

Programmation mobile avec Android

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

février-mars 2023

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 14/03/2023 à 18:30

Table des matières

1	Environnement de développement	17
1.1	Introduction	17
1.1.1	Qu'est-ce qu'Android ?	17
1.1.2	Historique	17
1.1.3	Remarque sur les versions d'API	19
1.1.4	Distribution des versions	19
1.1.5	Remarques diverses	19
1.1.6	Programmation d'applications	20
1.1.7	Applications natives	20
1.1.8	Kotlin	20
1.1.9	Exemple : objet pouvant être null	20
1.1.10	Pas de Kotlin pour ce cours	21
1.2	SDK Android et Android Studio	21
1.2.1	SDK et Android Studio	21
1.2.2	Android Studio	22

1.2.3	SDK Manager	22
1.2.4	Choix des éléments du SDK	22
1.3	Création d'une application	22
1.3.1	Assistant de création d'application	22
1.3.2	Modèle d'application	25
1.3.3	Résultat de l'assistant	25
1.3.4	Fenêtre du projet	25
1.3.5	Éditeurs spécifiques	27
1.3.6	Exemple <code>res/layout/main.xml</code>	27
1.3.7	Source XML sous-jacent	27
1.3.8	Reconstruction du projet	28
1.3.9	Gradle	28
1.3.10	Structure d'un projet AndroidStudio	28
1.3.11	Utilisation de bibliothèques	29
1.4	Exécution de l'application	29
1.4.1	Simulateur ou smartphone	29
1.4.2	Assistant de création d'une tablette virtuelle	29
1.4.3	Caractéristiques d'un AVD	31
1.4.4	Lancement d'une application	31
1.4.5	Application sur l'AVD	31
1.5	Communication AVD - Android Studio	31
1.5.1	Fenêtres Android	31
1.5.2	Fenêtre <code>Logcat</code>	31
1.5.3	Filtrage des messages	33
1.5.4	Émission d'un message vers <code>LogCat</code>	33
1.5.5	Logiciel ADB	33
1.5.6	Mode d'emploi de ADB	34
1.5.7	Système de fichiers Android	34
1.6	Création d'un paquet installable	35
1.6.1	Paquet	35
1.6.2	Signature d'une application	35
1.6.3	Création du <i>keystore</i>	35
1.6.4	Création d'une clé	35
1.6.5	Création du paquet	35
1.6.6	Et voilà	35

2	Création d'interfaces utilisateur	36
2.1	Présentation rapide des concepts	38
2.1.1	Composition d'une application	38
2.1.2	Structure d'une interface utilisateur	38
2.1.3	Création d'une interface	39
2.1.4	Création d'un écran	39
2.2	Ressources	39
2.2.1	Définition	39
2.2.2	Identifiant de ressource	40
2.2.3	Génération de la classe R	40
2.2.4	La classe R	40
2.2.5	Rappel sur la structure d'un fichier XML	41
2.2.6	Espaces de nommage dans un fichier XML	41
2.2.7	Ressources de type chaînes	41
2.2.8	Traduction des chaînes (<i>localisation</i>)	42
2.2.9	Emploi des ressources texte dans un programme	42
2.2.10	Emploi des ressources texte dans une interface	42
2.2.11	Images : <code>R.drawable.nom</code>	43
2.2.12	Tableau de chaînes : <code>R.array.nom</code>	43
2.2.13	Autres	43
2.3	Mise en page (<i>layouts</i>)	44
2.3.1	Structure d'une interface Android	44
2.3.2	Arbre des vues	44
2.3.3	Création d'une interface par programme	44
2.3.4	Ressources de type <i>layout</i>	45
2.3.5	Identifiants et vues	45
2.3.6	<code>@id/nom</code> ou <code>@+id/nom</code> ?	45
2.3.7	Paramètres de positionnement	46
2.3.8	Paramètres obligatoires	46
2.3.9	Autres paramètres géométriques	47
2.3.10	Marges et remplissage	47
2.3.11	Groupe de vues <code>LinearLayout</code>	47
2.3.12	Pondération des tailles	48
2.3.13	Exemple de poids différents	48

2.3.14	Groupe de vues <code>TableLayout</code>	49
2.3.15	Largeur des colonnes d'un <code>TableLayout</code>	49
2.3.16	Groupe de vues <code>RelativeLayout</code>	49
2.3.17	Utilisation d'un <code>RelativeLayout</code>	50
2.3.18	Autres groupements	50
2.4	Composants d'interface	50
2.4.1	Vues	50
2.4.2	<code>TextView</code>	50
2.4.3	<code>Button</code>	51
2.4.4	Bascules	51
2.4.5	<code>EditText</code>	51
2.4.6	Autres vues	52
2.4.7	C'est tout	52

3 Vie d'une application 53

3.1	Applications et activités	53
3.1.1	Présentation	53
3.1.2	Déclaration d'une application	53
3.1.3	Démarrage d'une application	54
3.1.4	Démarrage d'une activité et <code>Intents</code>	54
3.1.5	Lancement d'une activité par programme	54
3.1.6	Lancement d'une application Android	54
3.1.7	Lancement d'une activité d'une autre application	55
3.1.8	Autorisations d'une application	55
3.1.9	Sécurité des applications (pour info)	55
3.2	Applications	56
3.2.1	Fonctionnement d'une application	56
3.2.2	Navigation entre activités	56
3.2.3	Lancement avec ou sans retour possible	56
3.2.4	Terminaison d'une activité	58
3.2.5	Lancement avec attente de résultat	58
3.2.6	Méthode <code>onActivityResult</code>	58
3.2.7	Lancement avec attente, version améliorée	59
3.2.8	Lanceur d'activité	59

3.2.9	Écouteur de retour d'activité	59
3.2.10	Transport d'informations dans un Intent	60
3.2.11	Extraction d'informations d'un Intent	60
3.2.12	Contexte d'application	60
3.2.13	Définition d'un contexte d'application	60
3.2.14	Définition d'un contexte d'application, fin	61
3.3	Activités	61
3.3.1	Présentation	61
3.3.2	Cycle de vie d'une activité	62
3.3.3	Événements de changement d'état	62
3.3.4	Squelette d'activité	62
3.3.5	Terminaison d'une activité	63
3.3.6	Pause d'une activité	63
3.3.7	Arrêt d'une activité	63
3.3.8	Enregistrement de valeurs d'une exécution à l'autre	64
3.3.9	Restaurer l'état au lancement	64
3.4	Vues et activités	64
3.4.1	Obtention des vues	64
3.4.2	Mode d'emploi des ViewBindings	65
3.4.3	Génération des ViewBindings	66
3.4.4	Propriétés des vues	66
3.4.5	Actions de l'utilisateur	66
3.4.6	Définition d'un écouteur	67
3.4.7	Écouteur privé anonyme	67
3.4.8	Écouteur privé	68
3.4.9	L'activité elle-même en tant qu'écouteur	68
3.4.10	Distinction des émetteurs	69
3.4.11	Écouteur référence de méthode	69
3.4.12	Événements des vues courantes	70
3.4.13	C'est fini pour aujourd'hui	70

4	Application liste	71
4.1	Présentation	71
4.1.1	Principe général	71
4.1.2	Schéma global	72
4.1.3	Une classe pour représenter les items	72
4.1.4	Données initiales	73
4.1.5	Copie dans un <code>ArrayList</code>	73
4.1.6	Rappels sur le container <code>List<type></code>	73
4.1.7	Données initiales dans les ressources	74
4.1.8	Remarques	74
4.2	Affichage de la liste	75
4.2.1	Activité	75
4.2.2	Mise en place du layout d'activité	75
4.3	Adaptateurs et ViewHolders	76
4.3.1	Relations entre la vue et les données	76
4.3.2	Concepts	76
4.3.3	Recyclage des vues	76
4.3.4	ViewHolders	77
4.3.5	Exemple de ViewHolder	77
4.3.6	Rôle d'un adaptateur	78
4.3.7	Définition d'un adaptateur	78
4.3.8	Constructeur d'un adaptateur	78
4.3.9	Méthodes à ajouter	79
4.4	Configuration de l'affichage	80
4.4.1	Optimisation du défilement	80
4.4.2	LayoutManager	80
4.4.3	LayoutManager dans le layout.xml	80
4.4.4	Disposition en tableau	81
4.4.5	Disposition en blocs empilés	81
4.4.6	Séparateur entre items	82
4.5	Actions sur la liste	82
4.5.1	Présentation	82
4.5.2	Modification des données	82
4.5.3	Défilement vers un élément	82

4.5.4	Clic sur un élément	83
4.5.5	Notre écouteur de clics	83
4.5.6	Schéma récapitulatif	85
4.5.7	Ouf, c'est fini	85
5	Ergonomie	86
5.1	Barre d'action et menus	86
5.1.1	Barre d'action	86
5.1.2	Réalisation d'un menu	86
5.1.3	Spécification d'un menu	87
5.1.4	Icônes pour les menus	87
5.1.5	Écouteur pour afficher le menu	87
5.1.6	Réactions aux sélections d'items	88
5.1.7	Menus en cascade	89
5.2	Menus contextuels	89
5.2.1	Menus contextuels	89
5.2.2	<i>View Holder</i> écouteur de menu	90
5.2.3	Écouteur dans l'activité	91
5.3	Annonces et dialogues	91
5.3.1	Annonces : <i>toasts</i>	91
5.3.2	Dialogues	92
5.3.3	Dialogue d'alerte	92
5.3.4	Boutons et affichage d'un dialogue d'alerte	92
5.3.5	Dialogues personnalisés	93
5.3.6	Création d'un dialogue	93
5.3.7	Affichage du dialogue	93
5.4	Fragments et activités	94
5.4.1	Fragments	94
5.4.2	Tablettes, smartphones...	94
5.4.3	Différents types de fragments	94
5.4.4	Structure d'un fragment	95
5.4.5	Menus de fragments	96
5.4.6	Intégrer un fragment dans une activité	96
5.4.7	Fragments statiques dans une activité	96

5.4.8	Disposition selon la géométrie de l'écran	97
5.4.9	Changer la disposition selon la géométrie	97
5.4.10	Deux dispositions possibles	98
5.4.11	Communication entre Activité et Fragments	98
5.4.12	Interface pour un écouteur	98
5.4.13	Écouteur de l'activité	99
5.4.14	Relation entre deux classes à méditer	99
5.5	Préférences d'application	100
5.5.1	Illustration	100
5.5.2	Présentation	100
5.5.3	Définition des préférences	100
5.5.4	Explications	101
5.5.5	Accès aux préférences	101
5.5.6	Préférences chaînes et nombres	102
5.5.7	Modification des préférences par programme	102
5.5.8	Affichage des préférences	102
5.5.9	Fragment pour les préférences	103
5.6	Bibliothèque support	103
5.6.1	Compatibilité des applications	103
5.6.2	Compatibilité des versions Android	103
5.6.3	Bibliothèque support	104
5.6.4	Anciennes versions de l'Android Support Library	104
5.6.5	Une seule pour les gouverner toutes	104
5.6.6	Une seule pour les gouverner toutes, fin	104
5.6.7	Mode d'emploi	105
5.6.8	Programmation	105
5.6.9	Il est temps de faire une pause	105
6	Realm	106
6.1	Plugin Lombok	106
6.1.1	Présentation	106
6.1.2	Exemple	106
6.1.3	Placement des annotations	107
6.1.4	Nommage des champs	107

6.1.5	Installation du plugin Lombok	107
6.2	Realm	108
6.2.1	Définition de Realm	108
6.2.2	Autre ORM sur Android : <i>Room</i>	108
6.2.3	Realm vs les autres ORM	108
6.2.4	Configuration d'un projet Android avec Realm Legacy	108
6.2.5	Initialisation d'un Realm par l'application	109
6.2.6	Ouverture d'un Realm dans chaque activité	109
6.2.7	Fermeture du Realm	109
6.2.8	Autres modes d'ouverture du Realm	110
6.3	Modèles de données Realm	110
6.3.1	Définir une table	110
6.3.2	Table Realm et Lombok	111
6.3.3	Types des colonnes	111
6.3.4	Empêcher les valeurs null	111
6.3.5	Définir une clé primaire	111
6.3.6	Définir une relation simple	112
6.3.7	Relation multiple	112
6.3.8	Migration des données	112
6.4	Création de n-uplets	113
6.4.1	Résumé	113
6.4.2	Création de n-uplets par createObject	113
6.4.3	Création de n-uplets par new	113
6.4.4	Modification d'un n-uplet	113
6.4.5	Suppression de n-uplets	114
6.5	Requêtes sur la base	114
6.5.1	Résumé	114
6.5.2	Sélections	114
6.5.3	Conditions simples	114
6.5.4	Nommage des colonnes	115
6.5.5	Librairie <i>RealmFieldNamesHelper</i>	115
6.5.6	Conjonctions de conditions	116
6.5.7	Disjonctions de conditions	116
6.5.8	Négations	116

6.5.9	Classement des données	117
6.5.10	Agrégation des résultats	117
6.5.11	Jointures 1-1	117
6.5.12	Jointures 1-N	117
6.5.13	Jointures inverses	118
6.5.14	Jointures inverses, explications	118
6.5.15	Jointures inverses, exemple	118
6.5.16	Suppression par une requête	119
6.6	Requêtes et adaptateurs de listes	119
6.6.1	Adaptateur Realm pour un <code>RecyclerView</code>	119
6.6.2	Adaptateur Realm, fin	120
6.6.3	Mise en place de la liste	120
6.6.4	Réponses aux clics sur la liste	120
6.6.5	C'est la fin	121

7 Dessin 2D interactif 122

7.1	Dessin en 2D	122
7.1.1	Objectif : cette application	122
7.1.2	Principes	122
7.1.3	Layout pour le dessin	123
7.1.4	Méthode <code>onDraw</code>	123
7.1.5	Méthodes de la classe <code>Canvas</code>	123
7.1.6	Représentation des couleurs dans Android	124
7.1.7	Peinture <code>Paint</code>	124
7.1.8	Quelques accesseurs de <code>Paint</code>	124
7.1.9	Motifs	125
7.1.10	Shaders	125
7.1.11	Quelques remarques	126
7.1.12	« Dessinables »	126
7.1.13	Dessinables » vectoriels	126
7.1.14	Variantes	127
7.1.15	Utilisation d'un <code>Drawable</code>	127
7.1.16	Enregistrer un dessin dans un fichier	127
7.2	Interactions avec l'utilisateur	128

7.2.1	Écouteurs pour les touchers de l'écran	128
7.2.2	Modèle de gestion des actions	128
7.2.3	Automate pour gérer les actions	128
7.2.4	Programmation d'un automate	129
7.3	Boîtes de dialogue spécifiques	130
7.3.1	Sélecteur de couleur	130
7.3.2	Version simple	130
7.3.3	Concepts	130
7.3.4	Fragment de dialogue	131
7.3.5	Méthode <code>onCreateDialog</code>	131
7.3.6	Vue personnalisée dans le dialogue	131
7.3.7	Layout de cette vue	132
7.3.8	Écouteurs	132
7.3.9	Affichage du dialogue	133
8	Test logiciel	134
8.1	Introduction	134
8.1.1	Principe de base	134
8.1.2	Précision des nombres	134
8.1.3	Limitations	135
8.1.4	Généralisation des tests	135
8.2	Tests unitaires	135
8.2.1	Programmation des tests unitaires	135
8.2.2	Exécution des tests unitaires	136
8.2.3	Ouvrir une classe aux tests	136
8.2.4	JUnit4	136
8.2.5	Explications	137
8.2.6	Remarque : <code>import static</code> en Java	137
8.2.7	Assertions JUnit	138
8.2.8	Affichage d'un message d'erreur	138
8.2.9	Vérification de plusieurs assertions	138
8.2.10	Vérification des exceptions	139
8.2.11	Vérification de la durée d'exécution	139
8.3	Assertions complexes avec Hamcrest	140

8.3.1	Assertions Hamcrest	140
8.3.2	Catalogue des correspondants Hamcrest	140
8.3.3	Importation de Hamcrest	141
8.4	Patron Arrange-Act-Assert	142
8.4.1	Organisation des tests	142
8.4.2	Préparation des données (<i>arrange</i>)	142
8.4.3	Préparation des données avant chaque test	142
8.4.4	Préparation des données avant l'ensemble des tests	143
8.4.5	Clôture de tests	143
8.4.6	« Règles » de test	144
8.4.7	Implémentation de l'interface TestRule	144
8.4.8	Utilisation d'une règle	144
8.5	Tests paramétrés	145
8.5.1	Utilité des tests paramétrés	145
8.5.2	Exemple de test paramétré	145
8.5.3	Fourniture des paramètres	145
8.5.4	Importation de JUnitParams	146
8.6	Tests d'intégration	146
8.6.1	Introduction	146
8.6.2	Interface à la place d'une classe	146
8.6.3	Simulation d'une interface avec Mockito	147
8.6.4	Apprentissage de résultats	147
8.6.5	Apprentissage généralisé	147
8.6.6	<i>Matchers</i> pour Mockito	148
8.6.7	Autre syntaxe	148
8.6.8	Simulation pour une autre classe	148
8.6.9	Surveillance d'une classe	149
8.6.10	Installation de Mockito	149
8.7	Tests d'intégration Android sans AVD	150
8.7.1	Présentation	150
8.7.2	Installation de Robolectric	150
8.7.3	Lancement d'une activité par Robolectric	150
8.7.4	Emploi de Robolectric	151
8.7.5	Remarques	151

8.8	Tests sur AVD	151
8.8.1	Définition	151
8.8.2	Directives Espresso	152
8.8.3	<i>ViewMatchers</i> d'Espresso	152
8.8.4	<i>ViewActions</i> d'Espresso	152
8.8.5	Tests sur des listes	153
8.8.6	Test sur un spinner	153
8.8.7	Installation de Espresso	153
8.8.8	Classe de test	153
8.8.9	Initialisation des tests	154
8.8.10	Structure des tests	154
8.8.11	Écritures dans les vues de l'activité	155
8.8.12	C'est la fin du cours et du module	155
9	Capteurs	156
9.1	Réalité augmentée	156
9.1.1	Définition	156
9.1.2	Applications	156
9.1.3	Principes	156
9.1.4	Réalité augmentée dans Android	157
9.2	Permissions Android	157
9.2.1	Concepts	157
9.2.2	Permissions dans le manifeste	157
9.2.3	Raffinement de certaines permissions	158
9.2.4	Demandes de permissions à la volée	158
9.2.5	Test d'une autorisation	158
9.2.6	Demande d'une autorisation	159
9.2.7	Préférences d'application	159
9.2.8	Dialogue de demande de droits	159
9.2.9	Affichage du dialogue	160
9.2.10	Justification des demandes de droits	160
9.3	Capteurs de position	160
9.3.1	Présentation	160
9.3.2	Utilisation dans Android	161

9.3.3	Récupération de la position	161
9.3.4	Abonnement aux changements de position	162
9.3.5	Événements d'un <code>LocationListener</code>	162
9.3.6	Remarques	162
9.4	Caméra	163
9.4.1	Présentation	163
9.4.2	Vue <code>SurfaceView</code>	163
9.4.3	Fonctionnement du <code>SurfaceView</code>	163
9.4.4	Événements d'un <code>SurfaceHolder</code>	164
9.4.5	Écouteur <code>surfaceCreated</code>	164
9.4.6	Écouteur <code>surfaceCreated</code> , fin	164
9.4.7	Écouteur <code>surfaceChanged</code>	165
9.4.8	Choix de la prévisualisation	165
9.4.9	Suite de <code>surfaceChanged</code>	165
9.4.10	Orientation de la caméra	166
9.4.11	Orientation de l'écran	166
9.4.12	Associer la caméra et la vue	166
9.4.13	Écouteur <code>onResume</code>	167
9.4.14	Écouteur <code>onPause</code>	167
9.4.15	Écouteur <code>surfaceDestroyed</code>	167
9.4.16	Organisation logicielle	168
9.5	Capteurs d'orientation	168
9.5.1	Présentation	168
9.5.2	Angles d'Euler	169
9.5.3	Matrice de rotation	169
9.5.4	Accès au gestionnaire	169
9.5.5	Accès aux capteurs	170
9.5.6	Abonnement aux mesures	170
9.5.7	Réception des mesures	170
9.5.8	Atténuation des oscillations	171
9.5.9	Orientation avec <code>TYPE_ROTATION_VECTOR</code>	171
9.5.10	Orientation sans <code>TYPE_ROTATION_VECTOR</code>	172
9.5.11	Orientation sans <code>TYPE_ROTATION_VECTOR</code> , fin	172
9.5.12	Orientation avec <code>TYPE_ORIENTATION</code>	173

9.6	Réalité augmentée	173
9.6.1	Objectif	173
9.6.2	Assemblage	173
9.6.3	Transformation des coordonnées	173
9.6.4	Transformation des coordonnées, fin	174
9.6.5	Dessin du POI	174
10	Dessin 2D interactif et Cartes	175
10.1	AsyncTasks	175
10.1.1	Présentation	175
10.1.2	Tâches asynchrones	175
10.1.3	Principe d'utilisation d'une AsyncTask	176
10.1.4	Structure d'une AsyncTask	176
10.1.5	Autres méthodes d'une AsyncTask	176
10.1.6	Paramètres d'une AsyncTask	176
10.1.7	Exemple de paramétrage	177
10.1.8	Paramètres variables	177
10.1.9	Définition d'une AsyncTask	177
10.1.10	Lancement d'une AsyncTask	178
10.1.11	Schéma récapitulatif	178
10.1.12	execute ne retourne rien	178
10.1.13	Récupération du résultat d'un AsyncTask	179
10.1.14	Simplification	179
10.1.15	Fuite de mémoire	180
10.1.16	Recommandations	180
10.1.17	Autres tâches asynchrones	181
10.2	OpenStreetMap	181
10.2.1	Présentation	181
10.2.2	Documentation	181
10.2.3	Pour commencer	181
10.2.4	Layout pour une carte OSM	183
10.2.5	Activité pour une carte OSM	183
10.2.6	Positionnement de la vue	183
10.2.7	Calques	184

10.2.8	Mise à jour de la carte	184
10.2.9	Marqueurs	184
10.2.10	Marqueur personnalisés	185
10.2.11	Réaction à un clic	185
10.2.12	Itinéraires	185
10.2.13	Position GPS	186
10.2.14	Mise à jour en temps réel de la position	186
10.2.15	Positions simulées	186
10.2.16	Clics sur la carte	187
10.2.17	Traitement des clics	187
10.2.18	Autorisations	187



Figure 1: Robot Android

Semaine 1

Environnement de développement

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

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.
- la définition d'une interface d'application

1.1. Introduction

1.1.1. Qu'est-ce qu'Android ?

Android est une sur-couche au dessus d'un système Linux : Voir la figure 2, page 18.

([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 13 (août. 2022) et l'API 33. La version 13 est le numéro pour le grand public, et les versions d'API sont pour les développeurs. Exemples :
 - Android 4.1 JellyBean = API 16,
 - Android 7.0 Nougat = API 24,
 - Android 13 Tiramisu = API 33

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

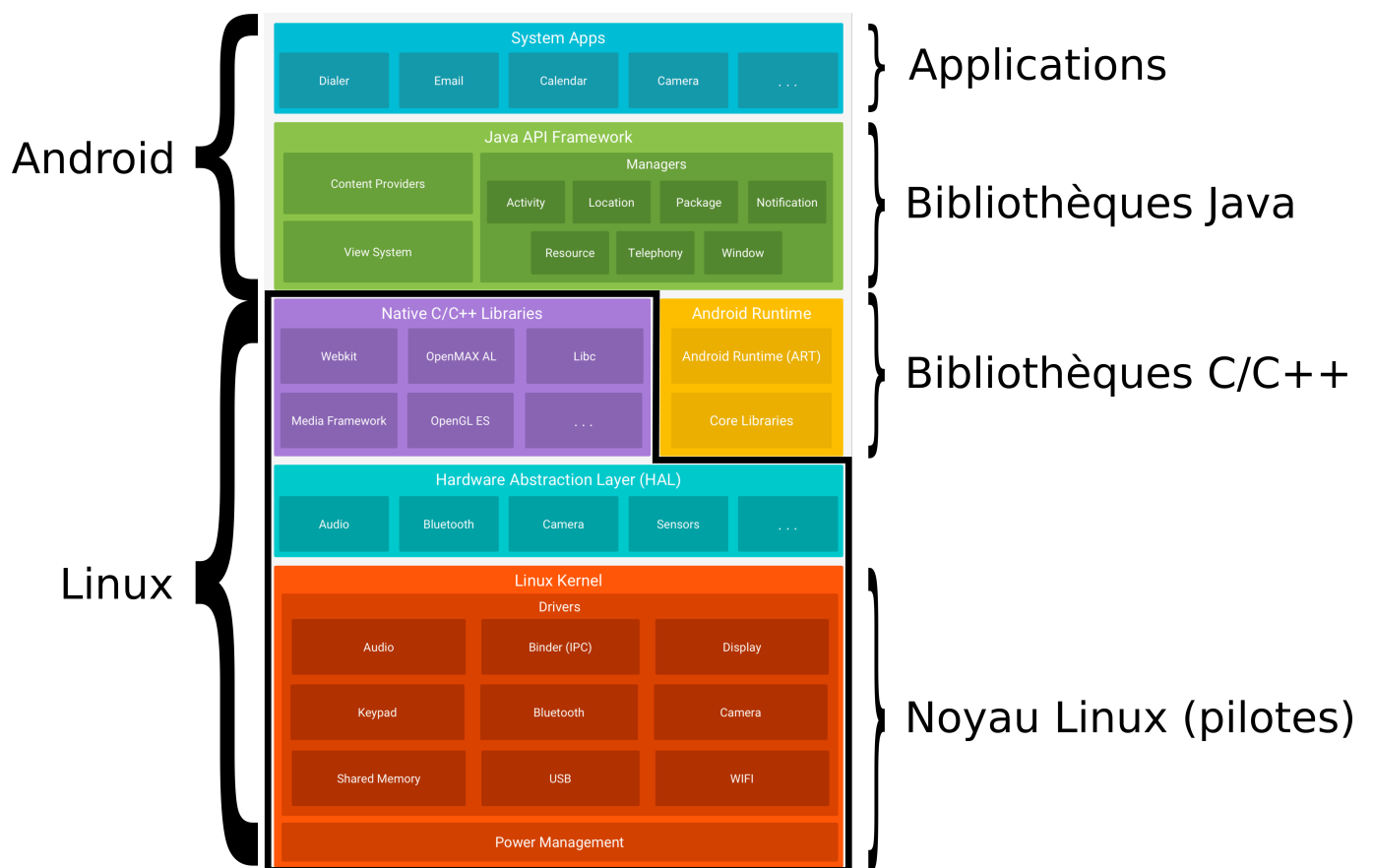


Figure 2: Constituants d'Android

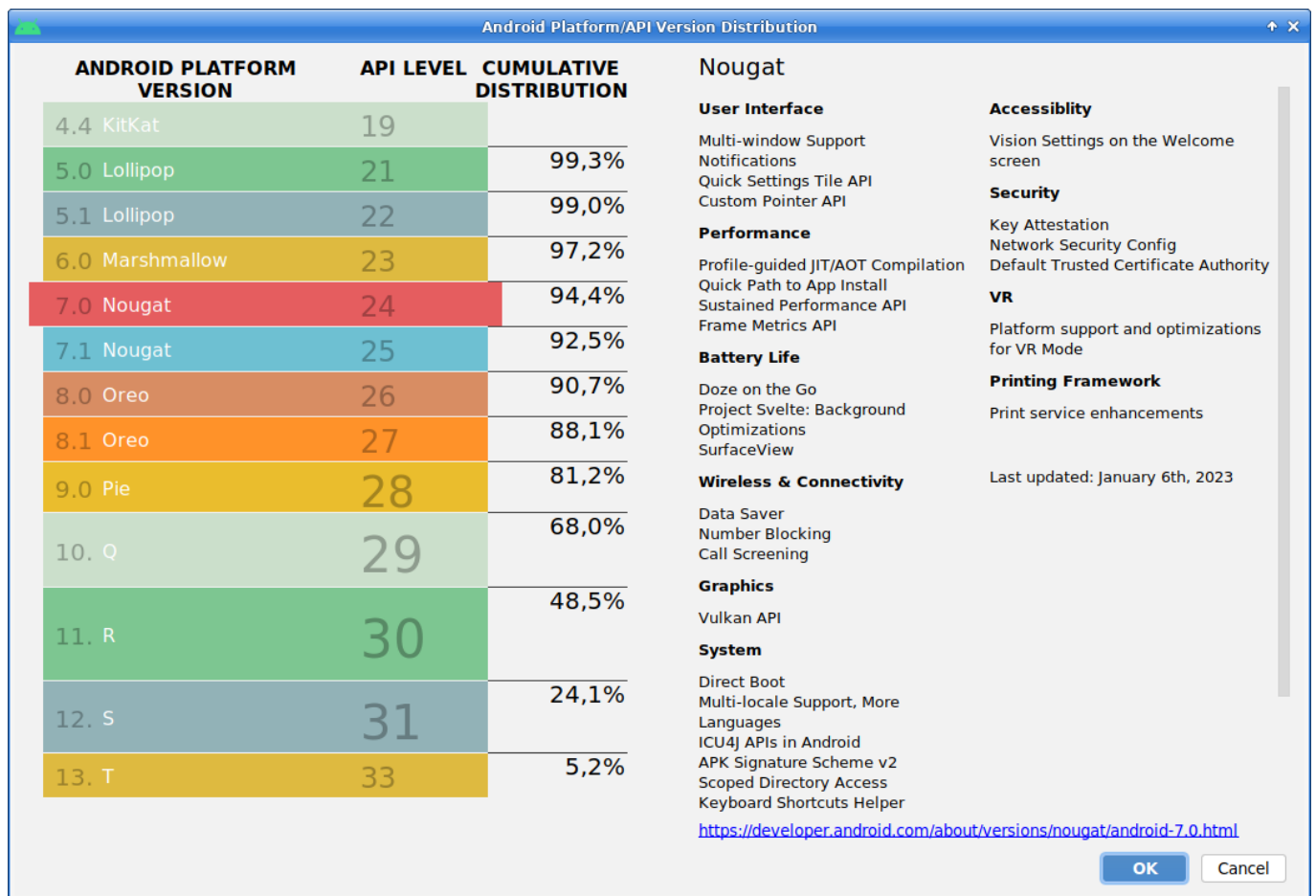


Figure 3: Distribution d'Android

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 2023 : 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...)
 - Exemple : Jetpack Compose dans Android...
 - 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, React, Vue...
- « **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 pour vous 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* » (\neq `.class` Java)
- 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, ça plante à l'exécution (ce n'est pas souhaitable) :

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

En Kotlin, le compilateur refuse le source :

```
fun getNomClient(p: Personne): String {  
    return p.prenom+" "+p.nom  
}  
var p1: Personne? = getPersonne()      // p1 peut être null  
println(getNomClient(p1))               // refus de compiler
```

En Java amélioré avec des annotations :

```
import androidx.annotation.NonNull;  
import androidx.annotation.Nullable;  
  
void getNomClient(@NonNull Personne p) {  
    return p.getPrenom()+" "+p.getNom();  
}  
  
@Nullable Personne p1 = getPersonne();  
System...println(getNomClient(p1));    // refus de compiler
```

En Java, il faut y penser, tandis que 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 d'éviter de 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 bibliothèques de classes et fonctions pour créer des logiciels
- les outils de fabrication des logiciels (compilateur...)
- *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 bibliothèques, 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 23.

1.2.4. Choix des éléments du SDK

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

- Android 13 (API 33)
- ...
- 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).

1.3. Création d'une application

1.3.1. Assistant de création d'application

Android Studio contient un assistant de création d'applications : Voir la figure 5, page 24.

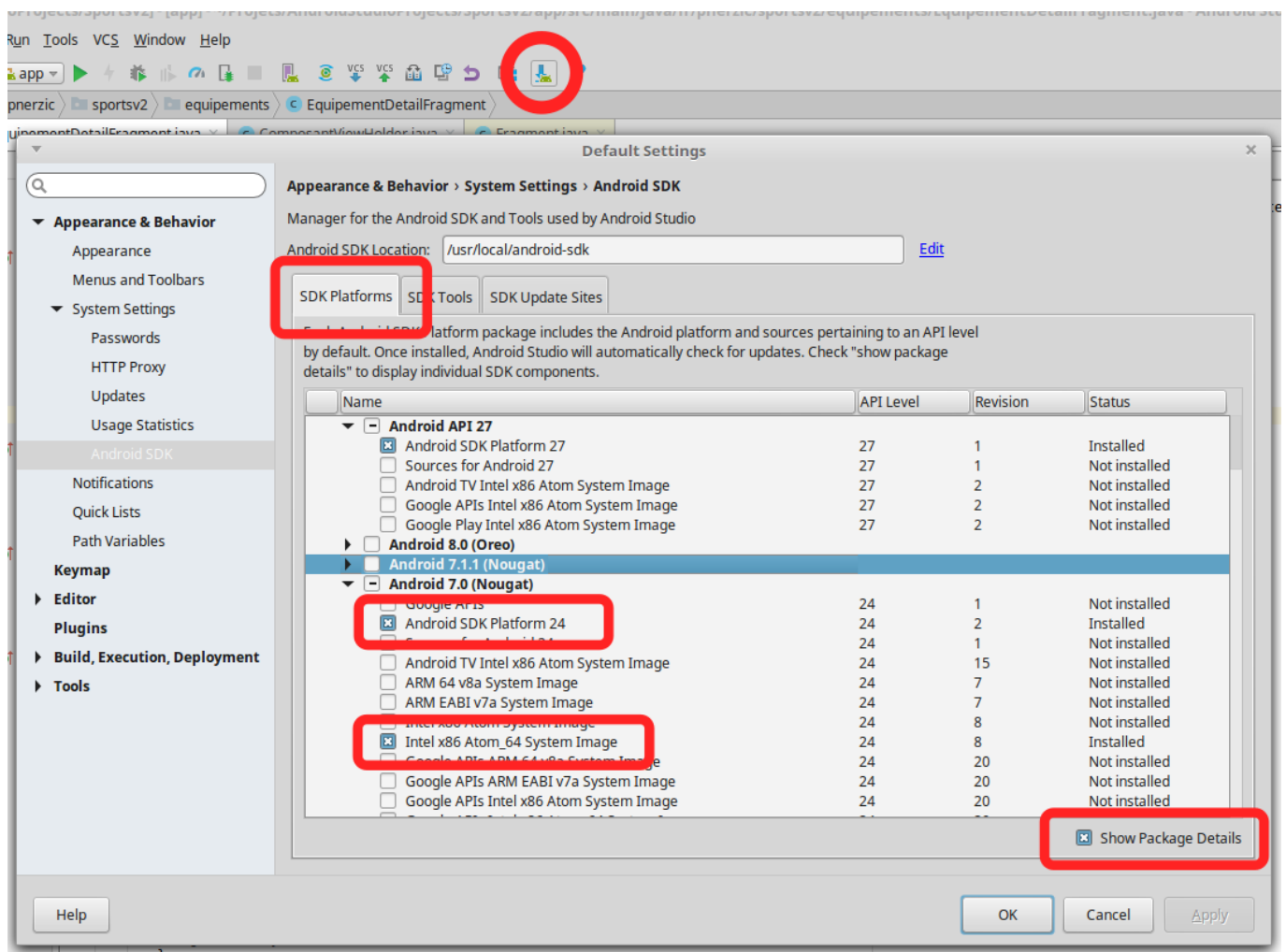


Figure 4: Gestionnaire de paquets Android

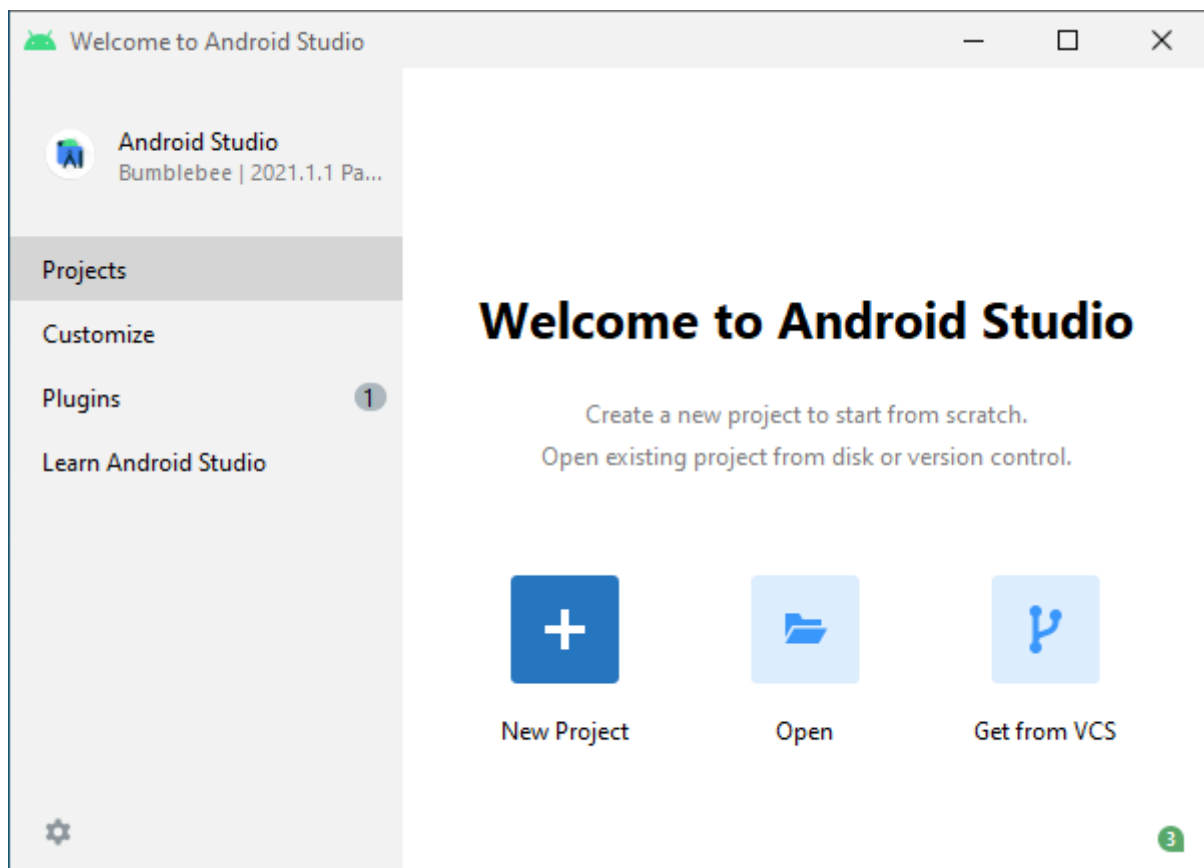


Figure 5: Assistant de création de projet

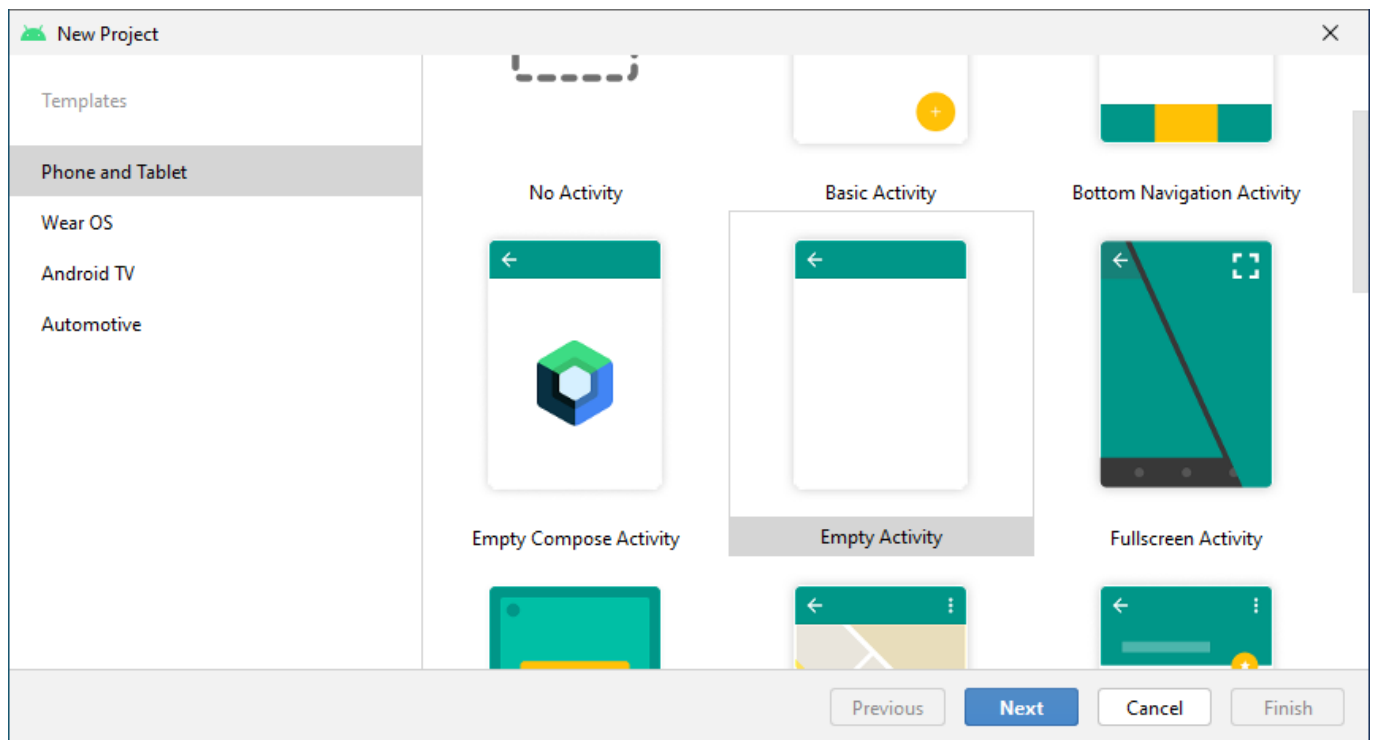


Figure 6: Choix du type d'activité

1.3.2. Modèle d'application

Android Studio propose plusieurs modèles de projet. Voir la figure 6, page 25. En général, on part de celui appelé *Empty Activity*.

1.3.3. Résultat de l'assistant

L'assistant crée de nombreux éléments :

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

NB: ne pas chercher à tout comprendre dès le début.

1.3.4. Fenêtre du projet

Voir la figure 7, page 26.

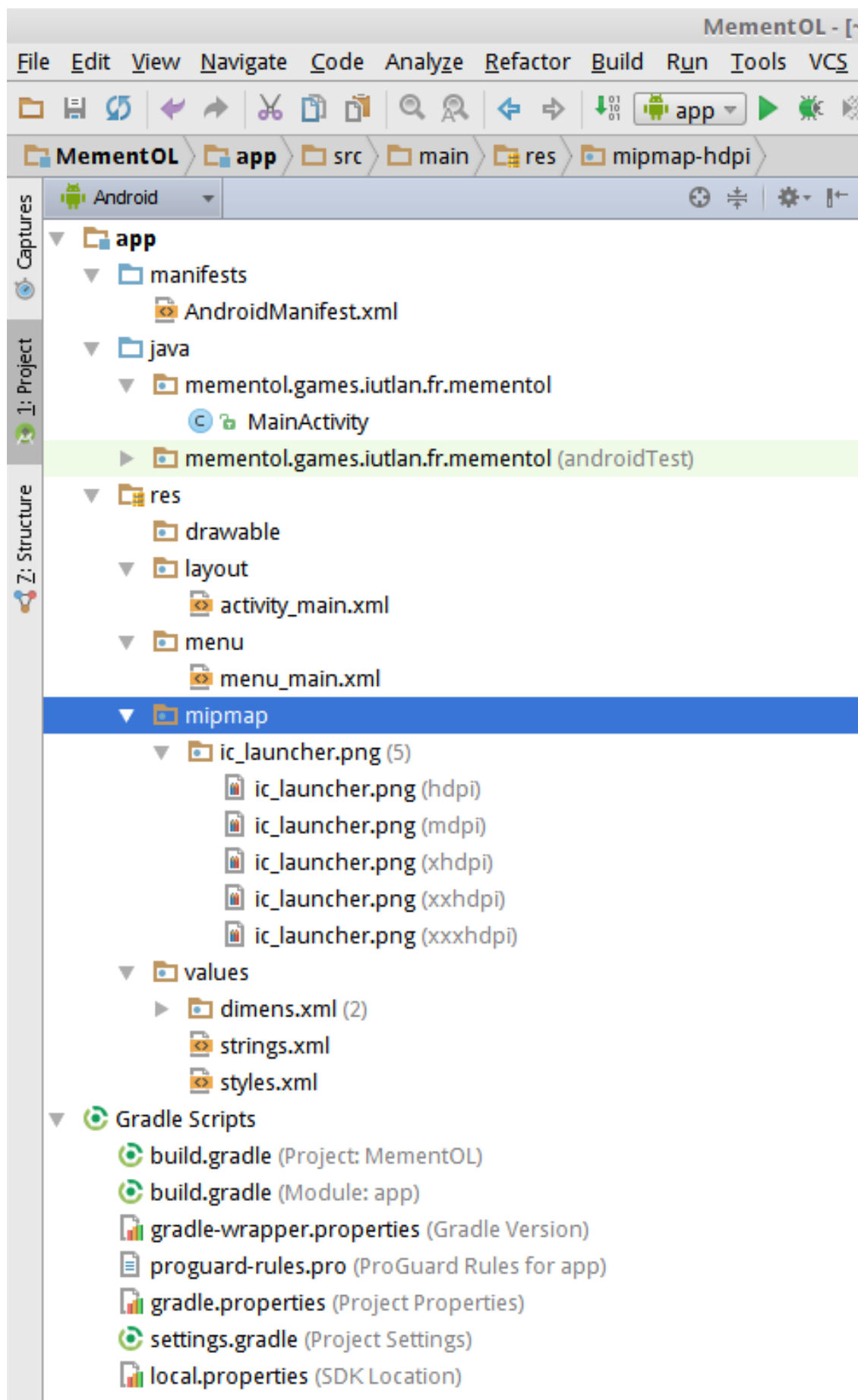


Figure 7: Éléments d'un projet Android

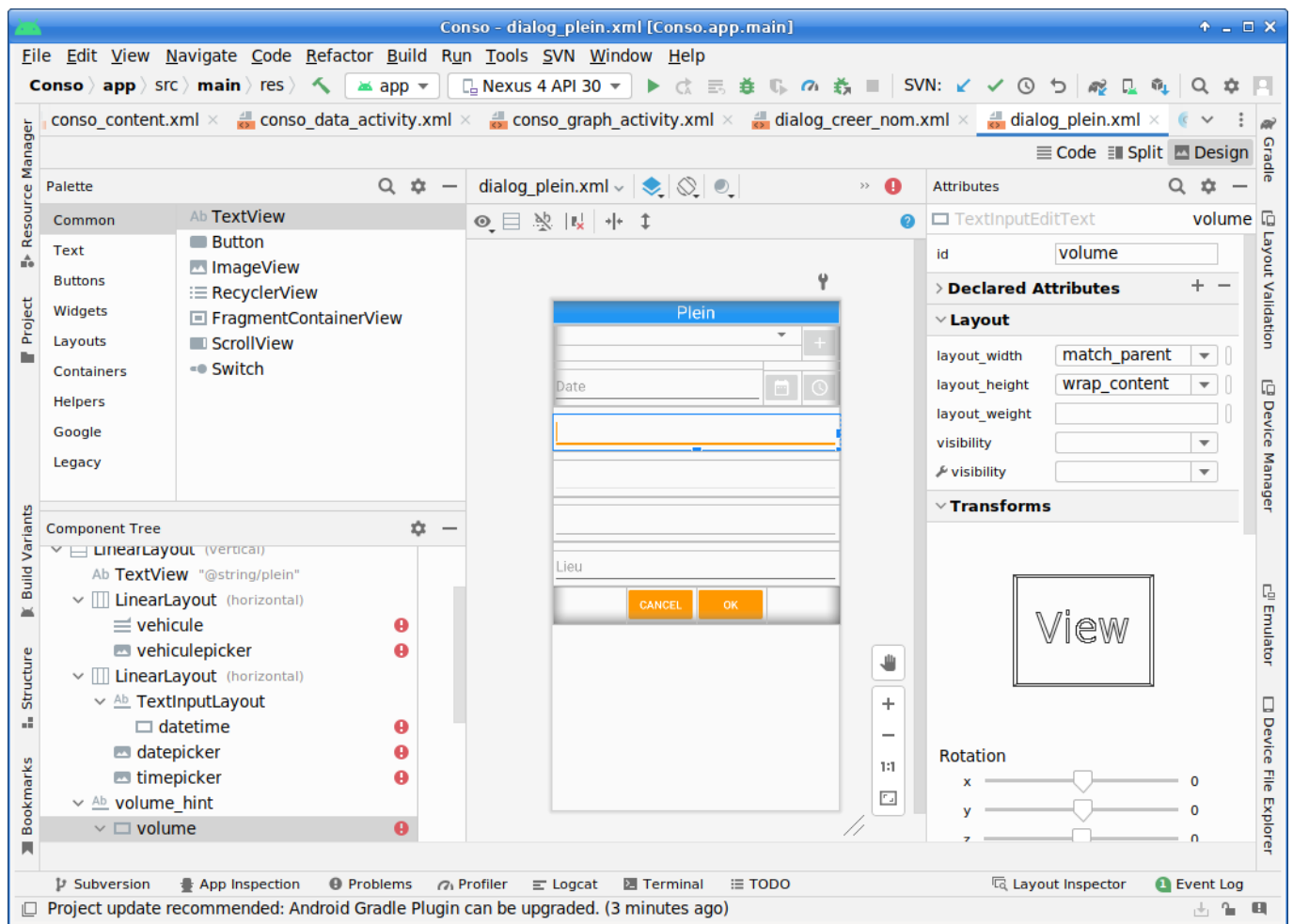


Figure 8: Éditeur graphique

1.3.5. 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.

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

figure 8

1.3.7. Source XML sous-jacent

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

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.

1.3.8. 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*.

1.3.9. 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 bibliothèques utilisées).
- un dossier *app* contenant l'application du projet.
- un script *build.gradle* dans le dossier *app* pour compiler l'application.

1.3.10. 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
```

1.3.11. 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:1.6.0'

    // annotations
    annotationProcessor 'org.projectlombok:lombok:1.18.22'

    // fuites mémoire
    debugImplementation 'com.squareup.leakcanary:leakcanary-android:2.5'
}
```

Les bibliothèques indiquées sont automatiquement téléchargées.

1.4. Exécution de l'application

1.4.1. Simulateur ou smartphone

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*

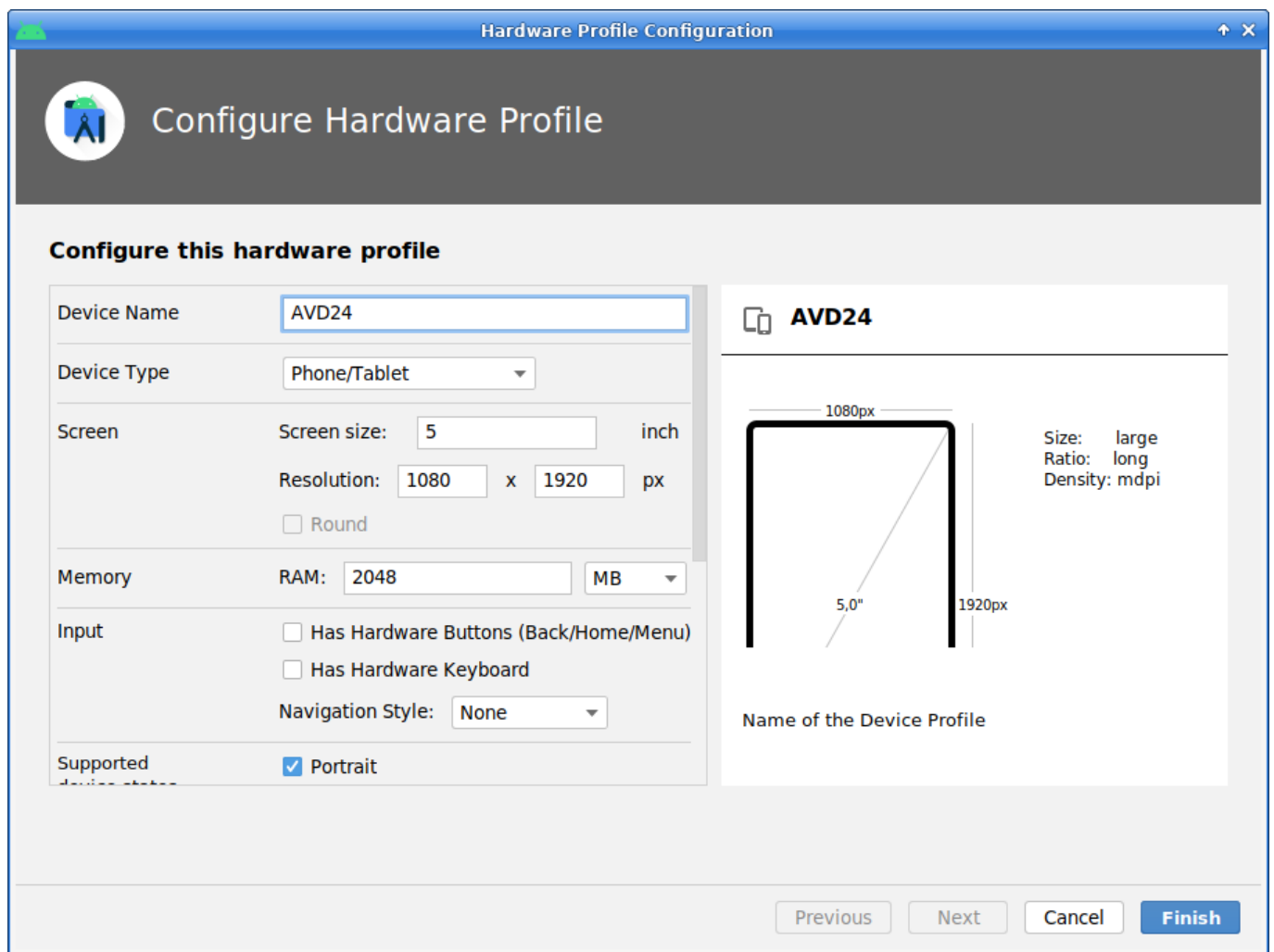
AVD = Android Virtual Device

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 9, page 30.

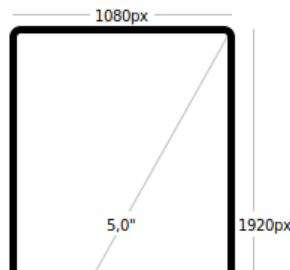


Hardware Profile Configuration

Configure Hardware Profile

Configure this hardware profile

Device Name	AVD24		
Device Type	Phone/Tablet		
Screen	Screen size:	5	inch
	Resolution:	1080	x 1920 px
	<input type="checkbox"/> Round		
Memory	RAM:	2048	MB
Input	<input type="checkbox"/> Has Hardware Buttons (Back/Home/Menu)		
	<input type="checkbox"/> Has Hardware Keyboard		
	Navigation Style:	None	
Supported device states	<input checked="" type="checkbox"/> Portrait		



Size: large
Ratio: long
Density: mdpi

Name of the Device Profile

Previous Next Cancel Finish

Figure 9: Création d'un AVD

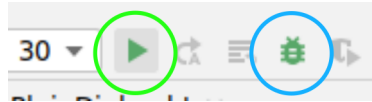


Figure 10: Barre d'outils pour lancer une application

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).

1.4.4. Lancement d'une application

Bouton vert pour exécuter, bleu pour déboguer : figure 10

NB: les icônes, styles et emplacements, varient d'une version d'AndroidStudio à l'autre.

Ces deux boutons installent l'application sur l'AVD ou le smartphone et la démarrent.

1.4.5. Application sur l'AVD

Voir la figure 11, page 32.

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 12, page 33.

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

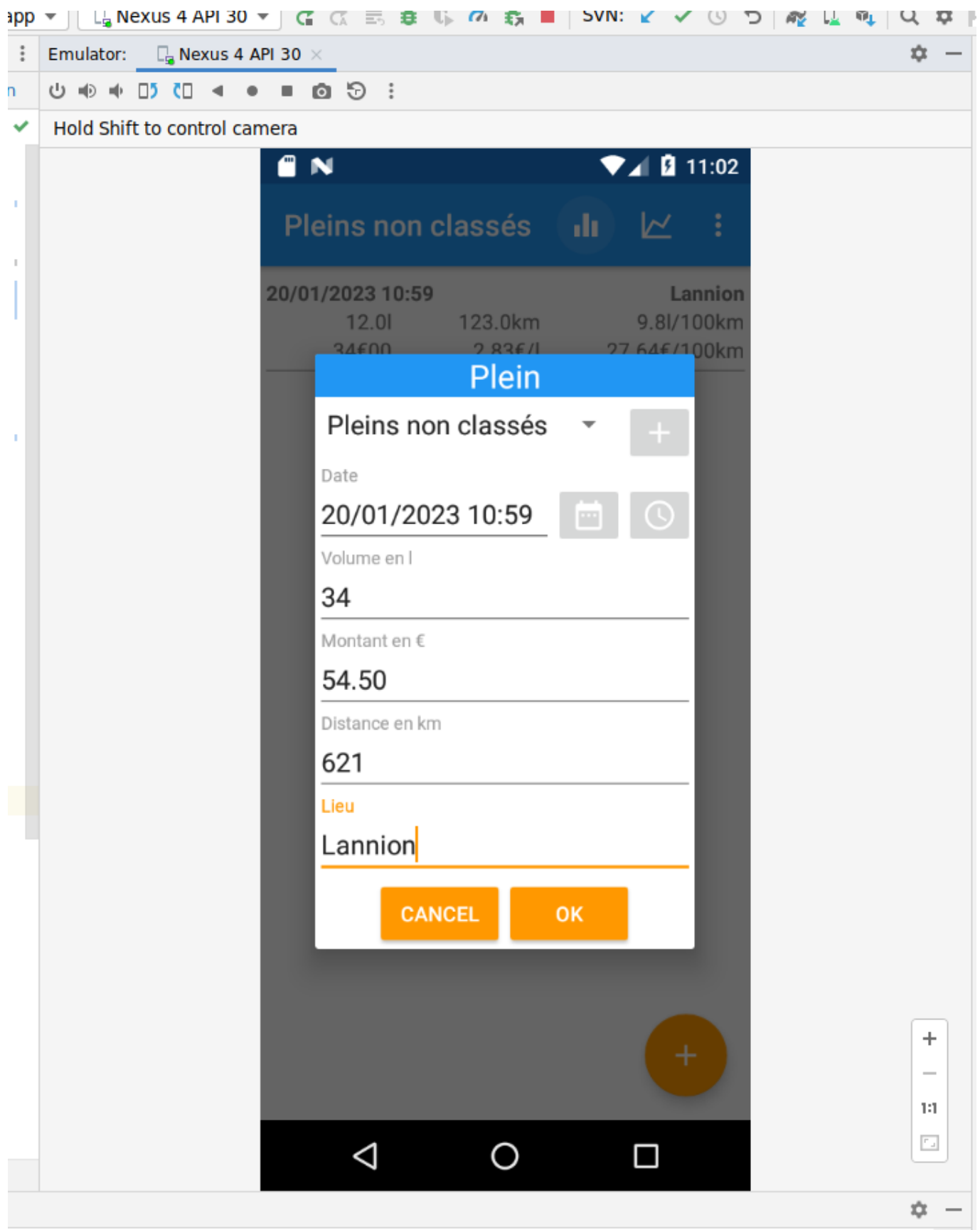


Figure 11: Résultat sur l'AVD

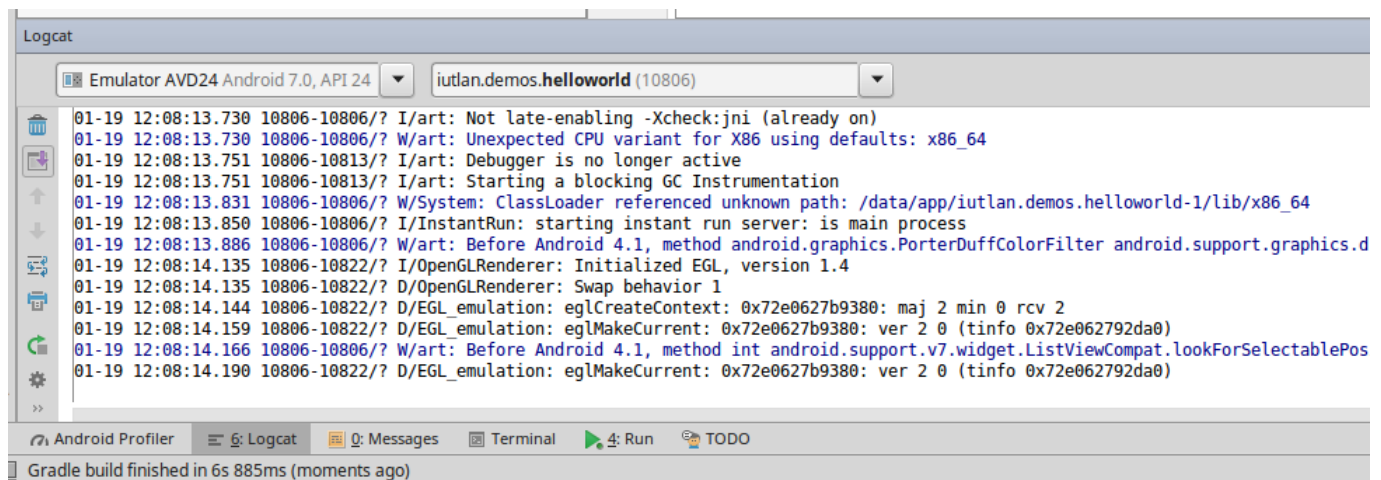


Figure 12: Fenêtre LogCat

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.* :

- 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/smartphone (réel ou virtuel) et votre PC

- Serveur de connexion des tablettes
- Commande de communication

ADB regroupe plusieurs outils :

- FTP : transfert de fichiers,
- SSH : connexion à un shell.

1.5.6. Mode d'emploi de ADB

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

- `adb devices` : liste les appareils connectés
- `adb shell` : connexion à l'appareil

Exemple :

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

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 un vrai smartphone, 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 (authentifiée) contenant les binaires, ressources compressées et autres fichiers de données.

La création est relativement simple avec Studio :

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

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

1.6.2. Signature d'une application

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

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

Ensuite créer une *clé privée* :

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

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

1.6.3. Création du *keystore*

Voir la figure 13, page 36.

1.6.4. Création d'une clé

Voir la figure 14, page 37.

1.6.5. Création du paquet

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

Voir la figure 15, page 38.

1.6.6. Et voilà

C'est fini pour cette semaine, rendez-vous en TD/TP pour la mise en pratique.

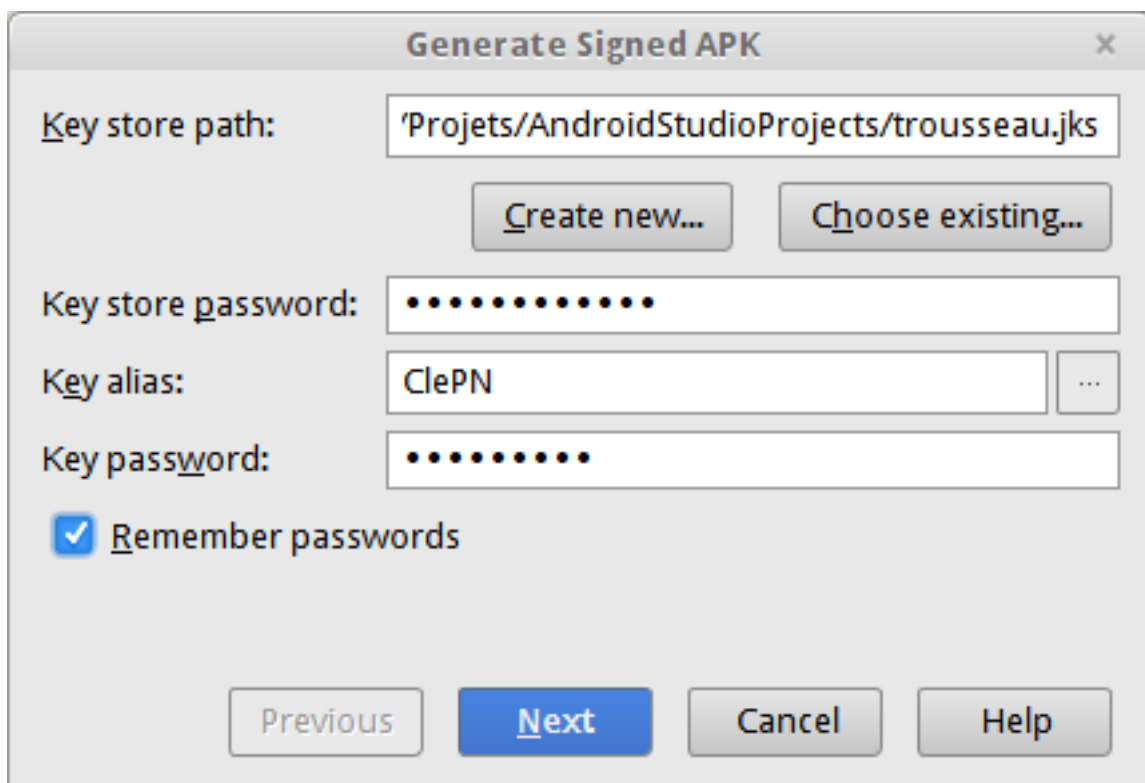


Figure 13: Création d'un trousseau de clés

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.

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 14: Création d'une clé

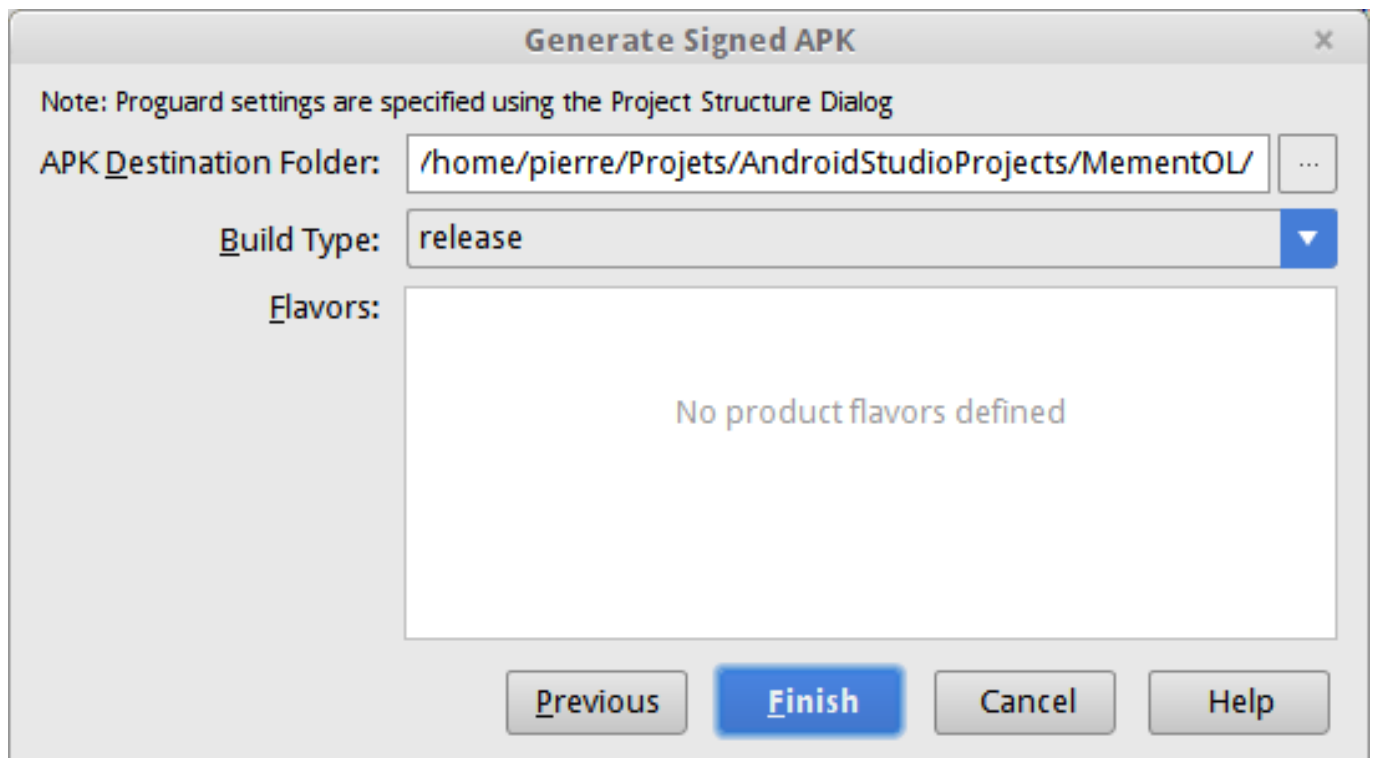


Figure 15: Création du paquet

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* appelé « 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 `R.layout.main` est l'identifiant 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 (classes, bibliothèques) 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 doit simplement prévoir 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, theme, 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 surcharges :

- `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`.

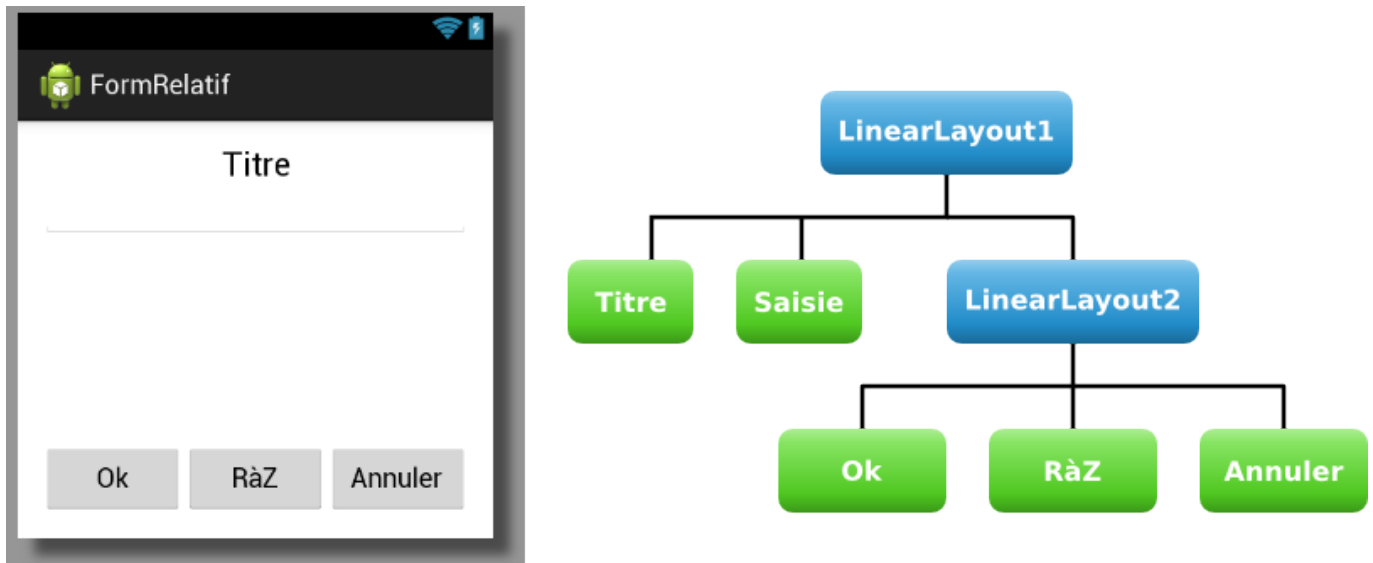


Figure 16: Arbres de vues

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 16

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 : `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 indépendants de l'écran ([explications](#)). 100dp font 100 pixels sur un écran de 160 dpi (160 *dots per inch*) tandis qu'ils font 200 pixels sur un écran 320 dpi. Ça fait la même taille apparente quelque soit la finesse des pixels.

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 17, page 47.

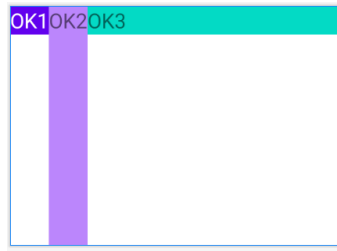


Figure 17: Arbre de vues

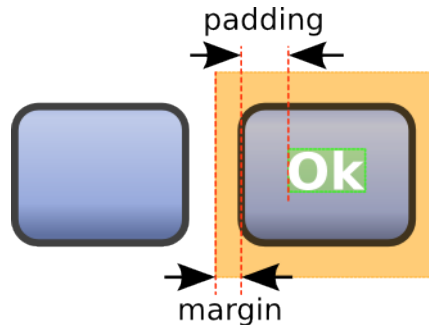


Figure 18: Bords et marges

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 18

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" />
```

C'est très similaire à CSS.

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">
```



Figure 19: Influence des poids sur la largeur

```
<Button android:text="Ok"
    android:layout_width="wrap_content"
    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 19

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 reductibles

```
<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("blablaba");
```

2.4.3. Button

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

Ok

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

- En général, on définit un identifiant pour chaque vue active, ici : `android:id="@+id/btnOk"`
- 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 :

☒ Inscrire newsletter

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

Les **ToggleButton** sont une variante :

☐ online

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

NB: l'esthétique de toutes ces vues change avec les versions d'Android.

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. Présentation

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 héritant de **Activity**.

Les vues d'une activité (boutons, menus, actions) permettent d'aller sur une autre activité. Le bouton *back* ◀ permet de revenir sur une précédente activité. C'est la *navigation entre activités*.

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 (ex: contacts, ...),
- des *récepteurs d'annonces* : pour gérer des messages envoyés d'une application à une autre (ex: notifications, ...).

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 ...>
    <activity android:name=".MainActivity" ... />
    <activity android:name=".ConfigActivity" ... />
    <activity android:name=".DessinActivity" ... />
    ...
  </application>
</manifest>
```

Une activité qui n'est pas déclarée dans le manifeste ne peut pas être lancée (`ActivityNotFoundException`).

3.1.3. Démarrage d'une application

L'une des activités est désignée comme étant « principale », démarrable de l'extérieur, grâce à un sous-élément `<intent-filter>` : 

```
<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. Celui-ci indique l'activité principale, celle qu'il faut lancer quand on clique sur son icône.


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 au premier plan, tandis que `Activ1` se met en sommeil.

`Activ1` reviendra au premier plan quand `Activ2` se finira ou quand l'utilisateur appuiera sur *back*.

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 20, page 57.

3.2.3. Lancement avec ou sans retour possible

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



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

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



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

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

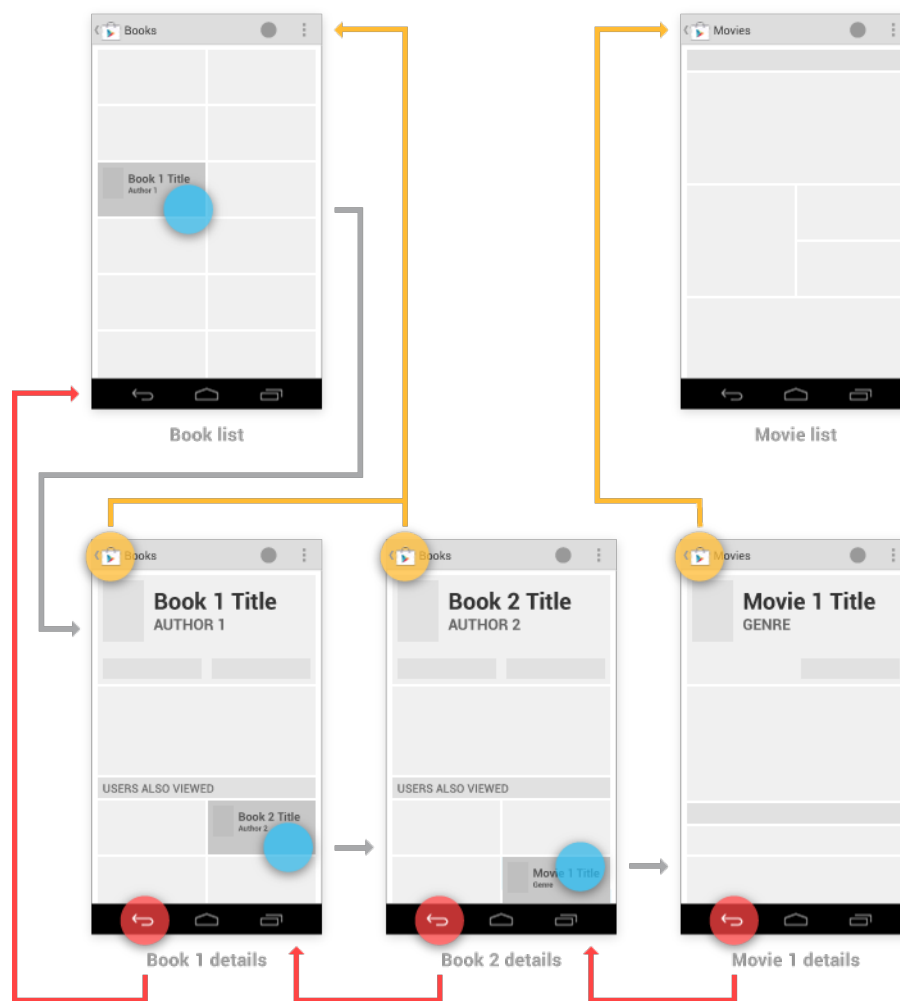


Figure 20: Navigation parmi les activités d'une application

3.2.4. 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 » ◀. Il est équivalent à ceci :

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

3.2.5. Lancement avec attente de résultat

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

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

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

Consulter [cette page](#).

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

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

3.2.6. Méthode `onActivityResult`

Cette méthode est appelée quand on revient dans l'activité initiale :

`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. Lancement avec attente, version améliorée

Les dernières versions de l'API proposent mieux pour récupérer un résultat d'une activité lancée. On n'a plus le couple `startActivityForResult` et `onActivityResult`, mais ceci :

- D'abord une variable du type `ActivityResultLauncher<Intent>` dans l'activité. Il en faut une par type d'action à faire au retour de l'activité lancée. Elle représente un lanceur pour la seconde activité.
- L'initialisation de ces lanceurs dans la méthode `onCreate`,
- Un écouteur qui sera appelé au retour de l'activité lancée,
- Le lancement de l'activité dans ce cadre.

3.2.8. Lanceur d'activité

Il faut commencer par définir une variable membre. Il en faut une par activité qui sera lancée ultérieurement : 

```
private ActivityResultLauncher<Intent> activ2Launcher;
```

Par exemple, cette variable `activ2Launcher` permettra de lancer `Activ2`. Elle doit être initialisée dans `onCreate()` : 

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    activ2Launcher = registerForActivityResult(
        new ActivityResultContracts.StartActivityForResult(),
        this::onActiv2Ended);
}
```

L'écriture `this::nomMeth` est appelée *référence de méthode*, voir page 69. C'est le nom de la méthode à appeler au retour.

3.2.9. Écouteur de retour d'activité

Il faut maintenant définir la méthode en question : 

```
private void onActiv2Ended(ActivityResult result) {
    if (result.getResultCode() == RESULT_CANCELED) return;
    // TODO actions à faire au retour d'Activ2
}
```

Elle ressemble à `onActivityResult` mais ses paramètres sont dans un objet `ActivityResult` et on utilise ses *getters*.

Pour finir, le lancement de `Activ2` : 

```
Intent intent = new Intent(this, Activ2.class);
activ2Launcher.launch(intent);
```

Au retour de `Activ2`, la méthode `onActiv2Ended` sera appelée automatiquement.

3.2.10. 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.

3.2.11. 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";
```

3.2.12. 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.

3.2.13. 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>
```

3.2.14. 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 :

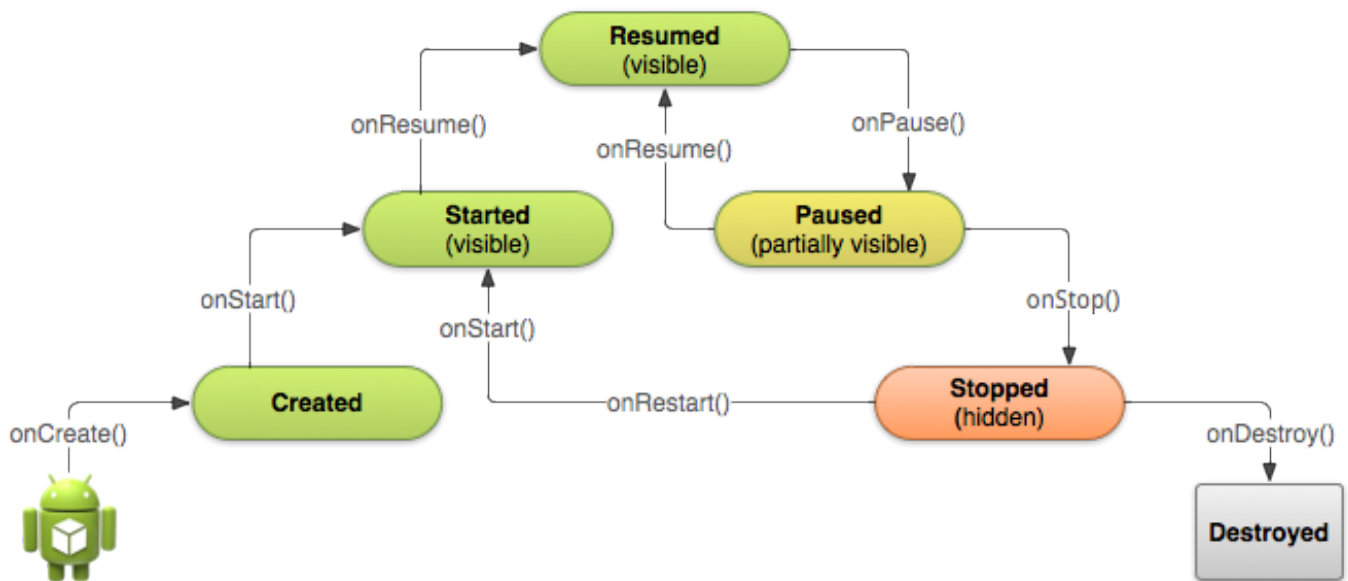


Figure 21: Cycle de vie

- 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 21

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/application est passée devant, il faut enregistrer les informations au cas où l'utilisateur ne relance pas l'application.

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 `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. C'est néanmoins surveillé par Android Studio.

Pour éviter les problèmes de typage et de vues absentes d'un layout, il existe un dispositif appelé **ViewBindings**. Ce sont des classes qui sont générées automatiquement à partir de chaque layout et dont les variables membres sont les différentes vues.

Par exemple, soit un layout appelé `activity_main.xml` :

```
<LinearLayout ...>

    <TextView android:id="@+id/titre" .../>

    <Button android:id="@+id/btnOk" .../>

</LinearLayout>
```

Cela fait générer une classe appelée `ActivityMainBinding.java` et contenant à peu près ceci :

```
public final class ActivityMainBinding implements ViewBinding
{
    private final LinearLayout rootView; // voir getRoot()
    public final Button btnOk;
    public final TextView titre;
```

Chaque vue du layout xml possédant un identifiant est reliée à une variable membre publique dans cette classe, et la vue racine est accessible par `getRoot()`.

Une méthode statique `inflate` instancie les différentes vues et la vue racine peut être fournie à `setContentView`.

3.4.2. Mode d'emploi des ViewBindings

Dans une activité, faire ceci :

```
public final class MainActivity extends Activity
{
    private ActivityMainBinding ui; // ui = interface uti.

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ui = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(ui.getRoot());
```

```
// exemple d'emploi
ui.titre.setText("super cool !");
}
```

3.4.3. Génération des ViewBindings

Il faut rajouter ceci dans `app/build.gradle` :

```
plugins {
    id 'com.android.application'
}
android {
    compileSdkVersion ...
    buildToolsVersion "..."
    defaultConfig {
        ...
    }
    buildFeatures {
        viewBinding true // génération des ViewBindings
    }
}
```

3.4.4. 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 = ui.titre;
tvTitre.setLines(2);
tvTitre.setText(R.string.debut);
```

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

3.4.5. 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/btnValider"
    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) {
    ...
}
```

3.4.6. 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 = ui.btnValider;
btn.setOnClickListener(ecouteur);
```

3.4.7. 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 = ui.btnValider;
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.

Il est intéressant de transformer cet écouteur en *expression lambda*. C'est une écriture plus compacte qu'on retrouve également en JavaScript, et très largement employée en Kotlin.


```
Button btn = ui.btnValider;  
btn.setOnClickListener((View btn) -> {  
    // faire quelque chose  
});
```

Une *lambda* est une fonction sans nom écrite ainsi :

```
(paramètres avec ou sans types) -> expression  
(paramètres avec ou sans types) -> { corps }
```


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

3.4.8. 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 = ui.btnValider;  
    btn.setOnClickListener(new EcBtnValider());  
}
```


3.4.9. 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 = ui.btnValider;  
        btn.setOnClickListener(this);  
    }  
    public void onClick(View btn) {  
        // faire quelque chose  
    }  
}
```

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

3.4.10. 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;  
    }  
}
```

Depuis peu, les identifiants de ressources ne sont plus des constantes et ne peuvent plus être employés dans des switch. On doit faire ainsi : 

```
public void onClick(View v) {  
    int id = v.getId();  
    if (id == R.id.btn_valider) {  
        ...  
    } else if (id == R.id.btn_effacer) {  
        ...  
    }  
}
```

3.4.11. Écouteur référence de méthode

Il y a une dernière façon très pratique d'associer un écouteur, avec une *référence de méthode*, c'est à dire son nom précédé de l'objet qui la définit : 

```
public void onCreate(...) {  
    ...  
    Button btn = ui.btnValider;  
    btn.setOnClickListener(this::onBtnClicked);  
}  
  
private void onBtnClicked(View btn) {  
    // faire quelque chose  
}
```

La syntaxe `this::nom_methode` est une simplification de l'expression *lambda* `(params) -> nom_methode(params)`

3.4.12. 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.

3.4.13. 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*.

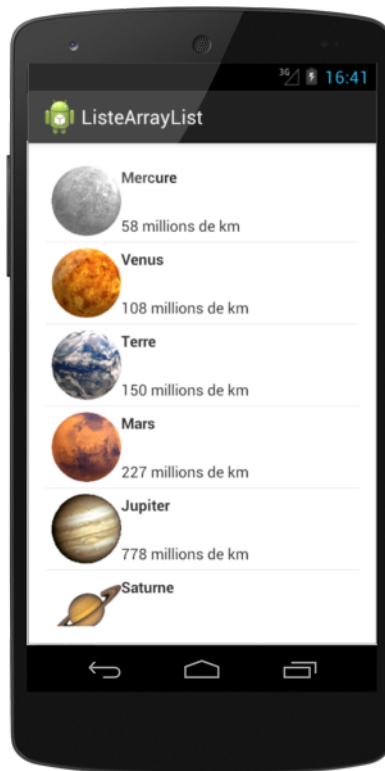


Figure 22: Liste d'items

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 22

4.1. Présentation

4.1.1. Principe général

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

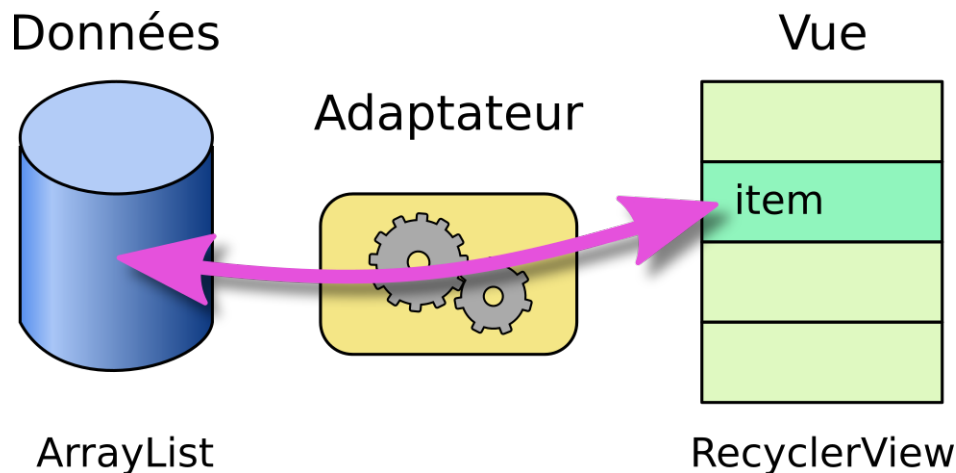


Figure 23: Vue, adaptateur et données

- 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 `RecyclerView`. C'est une vue spécialisée dans l'affichage de listes quelconques.

Consulter [ces explications](#) qui sont très claires et très complètes, mais qui n'utilisent pas les `ViewBindings`.

Il y a aussi [la documentation](#) Google, assez compliquée, sur les `RecyclerView`, et [celle là](#) sur les adaptateurs.

Anciennement, on utilisait des `ListView`, mais ils sont délaissés car trop polyvalents.

4.1.2. Schéma global

Modèle MVC : le contrôleur entre les données et la vue s'appelle un *adaptateur*.

figure 23

4.1.3. Une classe pour représenter les items

Pour commencer, il faut représenter les données :



```
public class Planete {
    public String nom;           // nom de la planète
    public int distance;        // distance au soleil en Gm

    Planete(String nom, int distance) {
        this.nom = nom;
        this.distance = distance;
    }
}
```

Lui rajouter tous les accesseurs (*getters*) et modificateurs (*setters*) pour en faire un *JavaBean* : objet Java simple (POJO) composé de variables membres privées initialisées par le constructeur, et d'accesseurs.

4.1.4. Données initiales

Deux solutions pour initialiser la liste avec des valeurs prédéfinies :

- Un tableau dans les ressources, voir page 74.
- Un tableau constant Java comme ceci :



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

`final` signifie constant, `initdata` ne pourra pas être réaffecté (par contre, ses cases peuvent être réaffectées).

4.1.5. Copie dans un ArrayList

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



```
private List<Planete> liste;  
  
void onCreate(...)  
{  
    ...  
  
    // copie du tableau initdata dans le ArrayList  
    liste = new ArrayList<>(Arrays.asList(initdata));  
}
```

NB: `Arrays.asList` crée une liste non modifiable, c'est pour ça qu'on la recopie dans un `ArrayList`.

4.1.6. Rappels sur le container List<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.

```
List<TYPE> liste = new ArrayList<>();
```

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

La variable est du type `List` (superclasse abstraite) et affectée avec un `ArrayList`. La raison est qu'il faut de préférence toujours employer le type le plus général qui possède les méthodes voulues. Mais quand c'est une classe abstraite (une interface), on l'instancie avec une sous-classe non-abstraite.

Par exemple un `List` peut être instancié avec un `ArrayList` ou un `LinkedList`. On choisit en fonction des performances voulues : un `ArrayList` est très rapide en accès direct, mais très lent en insertion. C'est l'inverse pour un `LinkedList`.

Quelques méthodes utiles de la classe abstraite `List`, héritées par `ArrayList` :

- `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>
```

Intérêt : traduire les noms des planètes dans d'autres langues en créant des variantes, ex: `res/values-en/arrays.xml`

Ensuite, on récupère ces ressources 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
liste = new ArrayList<>();
for (int i=0; i<noms.length; ++i) {
    liste.add(new Planete(noms[i], distances[i]));
}
```

C'est plus complexe, mais préférable à la solution du tableau pré-initialisé, pour bien séparer programme et données.

4.1.8. Remarques

Cette semaine, les données sont représentées dans un `ArrayList` volatile : quand on ferme l'activité, les données sont perdues. Pour faire un peu mieux que cela, il faut définir une classe `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.

Cependant, les données ne sont encore pas permanentes. Elles sont perdues quand on quitte l'application.

En semaine 6, nous verrons comment utiliser une base de données Realm locale ou distante, au lieu de ce tableau dynamique, ce qui résout le problème de manière élégante et rend les données persistantes d'une exécution à l'autre.

4.2. Affichage de la liste

4.2.1. Activité

L'affichage de la liste est fait par un `RecyclerView`. C'est une vue qui intègre un défilement automatique et qui veille à économiser la mémoire pour l'affichage.

Voici le layout le plus simple qui remplit tout l'écran, mais on peut rajouter d'autres vues : boutons... :



```
<androidx.recyclerview.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/recycler"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Notez la présence du package de cette vue dans la balise. Elle fait partie de l'ensemble `androidx` qui sera expliqué au cours n°5.

4.2.2. Mise en place du layout d'activité

En utilisant un `ViewBinding` :



```
private ArrayList<Planete> liste;
private ActivityMainBinding ui;

@Override protected void onCreate(Bundle savedInstanceState)
{
    // mettre en place le layout contenant le RecyclerView
    super.onCreate(savedInstanceState);
    ui = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(ui.getRoot());

    // initialisation de la liste avec les ressources
    liste = new ArrayList<>();
    ...
}
```

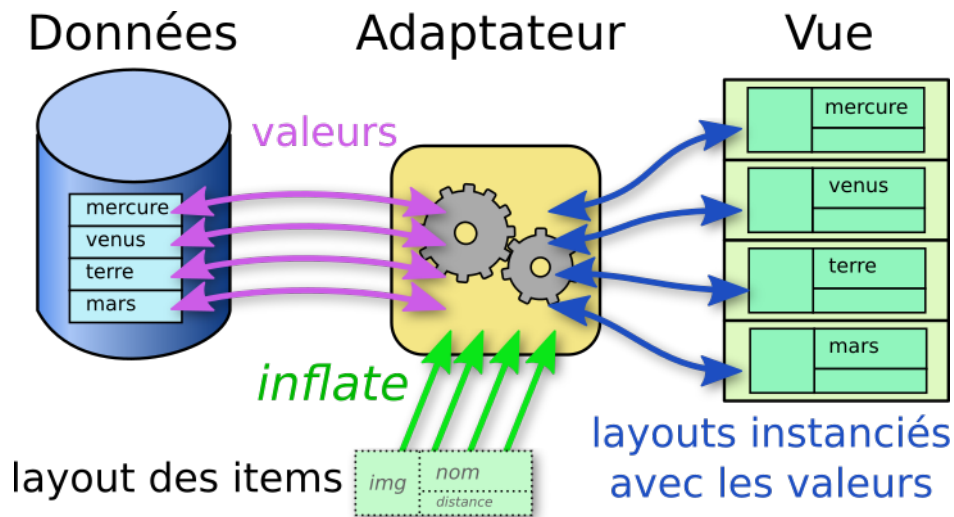


Figure 24: Adaptateur entre les données et la vue

4.3. Adaptateurs et ViewHolders

4.3.1. Relations entre la vue et les données

Un `RecyclerView` affiche les items à l'aide d'un *adaptateur* :

figure 24

4.3.2. Concepts

La vue ne sert qu'à afficher les éléments de la liste. En réalité, seuls quelques éléments seront visibles en même temps. Cela dépend de la hauteur de la liste et la hauteur des éléments.

Le principe du `RecyclerView` est de ne gérer que les éléments visibles. Ceux qui ne sont pas visibles ne sont pas mémorisés. Mais lorsqu'on fait défiler la liste ainsi qu'au début, de nouveaux éléments doivent être rendus visibles.

Le `RecyclerView` demande alors à l'adaptateur de lui instancier (*inflate*) les vues pour afficher les éléments.

Le nom « `RecyclerView` » vient de l'astuce : les vues qui deviennent invisibles à cause du défilement vertical sont recyclées et renvoyées de l'autre côté mais en changeant seulement le contenu à afficher.

4.3.3. Recyclage des vues

Une vue qui devient invisible d'un côté, à cause du scrolling, est renvoyée de l'autre côté, comme sur un tapis roulant, en modifiant seulement son contenu :

Voir la figure 25, page 77.

Il suffit de remplacer « Mercure » par « Uranus » et de mettre la vue en bas.

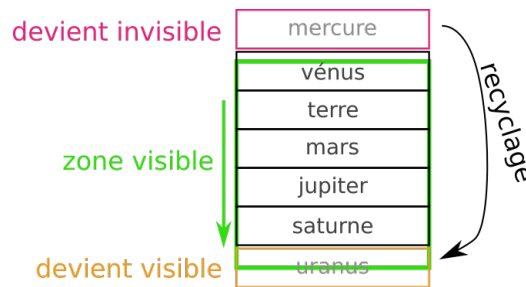


Figure 25: Recyclage des vues

4.3.4. ViewHolders

Pour permettre ce recyclage, il faut que les vues associées à chaque élément puissent être soit recrées, soit réaffectées. On les appelle des *ViewHolders*, parce que ce sont des mini-containers qui regroupent des vues de base (nom de la planète, etc.)

Un *ViewHolder* est instancié pour chaque élément visible de la liste d'items, ex: une planète \longleftrightarrow un *ViewHolder*.

Le *ViewHolder* est associé à un layout géré par un *ViewBinding* pour afficher les informations de l'élément concerné. Pour cela, le *ViewHolder* possède des méthodes pour placer les informations dans ses différentes vues.

4.3.5. Exemple de ViewHolder

D'abord, il faut un layout d'item, `res/layout/planete.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout ...>
    <TextView android:id="@+id/nom" .../>
    <TextView android:id="@+id/distance" .../>
</RelativeLayout>
```

D'autres informations peuvent facilement être rajoutées : images...

Ce même layout sera instancié pour chaque planète visible dans le `RecyclerView`. Les `TextView` seront affectés selon la planète associée.

On va utiliser son *ViewBinding*, c'est à dire la classe `PlaneteBinding`, pour accéder facilement aux `TextView`.

Voici la classe `PlaneteViewHolder` :

```
public class PlaneteViewHolder extends RecyclerView.ViewHolder
{
    private final PlaneteBinding ui;

    public PlaneteViewHolder(PlaneteBinding ui) {
        super(ui.getRoot());
        this.ui = ui;
    }
}
```

```
public void setPlanete(Planete planete) {  
    ui.nom.setText(planete.getNom());  
    ui.distance.setText(  
        Integer.toString(planete.getDistance()));  
}  
}
```

La classe `PlaneteViewHolder` mémorise un `PlaneteBinding`, c'est à dire l'ensemble des vues représentant une planète à l'écran, provenant de `res/layout/planete.xml`. Ce *ViewBinding* sera passé en paramètre par l'adaptateur¹ et mémorisé dans le *ViewHolder*.

La méthode `setPlanete` met à jour ces vues à partir de la donnée passée en paramètre. Cette méthode est appelée par l'adaptateur lors du recyclage.

D'autres méthodes seront ajoutées pour gérer les clics sur les éléments.

4.3.6. Rôle d'un adaptateur

L'adaptateur répond à la question que pose la vue : « *que dois-je afficher à tel endroit dans la liste ?* ». Il va chercher les données et instancie ou recycle un *ViewHolder* avec les valeurs.

L'adaptateur est une classe qui :

- stocke et gère les données : liste, connexion à une base de donnée, etc.
- crée et remplit les vues d'affichage des items à la demande du `RecyclerView`.

On retrouve donc ces méthodes dans sa définition.

4.3.7. Définition d'un adaptateur

Il faut surcharger la classe `RecyclerView.Adapter` qui est une classe générique. Il faut lui indiquer la classe des *ViewHolder*.

Par exemple, `PlaneteAdapter` :

```
public class PlaneteAdapter  
    extends RecyclerView.Adapter<PlaneteViewHolder>  
{  
    ... constructeur ...  
    ... surcharge des méthodes nécessaires...  
}
```

Cette classe va gérer l'affichage des éléments individuels et aussi gérer la liste dans son ensemble. Pour cela, on définit un constructeur et on doit surcharger trois méthodes.

4.3.8. Constructeur d'un adaptateur

La classe `RecyclerView.Adapter` ne contient aucune structure de donnée. C'est à nous de gérer cela :


¹La création du *ViewBinding* est faite en amont, par l'adaptateur. On ne peut pas faire autrement car le constructeur de la superclasse *ViewHolder* demande une interface déjà créée.

```
public class PlaneteAdapter
    extends RecyclerView.Adapter<PlaneteViewHolder>
{
    private final List<Planete> liste;


    PlaneteAdapter(List<Planete> liste) {
        this.liste = liste;
    }
    ...
}
```

La liste est stockée dans l'adaptateur. NB: c'est un partage de référence, il n'y a qu'une seule allocation en mémoire.

4.3.9. Méthodes à ajouter

Ensuite, pour communiquer avec le `RecyclerView`, il faut surcharger (redéfinir) trois méthodes. Pour commencer, celle qui retourne le nombre d'éléments : 

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

Ensuite, surcharger la méthode qui crée les `ViewHolder` : 

```
@Override public PlaneteViewHolder
onCreateViewHolder(ViewGroup parent, int viewType)
{
    PlaneteBinding binding =
        PlaneteBinding.inflate(
            LayoutInflater.from(parent.getContext()),
            parent, false);
    return new PlaneteViewHolder(binding);
}
```

Elle est appelée au début de l'affichage de la liste, pour initialiser ce qu'on voit à l'écran.
inflate = transformer un fichier XML en vues Java.

Enfin, surcharger la méthode qui recycle les `ViewHolder` : 

```
@Override public void
onBindViewHolder(PlaneteViewHolder holder, int position)
{
    Planete planete = liste.get(position);
    holder.setPlanete(planete);
}
```

Cette méthode est appelée pour remplir un `ViewHolder` avec l'un des éléments de la liste, celui qui est désigné par `position` (numéro dans la liste à l'écran). C'est très facile avec le *setter*.

D'autres méthodes seront ajoutées pour gérer les clics sur les éléments.

4.4. Configuration de l'affichage

4.4.1. Optimisation du défilement

On va s'intéresser à la mise en page des *ViewHolders* : en liste, en tableau, en blocs empilés... Il suffit seulement de configurer le `RecyclerView` ; c'est lui qui s'occupe de l'affichage.

D'abord, dans `MainActivity`, il est important d'indiquer au `RecyclerView` si les *ViewHolder* ont tous la même taille ou pas :

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

    // dimensions constantes
    ui.recycler.setHasFixedSize(true);
```

Mettre `false` si les tailles varient d'un élément à l'autre.

4.4.2. LayoutManager

Ensuite, et c'est indispensable, le `RecyclerView` doit savoir comment organiser les éléments : en liste, en tableau, en grille...

Cela se fait avec un *LayoutManager* :

```
...

// layout manager
RecyclerView.LayoutManager lm =
    new LinearLayoutManager(this);
ui.recycler.setLayoutManager(lm);
```

Sans ces lignes, le `RecyclerView` n'est pas affiché du tout.

Il existe plusieurs *LayoutManager* qui vont être présentés ci-après.

4.4.3. LayoutManager dans le layout.xml


Il est possible de spécifier le *LayoutManager* directement dans le fichier XML du layout :

```
<androidx.recyclerview.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/recycler"
    android:layout_width="match_parent"
```



```
android:layout_height="match_parent"
xmlns:app="http://schemas.android.com/apk/res-auto"
app:layoutManager=
    "androidx.recyclerview.widget.LinearLayoutManager" />
```


Mais on ne peut pas le configurer aussi bien que par programmation.

Un `LinearLayoutManager` organise les éléments en liste, en leur donnant tous la même taille. On peut le configurer pour afficher la liste horizontalement, mais il faut alors prévoir le layout des items en conséquence : 


```
// layout manager liste
LinearLayoutManager lm =
    new LinearLayoutManager(this,
        RecyclerView.HORIZONTAL,    // direction
        false);                    // sens
ui.recycler.setLayoutManager(lm);
```

Le 3^e paramètre est un booléen qui indique dans quel sens se fait le défilement, vers la droite ou vers la gauche.

4.4.4. Disposition en tableau

Au lieu d'un `LinearLayoutManager`, on peut créer un `GridLayoutManager` qui arrange en tableau d'un certain nombre de colonnes indiqué en paramètre : 

```
// layout manager tableau
GridLayoutManager lm = new GridLayoutManager(this, 2);
ui.recycler.setLayoutManager(lm);
```

On peut aussi choisir l'axe de défilement horizontal : 

```
GridLayoutManager lm =
    new GridLayoutManager(this,
        2,                                // nb lignes ou colonnes
        RecyclerView.HORIZONTAL,          // direction
        false);                           // sens
```

4.4.5. Disposition en blocs empilés

Encore une autre disposition, elle empile des `ViewHolders` qui peuvent avoir des hauteurs (ou largeurs, selon la direction d'empilement) différentes : 

```
// layout manager grille d'empilement
StaggeredGridLayoutManager lm =
    new StaggeredGridLayoutManager(
        2,                                // colonnes
        RecyclerView.VERTICAL);           // empilement
ui.recycler.setLayoutManager(lm);
```

4.4.6. Séparateur entre items

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

```
// séparateur
DividerItemDecoration dividerItemDecoration =
    new DividerItemDecoration(
        this, DividerItemDecoration.VERTICAL);
ui.recycler.addItemDecoration(dividerItemDecoration);
```

Voir [cet échange sur stackoverflow](#) pour davantage d'informations.

4.5. Actions sur la liste

4.5.1. Présentation

Avec tout ce qui précède, la liste s'affiche automatiquement et défile à volonté.

On s'intéresse maintenant à ce qui se passe quand on modifie la liste sous-jacente :

- modifications extérieures au `RecyclerView`, c-à-d le programme Java modifie les données directement dans le `ArrayList`,
- modifications effectuées par le `RecyclerView` suite aux gestes de l'utilisateur sur les éléments (clics, glissés...) mais c'est trop complexe pour ce cours, voir [ItemAnimator](#).

4.5.2. Modification des données

Toute modification extérieure sur la liste des éléments doit être signalée à l'adaptateur afin qu'à son tour il puisse prévenir le `RecyclerView`.

Selon la modification, il faut appeler :

- `notifyItemChanged(int pos)` quand l'élément de cette position a été modifié
- `notifyItemInserted(int pos)` quand un élément a été inséré à cette position
- `notifyItemRemoved(int pos)` quand cet élément a été supprimé
- `notifyDataSetChanged()` si on ne peut pas identifier le changement facilement (tri, réinitialisation...)

4.5.3. Défilement vers un élément

Pour faire défiler afin de rendre un élément visible, il suffit d'appeler l'une de ces méthodes sur le `RecyclerView` :

- `scrollToPosition(int pos)` : fait défiler d'un coup la liste pour que la position soit visible,
- `smoothScrollToPosition(int pos)` : fait défiler la liste avec une animation jusqu'à ce que la position devienne visible.

On peut configurer cette dernière pour avoir un meilleur comportement, voir [cette discussion](#).

4.5.4. Clic sur un élément

Gros **problème** : dans Android, rien n'est prévu pour les clics sur les éléments. On doit construire soi-même une architecture d'écouteurs. Voici d'abord la situation, pour comprendre la solution.

1. L'activité `MainActivity` veut être prévenue quand l'utilisateur clique sur un élément de la liste.
2. L'objet qui reçoit les événements utilisateur est le `ViewHolder`. Il suffit de lui ajouter la méthode `onClick(View v)` de l'interface `View.OnClickListener` (comme un simple `Button`) pour être prévenu d'un clic.
3. Un `RecyclerView` regroupe plusieurs `ViewHolder` ; chacun peut être cliqué (un à la fois). Celui qui est cliqué peut faire quelque chose dans sa méthode `onClick`, mais le problème, c'est que le `ViewHolder` ne connaît pas l'activité à prévenir.

Il faut donc faire le lien entre ces `ViewHolder` et l'activité. Ça va passer par l'adaptateur, le seul qui soit au contact des deux.

1. Il faut que l'activité définisse un écouteur de clics et le fournisse à l'adaptateur. Tant qu'à faire, on peut définir notre propre sorte d'écouteur qui recevra la position de l'objet cliqué en paramètre (c'est le plus simple à faire).
2. L'adaptateur transmet cet écouteur à tous les `ViewHolder` qu'il crée ou recycle.
3. Chaque `ViewHolder` possède donc cet écouteur et peut le déclencher le cas échéant.

Voyons comment créer notre propre type d'écouteur, puis quoi en faire.

4.5.5. Notre écouteur de clics

Il suffit d'ajouter une interface publique dans l'adaptateur :



```
public class PlaneteAdapter
    extends RecyclerView.Adapter<PlaneteViewHolder>
{
    public interface OnItemClickListener {
        void onItemClick(int position);
    }

    private OnItemClickListener listener;

    public void setOnItemClickListener(OnItemClickListener l) {
        this.listener = l;
    }
    ...
}
```

Il faut maintenant que l'adaptateur fournisse cet écouteur aux `ViewHolders` :



```
public void onBindViewHolder(PlaneteViewHolder holder, ...)
{
    ...
    holder.setOnItemClickListener(this.listener);
}
```

Dans le *ViewHolder*, il y a le même *setter* et la même variable. L'adaptateur se contente de fournir l'écouteur à chacun.

Les *ViewHolders* doivent mémoriser cet écouteur :



```
public class PlaneteViewHolder extends RecyclerView.ViewHolder
    implements View.OnClickListener
{
    private final PlaneteBinding ui;
    private PlaneteAdapter.OnItemClickListener listener;

    public void setOnItemClickListener(OnItemClickListener l) {
        this.listener = l;
    }
    ...
}
```

Remarquez comment on fait référence à l'interface définie dans la classe *PlaneteAdapter*.

Les *ViewHolders* doivent aussi recevoir les événements puis déclencher l'écouteur :



```
public PlaneteViewHolder(@NonNull PlaneteBinding ui) {
    super(ui.getRoot());
    this.ui = ui;
    itemView.setOnClickListener(this);
}

@Override public void onClick(View v) {
    if (listener != null)
        listener.onItemClick(getAdapterPosition());
}
```

La méthode *getAdapterPosition()* retourne la position de ce *ViewHolder* dans son adaptateur.

L'adaptateur définit une interface que d'autres classes vont implémenter, par exemple une référence de méthode de l'activité :



```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // adaptateur
        adapter = new PlaneteAdapter(liste);

        // écouteur pour les clics sur les éléments de la liste
        adapter.setOnItemClickListener(this::onItemClick);
    }
    private void onItemClick(int position) {
        Planete planete = liste.get(position);
        ... utiliser planete ...
    }
}
```

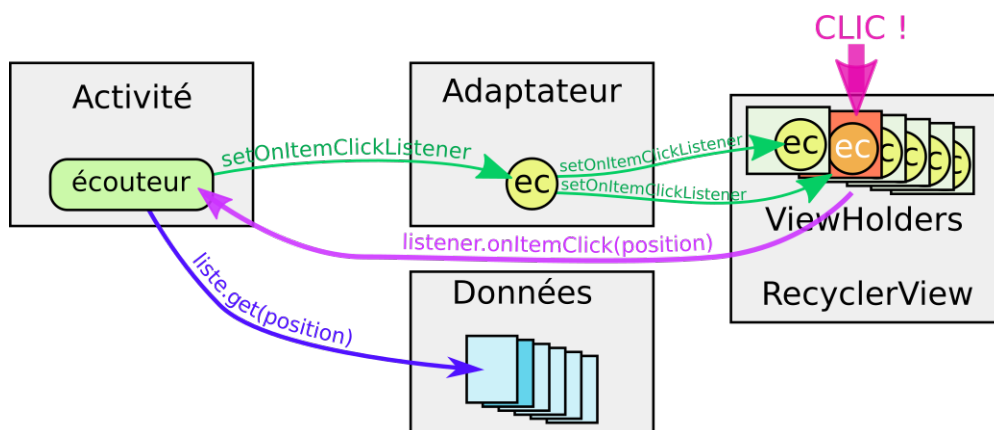


Figure 26: Principe écouteur des clics

4.5.6. Schéma récapitulatif

Ça commence à gauche dans l'activité. Son écouteur est transmis via l'adaptateur à tous les *ViewHolders*. L'un d'eux est activé par un clic et déclenche l'écouteur. L'activité, réveillée, accède à la donnée concernée.

figure 26

4.5.7. Ouf, c'est fini

C'est tout pour cette semaine. La semaine prochaine nous parlerons des menus, dont les menus contextuels qui apparaissent quand on clique longtemps sur un élément d'une liste, dialogues et fragments.

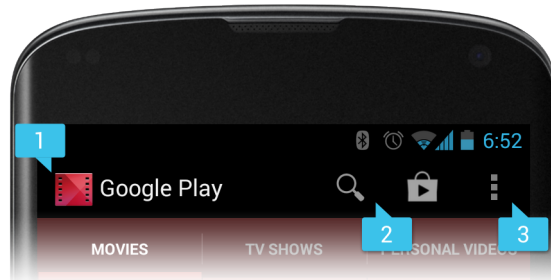


Figure 27: Barre d'action

Semaine 5

Ergonomie

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

- Menus et barre d'action
- Popup-up : messages et dialogues
- Activités et fragments

Et aussi, pour information :

- Préférences
- Bibliothèque support (androidx)

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 27

5.1.2. Réalisation d'un menu

Le principe général : un menu est une liste de d'items présentés dans la barre d'action. La sélection d'un item déclenche une *callback*.

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

Il faut définir :

- un fichier `res/menu/nom_du_menu.xml` qui est une sorte de layout spécialisé pour les menus,



Figure 28: Icônes de menus

- deux méthodes d'écouteur pour gérer les menus :
 - ajout du menu dans la barre,
 - activation de l'un des items.

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 *androidx*.

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 28

Téléchargez l'[Action Bar Icon Pack](#)

pour des icônes à mettre dans vos applications.

5.1.5. Écouteur pour afficher le menu


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.


On peut aussi ajouter des éléments de menu manuellement : 

```
public static final int MENU_ITEM1 = 1;
...


@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    ...
    // ajouter des items manuellement
    menu.add(Menu.NONE, MENU_ITEM1, ordre, titre).setIcon(image);
    ...
}
```

Cette fois, vous devrez choisir un identifiant pour les items. L'ordre indique la priorité de cet item ; mettre `Menu.NONE` s'il n'y en a pas.

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 MENU_ITEM1:
            ...
            return true;
        ...
        default: return super.onOptionsItemSelected(item);
    }
}
```

Mais, dans les versions récentes d'Android Studio, les identifiants de menu ne sont plus des constantes et ne peuvent plus être utilisés dans un `switch`. On doit le transformer en conditionnelles en cascade : 


```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.menu_creer) {
        ...
        return true;
    } else if (id == MENU_ITEM1) {
        ...
        return true;
    }
    ...
    } else 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>
```

5.2. Menus contextuels

5.2.1. Menus contextuels

Voir la figure 29, page 90.

Ces menus apparaissent généralement lors un clic long sur un élément de liste. La classe `RecyclerView` ne possède rien pour afficher automatiquement des menus. Il faut soi-même programmer le nécessaire.

Voici les étapes :

- Le *View Holder* doit se déclarer en tant qu'écouteur pour des ouvertures de menu contextuel (clic long).
- La méthode déclenchée fait apparaître le menu contextuel.
- L'activité doit se déclarer en tant qu'écouteur pour les clics sur les éléments du menu contextuel.

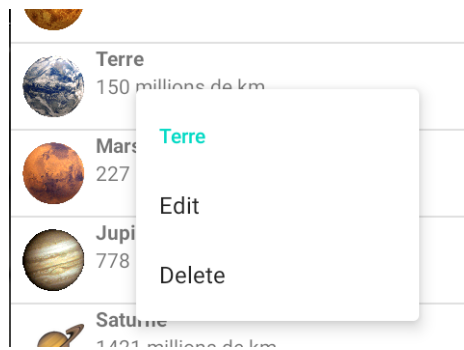


Figure 29: MenuContextuel

Le souci principal, c'est qu'il n'y a pas de lien entre d'une part le *View Holder* qui observe le clic long sur un élément de la liste et fait afficher le menu contextuel, et d'autre part l'activité qui est réveillée quand l'utilisateur sélectionne un item du menu.

Il faut fournir la position de l'élément de la liste à l'activité, et on est obligé de bricoler.

5.2.2. *View Holder* écouteur de menu

Il suffit d'ajouter ceci :



```
public class PlaneteViewHolder extends RecyclerView.ViewHolder
{
    private PlaneteBinding ui;

    public PlaneteViewHolder(@NonNull PlaneteBinding ui)
    {
        super(ui.getRoot());
        this.ui = ui;
        // pour faire apparaître le menu contextuel
        itemView.setOnCreateContextMenuListener(
            this::onCreateContextMenu);
    }
}
```

`itemView` est une variable de la classe `ViewHolder` qui est égale à `ui.getRoot()`.

Voici la *callback* d'un clic long sur un élément de la liste :



```
public static final int MENU_EDIT = 1;
public static final int MENU_DELETE = 2;

private void onCreateContextMenu(ContextMenu menu, View v,
    ContextMenu.ContextMenuInfo menuInfo)
{
    // position de l'élément
    int position = getAdapterPosition();
    // stocker la position dans l'ordre (3e paramètre)
    menu.add(Menu.NONE, MENU_EDIT, position, "Edit");
}
```

```
menu.add(Menu.NONE, MENU_DELETE, position, "Delete");  
// titre du menu  
menu.setHeaderTitle(ui.nom.getText());  
}
```

Il y a une astuce, mais un peu faible : utiliser la propriété `order` des items de menu pour stocker la position de l'élément cliqué dans la liste. Cette propriété permet normalement de les classer pour les afficher dans un certain ordre, mais comme on ne s'en sert pas...

On est obligé de faire ainsi car il manque une propriété *custom* dans la classe `MenuItem`. Une meilleure solution serait de sous-classer cette classe avec la propriété qui nous manque, mais il faudrait modifier beaucoup plus de choses.

Une dernière remarque : il n'est pas possible d'afficher un icône à côté du titre d'item. C'est un choix délibéré dans Android.

5.2.3. Écouteur dans l'activité

Enfin, il reste à rendre l'activité capable de recevoir les événements d'un clic sur un item du menu : 

```
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
    int position = item.getOrder(); // récup position  
    Planete planete = liste.get(position); // récup item  
    switch (item.getItemId()) {  
        case PlaneteViewHolder.MENU_EDIT:  
            // TODO éditer planete (activité ou fragment...)  
            return true;  
        case PlaneteViewHolder.MENU_DELETE:  
            // TODO supprimer planete (dialogue confirmation...)  
            return true;  
    }  
    return false; // menu inconnu, non traité ici  
}
```

5.3. Annonces et dialogues

5.3.1. 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.

Voir la figure 30, page 92.

Voici comment l'afficher avec une ressource chaîne : 

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

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

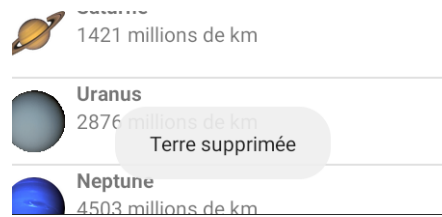


Figure 30: Toast

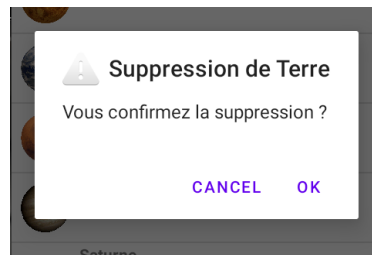


Figure 31: Dialogue d'alerte

5.3.2. 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 31

Il existe plusieurs sortes de dialogues :

- Dialogues d'alerte
- Dialogues généraux

5.3.3. 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 : 

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

Ensuite, on rajoute les boutons et leurs écouteurs.

NB: utiliser des ressources pour les chaînes.

5.3.4. 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. 

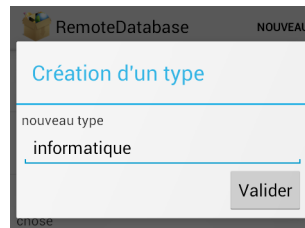


Figure 32: Dialogue perso

```
// rajouter un bouton "oui" qui supprime vraiment
builder.setPositiveButton(android.R.string.ok,
    // écouteur écrit sous la forme d'une lambda
    (dialog, idbtn) -> supprimerVraiment(planete));
// rajouter un bouton "non" qui ne fait rien
builder.setNegativeButton(android.R.string.cancel, null);
```

Enfin, on affiche le dialogue :

```
// affichage du dialogue
builder.show();
```

5.3.5. 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 32

5.3.6. 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, mais pas pour annuler car on peut fermer le dialogue avec le bouton `back`.

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

Ensuite cela ressemble à ce qu'on fait dans `onCreate` d'une activité : installation du layout et des écouteurs pour les boutons.

5.3.7. Affichage du dialogue

```
Dialog dialog = new Dialog(this);
DialogPersoBinding dialogUI =
    DialogPersoBinding.inflate(getLayoutInflater());
dialog.setContentView(dialogUI.getRoot());
dialog.setTitle("Création d'un type");
// bouton valider
dialogUI.valider.setOnClickListener(v -> {
    // récupérer et traiter les infos
    ...
    // ne surtout pas oublier de fermer le dialogue
    dialog.dismiss();
});
// afficher le dialogue
dialog.show();
```

5.4. Fragments et activités

5.4.1. Fragments

Un fragment est un sous-ensemble d'une interface d'application, par exemple :

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

Un *fragment* est une sorte de mini-activité restreinte à une seule chose : afficher une liste, afficher les informations d'un élément, etc.

Une activité peut être composée d'un ou plusieurs fragments, qui sont visibles ou non selon la géométrie du smartphone.

5.4.2. 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 33, page 95.

5.4.3. Différents types de fragments

Il existe différents types de fragments :

- `Fragment` superclasse, pour des fragments normaux.
- `DialogFragment` pour afficher un fragment dans une fenêtre flottante au dessus d'une activité.
- `PreferenceFragment` pour gérer les préférences.

En commun : il faut surcharger la méthode `onCreateView` qui met leur interface utilisateur en place. Si on utilise les View Bindings, `onCreateView` retourne simplement `ui.getRoot()`.

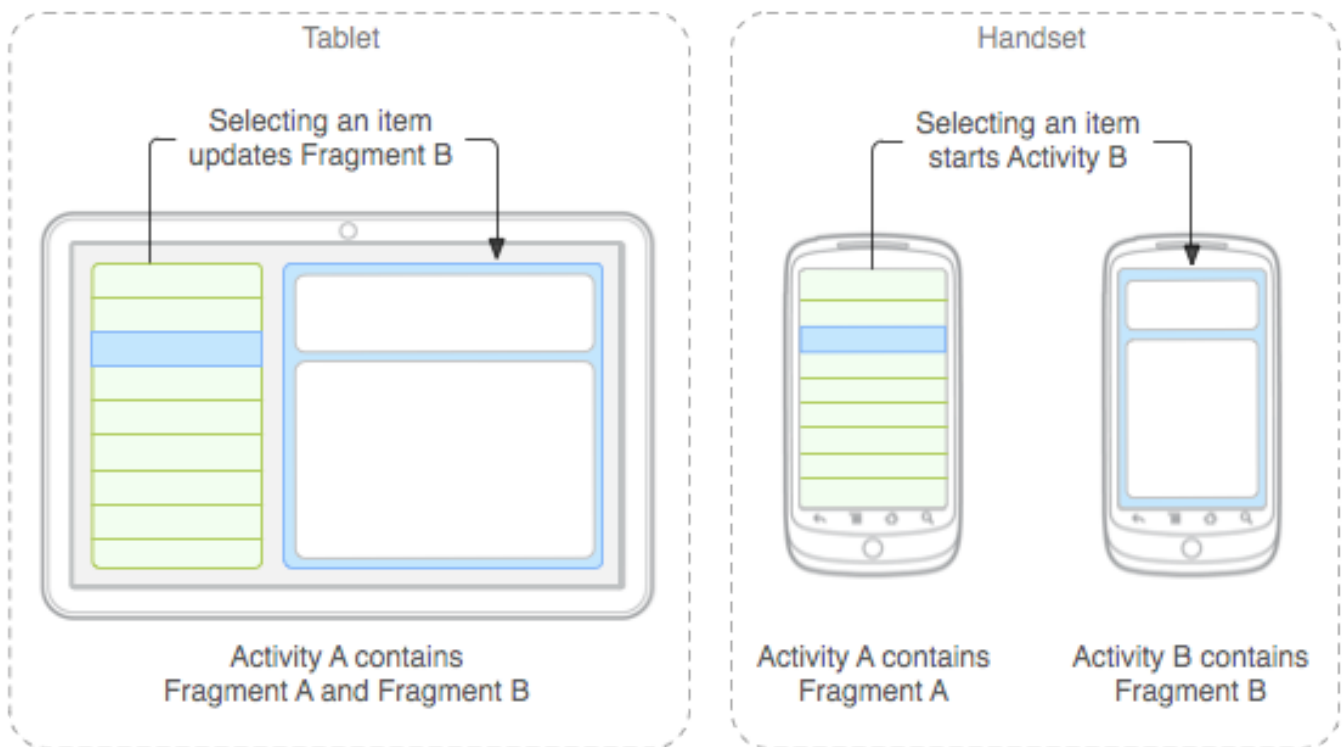


Figure 33: Différentes apparences

5.4.4. Structure d'un fragment

Un fragment est une activité très simplifiée qui contient au moins un constructeur vide et `onCreateView` surchargée :

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

    @Override
    public View onCreateView(LayoutInflater inflater,
        ViewGroup container, Bundle savedInstanceState) {
        ui = PlaneteInfosBinding.inflate(inflater, container, false);

        // écouteurs, adaptateur...

        return ui.getRoot();
    }
}
```

Dans le cas du fragment Liste, c'est lui qui crée l'adaptateur :


```
public class ListeFragment extends Fragment {
    ...
    private List<Planete> liste;
```

```
private PlaneteAdapter adapter;

@Override
public View onCreateView(LayoutInflater inflater, ...) {
    ...
    ... récupérer la liste (application ou BDD)
    adapter = new PlaneteAdapter(liste);
    ... layout manager, séparateur, écouteurs...

    return ui.getRoot();
}
}
```

5.4.5. Menus de fragments

Un fragment peut définir un menu d'options dont les 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 mInf)
{
    mInf.inflate(R.menu.edit_fragment, menu);
    super.onCreateOptionsMenu(menu, mInf);
}
```

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

5.4.6. 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.
- dynamiquement : les fragments sont ajoutés, enlevés ou remplacés en cours de route selon les besoins (on ne verra pas).

5.4.7. Fragments statiques dans une activité

Dans ce cas, c'est le layout de l'activité qui inclut les fragments, p. ex. `res/layout/activity_main.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"
```

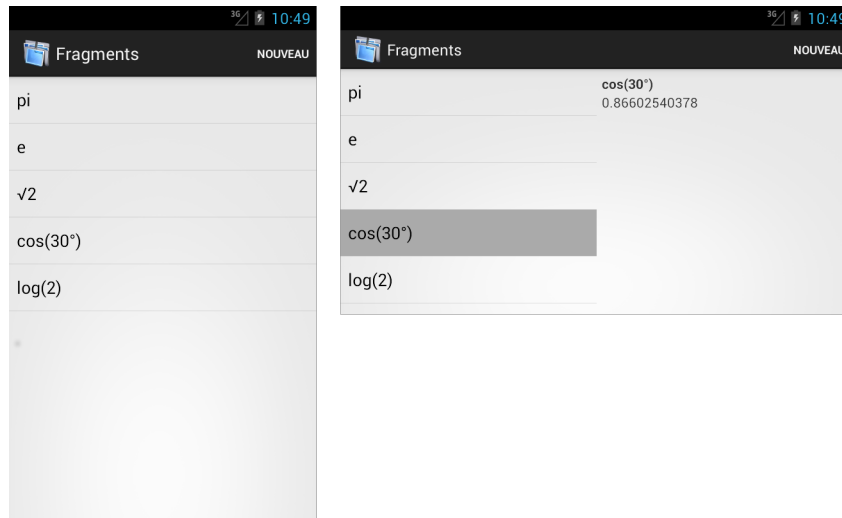



Figure 34: Un ou deux fragments affichés

```
... />  
</LinearLayout>
```

Chaque fragment doit avoir un identifiant et un nom de classe complet avec tout le *package*. Ne pas oublier les attributs des tailles et éventuellement poids.

5.4.8. Disposition selon la géométrie de l'écran

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

figure 34

5.4.9. Changer la disposition selon la géométrie

Pour cela, il suffit de définir deux layouts (des « variantes ») :

- res/layout-port/activity_main.xml en portrait :



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

- res/layout-land/activity_main.xml en paysage :



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

5.4.10. 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`.

Pour savoir si le fragment `frag_infos` est affiché :



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

Notez que le fragment sera aussi mis en `@Nullable` dans le *view binding* de l'activité.

5.4.11. 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 doit être 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.

5.4.12. Interface pour un écouteur

On reprend l'interface `OnItemClickListener` définie dans l'adaptateur, voir la fin du cours 4 :



```
public interface OnItemClickListener {
    void onItemClick(int position);
}
```

Ce sera l'activité principale qui sera cet écouteur dans l'adaptateur du fragment, grâce à :



```
@Override
public View onCreateView(LayoutInflater inflater, ...) {
    ...
    adapter.setOnItemClickListener(
        (PlaneteAdapter.OnItemClickListener) getActivity());
    ...
}
```

5.4.13. Écouteur de l'activité

Voici maintenant l'écouteur de l'activité principale :



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

5.4.14. Relation entre deux classes à méditer

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);
    }
}
```

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;
```

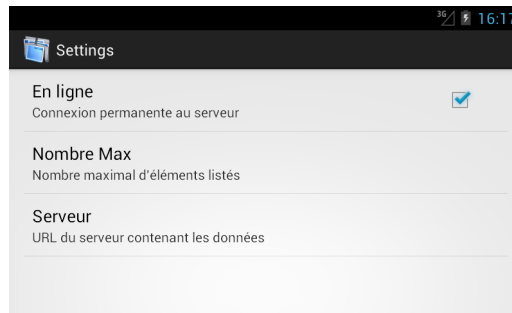


Figure 35: Préférences de l'application

```
public Classe2() {  
    ...  
    objet1.setOnEventListener(this);  
}  
  
public void onEvenement(int param) {  
    ...  
}  
}
```

5.5. Préférences d'application

5.5.1. Illustration

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

figure 35

5.5.2. 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.

5.5.3. 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>
```

5.5.4. 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.

5.5.5. 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.

5.5.6. 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"));
```

5.5.7. 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();
```

5.5.8. 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.

5.5.9. 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.

5.6. Bibliothèque support

5.6.1. 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.

5.6.2. 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="20"
    android:targetSdkVersion="32" />
```

Avec ce manifeste, si la tablette n'est pas au moins en API niveau 20, l'application ne sera pas installée. L'application est garantie pour bien fonctionner jusqu'à l'API 32 incluse.

5.6.3. 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.

5.6.4. 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`.

5.6.5. 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:1.4.1"
implementation "androidx.recyclerview:recyclerview:1.2.0"
```

5.6.6. 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.

5.6.7. 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` :

```
android {  
    compileSdkVersion 30  
  
    defaultConfig {  
        applicationId "mon.package"  
        minSdkVersion 20  
        targetSdkVersion 30  
    }  
}
```

Ensuite, il faut ajouter les dépendances :

```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    ...  
    implementation "androidx.appcompat:appcompat:1.4.1"  
    implementation "androidx.recyclerview:recyclerview:1.2.0"  
    ...  
}
```

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.

5.6.8. 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.

5.6.9. 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

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](#). 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 Lombok

Le plugin est installé par défaut dans toutes les versions récentes de Android Studio.

Pour l'utiliser dans un projet, il faut rajouter deux lignes dans le fichier `app/build.gradle` : 

```
annotationProcessor 'org.projectlombok:lombok:1.18.16'
implementation 'org.projectlombok:lombok:1.18.16'
```

NB: Le plugin n'a pas été mis à jour pour les versions actuelles de Android Studio. Le site GitHub semble à l'arrêt depuis un an.

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

6.2.2. Autre ORM sur Android : *Room*

Dans tout appareil Android, il y a un SGBD appelé [SQLite](#). C'est un moteur complet et performant pour des bases de données embarquées, de petite taille et mono-utilisateur.

Google propose un ORM appelé [Room](#) qui s'appuie sur SQLite. Pour simplifier, on définit une classe par table de la base de données – les variables membres de ces classes sont les colonnes des tables. Ensuite, on définit des classes contenant des méthodes d'accès aux tables : obtenir la liste des instances, ajouter, supprimer une instance, etc. Et pour finir on définit une dernière classe pour représenter la base de données entière : création, suppression.

Malheureusement, Room n'est pas utilisable pour des bases de données distantes, partagées entre différents utilisateurs.

6.2.3. Realm vs 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 un utilisateur sont transmises à tous les autres utilisateurs, et cette transmission est très efficace. Donc chaque utilisateur voit les changements en temps réel, quelque soit sa plateforme.
- Realm s'appuie maintenant sur MongoDB pour stocker les données. Comme c'est trop complexe pour ce cours, nous utiliserons une version obsolète plus facile à comprendre.

6.2.4. Configuration d'un projet Android avec Realm Legacy

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:7.0.8'
```

- celui du dossier `app` :

```
    apply plugin: 'realm-android'
    android {
        ...
    }
    dependencies {
        implementation 'io.realm:android-adapters:4.0.0'
        ...
    }
}
```


6.2.5. 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.

6.2.6. 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();
        ...
    }
}
```

6.2.7. 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.

6.2.8. 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 (payant mais raisonnable). 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.

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. Requêtes 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](#) (ancienne version de Realm).

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<Achats> 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. Requetes 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 ProduitViewHolder
    onCreateViewHolder(ViewGroup parent, int viewType)
{
    ProduitBinding binding =
        ProduitBinding.inflate(
            LayoutInflater.from(parent.getContext()),
            parent, false);
    return new ProduitViewHolder(binding);
}
```

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(ProduitViewHolder holder, int position)
{
    // afficher les informations du produit
    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.

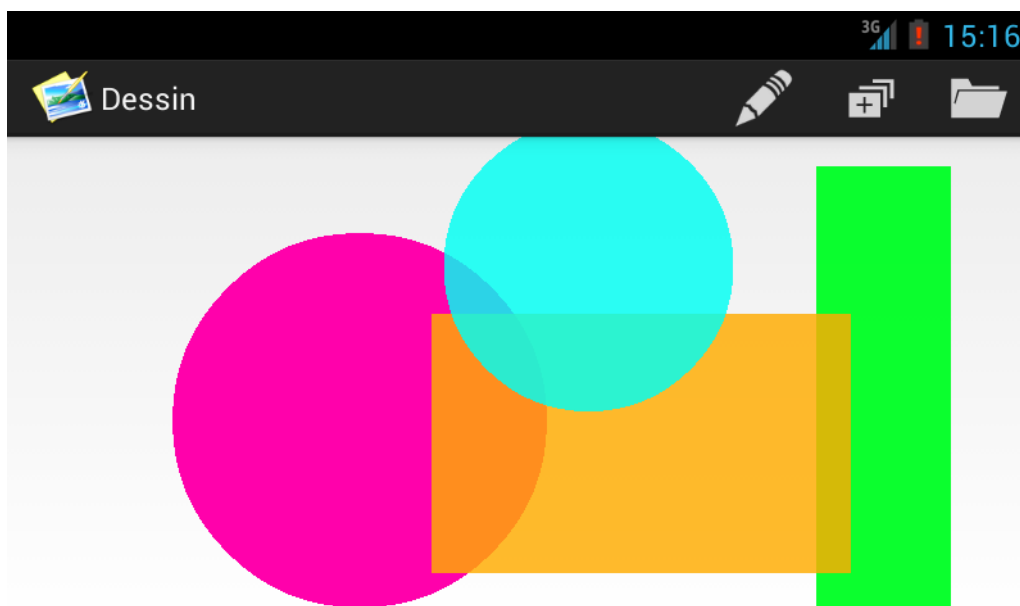


Figure 36: Application de dessin

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. Objectif : cette application

figure 36

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 {
    private 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)` est appelée pour dessiner la vue. Cette méthode doit être rapide. Également, elle ne doit faire aucun `new`. Il faut donc créer tous les objets nécessaires à l'avance, par exemple dans le constructeur de la vue.

Son paramètre `canvas` représente la zone de dessin. Un `Canvas` est une structure qui regroupe un *bitmap* (tableau de pixels) et des *transformations matricielles* (ex: rotations, translations...) qui modifient tous les dessins. Et la classe `Canvas` définit aussi de nombreuses méthodes de dessin.

7.1.5. Méthodes de la classe Canvas

La classe `Canvas` possède de nombreuses méthodes de dessin :

- `drawPoint (float x, float y, Paint peinture)` : dessine un point en (x,y) avec la peinture.
- `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.
- `drawText(String texte, float x, float y, Paint peinture)` avec des variantes...
- `drawColor(int color)` : efface le *canvas* entier avec la couleur indiquée (transparent suivant).
- etc.

7.1.6. Représentation des couleurs dans Android

Dans Android, les couleurs sont représentées à l'aide d'entiers 32 bits, voir [la doc](#).

- L'octet de poids fort est appelé A ou *canal alpha* et définit la transparence : 0=invisible, 255=opaque.
- L'octet suivant représente la composante rouge, le suivant pour le vert et l'octet de poids faible représente le bleu.

La classe `Color` offre des constantes et méthodes pour construire une couleur ou extraire les composantes :

- `Color.BLACK`, `Color.RED`... : couleurs prédéfinies,
- `Color.rgb(int r, int v, int b)` : regroupe des composantes RVB 0..255 séparées en un code de couleur,
- `Color.red(code)`, `Color.green(code)`, `Color.blue(code)`, `Color.alpha(code)`

7.1.7. Peinture Paint

Pour dessiner, on ne fournit pas une couleur, mais une « peinture ».

La classe `Paint` permet de représenter des modes de dessin complexes : couleurs des lignes, de remplissage, polices, lissage...

C'est extrêmement riche. Voici un exemple :



```
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`.

7.1.8. 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.

Ces méthodes sont inspirées de ce que propose la norme SVG.



Figure 37: Dégradé horizontal

7.1.9. 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);  
mPeinture = new Paint(Paint.ANTI_ALIAS_FLAG);  
mPeinture.setShader(shaderMotif);  
mPeinture.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`).

7.1.10. 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);  
mPeinture = new Paint(Paint.ANTI_ALIAS_FLAG);  
mPeinture.setShader(shader);
```

figure 37

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.



Figure 38: Dessin vectoriel XML

7.1.11. Quelques remarques

Lorsqu'il faut redessiner la vue, appelez `invalidate`. Si la demande de réaffichage est faite dans un autre *thread*, alors c'est `postInvalidate` qu'il faut appeler.

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.

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

7.1.12. « Dessinables »

Les canvas servent à dessiner des figures géométriques, rectangles, lignes, etc, mais aussi des `Drawable`, c'est à dire des « choses dessinables » telles que des images bitmap ou des formes quelconques. Il existe beaucoup de sous-classes de `Drawable`.

Un *drawable* est créé :

- par une image PNG ou JPG dans `res/drawable...`



```
Bitmap bm = BitmapFactory
    .decodeResource(getResources(), R.drawable.image);
Drawable d = new BitmapDrawable(getResources(),bm);
```

7.1.13. Dessinables » vectoriels

- Un *drawable* peut aussi être une forme vectorielle dans un fichier XML. Ex : `res/drawable/carre.xml` :



```
<?xml version="1.0" encoding="UTF-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <stroke android:width="4dp" android:color="#F000" />
    <gradient android:angle="90"
        android:startColor="#FFBB"
        android:endColor="#F77B" />
    <corners android:radius="16dp" />
</shape>
```

figure 38

7.1.14. Variantes

Android permet de créer des « dessinables » à variantes par exemple pour des boutons personnalisés.



```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="...">

    <item android:drawable="@drawable/button_pressed"
          android:state_pressed="true" />
    <item android:drawable="@drawable/button_checked"
          android:state_checked="true" />
    <item android:drawable="@drawable/button_default" />

</selector>
```

L'une ou l'autre des images sera choisie en fonction de l'état du bouton, enfoncé, relâché, inactif.

7.1.15. Utilisation d'un Drawable

Ces objets dessinable peuvent être employés dans un canvas. Puisque ce sont des objets vectoriels, il faut définir les coordonnées des coins haut-gauche et bas-droit, ce qui permet d'étirer la figure.

```
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, et non pas l'inverse.

NB: la première instruction est à placer dans le constructeur de la vue, afin de ne pas ralentir la fonction de dessin.

7.1.16. Enregistrer un dessin dans un fichier

C'est très facile. Il suffit de récupérer le bitmap associé à la vue, puis de le compresser en PNG.

```
public void save(String filename)
{
    // obtenir le bitmap associé au Canvas
    Bitmap bitmap = getDrawingCache();
    try {
        // écrire le bitmap dans un fichier, au format PNG
        FileOutputStream out = new FileOutputStream(filename);
        bitmap.compress(Bitmap.CompressFormat.PNG, 90, out);
        out.close();
    } catch (Exception e) {
        ...
    }
}
```

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, dont `onTouchEvent`.

Son paramètre, `event` 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.setP1(x, y);
        figures.add(figure);
        break;
    case MotionEvent.ACTION_MOVE:
        if (figures.size() < 1) return true;
        figure = figures.getLast();
        figure.setP2(x,y);
        break;
}
invalidate();
```

7.2.3. Automate pour gérer les actions

L'algo précédent peut se représenter à l'aide d'un automate 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.

Voir la figure 39, page 129.

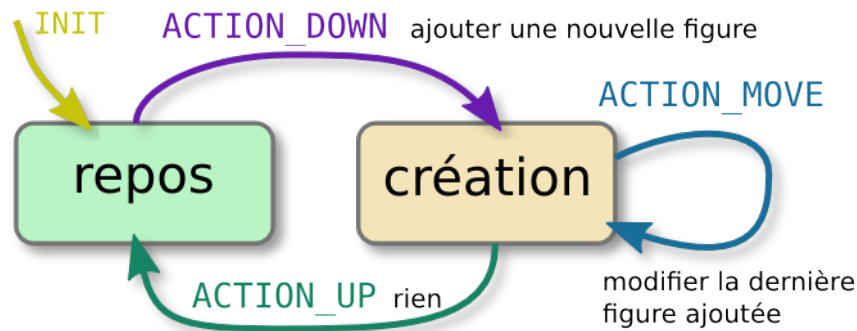


Figure 39: Automate

7.2.4. Programmation d'un automate

Pour coder l'automate précédent, on a besoin d'une variable qui représente l'état courant. Sa valeur est prise dans une énumération :

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

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

NB: on pourrait utiliser des classes comme dans le patron de conception **State**, à la place des **switch** du transparent suivant.

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



Figure 40: Sélectionneur de couleur

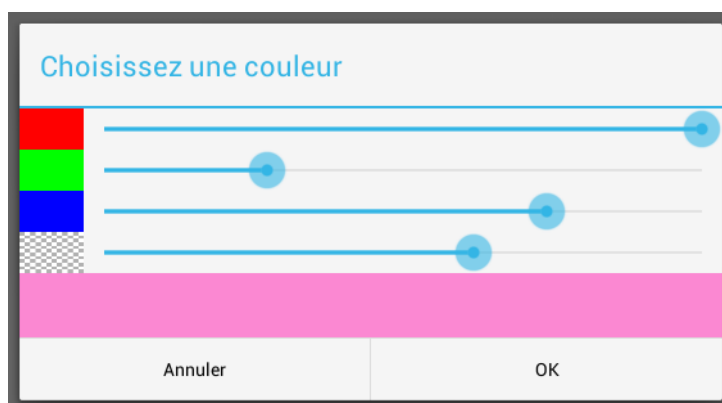


Figure 41: Sélectionneur de couleur simple

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 40

7.3.2. Version simple

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

figure 41

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 sélecteur.

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 OnColorChangedListener {  
    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.ok,  
        (dialog, btn) -> {  
            // prévenir l'écouteur s'il y en a un  
            if (mListener != null)  
                mListener.onColorChanged(cpv.getColor());  
        });  
    builder.setNegativeButton(android.R.string.cancel, 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
    ColorPickerViewBinding ui =
        ColorPickerViewBinding.inflate(
            LayoutInflater.from(context), this, true);
    ...
}
```

7.3.7. Layout de cette vue

Le layout color_picker_view.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 : 

```
ui.sbRouge.setOnSeekBarChangeListener(
    new OnSeekBarChangeListener() {
        public void onProgressChanged(SeekBar 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. Affichage du dialogue

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

```
new ColorPickerDialog(color -> {  
    // utiliser color = la couleur choisie dans le dialogue  
    ...  
}).show(getFragmentManager(), "colorpickerdlg");
```

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

Techniquement, cet écouteur est l'implémentation de l'interface `OnColorChangeListener` du dialogue. C'est une lambda et ses instructions entre `{}` sont celles de la méthode `onColorChanged`.

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, c'est vérifier qu'une classe et ses méthodes font réellement ce qui est prévu dans les spécifications.

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, en particulier en développement agile.

AndroidStudio permet d'effectuer deux sortes de tests :

- des tests unitaires pour vérifier des méthodes 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 $\forall x \in R^+, racine(x)^2 = x$.

Pour un début, on pourrait essayer ceci (?) :

```
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. Précision des nombres

Il faut se méfier de la précision des données. En effet, par nature, certains nombres sont mal représentés (format interne IEEE : base 2 et non pas base 10). 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 (cf [explications](#)).

Si on veut mieux tester la fonction *racine*, on doit faire 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** avec $v_1 == v_2$. Le ϵ est à déterminer empiriquement.

8.1.3. 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, si le test emploie la même définition que la fonction, par exemple, le même développement en série limitée, ça ne prouvera pas qu'elle est bonne. S'il y a une erreur, on ne s'en rendra pas compte. Pour tester correctement, il faut trouver un algorithme totalement différent.

Le testeur doit donc être 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. Un bon testeur possède une grande expérience en programmation, et un esprit « taquin » pour deviner les oublis des programmeurs.

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 une par une. 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 paquetage que les classes testées.

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

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

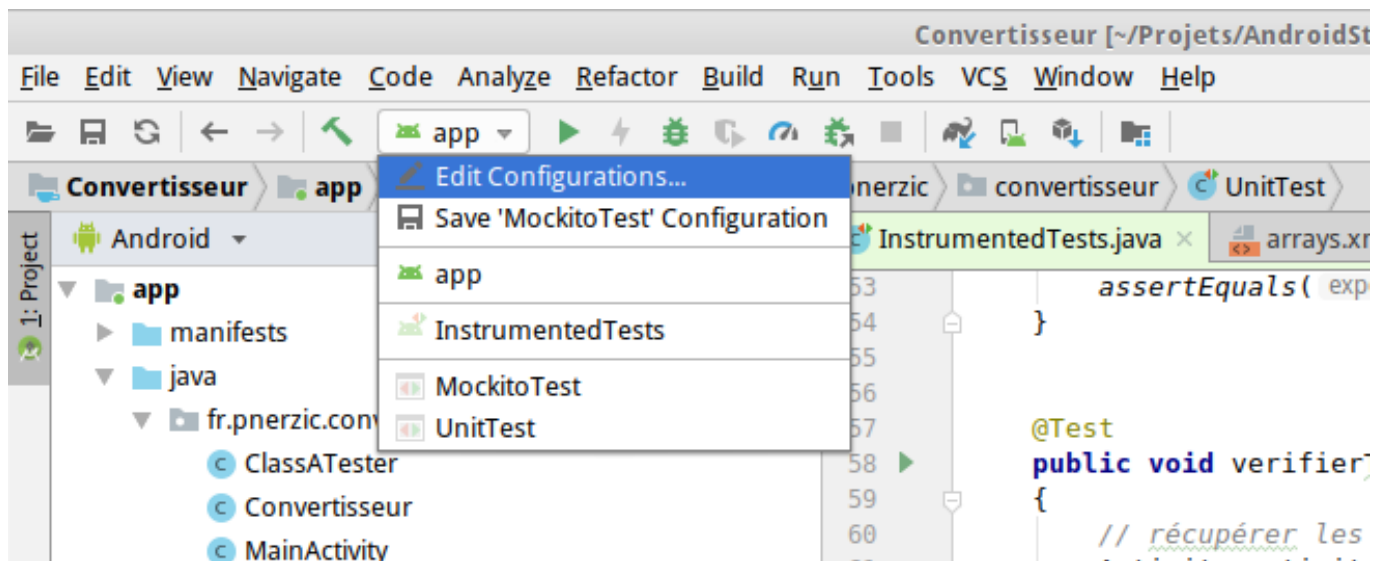


Figure 42: Configuration de lancement

- ses tests dans `app/src/test/java/fr/iutlan/tp8`,
- ses tests d'interface dans `app/src/androidTest/java/fr/iutlan/tp8`.

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 menu 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 42

8.2.3. Ouvrir une classe aux tests

Normalement, les méthodes d'une classe ne sont pas toutes publiques. Certaines sont privées, d'autres protégées. Pour qu'un test puisse y avoir accès, on peut rajouter l'annotation `@VisibleForTesting` :

```
@VisibleForTesting
CeciCela getCeciCela() {
    return this.cecicela;
}
```

Ainsi, la méthode est normalement visible seulement des classes du même package (pas de mot-clé `private` ou `protected`), mais grâce à l'annotation, elle devient accessible des tests comme si elle était publique.

8.2.4. JUnit4

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




```
package fr.iutlan.tp8;

import org.junit.Test;
import static org.junit.Assert.assertEquals;    // <-- static

public class TestClasseChose
{
    @Test          // <-- annotation à ne pas oublier
    public void testMethodeASur5()
    {
        int resultat = Chose.methodeA(5);    // appel methodeA
        assertEquals(50, resultat);         // vérif résultat
    }
}
```

8.2.5. 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 attendue 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.

8.2.6. Remarque : `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));
}
```

8.2.7. Assertions JUnit

Voici une partie du catalogue des méthodes d'assertions.

- 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)` échoue si l'objet vaut `null`
- 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 compris correctement : les messages d'erreurs seront inversés.

```
assert*(résultat attendu, résultat obtenu)
```

8.2.8. 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 but 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.

8.2.9. Vérification de plusieurs assertions

Plusieurs assertions peuvent être spécifiées à la suite. Si l'une des assertions échoue, toute la méthode de test est en échec, les assertions suivantes ne sont pas essayées. Par contre, les autres tests sont lancés quand même.

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

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

8.2.10. Vérification des exceptions

On peut vérifier qu'une méthode émet une exception dans certains cas. On ajoute un paramètre `expected` à l'annotation :

```
@Test(expected=ArithmeticException.class)
public void testRacineNegative() {
    float rac = racine(-2);
    fail("racine(<0) doit déclencher une exception");
}

@Test(expected=NumberFormatException.class)
public void testParseNonInt() {
    int n = Integer.parseInt("1001 nuits");
    fail("parseInt(non entier) doit déclencher une exception");
}
```

Le test échoue s'il n'y a pas l'exception attendue.

8.2.11. 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

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

Ces deuxièmes paramètres de `assertThat` sont appelés *matchers* (« correspondeurs »). Ils sont nombreux et 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 élaborées.

- Matchers 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));
```

- Matchers sur des chaînes
 - `isEmptyString()`
 - `hasLength(entier)`
 - `equalTo(texte)`
 - `equalToIgnoringCase(texte)`
 - `equalToIgnoringWhiteSpace(texte)`
 - `containsString(texte)`
 - `endsWith(texte), startsWith(texte)`
 - `matchesPattern(regex)` expression régulière Java

```
String msg = "Salut les gens";
assertThat(msg, hasLength(14));
assertThat(msg, containsString("les"));
assertThat(msg, startsWith("Salut"));
assertThat(msg, equalToIgnoringCase("saLut lEs gEnS"));
assertThat(msg, matchesPattern("[A-Z][a-z ]+"));
```

- 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:4.13.2'  
testImplementation 'org.hamcrest:hamcrest-library:2.2'
```

NB: comme toujours, les numéros de version évoluent régulièrement.

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 (*arrange*)

Les données permettant de mener les tests peuvent être préparées dans chaque méthode, rendant les tests indépendants. 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 ». Ce sont des éléments annotés par `@Rule`. Elles permettent, entre autres, de réemployer les mêmes préparations dans différentes classes de test. Mais ces règles, employées dans le cadre Android, sont trop complexes pour ce cours.

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);
}
```

Il y a une convention de nommage des méthodes : `setup` pour l'initialisation et `tearDown` pour la terminaison.

8.4.6. « Règles » de test

JUnit propose une autre écriture des méthodes `@Before` et `@After` : les *règles*.

Il faut implémenter l'interface `TestRule` qui contient une méthode nommée `apply`. Cette méthode doit retourner un `Statement`. C'est lui qui gère le lancement du test dans un cadre voulu. voir le transparent suivant.

```
public class MyTestRule implements TestRule {
    @Override
    public Statement apply(Statement base, Description description) {
        return new Statement() {
            ... voir le transparent suivant ...
        };
    }
}
```

8.4.7. Implémentation de l'interface `TestRule`

Voici maintenant comment on programme ce *statement* :

```
return new Statement() {
    @Override
    public void evaluate() throws Throwable {
        // initialisation = ce qu'il y a dans @Before
        ...
        try {
            // exécution du test
            base.evaluate();
        } finally {
            // terminaison = ce qu'il y a dans @After
            ...
        }
    }
};
```

8.4.8. Utilisation d'une règle

Enfin, voici son utilisation. Il suffit de définir une variable annotée par `@Rule`. Le *statement* sera lancé automatiquement :

```
@Rule
public MyTestRule rule = new MyTestRule();
```

L'intérêt des règles est de pouvoir y rajouter ce qu'on veut, membres et méthodes pour définir totalement le cadre du test.

8.5. Tests paramétrés


8.5.1. Utilité des 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.

8.5.2. 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.

8.5.3. Fourniture des paramètres

Il y a de très nombreuses possibilités pour fournir les paramètres, voir [ces exemples](#), et l'une consiste à programmer une méthode statique qui retourne un tableau de tableaux de paramètres.

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



```
static Object parametersForTestBissextile() {
    return new Object[][]{
        {2019, false},
        {2020, true},
        {2021, false},
    };
}
```

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

8.5.4. Importation de JUnitParams

Pour utiliser les JUnitParams, il faut importer ses classes :



```
import junitparams.JUnit4ParamsRunner;
import junitparams.Parameters;
```

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



```
testImplementation 'junit:junit:4.13.2'
testImplementation 'pl.pragmatists:JUnitParams:1.1.1'
```

8.6. Tests d'intégration

8.6.1. 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 pour la compilation,
- **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, mais seulement sur certaines valeurs.

8.6.2. 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;
}
```

8.6.3. 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 calcu;
```

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.

8.6.4. Apprentissage de résultats

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

- 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.

8.6.5. 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)`.

8.6.6. *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. 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);
```

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

8.6.7. 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. Dans ces cas, la première écriture provoque une `NullPointerException`.

8.6.8. Simulation pour une autre classe

Les exemples précédents consistent à simuler et tester la même instance d'une 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, donc :

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

8.6.9. 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) :

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

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

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

8.6.10. Installation de Mockito

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



```
testImplementation 'junit:junit:4.13.2'  
testImplementation 'org.mockito:mockito-core:3.8.0'
```

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



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

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

8.7. Tests d'intégration Android sans AVD

8.7.1. Présentation

On souhaiterait vérifier si une activité emploie correctement d'autres classes, par exemple que des saisies utilisateur sont correctement relues, et que des résultats sont correctement affichés sur l'interface. Le chapitre d'après, page 151, montre comment on fait cela sur un AVD, mais dans un premier temps, voici une technique plus simple sans lancer d'AVD.

Mockito ne pouvant pas être employé, on utilise un autre outil appelé **Robolectric**. Il redéfinit tout le système Android, contexte, vues, ressources, etc. sous forme de mocks et de stubs. Ils les appelle des ombres (*shadows*), mais c'est trop complexe pour ce cours. Ces maquettes permettent de faire comme si l'activité tournait vraiment.

8.7.2. Installation de Robolectric

Voici comment ajouter Robolectric à un projet Android. Il faut placer ceci dans `app/build.gradle` :



```
testImplementation 'org.robolectric:robolectric:4.7.3'
```

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



```
@RunWith(RobolectricTestRunner.class)
public class TestsMainActivity {

    @Before public void initEachTest() ...

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

8.7.3. Lancement d'une activité par Robolectric

Soit une activité `MainActivity` à tester. On veut savoir si ses méthodes concernant l'interface fonctionnent bien.

D'abord chaque méthode de test doit lancer l'activité et récupérer le *view binding* :



```
private MainActivity activity;
private ActivityMainBinding ui;

@Before
public void initEachTest() {
    activity = Robolectric
        .buildActivity(MainActivity.class)
        .create()
        .get();
    ui = activity.getUI();
}
```

8.7.4. Emploi de Robolectric

Ensuite, c'est très simple. On peut appeler les méthodes publiques de l'activité et utiliser les vues du *view binding* `ui` comme on le ferait dans l'activité. Par exemple :

```
@Test
public void TestSaisieChecked() {
    // arrange : mettre une chaîne dans le EditText
    ui.texte.setText("valeur à tester");

    // act : appeler une méthode de l'activité
    activity.processTexte();

    // assert : vérifier qu'un checkbox est coché
    assertThat(ui.check.isChecked(), is(true));
}
```

8.7.5. Remarques

Vous constatez qu'il est indispensable que l'activité soumise à des tests rende publiques un certain nombre de méthodes, afin qu'on puisse voir si elle se comporte comme prévu. Par exemple, il faut qu'on puisse récupérer son interface utilisateur (*View Binding*), donc il faut un *getter* pour cela. Et il faut des *getters* si on veut inspecter d'autres variables membres.

D'autre part il est préférable que les méthodes soient bien découpées, qu'elles ne fassent pas plusieurs choses à la fois. Cela impose une conception saine du logiciel.

8.8. Tests sur AVD

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

Le schéma général est ([] signifie *optionnel*) :

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

Par exemple :

```
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());
```

8.8.2. Directives Espresso

- Correspondeurs 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`.

8.8.3. ViewMatchers d'Espresso

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")));
```

8.8.4. ViewActions d'Espresso

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

- `click()`, `pressBackUnconditionally()`
- `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()));
```

8.8.5. 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)
```


8.8.6. 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());
```


8.8.7. Installation de Espresso

Il faut ajouter ceci dans `app/build.gradle` : 

```
testImplementation 'junit:junit:4.13.2'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'
androidTestImplementation 'androidx.test.espresso:espresso-intents:3.3.0'
androidTestImplementation 'androidx.test.espresso:espresso-web:3.3.0'
```

NB: cliquez sur l'icône de téléchargement pour voir les lignes entières.

8.8.8. Classe de test

Il faut également préparer la classe de test et les méthodes d'une manière un peu spéciale, avec une règle `@Rule` : 

```
@RunWith(AndroidJUnit4.class)
public class TestsMainActivitySurAVD
{
    public @Rule ActivityScenarioRule<MainActivity>
        asr = new ActivityScenarioRule<>(MainActivity.class);

    private MainActivity activity;
```

Cela définit un « scénario » permettant de lancer l'activité automatiquement à chaque test. Voir [la doc](#) pour d'autres types de lancements. Ces scénarios gèrent beaucoup mieux les changements d'états des activités ainsi que les threads qui en dépendent.

8.8.9. Initialisation des tests

Il faut récupérer ou mettre à jour l'activité. On fait appel à l'annotation `@Before` :



```
@Before
public void initEachTest() {
    // récupérer l'activité
    activityScenarioRule.getScenario().onActivity(
        (a) -> activity = a
    );
}
```

Cette instruction demande l'activité en cours au scénario. `onActivity()` prend une *lambda* en paramètre. Cette lambda n'a qu'un seul paramètre, `a` qui contient l'activité, et elle est recopiée dans la variable membre.

8.8.10. Structure des tests

Chaque test est structuré classiquement, par exemple :



```
@Test public void boutonOkFermeActivite()
{
    // arrange :

    // act : appuyer sur le bouton ok
    onView(withId(R.id.ok)).perform(click());

    // assert : activité détruite
    assertThat(activity.isDestroyed(), equalTo(true));
}
```

La première instruction sert à récupérer l'activité concernée dans la variable membre.

8.8.11. Écritures dans les vues de l'activité

Toute méthode qui, même indirectement, modifie les vues de l'activité, doit être impérativement exécutée dans le *thread* de l'interface utilisateur, sinon il y aura cette exception :

Only the original thread that created a view hierarchy can touch its views

Voici comment faire, avec une *lambda* et `runOnUiThread()` :



```
// modification d'une vue
activity.runOnUiThread() -> {
    // ici on peut appeler une méthode modifiant une vue
    activity.changerQuelqueChose();
};
```

Par exemple cette méthode appelle `ui.sortie.setText("...")`; Grâce à `runOnUiThread()`, elle est exécutée dans le même *thread* que l'interface.

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

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



Figure 43: Affichage tête haute

Semaine 9

Capteurs

Le cours de cette semaine est consacré aux capteurs des smartphones, et en particulier à ceux permettant la réalité augmentée.

9.1. Réalité augmentée

9.1.1. Définition

Cela consiste à ajouter des informations à ce qu'on perçoit autour de nous : afficher des mesures sur un écran invisible, ajouter des sons positionnés en 3D autour de nous, etc.

figure 43

Il ne faut pas confondre réalité augmentée et réalité virtuelle. Cette dernière consiste à simuler un environnement 3D ; ce qu'on voit n'existe pas ou est re-créé de toutes pièces.

9.1.2. Applications

- Aide à la conduite de véhicules : avions, voitures, etc.
- Médecine : assistance à la chirurgie
- Tourisme : informations sur les lieux de visites, traduction
- Architecture : visualiser un futur bâtiment, visualiser les risques en cas de tremblement de terre
- Ateliers, usines : aider au montage de mécanismes complexes
- Cinéma : simuler des décors coûteux ou imaginaires
- Visioconférences : rendre visibles les participants à distance
- Loisirs : jeux, arts, astronomie, randonnées

9.1.3. Principes

Dessiner par dessus la vue réelle implique d'avoir :

- un écran semi-transparent ou un couple écran-caméra reproduisant la réalité visible derrière,
- un capteur de position et d'orientation 3D,
- un système de dessin capable de gérer des positions dans l'espace.

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 ni précis, ni rapides, 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. Ces permissions doivent être déclarées dans le manifeste.

Dans les premières versions d'Android, les demandes d'autorisations du fichier manifest étaient examinées seulement 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 à chaque fois qu'elles sont nécessaires. L'utilisateur peut les accepter ou les refuser, et revenir sur sa décision. Dans le cas d'un refus, l'application choisit soit de s'arrêter, soit de continuer avec moins de fonctionnalités, soit d'insister (quand elle est mal programmée).

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="..." />
    ...
</manifest>
```

```
</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"/>
```

Inversement, si vous n'avez besoin que de prendre une photo, vous n'avez pas besoin de tout ce qui concerne la caméra. Il suffit d'un Intent spécial :

```
Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);  
startActivityForResult(intent, REQUEST_IMAGE_CAPTURE);
```

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

Le plus simple des tests s'écrit ainsi :

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

La méthode `checkSelfPermission` regarde si l'autorisation a été donnée. La réponse est `PERMISSION_GRANTED` ou non.



Figure 44: Demande de droit

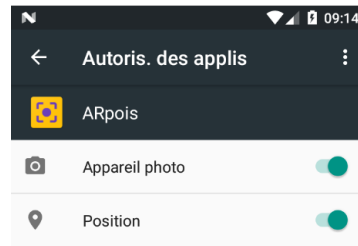


Figure 45: Préférences

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 à l’aide d’un dialogue :

figure 44

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).

figure 45

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

9.2.8. Dialogue de demande de droits

Deux étapes pour afficher ce dialogue et vérifier les droits :

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 appelle `checkSelfPermission` avant chaque action concernée.

9.2.10. Justification des demandes de 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`.

Se référer à [la documentation Google](#) pour en savoir plus.

9.3. Capteurs de position

9.3.1. Présentation

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

Voir la figure 46, page 161.

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.

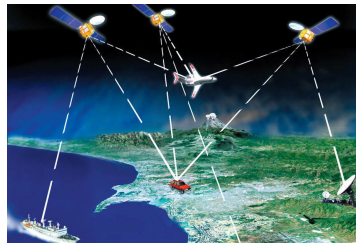


Figure 46: GPS

Dans tous les cas, on fait appel à un `LocationManager` (objet singleton) 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 change, l'écouteur est 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`.

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 d'un LocationListener

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. Elles sont à tester à chaque étape.

Le principe général est :

- Une sous-classe de `View` affiche ce que voit la caméra, en mode *preview*
- La caméra est configurée :
 - avec la même résolution que cette vue
 - avec la même orientation (portrait/paysage)

NB: ce cours présente une API dépréciée, mais qui fournit des méthodes utiles pour la réalité virtuelle qui ne sont pas dans la nouvelle. Les API pour caméra changent vraiment très souvent à cause des fabricants qui proposent chacun des évolutions.

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 7, 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 de cette manière un peu compliquée :



```
public class MainActivity extends AppCompatActivity
    implements SurfaceHolder.Callback
{
    private SurfaceView surfaceView;

    void initSurfaceView()
    {
        surfaceView = findViewById(R.layout.surface_view);
        surfaceView.getHolder().addCallback(this);
    }
}
```

On passe par une classe `SurfaceHolder` dont on appelle la méthode `addCallback`. L'activité 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 résolutions 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

Faire 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 (c'est une ancienne API).

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:
        default:                      return 0;
    }
}
```

Cette méthode sert également pour construire le bon changement de repère en réalité augmentée.

9.4.12. Associer la caméra et la vue


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.

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).

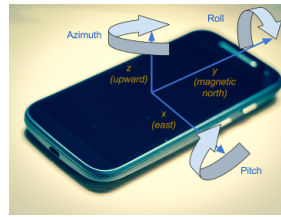


Figure 47: Angles d'Euler

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 47

- cap ou azimut (*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("no sensors !!");
    ...
}
```

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 : par exemple 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...

9.6.2. Assemblage

Il faut regrouper 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 POIs 3D sont à transformer par : $POI_{\text{écran}} = M * POI_{3d}$ avec $M = P * R_2 * R_1$. Le point $POI_{\text{écran}}$ est dessiné sur l'écran.

Pour cela, il faut connaître les coordonnées POI_{3d} 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 $POI_{\text{écran}} = M * POI_{3d}$.

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 $POI_{\text{écran}}$.

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. Cette partie est fournie toute faite dans le TP9.

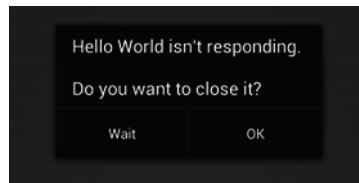


Figure 48: Application bloquée

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 48

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] );
}
```

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

Voir la figure 49, page 179.

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`.

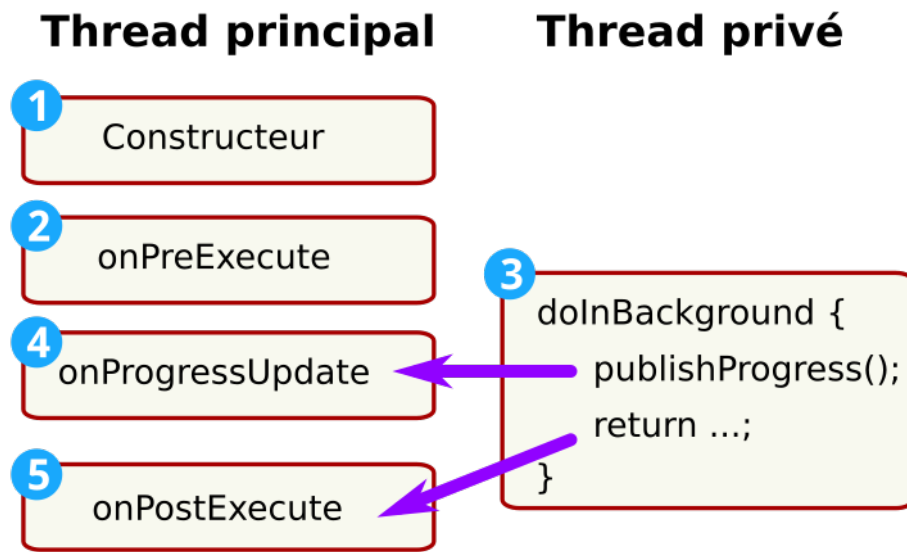



Figure 49: Méthodes d'un AsyncTask

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 50, page 182.

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.
- [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.

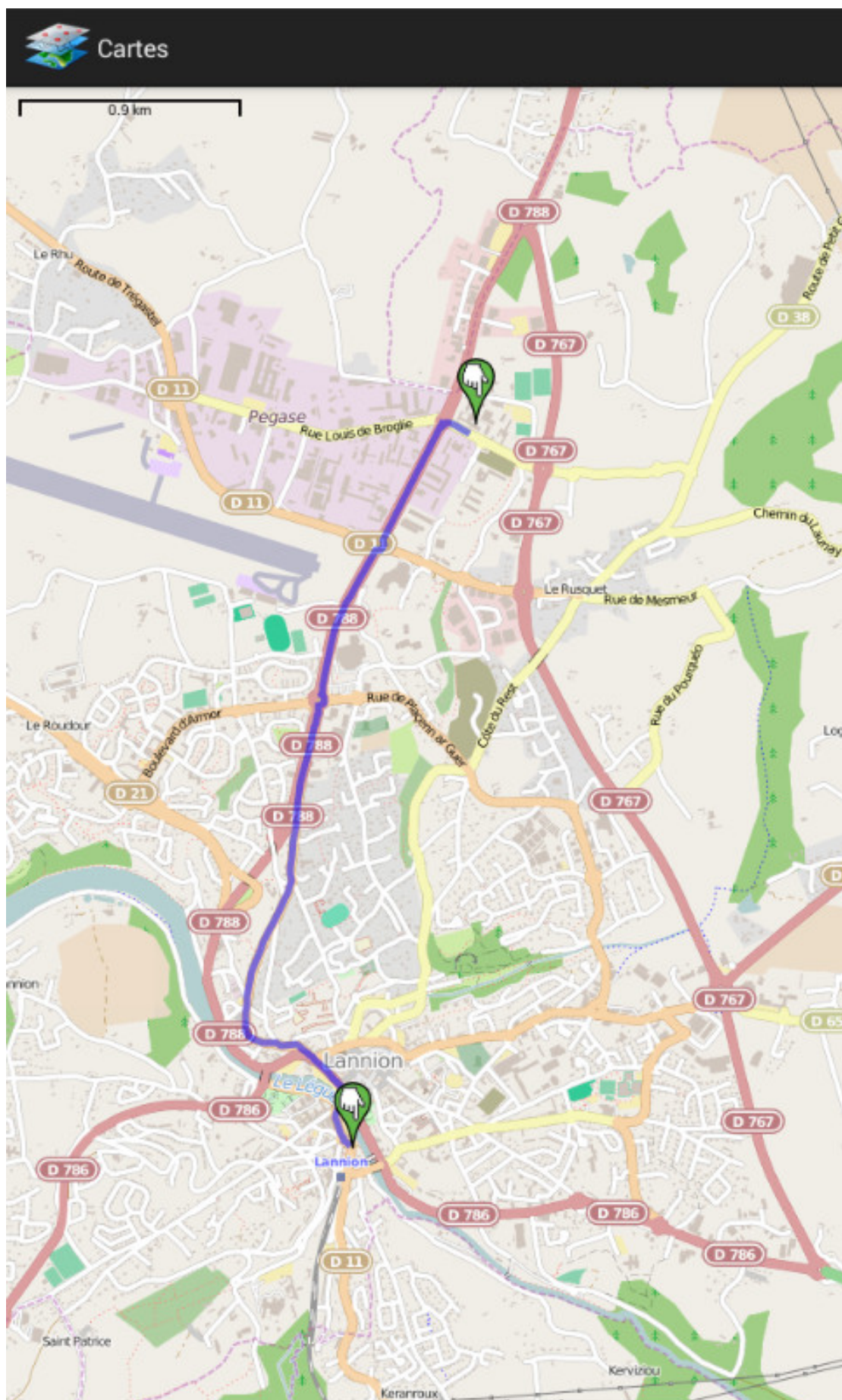


Figure 50: Google Maps

L'inclusion de bibliothèques est à la fois simple et compliqué. La complexité vient de l'intégration des bibliothèques 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.

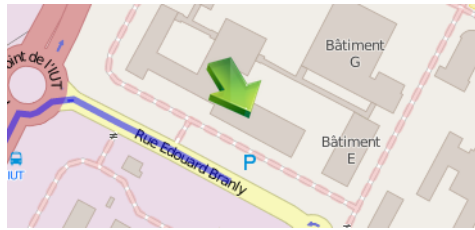


Figure 51: Marqueur personnalisé

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 51

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" />
```