Introduction

1 Principes généraux de la programmation sous Android

Android est un système d'exploitation conçu pour les terminaux mobiles, basé sur un noyau unix et une machine virtuelle Java. Pour développer des applications Android, il existe une boîte à outils (Toolkit) permettant de compiler les applications au format supporté par Android (.apk) et un émulateur. Eclipse dispose d'un plug-in permettant de lancer directement la compilation et l'exécution sur l'émulateur ou un périphérique android connecté.

Une application android est composée d'un certain nombre de fichiers :

- Dans le répertoire **src** se trouvent les fichiers sources en Java.
- Le répertoire **res/layout** contient les fichiers décrivant le style et la mise en page de l'application.
- Les répertoires **res/drawable**-(**ldpi**/**mdpi**/**hdpi**/**xhdpi**) contiennent les images à utiliser pour les différentes résolutions d'écran.
- Le répertoire **res/values** contient les définitions de constantes.
- Le répertoire **gen** contient les fichiers générés automatiquement (ne pas modifier ces fichiers manuellement !)
- **AndroidManifest.xml** décrit les propriétés de l'application (comment l'exécuter, version minimale d'Android etc...)

Un programme Android est décomposé en pages, appelées activités (Activity). La mise en page est définie dans un fichier xml, le code à exécuter est dans le fichier java correspondant.

2 Principes généraux de la programmation objet

En POO (Programmation orienté objet), la conception d'une application se fait en terme d'objets. Chaque élément concret ou abstrait est représenté en mémoire par un objet, possédant un certain nombre de propriétés (les champs) et de capacités (les méthodes). Les objets ayant les mêmes capacités et les mêmes types de propriétés (mais pas nécessairement les même valeurs) forment une classe. Les objets communiquent entre eux en appelant les méthodes pour demander l'exécution de tâches particulière.

Par exemple, dans une application comportant deux boutons et un champ texte, l'application, chacun des boutons et le champ sont tous des objets, les deux boutons étant probablement de la même classe. Un clic sur le bouton déclenchera l'exécution du code correspondant du bouton, qui pourra par exemple demander l'affichage d'un texte spécifique dans le champ texte.

Les objets en Java

3 Vocabulaire de la programmation objet.

La programmation objet permet de définir ses propres types de variables. Ces types sont nommés classes. Les objets sont les instances de ces classes.
Par exemple :

MaClasse objet; // objet est déclaré comme étant de type MaClasse.

Les objets et classes sont très similaires aux structures vues en S2. Le principal ajout concernes les fonctions, qui sont définies pour les objets ou pour les classes, et s'appellent méthodes dans le contexte de la programmation objet.

Attention :
Par convention, les noms de classe commencent par une majuscule. Quand le nom est composé, les différentes parties du nom sont mises en évidence en mettant la première lettre de chaque partie. Pour les noms de variables et méthodes, la convention diffère seulement sur la première lettre, qui doit être une minuscule. Quand ces conventions ne sont pas respectées, Eclipse le signale par un avertissement.

4 Définition d'une classe

La classe est l'élément de base d'un programme en POO chaque classe est normalement définie dans un fichier .java au nom de la classe (ex : MaClasse.java)

Chaque classe comporte une déclaration indiquant son nom, suivi d'un bloc entre accolades comportant les déclarations et définitions des champs et méthodes de la classe.

Exemple(fichier Exemple.java) :

```
class Exemple {
     int a ;
     void b(){
          a =1 ;
     }
}
```

5 Constructeur

L'instanciation d'une classe se fait en appelant une méthode spécifique, appelée constructeur. Il porte toujours le même nom que la classe, et est déclaré sans type. Le constructeur a en charge l'initialisation de l'objet, ce qui implique au minimum de donner une valeur à chaque champ.

Exemple :

```
Class Exemple {
     Entier champ ;
```

```
    Exemple(){
        champ = 0 ;
    }

    Exemple(Entier champ){
        this.champ = champ ;
    }
}
```

Instanciation d'un objet :

```
Exemple e1 = new Exemple() ; // e1.champ sera initialisé à 0
Exemple e2 = new Exemple(2) ; // e2.champ sera initialisé à 2
```

6 Héritage

L'héritage est un mécanisme qui permet de définir une classe en se basant sur une classe existante. En java, une classe ne peut hériter que d'une seule autre classe. La classe qui sert de modèle est un ancêtre de la classe en train d'être définie. La syntaxe est la suivante :

```
class Fille extends Mere{
}
```

Dans cet exemple, la classe Fille dispose automatiquement des champs et méthodes définis dans la classe Mere.

7 Redéfinition

Pour modifier une méthode héritée, on utilise un mécanisme appelé redéfinition. Dans la classe fille, on définit une méthode qui est déjà définie dans la classe mère (d'où le terme redéfinition). Redéfinir une méthode remplace la méthode héritée par la méthode redéfinie.

Exemple :

```
class Mere {
    void a(){
        System.out.println('Méthode a() de la classe mère') ;
    }
    void b(){
        System.out.println('Méthode b() de la classe mère') ;
    }
}
```

```
class Fille extends Mere {
```

```
    void b(){
        system.out.println('Méthode b() de la classe Fille') ;
    }

    public static void main(String[] arg){
        Fille fille = new Fille() ;
        fille.a() ;
        fille.b() ;
    }
}
```

L'exécution du main() donne le résultat suivant :

```
Méthode a() de la classe mère
Méthode b() de la classe fille
```

Attention, ne pas confondre redéfinition (définir à nouveau une méthode d'un ancêtre) et surcharge (définir plusieurs méthodes avec le même nom, mais une signature différente) !

8 Type déclaré, type réel

Un variable possède deux types, qui peuvent être différents :

- **Le type déclaré**, qui est type indiqué dans la déclaration de la variable. Pour connaître ce type, il suffit de trouver la déclaration de la variable, qui se trouve dans le code source. Le type déclaré est donc connu avant même de compiler ou d'exécuter le code.
- **Le type réel**, qui est le type de l'objet en mémoire désigné par la variable. Pour connaître le type réel, il peut être nécessaire de connaître le contenu de la mémoire. Ce contenu peut dépendre de choix fait par l'utilisateur, ou d'autres éléments extérieurs au programme (contenu d'un fichier ou d'une page web etc...). Le type réel peut parfois n'être connu avec certitude que lors de l'exécution du programme, et varier d'une exécution à l'autre.

Le type réel doit toujours être un type descendant du type déclaré (l'objet a toujours au moins les fonctionnalités correspondant à la déclaration).

9 Transtypage

Pour utiliser une méthode (ou un champ) d'un objet, alors que cette méthode a été déclarée uniquement dans le type réel, on utilise un mécanisme appelé transtypage. Il s'agit déclarer un objet comme étant d'un type donné. Attention, si le nouveau type n'est pas le type réel ou un de ses ancêtre, le transtypage échoue et provoque une erreur.

Exemple :

```
class Mere {
    void a(){
        System.out.println('Méthode a() de la classe mère') ;
```

```
    }
}
------
class Fille extends Mere {
    void b(){
        System.out.println('Méthode b() de la classe Fille') ;
    }

    public static void main(String[] arg){
        Mere mere = new Fille() ;
        mere.a() ;
        mere.b() ;// interdit :  mere est déclaré de type Mere,
                                  et n'a pas de méthode b
        ((Fille) mere).b() ; // transtypage :on sait que mere
                                              est de type fille
        Fille fille = mere ; //interdit : Fille n'est pas
                                                  un ancètre de Mere
        fille = (Fille) mere // transtypage correct
        fille.b() ; // pas besoin de transtypage ;
        mere = fille ; // autorisé : Mere est bien un
                                               ancetre de Fille
        if (mere instanceof Fille){ // Prudent : évite toute
                                                erreur à l'exécution.
            fille = (Fille) mere
        }
    }
}
```

10 Abstraction

Quand une méthode est déclarée, mais pas définie (c'est à dire quand on donne la signature de la méthode, mais pas son code), on dit que la méthode est abstraite. Quand une classe a au moins une méthode abstraite, on dit qu'elle est abstraite.

Les déclarations de classes et méthodes abstraites doivent être précédées du mot-clef abstract.

Exemple :

```
abstract class AbstractClass {
    abstract void abstractMethod() ; // notez l'absence de {}
}
```

Une classe abstraite ne peut pas être instanciée. Autrement dit, un objet ne peut jamais avoir comme type une classe abstraite.

Pour utiliser une classe abstraite, il faut créer une classe fille. Cette nouvelle classe héritera des méthodes non abstraites de la classe d'origine, et devra définir toutes les méthodes abstraites pour pouvoir être instanciée. Les classes abstraites sont souvent utilisées pour fournir au développeur une base, qui doit être complétée pour être utilisée, au lieu de devoir écrire toutes les méthodes.

11 Interface

Une classe complètement abstraite (c'est à dire ne comportant que des méthodes abstraites) peut être déclarée comme une interface. Une interface se comporte comme un type de données abstrait, les méthodes indiquant les capacités des types concrets qui hériteront de (on dit aussi implémenteront) cette interface.

Les règles d'héritage des interfaces sont différentes des règles classiques (héritage unique), et utilisent des mots clefs différents.

A	B	Syntaxe	Héritage multiple ?
Interface	Interface	`Interface B extends A`	autorisé
Interface	Class	`Class B implements A`	autorisé
Class	Class	`Class B extends A`	interdit

Remarques : Il est possible d'utiliser simultanément extends et implements. Une classe peut hériter d'autant d'interface que souhaité, mais d'une seule autre classe. Les interface ne peuvent hériter que d'autres interfaces (pas de limite au nombre d'interfaces parentes).

Dire qu'une classe implémente une interface revient à affirmer que cette classe dispose des fonctionnalités décrites dans l'interface au niveau de chaque méthodes.

12 Méthodes et variables de classes

Les méthodes sont définies dans les classes, mais sont exécutées sur les données de l'objet précisé (implicitement ou explicitement) lors de l'appel. Par exemple :

```
a.maMethode(1) ;
```

appelle la méthode maMéthode() de l'objet a. Si maMéthode() utilise un champ z, ce sera le champ z de l'objet a qui sera concerné (donc a.z).

Dans certains cas, il est nécessaire d'exécuter une méthode même sans disposer d'objet de la classe correspondante. Par exemple, la méthode main() est appelée avant la création d'un objet de la classe définissant la méthode. On utilise dans ce cas le modificateur static qui signifie que la méthode concerne la classe, et non une instance en particulier.

Comme une méthode static n'est pas liée à un objet en particulier, le code de la méthode ne peut pas faire référence aux champs (les champs sont définis pour chaque objet). Si l'on veut accéder à une information commune à la classe en entier, il faut déclarer un champ comme static.

Autrement dit, une méthode de classe ne peut accéder qu'aux champs de classe. Par contre, les méthodes normales peuvent accéder à tous les champs.

Attention : il faut éviter d'utiliser les méthodes et variables classe tant qu'une solution raisonnable pour s'en passer existe. Mais dans certains cas, on ne peut facilement faire autrement (par exemple, si on veut compter le nombre d'objets instanciés pour une classe donnée).

13 Visibilité

En POO, chaque objet est responsable de la cohérence de ses données (champs) et de ses comportements (méthodes). Il est donc parfois nécessaire de restreindre l'accès à ces champs et à ces méthodes pour garantir un bon comportement. L'accès à chaque champ et à chaque méthode peut ainsi être défini par un mot-clef :

- private : accessible de la classe elle-même seulement
- protected : accessible de la classe et des descendants de la classe
- public : accessible de partout

En l'absence d'un de ces mots-clefs, c'est le comportement par défaut : accessible de toutes classes définies dans le même répertoire.

En général, on donne le moins de droits possible, en particulier pour les champs. Si nécessaire, on peut toujours étendre ces droits plus tard, et cela évite que des données soient modifiées sans contrôle par la classe.

14 Accesseurs

Dans le cas d'un champ privé, l'accès ne peut se faire qu'en passant par des méthodes (généralement publiques) de la classe. Autrement dit, au lieu de modifier directement les valeur du champ, on demande à la classe de faire les modifications elle-même. Les méthodes permettant cet accès s'appellent les accesseurs (getters and setters en anglais). Ces méthodes peuvent être écrites automatiquement par éclipse.

Programmation Android

15 Principes fondamentaux

Une application Android est composée essentiellement de code (en java), de fichiers décrivant l'interface utilisateur (en xml) et d'élément graphique (.jpg, .png). Un certain nombre de fichiers sont générés automatiquement en fonction du contenu fourni par le développeur.

Une fois compilée, l'application se trouve dans un fichier .apk qui peut être installée sur un terminal Android.

16 Identifiants

Les principaux éléments d'une application Android peuvent être désigné par un identifiant unique, aussi bien dans le code java que dans le code XML. Ces identifiants sont listés dans le fichier R.java. Ce fichier étant généré automatiquement, il convient de ne jamais le modifier manuellement.

Les identifiants sont créés automatiquement pour chaque ressource (c.à.d. Chaque fichier présent dans un des répertoires de res/), et semi-automatiquement pour les éléments de mise en page.

La syntaxe pour ces identifiants est la suivante :

	.xml	.java
création	@+chemin/nom	
utilisation	@chemin/nom	R.chemin.nom

17 Mise en page

La mise en page d'une activité (un des écrans composant une application) est décrite dans un fichier xml, présent dans le répertoire layout (ou layout-land, pour la version paysage). Il est possible d'éditer directement ce fichier xml, ou de le modifier en passant par une interface graphique.

Les principales étapes dans création de l'interface sont les suivantes :

Sur papier	Schéma de l'application
	Version en fil de fer
	Arborescence
Fichier xml	
	Taille des éléments
	Centrages et justification (poids)
	Styles / contenus
	Noms des identifiants

18 Principes pour l'écriture du code

Les deux principales questions pour coder une nouvelle fonctionnalité sont :

- Que dois-je faire (quel code dois-je écrire) ?
- Quand dois-je le faire (où placer le code) ?

Pour la première question, il faut se souvenir qu'en POO, la plus grande part du travail est accomplie en demandant (par l'appel des méthodes) à d'autres objets de faire ce qu'il y a à faire. Autrement dit, la vraie question est de savoir comment répartir le travail sur les autres objets.

Pour trouver dans quelle méthode placer son code, il faut décider quel événement doit déclencher le code. Le code doit se trouver dans la méthode gérant cet événement, ou dans une méthode appelée à partir de ce code.

Il est recommandé de ne faire que des modifications conservant le fonctionnement du code, et de procéder par étapes, de manière systématique et ordonnée : par exemple ajouter une fonctionnalité avant d'essayer d'en coder une autre, et faire les tests avant de passer à la suite.

TD 1 – Classes et objets

Le code ci-dessous est un extrait de celui donné à l'adresse suivante :
http://download.oracle.com/javase/tutorial/java/javaOO/objects.html

---

```
public class CreateObjectDemo {

    public static void main(String[] args) {

        Point originOne = new Point(23, 94);
        Rectangle rectOne = new Rectangle(originOne, 100, 200);
        Rectangle rectTwo = new Rectangle(50, 100);

        System.out.println("Width of rectOne: " + rectOne.width);
        System.out.println("Height of rectOne: " + rectOne.height);
        System.out.println("Area of rectOne: " + rectOne.getArea());

        rectTwo.origin = originOne;

        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);

        rectTwo.move(40, 72);
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);
    }
}
```

---

1) Quel est le nom du fichier contenant ces lignes ?
2) Indiquez pour chaque objet rencontré :
    1. Sa classe
    2. La ligne où il est déclaré
    3. La ligne où il est instancié
3) Pour chaque classe, donner :
    1. La liste des champs, avec leur type
    2. La liste des méthodes, avec leur signature
4) Ecrire le code des classes décrites à la question 3)
5) Donner la trace et l'affichage de ce code.

TP1 Création d'une application Android

L'objectif de ce TP est de créer et de faire tourner et modifier légèrement une première application sous Android. Pour cela, nous utiliserons Eclipse, une machine virtuelle Android et le plug in ADT permettant de contrôler l'émulateur à partir d'Eclipse.

Remarque très importante:Le lancement d'une machine virtuelle est TRES lent (plusieurs minutes). Il est donc demandé de NE PAS QUITTER LA MACHINE VIRTUELLE une fois celle-ci lancée.

1 Création d'un AVD (Android Virtual Device)

A partir d'Eclipse, cliquez sur l'icone en forme de téléphone portable dans la barre d'outils, afin d'ouvrir le gestionnaire d'AVDs (Attention, il faudra refermer le gestionnaire (mais pas la machine virtuelle) avant de continuer sous eclipse).

Dans le gestionnaire, créez une nouvelle machine «telephone» avec les caractéristiques suivantes :

- Ecran HVGA, processeur ARM, carte de 512 Mo, dernière version d'Android (4.xx).

Une fois la machine créée, lancez-là (cela prend beaucoup de temps. Il est demandé de ne pas fermer la machine virtuelle entre deux utilisations. Profitez du temps de chargement pour avancer dans la lecture du sujet). Une fois la machine lancée, déverouillez-là.

2 Création d'un projet d'exemple : MultiResolution

Dans le menu File->New cliquez sur Other puis sélectionnez Android-> Android Sample project. Vérifiez que le SDK sélectionné est bien le plus récent, et choissez MultiResolution.

Dans la barre à gauche (Package), sélectionnez le nouveau projet (MultiResolution) et lancez-le (triangle blanc sur fond vert dans la barre d'outils). Confirmez qu'il s'agit bien d'une application Android. Si votre AVD est toujours ouvert, il sera utilisé pour lancer l'application. Sinon, un nouvel AVD est démarré. L'application est automatiquement chargée dans l'AVD, et démarrée (pourvu que votre AVD soit bien déjà lancé et déverrouillé). Testez l'application puis quittez-là en appuyant sur la touche 'home' de l'émulateur (en prenant bien soin de NE PAS fermer l'AVD...).

3 Détail de l'arborescence du projet

Dans l'explorateur de package (à gauche), développez l'arborescence de MultiResolution.

Vous devez trouver les répertoires et fichiers suivants :

- **src** : contient le code en java du projet
- **gen** : contient des fichiers générés automatiquement
- **bin** : contient la version compilé du projet (avec entre autre le fichier .apk)
- res : contient les fichiers nécessaires pour le projet (apppelés aussi ressources)
  - **drawable (-hdpi/-mdpi/-ldpi/-xhdpi)** : les images du projet,aux différentes définitions
  - **layout (-land)** : la composition de chaque page (activité), dans un fichier xml
  - **values** : les éléments autres que la mise en page, principalement les chaînes de caractère, dans un fichier xml
- **AndroidManifest.xml** : les caractéristiques de l'application

4 Mise en page

Référence (à lire!) :http://developer.android.com/guide/topics/ui/overview.html

Le plugin ADT propose entre autre un éditeur graphique pour les fichiers de mise en page Android.

Visualisez la mise en page pour l'orientation par défaut (portrait, dans le répertoire layout) ainsi que pour l'orientation paysage (dans le répertoire layout-land). En cliquant sur main.xml, en dessous de l'éditeur, le contenu du fichier est affiché.

1) Quels sont les éléments xml de ce fichier (en vert) ?
2) Donnez l'arborescence du fichier, c'est à dire que vous devez placer les éléments dans un arbre indiquant leur imbrication.
3) Quelles sont les unités utilisées pour placer les éléments ?
4) Quelles sont les unités utilisées pour la taille des polices ?
5) Quelles différences (dans le fichier xml) entre la version portrait et la version paysage ?

5 Code source

Ouvrez le code source qui se trouve dans le répertoire src, dans le paquetage com.example.android.com.

Le code commence par un copyright, la déclaration du paquetage et la liste des classes utilisées dans le code.

1) Quel est le nom de la classe définie dans ce fichier ?
2) Y a-t-il un héritage ? Si oui, précisez quelle classe hérite de quelle autre.
3) Donnez le nom de tous les champs et de toutes les méthodes définies dans ce fichier.

Les méthodes commençant par « on » correspondent généralement à des gestion d'évènement, c'est à dire que ces méthodes sont appelées lorsqu'un événement particulier se produit, afin de demander à un objet de la classe de réagir.

4) A votre avis, laquelle de ces méthode est appelée en premier ?

Le code suivant, extrait de la méthode onCreate, contient ce que l'on appelle une classe anonyme :

```
nextButton.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
          ...
    }
});
```

L'appel au constructeur View.OnClickListener est suivi, entre accolades, de la redéfinition de la méthode onClick(). Tout se passe comme si une classe héritant de View.OnclickListener était déclarée (dans un autre fichier), avec le code de la méthode onClick(), avant d'être instanciée en appelant son constructeur.

5) A votre avis, que fait le code de la méthode onClick() ?

6 Identifiants

Un des mécanismes essentiels de la programmation Android est l'utilisation des identifiants. Les identifiants sont affectés automatiquement à toutes les ressources et manuellement à chaque élément de mise en page, et définis dans un fichier java. Les identifiants permettent de manipuler les ressources et les élements de mise en page à partir du code java, en passant par des méthodes spécifiques. Ce sont donc les identifiants qui font le lien entre le code, les ressources, la mise ne page et le style.

1) Quel fichier java, généré automatiquement, contient les définitions des identifiants ?
2) Donnez un exemple, dans le code java, d'identifiant décrivant :
    1. Une image
    2. Un élement de la mise en page
3) Dans un des deux fichiers main.xml, donnez un exemple de référence :
    1. Au fichier Background.9.png

    (A propos du .9, les explications se trouvent à cette page : http://developer.android.com/guide/topics/graphics/2d-graphics.html#nine-patch)

    2. A l'élément image_container
4) Dans le fichier xml, à quoi correspondent les symboles suivants :
    1. @ ?
    2. + ?
    3. / ?
5) Dans quel fichier se trouve la chaîne de caractère à laquelle fait référence l'expression suivante :`@string/next_button ?`

7 Ajout d'une image

On se propose de modifier légèrement cette application, en ajoutant une image dans la liste

1) Dans quel répertoire ajouter l'image ?

    Pour réaliser l'ajout, Il suffit pour de le faire glisser l'image de l'explorateur vers éclipse (attention à déposer dans le bon répertoire)
2) Ajouter dans le code l'identifiant de l'image, au début de la liste.
3) Vérifier si tout fonctionne.

8 Ajout d'un bouton 'previous'

On souhaite maintenant ajouter un bouton permettant de faire défiler les images dans l'ordre inverse.

En vous inspirant de ce qui est fait pour le bouton 'next', ajoutez un bouton 'previous' :

1) Ajouter le bouton dans la mise en page (attention, il y a deux fichiers à modifier). Pour cela, le plus simple est probablement de faire un copier/colle : sélectionnez le bouton, copiez-le, sélectionnez la zone contenant le bouton et collez. Ensuite, double-cliquez sur le nouveau bouton pour trouver le code xml correspondant. Modifiez l'identifiant, ainsi que la référence au texte du bouton (n'oubliez pas d'ajouter la nouvelle chaîne de caractères).

2) Ajouter le code du bouton dans la méthode onCreate(), en vous inspirant de ce qui est fait pour nextButton.

   Remarque : x=(x+1)%n permet de passer au suivant en revenant à zéro quand on arrive à n. Pour faire le contraire, il faut faire x=(x+n-1)%n.

9 Pour aller plus loin

1) Ajout d'un bouton début, à mettre entre les boutons précédent et suivant.
2) Ajout d'un bouton fin, au dessous du bouton début et toujours entre les boutons précédent et suivant
3) Ajout du texte 'image :' devant les numéros des images (ex : image : 1/10)
4) Au lieu de image, mettre une légende, différente pour chaque image. Le plus simple est de créer un tableau de légendes au début du fichier java (au dessous du tableau avec les identifiants des images)
5) Placer la légende en haut de l'image, en laissant le numéro d'image en bas.

TP 2 Mise en page d'une activité

Création de l'application

1. Créer un nouveau projet se basant sur l'exemple Android 'SkeletonApp'.
2. Dans quel fichier est défini le style de l'élément @id/back ?
3. Comment sont définies les actions associées aux clics des boutons ?

Modification de l'interface utilisateur

L'objectif est de créer une interface graphique correspondant au programme suivant :

1. Tracer l'interface en fil de fer.
2. Déduire de ce tracé l'arborescence souhaitée.
3. Avant de commencer le codage,, commentez le code en rapport avec les boutons et les menus (dans le code java), puis dans l'éditeur d'interface graphique, supprimez tous les éléments.
4. Glissez/déposer les images du disque public src (répertoire android) dans le répertoire prévu à cet effet.
5. Ensuite, composez l'arborescence graphique sans vous préoccuper pour l'instant des dimensions, centrages des éléments, contenu (à part les images : ampoules éteintes), en utilisant les classes suivantes :

- LinearLayout : conteneur . Les éléments sont ajoutés dans la direction indiquée
- ToggleButton : interrupteur
- ImageView : ampoules
- TextView : texte
- Button : pour le bouton redémarrer

6. Pour configurer le placement et le style de chaque ligne (interrupteur + ampoule), nous utiliserons des styles (afin de ne pas répéter la même configuration manuellement) . Configurez correctement une ligne (un interrupteur + une ampoule), avant de placer cette configuration dans le style du bouton, et d'appliquer ce style aux autres boutons (ou de faire un copier/coller)

Indices :

– Configurer d'abord les dimensions. Utilisez match_parent (la taille sera celle de la boîte englobante), ou wrap_content (la taille sera celle du contenu)

– Configurer ensuite les poids, pour déterminer la répartition de l'espace entre les éléments à l'intérieur d'une boîte.

1. Une fois la ligne correctement affichée, déplacer dans un style ToggleButton tous les éléments pertinents.
2. Adaptez les identifiants, définissez les textes (quand c'est nécessaire).

Action des interrupteurs

L'interface étant correctement dessinée, il s'agit maintenant de coder le comportement de chacun des éléments.

1. Créer un gestionnaire d'évènement pour les boutons. Pour l'instant, ce gestionnaire se contentera d'obtenir le premier ImageView avec la méthode getViewById(), et de changer l'image affichée en appelant la méthode setImageResource(). Ce gestionnaire d'évènement sera associé au premier interrupteur par un appel à setViewListener().

2. Afin de faciliter le traitement des interrupteurs et ampoules, placez la liste des identifiants dans deux champs (de type tableau d'entier), comme cela a été fait pour les images dans le TP 1.

3. Avec une boucle, associez le gestionnaire d'évènement précédemment créé à tous les interrupteurs (indice : utilisez le tableau des identifiants).

4. Dans le gestionnaire d'évènement, déclarez, instanciez et initialisez un tableau de booléens avec l'état des interrupteur (la méthode isChecked() retourne un booléen indiquant l'état du ToggleButton).

5. Créez un tableau de booléens de la même taille que le tableau d'ampoules, et initialisez celui-ci avec les formules suivantes (l désigne le tableau de lampes, s le tableau d'interrupteurs) :

```
l[0] =  s[0] ||  s[2];
l[1] = !s[1] || !s[0];
l[2] =  s[0] ||  s[1];
l[3] = !s[3] ||  s[2];
```

6. Ecrire une boucle sélectionnant les images de toutes ampoules pour qu'elles correspondent au tableau qui vient d'être calculé (une ampoule est allumée quand le booléen est vrai).

7. Sélectionnez le contenu du gestionnaire d'évènement, et mettez-le dans une méthode calcul() (refactor->extract method)

8. Appelez cette méthode lors de l'initialisation, afin que les bonnes lampes soient allumées dès de début.

Mise en place des autres actions

1. Définir l'action du bouton permettant de recommencer (if faut mettre tous les interrupteur sur OFF, en appelant la methode setChecked sur chacun, et passant la valeur false, et appeler ensuite (une seule fois) la méthode calcul()).

2. Ajouter un compteur d'essais :
    1. configurer la barre de progression avec la méthode setMax() (de la classe ProgressBar) en fonction du nombre d'essais autorisés ;
    2. déclarer un nouveau champ entier pour le compteur d'essais ;
    3. initialiser ce champ à 0 lors de la création et lors du redémarrage ;
    4. incrémenter ce champ dans la méthode onClick() des interrupteurs ;
    5. dans la méthode calcul(), mettre à la jour l'état de la barre de progression avec la méthode setProgress() de la classe ProgressBar.
3. Ajout des conditions pour gagner/perdre :
    1. Dans la méthode calcul(), définir un booléen pour indiquer si le joueur a gagné. Ce booléen est vrai si et seulement si toutes les lampes sont allumées.
    2. Toujours dans la méthode calcul(), changer le texte du TextView en cas de victoire (methode setText(), avec en paramètre l'identifiant de la chaîne à afficher. N'oublier de créer la chaîne dans le fichier prévu à cet effet).
    3. Dans l'action du bouton pour recommencer, réinitialiser le texte (retour au message d'aide du début)
    4. Dans la méthode calcul(), définir un booléen pour indiquer que le joueur a perdu (nombre d'essai maximum atteint sans avoir gagné).
    5. Quand le joueur a perdu, afficher un message adapté.
4. Fin de partie

Quand la partie est finie (perdue ou gagnée), désactiver les interrupteurs jusqu'au clic sur le bouton pour recommencer (méthode setEnable() de la classe ToggleButton, avec un booléen en paramètre indiquant si le bouton doit être actif ou non.)

Aide-mémoire

Algorithme	Programme en Java
Types et déclarations de variables	
Entier a,b,c	int a,b,c;
Réel a,b,c	double a,b,c;
Booléen a,b,c	boolean a,b,c;
Affectations, calculs et opérations logiques	
a <-b	a = b;
2a.b + 3(a-b)(a+b)+f(2,5)	2*a*b+3*(a-b)*(a+b)+f(2.5)
A div b+ 3(a mod b)	A / b + 3*(a % b)
(a ≤ b) et (b > c)	(a <= b) && (b >c)
(A ou b) et (non c)	(a || b) && (! c)
(x=2) et (y ≠ 3)	(x==2) && (y!=3)
Ruptures de séquence	

Si *test* alors
action1
Sinon
action2
Finsi
	if (*test*) {
	action1
	} else {
	action2
	}

Tant que *test* faire
action
Fin tant que
	while (*test*) {
	action
	}

Pour *i* de *debut* à *fin* faire
action
Fin pour
	for(*i*=*debut* ; *i*<=*fin* ; ++*i*){
	action
	}

Entier Fonction f(Réel a){
action
retourne 2
}
	static int f(double a){
	action
	return 2;
	}

Entrées / sorties	
Ecrire a,'Texte',c	System.out.println(a+''Texte''+c);
Lire n	n=console.nextInt();
(*remarque* : il faut auparavant définir console comme suit :) Scanner console = new Scanner(System.in);	

Tableau		
Déclaration	Entier[] t	int[] t ;
	Réel[][] r	double[][] r ;
Allocation	T ← nouveau Entier[10]	t = **new** int[10] ;
	R ← nouveau Réel[5][5]	R = **new** double[5][5] ;
Valeur littérale	Entier[] l ← {1,2,3}	int[] l= **new** int[]{1,2,3} ;
	Entier[][] z ← {{1,2},{3,4}}	int[][] z = **new** int[]{{1,2},{3,4}} ;
Accès à un élémént	t[1] ← t[0]+2 r[1][2] ← 3,14	t[1] = t[0]+2 ; r[1][2] = 3.14 ;
Longueur	L ← longueur(t)	L = t.length ;
Structure		

Définition	Structure Point
	  Réel x,y
	  Entier couleur
	Fin Structure
		class Point {
		  double x,y ;
		  int couleur ;
		}

Déclaration	Point p	Point p ;
Instanciation	p ← nouveau Point	p = **new** Point() ;
Initialisation des champs	p.x ← 0	p.x = 0 ;
	p.y ← 1	p.y = 1 ;
	p.couleur ← bleu	p.couleur = 0x0000FF ;
Modification	p.x ← p.y+2	p.x = p.y + 2 ;
	Point q ← p	Point q = p ;
	p ← NULL	p= **null** ;
Chaînes de caractères		
Valeur littérale	s <-'essai'	String s = ''essai'';
	c <- 'c'	char c = 'c';
Concaténation	s3 <- s1+s2+c	String s3 = s1+s2+c;
Longueur	l <- longueur(s)	int l = s.length();
Élément	c <- s[0]	char c = s.charAt(0);
Sous-chaîne	sc <- s[10 20]	String sc = s.substring(10,20+1);
Entrée	lire s	String s = console.nextLine();
Comparaison	s1 =s2	s1.compareTo(s2) == 0
	s1 < s2	s1.compareTo(s2) < 0

Structure d'un fichier java

MaClasse.java

```
//imports pour toutes les classes utilisées (bibliothèques java principalement)
import chemin.classe;   // les '/' du chemin sont remplacés par des '.'
...

//déclaration de la classe héritant de MaSuperClasse et de l'interface MonInterface
class MaClasse extends MaSuperClasse implements MonInterface  {

        //déclaration des champs
        TypeChamp nomChamp;
        ...

        //déclaration des méthodes
        MaClasse(){                                          // Constructeur
            nomChamp = valeurParDefaut;                      // Initialisation des champs
            ...
        }
        TypeRetour  nomMethode  (TypeParametre paramètre1,
                                 TypeParametre paramètre2){
                // CODE
                ...
        }
        ...
}
```

Variables

Type de variable	Déclaration	Initialisation	Portée
Variable locale	Dans un bloc de code	Dans le bloc de code	Le bloc de code
Paramètre	Dans la signature d'une méthode	Implicite, avec les valeurs passées lors de l'appel	La méthode
Champ	Dans une classe	Dans les constructeurs	Précisée avant la déclaration (*public*, *private* ...)

Méthodes

Méthode	Utilisation
Constructeur	maClasse = **new** MaClasse() // instanciation de maClasse
Autres	this.nomMethode(valeur1,valeur2) *// Dans le fichier de la classe* nomMethode(valeur1,valeur2) *// dans une méthode non **static*** objetDeTypeMaClasse.nomMethode(valeur1,valeur2) *// ailleurs*