Comment porter une application en utilisant Multi-OS Engine, une fonctionnalité d'Intel® INDE en bêta

Par Varsha M

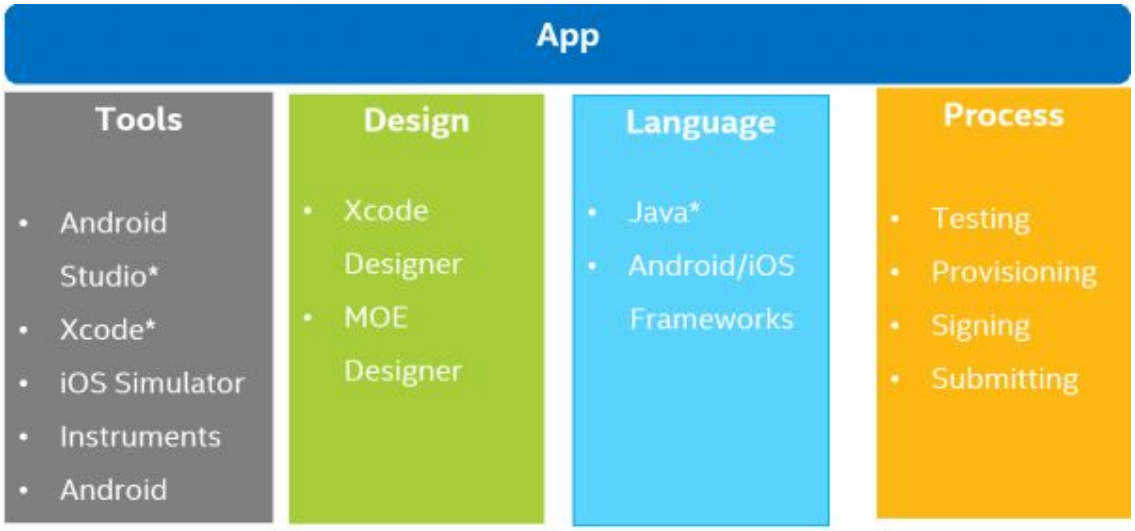
Date de publication : 2 décembre 2015

N'hésitez pas à donner votre avis sur le contenu de ce tutoriel sur le forum Android : **Commentez**

I - Vue d'ensemble

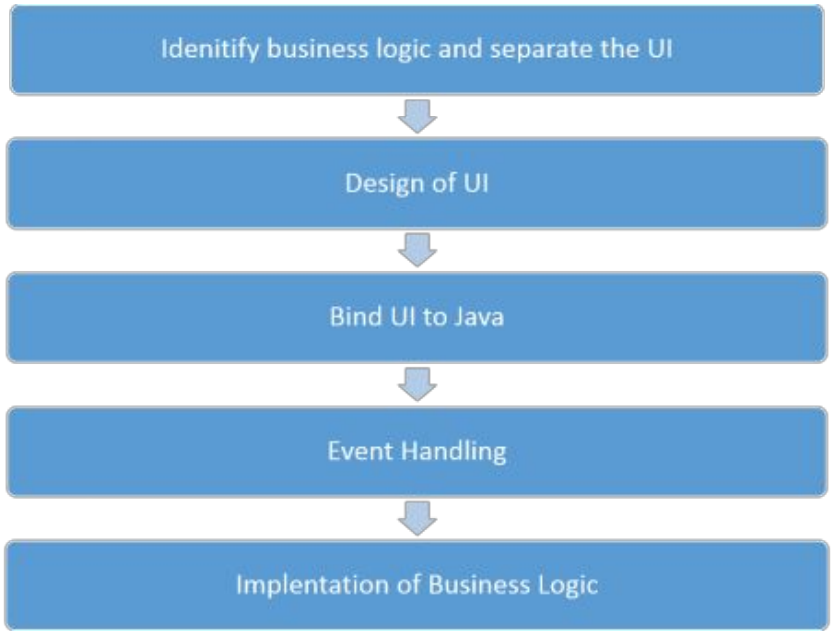
Avez-vous une application Android* existante ? Vous songez à la porter sur les plateformes iOS* ? Mais vous n'avez pas d'expérience en matière de développement iOS. Eh bien, vous n'en avez pas besoin ! Multi-OS Engine résoudra votre problème. Veuillez vous reporter à **ce lien** si vous ne connaissez pas déjà Multi-OS Engine. Pour avoir un aperçu technique de Multi-OS Engine, veuillez vous reporter à **cet article**.

Le SDK Multi-OS Engine fournit les ressources ci-dessous pour le développement d'application, de l'intégration dans l'IDE au déploiement de l'application.



Cet article vous donnera des détails sur le portage d'applications Android existantes vers iOS en utilisant Multi-OS Engine. Dans ce tutoriel nous utiliserons une machine MAC locale pour le développement de l'application. Pour un développement à distance, veuillez vous reporter au **guide de démarrage** de compilation à distance.

Le processus de portage d'une application Android existante avec Multi-OS Engine peut être divisé en plusieurs étapes.



Commençons par créer un projet avec Android Studio. Veuillez vous reporter au **guide de démarrage rapide** pour une compilation en local et des instructions pour créer un projet Android et le module Multi-OS Engine.

En partant du principe que le projet et le module Engine ont été créés, nous verrons rapidement chaque étape en illustrant de fragments de code pour plus de clarté.

Identifier la logique de fonctionnement et séparer l'interface utilisateur Multi-OS.

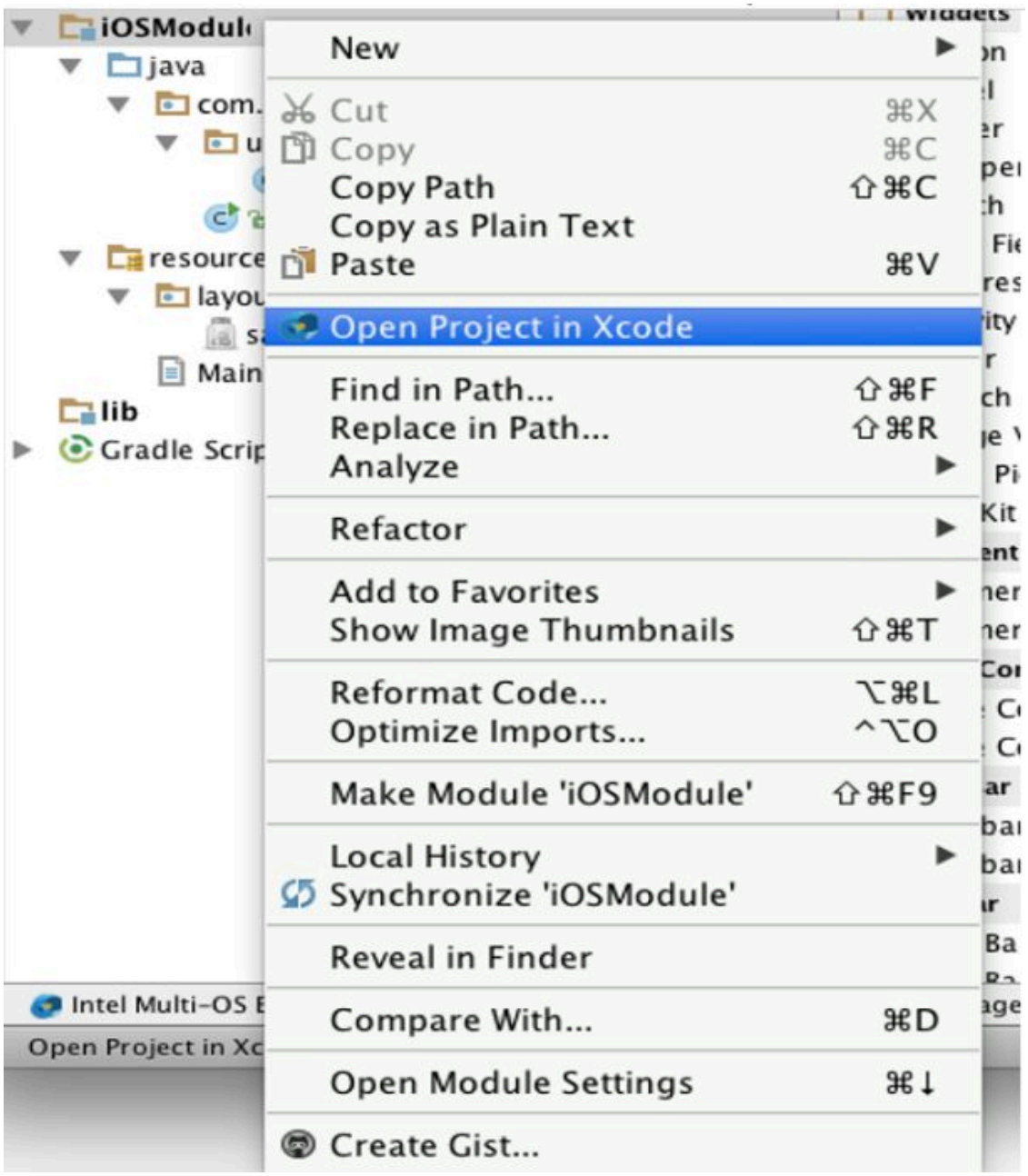
Avec Multi-OS Engine, vous pouvez réutiliser du code Java pour créer les applications Android et iOS. Multi-OS Engine vous permet de coder en Java avec des fonctionnalités communes qui peuvent être partagées entre Android et iOS.

II - Design de l'interface en utilisant Multi-OS Engine

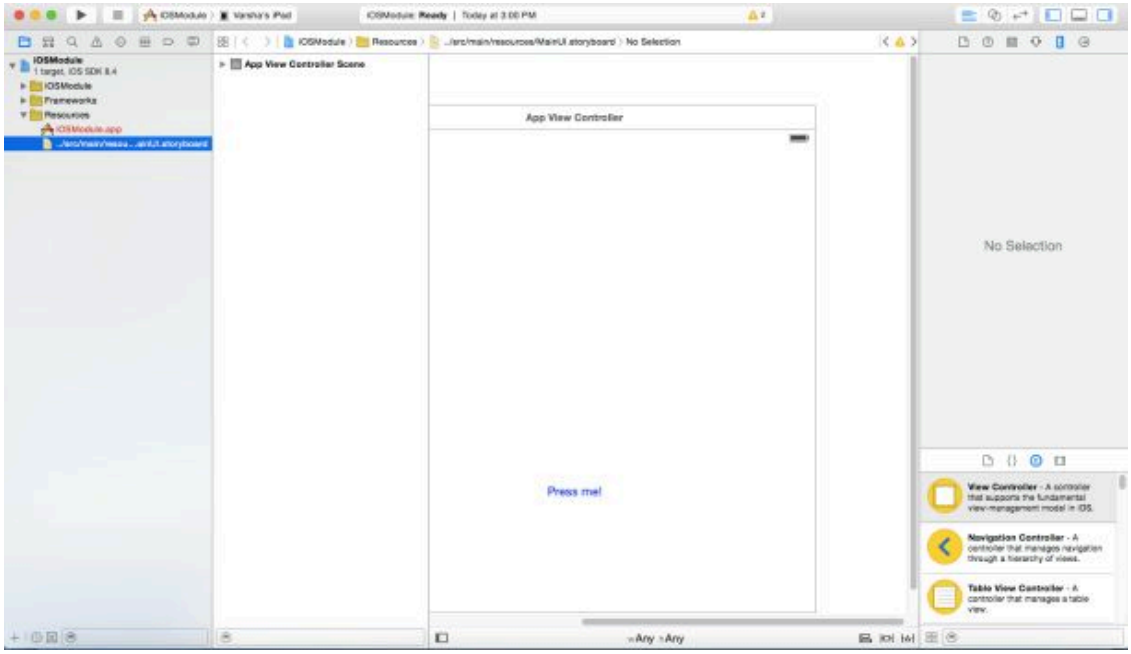
Ensuite, élaborez votre interface utilisateur iOS en utilisant l'une des deux méthodes mentionnées ci-dessous.

II-A - Design du storyboard sous Xcode

Pour utiliser le designer de Xcode, vous pouvez ouvrir votre module dans Xcode. Faites un clic droit sur votre module Multi-OS Engine, et ouvrez le projet en Xcode.



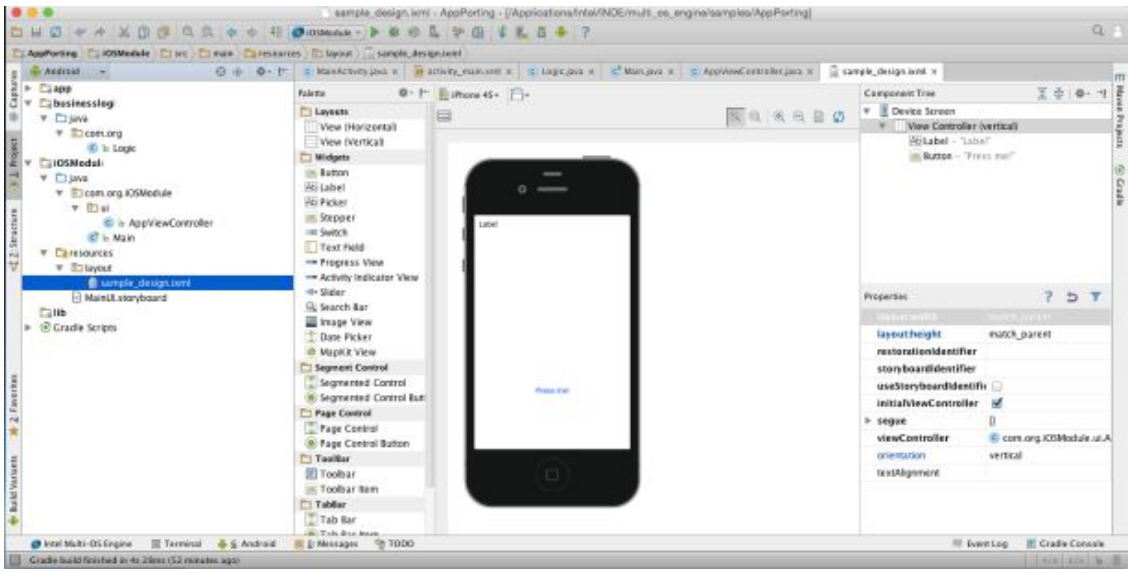
Le module s'ouvre en Xcode. Faites un clic droit sur le storyboard sous le dossier des ressources.



Une fois que le design de votre interface est prêt, vous pouvez copier le storyboard dans votre projet dans le dossier des ressources d'Android Studio.

II-B - Design sous Multi-OS Engine

L'outil de création de design est similaire à Android. Le format produit est un fichier .IXML. Plus d'informations pour l'utilisation de Multi-OS Engine sont disponibles **ici**.



III - Lier l'interface à Java

Ouvrez votre classe controller dans <nom_module> --> java --> <nom_package > → ui --> AppViewController. Dans cette classe, vous pouvez associer les liens au Layout d'UI Xcode.

Ajoutez des variables de classe à toutes les vues utilisées dans votre design dans votre fichier controller.

```
@com.intel.inde.moe.natj.general.ann.Runtime(ObjCRuntime.class)
```

```
@ObjCClassName("AppViewController")
@RegisterOnStartup
public class GameViewController extends UIViewController implements UIGestureRecognizeDelegate {

    static {
        NatJ.register();
    }

    cardCell cell = null;
    private List<Integer> emptyPoints = new ArrayList<>();


    private static final String CardCellID = "CardCell";
    static { NatJ.register(); }

    protected GameViewController(Pointer peer) { super(peer); }

    public UIButton reset = null;
    public UILabel score = null;
    @Owned
    @Selector("alloc")
    public static native GameViewController alloc();

    @Selector("init")
    public native GameViewController init();

    @Selector("release")
    public native GameViewController release();

    @Selector("itemsTable")
    @Property
    public native UICollectionView getItemsTable();

    @Selector("reset")
    @Property
    public native UIButton getReset();

    @Selector("score")
    @Property
    public native UILabel getScore();

}
```

Une fois que les variables de classe sont ajoutées, associez les accesseurs et les mutateurs pour IBActions ou IBoutlets (veuillez noter que cette étape n'est nécessaire que si vous utilisez l'outil de design de Xcode pour concevoir votre interface). Quoi qu'il en soit, l'annotation @Property ne concerne que les personnes qui n'utilisent PAS le Xcode Designer. Si vous utilisez l'outil de design de Xcode et avez des stubs natifs vous ne devez PAS utiliser l'annotation @Property.

IV - Gestion des événements

Si vous avez un gestionnaire d'événements associé à un composant, vous pouvez créer des instances des classes basées sur les événements qui se verront assigner une action afin de répondre à un événement donné.

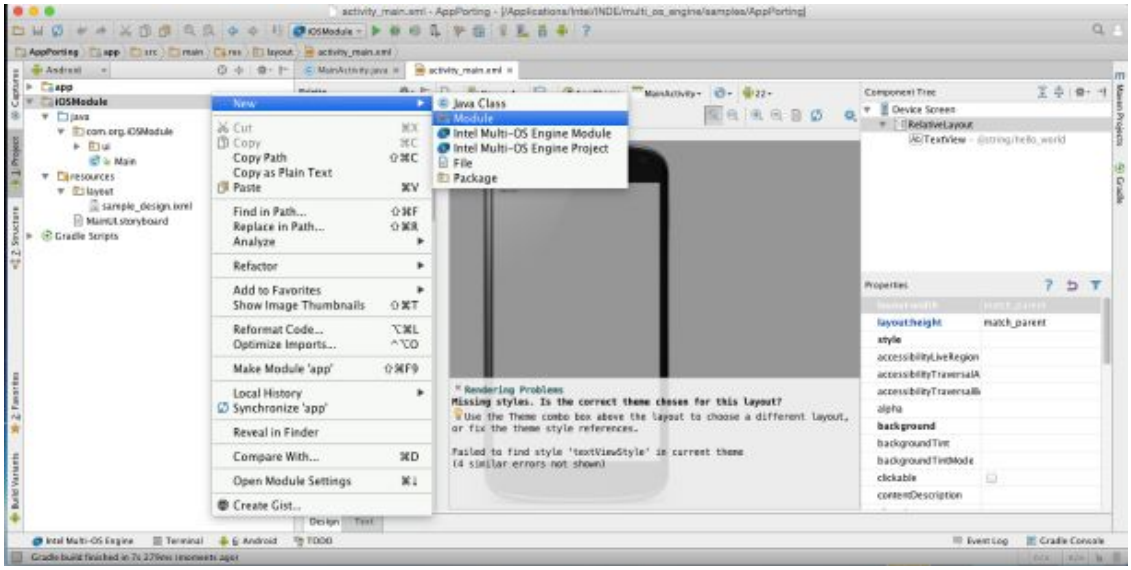
```
UISwipeGestureRecognizer swipeRight = UISwipeGestureRecognizer.alloc().
        initWithTargetAction(this, NSSelectorFromString("didSwipe:"));
swipeRight.setDirection(UISwipeGestureRecognizerDirection.Right);
swipeRight.setNumberOfTouchesRequired(1);
getItemsTable().addGestureRecognizer(swipeRight);

UISwipeGestureRecognizer swipeLeft = UISwipeGestureRecognizer.alloc().
        initWithTargetAction(this, NSSelectorFromString("didSwipe:"));
swipeLeft.setDirection(UISwipeGestureRecognizerDirection.Left);
swipeLeft.setNumberOfTouchesRequired(1);
getItemsTable().addGestureRecognizer(swipeLeft);

UISwipeGestureRecognizer swipeUp = UISwipeGestureRecognizer.alloc().
```

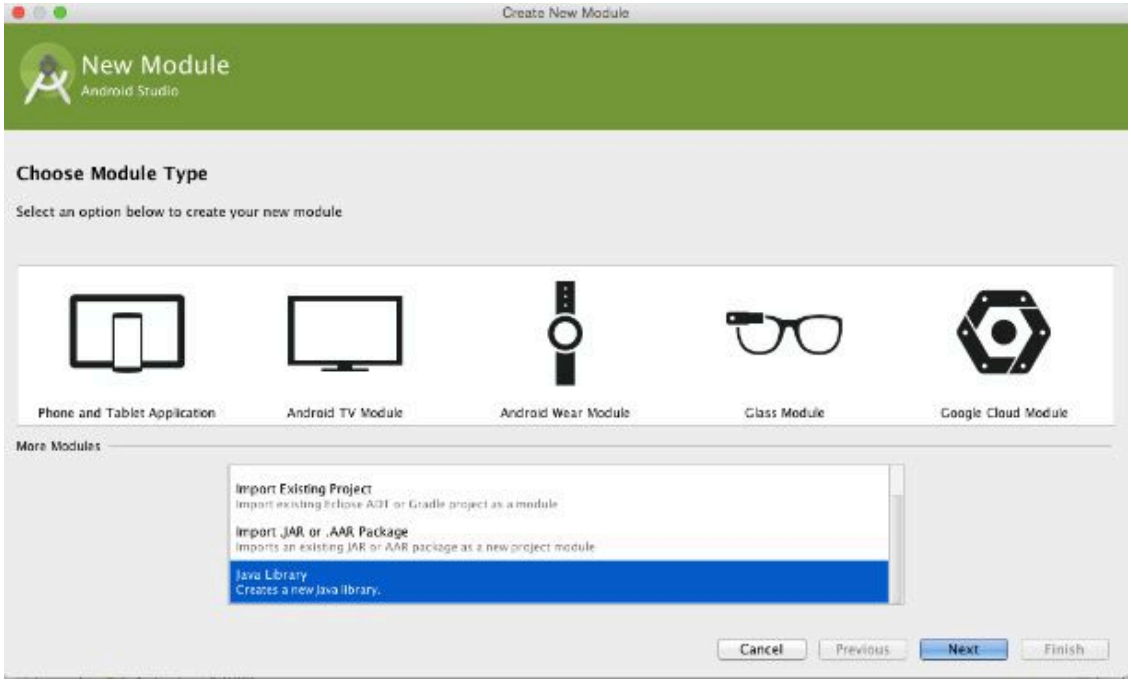
```
        initWithTargetAction(this, NSSelectorFromString("didSwipe:"));
swipeUp.setDirection(UISwipeGestureRecognizerDirection.Up);
swipeUp.setNumberOfTouchesRequired(1);
getItemsTable().addGestureRecognizer(swipeUp);

UISwipeGestureRecognizer swipeDown = UISwipeGestureRecognizer.alloc().
        initWithTargetAction(this, NSSelectorFromString("didSwipe:"));
swipeDown.setDirection(UISwipeGestureRecognizerDirection.Down);
swipeDown.setNumberOfTouchesRequired(1);
getItemsTable().addGestureRecognizer(swipeDown);
```

V - Implémentation de la logique d'application

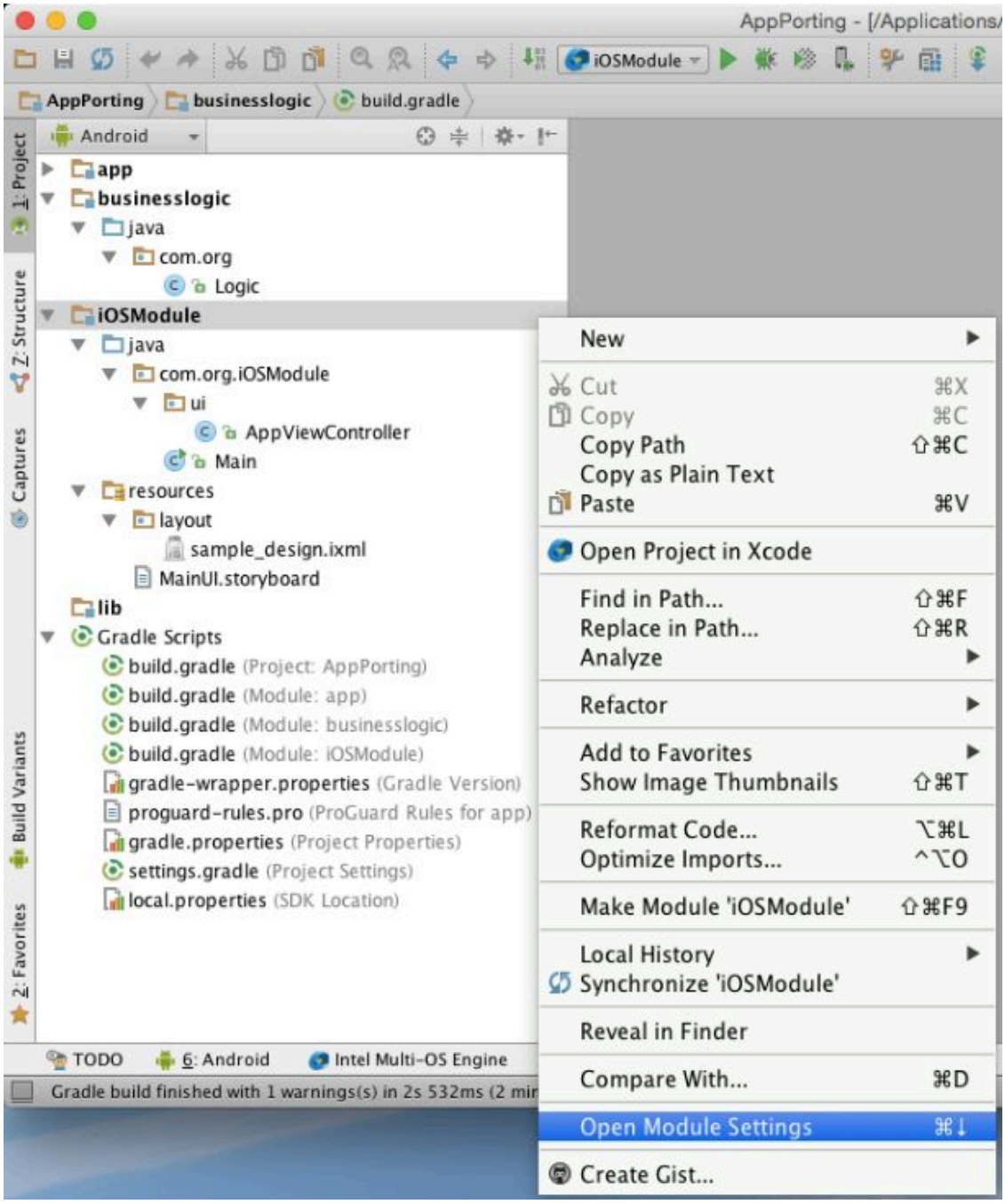
Nous créons ensuite un module séparé pour la logique d'application, qui sera en Java.

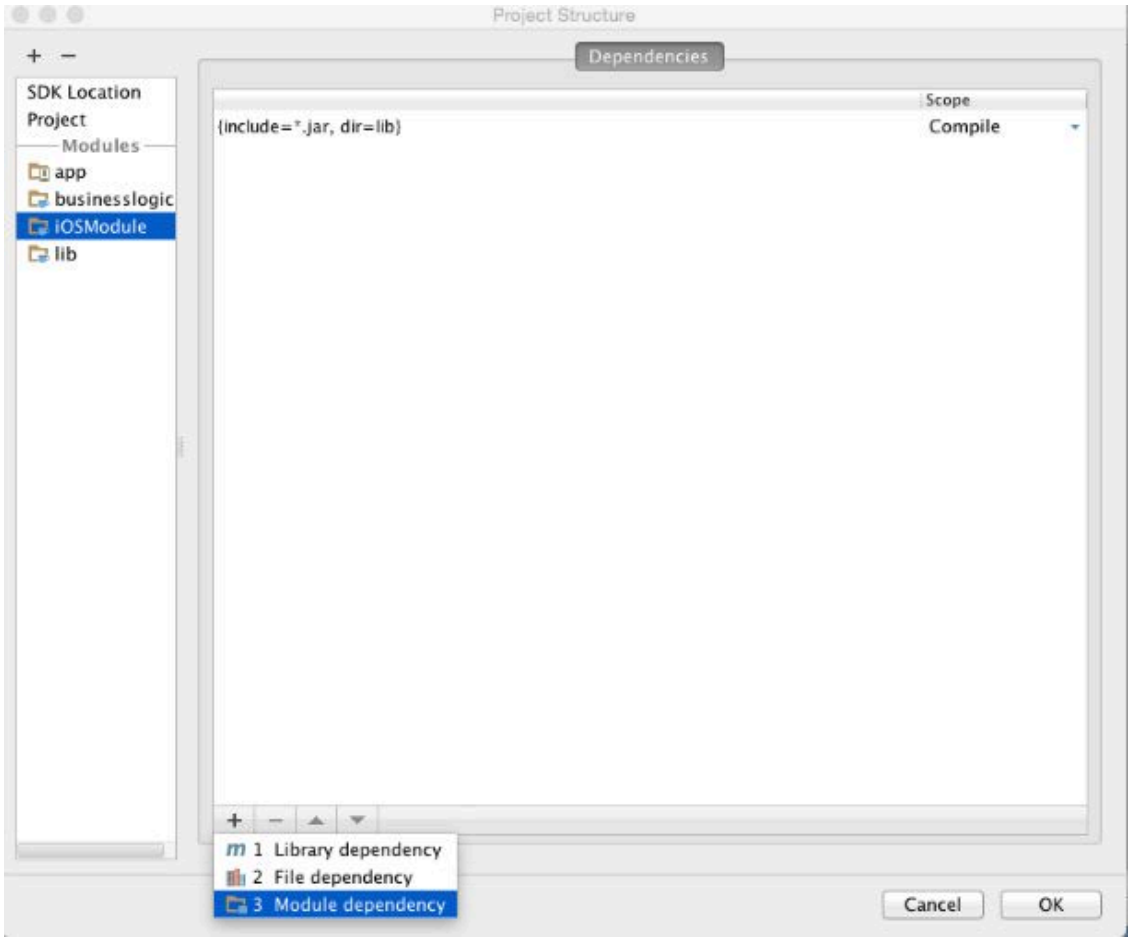
Pour créer le module, faites un clic droit sur votre projet dans l'Explorateur de Projets et créez un nouveau module. Choisissez un nouveau module de type Bibliothèque Java (« Java Library »).

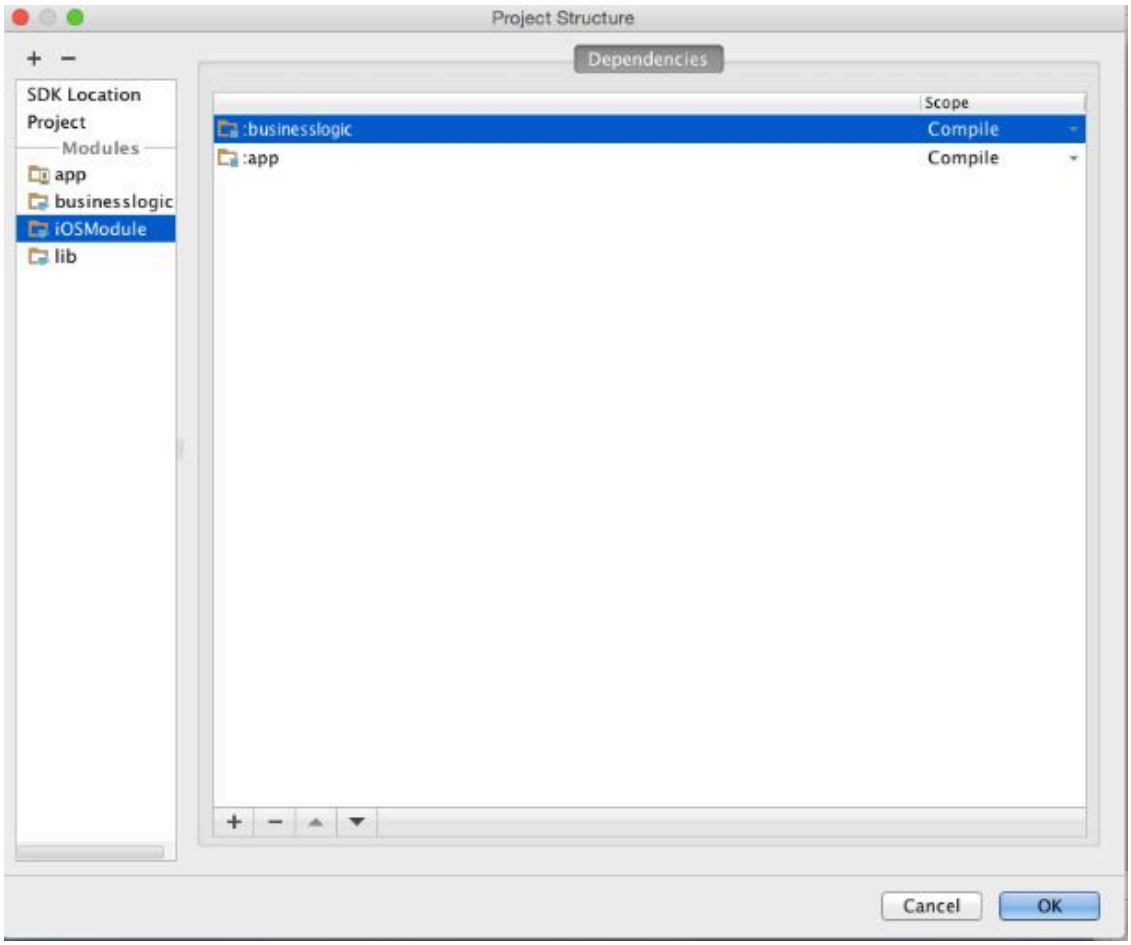




Une fois que le module est créé, il doit être accessible à la fois par le module Android et par le module iOS. Pour cela, faites un clic droit sur le projet, ouvrez les paramètres du module et ajoutez des dépendances pour le module iOS et le module Android.







Le module contenant la logique d'application est compilé en utilisant le compilateur Java. Il faut mettre à jour les paramètres de Gradle pour compiler le module. Dans les scripts Gradle, choisissez « build.gradle(Module:businesslogic) ».

- **Ajoutez le script suivant à la fin :**

```
CompileJava {
    targetCompatibility = 1.7
    sourceCompatibility = 1.7
}
```

Cela vous permet de porter votre application ! J'espère que cet article pourra vous être utile. Finalement, je vous suggère de partager votre code autant que possible entre les deux applications. Si vous avez des questions, n'hésitez pas à les poser sur notre forum Multi-OS Engine.

Pour plus d'informations sur Android et IntelR, cliquez **ici**. Vous pouvez aussi vous rendre sur le forum dédié aux **applications Android** et sur la page d'**INTEL INDE.**

Retrouvez toutes les ressources et outils Intel pour les développeurs Android sur la **Zone des Développeurs Intel Android**.

Le **forum Intel Android** pourra également vous aider dans vos questions.

VI - Ressources

- **Outils Intel Android**

- **Intel XDK**
- **Intel INDE**
- **Intel C Compiler**