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 densités de pixel de l'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, droits nécessaires à l'exécution du programme 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 réel 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	Classe	`Class B implements A`	autorisé
Classe	Classe	`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 des champs et à des 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 (menu source → générate getters and setters).

La forme typique d'un accesseur est la suivante :

```
class MaClasse {
     int champ;

     void setChamp(int champ){
          this.champ = champ;
     }

     int getChamp(){
          return champ;
     }
}
```

15 Collections

En java, les collections sont des classes utilisées pour gérer des regroupements d'objets. Ces classes sont paramétrées (on parle de template en anglais), c'est à dire que l'on peut préciser entre <> le type des objets gérés par la collection (un peu de la même manière que l'on précise le type de des tableaux, mais avec une syntaxe très différentes).

Les principales familles de collection sont données par les interfaces suivantes :

- Set : ensembles d'éléments tous différents (pas de possibilité d'avoir deux fois le même)
- List : Listes d'éléments, accessibles dans un ordre donné (par exemple celui dans lequel on les a ajoutés)

Pour chacune de ces interfaces, il existes plusieurs classe réelles possibles (on parle d'implémentations). Toutes ces implémentations disposent des fonctionnalités déclarées dans l'interface, mais utilisent une structure de donnée différente pour les données. Par exemple, ArrayList est une implémentation de List construite sur des tableaux.

Les principales opérations usuelles sont définies sur les collections : ajout, suppression, accès à un élément, taille de la collection et parcours des éléments...

Le parcours des éléments d'une collection peut être réalisé de deux manières différentes :

- Avec une boucle pour spéciale :

```
for(Element e : collection){
     // traitement de e
}
```

- Avec un objet de la classe Iterator (classe paramétrée elle aussi) :

```
Iterator<Element> it = collection.iterator() ;
while (it.hasNext()){
     Element e = it.next() ;
     // traitement de e
}
```

L'intérêt de la deuxième forme est que l'itérateur it dispose d'une méthode remove() qui permet de supprimer l'élément de la collection qui vient d'être consulté, alors que la première forme provoque une erreur si l'on tente de modifier la collection à l'intérieur de la boucle.

L'ensemble

Programmation Android

16 Principes fondamentaux

Une application Android est composée essentiellement de code (en java), de fichiers décrivant l'interface utilisateur (en xml) et d'éléments graphiques (.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.

17 Identifiants

Les principaux éléments d'une application Android peuvent être désignés 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

Le chemin est la partie du chemin après res/ pour les fichiers, id pour les composants définis dans des fichiers de mise en page (layout), ou le nom du fichier pour d'autres ressources nommées.
Le nom du fichier ne comporte aucune extension. Seuls les caractères alphanumériques sont acceptés dans les noms de fichiers.

18 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

19 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.

20 Création d'une activité

Pour créer une activité, il faut :

- créer le fichier layout correspondant (pour décrire la mise en page, voir plus haut)
- créer le fichier java (create class sous eclipse), en héritant de Activity
- déclarer la classe créée dans le manifeste

Dans une activité, vous devez redéfinir la méthode onCreate(). Cette méthode doit contenir l'appel à la méthode héritée (ligne commençant par super) et la construction de l'interface graphique à partir du xml avec la methode setContentView(). Dans le reste de la méthode onCreate, vous devez configurer le comportement des différentes vues (configuration des listener et adaptateurs).

21 Obtenir les composants de l'interface graphique

Les éléments de l'interface graphique sont tous des descendants de la classe View. Pour chaque élément, vous avez normalement une section dans le fichier layout en XML, de la forme suivante :

```
<NomDeClasse
     id= ''@+id/identifiant''
     …
/>
```

Le code java pour obenir une variable locale contenant votre élément est de la forme suivante :

```
NomDeClasse element = (NomDeClasse) findViewById(R.id.identifiant);
```

Une fois la variable obtenue, elle permet de configurer l'objet, par exemple en définissant le gestionnaire d'évènement.

22 Gestionnaire d'évènement

Pour définir un gestionnaire d'évènement, on peut considérer les deux cas les plus fréquents :

- L'objet est une instance d'une des classes de références (par exemple Button ou ImageView). Dans ce cas, il faut attacher le gestionnaire d'évènement avec une méthode setTrucMucheListener()
- L'objet est une instance d'une classe héritant d'une des classes de références (Activity, View). Dans ce cas, il suffit de redéfinir la méthode souhaitée, c'est à dire une méthode de la forme onTrucMuchEvent()

Dans le premier cas, il faut une instance héritant de la classe attendue par la méthode setTrucMucheListener(), généralement de la forme TrucMuchListener. Pour éviter d'avoir à ajouter un fichier pour une classe ne servant qu'une fois, on peut utiliser une classe anonyme. Pour cela, on instancie normalement la classe dont on doit hériter (ex : TrucMuchListener), et on ajoute entre accolade {} les redéfinitions de méthodes. Les classes anonymes peuvent être utilisées aussi bien pour définir la valeur d'une variable locale, comme paramètre à passer à une méthode ou comme initialisation d'un champ.

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.

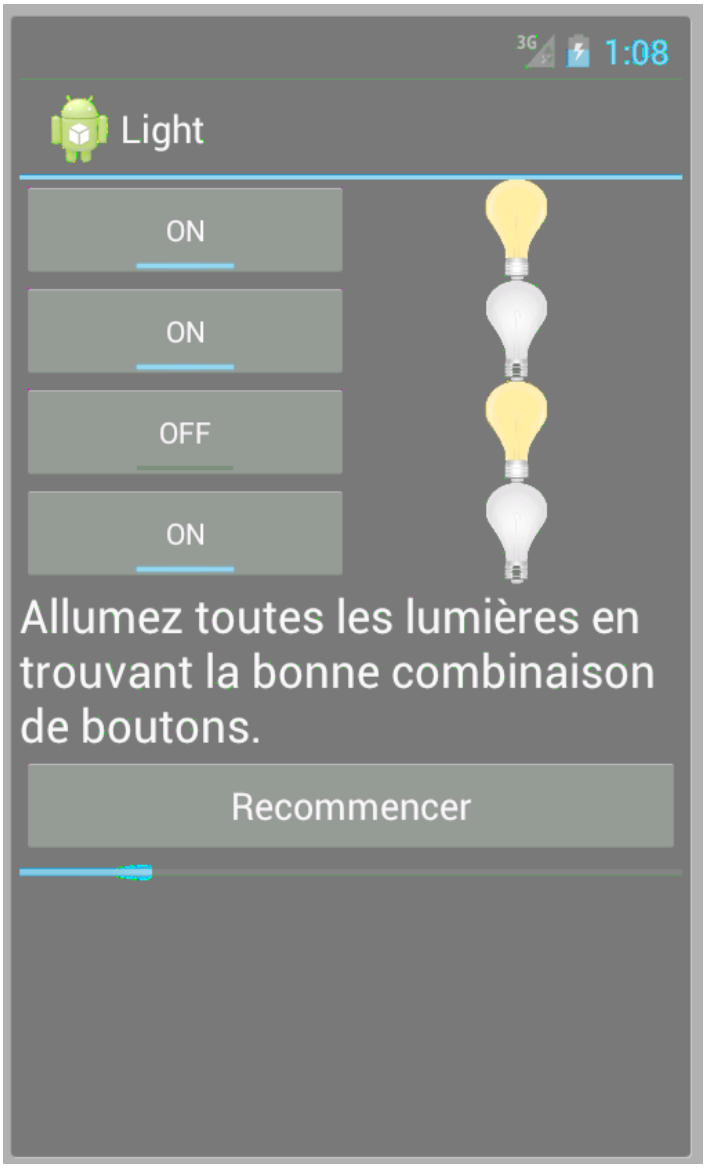
TP 2 Mise en page d'une activité

Création de l'application

1. Créer un nouveau projet se basant sur l'exemple Android 'SkeletonApp'.
2. Dans quel fichier se trouve l'élément dont l'identifiant est par @id/back ?
3. Quel est le contenu de la propriété style de l'élément de la question précédente ?
4. Dans quel fichier est défini ce style ?
5. Dans le fichier .java, 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 le contour (sous forme de boîte rectangulaire) de chaque composant graphique (ex. : bouton, image, texte...)
2. Regrouper les boîtes dans des lignes ou colonnes, c'est à dire des empilements uniquement verticaux (boîtes les unes au dessous des autres), ou verticaux (boîtes à cotés les unes des autres)
3. Donnez l'arborescence obtenue avec ces regroupements. Les classes utilisées sont les suivantes :

- LinearLayout : groupes horizontaux ou verticaux
- ToggleButton : interrupteurs
- ImageView : ampoules
- TextView : texte
- Button : pour le bouton redémarrer
- ProgressBar : barre de progression.

4. Avant de commencer le codage de l'interface, éditer le fichier java pour commentez ou supprimer tous les champs et méthodes sauf onCreate(). Dans onCreate, ne conserver que les deux premières lignes (hors commentaires). Dans l'éditeur d'interface graphique, supprimez tous les éléments.

5. Glissez/déposer les images du disque public src (répertoire android) dans le répertoire prévu à cet effet.
6. Ensuite, composez l'arborescence graphique sans vous préoccuper pour l'instant des dimensions, centrages des éléments, contenus (à part les images : ampoules éteintes).
7. 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 dimension sera celle du parent), 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 (inspirez-vous de l'exemple déjà présent).
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 findViewById(), 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 interrupteurs (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. Déplacez le contenu du gestionnaire d'évènement, et mettez-le dans une méthode calcul(). N'oubliez pas d'appeler cette méthode à partir du gestionnaire d'évènement.

8. Appelez aussi 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 méthode 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 setEnabled() de la classe ToggleButton, avec un booléen en paramètre indiquant si le bouton doit être actif ou non.)

TP3 Grille

L'objectif de ce TP est de rajouter sur le projet du premier TP une activité présentant l'ensemble des images dans une grille, sous forme de miniatures cliquables.

1. Ouvrir (ou créer) le projet MultiResolution
2. Fichiers XML

    a) grille

Créez le layout pour l'activité : Nouveau fichier -> Android -> XML Layout, en choisissant GridView comme élément de plus haut niveau, et un nom de fichier explicite, comme par exemple grid.

Affectez au GridView l'image de fond background, et un identifiant (par exemple grid).

Configurez le nombre de colonnes en automatique, la largeur des colonnes à 120dp et l'espace entre éléments à 10dp (horizontalement comme verticalement)

    b) éléments de la grille

Créez un nouveau layout pour les éléments à mettre dans la grille, avec un FrameLayout comme élément de plus haut niveau, et nommé par exemple image.

Dans le frameLayout, selectionnez image_container comme image de fond, mettez les dimensions à 120dp (horizontalement et verticalement), et insérez un ImageView d'identifiant image, de dimensions "match_parent", et avec "fit_center" comme type de mise à l'échelle

    c) AndroidManifest

Le fichier AndroidManifest.xml doit déclarer toutes les activités utilisées dans l'application, et préciser laquelle est lancée au démarrage.

Ajoutez (avec un copier/coller) une section pour l'activité GridActivity (même propriétés que Multires) et changez l'activité de démarrage pour GridActivity .

3. Définition de l'activité

Créez une nouvelle classe GridActivity héritant de la classe Activity dans le même paquetage que MultiRes.java.

Dans cette classe, surchargez onCreate() (clic droit -> source -> override/implement method -> Cochez onCreate (on peut commencer à taper le nom de la méthode recherchée pour naviguer plus vite dans la liste) et validez .

Dans le onCreate(), après l'appel à super, instanciez le layout (appel à setContentView avec l'identifiant du layout en paramètre)

Récupérez ensuite l'objet représentant le GridView dans une variable grid à l'aide de la méthode findViewById()

Nous allons maintenant configurer l'objet grid pour afficher les miniatures listées dans le tableau de MultiRes, avec le style indiqué dans le fichier xml. Cela nécessite ce que l'on appelle un adaptateur, c'est à dire une classe indiquant comment créer la vue associée à chaque élément.

Dans MultiRes, changez les modificateurs du tableau d'itendifiants de private en static (pour rendre ce tableau accessible à partir des autres classes), et le type des éléments de int vers Integer (attention, ne pas oublier de faire le même changement dans le constructeur, à la droite du égal).

En appelant la méthode setAdapter() sur l'objet grid, affectez à celui-ci une nouvelle instance de la classe ArrayAdapter. Cela nécessite de passer au constructeur les paramètres suivants, dans l'ordre :

- getBaseContext() // le contexte de l'activité
- R.layout.image // le fichier de layout à utiliser pour les éléments
- R.id.image // l'élément du fichier précédent recevant les données (ici les images)
- MultiRes.mPhotoIds // Le tableau contenant les éléments à afficher

Exécuter l'application maintenant provoque une erreur : ArrayAdapter est prévu pour afficher dans des TextView et non dans un ImageView. Il va donc falloir redéfinir la méthode réalisant l'affichage.

Pour cela, ajoutez des accolades {} juste après l'appel au constructeur (paramètres inclus).

En vous plaçant dans les accolades, utilisez le menu contextuel (source->override/implement methods) pour redéfinir la méthode getView()

Modifiez le contenu de getView() comme indiqué ci-dessous :

```
if (convertView == null) convertView =
     getLayoutInflater().inflate(R.layout.image,parent,false);
((ImageView)convertView.findViewById(R.id.image)).setImageResource(
     MultiRes.mPhotoIds[position]);
return convertView;
```

Que fait chacune des lignes précédentes ?

Lancez l'application et vérifiez que tout se passe comme souhaité

4. Ajout du gestionnaire d'évènement

Il faut maintenant récupérer le clic sur une image pour lancer la vue détaillée.

Toujours dans onCreate(), ajoutez un appel à setOnItemClickListener() sur grid avec this comme paramètre (on souhaite récupérer les évènements dans la classe GridActivity). Cliquez sur l'erreur et choisissez 'Let ... implements ...'

Cliquez sur la nouvelle erreur et choisissez 'add unimplemented methods'

La méthode onItemClick() est alors ajoutée à votre classe.

Dans cette méthode, nous allons lancer l'activité MultiRes.

Il faut pour cela créer une variable de la classe Intent qui décrira l'activité que l'on souhaite lancer.

Créez une variable intent de la classe Intent, et instanciez-là en appelant le constructeur avec les paramètres getBaseContext() et MultiRes.class

Démarrez l'activité en appelant la méthode startActivity() avec la variable intent comme paramètre.

Vérifiez que tout fonctionne .

5. Passage d'information entre activités.

Comme vous l'avez probablement constaté, l'activité lancée est toujours initialisée au début de la liste, et non sur l'élément cliqué. Pour démarrer sur la bonne image, il faut transmettre son numéro lors du lancement de l'activité. Cela se fait en passant des paramètres à l'activité par le biais de l'objet intent. Les paramètres doivent être identifiés par une clef. La clef est une chaîne de caractère commençant par le nom du paquet (ici com.example.android.multires), auquel on ajoute un nom expliquant à quoi sert le paramètre (ici .pos par exemple)

Appelez putExtra() sur intent, en passant comme paramètre la clef et le deuxième argument de la méthode onItemClick() (qui correspond à la position de l'élément cliqué dans la liste)

Dans MultiRes.java, juste avant l'appel à showPhoto(), il faut récupérer la valeur transmise. Pour cela, on appelle getIntent() qui retourne l'intent créé dans GridActivity.java, intent sur lequel on appelle getExtra() pour obtenir les données ajoutées, sur lesquelles on appelle getInt() avec la clef comme paramètre, le résultat étant placé dans le champ mCurrentPhotoIndex .

TP4 Modification d'une application

L'application "Snake", disponible dans les exemples d'application sous android, présente l'inconvénient de fonctionner au clavier.

Dans ce TP, nous allons étudier comment est construite cette application, avant dejouter des boutons permettant de jouer sans clavier.

Etude de l'application Snake

1) En consultant le fichier manifest et le fichier layout, répondez aux questions suivantes :

- Combien y at-t-il d'activités dans cette application ?
- Quels sont les noms des fichiers Java correspondant à ces activités ?
- Quelle est l'activité qui est lancée au démarrage ?
- Quelle vue (View) est utilisée pour afficher le jeu ?
- Dans quel fichier java est définie cette vue ?

2) On s'intéresse maintenant plus particulièrement au code Java de l'activité de démarrage

- De quelle classe hérite-t-elle ?

Dans la méthode onCreate() :

- Que fond les deux premières lignes ?
- Où est déclaré mSnakeView ? Quel est sa classe ?
- Quels sont les deux éléments de l'interface graphiques obtenus dans cette méthode (avec findViewbyId())? Quelles sont leurs classes respectives ?
- Quel paramètre permet de savoir si l'application vient de démarrer ou si elle est recrée d'après une sauvegarde ?
- Dans l'ensemble du onCreate(), quelles sont les méthodes de mSnakeView qui sont appelées ?

3) En consultant le fichier java correspondant à la vue affichant le jeu, répondre aux questions suivantes :

- Combien de classes différentes sont définies dans ce fichier ? combien devrait-il y en avoir ? Quelle est la classe qui correspond au nom du fichier ?

R*emarques* : Les classes définies à l'intérieur d'une autre classe sont appelées des classes internes. Il est préférable de limiter leur usage aux cas où le contenu de la classe est trop simple pour justifier un fichier java et où l'accès aux champs de la classe dans laquelle la classe interne est définie est indispensable.

- Combien y a-t-il de constructeurs (sans compter ceux des classes internes) ? Comment les différencier ? Comment faire pour ne pas avoir à copier/coller le code d'un constructeur à l'autre ?

- Où sont définies les méthodes resetTiles() et loadTile() ? (*penser à l'héritage !*)
- Les champs définis comme étant "static final" sont des constantes (on ne peut changer leur valeur). Quelle est la casse utilisée pour le nom de ces constantes ?
- Le champ mMode est utilisé pour indiquer l'état de l'objet. Quel avantage y a-t-il à votre avis à utiliser des constantes plutôt que de mettre les nombres directement ? Quels sont les états prévus ? Quelle méthode est appelée pour passer d'un état à l'autre ?

Remarque cette technique (un champ pour représenter l'état, une méthode pour changer celui-ci) est très souvent employée, et permet de faciliter la conception : on réfléchit d'abord à la liste des états, puis aux circonstances qui font changer cet état (transitions), et enfin aux conséquences de ces transitions. Dans le code de l'objet, il est très simple (un simple test sur l'état) d'adapter le comportement à la situation (par exemple, afficher ou pas un texte, réaliser une animation etc...)

- Quelles méthodes permettent de consulter ou modifier des champs ? Comment les reconnaître ? (penser à regarder les noms des méthodes)
- Quelle classe est utilisée pour déclarer les listes de coordonnées ?

Remarque : Cette classe fait partie de ce que l'on appelle en java les collections. La syntaxe avec les <> permet de préciser le type de contenu. Il existe des collections de différents types en fonction des propriétés et fonctionnalités souhaitées (par exemple : conserve l'ordre des éléments, les trie, interdit les doublons, permet de retoruver un élément avc une clef etc..)

- Quel est le nom de la méthode pour ajouter un élément à une collection ? pour obtenir l'élément à une position donnée ? pour supprimer un élément ?

Il existe une syntaxe de la boucle 'for' spécifique aux collections, pour parcourir tous les éléments. Donnez un exemple.

- Quelle méthode est appelée lorsque l'on appuie sur une touche de direction ?
- Où se trouve la méthode réalisant l'affichage proprement dit ? (pensez à l'héritage) ?

Ajout des boutons dans le layout

Ajoutez 4 boutons (pour les 4 directions) dans le layout, au dessous de la vue du jeu.

Remarque : il peut être utile de changer le groupe de plus haut niveau -- par exemple pour mettre un LinearLayout à la place du FrameLayout. Cela peut être fait simplement avec le menu contextuel (Wrap in container).

Pensez à adapter les textes et les identifiants des boutons !

Configuration des actions associées

Dans le fichier java gérant les touches, créer une méthode pour le code correspondant à chaque direction (par exemple en utilisant le menu contextuel : refactor -> extract method). Ne pas oublier de passer les méthodes créées en public (au lieu de private par défaut).

Dans le fichier configurant l'application, définir les 'OnClickListener' de chacun des boutons.

Projet (1/4) Animation d'un sprite

Le projet consiste en la réalisation d'un jeu de type Space Invader. Le travail se fait en binôme ou en trinôme, le code étant hébergé sur GitHub et partagé en utilisant Git.

Mise en place du projet

1) Préparation

L'intégrateur crée une copie du projet sur son compte Github (un fork), les autres membres créent ensuite une copie de la copie de l'intégrateur.

2) Installation sous Eclipse

Chacun copie l'adresse de son dépôt github (https:...). Sous eclipse, passer en perspective Git et coller l'addresse dans l'explorateur (colonne de gauche). Au moment de choisir le dossier utilisé pour le projet, remplacer workspace par git dans le chemin proposé.

Une fois le dépôt local créer, le sélectionner et faire « importer projet » (disponible avec un clic droit). Choisir le « wizard »quand cela est proposé. Dans le "android application wizard", choisir le code existant (si nécessaire, naviguer dans le répertoire adéquat -- ~/git/SpaceInvader). Passer ensuite en perspective java. Le nouveau projet doit normalement être visible.

Penser ensuite à fermer et réouvrir le projet nouvellement créé.
Les commandes spécifiques à Git sont accessibles dans le menu contxetuel "Team"

3) Prise en main du projet.

Comment sont indiqués les répertoires gérés par git ? Quels sont les répertoires non gérés par Git ?

Activité :

Quel est le fichier Java de l'activité ?

Quel est le fichier de layout associé ?

Quelle est le principal composant (View) ?

Composant :

Quel est fichier java du composant ?

Quels sont les champs et méthodes de cette classe ?

La méthode onMeasure() est appelée pour négocier la taille du composant. Les paramètres contiennent les informations sur les contraintes imposées au composant par la hiérarchie de vues. La méthode setMeasuredDimension() doit être appelée avec les dimensions choisies, en respectant les contraintes.

Les contraintes de dimensions peuvent être de trois types : non spécifié (aucune contrainte), exacte (la taille est imposée), au plus (la taille maximale est imposée).

Dimensions :

En étudiant le code de la méthode computeSize(), indiquer quelles sont les dimensions retenues quand aucune contrainte n'est imposée. Dans quel autre cas ces dimensions sont-elles retenues ?

Pour vérifier vos hypothèses, modifier les constantes TARGET_WIDTH et TARGET_HEIGHT et tester dans l'éditeur d'interface en changeant les dimensions de la vue (par exemple, en mettant wrap_content en largeur). Penser ensuite à remettre les dimensions par défaut à des valeurs raisonnables, dans le fichier java comme dans le fichier xml.

Affichage :

Quelle est la méthode réalisant l'affichage ?

Dans cette méthode, quel est l'objet sur lequel les méthodes de dessin sont appelées ?

Quelle est la méthode qui détermine la couleur du fond ?

Quelle est la méthode qui permet de tracer un rectangle ? Comment est déterminée la couleur ?

Quelle est la méthode qui affiche du texte ? Comment est déterminée la police ?

Où est instancié l'objet paint ? Combien de fois cet objet sera-t-il instancié ? Sachant que l'instanciation d'un objet est coûteuse en temps, indiquez pourquoi l'instanciation n'est pas faite dans la méthode qui réalise l'affichage.

Classe Sprite :

La classe Sprite est conçue pour afficher une image mobile.

Quelle type est utilisé pour l'image ? pour les coordonnées ?

A quoi correspond le rectangle retourné par getBbox() ?

Est-il possible d'obtenir une instance de la classe Sprite ?

Affichage d'une image mobile (Sprite)

1) Chargement de l'image

Les images sont disponibles en tant que ressources, mais l'affichage utilise la classe Bitmap. Il faut donc charger les ressources dans des objets de la classe Bitmap. Comme cela est nécessaire pour chaque image, il convient de créer une méthode réalisant cette tâche.

Dans le projet Snake, dans la classe TileView, la méthode loadTile() permet de charger un Drawable dans une case d'un tableau de Bitmap.

Méthode loadImage() :

Modifier la signature pour prendre comme seul paramètre un identifiant de ressource (donc un entier), et pour retourner un Bitmap (au lieu de le ranger dans un tableau). Le nouveau nom de la méthode sera loadImage() .

Au début de la méthode, récupérer dans une variable de type Drawable la ressource désignée par l'identifiant passé en paramètre. Vous pouvez pour cela vous inspirer du code de SnakeView (dans la méthode initSnakeView())

Récupérer la taille intrinsèque (c'est à dire la taille de la ressource, sans mise à l'échelle) dans des variables locales x et y en appelant les bonne méthodes sur l'objet de la classe Drawable

préalablement récupéré.

Dans le reste de la méthode, utilisez le drawable à la place de tile, et x et y pour les dimensions.

A la fin de la méthode, au lieu de placer le bitmap dans le tableau, le retourner comme résultat.

Au final, la méthode loadImage() réalise les opérations suivantes :

- Obtention du drawable à partir de l'identifiant
- Définition de la taille intrinsèque du drawable dans les variables x et y.
- Création d'un bitmap aux dimensions intrinsèques, et du caneva associé
- Définition de la taille du tracé, et tracé de l'image
- Retour de l'image créée.

Créez ensuite un champ de type Bitmap, qui recevra l'image d'un alien. Ce champ sera initialisé dans la méthode init() par un appel à la méthode loadFile() en passant la ressource adéquate en paramètre.

2) Création d'un Alien

Créer une classe Alien héritant de la classe Sprite, en définissant la méthode act (qui est abstraite dans Sprite) et le constructeur. Les versions créées par défaut par eclipse sont suffisantes pour l'instant.

Créez un champ de type Alien dans la vue. Ce champ sera initialisé dans la méthode init()

Dans la méthode onDraw, demandez à l'alien de s'afficher sur le canevas (canvas).

Vérifiez l'effet produit en consultant main.xml dans l'éditeur graphique (Si rien ne s'affiche, essayez de varier les coordonnées).

3) Animation

L'animation est réalisée image par image. Autrement dit, on alterne les modifications de la scène et le dessin d'une image, comme on le ferait dans un film en stop-motion. Pour cela, il faut d'un point de vue technique gérer trois aspects :

L'affichage de la scène : comme en l'absence d'animation, il faut dessiner les éléments de la scène, en prenant garde à l'ordre dans lequel on procède (comme en peinture, ce que l'on dessine après peut cacher ce qui a été dessiné avant)

La mise à jour de la scène : il faut définir ce que l'on appelle la logique du jeux, c'est à dire comment les éléments son animés et interagissent.

La fréquence des images : pour donner une bonne impression visuelle de fluidité, il faut afficher suffisamment d'images par seconde. Pour que la réponse aux actions du joueur soit fluide, il faut que le nombre de mises à jour par seconde (et donc d'images) soit constant.

Mise en place du compte à rebours :

L'animation nécessite de modifier l'image à intervalles réguliers, comme cela était fait dans SnakeView. Le plus simple est de reprendre la même technique, en recopiant mRedrawHandler et la classe RefreshHandler à partir de la classe SnakeView, et d'adapter ce qui doit l'être.

La commande sleep() de mRedrawHandler permet de demander le déplacement des éléments animés (méthode update()) et le rafraîchissement de l'écran (méthode invalidate())

La méthode update() n'est pas définie, il faut donc la créer (par exemple en cliquant sur l'erreur et en validant la création de la méthode dans la classe SpaceInvaderView). Cette méthode coordonnera la mise à jour des positions, et est aussi chargée de relancer le compte à rebours pour planifier la prochaine mise à jour 40 ms plus tard. Pour celà, ajoutez un appel à la méthode sleep()

Dans la méthode init(), ajoutez un appel à update() pour démarrer l'animation.

La méthode invalidate() est elle déjà définie. Trouvez dans quelle classe grace à la bulle d'information.

Mise à jour (méthode act()) :

Quel est le seul objet graphique (Sprite) disponible pour l'instant ?

Ajouter dans la méthode update() un appel à la méthode act() de ce sprite, afin de lui demander de réaliser son déplacement.

Remarques :

La méthode act() sera ainsi appelée avant chaque affichage (demandé par invalidate()). Le déplacement réalisé dans act() correspondra au déplacement entre deux images, et non à l'ensemble du déplacement. L'ensemble du déplacement est constitué de tous les déplacements act() au cours du temps. C'est le compte à rebours qui permet de demander la prochaine mise à jour, et qui réalise ainsi l'équivalent d'une boucle.

Définir plusieurs fois la position d'un objet à l'intérieur de la méthode act() – par exemple en utilisant une boucle – est une erreur malheureusement très fréquente : seule la dernière position calculée dans act() est alors affichée.

La méthode act() de la classe Alien est pour l'instant vide. Ajouter le code permettant de déplacer légèrement l'alien vers la droite (donc augmenter la valeur de x de quelques pixels).

Pour obtenir un aller-retour, on utilise une variable d'état indiquant si l'on est en train de se déplacer vers la gauche ou vers la droite. Ajoutez un champ pour refléter l'état, et définissez les constantes adéquates.

Comment évolue x dans chacun des deux états ? Codez ce comportement, à l'aide d'une conditionnelle.

A quelle condition l'état doit-il changer ?

Codez ce comportement, toujours dans la méthode act().

Ajoutez un état pour le déplacement vers le bas, et un compteur pour que cet état dure pendant 10 déplacements, et modifiez le code de la méthode act() en conséquence.

Projet (2/4) Déplacement par le joueur

Mise à l'échelle de la vue

La taille de la vue n'est pas nécessairement celle souhaitée. Il est donc nécessaire d'adapter l'affichage aux dimensions réelles. Pour réaliser le changement de repère du système de coordonnées souhaité vers celui obtenu, nous allons utiliser une transformation affine qui sera codée dans une matrice.

Ajoutez un champ transform de la classe Matrix dans la vue.

Quelles sont les dimensions souhaitées ?

La méthode onSizeChanged() est appelée à chaque dimensionnement de la vue. Implémentez cette méthode pour être averti des changements de taille de la vue.

Quels paramètres donnent les dimensions réelles ?

Dans la méthode onSizeChanged(), instanciez le champ transform.

La méthode setRectToRect de la classe Matrix calcule la transformation permettant de passer d'un rectangle à un autre. Appelez cette méthode en passant comme paramètres :

- un nouveau rectangle aux dimensions souhaités
- un nouveau rectangle aux dimensions réelles
- la constante définie dans Matrix.ScaleToFit correspondant à une mise à l'échelle centrée.

La transformation étant définie, il suffit de l'appliquer lors du dessin de la vue. La méthode concat() de la classe Canvas permet d'appliquer une transformation à tous les tracés réalisés par la suite. Ajouter l'appel à concat() dans la méthode onDraw.

Pour que le texte reste centré, il faut maintenant le placer en utilisant les coordonées fictives au lieu de la dimension réelle. Modifiez donc les paramètres de l'appel à drawText() en conséquence.

Ajout du vaisseau

Pour ajouter le vaisseau piloté par le joueur, il faut suivre les étapes suivantes, similaires à ce qui a été fait pour Alien :

- Création d'un champ pour l'image du vaisseau
- Chargement de l'image dans le champ correspondant
- Création de la classe Ship héritant de Sprite
- Création d'un champ ship dans la vue
- Affichage dans onDraw()
- Appel à la méthode act() dans update() (même si pour l'instant cette méthode ne fait rien)

Gestion des évènements de l'écran tactile

Pour controler le vaisseau, il faut récupérer les évènements en rapport avec le clavier tactile. Cela se fait en implémentant la méthode onTouchEvent(). Pour indiquer que les évènements sont effectivement gérés par notre vue, la méthode doit retourner vrai à la fin de chaque appel.

Le paramètre event contient toutes les informations relative à l'évènement qui vient de se produire :

- event.getAction() retourne le type d'action, et doit être comparé à des constantes de MotionEvent. Les évènements qui nous intéressent pour l'instant sont ACTION_DOWN et ACTION_MOVE.
- event.getX() qui retourne la dernière abscisse connue.

Dans la méthode onTouchEvent(), modifiez l'abscisse de ship pour qu'elle corresponde à la dernière abscisse connue à chaque fois qu'un évènement de type ACTION_DOWN ou ACTION_MOVE se produit.

Le vaisseau suit-il fidèlement le point de contact sur l'écran?

Transformation inverse

Comme l'affichage se fait en se basant sur un repère qui n'est pas celui de la vue, il faut corriger les coordonnées, fournies dans le repère de la vue, en appliquant la transformation inverse de celle utilisée pour le dessin.

Ajoutez un champ pour l'inverse de transform, et initialisez-le dans onSizeChanged(). Appelez ensuite la méthode inverse() sur transform en passant le champ devant recevoir l'inverse en paramètre.

L'image d'un vecteur par une matrice se fait en utilisant un tableau de float. Dans onTouchEvent(), initialisez un tableau de float avec les coordonnées de l'évènement (soit getX(), getY()). Appelez la méthode mapPoints() de la matrice inverse sur ce tableau pour transformer les coordonnées. Le résultat de la transformation est écrit dans le tableau qui a été passé en paramètre. Utilisez la première valeur (qui correspond à la valeur de x) pour déplacer le vaisseau.

Déplacement à vitesse constante

Avec la solution actuelle, le vaisseau se déplace immédiatement à l'abscisse sélectionnée. On souhaite, pour rendre le jeu plus intéressant, avoir un déplacement avec une vitesse maximale, c'est à dire que le déplacement entre deux image doit être limité. Définir la distance maximale autorisée (par exemple 5) dans une constante.

Pour obtenir un déplacement à vitesse limitée, on va utiliser le principe suivant :

- La position du TouchEvent indique la valeur cible pour x (à placer dans un nouveau champ)

- A chaque mise à jour de la position du vaisseau (méthode act()), celui-ci se rapproche du x cible autant que possible sans dépasser la vitesse limite.

Pour coder le déplacement dans act(), le plus simple est de réfléchir de la manière suivante :

- Quelle est la plus petite valeur de x possible en fonction de la position précédente et de la vitesse maximale ? Pour quelle position cible est-elle obtenue ? (Ne pas hésiter à faire un schéma)

- Que se passe-t-il si la position cible est plus petite que la valeur minimale atteignable ?

- Même questions avec la valeur maximale, mais en regardant ce qui passe pour une cible plus grande.

- Que se passe-t-il si la cible est entre la valeur minimale et la valeur maximale ?

Déduire des questions précédentes 3 cas possibles pour la nouvelle valeur de x, et coder les tests correspondant dans la méthode act()

Comment se place le vaisseau par rapport à la zone de l'écran qui a été touchée ?

Centrage

Pour obtenir le centrage du vaisseau, il faut prendre en compte la taille de celui-ci lorsque l'on fixe la valeur cible pour x. Il est logique de confier cette tâche d'ajustement à la classe Ship, puisque c'est dans cette classe que sont accessibles les informations nécessaires (la taille de l'image en l'occurence).

Actuellement, la valeur cible est modifiée directement à partir de la vue, sans considération de la taille du sprite.

- placez le champ contenant la valeur cible pour cible pour x en privé (private).
- Cela provoque normalement une erreur dans la vue, à l'endroit ou mondifiait la valeur de ce champ. Cliquez sur l'erreur et choisissez de créer les accesseurs (getters and setters)
- Le getter ne nous intéresse pas supprimez-le.
- Dans le setter, corriger la valeur affectée en retirant la moitié de la largeur du bitmap.

Projet (3/4) Collections de Sprite

Ajout des tirs du vaisseau

Le nombre de tirs affichés à un moment donné est variable. Pour gérer ces tirs, nous utiliserons une collection, ce qui facilitera l'ajout et la suppression des éléments

1) Création de la classe Missile

Comme pour les autres sprite, un certain nombre d'étapes est requis :

- Création d'un champ pour l'image du missile
- Chargement de l'image dans le champ correspondant
- Création de la classe Ship héritant de Sprite

En plus des champs hérités de Sprite, un Missile aura deux champs pour son vecteur vitesse (vx, vy).

Ajoutez ces deux champs, ainsi que les paramètres correspondant dans le constructeur et le code nécessaire pour l'initialisation.

Entre deux images, le missile se délacera de sa vitesse. Ajouter le code modifiant les champs x et y en conséquence dans la méthode act().

2) Création de la collection

La liste de missiles sera de type ArrayList<Missile>. Créez un champ pour la liste, et instanciez-le dans le constructeur.

Dans la méthode update(), ajoutez une boucle demandant à chacun des missiles de la collection de réaliser son déplacement. (utiliser la version spécifique aux collections :

```
for(Missile m : liste){
     m.act();
}
```

Dans la méthode paint(), faites de même pour afficher tous les missiles (avec le même type de boucle que ci-dessus)

3) Ajout d'un missile dans la collection

Dans un premier temps, on considère que le fait de relever le doigt de l'écran tactile correspond à un tir (il faudra plus tard limiter la cadence de tir et/ou proposer une autre méthode pour tirer, par exemple avec un Button).

Dans le gestionnaire d'évènement adéquat, effectuez un test pour détecter que le doigt quitte l'écran (ACTION_UP) et exécuter une méthode fire(). Cette méthode aura en charge le lancement d'un missile.

Dans la méthode fire(), ajoutez un missile (méthode add de la collection) créé avec les paramètres suivant :

- bitmap du missile
- x du le vaisseau +20
- y du vaisseau - 50
- vx nulle
- vy = -3

4) Destruction des missiles

Actuellement, les missiles continuent à être gérés même après être sortis de l'écran. La collection s'agrandit indéfiniment. Pour éviter d'encombrer la mémoire inutilement, il convient de supprimer les éléments qui ne sont plus visibles.

Dans le update(), ajouter un test dans la boucle pour retirer de la collection les missile dont le champ y est négatif. .

Que se passe-t-il à l'exécution ?

Comme il est impossible de supprimer un élément pendant que l'on parcours la collection avec la boucle for, il faut utiliser ce que l'on appelle un itérateur.

- Avant le début de la boucle, déclarer la variable it de type Iterator<Missile>
- Initialiser it en appelant la méthode .iterator() sur la collection à parcourir
- Remplacez la boucle for(...) par un while(it.hasNext())
- Déclarez la variable m de type missile dans la boucle, et initialisez-là en appelant next() sur l'itérateur it.
- Remplacez l'appel à remove sur la collection un appel à remove sur l'itérateur, sans paramètre (it.remove())

Vérifiez qu'il n'y a plus d'erreurs à l'exécution avec cette approche.

Gestion des vagues alien

Dans Space Invader, les aliens se déplacement en formation. La vitesse de chacun est donc la même à chaque instant, et dépend en plus de l'ensemble des positions des membres du groupe (dans le jeu original, les aliens se déplacent d'autant plus vite que leur effectifs sont réduits, et se déplacent latéralement jusqu'à ce qu'un des leurs atteigne le bord).

Donc, au lieu de gérer les aliens individuellement (comme le vaisseau ou les missiles), il est nécessaire de les gérer par vague.

1) Création de la vague

Créer une classe Wave, ayant comme champs un bitmap et une liste d'alien. Le bitmap est initialisé en ajoutant un paramètre du constructeur, la liste à une liste contenant un seul alien.

Dans la vue, changer la classe du champ alien en Wave, et ajoutez les méthodes manquantes (sauf le constructeur...) à la classe Wave. Dans la classe Wave, compétez les méthodes en réalisant le travail demandé sur tous les aliens de la collection.

2) Déplacement homogène de la vague

Le mouvement devant être le même pour tous les aliens de la vague, il faut déplacer le calcul du déplacement de la classe Alien vers la classe Wave. Cela se fait en plusieurs étapes :

- Ajout d'un champ de type Wave dans la classe Alien. Ce champ est initialisé dans le constructeur de Alien, avec une valeur fournie en paramètre. Le champ wave permettra de consulter le résultat des calculs effectués pour déterminer le mouvement commun à toute la vague.
- Déplacement des constantes, du champ etat et du compteur de Alien vers Wave.
- Déplacement du code contenu dans la méthode act() de Alien vers celui de Wave.
- Au lieu de modifier les champs x et y dans la méthode act() de Wave, créez deux méthodes getVx() et getVy() indiquant la valeur à ajouter respectivement à x et y;
- Dans la méthode act() de la classe Alien, obtenir la vitesse horizontale et verticale avec getVx() et getVy(), et mettre à jour x et y en fonction.
- Au lieu de tester la valeur de x et de y, il est nécessaire de considérer l'espace occupé par la formation d'alien dans son ensemble. Pour cela, on va calculer le plus petit rectangle contenant tous les aliens :
  1. Déclarer et instancier une variable locale rect de type RectF au début de la méthode act().
  2. Faire une boucle sur pour sur tous les aliens de la collection
  3. Dans la boucle, appeler la méthode union sur rect, en passant comme paramètre le rectangle englobant l'alien. (C'est la méthode getBbox(), définie dans la classe Sprite, qui permet d'obtenir ce rectangle englobant).
  4. A la place de x, utiliser les champ left et right (en fonction de la situation) du rectangle.

3) Ajout d'Aliens dans la vague

A l'aide d'une boucle, insérer une ligne d'aliens, par exemple en plaçant 5 aliens, répartis tous les 80 pixels à partir du pixel 50.

Placer cette boucle à l'intérieur d'une autre boucle pour réaliser plusieurs lignes en positionnant par exemple là première ligne à 20 pixels du haut, et 3 autres 70 pixels plus bas chacune.

4) Suppression des aliens

Dans la vue, rajouter dans update() un appel à une méthode testMissile() pour chaque missile (juste après le déplacement du missile par exemple)

La méthode testMissile sera définie dans Wave, prendra un Missile comme paramètre, et retournera un booléen indiquant si le missile a touché un alien.

Dans testMissile, parcourir avec un itérateur l'ensemble des alien en testant éventuellement l'intersection de la boîte englobante du missile avec la boîte englobante de l'alien (les boîtes englobantes sont de type RectF, et obtenues en appelant la méthode getBbox() sur un Sprite. La

méthode intersect() est définie sur les RectF, prend un rectF en paramètre et retourne un booléen indiquant si il y a intersection).

En cas d'intersection, supprimer l'alien concerné et retournez vrai.

Dans update(), si l'appel à testMissile() retourne vrai, supprimez aussi le missile.

Projet (4/4) Gestion de l'état du jeu

Pour gérer les transitions entre les différentes étapes du jeu (par exemple :début, jeu, pause, gagné, perdu), le plus simple est de procéder comme dans le projet Snake, avec dans la vue un champ pour l'état actuel, des constantes pour nommer les états et méthode réalisant le changement d'état proprement dit.

Préparations

- Créez le champ pour l'état
- Créez les constantes associées à l'état.
- Ajoutez une méthode (par exemple setState) pour changer l'état.
- Dans init(), configurez la valeur initiale de l'état du jeu.
- Dans onTouchEvent(), si l'état est l'état initial ou la pause, l'appui sur l'écran doit déclencher le passage à l'état actif du jeu, par un appel à setState()
- Dans setState(), le passage à l'état actif doit effacer le texte actuellement affiché.

Transitions vers les états perdu/gagné

Dans quel cas cas le jeu est-il gagné ?

Ajouter dans la méthode update() la détection de la condition de victoire et l'appel à setState si le joueur à gagné.

Dans setState(), affichez un message adapté en cas de victoire.

Dans quelle condition la partie peut-elle être actuellement perdue ?

Comme pour la victoire, ajoutez les étapes nécessaires pour la défaite.

Ajout des tirs ennemis

Actuellement, le jeu est beaucoup trop facile. Il faut donc ajouter des tirs ennemis pour empêcher le joueur de gagner trop facilement.

En vous inspirant de ce qui a été fait pour les tirs du joueur, créez une collection pour les tirs ennemis.

Dans la méthode update(), appelez la méthode fire() de wave, méthode que vous créerez pour l'occasion et qui prendra la collection comme paramètre, et ajoutera un missile pour chaque alien avec une probablilité de 0,05 (par exemple en testant si Math.random() < 0.05 pour chaque alien).

Codez ensuite les règles de comportement suivantes :

- Un missile alien qui touche le vaisseau du joueur entraîne la fin de la partie
- un missile alien qui touche un missile du vaisseau entraîne la destruction du missile du vaisseau (mais pas du missile alien)
- un missile alien qui sort de l'écran est détruit.

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*** variableDeMaClasse.nomMethode(valeur1,valeur2) *// ailleurs*

Fichiers pour une activité Android

res/layout/activity.xml

```
...
    <Button
      android:id="@+id/button"
      android:text="@string/button"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:drawableLeft="@drawable/android"
      android:textSize="24sp"/>
...
```

src/example/android/projet/MyActivity.java

```
...
 @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity);

        Button button = (Button) findViewById(R.id.button);
        button.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                ...
            }
        });
    }
```

Syntaxe des identifiants

	.xml	.java
Création	@+chemin/nom	
Utilisation	@chemin/nom	R.chemin.nom

Exemple de collection

Algorithme	Programme en Java
Liste de Sprite liste	ArrayList<Sprite> liste
Liste <- nouvelle Liste de Sprite	liste = new ArrayList<Sprite>;
liste.ajouter(sprite)	Liste.add(sprite) ;
Pour tout Sprite s **dans** liste **faire** .... **Fin Pour**	**for**(Sprite s : liste){ *... // pas de modification de la liste* }
	Iterator<Sprite> it = liste.iterator() ; **while**(it.hasNext()){ Sprite s = it.next() ; *... // appel à it.remove() possible* }