SYNTHÈSE

Contenu

1. Informatique embarquée et développement mobile

L'informatique embarquée concerne le développement d'applications sur tous les supports programmables autres qu'un ordinateur.

1A. Supports et contraintes

Il existe de nombreux supports programmables :

- smartphones (essentiellement Android et iPhone) ;
- tablettes (fonctionnement similaire aux smartphones, avec les mêmes OS) ;
- PDA (Personal Digital Assistant) ;
- consoles de jeux (Xbox, PSP, Nintendo DS...) ;
- appareils électroménagers intelligents ;
- voitures ;
- domotique ;
- ...

Chaque support impose des contraintes, même si la logique de programmation est similaire :

- dispositif de sortie (lisibilité suivant la taille de l'écran, adaptabilité) ;
- dispositif d'entrée (taille de la zone tactile) ;
- rapidité (temps de réponse en minimisant les accès internet, accès rapide à l'information en minimisant les manipulations) ;
- récupération des données (gyroscope, accéléromètre, géolocalisation, photos, contacts, fichiers...).

1B. Développement pour smartphone

Il existe 3 possibilités de développement d'applications pour smartphone :

	Avantages	Inconvénients
Application web responsive	Une seule application à coder.	Accès internet obligatoire. Moins pratique. Accès aux données et capteurs du mobile plus complexe.
Application native	Pratique d'utilisation. Sans accès internet (sauf si l'application en a besoin). Accès facile aux données et capteurs du mobile.	Plusieurs applications à coder (une par type d'OS de smartphone, a priori essentiellement une pour Android et une pour iOS).
Technologie multiplate-formes	Une seule application qui va générer le code natif pour chaque support.	Code généré souvent lourd. Parfois limité pour certaines fonctionnalités. Solution généralement payante (suivant l'utilisation).

Le développement natif est différent suivant l'OS. Les 2 OS les plus répandus sont **Android** (environ 85 % du marché) et **iOS** pour iPhone (environ 15 % du marché).

Pour développer en natif, il faut installer le **SDK** (Software Development Kit) correspondant dans l'IDE choisi. Certains IDE sont adaptés au développement mobile (Android Studio pour Android).

Il est possible de configurer un **émulateur** pour tester l'application directement sur l'ordinateur. Il est normalement aussi possible de brancher un smartphone sur l'ordinateur pour qu'il serve d'émulateur (les possibilités sont parfois plus limitées).

Le **déploiement** peut généralement se faire sur smartphone, pour contrôler comment l'application fonctionne en condition réelle.

Une application peut alors être **publiée** (sur Android Market, Apple Store...) pour être accessible à tout le monde. Pour cela, il faut suivre des étapes précises de publication. Elle peut être proposée en téléchargement gratuit ou payant. Elle peut aussi intégrer des publicités.

2. Développement Android

Rappel des différents points vus dans le guide, pour le développement d'une application Android, en Java, sous Android Studio :

2A. Préparation de l'environnement

Pour développer en Java sous Android Studio, il faut :

- installer le JDK (Java Dévelopment Kit) ;
- installer l'IDE Android Studio ;
- configurer le SDK dans l'IDE avec Android SDK Manager (en choisissant la version la plus ancienne possible pour que l'application soit compatible avec le plus de smartphones possibles, tout en prenant une version suffisante pour les fonctionnalités de l'application) ;
- créer un projet (type Empty si on désire tout construire) en choisissant Java comme langage et le niveau d'API adapté (en référence au SDK installé) ;

— créer un émulateur à l'aide de l'AVD Manager (Android Virtual Device Manager) pour tester l'application sur l'ordinateur.

2B. Création d'une interface

Chaque activity est composée :
— d'un fichier xml contenant le code des objets graphiques de l'interface ;
— d'un fichier java (classe) rattaché au fichier précédent, contenant le code qui gère les objets graphiques.

Pour gérer la présentation de l'interface graphique (le fichier xml), il y a 2 modes d'affichage :
— la partie visuelle qui permet de positionner les objets graphiques ;
— la partie code (xml) qui permet de voir le code généré.

Des modifications dans l'un de ces 2 modes modifient l'autre.

Les objets graphiques sont classés par catégories : les layouts (pouvant contenir des objets et facilitant le positionnement), les objets simples (textes, boutons, images...), les conteneurs (ascenseurs, onglets...), etc.

Chaque objet graphique qui doit être manipulé dans la classe java, doit recevoir un id.

2C. Gestion professionnelle du projet

2C1. Structurer l'application en MVC

L'application démarre forcément sur une activity. Il faut donc créer un contrôleur en singleton pour que chaque activity puisse le récupérer.

Rappel de l'architecture MVC :
— le modèle contient et gère les données manipulées ;
— la vue affiche les informations et sollicite le contrôleur en cas de besoin ;
— le contrôleur reçoit les demandes de la vue, fait appel au modèle pour récupérer les données nécessaires et les renvoie vers la vue.

2C2. Créer des tests unitaires

Les tests unitaires sont gérés avec JUnit.

Pour créer un test unitaire sur une classe : sélectionner la classe, menu "Navigate > Test > Create New Test", sélectionner les méthodes à tester, ok, écrire le code des tests dans la classe de tests.

Pour lancer les tests : clic droit sur le nom de la classe de tests > run. Le test est vert s'il passe, orange s'il échoue (en affichant le problème).

2C3. Générer la documentation technique

Placer les commentaires normalisés.

Sélectionner le package pour lequel vous voulez générer la documentation, puis menu "Tools > Generate javadoc", sélectionner le dossier de sortie dans "Output directory" et le niveau de la documentation ("protected" par défaut). Mettre le bon encodage dans "Other commande line arguments" :

```
-encoding utf8 -docencoding utf8 -charset utf8
```

La documentation est automatiquement générée sous forme de site consultable par un navigateur.

2C4. Gérer le versioning

Android Studio intègre un versioning local (VCS > Local history > Show history).

Il est possible de gérer un versioning sur un dépôt distant (GitHub, BitBucket...).

Une fois le lien fait avec le dépôt distant, il faut :

- en cas d'ajout de nouvelles classes, faire un clic droit sur "app" ou sur la classe concernée, puis "Git > Add" ;
- pour commiter et pousser : "VCS > Git > Commit Directory", ajouter un "Commit message" puis "Commit and Push".

2D. Enregistrement en local

Il est possible d'enregistrer les données en local, par sérialisation ou dans une BDD au format SQLite.

2D1. Sérialisation

La sérialisation fonctionne pour un objet issu d'une classe qui **implémente l'interface Serializable**. L'objet est alors transformé en binaire et enregistré dans un fichier directement dans le smartphone.

Il est possible de le désérialiser en le récupérant à partir du fichier binaire, au **format Object**, à ensuite transtyper dans sa classe d'origine.

Ce format de sauvegarde est léger mais ne permet aucune autre manipulation que la sérialisation et la désérialisation.

2D2. SQLite

SQlite est un SGBDR léger directement installé dans le smartphone.

Pour créer une BDD de ce format, il faut créer une classe qui **hérite de SQLiteOpenHelper**.

Il est alors possible d'écrire des requêtes de création de tables, exécutées dans la méthode redéfinie **onCreate**. Cette méthode n'est exécutée qu'une seule fois (pour ne créer qu'une fois la BDD).

Une instance de cette classe permet alors de créer un accès en lecture ou en écriture pour exécuter des requêtes SQL.

```
MySQLiteOpenHelper accesBD = new MySQLiteOpenHelper(context, nomBase, versionBase);
SQLiteDatabase bd = accesBD.getWritableDatabase();
// ou
SQLiteDatabase bd = accesBD. getReadableDatabase();
```

Dans le cas d'un accès en lecture, il faut alors déclarer un curseur en lui envoyant une requête select :

```
Cursor curseur = bd.rawQuery(req, null);
```

Puis manipuler ce curseur avec les différents ordres mis à disposition.

SQLite accepte les types suivants : TEXT, NUMERIC, INTEGER, REAL.

SQLite accepte les contraintes suivantes :

- sur une colonne : NOT NULL, CHECK, DEFAULT, AUTOINCREMENT ;
- sur plusieurs colonnes : PRIMARY KEY, UNIQUE, FOREIGN KEY.

Il est possible de visualiser le contenu de la BDD en récupérant le fichier dans l'émulateur (ouvrir "Device File Explorer" à droite, puis "data > data > come.example.coach > databases").

Le fichier téléchargé peut ensuite être ouvert avec le logiciel "DB Browser for SQLite".

2F. Enregistrement dans une BDD distante

Il est possible d'accéder à un serveur distant pour enregistrer les données dans une BDD, par exemple au format MySQL.

Pour cela, il faut :

- créer la base distante (par exemple une BDD MySQL) ;
- sur le serveur distant, coder la page PHP qui **reçoit en POST** les données **envoyées au format JSON** de l'application Android, qui décode ces données et les exploite en SQL pour gérer la BDD, et si nécessaire, qui **retourne des données au format JSON** ;
- dans Android Studio, coder une classe qui **hérite de AsyncTask** (un thread indépendant) qui force la redéfinition des méthodes doInBackground (qui envoie les informations au serveur distant et attend son retour) et onPostExecute (qui se déclenche au retour d'une information et **envoie le retour à la méthode de l'objet délégué**) ;
- dans Android Studio, coder une classe qui **implémente l'interface** contenant la méthode sollicitée au retour du serveur, envoie à la classe précédente les données à envoyer au serveur distant et son instance (ce sera **l'objet délégué**) pour récupérer les données reçues dans la méthode redéfinie.

Le format JSON est utilisé car il permet de transformer des données complexes en texte simple : seul type pouvant transiter via le réseau.

La réception provenant du serveur distant se fait forcément dans un **thread** indépendant pour éviter de bloquer l'application Android.

2G. Menu et liste "adapter"

2G1. Menu

Un menu peut se gérer par une activity principale qui permet d'accéder à d'autres activities. Pour cela, il faut :

- créer un objet de type Intent en précisant l'activity d'origine et celle qu'il faut ouvrir ;
- éventuellement préciser des paramètres (flags) ;
- utiliser la méthode startActivity avec l'objet de type Intent en paramètre.

```
Intent intent = new Intent(HistoActivity.this, MainActivity.class);
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_CLEAR_TASK);
startActivity(intent);
```

2G2. Liste "adapter"

Une liste "adapter" permet d'afficher et de gérer une liste interactive. Pour cela, il faut :

- une interface xml, liée à une classe java, pour l'affichage de la liste ;
- une interface xml indépendante pour formater une ligne avec les différents objets graphiques ;
- une classe (activity) liée à la première interface, récupérant une liste d'informations et créant un "adapter pour gérer l'affichage de la liste "adapter" ;
- une classe qui hérite de BaseAdapter pour gérer la liste "adapter" pour remplir chaque ligne avec les informations reçues et gérer les éventuels événements sur les objets graphiques de la ligne.

CNED, BP 60200, 86980 Futuroscope Chasseneuil Cedex, France

87D22FSWB1221