA_HIDDEN = "hiddencopy";
```

3.2.9. 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.

3.2.10. Définition d'un contexte d'application

Pour commencer, dériver une sous-classe de `Application` :



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

    public int getVarGlob() { return varglob; }

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

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>
```

3.2.11. 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.getVarGlob() ...
```

Remarquez la conversion de type du contexte.

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 :

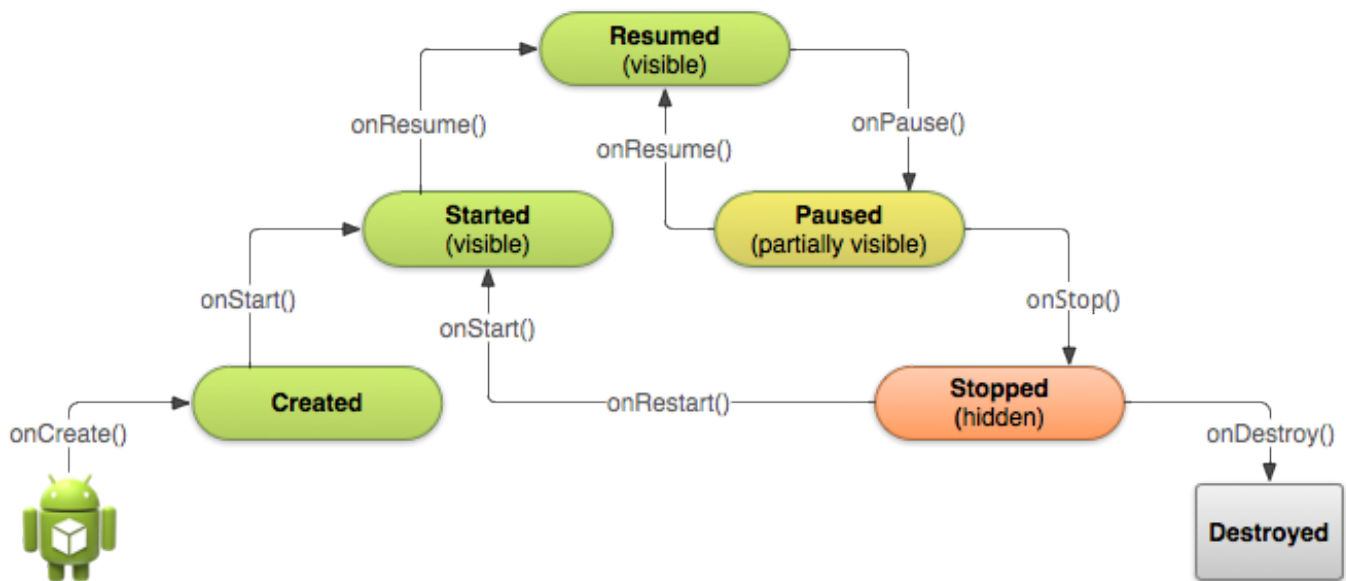


Figure 23: Cycle de vie

- 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 changement d'états d'une activité :

figure 23

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) {
        // obligatoire
    }
}
```

```
        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() {
    // obligatoire
    super.onDestroy();

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

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.

Voir [IcePick](#) pour une automatisation de ce concept.

3.4. Vues et activités

3.4.1. Obtention des vues

La méthode `setContentView` charge une mise en page (*layout*) 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 = 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. Cela peut arriver quand on se trompe de layout pour la vue.

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 = 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, ça appelle automatiquement la méthode `onValider` de l'activité grâce à l'attribut `onClick="onValider"`.

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

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


```
public void onValider(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*), comme un `EventHandler` dans JavaFX. Un écouteur est une instance de classe implémentant l'interface `View.OnClickListener` qui possède la méthode `public void onClick(View v)`.


Cela peut être :

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

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

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


Dans la méthode `onClick`, il faut employer la syntaxe `MonActivity.this` pour manipuler les variables et méthodes de l'activité sous-jacente.

Sur les dernières versions d'AndroidStudio, cet écouteur est transformé en *lambda*. C'est une écriture plus compacte qu'on retrouve également en JavaScript, et très largement employée en Kotlin.

```
Button btn = findViewById(R.id.btn_valider);  
btn.setOnClickListener((btn) -> {  
    // faire quelque chose  
});
```


Cette transformation est possible parce que l'interface `View.OnClickListener` ne possède qu'une seule méthode.

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 View.OnClickListener {
    public void onClick(View btn) {
        // faire quelque chose
    }
};
public void onCreate(...) {
    ...
    Button btn = 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 View.OnClickListener {
    public void onCreate(...) {
        ...
        Button btn = 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*.

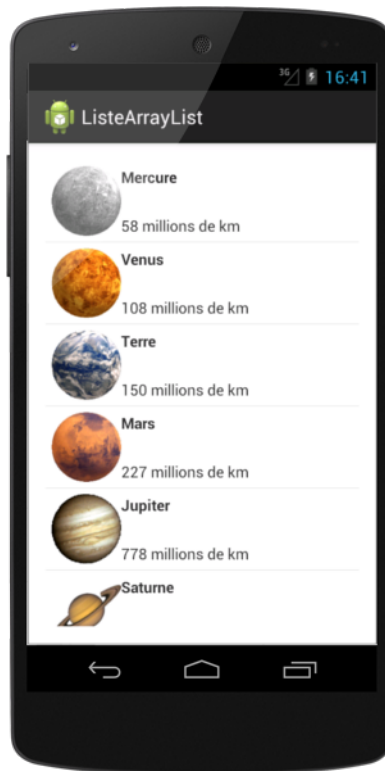


Figure 24: Liste d'items

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 24

4.1. Présentation

4.1.1. Principe général

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

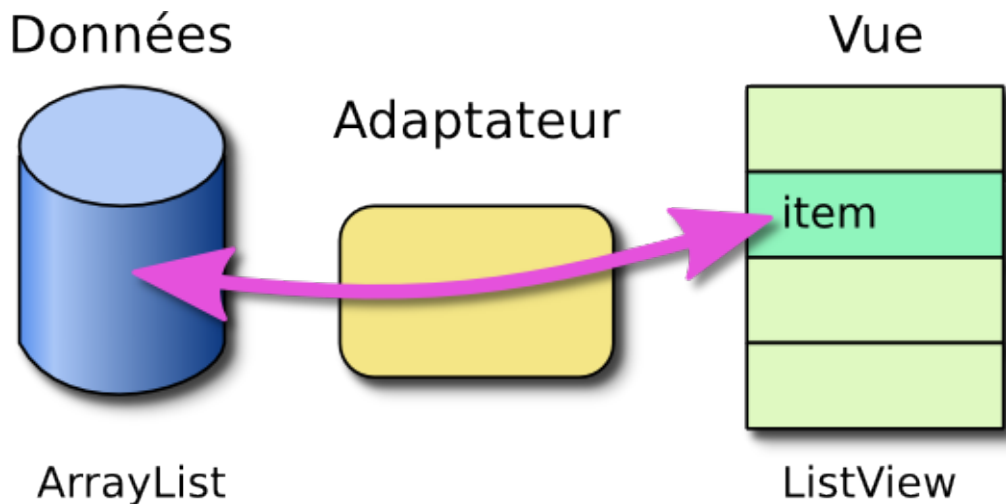


Figure 25: Vue, adaptateur et données

- Cette semaine, la liste est stockée dans un tableau type `ArrayList` ; en semaine 6, ça sera dans une BDD *Realm*.
- L'écran est occupé par un `ListView` ou un `RecyclerView`. Ce sont des vues spécialisées dans l'affichage de listes quelconques.

Consulter [cette documentation](#) sur les `ListView` et [celle-ci](#), très compliquée, sur les `RecyclerView`, mais surtout [celle là](#) sur les adaptateurs.

On va d'abord parler des `ListView`, les plus simples, puis des `RecyclerView` un peu plus complexes mais plus polyvalents.

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 25

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 75.
- Un tableau constant 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(...)  
{  
    ...  
  
    // copie du tableau dans un ArrayList  
    mListe = new ArrayList<>(Arrays.asList(initdata));  
}
```

On peut aussi allouer la liste classiquement et recopier les éléments dans une boucle.

4.1.6. Rappels sur le container 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<>();
```

NB: le type entre `<>` à droite est facultatif.

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>
```

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

Intérêt : traduire les noms des planètes dans d'autres langues.

4.1.8. 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 Realm locale ou distante, au lieu de ce tableau dynamique, ce qui résout proprement le problème de manière à la fois élégante et persistante d'une exécution à l'autre.

4.2. Affichage de la liste

4.2.1. Activité

Android offre deux possibilités :

- dériver la classe `ListActivity`,
- dériver la classe `Activity` de base.

Ces deux possibilités sont très similaires : leur *layout* contient un `ListView`, il y a 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. Par exemple, si on rajoute un `TextView` particulier, on peut avoir un message « La liste est vide ». Comme l'avantage est minime, nous n'en parlerons pas.

Tandis qu'avec une simple `Activity`, c'est à nous de tout faire, voir page 85 pour la gestion des clics.


4.2.2. Mise en œuvre

Dans tous les cas, deux layouts sont à définir :

1. Un layout pour l'activité ; il doit contenir un `ListView` identifié par `@android:id/list`,
2. Un layout d'item ; il contient par exemple un `TextView` identifié par `@android:id/text1`, ce layout est affiché pour chaque item des données.

Consulter la documentation de [ListActivity](#).

4.2.3. Layout de l'activité pour afficher une liste

Voici le layout `main.xml`. J'ai rajouté le `TextView` qui affiche « Liste vide ». Notez les identifiants spéciaux `list` et `empty`. 

```
<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="La liste est vide"
        ... />
</LinearLayout>
```

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

4.2.4. Mise en place du layout d'activité

Classiquement : 

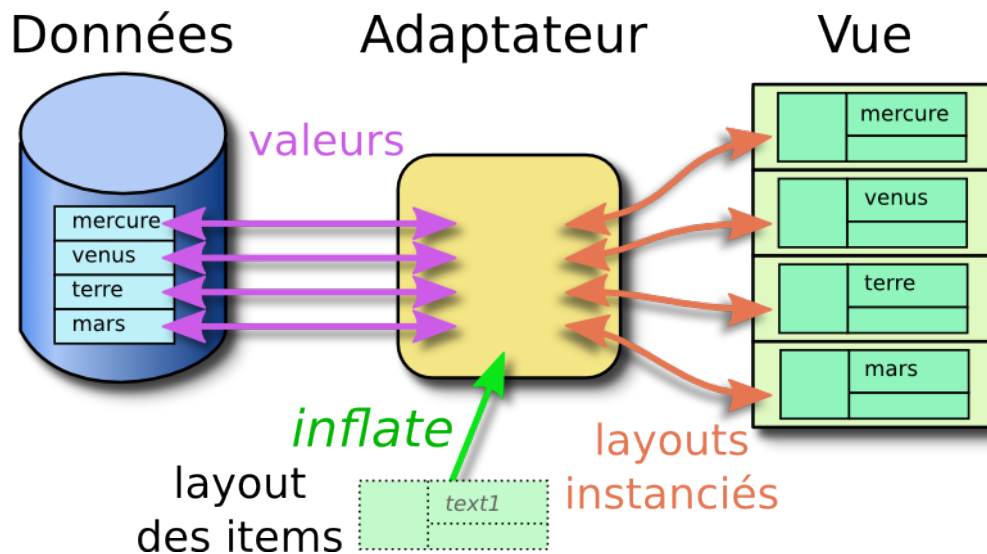


Figure 26: Adaptateur entre les données et la vue

```
@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<>();
    ...
}
```

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 26

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.

L'adaptateur 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 `ArrayList` simple,
- `SimpleCursorAdapter` pour accéder à une base de données SQLite (on ne verra pas).

En général, dans une application innovante, il faut définir son propre adaptateur, voir page 80, 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`

textview_id identifiant du `TextView` dans ce layout, p. ex. `android.R.id.text1` ou
`R.id.item_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<>(this,
    R.layout.item,
    R.id.item_nom,
    mListe);
// associer la liste affichée et l'adaptateur
ListView lv = findViewById(android.R.id.list);
lv.setAdapter(adapter);
```

La classe `Planete` doit avoir une méthode `toString()`, cf page 73. 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);
```



Figure 27: Layout complexe

```
mListe = new ArrayList<>();  
...  
  
ArrayAdapter<Planete> adapter = new ArrayAdapter...  
  
// association liste - adaptateur  
setListAdapter(adapter);  
}
```

4.3.7. Layout pour un item

Vous devez définir le layout `item.xml` pour afficher un item :



```
<TextView xmlns:android="..."  
    android:id="@+id/item_nom"  
    android:textStyle="bold"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"/>
```

Ce layout est réduit à un `TextView` dont l'identifiant Java est `R.id.item_nom`. Retrouvez les dans la création de l'adaptateur :

```
new ArrayAdapter<>(this,  
    R.layout.item, R.id.item_nom, mListe);
```

4.3.8. Autre layouts

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

figure 27



```
<RelativeLayout xmlns:android="..." ...>  
    <ImageView android:id="@+id/item_image" .../>  
    <TextView android:id="@+id/item_nom" .../>  
    <TextView android:id="@+id/item_distance" .../>  
</RelativeLayout>
```

Voir les adaptateurs personnalisés, page 80.

4.3.9. Layouts prédéfinis

Android définit deux 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.

4.3.10. Exemple avec les layouts prédéfinis

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



```
// créer un adaptateur standard pour mListe
ArrayAdapter<Planete> adapter =
    new ArrayAdapter<>(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 ctx, List<Planete> planetes)
    {
        super(ctx, 0, planetes);
    }
}
```

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

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`. Voir transpa suivant.
NB: ce paramètre s'appelle `convertView` dans les docs.
- `parent` : le `ListView` auquel sera rattaché cette vue.

4.4.2. Réutilisation d'une vue

À l'écran, un `ListView` ne peut afficher que quelques éléments en même temps. On peut faire défiler vers le haut ou vers le bas pour voir les autres.

Au lieu d'instancier autant de layouts que d'éléments dans la liste, un `ListView` réutilise ceux qui deviennent invisibles à cause du défilement. Par exemple, quand on défille vers le haut pour voir plus bas, les éléments du haut disparaissent de la vue ; on peut donc les réutiliser en changeant leur contenu et en les affichant en dessous.

C'est comme un escalator : les marches qui arrivent à un bout reviennent au début.

Le paramètre appelé `recup` (ou `convertView`) est une telle vue réutilisée. S'il est `null`, on doit créer la vue pour cet item, sinon on doit reprendre `recup` et changer son contenu.

4.4.3. Méthode `getView` personnalisée

Voici donc la surcharge de cette méthode :



```
@Override  
public  
View getView(int position, View recup, ViewGroup parent)  
{  
    // créer ou réutiliser un PlaneteView  
    PlaneteView itemView = (PlaneteView) recup;  
    if (itemView == null)  
        itemView = PlaneteView.create(parent); // <==(!!)  
  
    // afficher les valeurs  
    itemView.setItem(super.getItem(position));  
    return itemView;  
}
```

4.4.4. Méthode `PlaneteView.create`

Cette méthode crée une instance de `PlaneteView` qui est un groupe de vues pour afficher un item des données.

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



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

Dans le `ListView`, on va avoir plusieurs instances de `PlaneteView`, chacune pour un élément de la liste.

4.4.5. Layout d'item `res/layout/item.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 de la balise `RelativeLayout` :

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

Les propriétés de placement restent les mêmes. `PlaneteView` est seulement une sous-classe de `RelativeLayout` avec quelques variables d'instance et méthodes de plus.

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 81 :

```
// créer ou réutiliser un PlaneteView
PlaneteView itemView = (PlaneteView) recup;
if (itemView == null) itemView = 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 (objets Java) à partir d'un layout XML :

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

On lui fournit l'identifiant du layout, p. ex. celui des items. Elle crée les vues spécifiées dans `res/layout/item.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 PlanetextView create(ViewGroup parent)
{
    LayoutInflater li =
        LayoutInflater.from(parent.getContext());
    PlanetextView itemView = (PlanetextView)
        li.inflate(R.layout.item, parent, false);
    itemView.findViewById();
    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 `findViewById`

Cette méthode a pour but de récupérer les objets Java correspondant aux `TextView` et `ImageView` de l'item. Ils sont placés dans des variables de la classe : 

```
// vues du layout item_planete.xml
private TextView tvNom;
private TextView tvDistance;
private ImageView ivImage;

private void findViewById()
{
    tvNom = findViewById(R.id.item_nom);
    tvDistance = findViewById(R.id.item_distance);
    ivImage = findViewById(R.id.item_image);
}
```

4.4.12. Pour finir, la méthode `PlanetextView.setItem`

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

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

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

4.4.13. Récapitulatif

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

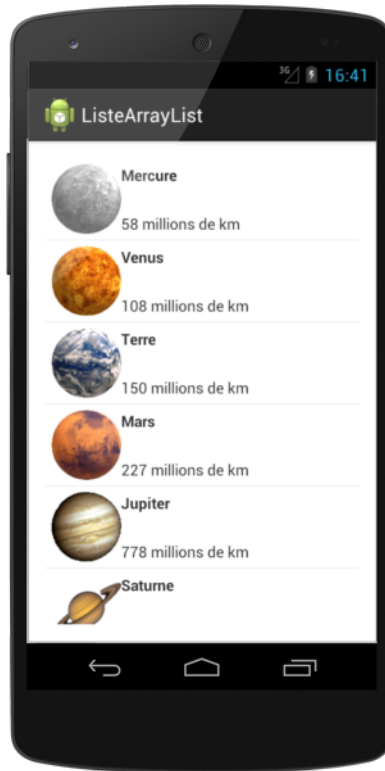


Figure 28: Liste d'items

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` pour lesquelles `PlaneteView.findViews` récupère les objets Java.
3. L'adaptateur appelle la méthode `setItem` du `PlaneteView` avec les données à afficher.
 - a. `PlaneteView.setItem` appelle `setText` des vues pour afficher les valeurs.

4.4.14. Le résultat

figure 28

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](#).


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();
```

Si on néglige de le faire, alors la liste affichée à l'écran ne changera pas.

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 idvue)
{
    // gérer un clic sur l'item identifié par position
    ...
}
```

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

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`.



```
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 = findViewById(android.R.id.list);
        lv.setOnItemClickListener(this);
    }
}
```

4.5.3. Clic sur un élément, fin

Et voici sa méthode `onItemClick` : 

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

Figure 29: Éléments cochables

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

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

4.5.4. Liste d'éléments cochables

Android offre des listes cochables comme celles-ci :

figure 29

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

4.5.5. Liste cochable simple

Android propose un layout prédéfini pour items cochables :



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

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

```
lv.setChoiceMode(ListView.CHOICE_MODE_SINGLE);  
}
```

4.5.6. 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 `onItemClickListener` est appelée sur chaque élément cliqué.

4.5.7. 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).

4.6. RecyclerView

4.6.1. Présentation

Ce type de vue affiche également une liste, comme un `ListView`. Il s'appuie également sur un adaptateur, et repose sur un mécanisme très similaire à la classe `PlaneteView` qu'on appelle *view holder*.

Les concepts du `RecyclerView` sont repris dans de nombreux autres contrôles Android : les boîtes à onglets, etc.

NB: cette vue appartient à la bibliothèque *support*, voir au prochain cours. Il faut rajouter ceci dans le `build.gradle` du projet :

```
implementation 'com.android.support:recyclerview-v7:27.1.1'
```

NB: le numéro de version correspond au SDK installé à l'IUT.

4.6.2. Principes

Comme pour un `ListView`, vous devez définir deux layouts :

- `main.xml` pour l'activité. Mettez `RecyclerView` au lieu de `ListView`
- `item.xml` pour les items de la liste.

Comme précédemment, la méthode `onCreate` de l'activité crée un adaptateur pour les données et l'associe au `RecyclerView`. Ce qui change, c'est l'adaptateur `PlaneteAdapter` et la classe liée aux items `PlaneteView`.

4.6.3. Méthode onCreate de l'activité



```
public class MainActivity extends AppCompatActivity {  
    @Override protected void onCreate(Bundle savedInstanceState) {  
        // mettre en place l'interface  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main);  
        // données  
        List<Planete> liste = new ArrayList<>();  
        ...  
        // mettre en place la liste  
        RecyclerView recyclerView = findViewById(R.id.list);  
        recyclerView.setAdapter(new PlaneteAdapter(liste));  
        recyclerView.setHasFixedSize(true);  
    }  
}
```

4.6.4. Disposition des éléments

Pour un affichage correct des éléments, il manque quelques instructions. Il faut spécifier un gestionnaire de mise en page :



```
import android.support.v7.widget.LinearLayoutManager;  
  
...  
LinearLayoutManager llm = new LinearLayoutManager(this);  
recyclerView.setLayoutManager(llm);
```

On peut indiquer l'axe de défilement de la liste :



```
LinearLayoutManager llm =  
    new LinearLayoutManager(this,  
        LinearLayoutManager.HORIZONTAL);
```

4.6.5. Mise en grille

Au lieu d'un LinearLayoutManager, on peut créer un GridLayoutManager qui fait une grille d'un certain nombre de colonnes indiqué en paramètre :



```
import android.support.v7.widget.GridLayoutManager;  
  
...  
GridLayoutManager glm = new GridLayoutManager(this, 2);  
recyclerView.setLayoutManager(glm);  
recyclerView.setHasFixedSize(true);
```

On peut aussi indiquer l'axe de défilement de la liste :



```
GridLayoutManager glm =  
    new GridLayoutManager(this,  
        2,  
        LinearLayoutManager.HORIZONTAL);
```

4.6.6. Séparateur entre items

Par défaut, un RecyclerView n'affiche pas de ligne de séparation entre les éléments. Pour en avoir :



```
import android.support.v7.widget.DividerItemDecoration;  
  
...  
DividerItemDecoration dividerItemDecoration =  
    new DividerItemDecoration(recyclerView.getContext(),  
        llm.getOrientation());  
recyclerView.addItemDecoration(dividerItemDecoration);
```

Voir [ce lien sur stackoverflow](#) pour un séparateur dans une grille.

4.6.7. Adaptateur de RecyclerView

Voici la nouvelle définition pour un RecyclerView :



```
public class PlaneteAdapter  
    extends RecyclerView.Adapter<PlaneteView>  
{  
    private final List<Planete> mListe;  
  
    PlaneteAdapter(List<Planete> liste)  
    {  
        mListe = liste;  
    }  
}
```

C'est une classe qui hérite de `RecyclerView.Adapter`. Son constructeur mémorise la liste des données dans une variable privée.

Parmi les méthodes à définir, il faut surcharger la méthode `getItemCount` ; elle retourne le nombre d'éléments de la collection :



```
@Override  
public int getItemCount()  
{  
    return mListe.size();  
}
```

Dans le cas plus général où les données ne sont pas stockées dans une liste, cela peut entraîner un calcul plus complexe.

Deux autres méthodes doivent être surchargées, voici la première :



```
@Override
public void onBindViewHolder(PlaneteView itemView, int pos)
{
    itemView.setItem(mListe.get(pos));
}
```

Son rôle est de remplir un `PlaneteView` avec la donnée spécifiée par la position. La classe `PlaneteView` est légèrement différente de la version pour `ListView`, voir plus loin.

La dernière méthode à surcharger est très simple :



```
@Override
public PlaneteView onCreateViewHolder(
    ViewGroup parent, int viewType)
{
    LayoutInflater li =
        LayoutInflater.from(parent.getContext());
    View itemView =
        li.inflate(R.layout.item, parent, false);
    return new PlaneteView(itemView);
}
```

Le rôle de cette méthode est le même que `create` d'un `PlaneteView` page 83. Elle expose les vues pour afficher un item. La réutilisation d'un item est gérée automatiquement par Android.


4.6.8. Affichage d'un item `PlaneteView`

C'est une variante de celle du `ListView`, les explications sont après :



```
class PlaneteView extends RecyclerView.ViewHolder {
    private TextView tvNom;
    ...
    PlaneteView(View view) {
        super(view);
        findViews(view);
    }
    private void findViews(View view) {
        tvNom = view.findViewById(R.id.item_nom);
        ...
    }
    void setItem(final Planete planete) {
        ...
    }
}
```


- Le constructeur est appelé par `onCreateViewHolder` de l'adaptateur. Il reçoit une `View` en paramètre, c'est la racine du layout d'item, donc un `RelativeLayout` ici.
- La méthode `findViews` permet d'associer des objets Java aux vues du layout d'item.

- Il devient donc inutile de mentionner une classe perso dans le layout d'item. Ce layout redevient « normal » : 

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout ...>
    <ImageView android:id="@+id/item_image" .../>
    <TextView  android:id="@+id/item_nom" .../>
    <TextView  android:id="@+id/item_distance" .../>
</RelativeLayout>
```


4.7. Réponses aux sélections d'items

4.7.1. Clics sur un RecyclerView

Petit [problème](#) : il n'y a pas d'écouteur pour les clics sur les éléments, comme avec un `ListView`. Une solution simple passe par la méthode `onBindViewHolder` de l'adaptateur : 

```
@Override
public void onBindViewHolder(PlaneteView holder, final int pos)
{
    final Planete planete = mPlanetes.get(pos);
    holder.setPlanete(planete);
    holder.itemView.setOnClickListener(
        new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // TODO l'action à faire sur l'item n° pos
            }
        });
}
```

4.7.2. Afficher la sélection

Pour mettre un item en évidence dans un `RecyclerView`, il faut gérer la position désignée par un clic : 

```
public class PlaneteAdapter extends RecyclerView.Adapter<PlaneteView>
{
    ...
    int clickedPosition = RecyclerView.NO_POSITION;

    public int getClickedPosition() {
        return clickedPosition;
    }
    private void setClickedPosition(int pos) {
        notifyItemChanged(clickedPosition);
        clickedPosition = pos;
    }
}
```

```
        notifyItemChanged(clickedPosition);  
    }
```

Ensuite, on modifie `onBindViewHolder` :



```
public void onBindViewHolder(PlaneteView holder, final int pos)  
{  
    ...  
    holder.itemView.setOnClickListener(  
        new View.OnClickListener() {  
            public void onClick(View v) {  
                setClickedPosition(pos);  
                ...  
            }  
        });  
    // accentuer la sélection  
    holder.itemView.setBackgroundColor(  
        getClickedPosition() == pos ?  
        Color.LTGRAY : Color.TRANSPARENT);  
}
```

4.7.3. Ouf, c'est fini

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



Figure 30: Barre d'action

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 30

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 :



Figure 31: Icônes de menus

- 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*, devenue *androidx* récemment, voir la fin du cours).

5.1.4. Icônes pour les menus

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

figure 31

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. 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)
{
    // ajouter mes items de menu
    getMenuInflater().inflate(R.menu.nom_du_menu, menu);
    // ajouter les items du système s'il y en a
    return super.onCreateOptionsMenu(menu);
}
```

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.

5.1.6. 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);
    }
}
```

5.1.7. 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>
```

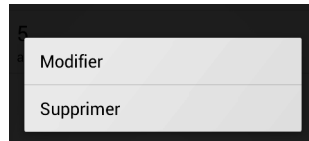



Figure 32: MenuContextuel

5.1.8. Menus contextuels

figure 32

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.

5.1.9. 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 = findViewById(android.R.id.list);
    registerForContextMenu(lv);
    ...
}
```

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



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

5.1.10. Callback d'affichage du menu

Un clic long sur un élément de la liste 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.xml`.

5.1.11. 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.

5.1.12. Menu contextuel pour un RecyclerView

Dans le cas d'un `RecyclerView`, c'est un peu plus compliqué car le menu ne se déclenche pas. Il faut ajouter un écouteur à l'adaptateur :



```
public class PlaneteAdapter extends RecyclerView.Adapter<...> {
    ...
    View.OnCreateContextMenuListener mMenuListener;

    public void setMenuListener(
        View.OnCreateContextMenuListener listener) {
        mMenuListener = listener;
    }
}
```

Cet écouteur est affecté lors de la création de l'adaptateur, et en général, c'est l'activité.

La suite est dans le `ViewHolder`, voir le cours précédent :



```
class PlaneteView extends RecyclerView.ViewHolder {
    ...findViews()...setItem()...onCreateViewHolder()...

    @Override
    public void onBindViewHolder(final PlaneteView h, final int pos)
    {
        // définir un écouteur pour les clics longs avec une lambda
        h.itemView.setOnLongClickListener(view -> {
            setClickedPosition(pos);
            return false;
        });
    }
}
```

```
// définir un écouteur pour le menu contextuel (clic long)
h.itemView.setOnCreateContextMenuListener(mMenuListener);
}
```

5.1.13. Parenthèse sur les *lambda* Java

Une expression *lambda* est une sorte d'objet-méthode sans nom, par exemple un écouteur anonyme sur un bouton.

Au lieu d'écrire ceci :

```
btn.setOnClickListener(
    new View.OnClickListener() {
        public void onClick(View v) {
            // faire quelque chose
        }
    });
```

Java 8 permet de simplifier en :

```
btn.setOnClickListener(v -> {
    // faire quelque chose
});
```

5.1.14. Parenthèse sur les *lambda* Java, fin

On ne laisse que les noms des paramètres, sans parenthèses s'il est seul, et le corps. On peut même supprimer les `{}` s'il n'y a qu'une seule instruction.

La syntaxe *lambda*, paramètre -> instruction ou (paramètres) -> { instructions; } est possible lorsqu'il s'agit de la seule méthode d'une interface :

```
public classe View ... {
    public interface OnClickListener {
        void onClick(View v);
    }
}
```

Dans ce cas, on peut créer un objet anonyme, qui consiste uniquement en un appel à `onClick`.

5.1.15. Menu contextuel pour un RecyclerView, fin

Enfin, on modifie l'écouteur du menu dans l'activité :



```
@Override
public boolean onContextItemSelected(MenuItem item)
{
    int position = adapter.getClickedPosition();
}
```

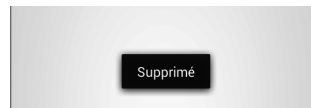


Figure 33: Toast

```
switch (item.getItemId()) {  
    ...  
}  
return false;  
}
```

NB: ce n'est encore pas une bonne solution quand les items ont un identifiant et non une position, voir le TP sur Realm.

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.

figure 33

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`.

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 :



```
// expander 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();
```

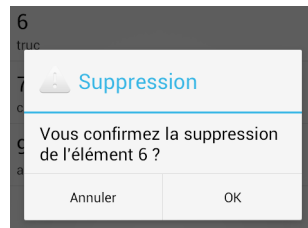


Figure 34: Dialogue d'alerte

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 34

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 :

```
// import android.support.v7.app.AlertDialog;

AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Suppression");
builder.setIcon(android.R.drawable.ic_dialog_alert);
builder.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
builder.setPositiveButton(android.R.string.yes,
    // écouteur écrit sous la forme d'une lambda
    (dialog, idbtn) -> {
        SupprimerElement(idElement);
    });
// rajouter un bouton "non" qui ne fait rien
```

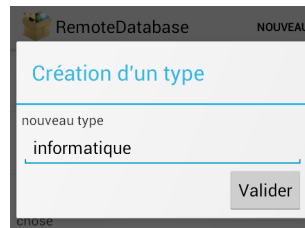


Figure 35: Dialogue perso

```
builder.setNegativeButton(android.R.string.no, null);  
// affichage du dialogue  
builder.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 `builder.setItems(R.array.notes, écouteur)`. L'écouteur est presque le même que pour un clic, sauf qu'il reçoit le numéro du choix en 2^e paramètre `idBtn`.

Dans ce cas, ne pas appeler `builder.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 35

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/dlg_titre" .../>
```

```
<EditText android:id="@+id/dlg_libelle" .../>
<Button android:id="@+id/dlg_btn_valider" ... />
</LinearLayout>
```

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

5.2.9. Affichage du dialogue



```
// créer le dialogue (final car utilisé dans les écouteurs)
final Dialog dialog = new Dialog(this);
dialog.setView(R.layout.edit_dialog);
dialog.setTitle("Création d'un type");
// bouton valider
Button btnValider = dialog.findViewById(R.id.dlg_btn_valider);
btnValider.setOnClickListener(v -> {
    // récupérer et traiter les infos
    ...
    // ne surtout pas oublier de fermer le dialogue
    dialog.dismiss();
});
// 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.

Voir la figure 36, page 104.

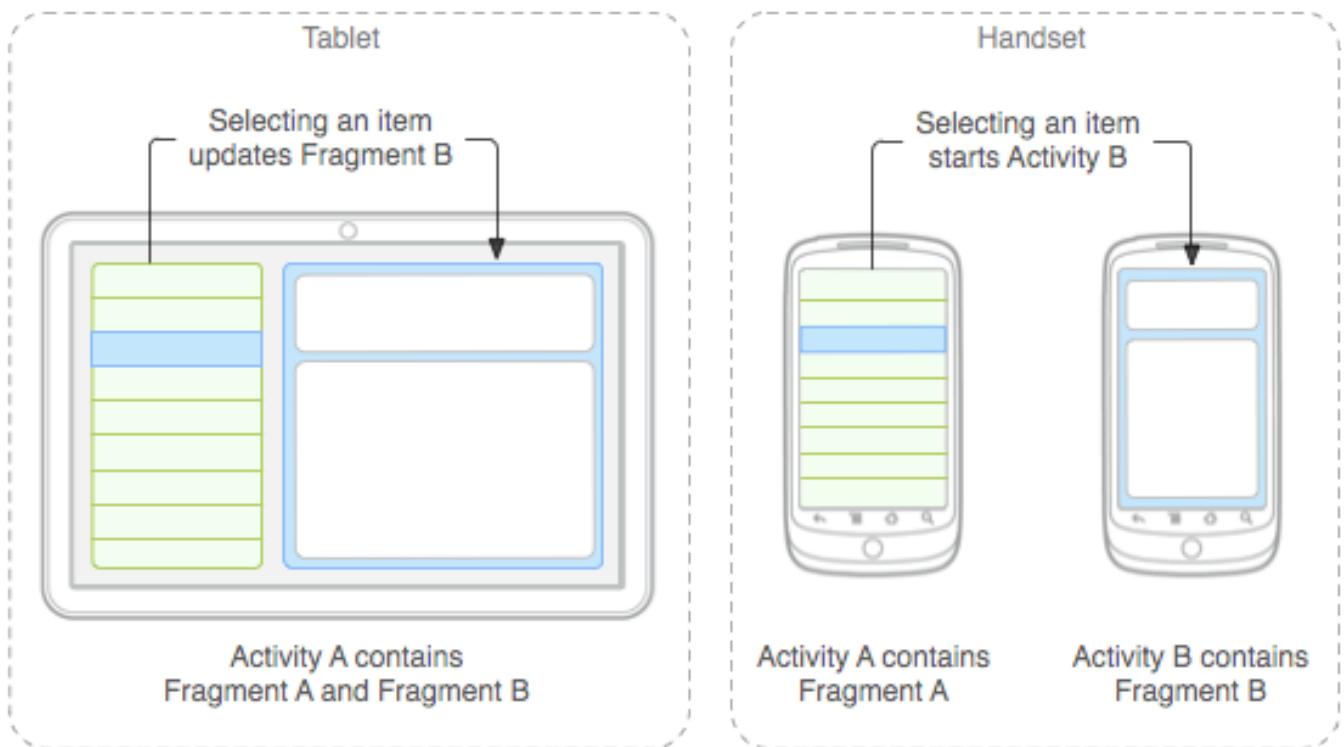



Figure 36: Différentes apparences

5.3.3. Structure d'un fragment

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

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

    @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.

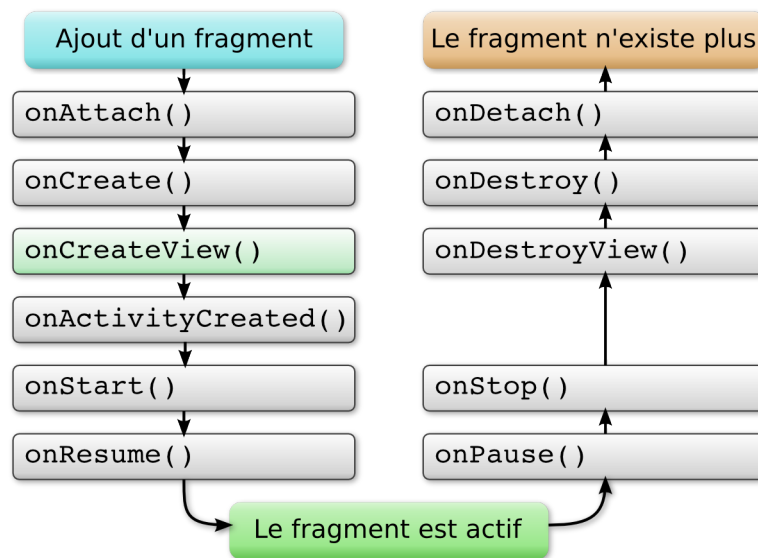


Figure 37: Cycle de vie d'un fragment

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 37

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
    return inflater.inflate(android.R.layout.list_content,
        container, false);
}
```

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);
}
```

5.3.7. 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)`;

5.3.8. 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.

5.3.9. 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.

5.3.10. 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.

5.3.11. 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
FragmentTransaction 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.

5.3.12. 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 »).

Voir la figure 38, page 108.

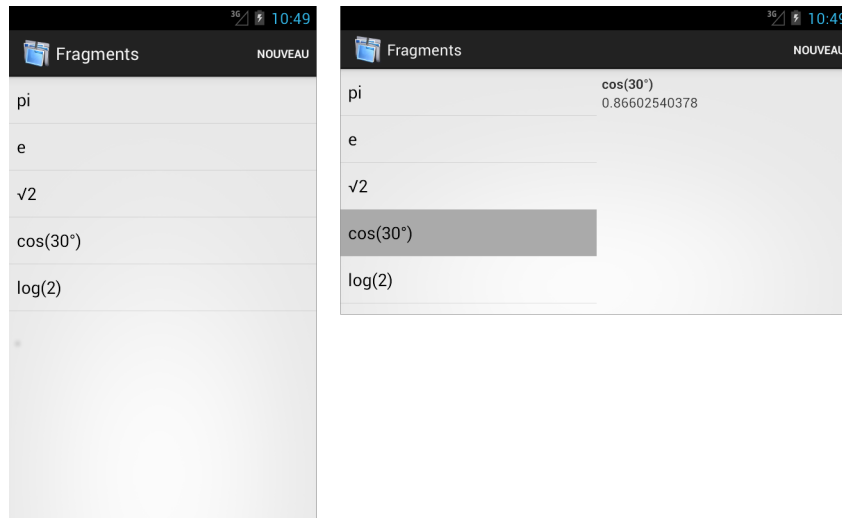


Figure 38: Un ou deux fragments affichés

5.3.13. 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>
```

5.3.14. 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 = getSupportFragmentManager();
    InfosFragment frag_infos = (InfosFragment)
        manager.findFragmentById(R.id.frag_infos);
```

```
if (frag_infos != null) {  
    // le fragment des informations est présent  
    ...  
}
```

5.3.15. 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.

5.3.16. 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;  
}
```

5.3.17. 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.

5.3.18. 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);  
    }  
}
```

5.3.19. 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);  
    }  
}
```

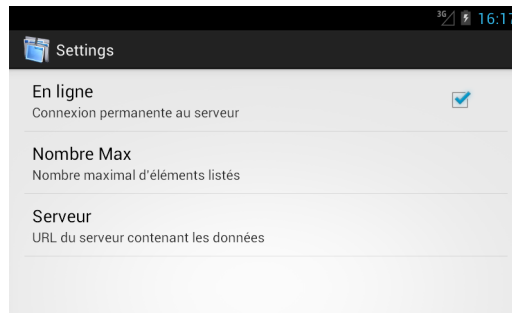


Figure 39: Préférences de l'application

5.3.20. À méditer, partie 2

Une 2^e 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.

figure 39

5.4.2. Présentation

Il y a deux concepts mis en jeu :

- 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](#). Les choses changent beaucoup d'une version à l'autre d'API.

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, télévisions, voitures, 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="28" />
```

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 28 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 grâce à des fichiers *jar* supplémentaires qui rajoutent les fonctionnalités manquantes.

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. Anciennes versions de l'Android Support Library

Il en existait plusieurs variantes, selon l'ancienneté qu'on visait. 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'était la plus grosse API, elle permettait 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. Une seule pour les gouverner toutes

Comme vous le constatez, il y avait une multitude d'API, pas vraiment cohérentes entre elles, et avec des contenus assez imprévisibles. Depuis juin 2018, une seule API remplace tout cet attirail, *androidx*. Elle définit un seul espace de packages, `androidx.*` pour tout.

- Dans `app/build.gradle`, par exemple :

```
implementation "androidx.appcompat:appcompat:1.0.2"
implementation "androidx.recyclerview:recyclerview:1.1.0"
```

5.5.6. Une seule pour les gouverner toutes, fin

- Dans les layouts, par exemple :

```
<androidx.recyclerview.widget.RecyclerView .../>
```

- Dans les imports, par exemple :

```
import androidx.annotation.NonNull;
import androidx.recyclerview.widget.RecyclerView;
```

Il reste à connaître les packages, mais on les trouve dans la documentation.

5.5.7. Mode d'emploi

La première chose à faire est de définir le niveau de SDK minimal nécessaire, `minSdkVersion`, à mettre dans le `app/build.gradle`. Il semble qu'il faille actuellement mettre 16 :

```
android {  
    compileSdkVersion 29  
  
    defaultConfig {  
        applicationId "mon.package"  
        minSdkVersion 16  
        targetSdkVersion 29  
    }  
}
```

Ensuite, il faut ajouter les dépendances :

```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    ...  
    implementation "androidx.appcompat:appcompat:1.0.2"  
    implementation "androidx.recyclerview:recyclerview:1.1.0"  
    ...  
}
```

On rajoute les éléments nécessaires. Il faut aller voir la documentation de chaque chose employée pour savoir quelle dépendance rajouter, et vérifier son numéro de version pour avoir la dernière.

5.5.8. Programmation

Enfin, il suffit de faire appel à ces classes pour travailler. Elles sont par exemple dans le package [androidx.fragment.app](#).

```
import androidx.fragment.app.FragmentActivity;  
import androidx.recyclerview.widget.RecyclerView;  
  
public class MainActivity extends FragmentActivity  
    ...
```

Il y a quelques particularités, comme une classe `AppCompatButton` qui est employée automatiquement à la place de `Button` dans les activités du type `AppCompatActivity`. Le mieux est d'étudier les documentations pour arriver à utiliser correctement tout cet ensemble.

5.5.9. 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

Realm

Le cours de cette semaine va vous apprendre à stocker des informations dans un SGBD appelé Realm. Ce système utilise de simples objets Java pour représenter les n-uplets et offre de nombreuses possibilités d'interrogation.

- Principes
- Modèles de données
- Requêtes
- Adaptateurs

NB: à l'IUT on utilisera une version un peu ancienne, la 3.3.2, ayant moins de possibilités que l'actuelle.

Avant tout, on va commencer par un *plugin* pour AndroidStudio bien pratique, Lombok.

6.1. Plugin Lombok

6.1.1. Présentation

Le [plugin Lombok](#), c'est son nom, regroupe un ensemble de fonctions qui se déclenchent lors de la compilation d'un source Java. Elles effectuent des transformations utiles sur le source.

Pour cela, on rajoute des *annotations* sur la classe ou sur les variables membres.

Une annotation Java est un mot clé commençant par un @. Il déclenche une méthode dans ce qu'on appelle un *Annotation Processor*. La méthode fait certaines vérifications ou génère du source Java d'après votre programme.

@Override, @SuppressWarnings("unused"), @Nullable, @NonNull sont des annotations prédéfinies dans Java (package `android.support.annotation`). Lombok en ajoute d'autres.

6.1.2. Exemple

Voici comment générer automatiquement les [setters et getters](#) et la méthode `toString()` :




```
import lombok.*;

@ToString @Setter @Getter
public class Personne
{
    private int id;
    private String nom;
```

```
private String prenom;  
}
```

L'avantage principal est que la classe reste très facilement lisible, et il y a pourtant toutes les méthodes. La méthode générée `toString()` affiche `this` proprement : `Personne(id=3, nom="Nerzic", prenom="Pierre")`.

6.1.3. Placement des annotations

On peut placer les annotations `@Setter` et `@Getter` soit au niveau de la classe, soit seulement sur certaines variables. 

```
@ToString  
@Getter  
public class Personne  
{  
    private int id;  
    @Setter private String nom;  
    @Setter private String prenom;  
}
```

On aura un *getter* pour chaque variable et un *setter* seulement pour le nom et le prénom.

6.1.4. Nommage des champs

Lombok est adapté à un nommage simple des champs, en écriture de chameaux [lowerCamelCase](#) et non pas en écriture hongroise. Il génère les *setters* et *getters* en ajoutant `set` ou `get` devant le nom, avec sa première lettre mise en majuscule.

- `private int nbProduits;` génère
 - `public void setNbProduits(int n)`
 - `public int getNbProduits()`

Dans le cas des booléens, le getter commence par `is`

- `private boolean enVente;` génère
 - `public setEnVente(boolean b)`
 - `public boolean isEnVente()`

6.1.5. Installation du plugin

Voir [cette page](#).

Il faut passer par les *settings* d'Android Studio, onglet **Plugins**, il faut chercher *Lombok Plugin* dans la liste.

À l'IUT, cause internet coupé, il faudra installer le plugin à l'aide d'un fichier zip. Voir le TP6.

Ensuite, dans chaque projet, il faut rajouter deux lignes dans le fichier `app/build.gradle` : 

```
annotationProcessor 'org.projectlombok:lombok:1.18.12'  
implementation 'org.projectlombok:lombok:1.18.12'
```

6.2. Realm

6.2.1. Définition de Realm

Realm est un mécanisme permettant de transformer de simples classes Java en sortes de tables de base de données. La base de données est transparente, cachée dans le logiciel. Le SGBD tient à jour la liste des instances et permet de faire l'équivalent des requêtes SQL.

C'est ce qu'on appelle un ORM (*object-relational mapping*).

Chaque instance de cette classe est un n-uplet dans la table. Les variables membres sont les attributs de la table.

Realm est très bien expliqué sur [ces pages](#). Ce cours suit une partie de cette documentation.

6.2.2. Realm contre les autres ORM

Realm présente plusieurs avantages :

- Il est multi-plateformes : Android (Java, Kotlin), iOS et Mac (Swift, Objective-C), JavaScript (Node.js, React Native) et .NET. Les mêmes bases sont accessibles de la même manière.
- Les bases Realm sont « vivantes », c'est à dire que des modifications faites par l'un des utilisateurs sont transmises à tous les autres qui sont connectés, et cette transmission est très efficace. Donc chaque utilisateur voit les changements en temps réel, quelque soit sa plateforme.
- Il y a un outil d'édition des bases Realm : [RealmStudio](#).

6.2.3. Configuration d'un projet Android avec Realm

Avant toute chose, quand on utilise Realm, il faut éditer les deux fichiers `build.gradle` :


- celui du projet, ajouter sous l'autre `classpath` :

```
classpath 'io.realm:realm-gradle-plugin:6.1.0'
```

- celui du dossier app :

```
apply plugin: 'realm-android'  
android {  
    ...  
}  
dependencies {  
    implementation 'io.realm:android-adapters:3.1.0'  
    ...  
}
```

6.2.4. Initialisation d'un Realm par l'application

Vous devez placer ces instructions dans la méthode `onCreate` d'une sous-classe d'`Application` associée à votre logiciel : 

```
import io.realm.Realm;
public class MyApplication extends Application
{
    @Override
    public void onCreate()
    {
        super.onCreate();
        Realm.init(this);
    }
}
```

avec `<application android:name=".MyApplication"...` dans le manifeste, cf cours 3.

6.2.5. Ouverture d'un Realm dans chaque activité

Ensuite, dans chaque activité, on rajoute ceci :



```
public class MainActivity extends Activity
{
    private Realm realm;

    @Override
    public void onCreate()
    {
        super.onCreate();
        ...
        realm = Realm.getDefaultInstance();
        ...
    }
}
```

6.2.6. Fermeture du Realm

Il faut également fermer le Realm à la fin de l'activité :



```
public class MainActivity extends Activity
{
    ...

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

On peut définir une classe `RealmActivity` qui hérite de `Activity` et qui effectue ces deux opérations. On n'a donc plus qu'à hériter de `RealmActivity` sans se soucier de rien.

6.2.7. Autres modes d'ouverture du Realm

La méthode précédente ouvre un Realm local au smartphone. Les données seront persistantes d'un lancement à l'autre, mais elles ne seront pas enregistrées à distance.

Il existe également des Realm distants, hébergés dans le cloud de l'entreprise Realm pour 30\$ par mois. On les appelle des *Realms synchronisés*. Cela impose l'établissement d'une connexion et d'une authentification de l'utilisateur.

Il existe également des Realms en mémoire, pratiques pour faire des essais. L'ouverture se fait ainsi :



```
RealmConfiguration conf =  
    new RealmConfiguration.Builder().inMemory().build();  
realm = Realm.getInstance(conf);
```

6.3. Modèles de données Realm

6.3.1. Définir une table

Pour cela, il suffit de faire dériver une classe de la superclasse `RealmObject` :



```
import io.realm.RealmObject;  
  
public class Produit extends RealmObject  
{  
    private int id;  
    private String designation;  
    private String marque;  
    private float prixUnitaire;  
}
```

Cela a automatiquement créé une table de `Produit` à l'intérieur de Realm. Par contre, cette table n'est pas visible en tant que telle, voir [RealmStudio](#) pour un outil d'édition de la base.

6.3.2. Table Realm et Lombok

On peut employer le plugin Lombok :



```
@ToString @Getter @Setter  
public class Produit extends RealmObject  
{  
    private int id;  
    private String designation;  
    private String marque;  
    private float prixUnitaire;  
}
```

Elle reste donc extrêmement lisible. On peut se concentrer sur les algorithmes.

NB: dans la suite, l'emploi du plugin Lombok sera implicite.

6.3.3. Types des colonnes

Realm gère tous les types Java de base : `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `String`, `Date` et `byte[]`.

Il faut savoir qu'en interne, tous les entiers `byte`, `short`, `int`, `long` sont codés en interne par un `long`.

Il y a aussi tous les types Java emballés (*boxed types*) : `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float` et `Double`. Ceux-là peuvent tous avoir la valeur `null`.

6.3.4. Empêcher les valeurs `null`

Si une variable objet ne doit pas être indéfinie, rajoutez l'annotation `@Required` :



```
import io.realm.annotations.Required;

public class Produit extends RealmObject
{
    private int id;
    @Required private String designation;
    private String marque;
    private float prixUnitaire;
}
```

Si vous tentez d'affecter `designation` avec `null`, ça déclenchera une exception. On ne peut hélas pas placer cette annotation sur un autre type d'objet, voir la [doc de @Required](#).

6.3.5. Définir une clé primaire

Dans l'exemple précédent, il y a une variable d'instance `id` qui est censée contenir un entier unique. Pour le garantir dans Realm, il faut l'annoter avec `@PrimaryKey` :



```
import io.realm.annotations.PrimaryKey;

public class Produit extends RealmObject
{
    @PrimaryKey private long id;
    private String designation;
    private String marque;
    private float prixUnitaire;
}
```

Si vous tentez d'affecter à `id` la même valeur pour deux produits différents, vous aurez une exception.

6.3.6. Définir une relation simple


En Realm, une relation est simplement une référence sur un autre objet Realm :



```
public class Achat extends RealmObject
{
    private Produit produit;
    private Date date;
    private int nombre;
}
```

La variable `produit` pourra soit contenir `null`, soit désigner un objet. Malheureusement, on ne peut pas (encore) appliquer `@Required` sur une référence d'objet.

6.3.7. Relation multiple

Dans le transparent précédent, un `Achat` n'est associé qu'à un seul `Produit`. Si on en veut plusieurs, il faut utiliser une collection `RealmList<Type>` : 

```
public class Achat extends RealmObject
{
    private RealmList<Produit> produits;
    private Date date;
}
```

Le champ `produit` est relié à plusieurs produits.

Le type `RealmList` possède les méthodes des `List` et `Iterable` entre autres : `add`, `clear`, `get`, `contains`, `size`...

6.3.8. Migration des données

Un point crucial à savoir : quand vous changez le modèle des données, il faut prévoir une procédure de migration des données. C'est à dire une sorte de recopie et d'adaptation au nouveau modèle. Cette migration est automatique quand vous ne faites qu'ajouter de nouveaux champs aux objets, mais vous devez la programmer si vous supprimez ou renommez des variables membres.

6.4. Création de n-uplets

6.4.1. Résumé

Un n-uplet est simplement une instance d'une classe qui hérite de `RealmObject`.

Cependant, les variables Java ne sont pas enregistrées de manière permanente dans Realm. Il faut :

- soit créer les instances avec une méthode de fabrique de leur classe,
- soit créer les instances avec `new` puis les ajouter dans Realm.

Dans tous les cas, il faut initialiser un Realm au début de l'application.

6.4.2. Création de n-uplets par `createObject`

Une fois le realm ouvert, voici comment créer des instances. Dans Realm, toute création ou modification de données doit être faite dans le cadre d'une transaction, comme de préférence dans SQL : 

```
realm.beginTransaction();
Produit produit = realm.createObject(Produit.class, 1L);
produit.setDesignation("brosse à dent");
produit.setPrixUnitaire(4.95);
realm.commitTransaction();
```

Le second paramètre de `createObject` est l'identifiant, obligatoire s'il y a une `@PrimaryKey`.

Si vous oubliez de placer les modifications dans une transaction, vous aurez une `IllegalStateException`.

6.4.3. Création de n-uplets par new

Une autre façon de créer des n-uplets consiste à utiliser `new` puis à enregistrer l'instance dans realm :



```
Produit produitJava = new Produit();
produitJava.setId(1L);
produitJava.setDesignation("brosse à dent");
produitJava.setPrixUnitaire(4.95);

realm.beginTransaction();
Produit produitRealm = realm.copyToRealm(produitJava);
realm.commitTransaction();
```

Le problème est que ça crée deux objets, or seul le dernier est connu de Realm.

6.4.4. Modification d'un n-uplet

Là également, il faut placer tous les changements dans une transaction :



```
realm.beginTransaction();
produit.setDesignation("fil dentaire");
produit.setPrixUnitaire(0.95);
realm.commitTransaction();
```

Si vous oubliez de placer les modifications dans une transaction, vous aurez une `IllegalStateException`.

D'autre part, contrairement à SQL, vous ne pourrez pas modifier la clé primaire d'un objet après sa création.

6.4.5. Suppression de n-uplets

On ne peut supprimer que des objets gérés par realm, issus de `createObject` ou `copyToRealm` : 

```
realm.beginTransaction();
produit.deleteFromRealm();
realm.commitTransaction();
```

Il est possible de supprimer plusieurs objets, en faisant appel à une requête.

6.5. Requetes sur la base

6.5.1. Résumé

Une requête Realm retourne une collection d'objets (tous appartenant à Realm) sous la forme d'un `RealmResults<Type>`.

Ce qui est absolument magique, c'est que cette collection est automatiquement mise à jour en cas de changement des données, voir [auto-updating results](#).

Par exemple, si vous utilisez un Realm distant, tous les changements opérés ailleurs vous seront transmis. Ou si vous avez deux fragments, une liste et un formulaire d'édition : tous les changements faits par le formulaire seront répercutés dans la liste sans aucun effort de votre part.

Les `RealmResults` ne sont donc pas comme de simples `ArrayList`. Ils sont vivants, grâce à des écouteurs transparents pour vous.

6.5.2. Sélections

La requête la plus simple consiste à récupérer la liste de tous les n-uplets d'une table :



```
RealmResults<Produit> results = realm
    .where(Produit.class)
    .findAll();
```

Cette manière d'écrire des séquences d'appels à des méthodes Java s'appelle *désignation chaînée*, *fluent interface* en anglais.

La classe du `RealmResults` doit être la même que celle fournie à `where`.

En fait, la méthode `where` est très mal nommée. Elle aurait due être appelée `from`.

6.5.3. Conditions simples

Comme avec SQL, on peut rajouter des conditions :



```
RealmResults<Produit> results = realm
    .where(Produit.class)
    .equalTo("designation", "dentifrice")
    .findAll();
```

La méthode `equalTo(nom_champ, valeur)` définit une condition sur le champ fourni par son nom et la valeur. D'autres comparaisons existent, voir [cette page](#) :

- pour tous les champs : `equalTo`, `notEqualTo`, `in`
- pour les nombres : `between`, `greaterThan`, `lessThan`, `greaterThanOrEqualTo`, `lessThanOrEqualTo`
- pour les chaînes : `contains`, `beginsWith`, `endsWith`, `like`

6.5.4. Nommage des colonnes

Cet exemple montre le gros défaut, à mon sens, de Realm : on doit nommer les champs à l'aide de chaînes. Si on se trompe sur le nom de champ, ça cause une exception. Une telle erreur ne peut être détectée qu'à l'exécution, il serait mieux de la voir à la compilation.

Il est donc très conseillé de faire ceci :



```
public class Produit extends RealmObject
{
    public static final String FIELD_ID = "id";
    public static final String FIELD_DESIGNATION = "designation";
    public static final String FIELD_MARQUE = "marque";
    ...
    private int id;
    private String designation;
    private String marque;
    ...
}
```

6.5.5. Librairie *RealmFieldNamesHelper*

[RealmFieldNamesHelper](#) est une bibliothèque très utile qui fait ce travail automatiquement. Il faut juste rajouter ceci dans la partie dépendances du `app/build.gradle` :



```
annotationProcessor 'dk.ilios:realmfieldnameshelper:1.1.1'
```

Pour chaque CLASSE de type `RealmObject`, ça génère automatiquement une classe appelée `CLASSEFields` contenant des chaînes constantes statiques du nom des champs de la classe :



```
public final class ProduitFields {
    public static final String ID = "id";
    public static final String DESIGNATION = "designation";
    public static final String MARQUE = "marque";
    ...
};
```

6.5.6. Conjonctions de conditions


Une succession de conditions réalise une conjonction :



```
RealmResults<Produit> results = realm
    .where(Produit.class)
    .notEqualTo(ProduitFields.ID, 0)
    .equalTo(ProduitFields.DESIGNATION, "dentifrice")
    .equalTo(ProduitFields.MARQUE, "Whiteeth")
    .lessThanOrEqualTo(ProduitFields.PRIX, 3.50)
    .findAll();
```


L'emploi du *RealmFieldNamesHelper* alourdit un peu l'écriture, mais on est certain que le programme fonctionnera, ou alors ne se compilera pas si on change le schéma de la base.

6.5.7. Disjonctions de conditions

Voici un exemple de disjonction avec `beginGroup` et `endGroup` qui jouent le rôle de parenthèses : 

```
RealmResults<Produit> results = realm
    .where(Produit.class)
    .beginGroup()
        .equalTo(ProduitFields.DESIGNATION, "dentifrice")
        .lessThanOrEqualTo(ProduitFields.PRIX, 3.50)
    .endGroup()
    .or()
    .beginGroup()
        .equalTo(ProduitFields.DESIGNATION, "fil dentaire")
        .lessThanOrEqualTo(ProduitFields.PRIX, 8.50)
    .endGroup()
    .findAll();
```


6.5.8. Négations

La méthode `not()` permet d'appliquer une négation sur la condition qui la suit : 

```
RealmResults<Produit> results = realm
    .where(Produit.class)
    .not()
    .beginGroup()
        .equalTo(ProduitFields.DESIGNATION, "dentifrice")
        .or()
        .equalTo(ProduitFields.DESIGNATION, "fil dentaire")
    .endGroup()
    .findAll();
```

On peut évidemment la traduire en conjonction de conditions opposées.

6.5.9. Classement des données

On peut trier sur l'un des champs : 

```
RealmResults<Produit> results = realm
    .where(Produit.class)
    .sort(ProduitFields.PRIX, Sort.DESCENDING)
    .findAll();
```

Le second paramètre de `sort` est optionnel. S'il est absent, c'est un tri croissant sur le champ indiqué en premier paramètre.

6.5.10. Agrégation des résultats

La méthode `findAll()` retourne tous les objets vérifiant ce qui précède. C'est en fait cette méthode qui retourne le `RealmResults`.

On peut aussi utiliser `findFirst()` pour n'avoir que le premier. Dans ce cas, on ne récupère pas un `RealmResults` mais un seul objet.

Pour les champs numériques, il existe aussi `min(nom_champ)`, `max(nom_champ)` et `sum(nom_champ)` qui retournent seulement une valeur du même type que le champ.

6.5.11. Jointures 1-1

On arrive au plus intéressant, mais hélas frustrant comparé à ce qui est possible avec SQL. On reprend cette classe :

```
public class Achat extends RealmObject {
    private Produit produit;
    private Date date;
}
```

On utilise une notation pointée pour suivre la référence :

```
RealmResults<Achat> achats_dentif = realm
    .where(Achat.class)
    .equalTo("produit.designation", "dentifrice")
    .findAll();
```

Bien mieux avec le *RealmFieldNamesHelper*, on écrit tout simplement : `AchatFields.PRODUIT.DESIGNATION`

6.5.12. Jointures 1-N

Cela marche aussi avec cette classe :


```
public class Achat extends RealmObject {
    private RealmList<Produit> produits;
    private Date date;
}
```

Cette requête va chercher parmi tous les produits liés et retourne les achats qui sont liés à au moins un produit dentifrice :

```
RealmResults<Achat> achats_dentif = realm
    .where(Achat.class)
    .equalTo(AchatField.PRODUITS.DESIGNATION, "dentifrice")
    .findAll();
```

Mais impossible de chercher les achats qui ne concernent *que* des dentifrices ou *tous* les dentifrices.

6.5.13. Jointures inverses

Avec les jointures précédentes, on peut aller d'un Achat à un Produit. Realm propose un mécanisme appelé *relation inverse* pour connaître tous les achats contenant un produit donné : 

```
public class Achat extends RealmObject {
    private RealmList<Produit> produits;
    private Date date;
}

public class Produit extends RealmObject {
    private String designation;
    private float prix;
    @LinkingObjects("produits")
    private final RealmResults<Achat> achats = null;
}
```

NB: On ne peut pas utiliser le *RealmFieldNamesHelper*.

6.5.14. Jointures inverses, explications

Pour lier une classe CLASSE2 à une CLASSE1, telles que l'un des champs de CLASSE1 est du type CLASSE2 ou liste de CLASSE2, il faut modifier CLASSE2 :

- définir une variable finale valant null du type `RealmList<CLASSE1>` ; elle sera automatiquement réaffectée lors de l'exécution
- lui rajouter une annotation `@LinkingObjects("CHAMP")` avec CHAMP étant le nom du champ concerné dans CLASSE1.

Dans l'exemple précédent, le champ `achats` de `Produit` sera rempli dynamiquement par tous les `Achat` contenant le produit considéré.

6.5.15. Jointures inverses, exemple


Exemple de requête : 

```
Produit dentifrice = realm
    .where(Produit.class)
    .equalTo(ProduitFields.DESIGNATION, "dentifrice")
    .findFirst();

RealmResults<Achat> achats_dentifrice = dentifrice.getAchats();
```

Ceci en utilisant Lombok pour créer le *getter* sur le champ `achats`.


6.5.16. Suppression par une requête

Le résultat d'une requête peut servir à supprimer des n-uplets. Il faut que ce soit fait dans le cadre d'une transaction et attention à ne pas supprimer des données référencées dans d'autres objets : 


```
realm.beginTransaction();
achats_dentif.deleteAllFromRealm();
dentifrices.deleteAllFromRealm();
realm.commitTransaction();
```

6.6. Requêtes et adaptateurs de listes

6.6.1. Adaptateur Realm pour un RecyclerView

Realm simplifie énormément l’affichage des listes, voir le cours 4 pour les notions. Soit par exemple un `RecyclerView`. Le problème essentiel est la création de l’adaptateur. Il y a une classe `Realm` pour cela, voici un exemple : 

```
public class ProduitAdapter
    extends RealmRecyclerViewAdapter<Produit, ProduitView>
{
    public ProduitAdapter(RealmResults<Produit> produits)
    {
        super(produits, true);
    }
}
```

Le booléen `true` à fournir à la superclasse indique que les données se mettent à jour automatiquement. Ensuite, il faut programmer la méthode de création des vues : 

```
@Override
public ProduitView onCreateViewHolder(
    ViewGroup parent, int viewType)
{
    View itemView = LayoutInflater
        .from(parent.getContext())
        .inflate(R.layout.item, parent, false);
    return new ProduitView(itemView);
}
```

En fait, c’est la même méthode que dans le cours n°4 avec les `PlaneteView`.

Ensuite, il faut programmer la méthode qui place les informations dans les vues : 

```
@Override
public void onBindViewHolder(
    ProduitView holder, final int position)
{
    // afficher les informations du produit
    final Produit produit = getItem(position);
    holder.setItem(produit);
}
```

Elle non plus ne change pas par rapport au cours n°4.

6.6.2. Adaptateur Realm, fin

Dans le cas d'une classe avec identifiant, il faut rajouter une dernière méthode :



```
@Override
public long getItemId(int position)
{
    return getItem(position).getId();
}
```

Elle est utilisée pour obtenir l'identifiant de l'objet cliqué dans la liste, voir `onItemClick` plus loin.

6.6.3. Mise en place de la liste

Dans la méthode `onCreate` de l'activité qui gère la liste :



```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    ...
    Realm realm = Realm.getDefaultInstance();
    RealmResults<Produit> liste =
        realm.where(Produit.class).findAll();
    RecyclerView.Adapter adapter = new ProduitAdapter(liste);
    recyclerView.setAdapter(adapter);
}
```

Comme la liste est *vivante*, toutes les modifications seront automatiquement reportées à l'écran sans qu'on ait davantage de choses à programmer.

6.6.4. Réponses aux clics sur la liste

L'écouteur de la liste doit transformer la position du clic en identifiant :



```
@Override
public void onItemClick(
    AdapterView<?> parent, View v, int position, long id)
{
    long identifiant = adapter.getItemId(position);
    ...
}
```

Le paramètre `position` donne la position dans la liste, et le paramètre `id` n'est que l'identifiant de la vue cliquée dans la liste, pas l'identifiant de l'élément cliqué, c'est pour cela qu'on utilise `getItemId` pour l'obtenir.

6.6.5. C'est la fin

C'est fini pour cette semaine, rendez-vous la semaine prochaine pour un cours sur les applications graphiques 2D.

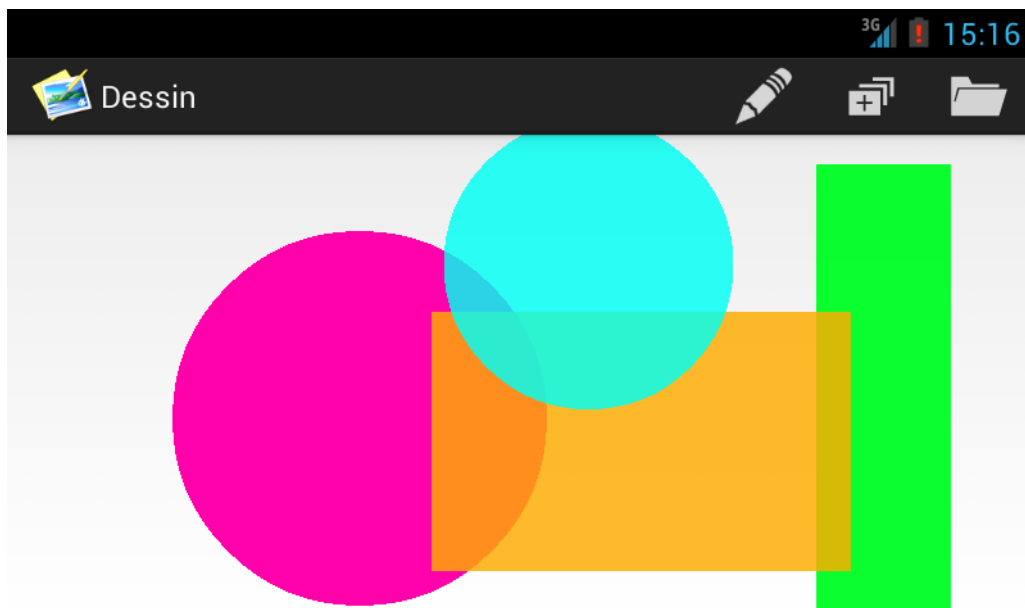


Figure 40: Application de dessin

Semaine 7

Dessin 2D interactif

Le cours de cette semaine concerne le dessin de figures 2D et les interactions avec l'utilisateur.

- CustomView et Canvas
- Un exemple de boîte de dialogue utile

7.1. Dessin en 2D

7.1.1. But

figure 40

7.1.2. Principes

Une application de dessin 2D doit définir une sous-classe de `View` et surcharger la méthode `onDraw`. Voici un exemple :

```
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);  
}  
@Override  
public void onDraw(Canvas canvas) {  
    canvas.drawCircle(100, 100, 50, mPeinture);  
}  
}
```

7.1.3. Layout pour le dessin

Pour voir ce dessin, il faut l'inclure dans un layout :



```
<?xml version="1.0" encoding="utf-8"?>  
<fr.iutlan.dessin.DessinView  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:id="@+id/dessin"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent" />
```

Il faut mettre le package et le nom de la classe en tant que balise XML.

L'identifiant permettra d'ajouter des écouteurs pour les touches et déplacements.

7.1.4. 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 pourrait associer 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()`.


7.1.5. 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.

7.1.6. 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`.

7.1.7. 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.

7.1.8. 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`).



Figure 41: Dégradé horizontal

7.1.9. 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 41

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.

7.1.10. 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.

Mais pour les jeux, autant faire appel à OpenGL (ou Unity, etc.), mais son apprentissage demande quelques semaines de travail acharné.

7.1.11. « 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`.

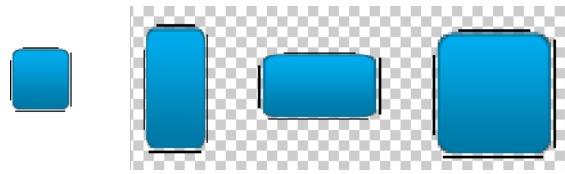


Figure 42: Image étirable



Figure 43: Dessin vectoriel XML

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 ».

7.1.12. 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 42

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.

- 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>
```

Voir la figure 43, page 137.

7.1.13. 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.

7.1.14. 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.

7.1.15. 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) {
        ...
    }
}
```

7.1.16. 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° 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

7.2. Interactions avec l'utilisateur


7.2.1. 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;
}
```

7.2.2. 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:
```

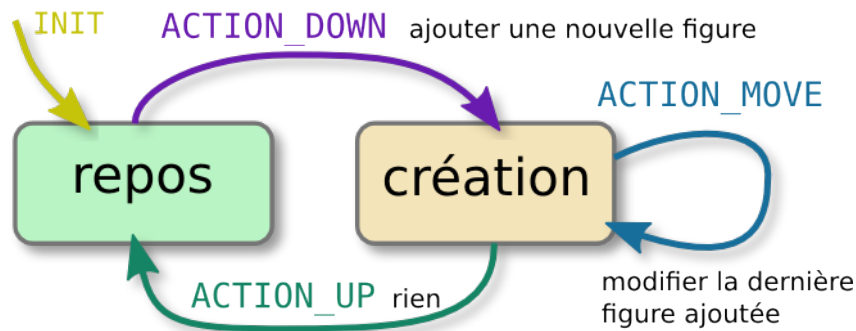


Figure 44: Automate

```
if (figures.size() < 1) return true;
figure = figures.getLast();
figure.setCoin(x,y);
break;
}
invalidate();
```

7.2.3. 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 44

7.2.4. Programmation d'un automate

Pour coder l'automate précédent, il faut une variable qui représente son état.

```
private enum Etat {
    REPOS, CREATION
};
private Etat mEtat = Etat.REPOS;
```

Ensuite, à chaque événement, on se décide en fonction de cet état, voir transparent suivant.

```
public boolean onTouchEvent(MotionEvent event) {
    switch (mEtat) {
        case REPOS:
            switch (event.getAction()) {
                case MotionEvent.ACTION_DOWN:
                    mEtat = Etat.CREATION;
                    break;
                ...
            }
        case CREATION:
            switch (event.getAction()) {
                case MotionEvent.ACTION_MOVE:
                    modifierLaDerniereFigureAjoutee();
                    break;
                case MotionEvent.ACTION_UP:
                    rien();
                    break;
            }
    }
}
```



Figure 45: Sélectionneur de couleur

```
    }  
    break;  
    case CREATION:  
        switch (event.getAction()) {  
            case MotionEvent.ACTION_MOVE: ...  
            case MotionEvent.ACTION_UP:  
                mEtat = Etat.REPOS;  
                ...  
        }  
    }
```

7.3. Boîtes de dialogue spécifiques

7.3.1. Sélectionneur de couleur

Android ne propose pas de sélectionneur de couleur, alors il faut le construire soi-même.

figure 45

7.3.2. Version simple

En TP, on va construire une version simplifiée afin de comprendre le principe :

Voir la figure 46, page 141.

7.3.3. Concepts

Plusieurs concepts interviennent dans ce sélectionneur de couleur :

- La fenêtre dérive de `DialogFragment`, elle affiche un dialogue de type `AlertDialog` avec des boutons Ok et Annuler,
- 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 sélectionneur.

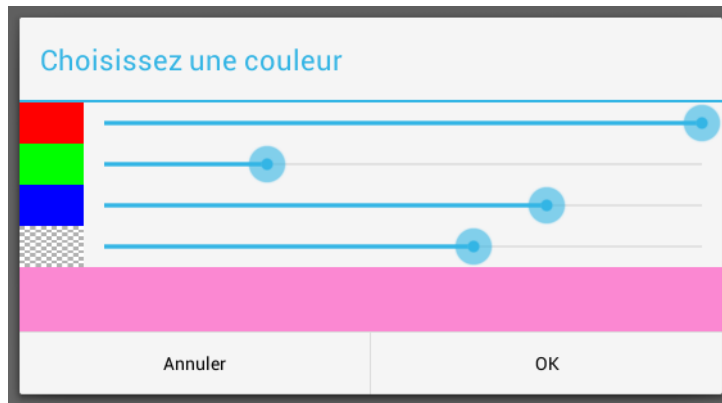


Figure 46: Sélecteur de couleur simple

7.3.4. 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 OnColorChangeListener {  
    void onColorChanged(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.

7.3.5. 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.onColorChanged(cpv.getColor());  
            }  
        });  
    builder.setNegativeButton(android.R.string.no, null);  
    return builder.create();  
}
```

7.3.6. 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.colorpickerview, this);  
        ...  
    }  
}
```

7.3.7. Layout de cette vue

Le layout `colorpickerview.xml` contient quatre `SeekBar`, rouge, vert, bleu et alpha : 

```
<LinearLayout xmlns:android="..."  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:orientation="vertical" >  
    <SeekBar android:id="@+id/sbRouge"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:layout_weight="1"  
        android:max="255" />  
    <SeekBar android:id="@+id/sbVert"  
        ...
```

7.3.8. Écouteurs

Tous ces `SeekBar` ont un écouteur similaire : 

```
SeekBar sbRouge = 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));  
            }  
    });
```

Celui-ci change seulement la composante rouge de la variable `mColor`. Il y a les mêmes choses pour les autres composantes.

7.3.9. Utilisation du dialogue

Pour finir, voici comment on affiche ce dialogue, par exemple dans un menu :



```
ColorPickerDialog dlg =  
    new ColorPickerDialog(  
        new ColorPickerDialog.OnColorChangedListener() {  
            @Override  
            public void onColorChanged(int color) {  
                // utiliser la couleur ....  
            }  
        }  
    );  
dlg.show(getFragmentManager(), "colorpickerdlg");
```

L'écouteur reçoit la nouvelle couleur du sélecteur et peut la transmettre à la classe de dessin.

7.3.10. Sélecteur de fichier

Dans le même genre mais nettement trop complexe, il y a le sélecteur de fichiers pour enregistrer un dessin.

Voir la figure 47, page 144.

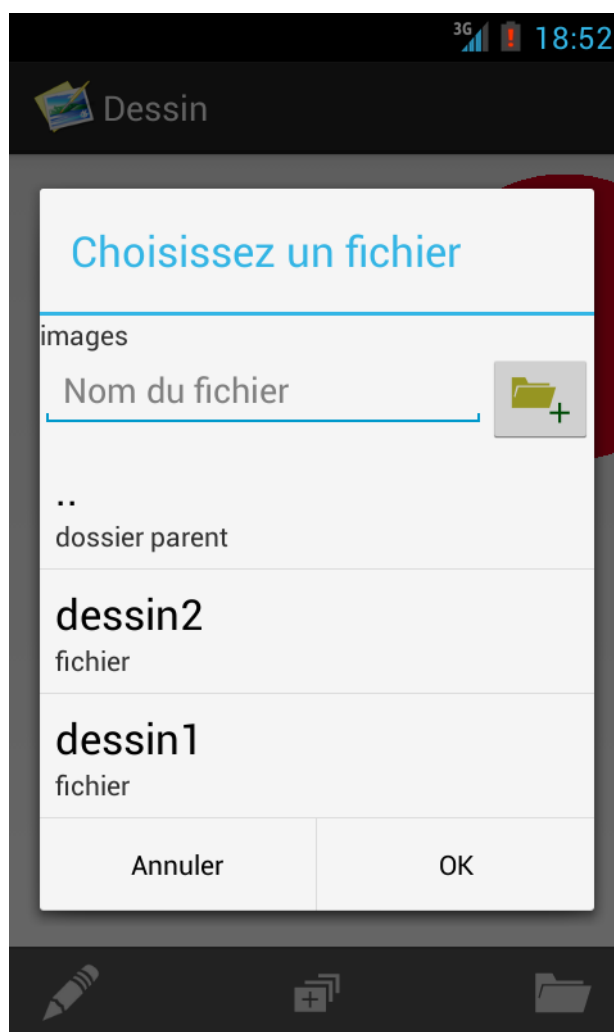


Figure 47: Sélecteur de fichier

Semaine 8

Test logiciel

Le cours de cette semaine est consacré au test systématique d'une application Android. Cela se fait en programmant de nouvelles classes et méthodes spéciales, qui font des vérifications sur les classes et fonctions de l'application.

Tester un logiciel de manière automatisée permet de garantir une non-régression lors du développement. Sans tests systématiques, il est facile de casser involontairement un logiciel complexe : oublis, écrasements, doublons, etc.

AndroidStudio permet d'effectuer deux sortes de tests :

- des tests unitaires pour vérifier des classes individuellement,
- des tests sur AVD pour vérifier le comportement de l'interface.

8.1. Introduction

8.1.1. Principe de base

Le principe général consiste à programmer des fonctions qui vont appeler d'autres fonctions pour vérifier leurs résultats.

Soit une fonction `float racine(float x)` qui est censée calculer la racine carrée d'un réel positif. Pour savoir si elle fonctionne bien, il faudrait calculer $racine(x) * racine(x)$ pour chaque nombre réel x positif, c'est à dire appliquer sa définition mathématique $\forall x \in R^+, racine(x)^2 = x$.

Ainsi, la fonction de test pourrait s'écrire :

```
for (float x=0.0f; x<=100.0f; x += 0.01f) {  
    float rac = racine(x);  
    if (rac * rac != x)  
        throw new Exception("test racine échoué");  
}
```

8.1.2. Limitations

On voit qu'il n'est pas possible de vérifier chaque réel. On se limite à quelques valeurs représentatives et on suppose que la fonction est correcte pour les autres.

D'autre part, il est possible que le test emploie la même définition que la fonction, ce qui ne prouvera pas qu'elle est bonne ; par exemple, la même série limitée pour *sinus*. Pour bien tester, il faut trouver un algorithme totalement différent.

Souvent, le testeur est indépendant des programmeurs. Il ne doit pas savoir comment les fonctions ont été codées. Il doit s'appuyer sur le cahier des charges et explorer toutes les limites. Une grande expérience en programmation est utile pour faire de bons tests.

8.1.3. Précision des nombres

Il faut se méfier de la précision des données. En effet, sur un ordinateur certains nombres sont mal représentés (format IEEE interne). Par exemple, $0,1 * 0,1$ n'est pas égal à $0,01$; il y a un petit écart à cause du défaut de précision.

Si on veut tester la fonction *racine*, on doit programmer ainsi :

```
final float epsilon = 1e-5f;
for (float x=0.0f; x<=100.0f; x += 0.01f) {
    float rac = racine(x);
    if (fabs(rac*rac - x) > epsilon)
        throw new Exception("test racine échoué");
}
```

On doit *toujours* comparer deux réels v_1 et v_2 par $|v_1 - v_2| \leq \epsilon$, *jamais* par $v_1 == v_2$. Le ϵ est à déterminer empiriquement.

8.1.4. Généralisation des tests

Les tests peuvent s'appliquer à tout élément programmé : classe, entrepôt de données, interface utilisateur.

On sépare généralement les tests en plusieurs catégories :

- **tests unitaires** : ils ne concernent qu'une seule classe à la fois, et on teste chaque méthode. Si cette classe fait appel à une autre, cette autre classe est soit totalement vérifiée, soit simulée. Ils représentent généralement 70% des tests.
- **tests d'intégration** : leur but est de vérifier les relations entre classes, les appels de méthodes et les traitements globaux. Ce sont 20% des tests.
- **tests d'instrumentation** : ils vérifient l'interface utilisateur, qu'elle déclenche les bonnes actions et que les informations sont correctement affichées. Ce sont 10% des tests.

8.2. Tests unitaires

8.2.1. Programmation des tests unitaires

Les tests unitaires consistent à vérifier chaque méthode de chaque classe indépendamment : appeler telle méthode avec tels paramètres doit retourner telle valeur.

Sur Android, on utilise l'API **JUnit4** et AndroidStudio s'attend à ce qu'ils soient placés dans le dossier `test` des sources : dans `app/src/test` et avoir le même packaging que les classes testées.

Soit une application dont le packaging est `fr.iutlan.tp8`. On va avoir :

- ses sources dans `app/src/main/java/fr/iutlan/tp8`,
- ses tests dans `app/src/test/java/fr/iutlan/tp8`.

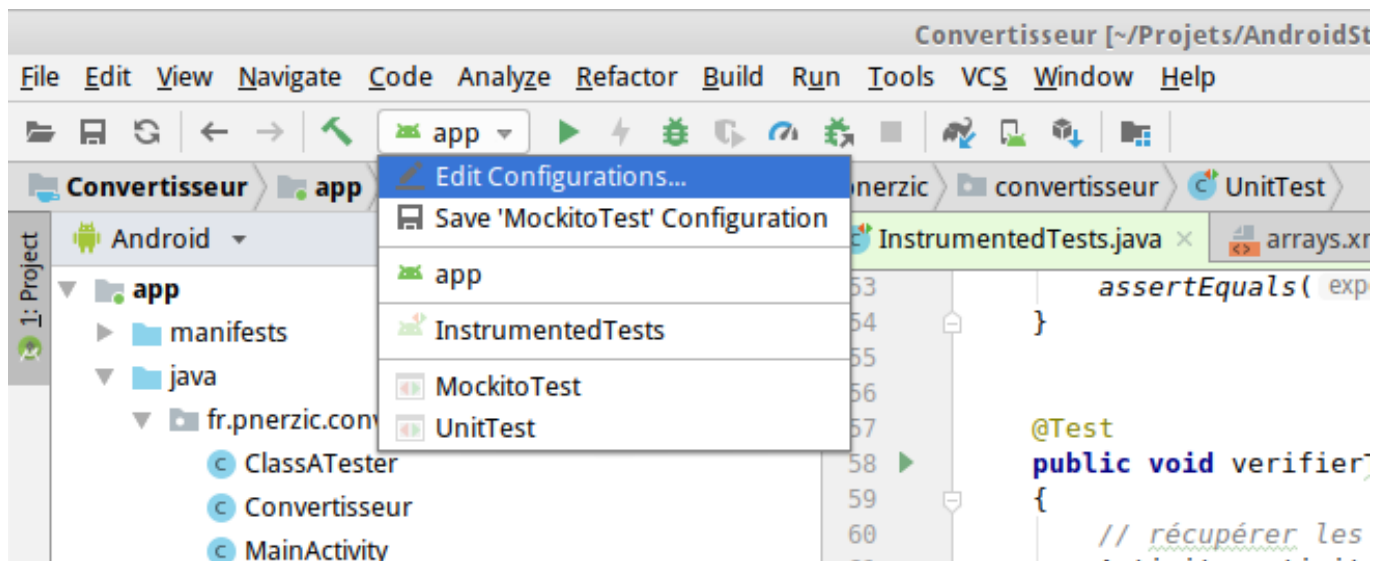


Figure 48: Configuration de lancement

8.2.2. Exécution des tests unitaires

Pour lancer un test, il suffit de déplier les sources des tests dans le navigateur à gauche de l'écran, cliquer droit sur celui voulu et choisir **Run test**.

Il est également possible de configurer le lancement de l'application principale pour qu'elle effectue tous les tests avant. Voir en TP.

figure 48

8.2.3. JUnit4

Soit une classe **Chose** à tester. On doit programmer une classe contenant des méthodes annotées par **@Test** :

```
package fr.iutlan.tp8;

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestClasseChose
{
    @Test
    public void testMethode1()
    {
        assertEquals(50, Chose.methode1(5));
    }
}
```

8.2.4. Explications

La bibliothèque JUnit4 définit l'annotation **@Test** pour dire que la méthode annotée est un test. Dans ce test, **assertEquals** vérifie l'égalité entre une valeur et un résultat d'appel de fonction.

L'API contient de nombreuses directives comme `assertEquals`, appelées *assertions*, permettant toutes sortes de vérifications. Le principe reste toujours de lancer l'exécution d'une méthode et d'analyser le résultat :

- comparer le résultat à une constante : égalité, différence,
- vérifier l'absence ou la présence d'une exception,
- vérifier que le temps d'exécution ne dépasse pas une limite.

Quand une assertion échoue, cela déclenche une `AssertionError`. Malheureusement, JUnit affiche la trace de la pile, ce qui n'est pas très agréable.

8.2.5. `import static` en Java

La directive `import static`, utilisée pour importer les `assert`, permet d'utiliser des définitions de classe sans devoir mettre le nom de la classe devant.

```
import java.lang.Math;

public void essai1() {
    System.out.println("sin(pi/2) = "+Math.sin(Math.PI/2));
}
```

peut s'écrire plus simplement :



```
import static java.lang.Math.*;

public void essai1() {
    System.out.println("sin(pi/2) = "+sin(PI/2));
}
```

8.2.6. Assertions JUnit

Voici le catalogue des méthodes d'assertions. Leur signification est évidente.

- Tests booléens
 - `assertTrue(condition)` passe si la condition est vraie
 - `assertFalse(condition)`
- Tests de nullité
 - `assertNull(objet)` passe si l'objet vaut null
 - `assertNotNull(objet)`
- Tests d'objets
 - `assertSame(objet, autre)` passe si les deux paramètres désignent le même objet java
 - `assertNotSame(objet, autre)`
- Comparaisons
 - `assertEquals(voulu, calcul)` passe si $calcul = voulu$
 - `assertEquals(voulu, calcul, tolerance)` pour des float ou double, passe si $|calcul - voulu| \leq tolerance$
 - `assertArrayEquals(tab1, tab2)` passe si les tableaux contiennent les mêmes éléments
 - `assertArrayEquals(tab1, tab2, tolerance)` idem pour des float[] ou double[] avec une tolérance

Attention à ne pas intervertir les paramètres, sans quoi rien ne sera testé correctement. L'appel de la méthode à tester est à mettre en seconde position, et son résultat attendu en premier.

8.2.7. Affichage d'un message d'erreur

Toutes ces méthodes `assert*` peuvent prendre un premier paramètre chaîne qui contient un message informatif permettant de comprendre le rôle du test.

```
@Test
public void testCarre()
{
    assertEquals("carre(5)=25", 25, Chose.carre( 5));
    assertEquals("carre(-2)=4", 4, Chose.carre(-2));
}
```

Ce message sera affiché en cas d'échec du test.

Il est aussi recommandé de donner des noms intelligibles aux méthodes de test, car ils sont affichés lors de l'exécution des tests.

8.2.8. Vérification des exceptions

La vérification des exceptions se fait à l'aide d'un paramètre `expected` fourni à l'annotation `@Test` :



```
@Test(expected=ArithmeticException.class)
public void testRacineNegative()
{
    float rac = racine(-2);    // doit déclencher une exception
}

@Test(expected=NumberFormatException.class)
public void testParseNonInt()
{
    int n = Integer.parseInt("1001 nuits");
}
```

8.2.9. Vérification de la durée d'exécution

Pour vérifier que l'exécution ne dure pas trop longtemps, on paramètre l'annotation `@Test` avec une propriété `timeout` :



```
@Test(timeout=100)
public void testDuree()
{
    Chose.calcul();
}
```

La limite de temps est exprimée en millisecondes.

Le test échouera si le temps d'exécution dépasse la durée indiquée.

8.3. Assertions complexes avec Hamcrest

8.3.1. Assertions Hamcrest

L'évolution des assertions dans le but de faciliter leur lisibilité a conduit au développement de bibliothèques compagnons de JUnit, comme Hamcrest. Elle permet d'écrire les assertions autrement, avec une seule méthode `assertThat`.

```
assertEquals(voulu, result);           // JUnit4
assertThat(result, equalTo(voulu));    // Hamcrest

assertTrue(result instanceof String);  // JUnit4
assertThat(result, instanceof(String.class)); // Hamcrest

assertTrue(3, result.size());          // JUnit4
assertThat(result, hasSize(3));        // Hamcrest
```

Tous ces second paramètres de `assertThat` sont appelés *matchers* (correspondants). Ils sont très nombreux et très utiles.

8.3.2. Catalogue des correspondants Hamcrest

Le principe est d'écrire `assertThat(calcul, matcher)`. Le *matcher* peut être simple comme dans les exemples précédents ou très complexe pour des vérifications subtiles.

- Comparaisons numériques
 - `equalTo(valeur)` passe si la valeur est égale au calcul
 - `closeTo(valeur, err)` passe si $|valeur - calcul| \leq err$
 - `greaterThan(valeur)`
 - `greaterThanOrEqualTo(valeur)`
 - `lessThan(valeur)`
 - `lessThanOrEqualTo(valeur)`

```
assertThat(14, equalTo(14));
assertThat(14, greaterThan(racine(14)));
assertThat(Math.PI, closeTo(3.14, 0.01));
```

- Comparaisons de chaînes
 - `isEmptyString()`
 - il manque un matcher pour la longueur d'une chaîne !!
 - `hasToString(texte)` passe si `calcul.toString()` retourne "texte"
 - `equalToIgnoringCase(texte)`
 - `equalToIgnoringWhiteSpace(texte)`
 - `containsString(texte)`
 - `endsWith(texte), startsWith(texte)`
 - `matchesPattern(regex)` expression régulière Java

```
String msg = "Salut les gens";
assertThat(msg.length(), equalTo(14));    // pas terrible
assertThat(msg, containsString("les"));
assertThat(msg, equalToIgnoringCase("saLut lEs gEnS"));
```

- Si calcul est une collection :
 - `hasSize(nb)` passe si calcul contient nb éléments
 - `hasItem(valeur)` passe si calcul contient la valeur
 - `contains(valeurs...)` passe si calcul contient exactement toutes ces valeurs, dans cet ordre
 - `containsInAnyOrder(valeurs...)` passe si calcul contient toutes ces valeurs, dans n'importe quel ordre
 - `everyItem(matcher)` passe si tous les éléments de calcul vérifient le matcher

```
List<Integer> liste = Arrays.asList(4, 1, 8);
assertThat(liste, hasSize(3));
assertThat(liste, hasItem(1));
assertThat(liste, containsInAnyOrder(1, 4, 8));
assertThat(liste, everyItem(greaterThan(0)));
```

- Classes
 - `instanceOf(classe)` passe si calcul est de cette classe
- Agrégation
 - `allOf(matchers...)` passe si tous les matchers passent
 - `anyOf(matchers...)` passe si l'un des matchers passe

```
assertThat(result, instanceOf(String.class));
assertThat("ok@free.fr",
    allOf(endsWith(".fr"), containsString("@")));
assertThat("ok@free.fr",
    anyOf(endsWith(".org"), endsWith(".fr")));
```

Attention, certaines agrégations sont impossibles quand les types des matchers ne correspondent pas.

8.3.3. Importation de Hamcrest

Pour utiliser les matchers, il faut importer leurs classes :



```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;
```

En plus, dans `build.gradle`, il faut ajouter la dépendance :



```
testImplementation 'junit:junit:4.12'
testImplementation 'org.hamcrest:hamcrest-library:1.3'
```

8.4. Patron Arrange-Act-Assert

8.4.1. Organisation des tests

Pour bien écrire les méthodes de test, on fait appel au patron de conception « AAA » du nom de ses trois étapes :

- **Arrange** (ou *given*) : préparer les données pour le test,
- **Act** (ou *when*) : effectuer le test et récolter le résultat,
- **Assert** (ou *then*) : comparer le résultat à ce qui est attendu.

Par convention, on sépare ces trois étapes avec une ligne vide :

```
@Test
public void testStringUtilsReverse() {
    String input = "abc";

    String result = StringUtils.reverse(input);

    assertEquals("cba", result);
}
```

8.4.2. Préparation des données

Les données permettant de mener les tests peuvent être préparées dans chaque méthode, les rendant toutes indépendantes. Mais cela conduit parfois à beaucoup de redondance.

Au lieu de recopier les mêmes instructions dans chaque test, on préfère les programmer dans une seule méthode qui est exécutée automatiquement avant chaque test. Il faut l'annoter par `@Before`, voir le transparent suivant.

NB: il existe une manière plus puissante, à l'aide de « règles » annotées par `@Rule` mais c'est trop complexe pour ce cours. Les règles permettent de réemployer les mêmes préparations dans différentes classes de test.

8.4.3. Préparation des données avant chaque test

L'annotation `@Before` sur une méthode, dans la classe de test, fait exécuter cette méthode avant chaque test.

```
private ArrayList<Integer> liste;

@Before public void initEachTest() {
    liste = new ArrayList<>(Arrays.asList(1,4,8));
}

@Test public void testSize() {
    int result = liste.size();

    assertEquals(3, result);
}
```


Inconvénient : c'est la même initialisation pour tous les tests.

8.4.4. Préparation des données avant l'ensemble des tests

Dans des cas plus complexes, par exemple la connexion à une base de données, il ne faut initialiser qu'une seule fois pour tout un jeu de tests. On emploie l'annotation `@BeforeClass` :

```
private RealmConfiguration conf;
private Realm realm;

@BeforeClass public void initAllTests() {
    Realm.init(context);
    conf = new RealmConfiguration.Builder().inMemory().build();
}

@Before public void initEachTest() {
    realm = Realm.getInstance(conf);
}
```

NB: cet exemple n'est pas viable tel quel, Realm ne fonctionne pas sans Android.

8.4.5. Clôture de tests

Certaines ressources demandent à être libérées après usage. On définit donc des méthodes miroir des précédentes :

```
@After public void endEachTest() {
    realm.close();
}

@AfterClass public void endAllTests() {
    Realm.close(this);
}
```

Pour nommer les méthodes, on trouve aussi `setup` et son contraire `tearDown`.

8.4.6. Vérification des assertions

Plusieurs assertions peuvent être spécifiées pour vérifier le résultat d'une même action. Si l'une des assertions échoue, tout le test est en échec, les autres assertions ne sont pas essayées. Par contre, les autres tests sont lancés quand même.

```
@Test public void test1() {
    // arrange...
    // act...
    assertThat(result, matcher1a);
    assertThat(result, matcher1b);
}

@Test public void test2() {
    // arrange...
    // act...
    assertThat(result, matcher2a);
    assertThat(result, matcher2b);
}
```

8.4.7. Tests paramétrés

Les fonctions de test vues jusque là ne prennent aucun paramètre. Si on voulait appliquer le même test avec des données différentes, il faudrait écrire autant de fonctions distinctes.

Avec une librairie compagnon de JUnit appelée JUnitParams, il est possible de faire le même test avec des données différentes. Cela s'appelle paramétrer les tests. Le principe est de préparer les données dans un tableau ayant des colonnes du même type que les paramètres de la fonction de test. L'exécuteur du test gère la boucle qui fournit les paramètres ligne par ligne.

L'intérêt est de ne pas devoir multiplier les fonctions ou au contraire grouper des tests, et d'avoir un message d'erreur vraiment spécifique avec les données qui n'ont pas passé le test.

8.4.8. Exemple de test paramétré

Soit une classe `Calendrier` dont on veut tester la méthode `isBissextile(annee)` sur plusieurs années. Au lieu de prévoir plusieurs tests séparés, on prépare différents paramètres qui seront injectés successivement dans la même méthode. Voici le début :

```
@RunWith(JUnitParamsRunner.class)
public class TestsCalendrier {

    @Test @Parameters
    public void testBissextile(int annee, boolean bissextile)
    {
        assertThat(bissextile, Calendrier.isBissextile(annee));
    }
}
```

Notez l'annotation initiale qui déclenche un exécuteur différent. Notez que la méthode `testBissextile` demande des paramètres.

8.4.9. Fourniture des paramètres

Il y a de nombreuses possibilités pour fournir les paramètres, et l'une consiste à programmer une méthode statique qui retourne une collection de tableaux de paramètres.

Le nom de cette méthode est important : "parametersFor" suivi du nom de la méthode de test concernée :

```
static Collection<Object[]> parametersForTestBissextile() {
    return Arrays.asList(new Object[][]{
        {2019, false},
        {2020, true},
        {2021, false},
    });
}
```

D'autres tests JUnit normaux peuvent être placés avec ceux-ci.

8.4.10. Importation de JUnitParams

Pour utiliser les JUnitParams, il faut importer ses classes :

```
import junitparams.JUnitParamsRunner;
import junitparams.Parameters;
```

Et dans `build.gradle`, il faut ajouter la dépendance :

```
testImplementation 'junit:junit:4.12'
testImplementation 'pl.pragmatists:JUnitParams:1.1.1'
```

8.5. Tests d'intégration

8.5.1. Introduction

Quand on a besoin de vérifier les relations entre une classe à tester et une autre, et que l'autre n'est pas encore au point, on construit une sorte de maquette de l'autre classe (*mock-up*), c'est à dire une fausse classe qui se comporte comme il faudrait là où on en a besoin.

Ça se décline en plusieurs variantes :

- **objet bidon** (*dummy object*) : une classe vide qui n'est jamais appelée et qui sert de bouche-trou,
- **objet factice** (*fake object*) : une classe qui ne fait pas réellement le travail prévu, par exemple une base de donnée seulement en mémoire,
- **embryon** (*stub*) : une classe pas complètement codée,
- **objet simulé** (*mock object*) : une classe qui simule le comportement attendu, seulement sur certaines valeurs.

8.5.2. Interface à la place d'une classe

Soit une classe encore non programmée : `Calculatrice.java`. Cette classe devra posséder différentes méthodes de calcul comme `add` et `div`. On voudrait écrire ce qui suit dans une autre classe :

```
private Calculatrice calcul = new Calculatrice();
float total = calcul.add(13, 6);
```

Et on voudrait tester cette autre classe, malgré l'absence de `Calculatrice.java`.

En attendant, il est proposé d'en faire une simple interface :

```
public interface Calculatrice {
    float add(float a, float b);
    ...
    float div(float a, float b) throws ArithmeticException;
}
```

8.5.3. Simulation d'une interface avec Mockito

Le problème de cette interface, c'est qu'on ne peut ni l'instancier, ni l'utiliser. La solution, c'est de la simuler.

C'est très simple avec **Mockito**. Il suffit d'annoter la variable qui contient la calculatrice avec `@Mock` :

```
@Mock private Calculatrice calcul;
```

Cela va instancier la variable avec une classe bidon qui se comporte selon l'interface. On pourra manipuler cette calculatrice comme la vraie. Le problème, c'est que la classe bidon ne peut pas faire les calculs de la vraie classe...

On va donc faire apprendre quelques calculs prédéfinis à cette classe bidon, c'est à dire inventorier tous les appels à `Calculatrice` et préparer les réponses qu'elle devrait fournir.

8.5.4. Apprentissage de résultats

Voici comment faire avec Mockito. C'est en deux parties :

- la circonstance : lorsqu'il y a tel appel de méthode,
- l'action : alors retourner telle valeur ou lancer telle exception.

Voici quelques exemples :

```
@Before
public void initCalculatrice() {
    when(calcu.add(13, 6)).thenReturn(19);
    when(calcu.div(6, 2)).thenReturn(3);
    when(calcu.div(3, 0))
        .thenThrow(ArithmeticException.class);
}
```

Évidemment, il ne faut pas avoir des dizaines d'appels différents, sinon la classe `Calculatrice` serait vraiment indispensable.

8.5.5. Apprentissage généralisé

Dans le dernier exemple précédent, au lieu d'apprendre seulement le résultat de la division de 3 par 0, il est possible d'apprendre que toutes les divisions par zéro déclenchent une exception. Cela se fait avec des *matchers* :

```
@Before
public void initCalculatrice() {
    ...
    when(calcu.div(anyFloat(), eq(0)))
        .thenThrow(ArithmeticException.class);
}
```

Il faut associer un *matcher* comme `anyInt()`, `anyFloat()`... avec un autre *matcher* comme `eq(valeur)`.

La liste des matchers est documentée sur [cette page](#).

8.5.6. *Matchers* pour Mockito

Les *matchers* de Mockito sont un peu différents en syntaxe de ceux de Hamcrest, mais ils ont la même signification, et on peut utiliser ceux de Hamcrest. Voir [cette liste](#), en voici quelques uns :

- types : `anyBoolean()`, `anyInt()`..., `any(MaClasse.class)`
- objets : `isNull()`, `isNotNull()`
- chaînes : `contains(s)`, `startsWith(s)`, `endsWith(s)`
- Hamcrest : `booleanThat(matcher)`, `intThat(matcher)`...
- lambda : `booleanThat(lambda)`, `intThat(lambda)`...

```
when(calcu.div(anyFloat(), 0.0)).thenThrow(...);
when(calcu.sign(floatThat(lessThan(0.0))).thenReturn(-1);
when(calcu.sign(floatThat(nb -> nb > 0))).thenReturn(+1);
```

8.5.7. Autre syntaxe

La syntaxe

```
when(objet.methode(params)).thenReturn(valeur);
```

peut s'écrire différemment :

```
doReturn(valeur).when(objet).methode(params);
```

On utilise cette seconde syntaxe lorsque l'objet n'est pas initialisé « normalement », par exemple lorsqu'il est simulé ou espionné, voir plus loin. La première écriture cause une `NullPointerException`.

8.5.8. Simulation pour une autre classe

Dans les exemples précédents, on simule et on teste la même classe encore non programmée. Dans le cas général, c'est trop restrictif. Voici un exemple :

```
public class Voiture {
    private String modele;
    private Personne proprietaire;

    public Voiture(String modele, Personne proprietaire) {
        this.modele = modele;
        this.proprietaire = proprietaire;
    }

    public String toString() {
        return modele+" de "+proprietaire.getNom();
    }
}
```

On voudrait tester la méthode `toString()` de `Voiture`, mais la classe `Personne` n'est pas encore programmée :

```
public interface Personne {
    String getNom();
}
```

Mockito va simuler une personne pour permettre de tester la voiture, mais il faut ajouter l'annotation `@InjectMocks` à la voiture :

```
@Mock private Personne client;
@InjectMocks Voiture voiture = new Voiture("Tesla S", client);

@Test public void testToString() {
    when(client.getNom()).thenReturn("Pierre");
    assertEquals("Tesla S de Pierre", voiture.toString());
}
```

8.5.9. Surveillance d'une classe

Certains tests ont pour objectif de surveiller les appels aux méthodes d'une certaine classe (entièrement programmée) : telle méthode est-elle appelée, combien de fois et avec quels paramètres ?

On commence par ajouter l'annotation `@Spy` à l'objet surveillé :

```
@Spy private CalculatriceOk calcul = new CalculatriceOk();
```

Ensuite, on fait appeler ses méthodes (directement ou pas) :

```
calcu.add(x, 1.0f);
```

Enfin, on vérifie qu'elles ont été appelées, N fois ou jamais :

```
verify(calcu).add(anyFloat(), anyFloat());  
verify(calcu, times(1)).add(anyFloat(), anyFloat());  
verify(calcu, never()).sub(anyFloat(), anyFloat());
```

8.5.10. Surveillance d'une activité Android

On utilise cette technique d'espionnage pour tester certains comportements d'une activité Android. Voir page 160 pour d'autres vérifications en situation sur AVD.

On souhaite vérifier par exemple des méthodes d'affichage dans des TextView, de lecture dans des EditText, et autres :

```
TextView tvSortie;  
  
void putSortie(double nb)  
{  
    String texte = String.format(Locale.FRANCE, "%.3f", nb);  
    tvSortie.setText(texte);  
}
```

Dans un cas réel, cela peut être nettement plus complexe que ça.

8.5.11. Espionnage et simulation

La technique consiste à espionner l'activité et à simuler la vue :

```
// activité espionnée  
@Spy MainActivity activity = new MainActivity();  
  
// fausse vue  
@Mock TextView textViewMock;
```

L'idée est :

- d'associer ce `textViewMock` à celui de l'activité, c'est à dire que l'activité va accéder à ce `TextView` en croyant manipuler le vrai,
- de simuler les réactions de ce `textViewMock` : simuler sa méthode `setText()` et voir ce que l'activité voulait y mettre pour comparer avec ce qu'on attend.

8.5.12. Liaison des vues à l'activité

Ensuite on initialise ce couple à chaque test :

```
@Before public void initEachTest() {  
    // lier les ressources et les vues  
    doReturn(textViewMock)  
        .when(activity).findViewById(R.id.sortie);  
  
    // l'activité récupère ses vues  
    activity.findViews();  
}
```

La méthode `findViews()` de l'activité est classique :

```
void findViews()
{
    tvSortie = findViewById(R.id.sortie);
}
```

8.5.13. Test d'appel

Test = appel de la méthode et vérification du résultat :

```
@Test public void putSortieEcritValeurVoulue()
{
    // arrange

    // act
    activity.putSortie(3.14159);

    // assert
    verify(textViewMock).setText("3,142");
}
```

L'assertion vérifie que l'activité a demandé au `TextView` d'afficher cette chaîne. C'est possible car Mockito a surveillé tous les appels aux méthodes de ce `TextView`.

8.5.14. Installation de Mockito

Voici comment installer Mockito dans un projet Android. Il faut ajouter ceci dans `app/build.gradle` :



```
testImplementation 'junit:junit:4.12'
testImplementation 'org.mockito:mockito-core:2.25.0'
androidTestImplementation 'org.mockito:mockito-android:2.25.0'
```

Ensuite, annoter chaque classe de tests qui y fait appel :



```
@RunWith(MockitoJUnitRunner.class)
public class TestsAvecMockito {

    @Test public void test1() ...
    @Test public void test2() ...
    ...
}
```

8.6. Tests sur AVD

8.6.1. Définition

On veut maintenant tester l'application sur un AVD : vérifier ce qui est affiché et ce qui se passe quand l'utilisateur saisit des textes et actionne des contrôles.

Avec l'API **Expresso**, le principe est de spécifier des actions sur certaines vues de l'interface, accompagnées éventuellement d'assertions.

Le schéma général est (*//=optionnel*) :

```
onView(matcher)[.perform(action)][.check(assert)]
```

```
@Test public void testBtnValider() {  
    onView(withId(R.id.et_prenom)).perform(typeText("Pierre"));  
    onView(withId(R.id.btn_ok)).check(matches(withText("Ok")));  
    onView(withId(R.id.btn_ok)).perform(click());  
}
```

8.6.2. Correspondants de vues

- Correspondants de vues `onView(viewMatcher)`

Le paramètre `ViewMatcher` désigne la vue par son identifiant `withId(identifiant)` ou son libellé `withText(label)`. Comme pour JUnit, il y a beaucoup de matchers possibles, voir une sélection au transparent suivant.

- Actions de vues `perform(viewAction, ...)`

Parmi les `ViewAction`, il y a `scrollTo()`, `click()`, `pressBack()`, et `typeText()`, voir plus loin.

- Assertions de vues `check(viewAssertion, ...)`

L'assertion vaut généralement `matches(matcher)` avec `matcher` étant un `ViewMatcher`.

8.6.3. *ViewMatchers* d'Expresso

Voici un petit extrait de ce qu'il est possible de tester :

- `withText(m)` sur des `TextView`, `EditText`, `Button`, `m` étant soit une chaîne, soit une ressource, soit un matcher JUnit.
- `withSpinnerText(m)` sur un `Spinner` (idem pour `m`)
- `isChecked()`, `isChecked()` sur des `CheckBox`, `ToggleButton`

```
onView(withId(R.id.et_prenom)).check(withText("Pierre"));  
onView(withId(R.id.cb_logged)).check(isChecked());  
onView(withId(R.id.sp_metier))  
    .check(matches(withSpinnerText("enseignant")));
```

8.6.4. *ViewActions* d'Expresso

Voici un petit extrait de ce qu'il est possible de mettre en paramètre de `perform()` :

- `click()`, `pressBack()` un peu partout
- `clearText()`, `typeText()`, `closeSoftKeyboard()` sur des `EditText`
- `scrollTo()` sur des `Spinner` et `ListView`
- la classe `RecyclerViewActions` fournit des actions spécifiques, comme `actionOnItemAtPosition(pos, action)`, `scrollToPosition(pos)` sur des `RecyclerView`

```
onView(withId(R.id.et_prenom)).perform(typeText("Pierre"));  
onView(withText("Ok")).perform(click());  
onView(withId(R.id.liste)).perform(  
    RecyclerViewActions.actionOnItemAtPosition(1, click()));
```

8.6.5. Tests sur des listes

Pour vérifier la sélection d'items dans des `Spinner` et `ListView`, c'est un peu plus compliqué. Comme les éléments ne sont pas forcément affichés à l'écran, on doit faire appel aux données et non pas aux vues, utiliser `onData` au lieu de `onView`.

Le schéma général est :


```
onData(matcher)[.perform(action)][.check(assert)]
```

On doit lui fournir un *matcher* qui désigne les données des adaptateurs de l'écran actuel. Le problème, c'est que le matcher dépend du type d'adaptateur et c'est compliqué. Alors pour simplifier, on ne verra que la sélection en fonction de la position :

```
onData(anything()).atPosition(pos)
```

8.6.6. Test sur un spinner

Il y a un petit cas particulier, celui du **Spinner** (liste déroulante). Il faut d'abord cliquer dessus par **onView**, puis cliquer sur un élément par **onData**. Si on manque l'une des étapes, alors Espresso reste bloqué sur le layout du spinner.

Voici la bonne séquence pour sélectionner l'item n° position :

```
onView(withId(R.id.spinner)).perform(click());  
onData(anything()).atPosition(position).perform(click());
```

8.6.7. Installation de Espresso

Il faut ajouter ceci dans app/build.gradle :

```
testImplementation 'junit:junit:4.12'  
androidTestImplementation('androidx.test.espresso:espresso-core:3.1.0', {  
    exclude group: 'com.android.support', module: 'support-annotations'  
})  
androidTestImplementation 'androidx.test.espresso:espresso-intents:3.2.0'  
androidTestImplementation 'androidx.test.espresso:espresso-web:3.2.0'
```

NB: cliquez sur l'icône de téléchargement pour voir les lignes entières. D'autre part, les numéros de version sont susceptibles de changer.

8.6.8. Classe de test

Il faut également préparer la classe de test et les méthodes d'une manière un peu spéciale :

```
@RunWith(AndroidJUnit4.class)  
public class TestsInterface  
{  
    @Rule public ActivityTestRule<MainActivity> mActivityRule =  
        new ActivityTestRule<>(MainActivity.class);  
  
    @Test public void testSurActivity()  
    {  
        // arrange: lancer MainActivity  
        final MainActivity activity = mActivityRule.getActivity();  
        ...  
    }  
}
```

8.6.9. Manipulations directes de l'activité

Lorsqu'il faut appeler une méthode publique de l'activité, il faut impérativement se placer dans le même thread d'interface utilisateur qu'elle, sinon il y aura une exception.

Voici comment faire :

```
// arrange ou act
activity.runOnUiThread(new Runnable() {
    @Override
    public void run() {
        // ici on peut appeler une méthode de activity
        activity.doSomething();
    }
});
```

8.6.10. C'est la fin du cours et du module

C'est fini, nous avons étudié tout ce qu'il était raisonnable de faire en 8 semaines.

Le logiciel calcule les coordonnées 3D exactes de l'écran et des dessins, afin de les superposer avec précision sur la vue réelle.

Ce sont les mêmes types de calculs qu'en synthèse d'images 3D, mais inversés : au lieu de simuler une caméra, on doit retrouver ses caractéristiques (matrices de transformation), et ensuite dans les deux cas, on dessine des éléments 3D utilisant ces matrices.

9.1.4. Réalité augmentée dans Android

Les tablettes et téléphone contiennent tout ce qu'il faut pour une première approche. Les capteurs ne sont pas très précis et l'écran est tout petit, mais ça suffit pour se faire une idée et développer de petites applications.

La suite de ce cours présente les capteurs et la caméra, puis leur assemblage, mais avant cela, il faut se pencher sur le mécanisme des permissions, afin d'avoir le droit d'utiliser les capteurs.

9.2. Permissions Android

9.2.1. Concepts

Certaines actions logicielles sont liées à des permissions. Ex : accès au réseau, utilisation de la caméra, enregistrement de fichiers, etc.

Dans les premières versions d'Android, les applications devaient spécifier toutes les demandes d'autorisations dans le fichier manifest. Ces demandes étaient examinées lors de l'installation de l'application. Elles devaient être intégralement acceptées par l'utilisateur, ou alors l'application entière n'était pas installée.

Depuis l'API 23, les permissions sont demandées au moment où elles sont nécessaires. L'utilisateur peut les accepter ou les refuser. Dans le cas de refus, l'application peut, si elle est programmée correctement, partiellement continuer à fonctionner.

9.2.2. Permissions dans le manifeste

Les droits demandés par une application sont nommés à l'aide d'une chaîne, par exemple "android.permission.CAMERA". Ils doivent être déclarés dans le `AndroidManifest.xml` à l'aide d'une balise `<uses-permission android:name="permission"/>`.

```
<manifest xmlns:android="..." package="...">
  <uses-permission
    android:name="android.permission.ACCESS_FINE_LOCATION"/>
  <uses-permission
    android:name="android.permission.CAMERA"/>
  <application android:label="..." ...>
    <activity android:name="...">
      <intent-filter>...</intent-filter>
    </activity>
  </application>
</manifest>
```

9.2.3. Raffinement de certaines permissions

Certains dispositifs demandent des droits plus fins. C'est le cas de la caméra. Cela fait rajouter d'autres éléments : 

```
<uses-permission android:name="android.permission.CAMERA"/>
<uses-feature android:name="android.hardware.camera"/>
<uses-feature android:name="android.hardware.camera.autofocus"/>
```



Figure 50: Demande de droit

9.2.4. Demandes de permissions à la volée

À partir de Android 6 Marshmallow, API 23, les permissions ne sont plus vérifiées seulement au moment d'installer une application, mais en permanence. Et d'autre part, l'utilisateur est maintenant relativement libre d'en accepter certaines et d'en refuser d'autres.

Certaines permissions sont automatiquement accordées si on décide d'installer l'application, mais d'autres qui concernent la vie privée des utilisateurs (carnet d'adresse, réseau, caméra, etc.) font l'objet d'un contrôle permanent du système Android. Une application qui n'a pas une autorisation ne peut pas utiliser le dispositif concerné, et se fait interrompre par une exception (plantage si pas prévu).

Cela impose de programmer des tests à chaque opération concernant un dispositif soumis à autorisation.

9.2.5. Test d'une autorisation

Au plus simple, ça donne ceci :



```
// le test des permissions concerne Android M et suivants
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
    int autorisation =
        this.checkSelfPermission(Manifest.permission.CAMERA);
    if (autorisation != PackageManager.PERMISSION_GRANTED) {
        // l'activité n'a pas le droit d'utiliser la caméra
        Log.e(TAG, "accès à la caméra refusé");
        return;
    }
}
// l'activité a le droit d'utiliser la caméra
```

L'activité `this` regarde simplement si elle peut utiliser la caméra. La réponse est oui (`PERMISSION_GRANTED`) ou non.

9.2.6. Demande d'une autorisation

L'exemple précédent se contentait de tester l'autorisation. Ce qui est plus intéressant, c'est de demander à l'utilisateur de bien vouloir autoriser l'application à utiliser la caméra. Android va afficher un dialogue :

figure 50

L'utilisateur est libre d'accepter ou de refuser. S'il refuse, `checkSelfPermission` ne renverra jamais plus `PERMISSION_GRANTED` (sauf si on insiste, cf plus loin).

9.2.7. Préférences d'application

Les droits sont stockés dans les préférences de l'application (Applications/application/Autorisations).

Voir la figure 51, page 166.

Ils peuvent être révoqués à tout moment. C'est pour cette raison que les applications doivent tester leurs droits en permanence.

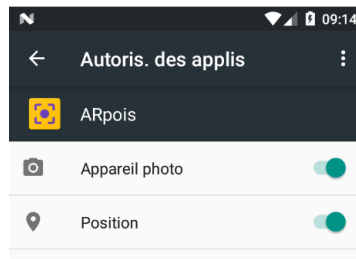


Figure 51: Préférences

9.2.8. Dialogue de demande de droits

C'est en deux temps :

1. l'activité émet une demande qui fait apparaître un dialogue :

```
...  
// l'activité n'a pas encore le droit mais fait une demande  
requestPermissions(new String[] {Manifest.permission.CAMERA}, ...);
```

2. Lorsque l'utilisateur a répondu, Android appelle cet écouteur :

```
@Override  
public void onRequestPermissionsResult(...) {  
    ... regarder les permissions accordées ou refusées ...  
}
```

9.2.9. Affichage du dialogue

On doit appeler la méthode `requestPermissions` en fournissant un tableau de chaînes contenant les permissions demandées, ainsi qu'un entier `requestCode` permettant d'identifier la requête. Cet entier sera transmis en premier paramètre de l'écouteur.

```
requestPermissions(String[] permissions, int requestCode)
```

L'écouteur reçoit le code fourni à `requestPermissions`, les permissions demandées et les réponses accordées :

```
public void onRequestPermissionsResult(  
    int requestCode,  
    String[] permissions, int[] grantResults)
```

Cet écouteur n'est pas nécessaire si on fait tous les tests de permissions avant chaque appel sensible.

9.2.10. Justification des droits

Android a ajouté une sophistication supplémentaire : une application qui demande un droit et qui se le voit refuser peut afficher une explication pour essayer de convaincre l'utilisateur d'accorder le droit.

Quand on constate qu'un droit manque, il faut tester `shouldShowRequestPermissionRationale(String permission)`. Si elle retourne `true`, alors il faut construire un dialogue d'information pour expliquer les raisons à l'utilisateur, puis retenter un `requestPermissions`.

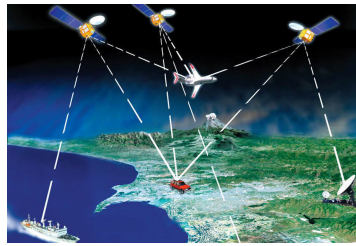


Figure 52: GPS

9.3. Capteurs de position

9.3.1. Présentation

Les tablettes et smartphones sont généralement équipés d'un capteur GPS.

figure 52

La position sur le globe peut être déterminée par triangulation, c'est à dire la mesure des longueurs des côtés du polyèdre partant du capteur et allant vers trois ou quatre satellites de position connue. Les distances sont estimées en comparant des horloges extrêmement précises ($1\mu s$ de décalage = 300m d'écart).

À défaut d'un GPS (droit manquant ou pas de capteur), on peut obtenir une position grossière (*coarse* en anglais) à l'aide des réseaux de téléphonie ou Wifi.

Dans tous les cas, on fait appel à un singleton Java, un `LocationManager` qui gère les capteurs de position, appelés « fournisseurs de position » et identifiés par les constantes `GPS_PROVIDER` et `NETWORK_PROVIDER`.

Le principe est de s'abonner à des événements, donc de programmer un écouteur pour ces événements. Chaque fois que la position aura changé, l'écouteur sera appelé avec la nouvelle position.

Les positions sont représentées par la classe `Location`. C'est essentiellement un triplet (longitude, latitude, altitude).

9.3.2. Utilisation dans Android

La position étant une information sensible, personnelle, il faut demander la permission à l'utilisateur. C'est l'objet de deux droits :

- `Manifest.permission.ACCESS_FINE_LOCATION` pour la position GPS, très précise, donnée par le `GPS_PROVIDER`,
- `Manifest.permission.ACCESS_COARSE_LOCATION` pour la position imprécise du `NETWORK_PROVIDER`.

Rappel : les mettre dans le manifeste et les tester à chaque demande au gestionnaire.

Une fois les permissions obtenues, l'activité peut récupérer le gestionnaire :



```
LocationManager locationManager =  
(LocationManager) getSystemService(Context.LOCATION_SERVICE);
```

9.3.3. Récupération de la position

On peut obtenir la position actuelle, ou la dernière connue par :



```
Location position =  
locationManager.getLastKnownLocation(FOURNISSEUR);
```

- `FOURNISSEUR` vaut `LocationManager.GPS_PROVIDER` ou `LocationManager.NETWORK_PROVIDER`.

Le résultat est une instance de `Location` dont on peut utiliser les *getters* :



```
float lat = location.getLatitude();  
float lon = location.getLongitude();  
float alt = location.getAltitude();
```

9.3.4. Abonnement aux changements de position

Pour l'abonner aux événements :



```
locationManager.requestLocationUpdates(  
    FOURNISSEUR,  
    PERIODICITE, DISTANCE,  
    this);
```

- PERIODICITE donne le temps en millisecondes entre deux événements, mettre 2000 pour toutes les 2 secondes.
- DISTANCE donne la distance minimale en mètres qu'il faut parcourir pour faire émettre des événements.
- this ou un écouteur qui implémente les quatre méthodes de [LocationListener](#).

Appeler `locationManager.removeUpdates(this);` pour cesser de recevoir des événements.

9.3.5. Événements de position

La méthode la plus importante est `onLocationChanged(Location location)`. Son paramètre est la position actuelle détectée par les capteurs. Par exemple :



```
@Override  
public void onLocationChanged(Location location)  
{  
    tvPosition.setText(String.format(Locale.FRANCE,  
        "lon: %.6f\nlat: %.6f\naltitude: %.1f",  
        location.getLongitude(),  
        location.getLatitude(),  
        location.getAltitude()));  
}
```

Les autres méthodes sont `onStatusChanged`, `onProviderEnabled` et `onProviderDisabled` qui peuvent rester vides.

9.3.6. Remarques

Normalement, une activité ne doit demander des positions que lorsqu'elle est active. Quand elle n'est plus visible, elle doit cesser de demander des positions :



```
@Override  
public void onResume() {  
    super.onResume();  
    // s'abonner aux événements GPS  
    if (checkPermission(Manifest.permission.ACCESS_FINE_LOCATION))  
        locationManager.requestLocationUpdates(..., this);  
}  
  
@Override  
public void onPause() {  
    super.onPause();  
    // se désabonner des événements  
    locationManager.removeUpdates(this);  
}
```


9.4. Caméra

9.4.1. Présentation

La quasi totalité des smartphones possède au moins une caméra capable d'afficher en permanence un flot d'images prises à l'instant (*live display* en anglais). Cette caméra est dirigée vers l'arrière de l'écran. On ne se servira pas de la caméra dirigée vers l'avant.

La direction de la caméra est représentée par une constante, ex: `Camera.CameraInfo.CAMERA_FACING_BACK`.

Comme pour la position, l'utilisation de la caméra est soumise à autorisations. Voir le début de ce cours. Elles sont à tester à chaque phase du travail.

NB: on va utiliser une API dépréciée, car elle fournit des méthodes utiles pour la réalité virtuelle qui ne sont pas dans la nouvelle.

9.4.2. Vue SurfaceView

Pour commencer, il faut une vue spécialisée dans l'affichage d'un flot d'images. Cette vue 2D est d'un type spécial, évoqué dans le cours précédent, un `SurfaceView`, voir ce [tutoriel](#). C'est une vue associée à une `Surface` : un mécanisme matériel pour produire des images, ex: caméra ou OpenGL, voir cette [documentation](#) pour comprendre l'architecture.

Donc, le layout de l'activité contient :

```
<SurfaceView
    android:id="@+id/surface_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

9.4.3. Fonctionnement du SurfaceView

Il faut fournir un écouteur à cette vue, via une classe `SurfaceHolder` dont on appelle la méthode `'addCallback` :

```
public class MainActivity extends AppCompatActivity
    implements SurfaceHolder.Callback
{
    private SurfaceView surfaceView;

    void initSurfaceView()
    {
        surfaceView = findViewById(R.layout.surface_view);
        surfaceView.getHolder().addCallback(this);
    }
}
```

La méthode `addCallback` indique que `this` va réagir aux événements de la `SurfaceView` (écouteur à 3 méthodes).

9.4.4. Événements d'un SurfaceHolder

Il faut programmer trois méthodes :

- `surfaceCreated` : ouvrir la caméra
- `surfaceChanged` : paramétrer la dimension de l'écran
- `surfaceDestroyed` : libérer la caméra

D'autre part, il faut gérer les événements de l'activité :

- `onResume` : démarrer l'affichage de la vue caméra
- `onPause` : mettre l'affichage de la caméra en pause

NB: chacune de ces méthodes devra tester les permissions.

9.4.5. Écouteur `surfaceCreated`

La caméra est représentée par une instance de `Camera` (package `android.hardware.Camera`, attention il y a un autre package `android.graphics.Camera`) :

```
public class MainActivity... implements SurfaceHolder.Callback
{
    // caméra arrière
    private Camera camera;

    public void surfaceCreated(SurfaceHolder holder)
    {
        // ouverture de la caméra
        camera =
            Camera.open(Camera.CameraInfo.CAMERA_FACING_BACK);
        // paramétrage, voir la suite ...
    }
}
```

9.4.6. Écouteur `surfaceCreated`, fin

Avant de commencer à afficher les images, il faut modifier le paramètre de l'autofocus :

```
public void surfaceCreated(SurfaceHolder holder)
{
    ...
    Camera.Parameters params = camera.getParameters();
    List<String> focusModes = params.getSupportedFocusModes();
    if (focusModes.contains(Camera.Parameters.FOCUS_MODE_AUTO)) {
        params.setFocusMode(Camera.Parameters.FOCUS_MODE_AUTO);
        camera.setParameters(params);
    }
}
```

Il n'y a pas besoin d'autre chose (reconnaissance faciale, zoom, etc.) pour la réalité augmentée.

9.4.7. Écouteur `surfaceChanged`

Cet écouteur est appelé pour indiquer la taille de la `SurfaceView`. On s'en sert pour configurer la taille des images générées par la caméra. Il faut demander à la caméra ce qu'elle sait faire comme prévisualisations, et on doit choisir parmi cette liste :

```
public void surfaceChanged(SurfaceHolder holder, int format,
                           int width, int height)
{
    Camera.Parameters params = camera.getParameters();
    Camera.Size size = getOptimalPreviewSize(width, height);
    if (size != null) {
        // adopter cette taille d'affichage
        params.setPreviewSize(size.width, size.height);
        camera.setParameters(params);
    }
    ...
}
```


9.4.8. Choix de la prévisualisation

Voici un extrait de `getOptimalPreviewSize` :

```
private Camera.Size getOptimalPreviewSize(int width, int height)
{
    Camera.Parameters params = camera.getParameters();
    List<Camera.Size> sizes = params.getSupportedPreviewSizes();
    for (Camera.Size size : sizes) {
        if ( Math.abs(size.width - width)/width < 0.1 &&
            Math.abs(size.height - height)/height < 0.1) {
            return size;
        }
    }
    return sizes.get(0); // la première si aucune satisfaisante
}
```

Il parcourt toutes les resolutions d'affichage et choisit celle qui est proche de la taille de l'écran.

9.4.9. Suite de surfaceChanged

Un autre réglage doit être fait dans la méthode `surfaceChanged` : prendre en compte l'orientation de l'écran, afin de faire pivoter la caméra également. 

```
public void surfaceChanged(SurfaceHolder holder, int format,
                           int width, int height)
{
    ...
    // orientation de la caméra = orientation de l'écran
    int orientation = getCameraCorrectOrientation();
    camera.setDisplayOrientation(orientation);
    camera.getParameters().setRotation(orientation);
    ...
}
```

9.4.10. Orientation de la caméra

Pour simplifier, on fait pivoter la caméra si son angle n'est pas celui de l'écran : 

```
private int getCameraCorrectOrientation()
{
    Camera.CameraInfo info = new Camera.CameraInfo();
    Camera.getCameraInfo(
        Camera.CameraInfo.CAMERA_FACING_BACK, info);

    int degrees = getWindowRotation();

    return (info.orientation - degrees + 360) % 360;
}
```

Remarquez l'emploi de `getCameraInfo`, notamment son second paramètre. On sent que ce n'est pas tout à fait du Java derrière.

9.4.11. Orientation de l'écran

Il suffit de traduire un identifiant en valeur d'angle : 

```
private int getWindowRotation()
{
    int rotation = surfaceView.getDisplay().getRotation();
    switch (rotation) {
        case Surface.ROTATION_90:    return 90;
        case Surface.ROTATION_180:   return 180;
        case Surface.ROTATION_270:   return 270;
        case Surface.ROTATION_0:     return 0;
        default:                     return 0;
    }
}
```

Cette méthode sert également pour construire le bon changement de repère en réalité augmentée.

9.4.12. Fin de surfaceChanged

Enfin, il reste à activer l'affichage des images :



```
public void surfaceChanged(SurfaceHolder holder, int format,
                           int width, int height)
{
    ...
    // associer la vue et la caméra
    try {
        camera.setPreviewDisplay(holder);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

9.4.13. Écouteur onResume

Cette méthode est appelée quand l'activité est prête à fonctionner. Elle demande à la caméra d'afficher les images :



```
public void onResume()
{
    // test des permissions et de validité de la caméra
    ...
    if (camera == null) return;

    // affichage des images
    camera.startPreview();
}
```

9.4.14. Écouteur onPause

Inversement, onPause est appelé quand l'activité est recouverte par une autre, temporairement ou définitivement. Il faut juste arrêter la prévisualisation :



```
public void onPause()
{
    // test des permissions et de validité de la caméra
    ...
    if (camera == null) return;
```

```
camera.stopPreview();  
}
```

Si l'activité revient au premier plan, le système Android appellera `onResume`. Ces deux fonctions forment une paire. Il en est de même avec le couple `surfaceCreated` et `surfaceDestroyed`.

9.4.15. Écouteur `surfaceDestroyed`

Son travail consiste à fermer la caméra, au contraire de `surfaceCreated` qui l'ouvrait :



```
public void surfaceDestroyed(SurfaceHolder holder)  
{  
    if (camera != null) camera.release();  
    camera = null;  
}
```

9.4.16. Organisation logicielle

Il est préférable de confier la gestion de la caméra à une autre classe, implémentant ces 5 écouteurs. Cela évite de trop charger l'activité. L'activité se contente de relayer les écouteurs `onPause` et `onResume` vers cette classe.

```
public class CameraHelper implements SurfaceHolder.Callback  
{  
    Camera camera;  
  
    // constructeur  
    public CameraHelper(SurfaceView cameraView) {  
        surfaceView.getHolder().addCallback(this);  
    }  
    // les 5 écouteurs ...  
}
```

```
public class MainActivity extends AppCompatActivity  
{  
    private CameraHelper cameraHelper;  
  
    protected void onCreate(Bundle savedInstanceState) {  
        ...  
        SurfaceView cameraView = findViewById(R.id.surface_view);  
        cameraHelper = new CameraHelper(cameraView);  
    }  
  
    public void onResume() {  
        super.onResume();  
        cameraHelper.onResume();  
    }  
}
```

Idem pour `onPause`.

9.5. Capteurs d'orientation

9.5.1. Présentation

On arrive à une catégorie de dispositifs assez disparates, intéressants mais parfois difficiles à utiliser : accéléromètre, altimètre, cardiofréquencemètre, thermomètre, etc.

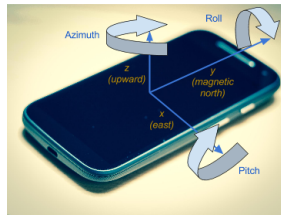


Figure 53: Angles d'Euler

Il faut savoir qu'un smartphone n'est pas un instrument de mesure précis et étalonné. Les valeurs sont assez approximatives (et parfois décevantes, il faut bien l'avouer).

On s'intéresse aux capteurs qui indiquent l'orientation, c'est à dire une information sur la direction dans laquelle est orientée la tablette par rapport au nord. Ça peut être un vecteur 3D orienté comme la face supérieure de l'écran, un triplet d'angles, une matrice de rotation ou un quaternion.

9.5.2. Angles d'Euler

L'information d'apparence la plus simple est un triplet d'angles (cap, tangage, roulis) :

figure 53

- cap ou azimuth (*azimuth* en anglais) = angle à plat donnant la direction par rapport au nord,
- tangage (*pitch*) = angle de bascule avant/arrière,
- roulis (*roll*) = angle d'inclinaison latérale.

Le problème de ces angles est le blocage de Cardan : quand le tangage vaut 90° , que signifie le cap ?

9.5.3. Matrice de rotation

Une matrice représente un changement de repère. Elle permet de calculer les coordonnées d'un point ou d'un vecteur qui sont exprimées dans un repère de départ, les obtenir dans un autre repère qui est transformé par rapport à celui de départ.

Le calcul des coordonnées d'arrivée se fait à l'aide d'un produit entre la matrice et les coordonnées de départ. Android offre tout ce qu'il faut pour manipuler les matrices dans le package `android.opengl.Matrix` (faite pour OpenGL) et on n'a jamais à construire une matrice nous-même.

Une matrice est le meilleur moyen de représenter une rotation, il n'y a pas de blocage de Cardan, mais ça semble rebutant d'y faire appel.

Voyons d'abord comment récupérer des mesures, puis comment les utiliser.

9.5.4. Accès au gestionnaire

Comme pour la position GPS, l'activité doit s'adresser à un gestionnaire :



```
// gestionnaire
private SensorManager sensorManager;

public void onCreate(...) {
    sensorManager =
        (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    if (sensorManager == null)
        throw new UnsupportedOperationException("aucun gestionnaire de capteurs");
    ...
}
```

Il n'y a aucune permission à demander pour les capteurs. Ils ne fournissent pas des informations jugées sensibles.

9.5.5. Accès aux capteurs

Chaque capteur est représenté par une instance de `Sensor` et il y en a de plusieurs types identifiés par un symbole, comme `TYPE_ROTATION_VECTOR`, `TYPE_ACCELEROMETER`, `TYPE_MAGNETIC_FIELD`. Il est possible d'ouvrir plusieurs capteurs en même temps :

```
private Sensor rotationSensor;
private Sensor accelerometerSensor;

...
rotationSensor = sensorManager.getDefaultSensor(
    Sensor.TYPE_ROTATION_VECTOR);
accelerometerSensor = sensorManager.getDefaultSensor(
    Sensor.TYPE_ACCELEROMETER);
```

Il manque des tests pour savoir si certains sont null.

9.5.6. Abonnement aux mesures

Comme pour les positions, on demande au gestionnaire de nous prévenir à chaque fois qu'une mesure est faite :

```
public void onResume() {
    super.onResume();
    sensorManager.registerListener(this, rotationSensor,
        SensorManager.SENSOR_DELAY_GAME);
    sensorManager.registerListener(this, accelerometerSensor,
        SensorManager.SENSOR_DELAY_NORMAL);
}
public void onPause() {
    super.onPause();
    sensorManager.unregisterListener(this);
}
```

Le troisième paramètre est un code indiquant la périodicité souhaitée (très fréquente ou moins).

9.5.7. Réception des mesures

L'abonnement implique la programmation d'un écouteur :

```
public void onSensorChanged(SensorEvent event)
{
    switch (event.sensor.getType()) {
        case Sensor.TYPE_ROTATION_VECTOR:
            ... utiliser event.values en tant que rotation
            break;
        case Sensor.TYPE_ACCELEROMETER:
            ... utiliser event.values en tant que déplacement
            break;
    }
}
```

Les données `event.values` sont un tableau de `float` qui dépend du capteur. Il y a un calcul spécifique et la [documentation](#) n'est pas toujours assez précise.

9.5.8. Atténuation des oscillations

La plupart des capteurs fournissent une information bruitée : les mesures oscillent aléatoirement autour d'une moyenne. Il faut *filtrer* les valeurs brutes de `event.values` à l'aide d'un algorithme mathématique : un filtre passe-bas.

Cela consiste à calculer $V_{ok} = \alpha * V_{brute} + (1 - \alpha) * V_{ok}$ avec α étant un coefficient assez petit, entre 0.01 et 0.2 par exemple, à choisir en fonction du capteur, de la vitesse d'échantillonnage et de la volonté d'amortissement voulu.

Cette formule mélange la valeur brute du capteur avec la valeur précédemment mesurée. Si α est très petit, l'amortissement est très lent mais la valeur est stable. Inversement si α est assez grand, l'amortissement est faible, la valeur peut encore osciller, mais c'est davantage réactif.

Le filtrage se fait facilement avec une petite méthode :

```
private void lowPass(final float[] input, float[] output)
{
    final float alpha = 0.02;
    for (int i=0; i<input.length; i++) {
        output[i] = output[i] + alpha*(input[i] - output[i]);
    }
}
```

Le calcul a été programmé pour optimiser les calculs.

Voici comment on utilise cette méthode :

```
private float[] gravity = new float[3];
lowPass(0.05f, event.values, gravity);
```

9.5.9. Orientation avec TYPE_ROTATION_VECTOR

C'est le meilleur capteur pour fournir l'information d'orientation nécessaire pour la réalité augmentée. Il permet de calculer la matrice de transformation en un seul appel de fonction. Nous en avons besoin pour calculer les coordonnées écran de points qui sont dans le monde 3D réel et pivotés à cause des mouvements de l'écran.

```
float[] rotationMatrix = new float[16];
case Sensor.TYPE_ROTATION_VECTOR:
    SensorManager.getRotationMatrixFromVector(
        rotationMatrix, event.values);
```

La variable `rotationMatrix` est une matrice 4x4. Elle représente la rotation à appliquer sur un point 3D pour l'amener dans le repère du smartphone, donc exactement ce qu'il nous faut.

9.5.10. Orientation sans TYPE_ROTATION_VECTOR

Le capteur de rotation est le plus précis et celui qui donne la meilleure indication de l'orientation du smartphone. Hélas, tous n'ont pas ce capteur. Ce n'est pas une question de version d'Android, mais d'équipement électronique interne (prix).

Quand ce capteur n'est pas disponible, Android propose d'utiliser deux autres capteurs : l'accéléromètre `TYPE_ACCELEROMETER` et le capteur de champ magnétique terrestre (une boussole 3D) `TYPE_MAGNETIC_FIELD`.

L'accéléromètre mesure les accélérations 3D auxquelles est soumis le capteur. La pesanteur est l'une de ces accélérations, et elle est constante. Si on arrive à filtrer les mesures avec un filtre passe-bas, on verra où est le bas ; c'est la direction de l'accélération de $9.81m.s^{-2}$.

La boussole nous donne une autre direction 3D, celle du nord relativement au smartphone. En effet, le capteur géomagnétique est capable d'indiquer l'intensité du champ magnétique dans toutes les directions autour du smartphone.

Android propose même une méthode pour lier l'accélération et le champ magnétique, et en déduire l'orientation 3D du téléphone. Il faut mémoriser les valeurs fournies par chacun des deux capteurs, après filtrage.


```
private float[] gravity = new float[3];  
private float[] geomagnetic = new float[3];
```

9.5.11. Orientation sans TYPE_ROTATION_VECTOR, fin

Ensuite, dans `onSensorChanged` :



```
case Sensor.TYPE_ACCELEROMETER:  
    // filtre sur les valeurs  
    lowPass(0.05f, event.values, gravity);  
    // calculer la matrice de rotation  
    SensorManager.getRotationMatrix(rotationMatrix, null,  
        gravity, geomagnetic);  
    break;  
case Sensor.TYPE_MAGNETIC_FIELD:  
    // filtre sur les valeurs  
    lowPass(0.05f, event.values, geomagnetic);  
    // calculer la matrice de rotation  
    SensorManager.getRotationMatrix(rotationMatrix, null,  
        gravity, geomagnetic);  
    break;
```

9.5.12. Orientation avec TYPE_ORIENTATION

Quand, enfin, aucun de ces précédents capteurs n'est disponibles, on peut tenter d'utiliser le plus ancien, mais aussi le plus imprécis, un capteur d'orientation. Les valeurs qu'il fournit sont des angles d'Euler, et voici comment les combiner pour obtenir une matrice de rotation :



```
case Sensor.TYPE_ORIENTATION:  
    Matrix.setIdentityM(rotationMatrix, 0);  
    Matrix.rotateM(rotationMatrix, 0, event.values[1], 1,0,0);  
    Matrix.rotateM(rotationMatrix, 0, event.values[2], 0,1,0);  
    Matrix.rotateM(rotationMatrix, 0, event.values[0], 0,0,1);
```

9.6. Réalité augmentée

9.6.1. Objectif

On voudrait visualiser des points d'intérêt (*Point(s) Of Interest*, POI en anglais) superposés en temps réel et en 3D sur l'image de la caméra.

mettre une copie écran, mais de préférence avec un joli fond... faire une photo devant Open ?

9.6.2. Assemblage

Il faut assembler plusieurs techniques :

- la caméra nous fournit l'image de fond.
- Une vue est superposée pour dessiner les icônes et textes des POIs. C'est une vue de dessin 2D comme dans le cours précédent.
- Le GPS donne la position sur le globe terrestre permettant d'obtenir la direction relative des POIs.
- Le capteur d'orientation permet de déterminer la position écran des POIs, s'ils sont visibles.

Le lien entre les trois derniers points se fait avec une matrice de transformation. Le but est de transformer des coordonnées 3D absolues (sur le globe terrestre) en coordonnées 2D de pixels sur l'écran.

9.6.3. Transformation des coordonnées

Ce sont des mathématiques assez complexes, les mêmes que pour définir une caméra avec OpenGL :

- Déterminer l'orientation 3D du smartphone sous forme d'une matrice : R_1 , elle vient du capteur d'orientation.
- Déterminer la rotation de l'écran du smartphone (s'il est en portrait ou paysage, la visualisation est renversée), c'est également une rotation : R_2 , elle vient de la caméra.
- Déterminer le champ de vision de la caméra, c'est une projection en perspective : P , elle vient de la caméra.

Les points 3D sont à transformer par : $P' = M * P$ avec $M = P * R_2 * R_1$. Le point P' peut être dessiné sur l'écran si ses 3 coordonnées sont entre -1 et +1 (en fait, c'est un peu plus complexe car ce sont des coordonnées homogènes).

Pour cela, il faut connaître les coordonnées P du POI à dessiner. On dispose de ses coordonnées géographiques, longitude, latitude et altitude.

Il existe un repère global 3D attaché à la Terre. On l'appelle **ECEF** *earth-centered, earth-fixed*. C'est un repère dont l'origine est le centre de la Terre, l'axe X passe par l'équateur et le méridien de Greenwich, l'axe Y est 90° à l'est.

Connaissant le rayon de la Terre, et son excentricité (elle est aplatie), on peut transformer tout point géographique en point 3D ECEF. Le calcul est complexe, hors de propos ici, voir [cette page](#).

9.6.4. Transformation des coordonnées, fin

On n'a pas encore tué la bête : les coordonnées ECEF ne sont pas utilisables directement pour notre application, en effet, la rotation R_1 est relative au point où nous nous trouvons en particulier à la direction du nord et de l'est locale, et surtout à l'orientation du smartphone.

Il faut encore transformer les coordonnées ECEF dans un repère local appelé ENU (*East, North, Up*). C'est un repère 3D lié à l'emplacement local, voir [ce lien](#).

L'algorithme devient :

- Transformer la position du smartphone dans le repère ECEF,
- Transformer la position du POI dans le repère ECEF,
- Calculer les coordonnées relatives ENU du POI par rapport au smartphone, c'est P à multiplier par M .

9.6.5. Dessin du POI

Il reste à dessiner un bitmap et écrire le nom du POI sur l'écran, à l'emplacement désigné par P' .

La projection fournit des coordonnées entre -1 et +1 qu'il faut modifier selon le système de coordonnées de l'écran. Le coin (0,0) est en haut et à gauche. Il faut tenir compte de la largeur et la hauteur de l'écran.

Le tout est assez difficile à mettre au point et demande beaucoup de rigueur dans les calculs.

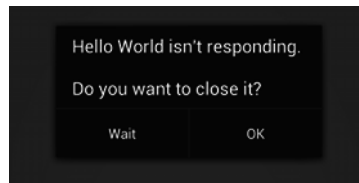


Figure 54: Application bloquée

Semaine 10

Dessin 2D interactif et Cartes

Le cours de cette semaine concerne le dessin de cartes géographiques en ligne. Il est lié au cours 7 et en faisait partie auparavant.

On commence par les *AsyncTasks* qui sont nécessaires pour faire des calculs longs comme ceux de l'affichage d'une carte, ou des requêtes réseau.

10.1. AsyncTask

10.1.1. 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 54

10.1.2. Tâches asynchrones

Pourtant dans certains cas, une *callback* peut durer longtemps :

- gros calcul
- 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`.

10.1.3. 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.

10.1.4. Structure d'une AsyncTask

Une tâche asynchrone est définie par au moins deux méthodes :

doInBackground C'est le corps du traitement. Cette méthode est lancée dans son propre *thread*. Elle peut durer aussi longtemps que nécessaire.

onPostExecute Elle est appelée quand `doInBackground` a fini. On peut lui faire afficher des résultats sur l'interface. Elle s'exécute dans le *thread* de l'interface, alors elle ne doit pas durer longtemps.

10.1.5. Autres méthodes d'une AsyncTask

Trois autres méthodes peuvent être définies :

Constructeur Il permet de passer des paramètres à la tâche. On les stocke dans des variables d'instance privées et `doInBackground` peut y accéder.

onPreExecute Cette méthode est appelée avant `doInBackground`, dans le *thread* principal. Elle sert à initialiser les traitements. Par exemple on peut préparer une barre d'avancement (`ProgressBar`).

onProgressUpdate Cette méthode permet de mettre à jour l'interface, p. ex. la barre d'avancement. Pour ça, `doInBackground` doit appeler `publishProgress`.

10.1.6. 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;`).

10.1.7. 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)
```

10.1.8. 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.

10.1.9. 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;
    }
}
```

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] );
}
```

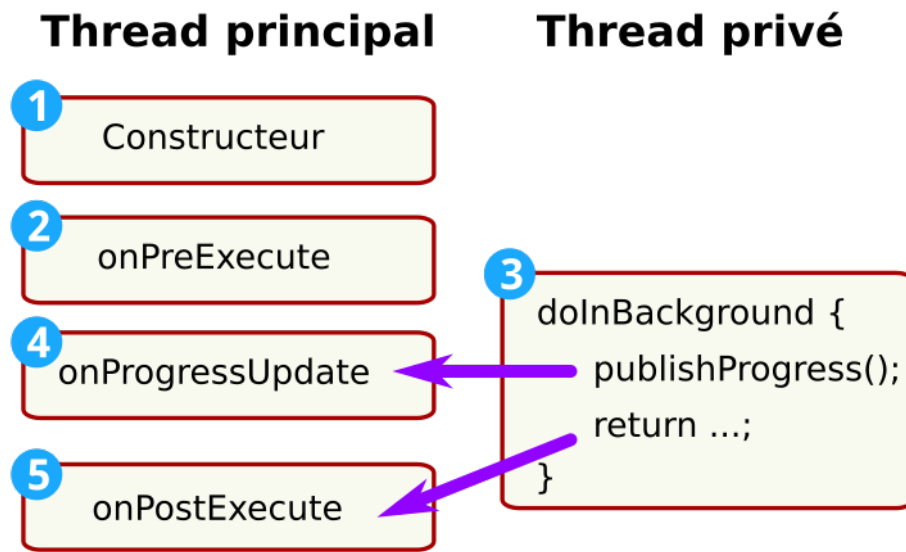


Figure 55: Méthodes d'un AsyncTask

10.1.10. 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 = 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.

10.1.11. Schéma récapitulatif

figure 55

10.1.12. 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`.

10.1.13. 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(...);
```

10.1.14. 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
        ...
    }
}
```

10.1.15. Fuite de mémoire

Certaines situations posent un problème : lorsque l'AsyncTask conserve une référence sur une activité ou un `ProgressBar`, par exemple pour afficher un avancement. Si jamais cet objet est supprimé avant la tâche, alors sa mémoire reste marquée comme occupée et jamais libérée.

Dans un tel cas, il faut que l'AsyncTask conserve l'objet sous la forme d'une `WeakReference<classe>`. C'est un dispositif qui stocke un objet sans empêcher sa libération mémoire s'il n'est plus utilisé par ailleurs. On utilise la méthode `get()` pour récupérer l'objet stocké, et c'est `null` s'il a déjà été libéré.

Voici une application de cette solution : 

```
private class PrevisionPluie extends AsyncTask... {

    private final WeakReference<ProgressBar> wrProgressBar;

    public PrevisionPluie(ProgressBar progressBar) {
        wrProgressBar = new WeakReference<>(progressBar);
    }
}
```

```
}  
  
protected void onProgressUpdate(...) {  
    ProgressBar progressBar = wrProgressBar.get();  
    if (progressBar == null) return;  
    ...  
}
```

10.1.16. 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](#)

10.1.17. 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.

10.2. OpenStreetMap

10.2.1. Présentation

Au contraire de Google Maps, OSM est vraiment libre et OpenSource, et il se programme extrêmement facilement. Voir la figure 56, page 185.

10.2.2. Documentation

Nous allons utiliser deux librairies :

- [OSMdroid](#) : c'est la librairie de base, super mal documentée. Attention à ne pas confondre avec un site de piraterie.

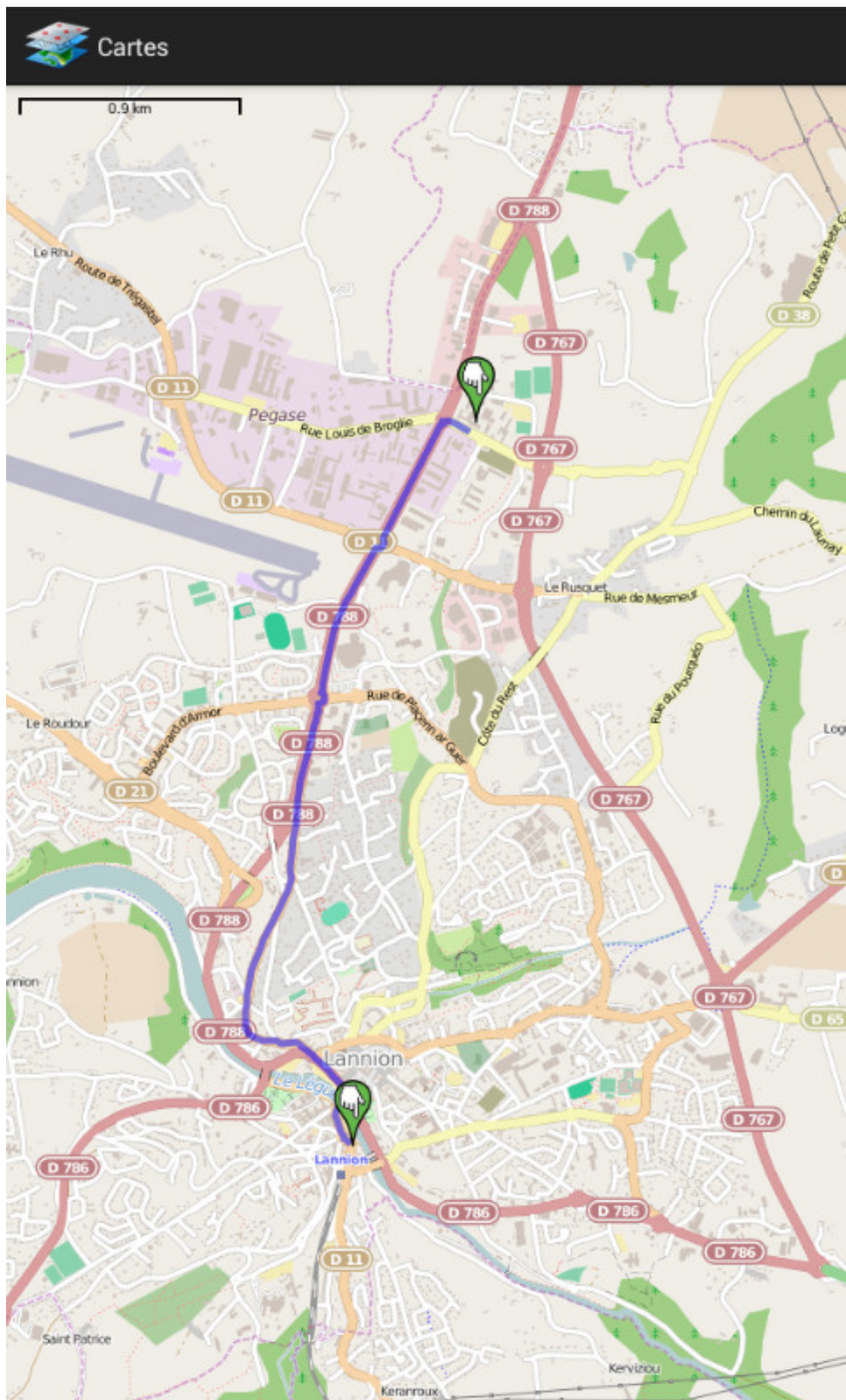


Figure 56: Google Maps

- [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.

10.2.3. 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 TP7 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 ».

10.2.4. 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.

10.2.5. 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 = findViewById(R.id.map);
    mMap.setMultiTouchControls(true);
    mMap.setBuiltInZoomControls(true);
}
```

10.2.6. 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.

10.2.7. 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);
```

10.2.8. 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.

10.2.9. 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.

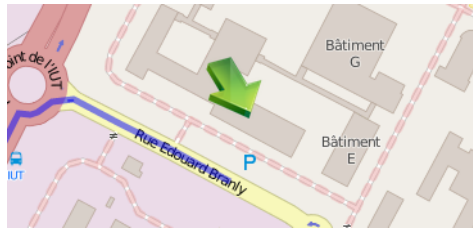


Figure 57: Marqueur personnalisé

10.2.10. 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 57

10.2.11. 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*.


10.2.12. 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 TP7)

10.2.13. 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.

10.2.14. 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();  
}
```


10.2.15. 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.

10.2.16. 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());
```

10.2.17. 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.

10.2.18. 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" />
```