**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.   
  
-Dans /values/themes/themes.xml on peut changer notre thème ainsi que les couleurs de notre thème.   
  
**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.   
  
-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 the spinner.setAdapter(adapter).   
  
-ChipGroup and Chip sont un type de bouton où une seule 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.   
  
-android :   
  
***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).   
-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, 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.   
  
**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.   
  
-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.   
  
-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.  
  
-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.   
  
-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.   
  
-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.   
  
-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().   
  
-enum et values enum  
  
***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.   
  
***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;   
  
***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.   
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.   
  
-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.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);. En suite 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.create().   
  
***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.   
  
-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.