**Activity**  
Est une fenêtre pour les appareils Android.  
  
**XML**Est similaire au HTML, mais on choisit le nom de nos balises.  
  
-Si on ne veut pas hardcode du texte dans les widgets de notre layout, on va dans le fichier /values/strings.xml et on créer notre texte selon la syntaxe <string name=’’string\_name’’>My String</string>. On peut ensuite l’utiliser dans le champ texte de notre widget en utilisant la syntaxe @string/string\_name.   
  
**Fichiers importants**-/manifests/AndroidManifest.xml Fichier de publication de l’application, signature de l’application  
-/java/my.package.project/MainActivity Fichier java qui contient la logique de l’application. Contient l’équivalent de notre classe main sous la forme de onCreate();.   
-/java/my.package.project(android test)/ExempleInstrumentedTest et /java/my.package.project(test)/ExempleUnitTest Permet de faire des tests junit.   
-/drawable/ C’est ici que l’on va mettre toutes nos ressources visuelles pour pouvoir les utiliser dans notre code plus tard.   
-/layout/activity\_main.xml C’est le fichier où on dessine l’application. On peut le faire visuellement ou bien avec du code.   
-/mipmap/ Dossier qui contient les icônes qui représentent notre application.   
-/values/colors.xml Contient les couleurs de notre thème. Les couleurs sont définies en hexadécimal avec le standard RGBA mais le canal alpha est représenté par la première valeur.   
-/values/strings/ Contient les différentes langues et traduction pour notre application. Endroit où on doit écrire le texte statique pour les différentes langues supportées par notre application.   
-/values/themes/ Contient un peu l’équivalent du css pour android/java.   
-/Gradle Scripts/build.gradle (Modele : project) Contient les différentes dépendances et version nécessaire pour le fonctionnement de notre projet. (Si jamais la version de material n’est pas 1.8.0, la mettre à ça et faire sync now en haut à droite)  
  
**Android Studio**Pour créer une deuxième activité dans notre projet, il faut faire un clique droit sur le package contenant notre main activity New🡪Activity🡪TypeOfActivity.   
***Layout***On utilise les fichiers xml qui sont dans /res/layout/ pour modifier le layout de notre application. Le layout par défaut est un constraint layout. Lorsque nos widgets sont présents dans un constraint layout, on le perçois par les petites boules autour de nos widgets, il faut au moins en connecter 2 aux bordures de notre écran, sinon à d’autres widgets.  
Pour avoir accès à un widget spécifique dans notre code et à ses méthodes, il nous faut vérifier qu’elle est son ID, chose que l’on on peut changer dans les détails de notre widget.   
Dans la bar au-dessus de notre layout, il est possible de changer le téléphone qui est utilisé comme étant un preview.   
Dans les attributs de nos éléments, on peut changer la largeur ainsi que la hauteur de nos éléments avec les options layout\_width et layout\_height. Ils existent deux options par défaut soit match\_parent et wrap\_content. On peut aussi mettre nos propres valeurs en mettant dp suite à un chiffre tel que 48dp. Dp signifie density independant pixel.   
Dans les attributs de nos éléments, on peut aussi changer la taille de notre texte, cette valeur est en sp tel que 18sp. Sp signifie scale independent pixel.   
Dans les attributs des éléments qui sont normalement modifiable par l’utilisateur, il existe des champs nommées editable, clickable et enabled que l’on peut mettre à false si on ne veut pas que l’utilisateur puisse le changer/interagir avec.  
Si on veut changer la couleur d’un élément en particulier, on change la valeur de background tint.   
-Common/TextView contient du texte qui ne pourra pas changer pendant que l’application tourne.   
  
-Text/Numbers contient un champ dans lequel l’utilisateur pourra rentrer de l’information. Le nom de la classe dans le code est EditText de type Number.   
  
-Text/Password contient un champ dans lequel l’utilisateur pourra rentrer son mot de passe. Le nom de la classe dans le code est EditText de type Password.   
  
-Buttons/Switch est un bouton à deux états avec lequel l’utilisateur peut interagir. Le nom de la classe dans le code est Switch.   
  
-Common/Button est un bouton sur lequel l’utilisateur peut cliquer. Le nom de la classe dans le code est Button.   
  
-Widgets/ImageView est une zone où une image sera affichée. Pour inclure nos propres images, il faut les placer dans le dossier drawable de notre projet. Le nom de la classe dans le code est ImageView.  
  
-Widgets/ProgressBar (Horizontal) est une bar de progrès horizontale. Le nom de la classe dans le code est ProgressBar. Pour changer l’épaisseur d’une progress bar, on va changer la valeur du paramètre scale Y.  
  
-Widgets/Spinner est un menu déroulant. Le nom de la classe dans le code est Spinner. Pour intégrer des options dans le Spinner il nous faut un ArrayAdapter(context, android.R.layout.simple\_list\_item\_1, arrayOfPossibilities). Ensuite on doit associer notre Spinner à notre adapter tel que spinner.setAdapter(adapter).   
  
-Widgets/ListView est un menu de bouton qui peuvent être cliqué. Le nom de la classe dans le code est ListView. Pour intégrer des options dans le ListView il nous faut un ArrayAdapter(context, android.R.layout.simple\_list\_item\_1, arrayOfPossibilities). Ensuite on doit associer notre ListView à notre adapter tel que listView.setAdapter(adapter).   
  
-Scrollview est une zone de notre application qu’on pourra faire glisser de la même manière qu’un menu déroulant. Le nom de la classe dans le code est ScrollView bien qu’on va généralement plus avoir besoin du linear layout que le scroll view contient.   
  
-RatingBar est une suite d’étoile qui représente un rating. Le nombre d’étoile est déterminer par la largeur du widget. Il est possible d’assigner une note qui contient des moitiés d’étoile, pour cette raison les méthodes pour modifier le rating ou obtenir le rating retourne des floats.   
  
-ChipGroup and Chip sont un type de bouton où un seul chip à l’intérieur d’un chip group peut être sélectionner à la fois par défaut, ce comportement peut par contre être modifié. Il est important de noter que android :gravity et android :weight ne fonctionne pas avec ces widgets. Il est aussi important d’appliquer un style à chacune de nos chips pour que leur comportement par défaut fonctionne, les styles appropriés sont style=’’@style/Widget.MaterialComponents.Chip.Action/Choice/Entry/Filter’’. Il faut aussi appliquer à notre ChipGroup les paramètres app :selectionRequired=’’true’’, app :singleSelection=’’true’’ qui font en sorte qu’une seul chip peut être sélectionné à la fois. Si on veut qu’une chip soit sélectionnée par défaut, on applique le paramètre app :checkedChip=’’@id/ourChip’’. Pour changer l’icon checked de nos chip on applique app :checkedIcon=’’@drawable/ourIcon’’. Pour changer la couleur de notre chip on applique app :chipBackgroundColor=’’@color/ourColor’’. Les noms dans le code de ces éléments sont ChipGroup et Chip respectivement.   
  
*LinearLayout/TableLayout*Est un layout où les composantes sont organisées en ligne ou en colonne. C’est similaire à du flex en web, mais genre un milliard de fois moins facile à utiliser. L’ordre de nos éléments est essentiel.   
Un table layout est similaire à un linear layout sauf qu’il n’y a pas d’orientation, elle est toujours horizontale.   
Tout le code pour l’affichage se fait dans le code xml de l’affichage de notre activité (/res/layout/nameOfActivity.xml) :  
-LinearLayout sont l’équivalent de nos div en html/css. On va donc mettre des linears layouts dans d’autres linears layouts pour faire notre positionnement.   
  
-android :layout\_width/height nous permettent de déterminer la largeur ainsi que la hauteur de widgets/layout. Les valeurs par défaut sont match\_parent qui va prendre toute la place donnée par le parent, et wrap\_content qui va prendre que l’espace nécessaire pour montrer le contenu de notre widget/layout.  
  
-android :orientation=’’horizontal/vertical’’ nous permet de changer l’orientation de notre widget/layout. Horizontal va prendre tout l’espace disponible dans la première rangé disponible. Alors que vertical va prendre tout l’espace disponible dans la première colonne disponible.   
Dans un layout, ce paramètre change si le paramètre android :layout\_weight va affecter la width ou la height des enfants du parent layout. Donc si mon layout parent a une orientation vertical, le weight affectera le paramètre layout\_height, alors que si l’orientation de mon layout\_parent a une orientation horizontal, le weight affectera le paramètre layout\_width.   
  
-android :layout\_weight=’’aDoubleBetween0And1’’ La valeur du weight représente un pourcentage de l’espace du parent que mon enfant va prendre. On doit mettre la valeur affecter par le weight à 0dp.  
La valeur affecté varie selon l’orientation du parent de notre widget. Si l’orientation de mon layout parent est vertical, alors le weight affectera le paramètre layout\_height. Si l’orientation de mon layout parent est horizontal, alors le weight affectera le paramètre layout\_width.   
  
-android :layout\_margin=’’aValuedp’’ Nous permet de changer la marge de notre widget.  
  
-android :gravity=’’multiplePossibleValues’’ Nous permet de déterminer où seront placé nos widgets par défaut à l’intérieur du parent. Est un paramètre du parent qui affecte les enfants.   
  
-android :text=’’textValue’’ Nous permet de changer le texte contenu dans le widget.  
  
-android :textSize=’’30sp’’ Nous permet de changer la taille du texte de notre widget.   
  
-android :textColor=’’@color/color\_value’’ Nous permet de changer la couleur du texte de notre widget selon une couleur de notre fichier colors.xml.   
  
-android :textStyle=’’bold/italic’’ Nous permet de changer le style de notre texte.   
  
***Thème et style***Dans /values/themes/themes.xml on peut changer notre thème ainsi que les couleurs de notre thème.Un thème s’applique à plusieurs éléments voir l’entièreté de l’application, alors qu’un style peut être indépendant d’un thème et représente l’apparence d’un élément individuel.  
On utilise maintenant le parent Theme.MaterialComponents, mais pour obtenir plus de compatibilité rétroactive on utilise Theme.AppCompat. Il permet de modifir l’apparence des composantes une à une, les effets de clic sont moins présents qu’avec Material et les couleurs sont pleines plutôt que composées de teintes en dégradé.   
  
-Par défaut, le thème de notre application est Theme.MaterialComponents.DayNight.DarkActionBar.   
  
-Theme.MaterialComponents.DayNight.NoActionBar va retirer l’action bar au haut de notre application qui montrait le nom de notre application à l’usager.  
  
Il est bonne pratique de définir les couleurs de nos composants dans le fichier colors.xml et de les appliqués dans notre fichier themes.xml au lieu de les définir individuellement pour chaque widget. Cela aura pour effet de maintenir une cohérence à travers nos activités.   
En générale les couleurs On sont pour le texte sur les composants.   
  
Pour changer la couleur ou les valeurs d’un élément dans notre thème, il nous faut le nom de l’item en question, tel que :   
-<item name=’’strokeWidth’’>12dp</item>  
-<item name=’’strokeColor’’>@color/teal\_200</item>  
-<item name="backgroundTint">@color/purple\_500</item>  
Pour trouver le nom d’un item je recommande d’aller dans le git hub suivant : <https://github.com/material-components/material-components-android/tree/master/lib/java/com/google/android/material>

On cherche à se rendre dans le dossier nameOfWidget/res/values/styles.xml  
Sinon on peut généralement utiliser le nom des paramètres qu’on retrouverait dans ActivityName.xml.   
Généralement je recommande créer un style pour nos widgets, ainsi si on pourra appliquer des paramètres données par d’autres styles et en overwrite certain avec nos propres paramètres tel que:   
<style name="chipStyle" parent="@style/Widget.MaterialComponents.Chip.Filter">  
<item name="chipBackgroundColor">@color/green\_500</item>  
<item name="chipIcon">@drawable/crochet</item>   
<item name="checkedIcon">@drawable/crochet</item>   
<item name="android:textColor">"@color/white"</item>  
<item name="android:layout\_width">100dp</item>  
<item name="android:textAlignment">center</item>   
<item name="android:layout\_height">wrap\_content</item>   
</style>  
**NOTE :** Il est important de définir nos thèmes séparément les uns des autres, on ne peut pas déclarer un thème à l’intérieur d’un autre thème. Ils peuvent être dans le même fichier à l’intérieur de la même balise <resources></resources> mais ils ne peuvent pas être imbriqué!   
**NOTE 2 :** Si on applique un style à un widget des changements généraux fait dans le style de l’app seront overwrite par le style!   
**NOTE 3 :** Il est possible de changer certaines fonctionnalités de notre application en ajoutant des commandes à l’intérieur de la section activity du manifest de notre application.   
Un exemple serait android :windowSoftInputMode=’’adjustResize’’ qui a pour conséquence d’ajuster la taille de notre activité pour qu’elle reste entièrement visible lorsqu’un clavier est ouvert.   
On peut aussi ajouter des fonctionnalités en ajoutant des commandes à l’intérieur de la section application du manifest de notre application.  
Un exemple serait android :largeHeap=’’true’’ qui a pour conséquence de permettre à l’application de s’allouer automatiquement plus de mémoire.   
  
***Gestion des évènements***Il existe 3 grands concepts :  
-La source : La composante avec laquelle l’usager interagit. Elle lance un objet évènementiel lorsqu’un évènement survient. Elle comprend des méthodes afin de s’inscrire à un écouteur (Ex : setOnClickListener, setOnTouchListener). Si on veut désinscrire un élément d’un écouteur on rappelle la méthode pour associer l’écouteur et au lieu de donner un écouteur on donne la valeur null.   
-Objet évènementiel (Event) : Il renferme des informations sur l’évènement qui vient de survenir (Ex : l’endroit, la source, etc…). En java du moins, les noms des évènements finissent toujours en Event (Ex : TouchEvent, MenuEvent, MotionEvent, etc…).   
-Écouteur(Listener) : Un objet (pas une composante) provenant d’une classe qui implémente une ou plusieurs interfaces-écouteurs. La réaction à l’évènement sera le contenu d’une méthode d’une interface-écouteur. Le même écouteur peut écouter plusieurs sources et plusieurs types d’évènements. (Exemple listeners : OnClickListener, OnLongClickListener, OnDragListener, OnTouchListener, OnHoverListener, OnKeyListener, OnAttachStateChangeListener, OnLayoutChangeListener, OnCreateContextMenuListener, OnFocusChangerListener, OnGenericMotionListener, OnSystemUiVisibilityChangeListener, etc…)  
  
Donc quand on clique sur une source, un objet évènementiel est lancé et cet objet sera capté par un écouteur qui exécutera une action.   
  
Il existe 3 étapes pour gérer un évènement :  
1. Créer un objet écouteur. Cela doit être fait dans la fonction onCreate. (La variable peut être déclaré avant la fonction, mais elle doit être initialisé dans la fonction)  
2. Il faut inscrire la/les source(s) à notre écouteur.   
3. Coder une classe interne de l’écouteur en y codant les méthodes de l’interface-écouteur qu’elle met en œuvre.   
  
**adb logcat**Nous permet d’écrire les console logs d’android sutdio dans un document texte.   
Il nous faut en premier lieu nous assurer d’ajouter adb à notre system PATH.  
Une fois cela fait, on peut exécuter la commande adb logcat > mySaveFile.txt ce qui aura pour effet de créer un fichier dans lequel les consoles logs d’android studio seront enregistrés.   
  
**Code**  
-@Overrideprotected void onCreate(Bundle savedInstanceState{super.onCreate(savedInstanceState); setContentView(R.layout.activity\_main);} Est l’équivalent de notre main pour une application Android. Est la première fonction qui sera exécuté lors du démarrage de l’application.   
  
-@Override  
protected void onStop(){super.onStop();} Nous permet d’exécuter du code lorsque l’activité est arrêté. Cela peut se produire lorsqu’on change d’application, d’activité ou si l’application est fermée. Par exemple, cela peut être pertinent lorsqu’on veut être certain de fermer la connexion à une base de données.   
  
-@Override  
protected void onStart(){super.onStart();} Nous permet d’exécuter du code lorsque l’activité est ouverte. Cette méthode est appelée la première fois que l’activité est ouverte tout comme onCreate, mais aussi lorsqu’on y retourne. Il peut être pertinent de modifier cette fonction lorsqu’on veut s’assurer qu’une base de donnée est ouverte par exemple.   
  
-MyActivity.finish() ferme l’activité. S’il n’y a qu’une seule activité dans notre application, alors on retournera au menu principale du téléphone.   
  
-findViewById(anIntId) nous retourne un élément de notre layout selon le Id donnée.   
  
-R.id.IdOfWidget nous retourne un int selon le Id qu’on avait donnée à notre widget. Souvent utilisé avec la fonction findViewById().   
  
-AWidget.setOnClickListener(AListener) nous permet d’inscrire une source à un listener/écouteur.   
  
-AWidget.setOnTouchListener(AListener) nous permet d’inscrire une source à un listener/écouteur de type onTouch.   
  
-implements View.OnClickListener nous permet d’implémenter l’interface du listener pour les actions on click. Sa méthode est public void onClick(View v){}, celle-ci doit être override. Le v représente la source dans notre layout qui a appelé la fonction.  
  
-implements View.OnTouchListener nous permet d’implémenter l’interface du listener pour les actions de toucher, tel que lorsque l’utilisateur dépose son doigt, retire son doigt, déplace son doigt sur l’écran, place son doigt hors ou à l’intérieur d’un widget associer au listener, etc…   
Ce listener possède une méthode obligatoire qu’il nous faut override tel que public boolean onTouch(View v, MotionEvent event) où le paramètre v nous retourne de l’information sur les vus associer à ce listener et event nous retourne de l’information sur l’action performer par l’utilisateur. Il est important de noter que si la fonction retourne false, aucune action mis à part ACTION\_DOWN n’appellera le listener.   
On peut obtenir la position du doigt de l’utilisateur avec (int)event.getX/Y().  
Pour savoir quel action a été performé par l’utilisateur on utilise event.getAction(). On peut comparer la valeur que cette fonction retourne avec des indexes donnés par MotionEvent.ACTION\_MOVE/ACTION\_UP/ACTION\_DOWN/etc…   
  
-implements SeekBar.OnSeekBarChangeListener nous permet d’implémenter l’interface du listener pour toutes modifications qu’une seekBar pourrait subir.   
Ce listener possède trois méthode obligatoire mais la plus importante est public void onProfressChanged(SeekBar seekBar, int progress, boolean fromUser) qui nous permet de savoir le nouveau progrès de notre seekBar une fois que celle-ci a été touché par l’utilisateur.   
  
-AWidgetThatContainsText.setText(AString) nous permet de changer le texte de notre widget.  
  
-AWidget.getText().toString() nous retourne le string présent dans notre widget. On doit faire toString suite au getText car sinon on obtient un objet Editable.   
  
-AProgressBar.setMax(int max) nous permet de changer le maximum de notre progress bar.   
  
-AProgressBar.setProgress(int progress, bool animate) nous permet de changer le progrès de notre progress bar et de déterminer si on veut que l’augmentation/diminution soit animé.   
  
  
-ARatingBar.getRating() nous retourne un float représentant le nombre d’étoile choisit par l’utilisateur.   
-ARatingBar.setRating(float newRating) nous permet de modifier le rating courant de notre rating bar.   
  
-AParent.GetChildCount() Nous retourne le nombre d’enfant directe présent à l’intérieur d’un parent. Généralement utiliser avec un layout.  
  
-AParent.GetChildAt(int index) Nous retourne un objet view qui représente un enfant. Si on connait le type original de cette objet, on peut le cast vers ledit type tel que (LinearLayout/Button/EditText)parent.GetChildAt(int index). Cette méthode remplace un findViewById quand on a plusieurs éléments du même type.   
  
-AWidget.setBackgroundColor(Color.NAMEOFCOLOR) Nous permet de changer la couleur du background d’un widget à partir de la référence dudit widget. Prend en paramètre un int que l’on peut obtenir grâce à la classe Color qui possède en attribut static plusieurs couleurs par défaut.   
  
-AWidget.setBackgroundResource(R.drawable.myDrawable) Nous permet de changer le background d’une widget et d’y mettre un fond. Prend en paramètre un int que l’on peut obtenir grâce à R.id.   
  
-objectA instanceof NameOfClass Retourne un boolean si l’objet A est un objet de la classe indiqué.   
  
-implements AdapterView.OnItemSelectedListener est utile pour des objets où on peut sélectionner des éléments tel qu’un Spinner. Possède deux méthodes à implémenter soit public void onItemSelected(AdapterView<?> parent, View view, int position, long id) où on va principalement utiliser le paramètre position qui représente la position de l’élément sélectionner dans la liste de possibilité ainsi que le paramètre view qui représente l’objet sélectionner dans la liste qu’on va pouvoir cast au type de l’objet sélectionné. Et la deuxième méthode public void onNothingSelected(AdapterView<?> parent) dont le paramètre représente l’objet utiliser pour faire la sélection.   
  
-implements AdapterView.OnItemClickedListener est utile pour des objets où on peut clicker sur des éléments tel qu’un ListView. Possède une méthode à implémenter soit onItemClick(AdapterView<?> parent, View view, int position, long id) où on va principalement utilisé le paramètre position qui nous permet de savoir quel élément a été cliqué par l’utilisateur. Le paramètre view nous permet de manipuler l’élément enfant du ListView qui vient d’être cliqué. Les autres paramètres ne seront pas particulièrement utilisés.   
  
-aSpinner.getSelectedItem() Nous retourne un objet contenant la valeur de spinner étant présentement sélectionnée.   
  
-aString.matches(string Regex) nous permet de ne pas avoir à initialiser un pattern et un matcher pour vérifier si le string respecte un regex.   
  
-aString.trim() nous retourne un string où tous les espaces blanc ont été retirés.   
  
-getActivity() nous retourne le contexte de notre élément. Si jamais on est déjà dans notre activity, donc dans onCreate, on peut utiliser this pour obtenir le contexte. Si on est dans un écouteur, on peut utiliser getApplicationContext().   
  
-getContext() nous retourne le contexte dans lequel notre élément est utilisé. Est utile si on se trouve dans un document/classe autre que celle de notre activité.   
  
-To note layoutParams, addView, getRessourceEntryName, R.drawable et R.id  
  
-aLayout.removeAllViewsInLayout() Comme le nom le propose, retire toutes les vues présentent dans un layout.   
  
-getResources().getResourceEntryName(aViewId) nous retourne le nom de l’Id d’une ressource sous forme de string à partir de l’Id int de ladit ressource. À ne pas confondre avec la commande getResources().getResourceName(aViewId) qui retourne le chemin vers la nom sous forme de string de notre ressource.   
  
-aLayout.addView(aView) nous permet de rajouter une view dans un layout.   
  
-LinearLayout.LayoutParams layoutParams = new LinearLayout.LayoutParams(int widthInPixels, int heightInPixels) Crée une variable pouvant contenir les paramètres d’une view. Par défaut on doit au moins initialiser la width ainsi que la height du param. Pour assigner match parent ou warp content on utilise RelativeLayout.LayoutParams.WRAP\_CONTENT ou RelativeLayout.LayoutParams.MATCH\_PARENT. Pour changer les marges on utilise layoutParams.setMargins(int left, int top, int right, int bottom). Pour changer le weight on utilise layoutParams.weight = aFloat.f.   
S’applique aux autres type de layout tel que le constraint layout ConstraintLayout.LayoutParams layoutParams = new ContraintLayout.LayoutParams(int widthInPixels, int heightInPixels).   
  
-aView.setLayoutParams(layoutParams) nous permettent d’appliquer un layoutParams à une view.   
  
-aView.setTextSize(int valueInPixel) nous permet de changer la taille du texte d’une view.   
  
-aView.setTextColor(Color.ACOLOR) nous permet de changer la couleur du texte d’une view. À ma connaissance, on ne peut utiliser que les couleurs présentes en variable statique de la classe Color.   
  
-aView.setGravity(Gavity.ADIRECTION) nous permet de changer la gravité de notre élément. On peut utiliser les variables statiques de la classe Gravity pour choisir la direction.   
  
-aView.setTypeFace(null, Typeface.BOLD/ITALIC/DEFAULT/etc) nous permet de changer le style du texte de notre widget.   
  
-R.id/drawable/color.aValue nous permet d’obtenir des valeurs présentes dans ces fichiers xml.   
  
-aView.getTag() si notre view possède une valeur Tag, nous retourne un objet tag qui peut être convertie en String grâce à toString().  
  
-getResources().getColor(R.color.myColor) nous permet d’utiliser notre couleur xml dans notre code java. C’est méthode et vieille et la plus récente est ContextCompat.getColor(MyActivity.this, R.color.myColor).   
  
-DisplayMetrics displayMetrics = getResources().getDisplayMetrics(); Les display metrics nous permette de savoir qu’elle est la largeur ainsi que la hauteur en pixel de notre écran grâce à displayMetrics.widthPixels/heightPixels. Ils nous permettent aussi de connaitre la densité de l’écran grâce à displayMetrics.density. Il existe aussi d’autre paramètre relié à la densité.   
  
-aView.setImageResource(int aDrawableId)/setImageDrawable(Drawable aDrawable) à la même résultat de changer l’image contenu dans notre view. La seule différence est le type du paramètre.  
  
***Paramètres fonctions Varargs***Un Varargs est un paramètre qui nous permet de rassembler sous un seul nom plusieurs valeurs du même type. On ne peut avoir qu’un Varargs dans la signature de notre fonction et celui-ci doit être le dernier paramètre. Il est aussi impossible d’utiliser un type primitif dans un Varargs.   
Ex : public void myFunction(Double… values){}  
On se déplace avec [] à travers le paramètre tel un tableau. On peut aussi utiliser la fonction values.length pour obtenir le nombre de valeurs présente dans notre Varargs.   
  
***Try Catch***Nous permet de prévenir des erreurs sans utiliser une multitude de ifs. On traite alors les situations inhabituelles / problématiques selon le contexte à l’aide d’objets Exception.   
Cette méthode a pour avantage de : 1. Séparer entre le traitement des exceptions et le déroulement normal du programme. 2. Richesses des objets exceptions qui peuvent contenir plusieurs infos complémentaires sur l’erreur en question plutôt qu’un simple code d’erreur. 3. Nous permet de traiter la situation problématique uniquement à l’endroit voulu. L’exception se propage dans la pile d’appel tant qu’elle n’est pas captée pour être traitée. 4. Hiérarchie des objets Exception : une erreur de lecture (transmettre une string quand on demande un int) peut être considérée comme telle ou comme une erreur générale d’entrée.   
  
Hiérarchie des classes :

Object

Throwable ( interface )

Exception Error

SQLException IOException Nos propres classes D’exception RuntimeException StackOverflowError OutOfMemoryError

NullPointerException IndexOurOfBoundsException ArithmeticException NumberFormatException …

Les *exceptions non contrôlées* sont celles qui se trouve être un enfant/sous-classe de Runtime Exception tel que : NullPointerException, ArithmeticException, NumberFormatException, etc…  
  
Les *exceptions contrôlées* sont des exceptions créées par des utilisateurs. En soit, ce sont des exceptions qui ne sont pas des sous-classes de RuntimeException. Tel que : SQLException (problème avec l’accès aux tables SQL), FileNotFoundException (Exception si le fichier à accéder n’est pas trouvé / n’existe pas), Nos propres exceptions, etc…   
On va lancer nos exceptions contrôlées dans le modèle pour que la logique de l’application soit complète 🡪 donc au premier endroit où il est possible de s’apercevoir de l’exception.   
On va capter l’exception/l’affiché dans la vue.   
On peut gérer les exceptions contrôlées à l’aide d’un try catch ou bien on la lance explicitement dans la pile d’appel.   
  
Pour utiliser un try catch, on commence par écrire notre try, à l’intérieur duquel on inclut les lignes qui pourront potentiellement produire une exception ou une erreur. Il nous faut ensuite écrire un ou plusieurs catchs associer à notre try pour éviter que ces exceptions/erreurs fassent planter le programme. Notre programme arrêtera d’exécuter le try du moment où il rencontre une ligne qui envoie une erreur/exception. (Ex : Si le try possède 3 lignes et que l’erreur/exception ce trouve à la ligne 1, alors les lignes 2 et 3 ne seront pas exécutées.   
Nos catchs prennent en paramètre le type d’exception/erreur qu’ils doivent corriger. Si on écrit catch(Exception e), toutes les exceptions ainsi que les sous classes d’exception seront attrapé par ce catch. Si on écrit catch(Error e), toutes les erreurs ainsi que les sous classes d’error seront attrapé par le catch. Si on écrit catch(NumberFormatException nfe), uniquement les number format exception et ses sous classes seront attrapés par le catch.   
Si on a plusieurs catchs et qu’un des catchs attrape l’erreur/exception, les catchs subséquents ne seront pas effectués. Car le besoin de gérer le problème à déjà été comblé par un des catchs, il n’est donc pas nécessaire de combler ce besoin à nouveau pour la même erreur/exception. Il faut donc mettre les catchs avec les erreurs/exceptions les plus spécifique avant ceux avec des erreurs/exceptions plus générale si on veut que ceux-ci soient exécutées. À noter que le compilateur ne retournera pas dans le bloc try une fois l’exception/erreur captée.   
Si aucun catch n’attrape l’erreur/exception, c’est à ce moment-là qu’on aura une erreur/exception dans la console.   
Le dernier bloc que l’on peut inclure dans notre try catch est finally et est facultatif. C’est un bloc de code qui sera exécuter indépendamment de si notre try a lancé une erreur/exception ou non. Même si l’erreur/exception n’est pas catch et que le programme crash, le bloc finally sera exécuté.   
  
-anException.getMessage() nous permet d’obtenir le message d’erreur de notre exception.   
  
-anException.printStackTrace() Imprime dans la console quel méthode a lancé notre exception.   
  
***Custom exceptions (custom controlled exception)***Il nous faut en premier lieu créer une classe qui va extends Exception. Le nom d’une classe qui extend Exception devra toujours finir par Exception.   
La classe Exception possède une variable message, une fonction getMessage et un constructeur pour initialiser le message.  
À l’intérieur de notre constructeur, on peut appeler super() et mettre un string en paramètre pour initialiser le message de notre exception.  
Pour appeler notre exception, on commence par mettre à la suite du nom de la méthode qui pourra appeler mon exception throws myException. Si notre méthode peut lancer plus d’une exception, on sépare simplement nos exceptions avec une virgule tel que throws myException, mySecondException. On inclut ensuite dans la méthode throw new myException lorsque celle-ci pourra se produire.   
En soit, on peut gérer notre exception contrôlée de deux manières différentes. La première est d’utiliser un try catch à l’intérieur de notre méthode (Lancer et attrapé l’exception à l’intérieur du modèle). La seconde est de throw notre exception vers la méthode qui a appelé la méthode qui a throw l’exception (Lancer l’exception dans le modèle et l’attraper dans la vue).   
Il est possible d’attraper une exception contrôlée beaucoup plus loin que dans la méthode qui a appelé la méthode qui a lancé l’exception. Par exemple, si methodeA qui appelle methodeB qui appelle methodeC qui elle peut lancer une exception, on peut attraper cette exception à l’intérieur de methodeA tant que les méthodes methodeB et methodeC possède l’attribut throws myException. Il n’est pas nécessaire d’inscrire le throws pour une exception non contrôlée.   
  
***Delay via Handler***Pour créer un délai suite auquel notre code sera exécuté, il nous faut en premier lieu créer un objet Handler tel que final Handler handler = new Handler(); Ensuite on utilise la méthode postDelayed de notre handler tel que handler.postDelayed(() -> { /\*Code to be executed after delay here\*/}, int delayInMilliSeconds); Notre code sera donc exécuté après le délai en milli seconde indiqué.   
  
***Continue***Utiliser à l’intérieur d’un loop et ressemble un peu à un break. Contrairement à un break qui va complétement quitter le loop lorsqu’il est lu, continue passera à l’itération suivante du loop dès qu’il est lu.   
  
***List***Forme de tableau. Il est important de déclarer le type des valeurs lorsqu’on déclare la List. On utilise myList.get(int index) pour obtenir une des valeurs de la List. On utilise myList.size() pour obtenir le nombre d’élément présent dans la List.   
Ex : List<Double> myList;   
  
***LinkedList***  
LinkedList<TypePrimitifContenu> est un type de tableau/vecteur qui agit comme une liste chainée.   
  
-linkedList.add() nous permet d’ajouter un objet du type primitif contenu à la fin de notre liste chainée.  
-linkedList.remove() retourne et retire le premier élément de notre liste chainée.   
  
***Enum***  
Une enum est une énumération de valeur contenu dans une variable. Notre enum est un type en soit qui ne peut être égal qu’au valeur contenu dans l’enum. myEnum.values() nous retourne une liste contenant toutes les valeurs possibles de notre enum.  
Ex : enum Tailles{petit, moyen, grand}; Tailles taille = petit;  
  
***Vector***  
-Vector<AType> nameOfVector = new Vector();   
Un vecteur est un tableau qui a une allocation dynamique de la mémoire. On peut donc rajouter des valeurs au fur et à mesure du programme dans notre vecteur. On peut limiter les types qui seront présent dans notre vecteur en spécifiant le type voulu à l’intérieur des <>. Un vecteur ne peut pas avoir de champ vide, on ne peut donc pas placer un élément à la position 10 si je n’ai que 3 éléments dans mon vecteur. On n’utilise pas de [] avec un vecteur.   
On peut uniquement mettre des objets dans un vecteur, pour les types primitifs on utilise les objets englobant du type tel que Double pour des doubles.   
La capacité initiale d’un vecteur est de 10 lorsqu’initialiser avec le constructeur de base, la place en mémoire pour 10 éléments est donc allouée à notre vecteur même si ces places sont présentement vides. Si on met un int en paramètre dans le constructeur de notre vecteur, la capacité initiale de notre vecteur sera la valeur du int.  
La taille/size d’un vecteur est le nombre d’objet présent dans le vecteur.  
La capacité d’un vecteur est le nombre d’objet que le vecteur peut contenir avant de devoir augmenter sa taille.   
Par défaut, lorsque toutes les places sont prise, la prochaine valeur ajouter au vecteur fera en sorte que la capacité courante de mon vecteur sera doublé donc 10 🡪 20 🡪 40 🡪 etc… Le deuxième paramètre du constructeur prend un int qui détermine de combien on veut augmenter la capacité de notre vecteur lorsqu’il ne reste plus de place donc Vector<AType> AVector = new Vector(3, 1); Les places augmenteront alors tel que 3 🡪 4 🡪 5 🡪 etc…  
Il existe un constructeur de vecteur qui prend en paramètre un Set, soit toutes les clés d’une hash table. Il prend en paramètre en soit une collection, donc soit un set (hashtable.keyset()) si on veut lui donner toutes les clés d’une hashtable ou encore une collection (hashtable.values()) si on veut lui donner tous les éléments de la hashtable.   
  
-nameOfVector.add(Object o) Nous permet d’ajouter un élément à notre vecteur à la dernière position possible. Si j’ai 2 éléments, le nouvel élément sera placer à la position 2.   
-nameOfVector.insertElementAt(Object o, int index) nous permet de placer une valeur à l’index voulu. Il faut placer l’objet à une position qui ne créera pas d’espace vide dans le vecteur. Si j’ai déjà un élément à la position 1 et que je place le nouvel élément à la position 1, l’ancienne valeur sera déplacée à la position 2 et ainsi de suite pour les autres valeurs aux indexes plus grand.   
-nameOfVector.contains(Objet o) Nous permet de vérifier si un vecteur contient la valeur/objet/référence à un objet mis en paramètre.   
-nameOfVector.get(int index) Nous retourne une référence à l’objet présent à l’index indiqué.  
-nameOfVector.set(int index, Object o) Remplace la valeur à l’index indiqué.   
-nameOfVector.remove(int index) Retourne une référence de l’objet et le retire du vecteur. Si on retire l’objet à la position 0, l’objet à la position 1 va combler la position 0, celui à la position 2 va combler la position 1, etc…  
-nameOfVector.remove(Objet o) Retourne un boolean true si on trouve une occurrence de l’objet mis en paramètre, retire alors la première occurrence. Si on retire l’objet à la position 0, l’objet à la position 1 va combler la position 0, celui à la position 2 va combler la position 1, etc…  
-nameOfVector.size() Retourne un int représentant le nombre d’objet contenu dans le vecteur.   
-nameOfvector.capacity() Retourne un int représentant le nombre d’objet que le vecteur peut contenir présentement, cela, avant d’augmenter sa capacité.   
-nameOfVector.trimToSize() Réduit la capacité au nombre d’élément contenu dans le vecteur.   
-nameOfVector.firstElement()/lastElement() Nous retourne le premier/dernier élément de notre vecteur.   
  
***Toast***Permet de faire apparaitre une notification à l’écran. Pour ce faire, il nous faut un objet Toast tel que Toast toast. Il nous faut une durée tel que int duration = Toast.LENGTH\_SHORT / Toast.LENGTH\_LONG. Un message qui serait contenu dans un string. Dans notre onCreate on crée ensuite la notification tel que toast = Toast.makeText(ClassName.this /\*MyActivity\*/, string myMessage, int duration). On peut ensuite utiliser toast.show() à l’endroit où on eut faire apparaitre notre notification.   
  
***AlertDialog***Permet de faire afficher un message dans un pop-up. La première étape est de créer un AlerDialog.Builder, important de s’assurer que l’import est android.app.AlertDialog;. Ensuite on créer un builder en lui donnant le contexte de l’activité tel que new AlertDialog.Builder(context);. Ensuite on peut associer un message et un titre à notre builder tel que builder.setMessage(‘’myMessage’’).setTitle(‘’myTitle’’). Il nous faut ensuite créer notre builder dans un AlertDialog dialog = builder.create(). Pour afficher notre message on fait dialog.show().   
Pour faire disparaitre une boite de dialogue on peut utiliser dialog.dismiss().  
  
Il est possible de créer des AlertDialog customisé. Pour cela :  
1. Il nous faut en premier lieu crée une nouvelle activité vide. On va ensuite changer le type de notre activité pour que celle-ci extends Dialog tel que public class MyAlertDialog extends Dialog.  
2. En soit, c’est tout. À partir d’ici on peut personnaliser notre boite de dialogue comme on le ferait pour une activité normale.   
3. Pour afficher notre boite de dialogue dans une autre activité, on crée un objet de notre classe de dialogue tel que MyAlertDialog myAlertDialog; (À l’intérieur de on create) myAlertDialog = new MyAlertDialog(Context myActivityContext); On peut maintenant simplement faire myAlertDialog.show() pour la faire apparaitre.   
  
***Bitmap***Une bitmap est l’agglomération de pixel qui compose une vue.   
Pour obtenir un objet Bitmap il nous faut en premier lieu utilisé le contexte qu’est notre vue pour appeler la méthode buildDrawingCache() tel que this.buildDrawingCache().   
Ensuite on va pouvoir sauvegarder dans un objet Bitmap nous bitmap avec la fonction Bitmap.createBitmap(this.getDrawingCache()) qui nous retourne une bitmap.   
Il ne faut pas oublier de supprimer la cache créé avec la fonction this.destroyDrawingCache().   
  
À partir d’une bitmap on peut faire plusieurs choses tel que :  
-myBitmap.getColor(int x, int y).toArgb() retourne la couleur du pixel dont la position est spécifié sous la forme d’un int.   
-myBitmap.getHeight()/getWidth() retourne respectivement le nombre de pixels qui compose la hauteur/largeur de notre bitmap.   
-myBitmap.setPixel.(int x, int y, int couleur) Nous permet de changer la couleur du pixel spécifié dans notre bitmap.   
  
Il est possible d’enregistrer notre bitmap dans la section photo d’un téléphone. Pour cela il faut :  
1. Créer un objet ContentValues et y placer le nom de l’image, le type de l’image ainsi que l’emplacement où l’image sera sauvegardée. Tel que ContentValues values = new ContentValues(); values.put(MediaStore.MediaColumns.DISPLAY\_NAME, ‘’myBitmap’’); values.put(MediaStore.MediaColumns.MIME\_TYPE, ‘’image/png’’); values.put(MediaStore.MediaColumns.RELATIVE\_PATH, Environment.DIRECTORY\_PICTURES);  
2.On crée un objet Uri qui va, à partir du contexte de notre activité, contenir le lien pour sauvegarder l’image tel que Uri uri = myActivity.this.getContentResolver().insert(MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI, values);  
3.On ouvre un output stream qui sauvegarder l’image compresser de notre bitmap dans le dossier voulu tel que try(OutputStream stream = MainActivity.this.getContentResolver().openOutputStream(uri)){ myBitmap.compress(Bitmap.CompressFormat.PNG, 100, stream); stream.close(); } catch(IOException e) { e.printStackTrace(); }   
  
***ColorWheel***Il existe plusieurs méthodes mais en voici une pour obtenir une boite de dialogue qui affichera une palette de couleur.   
La première étape est d’aller dans le fichier build.gradle (Module :app), aller dans la section dependencies inclure implementation ‘com.github.yukuku:ambilwarna:2.0.1’.  
Ensuite on crée en soit une boite de dialogue qui contiendra la palette en utilisant la classe AmbilWarnaDialog tel que AmvilWarnaDialog dialog = new AmbilWarnaDialog(Context context, int oldColor, newAmbilWarnaDialog.OnAmbilWarnaListener() {@Override public void onCancel(AmbilWarnaDialog dialog){} @Override public void onOk(AmbilWarnaDialog dialog, int newColor){}});  
Cette boite de dialogue prend en paramètre le contexte, la vieille couleur sous forme de int qui va être remplacé par la pallete. Les fonctions incluses dans le listener de la boite de dialogue sont onCancel qui nous permettra d’effectuer une action lorsque l’utilisateur fermera la boite de dialogue à l’aide du bouton cancel. Ainsi que la fonction onOk qui possède en paramètre la nouvelle couleur que l’on pourra récupérer à l’intérieur de la fonction et nous permet d’effectuer d’autres actions lorsque l’utilisateur appuie sur le bouton ok de la boite de dialogue.   
  
***Hashmap***Est une structure simple afin d’entreposer des paires clé-élément. À l’aide de la clé, on peut retrouver l’élément correspondant, il n’y a donc pas d’index. Les clés doivent donc être unique. En soit le hachage est lorsqu’on transforme une clé en un nombre par une fonction de hachage et c’est ce nombre qui est lié à l’élément correspondant, cela accélère la recherche.   
Noter qu’une hashtable est plus efficace pour des opérations qui sont multi threaded, alors qu’une hashmap est plus efficace pour des opérations qui sont single threaded. Autrement, ils n’ont pas de différence en terme de code.   
  
-Hashtable<keyType, elementType> nameOfHashtable = new Hashtable(); Nous permet de créer une nouvelle hashtable, on peut y mettre autant d’objet que l’on souhaite qui sont du type déclaré comme étant le elementType.   
  
-nameOfHashtable.put(key, element) Nous permet d’ajouter un objet à notre hashtable en y associant une clé.   
  
-nameOfHashtable.get(key) Nous retourne l’objet auquel la clé est associée.  
  
-nameOfHashtable.size() Nous retourne la taille de notre hash table.   
  
-nameOfHashtable.isEmpty() Nous retourne si la hash table est vide.   
  
-nameOfHashtable.keySet() Nous retourne un objet Set<String> qui contient toutes les clés de notre hash table sous forme de String.   
  
-nameOfHashtable.keySet().toArray(new String[0])) Nous retourne un tableau de String dont chacun des strings est une clé de notre hash table.   
  
-nameOfHashtable.values() Nous retourne un objet Collection<elementType> qui contient toutes les valeurs des éléments présent dans notre hash table.   
  
***Custom widgets***  
Pour créer notre propre widget, on commence par créer une classe qui va être une sous classe de View tel que private class SurfaceDessin extends View. Cette classe doit avoir un constructeur tel que public SurfaceDessin(Context context) {super(context);}  
Il nous faut ensuite override la fonction onDraw de la classe View tel que @Override protected void onDraw(Canvas canvas). Il faut inclure super.onDraw(canvas) dans notre fonction au minimum. Cette fonction se fera appeler lors de l’ajout de notre widget dans un layout, lorsqu’on utilise la fonction invalidate et lorsqu’une autre application apparait.  
Avant d’ajouter notre widget dans un layout, il nous faut lui appliquer des layoutParams. La taille de notre widget est en pixel, si on veut que la taille reste consistante entre différente machine il nous faut convertir la taille dp en pixel. Pour cela on multiplie le nombre de dp \* getResources().getDisplayMetrics().density = nbPixels. On doit arrondir le résultat pour obtenir un int qui sera utilisable comme nombre de pixel. Si on a la largeur et la hauteur de l’écran en pixel avec getResources().getDisplayMetrics().widthPixel/heightPixel on peut le multiplier par un pourcentage et arrondir ce chiffre pour obtenir un nombre de pixels utilisable.   
Pour ajouter des formes à notre widget, il nous faut un Canvas et un Paint. Paint va décider de plusieurs paramètres de notre forme tel que la couleur, si elle sera remplit, si elle utilise du antiAlias, la grosseur des traits de la forme, etc… Canvas va plutôt décider de l’emplacement de la forme dans notre widget, ainsi que de la forme de ladite forme. Le canvas nous est donné à l’intérieur de la fonction onDraw en tant que paramètre.   
  
-paint.setAntiAlias(boolean aBool) nous permet de déterminer si notre forme utilisera de l’antiAlias.   
  
-paint.setColor(int anInt) nous permet de déterminer la couleur de notre forme. À noter qu’on peut utiliser Color.ACOLOR pour choisir notre couleur ou encore les couleurs de notre fichier colros.xml grâce à getResources().getColor(R.color.myColor).   
  
-paint.setStyle(Paint.Style.STROKE/FILL) nous permet de déterminer si uniquement le contour de notre forme sera dessinée (STROKE) ou si elle sera aussi remplit (FILL).   
  
-paint.setStrokeWidth(aFloat) nous permet de changer la taille du trait produit par notre pinceau.   
  
-paint.setStrokeCap(Cap aCap) nous permet de changer la forme de notre crayon. Il existe plusieurs Cap pré fait à l’intérieur de la classe Paint tel que Paint.Cap.ROUND.   
  
-canvas.drawCircle(int xPosition, int yPosition, int radius, Paint paintUsed) nous permet de dessiner un cercle à la position de notre choix à l’intérieur de notre widget.   
  
-canvas.drawRect(new RectF(float leftMostPoint, float topMostPoint, float rightMostPoint, float bottomMostPoint), Paint paintUsed) nous permet de dessiner un rectangle à la position de notre choix en utilisant un objet RectF pour déterminer ses coordonnées.   
  
-canvas.drawArc(new RectF(float leftMostPoint, float topMostPoint, float rightMostPoint, float bottomMostPoint), float startAngle, float sweepAngle, boolean useCenterOfRectF, Paint paint) nous permet de dessiner des arcs d’un cercle. On indique le degrés de depart de notre arc (startAngle) et le nombre de degrés à partir de ce point que fera notre arc (sweepAngle). Je recommande de mettre useCenterOfRectF à true à part si tu souhaites expérimenter. On utilise le RectF pour déterminer la position ainsi que le rayon de notre arc de cercle.   
  
-canvas.drawLine(int PointDeDépartX, int PointDeDépartY, PointD’arrivéX, PointD’arrivéY, Paint paintUsed) Nous permet de dessiner une ligne allant du point de départ au point d’arrivé indiqué.   
  
-canvas.drawPath(Path myPath, Paint myPaint) Dessine un objet de type path selon les coordonnées incluent dans le path.   
  
-canvas.drawBitmap(Bitmap bitmap, int left, int top, Paint myPaint) Dessine un objet de type bitmap à partir du point left (x) et du point top (y) de notre canvas en utilisant la peinture spécifiée.   
  
-surfaceDessin.invalidate() permet d’effacer et de redessiner la surface de dessin. En soit, appelle onDraw à nouveau.  
  
***Densité***  
Une image ayant une grandeur fixe de 50 pixels peut avoir une grandeur différente sur deux écrans différents de même taille pour cause de la densité de ses écrans. Si un écran est plus dense en pixel que l’autre, alors l’image sera plus petite sur ce premier.   
  
***Path***Est un type d’objet qui nous permet de dessiner un chemin. Relie plusieurs points ensemble sans couture.   
  
-myPath.isEmpty() nous retourne une bolean qui nous indique si le chemin possède des coordonnées ou non.   
  
-myPath.reset() retire les coordonnées présente dans notre path.   
  
-myPath.moveTo() place le point de départ de notre path.   
  
-myPath.lineTo() si un point de départ a déjà été ajouté au path, relie le point précédent à ce nouveau point.   
  
***Fichier Jar***Il est possible de compiler une classe, en soit des fichiers java, sous forme jar. On peut ensuite plus tard utiliser ces fichiers jar en les incluant dans notre fichier gradle dans la section dependencies tel que implementation files(‘libs\\myFile.jar’).  
  
***Intent***Un objet Intent nous permet de changer d’activité lorsque l’application tourne. Lorsqu’on crée notre intent, il nous faut lui passer en paramètre le contexte de l’activité qui est notre point de départ et en second paramètre la classe de l’activité qui sera notre point d’arrivé tel que new Intent(MyActivity.this, MyOtherActivity.class);   
Pour changer d’activité, on utilise la méthode startActivity(myIntent).   
  
***SQLite***Est un système de base de données qui est intégrée par défaut dans le moteur d’exécution d’Android.   
Les types de données sont INTEGER (int), REAL (float), TEXT (string), NULL. Pour un booléen on recommande d’utiliser 0 pour faux et 1 pour vrai.   
Si on veut que notre clé primaire (id) s’incrémente tout seul la syntaxe est INTEGER PRIMARY KEY AUTOINCREMENT et notre clé primaire doit se nommer \_id.   
Pour créer une base de données SQLite, notre classe doit extends SQLiteOpenHelper.   
Le super du constructeur de notre base de données prend en paramètre le context (synonyme d’activité ou d’application), le nom de la base de donnée, factory (fabrique de curseur, qui servent à générer des requêtes) peut être mis à nul si on ne compte pas l’utilisé et version qu’on met à 1.  
Notre classe a deux méthodes obligatoires soit onCreate(SQLiteDatabase db) ainsi que onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion). Important de noter que onCreate est appelée uniquement lorsque l’application est installée. onCreate est donc uniquement appelé une fois.   
Si on veut rappeler onCreate, il est possible de changer dans notre constructeur la version associé à notre table. Si le chiffre de la version est plus grand que celle de table présentement créer, onUpgrade sera appelé lors de l’ouverture de l’application. On peut donc à l’intérieur de onUpgrade drop l’ancienne table et rappeler onCreate.   
  
Même si la base de données est locale, il est important d’ouvrir et de fermer la connexion à cette dernière.   
Pour ouvrir la connexion à notre base de données, il nous faut stocker dans un objet SQLiteDatabase à l’intérieur de notre classe qui extends SQLiteOpenHelper soit this.getWritableDatabase() si on compte écrire dans nos tables, autrement on peut placer dans notre objet this.getReadableDatabase() si on compte uniquement lire des valeurs de nos tables. this représentant le contexte de notre classe. Une fois la connexion ouverte, la variable qui contient la base de données est équivalent au paramètre db de nos fonction onCreate et onUpgrade.  
Pour fermer la connexion à notre base de données, on va simplement utiliser la méthode .close() sur notre objet SQLiteDatabase tel que database.close();.   
  
On exécute généralement les commandes suivantes à l’intérieur de nos fonctions onCreate et onUpgrade. On peut par contre les utilisé n’importe où tant qu’on accès à notre object SQLiteDatabase db :  
-myDataBase.execSQL(‘’\*\*\*your sql commande here\*\*\*’’); Nous permet d’exécuter du code sql dans notre base de données autre que les commandes insert ou select. Tel que db.execSQL(‘’create table inventeur(\_id INTEGER PRIMARY KEY AUTOINCREMENT, nom TEXT)’’);  
  
-myDataBase.insert(String tableName, String nullColumnHack, ContentValues values) Permet d’insérer des valeurs dans nos tables. Le premier paramètre est le nom de notre table, le deuxième on peut lui donner la valeur null, et le troisième est un objet ContentValues qu’on va remplir des valeurs que l’on veut insérer dans notre table.   
  
-myDataBase.rawQuery(String sql, String selectionArgs); On place dans le premier paramètre notre SELECT. Si on veut faire des comparaisons entre les valeurs de nos tables avec des variables java, on place à l’endroit où notre variable irait un point d’interrogation. Le deuxième argument prend un tableau de String dans lequel on va placer les valeurs des variables que l’on veut comparer avec les valeurs de nos tables. Si on n’a pas de tableau on met null. Nous retourne un Cursor.   
  
-object ContentValues cv Contient les données à ajouter sous forme de paires clé-valeur ou la clé est le nom du champ dans la table. Pour insérer des valeurs dans un objet ContentValues on utilise la méthode cv.put(String key, Byte value). Noter que si on veut insérer les valeurs de notre ContentValues il faut absolument que les clés est pour nom la même chose que les colonnes de notre table.   
  
-object Cursor c Peut contenir les valeurs retourner par un SELECT. Il faut visualiser cet objet vraiment tel un curseur. Au début, cet objet n’est pas positionné à (0, 0) mais plutôt à (0, -1). Pour le placer à (0, 0) on peut utiliser la fonction c.moveToFirst() ou encore simplement c.moveToNext(). Ensuite pour se déplacer à la prochaine rangée on continue d’utilisé c.moveToNext(). Noter que s’il existe une prochaine valeur, ces méthodes vont retourner true, autrement ils vont retourner false.   
Imaginons que notre query nous a retourné toutes les valeurs de la rangée bonjour et de la rangée allo dans cette ordre. Indépendamment de l’ordre de ces colonnes dans la table, la position x de bonjour est maintenant 0 et allo est 1. Si je veux obtenir les valeurs présentes dans la colonne allo, je vais donc utiliser c.getString/Int/Float(1) dépendement du type de valeur contenu dans cette colonne et assumant que position y est au moins 0.   
  
**Patron de conception : Singleton**  
Un patron de conception représente les grandes lignes que le programmeur pourra ensuite adapter à son projet pour régler un problème commun.   
Le singleton est le patron qu’on utilise quand on se rend compte qu’on a besoin d’exactement une instance pour tout le projet. On ne veut pas avoir à créer des objets bidons seulement pour appeler les méthodes de cette classe. Le singleton est un point d’accès global qui sera accessible à travers tout notre projet. Le singleton continue d’exister tant que l’application est vivante, car on travaille avec une instance statique.   
Le but d’un singleton est de s’assurer qu’on puisse créer qu’un seul objet de cette classe. On pourra récupérer des références à cet objet quand on a besoin de ses méthodes. Encore une fois, c’est un point d’accès global pour les méthodes/données stockées dans l’objet/singleton.   
L’instance de notre objet singleton sera créer en tant que variable static à l’intérieur de notre singleton. On utilise une méthode static pour avoir accès à l’instance de notre singleton. Cette méthode crée l’instance de notre objet si elle n’existe pas déjà et retourne ladite instance. Elle utilise le constructeur de notre classe qui est toujours private quand on parle d’un objet singleton.   
  
Note : xml shapes and vector, OnDragListener, view.startDragAndDrop, view.getParent, view.setVisibility, getResources().getDrawable(R.drawable.myDrawable),