M4206C

Développement d'applications mobiles

iut réseaux & télécoms

LA RÉUNION
Institut Universitaire de Technologie

Maxime Ferrino — Denis Payet
01/2018 — IUT de Saint-Pierre

Module M4206C

Durée : 45h

- 6H de CM
- 6H de TD
- 27H de TP
- 1H de devoir

Objectif du module :

- Développer des applications client-serveur pour Smartphone

Compétences visées :

- Programmer des mécanismes événementiels
- Réaliser des interfaces graphiques utilisateur pour Smartphone
- Mettre en œuvre des applications client-serveur

Maxime Ferrino — Denis Payet
01/2018 — IUT de Saint-Pierre

Ce document

- Téléchargement
  - Les documents de ce cours peuvent être téléchargés via l'adresse suivante : https://github.com/Dagrut/M4206C
- Auteurs
  - Maxime FERRINO, avec la participation de Denis PAYET et de Bruno GUEGAN.
- Contact
  - mf+m4206c@dagrut.info

Sommaire

Langage Java - Rappels

Maxime Ferrino — Denis Payet
01/2018 — IUT de Saint-Pierre

Le langage java

- Les variables
- Le nom des variables
- Les types
- Les expressions
- Les opérateurs
- Les appels de fonctions
- Les instructions
- Les structures conditionnelles
- Le switch
- Les boucles
- Le type String
- Les tableaux
- Les fonctions
- La fonction main()

Le langage java – les types

- char
- byte
- short
- int
- long
- float
- double
- boolean

Parenthèse : Caractère, table de caractère et encodage

- Qu'est-ce qu'un caractère?
- Qu'est-ce qu'une table de caractère?
- Qu'est-ce qu'un encodage?

- Quelques mots clés :
  - ASCII
  - Utf-8
  - Utf-16
  - Unicode

Le langage java – Le type String

- Qu'est-ce que le type **String**?
- Exemples :

```
String hello = "Hello world\n";
int b = hello.length;
System.out.println("Le texte\"" + hello + "\" fait " + b + "
caractères de long");
b = hello.indexOf("ll");
System.out.println("Le texte \"ll\" est à la position " + b);
char lettre;
for(int i = 0 ; i < hello.length() ; i++ ) {
    lettre = hello.charAt(i);
    System.out.println(lettre);
}
```

Le langage java – Les tableaux

- Exemples :

```
double d[] = new double[10];
for(int i = 0 ; i < d.length ; i++) {
    d[i] = 3.14;
    System.out.println("Valeur à
l'indice " + i + " : " + d[i]);
}
d[10] = 3.14; // Provoque une erreur
```

Le langage java – Les fonctions

- À quoi sert une fonction?
- Toujours de la forme :

```
motsClés nom(arguments) {
    contenu;
}
```

- Exemple :

```
public static int multiplier(int a, int b) {
    return a * b;
}
```

- Cas particulier : la fonction main()

La Programmation Orientée Objet (1/2)

- La programmation procédurale
- Les classes
  - Les attributs
  - Les méthodes
- L'instanciation et les instances
- Le constructeur
- Les propriétés statiques
- L'héritage
  - Le masquage

La POO : différences avec la programmation procédurale

- Programmation procédurale :
  - Fonctions (code, logique) séparées des données
- Programmation Orientée Objet :
  - Fonctions (méthodes) liées aux données (attributs)

La POO : Classes et objets

- Une classe est un plan, une structure
- Un objet est l'**instance** d'une classe
- L'instance d'une classe se crée via le mot clé **new**.
- Deux classes différentes permettent de créer (d'**instancier**) deux types d'objets différents
- Une classe définit donc un type évolué, comme String par exemple

La POO :
Les attributs et méthodes

- Dans une classe :
  - Une variable de cette classe est un **attribut**
  - Une fonction de cette classe est une **méthode**
- Exemple :

```
public class MaClasse {
    int mon_attribut;
    public void maMethode() {
        return;
    }
    public static void maFonction() {

    }
}
```

La POO : Les constructeurs

- Un constructeur est une méthode particulière d'une classe :
  - Elle est appelée lorsqu'un objet de cette classe est instancié
  - Elle possède le même nom que sa classe
- Un constructeur par défaut est présent :
  - Il ne fait rien et ne prend pas d'arguments
  - Il n'existe plus si l'utilisateur définit son propre constructeur

La POO : L'héritage

- Lorsqu'une classe B **hérite** d'une classe A, elle reprend toutes ses méthodes et ses attributs.
  - La classe B est alors **compatible** avec la classe A : Un objet de type B peut se faire passer pour un objet de type A.
  - L'héritage est une spécialisation
- Exemple : Une classe Vtt pourrait hériter d'une classe Velo (parce qu'un VTT est un vélo!)

La Programmation Orientée Objet (2/2)

- Le polymorphisme
- Les classes abstraites
- Les interfaces
- Les types génériques

La POO : Le polymorphisme

- Soit l'exemple suivant :

```
public class A {
    public int maMethode() {
        return 0;
    }
}
public class B extends A {
    public int maMethode() {
        return 1;
    }
}
```

- Ailleurs dans une autre classe :

```
A monObjet = new B();
System.out.println("" + monObjet.maMethode());
```

- Que va afficher ce code, 0 ou 1?

La POO : Le mot clé **super**

- Le mot clé **super** permet d'accéder aux méthoes de l'instance de la classe parente.
- En reprenant le même exemple qu'à la diapositive précédente, on pourrait avoir :

```
public class B extends A {

    public int maMethode() {

        // On fait d'autres opérations ici,
par exemple

        return super.maMethode();

    }
}
```

La POO : Classes abstraites

- Une classe abstraite est une classe qui ne peut pas être instanciée.
- Dans une classes abstraite, au moins une méthode est définie comme abstraite : Son code n'est pas défini
  - Ce sont aux classes héritant de celle-ci que revient la charge de définir ces méthodes
- Pourquoi : Car dans certains cas, définir une méthode dans la classe parent n'a pas de sens
- Une méthode abstraite se définit avec le mot clé **abstract** sur celle-ci et sur une de ses méthodes : **public abstract int** maMethode();

La POO : Les interfaces (1/2)

- Une interface fonctionne comme une classe abstraite, sauf qu'elle :
  - Ne contient aucun code (toutes les méthodes sont vides)
  - Ne contient aucun attribut
  - N'est qu'un squelette de classe
- Pour être utilisé, il faut qu'une classe **implémente** cette interface (et donc toutes ses méthodes)
- Notez qu'une classe peut implémenter plusieurs interfaces! (elles sont alors séparées par des virgules)
- Attention : Même si on ne peut pas instancier une interface, elle peut quand même être un type!

La POO : Les interfaces (2/2)

- Exemple :

```
public interface MonInterface {
    public void maMethode();
}
public class MaClasse implements
MonInterface {
    // […] Ici on implémente maMethode
}
// Ailleurs on pourra écrire :
MonInterface ex = new MaClasse();
```

La POO : Les types génériques (1/2)

- Un type générique permet d'avoir une classe ou une interface dont certains attributs, arguments de méthodes et/ou valeurs de retour auront des types définis par l'utilisateur
- Par exemple pour créer une classe générique, on écrira :

```
public class Paire<P,S> {
    P premier;
    S second;
    public Paire(P p, S s){
        premier = p;
        second = s;
    }
    public P getPremier(){ return(premier); }
    public S getSecond(){ return(second); }
}
```

La POO : Les types génériques (2/2)

- Pour utiliser cette classe on pourra alors écrire :

```
Paire<String, String> ma_paire =
    new Paire<String, String> ("Date", "01/01/2017");
```

- Un type générique permet :
  - D'éviter des **cast** superflus
  - De créer une infinité de types en fonction de ses paramètres
  - De créer une classe pouvant manipuler une grande variété d'objets sans pour autant définir explicitement les quels
- Un type générique ne peut prendre en paramètre **que des classes** (pas de int, float, byte, ...)

Notions Java avancées

Maxime Ferrino — Denis Payet
01/2018 — IUT de Saint-Pierre

Mot clé **package**

- Permet de définir à quel package appartient le fichier actuel : cela crée un **namespace**
- Doit être la première ligne non commentée du fichier
- Exemple :
  - **package com.android.example.example2;**
- Dans l'exemple précédent, il faut alors que le fichier se trouve dans le dossier **com/android/example/example2**
- **Attention :** Il est important que le nom du package soit unique globalement, afin de ne pas créer de conflits!

Mot clé **package**

- Chaque classe ou méthode d'un package peut accéder à toutes les classes de ce package, y compris les classes **privées** ou **protégées**
- Les classes et méthodes d'un autre package n'y auront en revanche pas accès
- La création d'un package en java passe par la création d'un fichier **.jar** (Java Archive)
- Lorsque vous ne précisez pas le nom du package, le package par défaut est utilisé (nom vide)

Mot clé **import**

- Permet d'utiliser un package
  - Cela permet de ne pas entrer le namespace complet définissant le package pour accéder à ses classes
- Exemple d'utilisation :
  - **`import java.awt.event.ActionEvent;`**
- Avec l'appel précédent, il est possible de faire :
  - **`new ActionEvent();`**
- Sans cet import, pour obtenir le même résultat, il faut faire :
  - **`new java.awt.event.ActionEvent();`**

Mot clé **import**

- Java permet l'importation de plusieurs packages ou classes en même temps :
  - **`import java.awt.event.*;`**
- **Attention :** Cela peut certes simplifier votre code certaines fois, mais si plusieurs classes de différents packages comportent le même nom, il se produira un conflit lors de leur utilisation
  - Cela peut être résolu en utilisant le nom complet de la classe ou méthode, comme montré précédemment

Classes anonymes

- Description du problème :
  - Certains packages (principalement avec Android) fournissent une classe abstraite ou interface avec quelques fonctions seulement à surcharger (typiquement une seule)
    - Exemple de la classe OnClickListener sur Android
  - Cette interface pourrait être définie ainsi :

```
public interface OnClickListener {
    public void onClick(DialogInterface dialog, int which);
};
```

Classes anonymes

```
public interface OnClickListener {
    public void onClick(DialogInterface dialog, int which);
};
```

- Une utilisation usuelle de cette classe serait alors de créer une classe comme celle-ci :

```
public class MonListener {
    @override
    public void onClick(DialogInterface dialog, int which) {
        System.out.println("Clic!");
    }
};
```

- On se rend rapidement compte que créer une classe pour si peu de choses peut rapidement devenir rédhibitoire
  - C'est ici qu'interviennent les classes anonymes

Classes anonymes

```
public interface OnClickListener {
    public void onClick(DialogInterface dialog, int which);
};
```

- En reprenant ce même exemple, une classe fille peut être créée et instanciée en même temps :

```
/* Quelque part dans une de vos fonctions ou méthodes : */
OnClickListener monListener = new OnClickListener() {
    @override
    public void onClick(DialogInterface dialog, int which) {
        System.out.println("Clic!");
    }
};
```

- Cette classe fonctionne comme toute autre. Cependant, elle n'a pas de nom et ne peut être instanciée qu'une fois.

Classes anonymes

- Cette méthode est très souvent utilisée pour des développements sur Android
- Ne vous étonnez donc pas de la voir à plusieurs reprises!

Les exceptions

- Java, comme d'autres langages orientés objets, permet une gestion d'erreur via des exceptions
- Cinq mots clés sont disponibles pour gérer les exceptions
  - `try`
  - `throw`
  - `throws`
  - `catch`
  - `finally`

Les exceptions

- Exemple d'utilisation (1/2) :

```
/* Dans un fichier Division.java */
public class Division {
    private int mDiviseur;
    public Division(int diviseur) throws ArithmeticException {
        if(diviseur == 0)
            throw new ArithmeticException("Diviseur ne peut pas être
zéro!");
        this.mDiviseur = diviseur;
    }
};
```

Les exceptions

- Exemple d'utilisation (2/2) :

```
/* Dans un fichier JeGereLesErreurs.java */
public class JeGereLesErreurs {
    public void jeGere {
        Division p;
        try {
            p = new Division(0);
        }
        catch(ArithmeticException e) {
            System.out.println("Erreur d'arithmétique!");
            e.printStackTrace();
        }
        catch(Exception e) {
            System.out.println("Le constructeur de Division a émis une
exception inconnue!");
            e.printStackTrace();
        }
        finally {
            p = null;
        }
    }
};
```

Le mot clé **final**

- Permet plusieurs choses :
  - Sur une classe
    - Il n'est alors plus possible d'hériter de cette classe
  - Sur une fonction
    - Il n'est alors plus possible de surcharger cette fonction
  - Sur une variable
    - Il n'est possible de définir sa valeur qu'une fois
    - C'est le compilateur qui s'en assure

Le mot clé **final** : Exemple

```
/* Une classe et un attribut statiques */
public final class MaClasse {
    public final uneValeurConstante;

    public MaClasse() {
        uneValeurConstante = 42;
        /* La valeur ne peut être changée */
    }
};

/* Il est possible d'hériter de LaClasse, mais ses fonctions ne
peuvent être surchargées */
public class LaClasse {
    public final jeNeFaisRien() {
        System.out.println("Je ne peux pas être surchargée!");
    }

    public static final JeNenFaisPasPlus() {
        System.out.println("De même!");
    }
};
```

Les classes imbriquées

- Java permet la création de classes à l'intérieur d'autres classes
- Ladite classe peut être définie :
  - Dans la classe même
  - Dans une méthode de la classe (On l'appelle alors classe locale)
- Ce type de classes est assez souvent utilisé, pour un gain de place et une meilleure lisibilité.

Les classes imbriquées

- Raisons d'utiliser des classes imbriquées :
  - Cela permet de **regrouper** des classes qui ont un fort lien logique entre elles
  - Cela augmente l'**encapsulation** :
    - Les classes internes, si elles sont privées, peuvent accéder à des attributs privées de la classe principale
    - L'accès est direct si un attribut ou une variable est déclaré **final**
  - Dans certains cas, cela peut rendre le code plus court et **plus lisible**

Les classes imbriquées : Exemple

```
public class ClasseExterne {
    private int unEntier;

    private class ClasseInterne {
        public void jeNePlantePas() {
            ClasseExterne.this.unEntier = 42;
        }
    };

    public void uneFonctionQuelconque() {
        ClasseInterne c = new ClasseInterne();
        c.jeNePlantePas();
    }
};
```

Bonnes pratiques en programmation

Maxime Ferrino — Denis Payet
01/2018 — IUT de Saint-Pierre

L'indentation

- L'indentation du code consiste en l'ajout de tabulations ou d'espaces dans un fichier
  - Suivant les langages elle n'est pas qu'esthétique (Ex: python, occam)
- La manière d'indenter un code forme un style d'indentation
- Vous **DEVREZ** respecter un même style dans tous les fichiers d'un même projet
- Styles conseillés pour ce cours : K&R ou Allman
- Référez-vous à https://fr.wikipedia.org/wiki/Style_d'indentation pour en savoir plus

L'indentation

- Exemple avec l'indentation Allman :

```
public static void main(String argv[])
{
    uneFonction();
    if(quelquechose)
    {
        erreur();
    }
}
```

Conventions de nommage

- C'est l'ensemble des règles destinées à choisir des identifiants (noms de variables, fonctions, classes, etc.) uniformément dans un projet ou code.
- Cela permet de faciliter la lecture du code
- Vous **DEVREZ** respecter une même convention de nommage dans tous les fichiers d'un même projet

Conventions de nommage

- À suivre dans vos projets :
  - Classe: `ExempleDeClasse`
  - Méthode: `exempleDeMethode`
  - Méthode statique: `ExempleDeMethodeStatique`
  - Variable: `exempleDeVariable`
  - Attribut: `mExempleDAttribut`

Gestionnaire de versions

- Pensez à utiliser un gestionnaire de versions comme git, qui vous permettra de :
  - voir l'évolution de votre code
  - Revenir à une version antérieure fonctionnelle
- Vous pourrez exclure les fichiers temporaires d'Android Studio en mettant un fichier `.gitignore` à la racine de votre projet
  - Un exemple de fichier .gitignore peut se télécharger via l'Url https://github.com/github/gitignore/blob/master/Android.gitignore

Généralités sur Android

Maxime Ferrino — Denis Payet
01/2018 — IUT de Saint-Pierre

Généralités sur Android

- Racheté en 2005 par une startup éponyme
- Proposé gratuitement aux fabricants de téléphones
- Codes sources ouverts
- Réellement lancé sur le marché en 2008
- Précédé par Apple avec l'iPhone en 2007
- Objectif initial : pouvoir prendre des photos avec un téléphone portable

Généralités sur Android

Aujourd'hui :

Android est rentable grâce aux publicités affichées sur les téléphones (1 Milliard de dollars US par an dès 2010)

À noter :

Android n'est pas seulement un système d'exploitation, c'est aussi un ensemble d'applications, d'APIs & d'outils

Utilisation actuelle

Où trouve t-on Android ?

- Téléphones
- Tablettes
- Montres
- Consoles de jeux
- Vêtements (Android Wear)
- Voitures
- Lunettes

Utilisation actuelle

Qui utilise Android ?

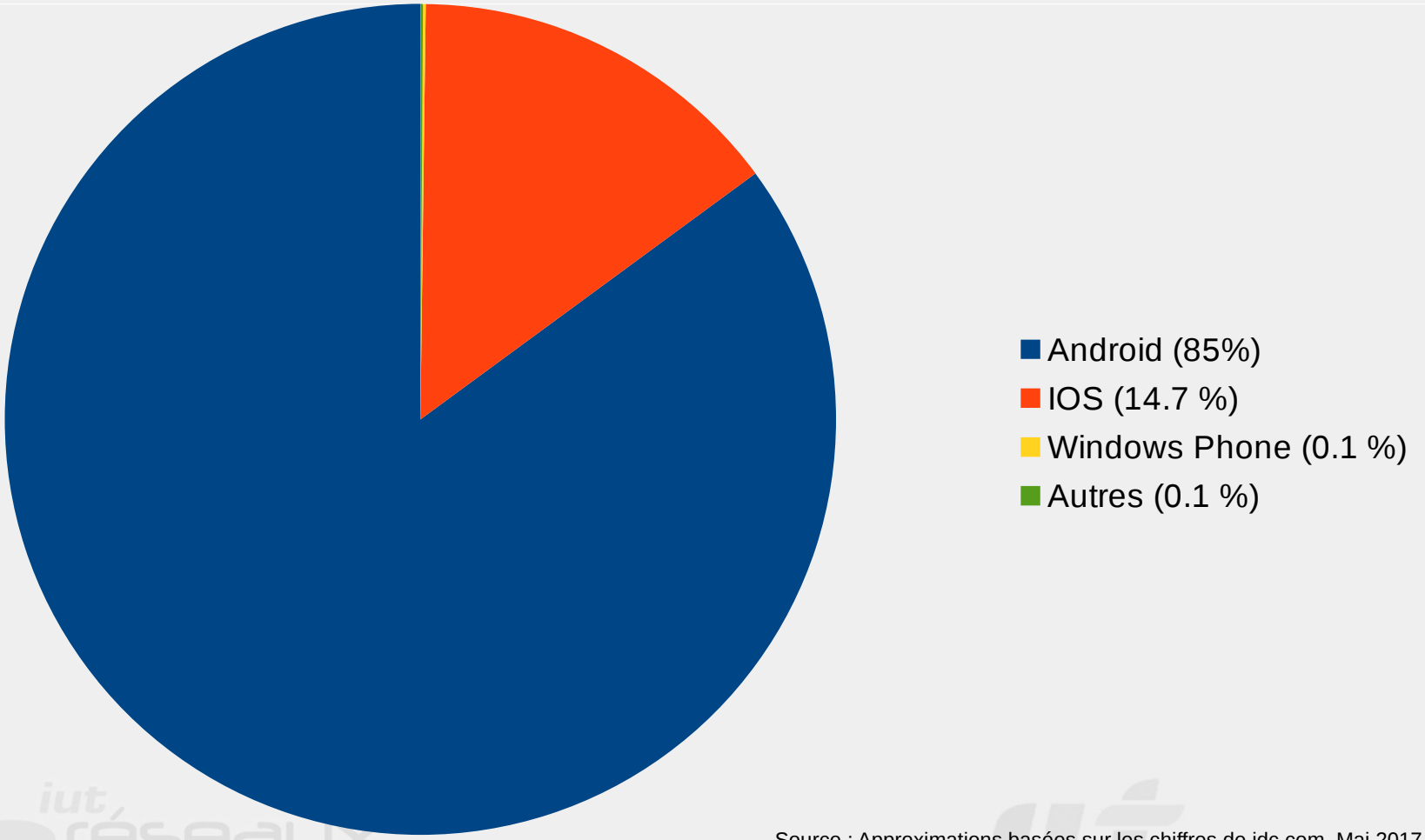
- Constructeurs (de téléphones par ex.)
- *Google* permet la personnalisation de l'*OS* avant installation par les constructeurs
- C'est du coup au constructeur de gérer les mises à jours

Utilisation actuelle

Avantages pour Google ?

- Récolte de données
- Mise en commun des informations
- Anticiper les comportements
- Proposer des publicités adaptées

Répartition des OS sur le marché des mobiles



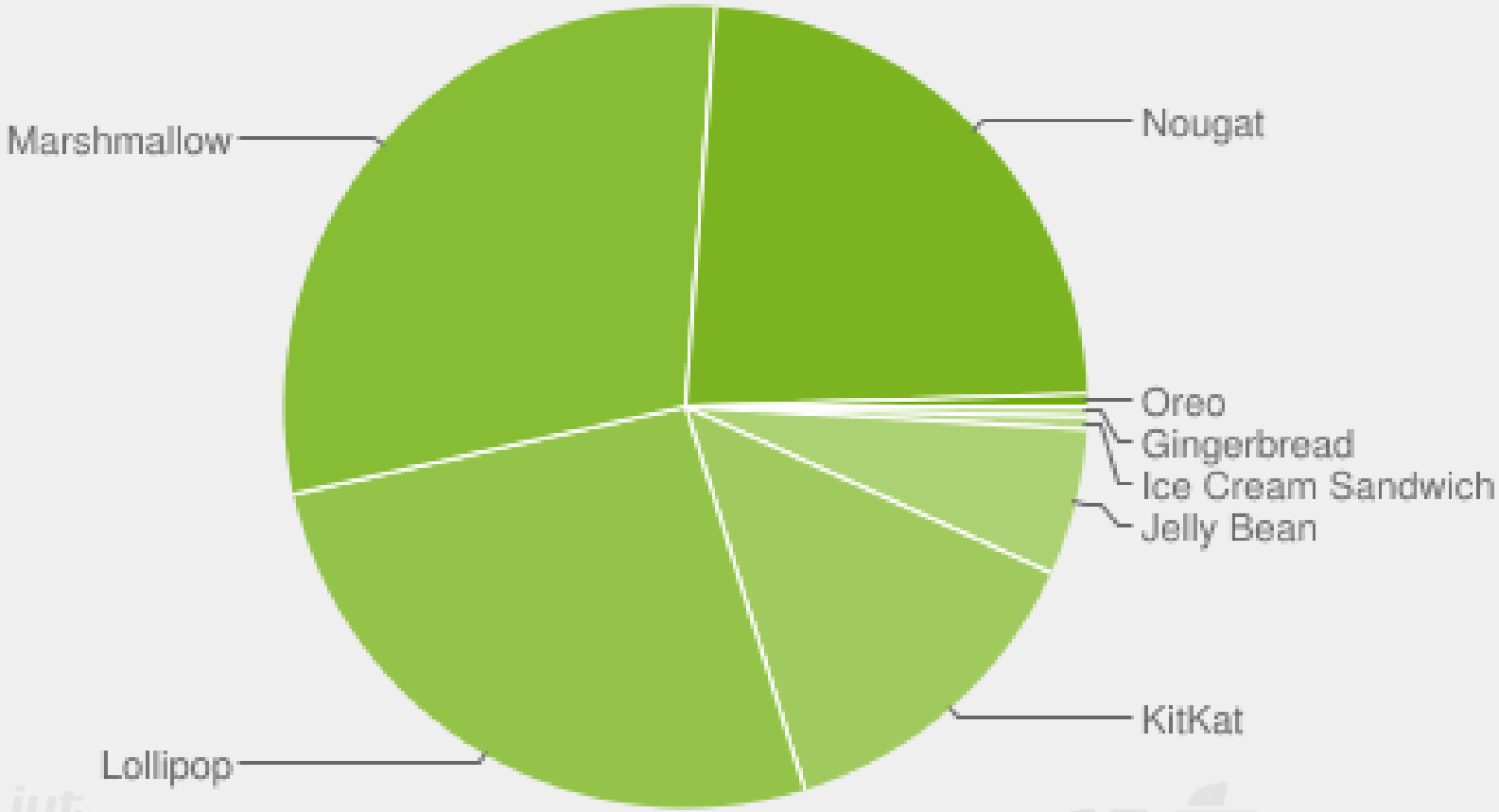
Source : Approximations basées sur les chiffres de idc.com, Mai 2017
Pour le 1er trimestre 2017

Répartition des versions d'Android

Version	Nom de code	API	Répartition
2.3.3 -2.3.7	Gingerbread	10	0.4%
4.0.3 -4.0.4	Ice Cream Sandwich	15	0.5%
4.1.x	Jelly Bean	16	2.0%
4.2.x		17	3.0%
4.3		18	0.9%
4.4	KitKat	19	13.4%
5.0	Lollipop	21	6.1%
5.1		22	20.2%
6.0	Marshmallow	23	29.7%
7.0	Nougat	24	19.3%
7.1		25	4.0%
8.0	Oreo	26	0.5%

Source : https://developer.android.com/about/dashboards/index.html, du 11/12/2017

Répartition des versions d'Android



Source : https://developer.android.com/about/dashboards/index.html, du 11/12/2017

Répartition des versions d'Android

Raisons de la fragmentation importante ?

- Certains fabricants ne proposent que très peu de mises à jour
- Les utilisateurs n'ont pas toujours un accès internet ou un débit suffisant pour les télécharger

Répartition des versions d'Android

Problèmes que cela provoque ?

- Failles de sécurités parfois importantes
- Bugs sur certains appareils

=> Cela diminue la confiance des gens dans l'OS

Versions alternatives d'Android

CyanogenMod

- Offre une alternative sans certaines fonctionnalités intrusives

Replicant

- Orienté 100% libre, ne contient aucun paquet propriétaire

Les versions AOSP

- Faites par chaque fabricants, qui peuvent parfois les difuser ou non

Versions alternatives d'Android

Les versions AOSP

- Android Open Source Project
- Versions 100% compatibles avec Android, les modifications sont apportées par les constructeurs

Exemple : **MIUI**

Développé par Xiaomi

Autres versions ? *On en dénombre une vingtaine*

Concurrents à Android

- **iOS**
  - OS Mobile d'Apple, compatible uniquement avec ses propres équippements
- **Windows Phone**
  - OS Mobile de Microsoft, part de marché très faible
- **Blackberry**
  - OS mobile de Blackberry, développement pour mobile arrêté en 2016 faute de ventes suffisantes

Comparaisons des principaux concurrents d'Android

	Langages utilisés	Nature de l'OS	Année de sortie initiale
Android	Principal : Java Autres : C++	Open source	2008
iOS	Principaux : Objective-C et Swift Autres : C, C++	Propriétaire	2007
Windows Phone	Principal : C# Autres : C++	Propriétaire	2010
Blackberry	Principal : C++/Qt Autres: C, web (HTML+CSS+JS), ActionScript/AIR, Java	Propriétaire	2002

API Android

Maxime Ferrino — Denis Payet
01/2018 — IUT de Saint-Pierre

Vue d'ensemble

Ce qui sera abordé dans ce cours

- Principaux éléments graphiques
- Principales APIs standards sur Android

Ce qui ne le sera pas

- Éléments graphiques redondants
- APIs spécifiques ou APIs externes
- Il existe déjà plus de 230 bibliothèques standards inclues dans Android

Documentation complète sur
https://developer.android.com/reference/packages.html

Couches de l'API

1) Le noyau (basé sur Linux)
2) Les librairies C/C++ telles que Webkit ou OpenGL
3) Un environnement d'exécution (Machine virtuelle **Dalvik**)
4) Un framework Java
5) Des applications Java

C'est à ce niveau que nous allons être

Développement d'applications

Les APIs Android permettent l'accès à l'ensemble des périphériques & fonctionnalités matérielles du téléphone (c'est du moins un des principaux objectifs de ces APIs)

Développement d'applications

3 approches de développement possibles

- Applications java qui s'exécute sur une machine virtuelle Dalvik
  - Mode de développement privilégié par les concepteurs d'Android
  - Ce que nous allons utiliser ici
- Applications web (ou Webapp), accessibles dans un navigateur
- Applications natives écrites en C

On choisit généralement l'approche en fonction des contraintes & besoins de l'application à développer

Dalvik

- Similaire à une machine virtuelle *Java*
- Compilation en **.class** puis conversion en **.dex**
- Format spécifique à *Dalvik* (ne peut être interprété par **j2se** de *Java*
- Plus petit en terme d'espace disque utilisé que les **.class**

Format d'une application Android

Fichier avec l'extension **.apk**

Compressée au format *zip*, avec un format basé sur les fichier **.jar** (*Java ARchive*)

Contient

- Un fichier **Manifest** qui recense le nom de l'application, les bibliothèques nécessaires, les permissions requises par l'application, les composants de l'applications (activités, services, etc.)

Format d'une application Android

Contient également

- Certificats
- Différentes versions de l'application, compilés pour plusieurs architectures (Ici il n'est pas question de *java*)
- Diverses ressources (images, vidéos, fichiers de données, traductions, fichiers d'interface, etc.)
- Les classes au format **.dex**

Éléments principaux d'une application

- Activity (Activités)
- Services
- Broadcast Receivers
- Content Providers
- Intents
- Les 4 premiers éléments sont définis comme des "composants" : Ils doivent généralement être déclarés dans le fichier **Manifest**

Les activités

- C'est un élément de l'interface utilisateur (Une « page » de l'application), ainsi que le code associé pour la gérer
- Elles sont indépendantes les unes des autres

Les services

- C'est soit une tâche de fond qui a besoin de faire des calculs quand l'application n'est plus lancée, soit une manière de fournir des fonctionnalités à d'autres applications.
- Pas d'interface visuelle pour les services
- Ils tournent en arrière plan
  - Par exemple jouer de la musique
  - L'utilisateur peut faire autre chose en parallèle
- Exposent une interface de communication, pour gérer le fonctionnement de ce service.

Les Broadcast Receivers

- Permet la réception de messages émis sur l'ensemble du système
  - Ex: Batterie faible
- Permet la communication entre plusieurs composants
- Pas d'interface utilisateur
- Possibilité de :
  - Démarrer une activité
  - Alerter un utilisateur (Via une notification par exemple)

Les Content Providers

- Rend disponible à d'autres applications certaines données
- Moyen de partage de données entre deux applications
- Format de stockage des données laissé à la discrétion de l'application (fichier, base de données, réseau, ...)
- Accès des données uniquement possible via la classe ContentResolver

Les Intents

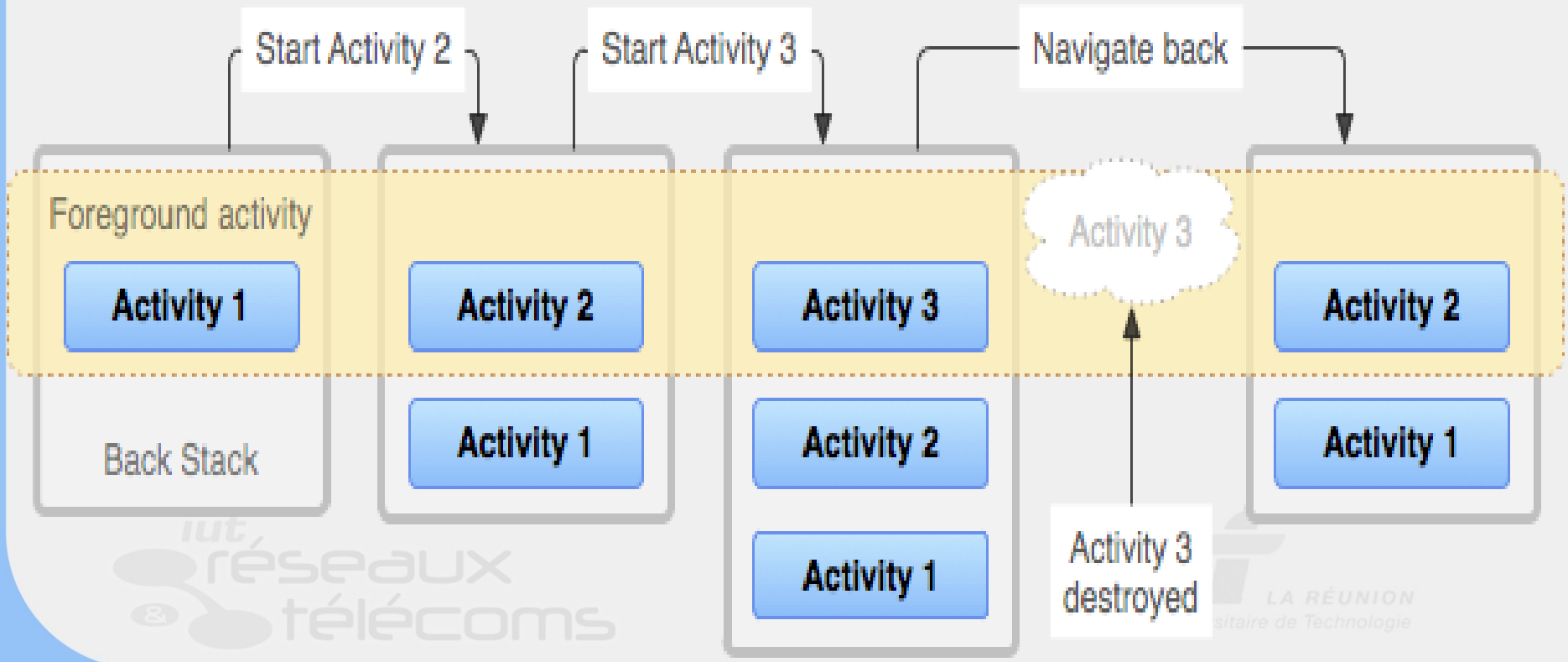
- Description abstraite d'une opération à réaliser (par exemple lancer une activité, ou déclarer un broadcast receiver)
- Peut contenir des données supplémentaires pour réaliser l'action souhaitée
  - Les données utiles au système pour exécuter ce qui est demandé
  - Les données utiles au récepteur (par exemple à l'activité) qui va être exécuté

Démarrage d'une application Android

- Android ne fournit pas de fonction main() unique, comme en java
  - Plusieurs points d'entrées sont possibles
- Une application est exécutée lorsqu'elle ou un de ses composants (Ex. Services) est requis.

Les Tâches & activités

- Les tâches sont des groupes d'activités
- Ces activités sont ordonnées



Les Tâches & activités

Attention :

- Une même activité peut être instanciée plusieurs fois !
- Une tâche peut être une application différente, mais une activité peut aussi se définir comme étant une autre tâche !

Les Tâches & activités

- Une activité peut démarrer une seconde activité appartenant à une autre application
- Exemple : L'application Mail peut lancer l'activité Gallery pour sélectionner une image à joindre à un mail.
- Si plusieurs applications permettent de réaliser une même opération, c'est à l'utilisateur de choisir quelle application ouvrir
  - Par exemple pour une page web : Utiliser le navigateur par défaut, chrome, ou encore firefox

Démarrage d'une application

- Peut se faire depuis le « home » ou depuis une autre application
- Points d'entrées possibles définis par l'application dans le fichier **AndroidManifest.xml**

Note : Lors du lancement de ladite application, il se peut qu'aucune application ne puisse répondre. Il faut alors vérifier que ce soit le cas, afin de ne pas provoquer de plantage. (Via la fonction `packageManager.queryIntentActivities`)

Durée de vie d'une activité

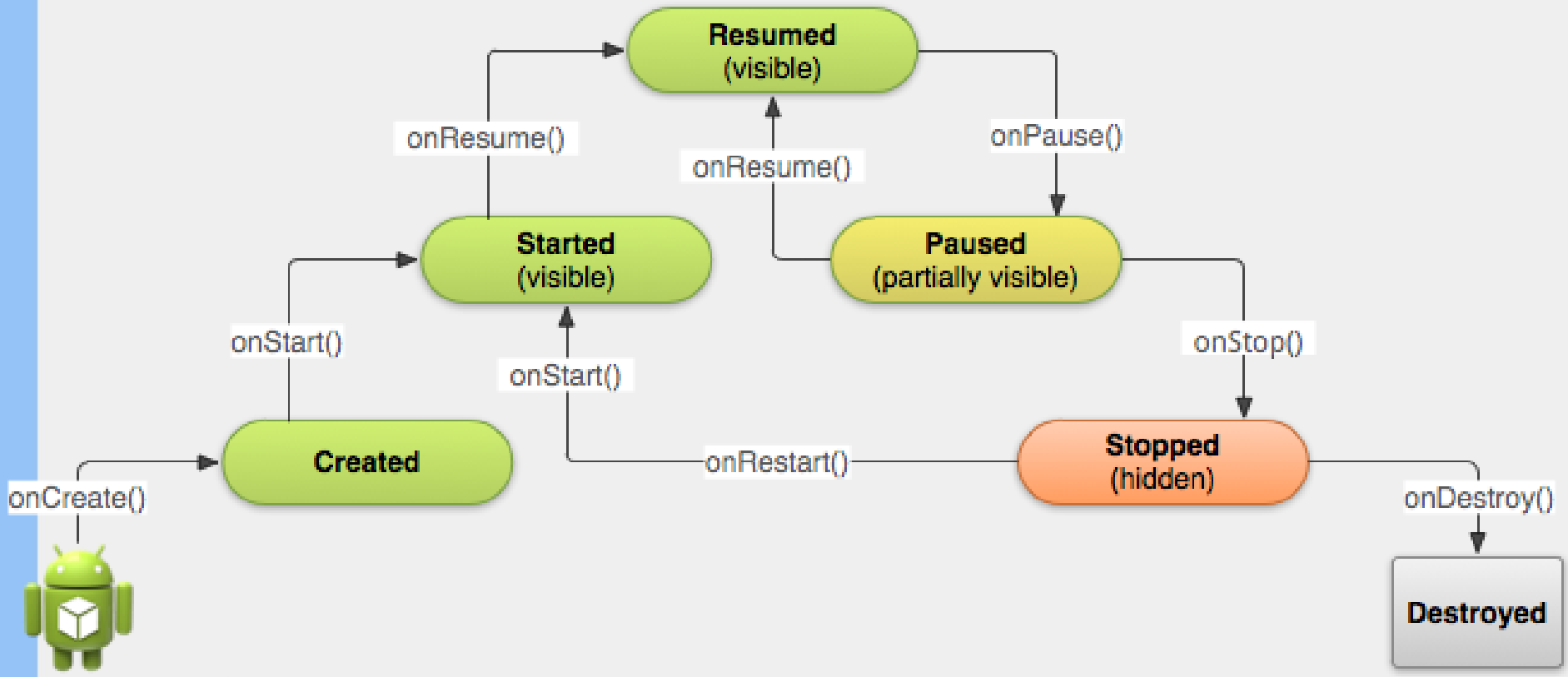
Les applications & activités n'ont pas de contrôle sur leur durée de vie

Trois priorités pour une application

- Critique (Processus actif)
- Haute (Processus visible ou en cours d'exécution)
- Basse (Processus d'arrière plan ou processus vide)

La priorité d'une application influe sur l'ordre dans lequel les activitées seront tuées (en cas de ressources faibles par exemple)

Cycle de vie d'une application



Passage d'informations entre activités

- Se fait via un objet `Intent`
- Pour des données simple, via les fonctions
  - `getTypeExtra()`
  - `setTypeExtra()`
- Type étant un type standard, comme « `int` » ou « `short` », ainsi que « `String` ».

Passage d'informations entre activités

Pour des objets complexes via les fonctions

- `getExtra()`
- `putExtra()`
- Le second paramètre est soit un objet « Serializable » ou un objet « Parcelable »
  - Serializable : Encode l'ensemble de l'objet. Pas très performant.
  - Parcelable : Encode seulement les éléments nécessaires à l'objet. Mieux, mais demande l'écriture de code supplémentaire pour savoir quoi conserver

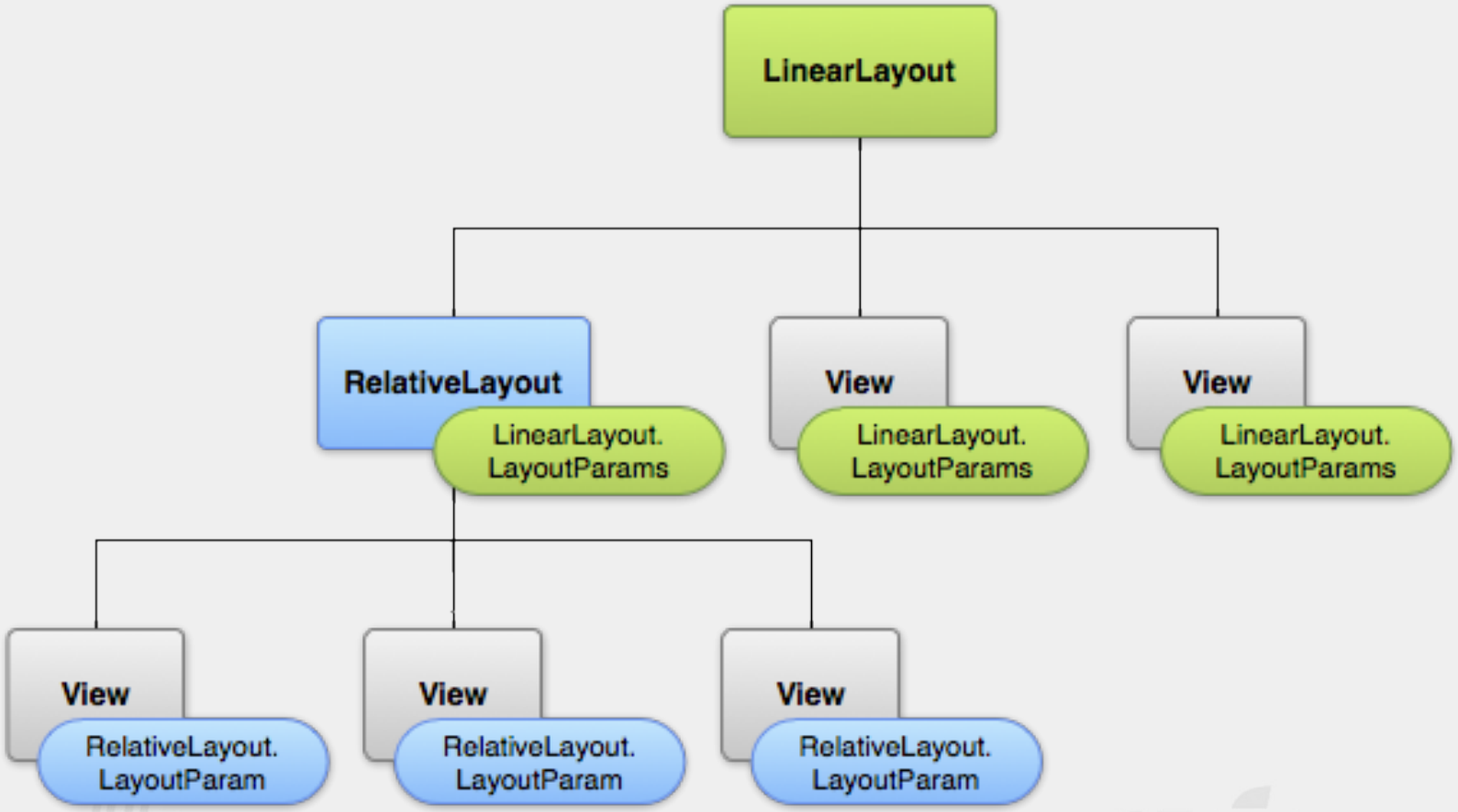
La création d'interfaces (UI)

- Deux méthodes de conception
  - Procédurale (Via un code java qui va ajouter tous les éléments)
  - Déclarative (Via un format spécifique XML)
- Les deux méthodes peuvent se combiner
- Seconde méthode à privilégier
  - Code plus court
  - Plus facile à débugger
- Méthode procédurale parfois nécessaire (pour des modifications dynamiques de l'interface)

La création d'interfaces (UI)

- Tous les objets affichés à l'écran héritent de la classe **`View`**
- Les objets héritant de la classe **`ViewGroup`** permettent de regrouper différents objets héritant de **`View`** (Exemple : les Layouts)
- Il est possible de créer ses propres éléments graphiques en ayant une classe qui hérite de `View`
- La racine de tous les éléments est forcément un `ViewGroup`

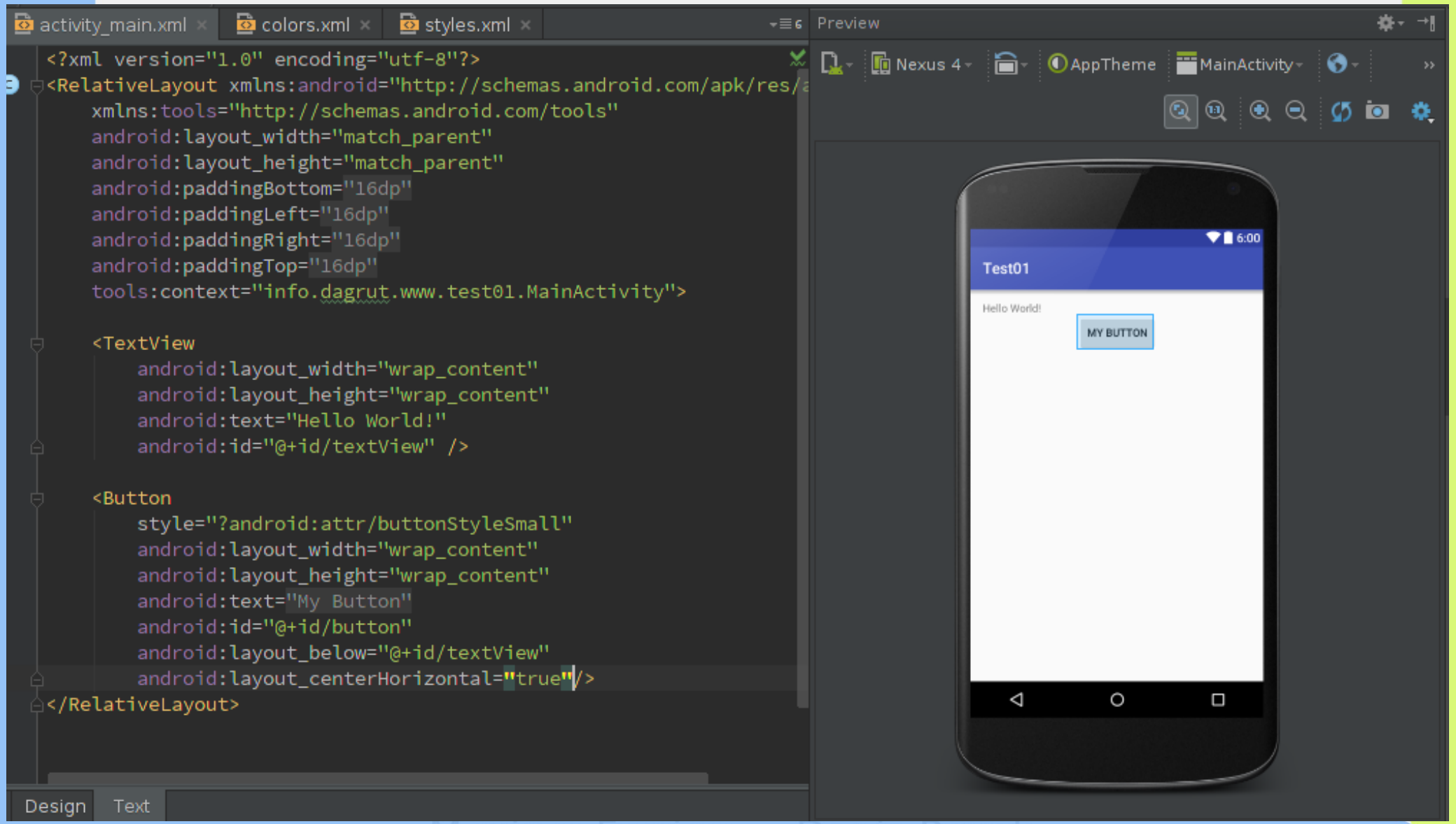
La création d'interfaces (UI)



La création d'interfaces (UI)

- La création déclarative d'interfaces se fait par un fichier Layout
- Le fichier est au format XML
- Android Studio permet l'édition graphique et la prévisualisation de l'interface sur plusieurs périphériques et plusieurs paramètres de configuration (versions d'Android, sens d'écriture suivant la langue, etc.)
- Exemple dans : CM-sources/00-CreationUI

La création d'interfaces (UI)



La création d'interfaces (UI)

- Du côté du code, chaque Activité peut définir le Layout à utiliser via la fonction **`setContentView`**
- Cet appel se fait généralement au début de la fonction **`onCreate()`** d'une activité
- Exemple
  - `setContentView(R.layout.main_layout);`

La création d'interfaces (UI)

Les éléments graphiques ont plusieurs attributs, certains hérités des classes parentes (comme `width` et `height` qui sont hérités de **`View`**) ou spécifiques à l'élement (comme `textColor` sur les éléments de type **`TextView`**)

La création d'interfaces (UI)

L'attribut id permet d'identifier chaque élément de manière unique

- Il est normalement généré automatiquement par l'IDE
- Il doit être unique dans tout le projet
- Il est référencé par un entier de manière globale, et peut ainsi être récupéré dans l'ensemble du projet si besoin.
- Pour créer un id dans le fichier xml :
  `android:id="@+id/mon_bouton"`
- Pour référencer un ID depuis le fichier xml :
  `android:layout_below="@id/mon_texte"`

Parenthèse : Référencement des ressources

Dans Android les ressources sont référencées de manière unique dans le projet

Elles sont accessibles dans la variable globale « R ».

Exemples de ressources

- Couleur (`R.color`)
- Texte & traductions (`R.string`)
- Layout (`R.layout`)
- Menu (`R.menu`)
- Thème (`R.style`)
- ...

Parenthèse : Référencement des ressources

Pour y accéder dans un fichier XML, on préfixe le nom de la ressource par un arobase. Exemples :

- `@color/dark_red`
- `@string/text_of_my_button`

Pour y accéder dans un code java, on utilise la variable R. Exemples :

- `R.layout.page_principale`
- `R.color.dark_red`
- `R.string.app_name`

Parenthèse : Référencement des ressources

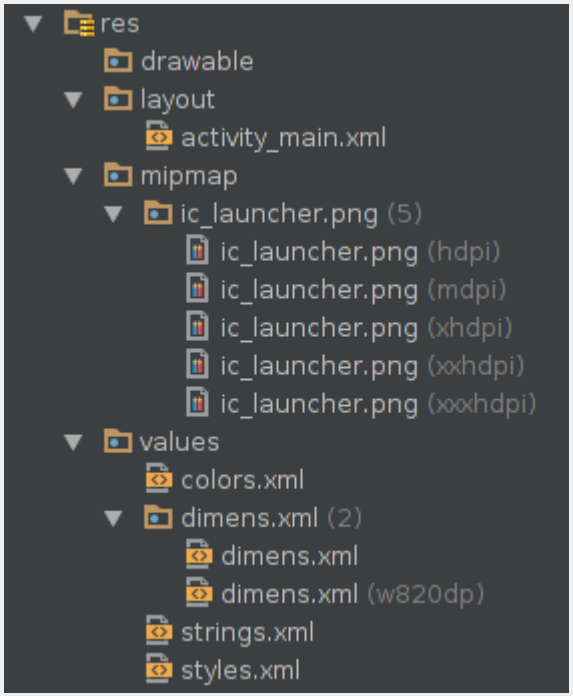
Cette valeur est soit utilisée pour extraire des éléments, soit fournie telle quelle à d'autres fonctions.

ATTENTION ! Les valeurs contenues dans R sont des ENTIERS. Les véritables valeurs sont indexés par les outils de compilation d'Android.

Exemple :

```
this.getString(R.string.titre_page);
```

Parenthèse : Référencement des ressources



Les différents Layouts existants

`FrameLayout`

- Permet de réserver de l'espace pour un unique élément
- L'espace réservé correspond à l'élément le plus grand contenu dans le Layout

`LinearLayout` (horizontal & vertical)

- Permet de lister des éléments, horizontalement ou verticalement
- Espace attribué pour chaque élément proportionnel aux autres via l'attribut `weight`

Les différents Layouts existants

`TableLayout`

- Permet d'organiser les éléments sous forme de tableau
- Ses enfants sont nécessairement des Layouts du type `TableRow`

`GridLayout`

- Très proche de TableLayout, mais plus polyvalent (plus récent aussi). À éviter si on veut supporter d'anciennes versions d'Android.

Les différents Layouts existants

`RelativeLayout`

- Permet de placer les éléments les uns par rapport aux autres, et par rapport au Layout lui même.

Les attributs hérités des Layouts

- Un élément contenu dans un layout peut hériter de paramètres.
- Ceux-ci permettent généralement de placer l'élément dans le layout.
- Par exemple dans un **`RelativeLayout`**, il est possible d'avoir un attribut xml **`android:layout_centerHorizontal`** permettant de centrer l'élément dans le layout.

Dimensions des Layouts & des Vues

Chaque layout et chaque vue a une hauteur et une largeur

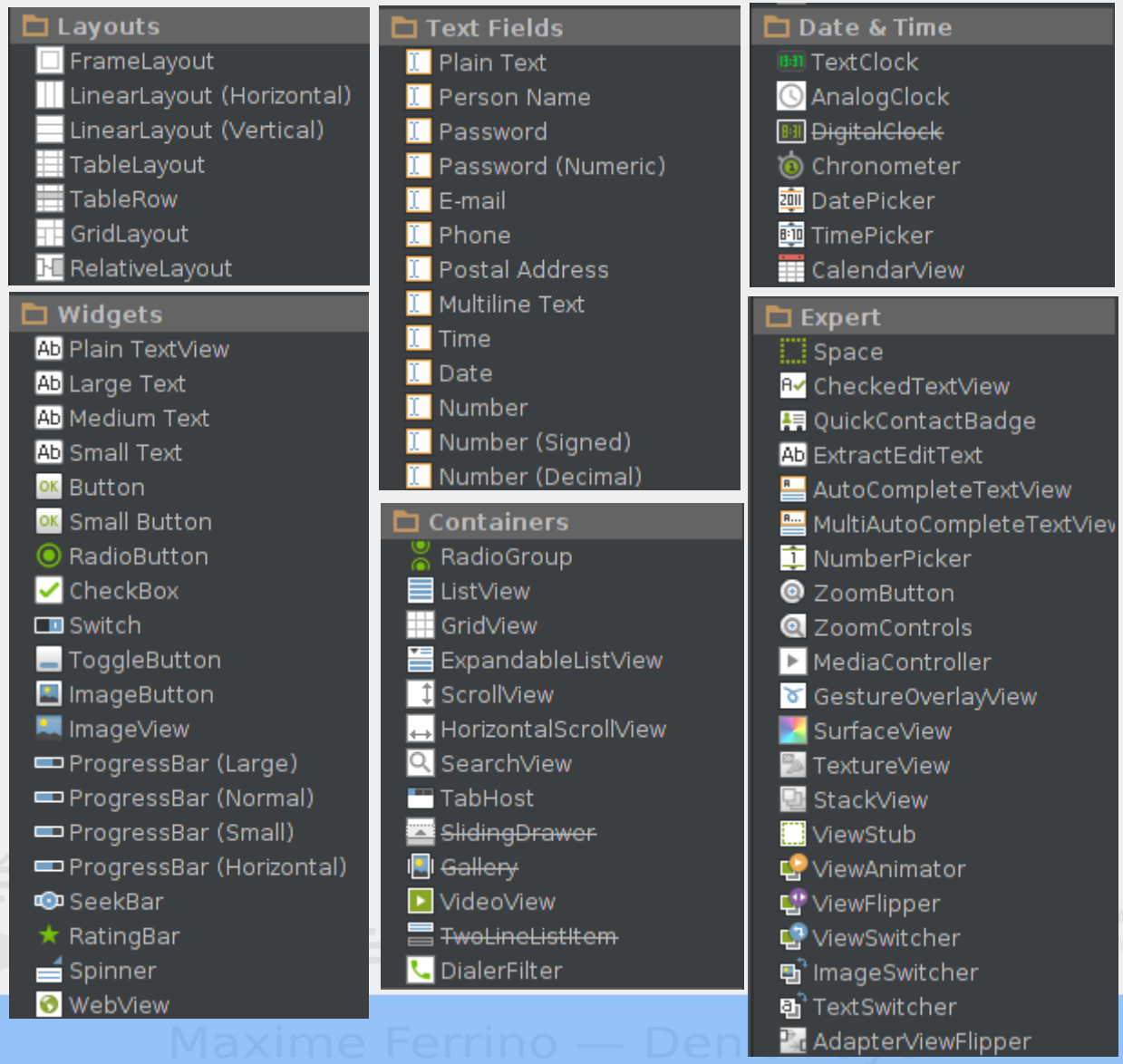
- Chaque vue se dimensionne elle même
- La vue parente a cependant le dernier mot sur ses dimensions

Les valeurs fixes sont possibles

Il est possible de limiter les dimensions à l'espace nécessaire pour le contenu via le mot clé **`wrap_content`**

Il est possible d'occuper autant d'espace que l'élément parent via le mot clé **`fill_parent`**

Les éléments graphiques prédéfinis



Les Menus

Trois types de menus

- Option Menu
  - Icon menu
  - Expended menu
- Context Menu
- Submenu

Les Option Menu

Principal ensemble d'items de menu

S'affiche quand on presse la touche « menu »

Composé de deux types d'items

Icon Menu

- 6 items maximum
- Seuls ces items peuvent avoir une icône

Expanded Menu

- Items visibles via le bouton "More" pour afficher plus d'items si besoin
- N'existent pas si il y a moins de 6 items au total

Les Option Menu : Implémentation

Pour l'implémenter, il faut surcharger la méthode **`onCreateOptionsMenu`** de l'Activité

Deux méthodes à partir de ce moment

- Utiliser un fichier xml contenant le menu
- Ajouter les éléments au menu via la méthode `add()`
  - Cette méthode retourne un objet représentant l'entrée dans le menu
  - Il faut appeler ses méthodes pour modifier cette entrée
- **Note :** Suivant la version d'Android utilisée, l'affichage et l'utilisation de ces menus peuvent varier

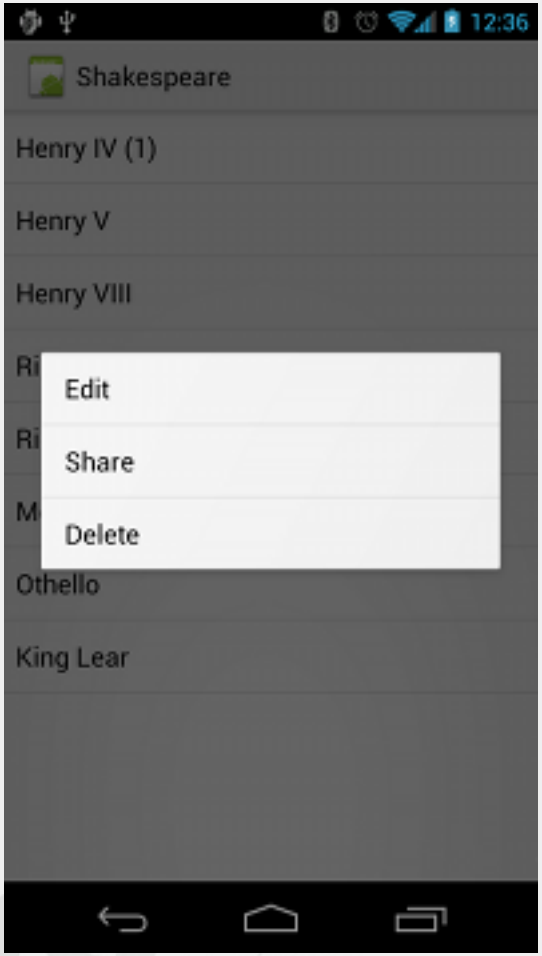
Les Option Menu : Implémentation

- Afin d'utiliser un fichier XML référencé comme « `mon_menu` », il faut surcharger la méthode `onCreateOptionsMenu` ainsi :

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.mon_menu, menu);
    return true;
}
```

Les Menus Contextuels (Context Menu)

- S'affiche lors de la pression longue sur un élément
- Équivalent du clic droit sur PC
- Les items ne supportent pas les icônes et les raccourcis (pas nativement, du moins)



Les Menus Contextuels : Implémentation

Il faut surcharger les deux fonctions

- **`onCreateContextMenu`**
  - Les entrées s'ajoutent via la méthode add(), comme pour les Option Menu
- **`onContextItemSelected`**
  - Dans cette fonction, il faut appeler la méthode `getMenuInfo()` pour récupérer les informations du menu cliqué

Il faut appeler **`RegisterForContextMenu`** pour activer le menu contextuel

Les sous-menus

- Peuvent être ajoutés à tout types de menus
- S'ajoute via les méthodes `addSubMenu()` puis `add()`
  - L'ajout est similaire à l'ajout manuel d'entrées dans les Option Menus

Création d'un menu en XML

Les fichiers xml doivent être dans le dossier res/menu du projet

Peut contenir les balises `<menu>`, `<group>` et `<item>`

Une balise `<menu>` est forcément à la racine d'un menu.

`<menu>` peut avoir `<item>` et `<group>` comme enfant

`<group>` peut avoir `<item>` comme enfant

`<item>` peut avoir `<menu>` comme enfant

Cela permet la création de sous-menus

Création d'un menu en XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/file" android:title="Fichier" >
        <menu>
            <item android:id="@+id/nouveau"
                android:title="Nouveau" />
            <item android:id="@+id/ouvrir"
                android:title="Ouvrir..." />
        </menu>
    </item>
</menu>
```

Manipulation de données : les Adapters

- C'est l'intermédiaire entre une source de données et une vue de ces données
- C'est un élément qu'il faut implémenter
- Deux méthodes pour implémenter notre `Adapter` :
  - Implémenter notre propre adapter héritant de la classe « Adapter »
  - Utiliser une classe existante, par exemple « `ArrayAdapter` » (qui est une classe abstraite)
- La 2eme option est préférable vu qu'elle va limiter le code que nous devront écrire
- Plusieurs variantes existent, pour différentes utilisations

Manipulation de données : les Adapters

Exemples

- On a une liste d'éléments dans un tableau
- On veut afficher ces éléments dans une `ListView`
- On va donc naturellement se tourner vers un `ArrayAdapter`
- Exemple disponible dans CM-sources/02-ListView

- On a un champ de recherche avec des suggestions
- Toutes les entrées sont stockées dans une base `SQLite`
- On va alors utiliser un `SpinnerAdapter`

Les graphismes (2D/3D/Animations)

Pour la 2D, deux packages existent

- android.graphics.drawable
- android.view.animation

Les dessins 2D peuvent aussi se faire dans le code, ou dans un fichier xml (à placer dans res/drawable)

De même pour les animations simples

Pour la 3D, il faut passer par OpenGL (Plus précisément OpenGL ES, spécialisé pour les systèmes embarqués)

La gestion des événements graphiques

Plusieurs méthodes pour gérer les événements

Méthode 1

- Pour chaque élément, créer une classe héritant du type souhaité, et surcharger la méthode `onClick` de la classe
- Permet d'accéder à des événements supplémentaires (dits « Event Handlers »)
- Ex: `onKeyDown`, `onKeyUp`, ...
- Lourd et complexe à mettre en place
- Ne sera pas abordé dans ce cours

La gestion des événements graphiques

Méthode 2

- Définir une classe (et l'instancier) permettant de capturer les événements des éléments voulus
- Transmettre cette instance aux Views voulues via les méthodes `setListener()`. (Par exemple `setOnClickListener()`).

Cette méthode **ne permet pas** d'accéder aux « Event Handlers »

Il est aussi possible de faire en sorte que la classe `Activity` implémente l'interface associée à l'événement choisi afin de transmettre l'activité directement a là/aux view(s) concernée(s).

- Cela permet d'éviter quelques lignes de code supplémentaire
- Cela rend l'accès a l'activité plus simple (et même direct) depuis la callback de l'événement
- Cela complique en revanche le traitement des événements si ils sont capturés de cette manière sur plusieurs views

Les Gestures

- Le terme « **Gesture** » regroupe tout un ensemble d'événement, dont principalement :
  - **Down** : Contact avec l'écran (similaire au click)
  - **Fling** : Mouvement de « jet » d'un élément (typiquement à la fin de l'événement **Scroll**)
  - **LongPress** : Pression longue sur l'écran
  - **Scroll** : Défilement
  - **Scale** : Étirement ou rétrécissement
- Permet de faire des transitions plus souples, et de rendre l'interface plus intuitive
- Cas spécifique des événements graphiques
  - Fonctionne de la *même manière*
  - Certains événements peuvent ainsi être redondants

Les Gestures

- Plusieurs gestures ne sont pas supportées nativement par Android mais peuvent être implémentées
  - Rotation
  - Les gestures combinées (Pression longue suivie d'un drag & drop, par exemple)
- La seule limite des gestures est l'imagination

Les Gestures : Utilisation

- Plusieurs méthodes d'utilisation :
  - Implémenter certaines classes internes de la classe **`GestureDetector`** directement sur une **Activité**
  - Utiliser la méthode **`setOnTouchListener`** (ou équivalents) d'une `View` pour ne capturer que les événement liés à celle-ci
  - Utiliser la classe **`GestureDetectorCompat`** qui permet de transformer les événements « basiques » en gestures
    - Cette dernière possibilité est à utiliser si d'anciennes versions d'Android doivent être supportées, et demande de télécharger et d'inclure une bibliothèque spécifique

Les Fragments

Les fragments sont des parties d'une interface graphique

- Ils peuvent et doivent être placés dans une activité
- Ils peuvent faire la taille d'une page complète ou au contraire être plusieurs dans une seule activité

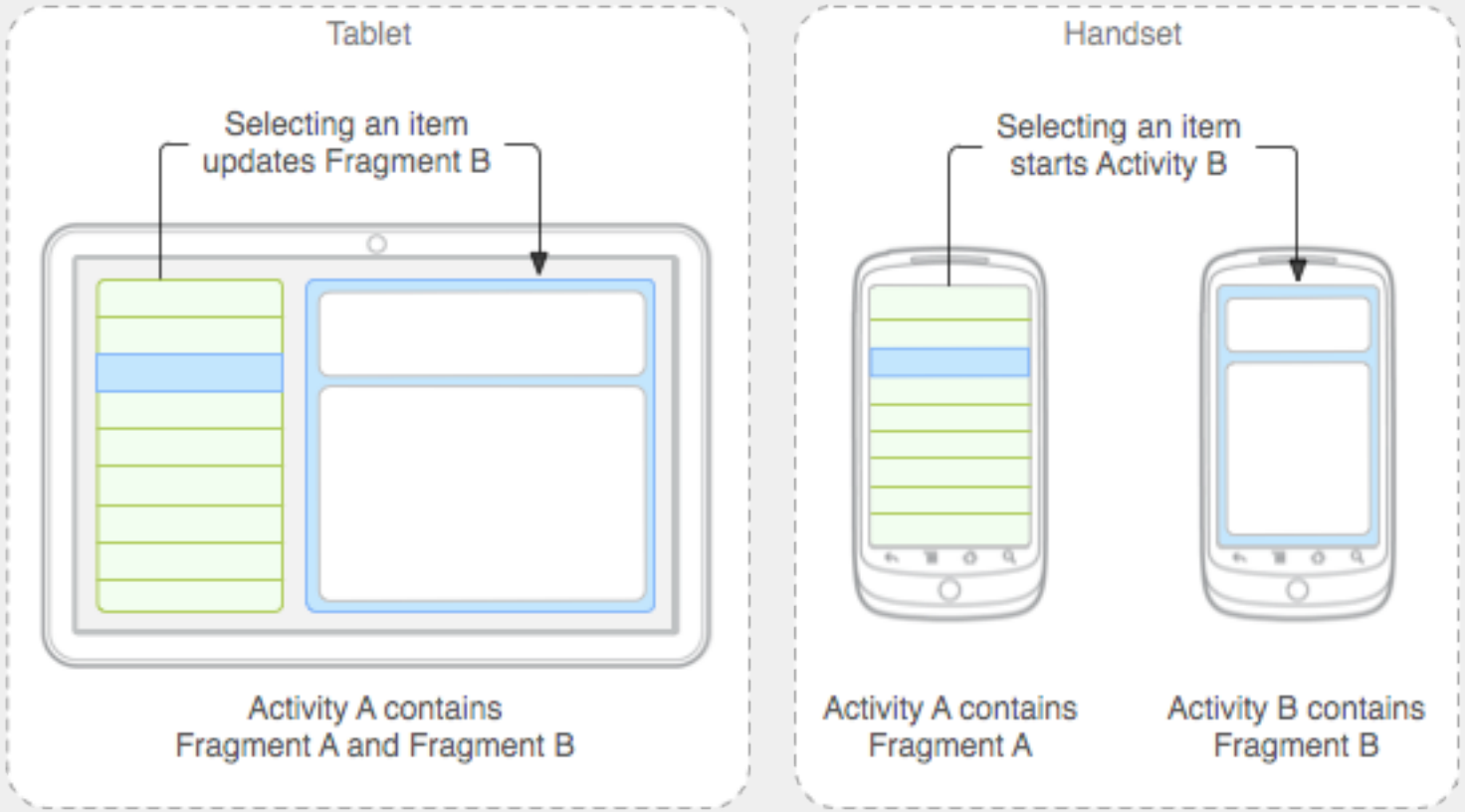
L'exemple le plus courant est le cas de plusieurs panneaux d'affichage, typiquement une liste et les détails d'un élément de cette liste

- Il est ainsi possible d'afficher cela comme deux écrans séparés sur mobile, mais un seul écran divisé sur tablette

Les interactions avec un `Fragment` peuvent être faites via un `FragmentManager`

Ils permettent une grande flexibilité de l'interface, et plusieurs méthodes d'implémentation les utilisants sont possibles sont nombreuses

Les Fragments



Les Fragments

La création d'un Fragment peut se faire soit via le code, soit directement dans le fichier XML représentant une activité

Plus pratique à utiliser lorsque les scénarios d'affichage sont déjà définis à la création de l'application

Attention : même lorsque les Fragments sont définis dans le fichier XML, ils doivent quand même avoir une classe associée!

Les Fragments

Tout comme les activités, il est possible d'ajouter d'autres Fragments à une interface, puis de revenir en arrière en utilisant une pile de retour (backstack) gérée par le système

- Le passage entre Fragments doit alors se faire via la classe `FragmentTransition`

Les Fragments permettent des choses plus poussées, notamment la conservation de données lors du rechargement d'une application (dans le cas ou le Fragment n'est pas lié à une Activité, ce qui est un cas très spécifique), mais cela sort du cadre de ce cours

Les Broadcast receivers

Permettent de recevoir des messages émis sur l'ensemble du système

Exemples :

- Changement d'état de la batterie
- Changement du réglage de l'heure
- Extinction de l'appareil

Les Broadcast receivers

Des messages plus spécifiques peuvent aussi être émis à une application particulière

Exemples :

- Une nouvelle version de l'application a été installée
- Un élément copié viens d'être collé
- Un appareil **bluetooth** viens de se connecter à notre application

Les Broadcast receivers : Utilisation

L'émission de messages peut se faire avec des Intents via la méthode **`sendBroadcast`** de la classe **`Context`**

Pour créer un **Broadcast Receiver**, il faut avoir une classe héritant de la classe **`BroadcastReceiver`**

Dans cette classe il faut surcharger la méthode **`onReceive`**

- Cette méthode reçoit une instance de **`Context`** et une autre de **`Intent`** qui ont été utilisés par l'émetteur de l'événement

Ce récepteur doit aussi être déclaré dans le fichier **Manifest**, et doit lister les messages qu'il peut recevoir

Les services

- Les services permettent principalement la création de tâches de fond, qui vont être exécutées alors que l'application même ne sera pas lancée.
- Une deuxième utilisation, plus rare dans les applications tierces est la création d'une fonctionnalité qui sera accessible à d'autres processus

Exemple :

- Le scan de périphériques bluetooth est assuré par un service dans une autre application
- Ainsi, scanner des périphériques lance indirectement une autre application

Les Notifications

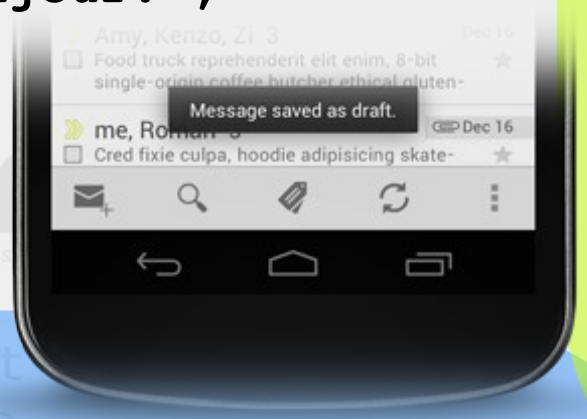
Trois méthodes de notifications sur Android

- Les Toasts
- Les Notifications
- Les Notifications Push

Les Toasts

Un **`Toast`** permet d'afficher une information brève et pas nécessairement importante à l'utilisateur, généralement en bas de l'écran

- Peut être utilisé pour débugger une application plutôt que les *logs* mais pas forcément conseillé
- Il est possible d'étendre la classe Toast afin de leur donner un affichage différent
- Exemple courant : Notification de connexion/déconnexion à un réseau *wifi*
- **Exemple de code**
  - **`Toast toast = Toast.makeText(this, "Bonjour!", Toast.LENGTH_SHORT);`**
  - **`toast.show();`**
- Ici ***`this`*** est une instance de **`Context`**



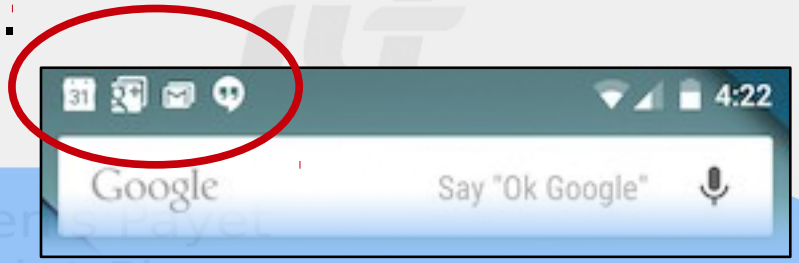
Les Notifications

- Sert à alerter l'utilisateur sur un événement
- Peuvent être gérées par les applications via un **`Intent`** spécifique : un **`PendingIntent`**

Sur *Android*, les Notifications peuvent avoir différentes priorités et différentes catégories

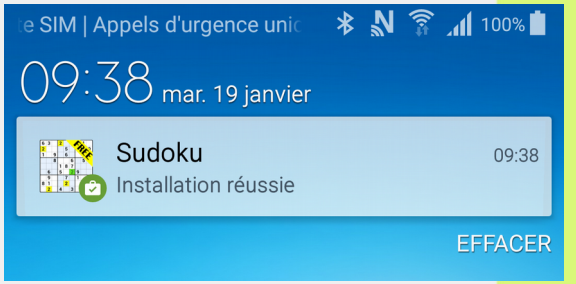
Par exemple, un appel téléphonique va afficher une notification avec une haute priorité et sera dans la catégorie **`CATEGORY_CALL`**.

Suivant la priorité et la configuration de la notification, elle peut aussi générer d'autres choses, comme le clignottement d'une *led*, une *sonnerie*, une *vibration*, etc.



Les Notifications : Utilisation

Pour créer une notification, il faut :



- Créer une instance de la classe **`NotificationManager`**
  - Cette classe gère l'affichage des notifications une fois qu'elles ont été créées
  - Elle peut aussi cacher une notification que l'on a affichée précédemment
- Créer une instance de la classe **`Notification`**
  - Représentation de la notification (contenu, événements, ...)

Les Notifications : Utilisation

- Créer un **`PendingIntent`**
  - Permet de spécifier quelle Activité prendra en charge les actions sur cette notifications (comme un *clic*, ou un rejet de la notification)
- Appeler la méthode **`setLatestEventInfo`** sur l'instance de **`Notification`**
  - Permet de lier le **`PendingIntent`** à la notification
  - Permet de configurer les actions et le comportement de la notification
- Appeler la méthode **`notify()`** sur l'instance de **`NotificationManager`** pour émettre la notification

Les Notifications : Utilisation

- À partir de *Android 5.0*, une autre méthode existe
  - Il est possible de créer un objet **`Notification.Builder()`**
  - De configurer ledit objet (titre, image, etc.)
  - D'appeler la méthode **`build()`** pour générer un objet **`Notification`**
- Une notification contient au moins
  - Une icône (qui sera par défaut celle de l'application qui émet la notification)
  - Un timestamp (l'heure a laquelle la notification a été envoyée)
  - Un titre et un texte additionnel

Les Notifications : Utilisation

- **Changements et améliorations notables dans *Android 5.0***
  - Diverses améliorations, tant au niveau visuel que de la gestion des notifications
  - Possibilité de les afficher sur la page de login, ou au contraire de ne pas les y afficher si elles contiennent des informations sensibles
  - Fermer une notification émise en réseau peut la ferme sur tous les périphériques ou elle est affichée
  - Possibilité d'afficher un contenu étendu dans la notification, comme une image par exemple

Les Widgets

- Les widgets sont les éléments que l'on peut afficher sur les pages principales du téléphone
  - En anglais, Widget signifie "Gadget, appareil, truc, bidule, machin"
- Exemples courants
  - La météo
  - La date et l'heure
  - Une barre de recherche
  - ...

- Permettent l'affichage rapide d'informations, ainsi que généralement le lancement d'une application

Attention : ces widgets n'ont rien à voir avec ceux que nous avons vu dans les éléments graphiques d'Android!

Les Widgets : Utilisation

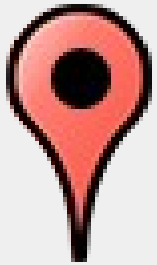
Pour créer un widget, il faut :

- Définir un *`Layout`* pour le *`widget`*
- Créer son fichier de configuration contenant ses propriétés (dimensions, fréquence de mise à jour, ...)
  - **Note :** La fréquence minimale de mise à jour est de 30 minutes. Il n'est donc pas possible de rafraîchir l'affichage plus rapidement que cela.
- Créer un **`BroadcastReceiver`**
  - Les *widgets* sont en effet des **`BroadcastReceivers`**
  - Ils réagissent à la notification **`APPWIDGET_UPDATE`**
- Déclarer le *`widget`* dans le **Manifest**

Les Widgets : Utilisation

- Pour mettre à jour les informations d'un widget, deux méthodes sont possibles
  - L'utilisation d'un fichier *XML* de configuration pour définir la vitesse de rafraîchissement
  - La création d'un Intent qui reçoit les événements de l'**`AlarmManager`**
- La première méthode est celle qui est le plus souvent utilisée
- Il est possible de permettre à l'utilisateur de redimensionner le widget
  - Cela permet d'afficher plus ou moins d'informations, selon sa convenance
- **Note :** Une classe **`AppWidgetProvider`** existante peut être utilisée en lieu et place de la classe **`BroadcastReceiver`**. Elle permet une gestion plus aisée des événements pour les *Widgets*

La géolocalisation



La géolocalisation se fait via le package **`android.location`**

Permet de géolocaliser l'appareil

Requiert une ou plusieurs permissions spéciales à ajouter dans le fichier Manifest pour pouvoir l'utiliser

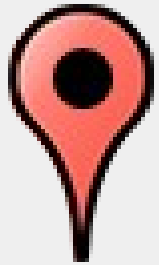
- **`ACCESS_COARSE_LOCATION`**
- **`ACCESS_FINE_LOCATION`**
- Et d'autres suivant les besoins spécifiques de l'application

La gestion de la géolocalisation est gérée par la classe **`LocationManager`**

Note : Cette classe ne s'instancie pas directement (avec **`new`**)

Pour récupérer une instance de cette classe, il faut appeler **`getSystemService(Context.LOCATION_SERVICE)`** sur une instance de **`Context`**

La géolocalisation



Cette classe permet

- D'obtenir les sources desquelles ont peut récupérer la géolocalisation de l'appareil
- D'obtenir la géolocalisation du téléphone suivant une ou plusieurs sources
- D'être alerté lors de l'entrée dans une zone que l'on aura préalablement définie
- Les sources de géolocalisation peuvent être
  - Les services de google, qui permettent une localisation basique suivant l'adresse IP
  - Le GPS
  - D'autres applications qui demandent une géolocalisation (la demande est donc passive ici), et ne retournera de résultats que si d'autres applications sont lancées et utilisent ce service

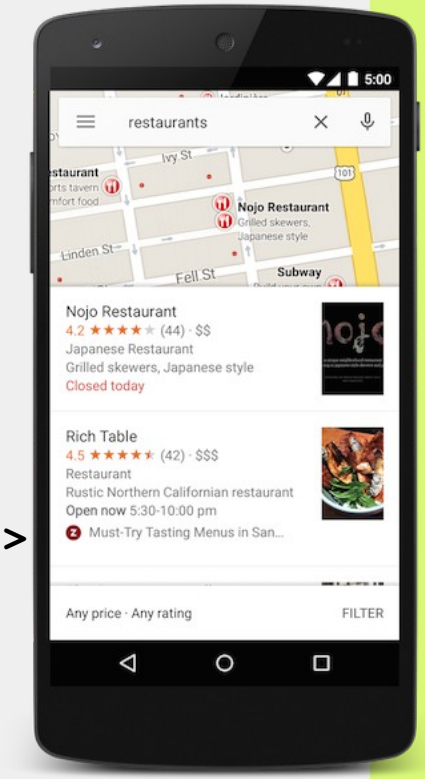
La géolocalisation : L'intégration de Google Maps

S'utilise avec la classe **`MapView`** disponible dans : **`com.google.android.maps`**

Permet l'affichage d'une carte fournie par les services de *Google*

Requiert plusieurs choses

- L'ajout d'une bibliothèque supplémentaire
- L'ajout d'une bibliothèque au fichier manifest
  - **`<uses-library android:name="com.google.android.maps" />`**
- L'ajout d'une permission supplémentaire
  - **`<uses-permission android:name="android.permission.INTERNET" />`**
- Un compte Google
- Une clé d'authentification associée à ce compte
- Une classe héritant de `MapActivity` et permettant la gestion de l'affichage



Le multithreading

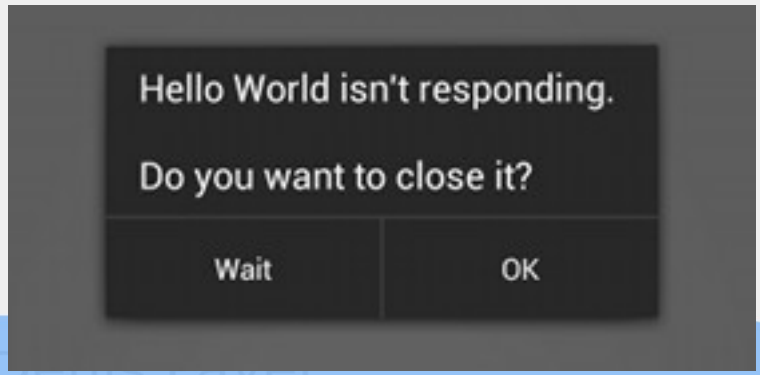
L'API Android permet la création de **« *Threads* »**

- Toute application a **au moins** un thread
- Similaire à un second processus (S'exécutent en parallèle sur plusieurs processeurs)
- Plusieurs Threads dans un même processus (une même application)
- Partage les **mêmes ressources**
  - CPU
  - Mémoire
- Est toujours en lien avec les autres *threads* d'un même processus (peut accéder à leur mémoire)

Le multithreading

Permet de :

- Décharger le thread principal
  - Sur Android (et sur mobile en général), le thread principal est celui de l'interface utilisateur
  - Cela implique que le charger ou le bloquer peut complètement geler l'interface utilisateur
- Faire des calculs intensifs en utilisant tous les cœurs du processeur
- Gérer des tâches qui prennent du temps
  - Principalement les tâches liées à la manipulation de fichiers ou de flux réseau
- Etc.



Le multithreading

Plusieurs méthodes pour cela :

- L'utilisation de la classe « `Thread` »
  - Fonctions bas niveau, qui ne sont pas entièrement codées en java
  - Intégrée dans le langage java
  - Les deux autres classes utilisent celle-ci
  - Utilisation possible mais généralement déconseillée, pour des questions de simplicité

Le multithreading

- L'utilisation de la classe « `Handler` »
  - Fonctions codées entièrement en java
  - Intégrée dans le framework java.
  - Permet de créer des threads plus facilement

Le multithreading

- L'utilisation de la classe « `AsyncTask` »
  - Fonctions codées entièrement en java
  - Intégrées dans Android, mais pas dans le langage java
  - Permet de simplifier l'utilisation des threads par rapport à la classe `Handler`
  - Généralement conseillée pour un développement sur Android

Le multithreading

Nous préfèreront donc la classe **`AsyncTask`**, d'autant que c'est celle qui est conseillée pour le développement sur *Android*

Les services *Android* utilisent cette classe

Note : *Android* permet d'exécuter du code de votre application dans un autre processus (une autre application), mais il faut pour cela que les deux applications partagent les mêmes certificats, et appartiennent au même utilisateur (Voir la partie Sécurité des Annexes).

La classe AsyncTask

Classe abstraite prenant trois types

- Le type de valeur d'entrée
- Le type de valeur intermédiaire
- Le type de valeur de sortie

Ces classes peuvent être la classe « `Void` » permettant de définir une absence de valeurs.

Pour l'utiliser, il faut créer une classe qui en hérite

Les opérations à effectuer doivent être mises dans la méthode `doInBackground()`

Exemple le plus simple d'utilisation :

- ```
  public class MyTask extends AsyncTask<Void, Void, Void> { /* Votre code */ }
  ```
  ```

La classe AsyncTask

D'autres méthodes peuvent également être surchargées pour effectuer des traitements avant et après l'exécution de cette méthode.

- **`onPreExecute()`**
- **`onPostExecute()`**
- **`onCancelled()`**
- **`onProgressUpdate()`**

Les connexions réseau

- Java permet l'utilisation de sockets **TCP** et **UDP** brutes
- Le développeur doit alors s'occuper de toute la partie encodage et décodage des données
- Se fait via les classes **`java.net.Socket`** et **`java.net.DatagramSocket`**

Java permet aussi la création de requêtes http via la classe **`java.net.HttpURLConnection`**

Les connexions réseau

Android ajoutait une classe permettant de faire des requêtes http via la classe **`HttpClient`**, mais cette *API* a été dépréciée et est maintenant supprimée.

En revanche, *Android* apporte toujours des fonctionnalités supplémentaires permettant la gestion d'un cache **http** ou des certificats **SSL** sur une connexion établie avec la classe **`HttpURLConnection`**

La classe HttpURLConnection

Ne peut être instanciée directement

- Une instance peut être retournée via la méthode `openConnection()` de la classe `java.net.URL`.

La lecture des données se fait ensuite via la manipulation du flux de retour fourni par la méthode `getInputStream()`

Attention : L'utilisation de cette classe nécessite l'utilisation d'un thread différent du thread principal car l'appel à `getInputStream()` bloque le thread actuel jusqu'à avoir une réponse du serveur.

Le stockage de données

Sur *Android* plusieurs méthodes de stockage existent :

- Les préférences
- Les fichiers
- Les bases de données
- Le stockage en réseau

Par défaut, toutes les données stockées dans l'application lui sont propres et ne sont pas accessibles par d'autres applications, à moins de mettre en place un **Content Provider**

Les préférences

Mécanisme simple permettant de stocker idéalement des préférences

Limité en terme de taille : On ne peut pas y stocker des fichiers entiers (c'est du moins déconseillé)

Le stockage se fait sous forme de paires clé/valeur

Rend aisé le partage de paramètres avec d'autres composants de l'application

Les préférences : Utilisation

Une instance de la classe **`android.content.SharedPreferences`** est retournée à l'appel de la méthode **`getSharedPreferences()`** de la classe **`Context`**

L'édition se fait via la classe **`SharedPreferences.Editor`**

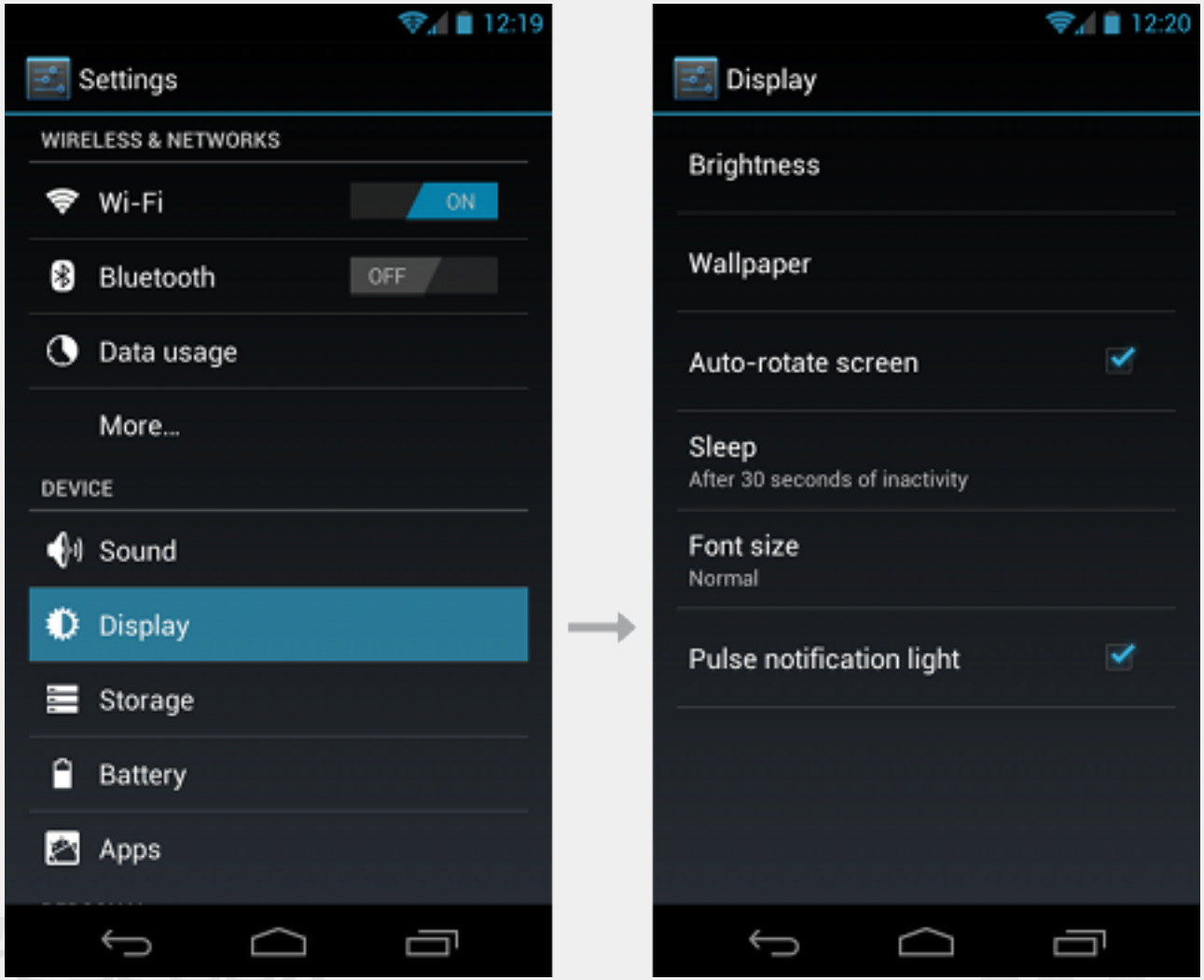
Une instance de cette classe est retournée par l'appel à la méthode **`edit()`** de **`SharedPreferences`**

À la fin de l'édition des préférences, la méthode **`commit()`** de la classe d'édition doit être appelée pour finaliser l'écriture de ces préférences.

Les préférences : Utilisation

- Il est aussi possible de permettre à l'utilisateur de modifier directement une partie ou la totalité de ces préférences
- Cela se fait via l'utilisation d'une activité **`PreferenceActivity`** et de la classe **`Preference`**
- Le design de cette interface peut aussi passer par un fichier XML (la racine de ce fichier est alors une balise **`PreferenceScreen`**)

Les préférences : Exemple



On notera aussi ici l'utilisation de `Fragments`

Le stockage de données : Les fichiers

Plusieurs endroits possibles

- Sur le support de stockage de l'appareil (là ou sont installées les applications)
- Sur un support de stockage amovible (typiquement une clé *USB* ou une carte *MicroSD*) si présent

Sur *Android*, les classes **java standard** pour la manipulation de fichiers fonctionnent aussi (**`java.io.File`**, **`java.io.FileInputStream`**, **`java.io.FileOutputStream`**, **...**)

On peut noter cependant que l'utilisation de ces fonctions est bridée : On ne peut accéder à tous les dossiers, ce qui inclut les dossiers d'autres applications

Pour le stockage de paramètres et de données plus volumineuses, une utilisation des méthodes **`openFileInput`** et **`openFileOutput`** de la classe **`Context`** peut faciliter les choses

Note : La classe **`Activity`** hérite de la classe **`Context`**

Le stockage de données : Les fichiers

Enfin, il est possible d'utiliser des fichiers packagés dans l'application, si on les place dans le dossier **res/raw** ou le dossier **assets**. On peut y accéder par **`R.raw.nom_du_fichier`** ou dans le cas d'un fichier stocké dans **assets** par son nom directement

- Un flux peut alors se récupérer sur le fichier grâce la diverses méthodes de la classe **`AssetManager`**, comme **`open()`** par exemple
- Il n'est parfois pas possible d'utiliser le fichier tel quel (par exemple pour les APIs qui nécessitent un fichier en entrée, et non un flux), il faut alors le copier ailleurs pour pouvoir l'utiliser

Ces fichiers ne sont pas accessibles en écriture : Les données sont donc définies à la création de l'application

Le stockage de données : Les fichiers

Comme vu précédemment, deux dossiers existent pour le stockage de fichiers

- ***/res*** (ou ***/res/*** est suivi d'un autre nom de dossier, faisant office de catégorie)
- ***/assets***

Le premier dossier est généralement préférable à la seconde car celui-ci permet de spécifier des versions différentes des fichiers en fonctions de plusieurs paramètres, telle que :

- La résolution de l'écran
- la densité d'affichage
- L'orientation de l'écran
- La langue
- La version de l'OS
- …

C'est une fonctionnalité souvent utilisée pour les `Fragments`

Le stockage de données : Les fichiers

- Grâce au dossier **res**, l est ainsi possible de définir des fichiers de configuration "**statiques**" qui ne changeront qu'en fonction des paramètres qui vous intéressent, afin d'avoir directement les bonnes valeurs dans le code, et ne pas avoir à gérer de nombreux cas d'un coup.
- Le dossier **assets** ne permet pas cela.

Le stockage de données : Les bases de données

Plusieurs outils existent déjà en java pour la manipulation de bases de données SQL

- Voir les classes **`java.sql.*`**

Android propose une autre api via **`android.database.*`** permettant de manipuler les bases de données, et principalement les bases **SQLite** (fonctions et classes regroupées dans **`android.database.sqlite`**)

Pratique lorsqu'on a des données structurées à stocker

Rappel : SQL veux dire *Structured Query Language*

La téléphonie

Il est possible depuis une application de :

- Passer des appels
- Intercepter des appels entrants
- Envoyer et recevoir des *SMS*
- Accéder aux informations téléphoniques du téléphone, telles que le numéro d'**IMEI**, le numéro de la ligne, l'emplacement du téléphone, etc.

L'accès à ces fonctionnalités demande l'acceptation de permissions spécifiques

Autres APIs notables pour l'accès au matériel

- Capteurs
  - Accéléromètre
  - Température
  - Gravité
  - Gyroscope
  - Lumière
  - Champ magnétique
  - Orientation
  - Pression atmosphérique
  - Proximité
  - Humidité
- Bluetooth
- Bluetooth Low Energy
- NFC
- Lecture et capture Audio/Vidéo/Photo

Outils Android

Maxime Ferrino — Denis Payet
01/2018 — IUT de Saint-Pierre

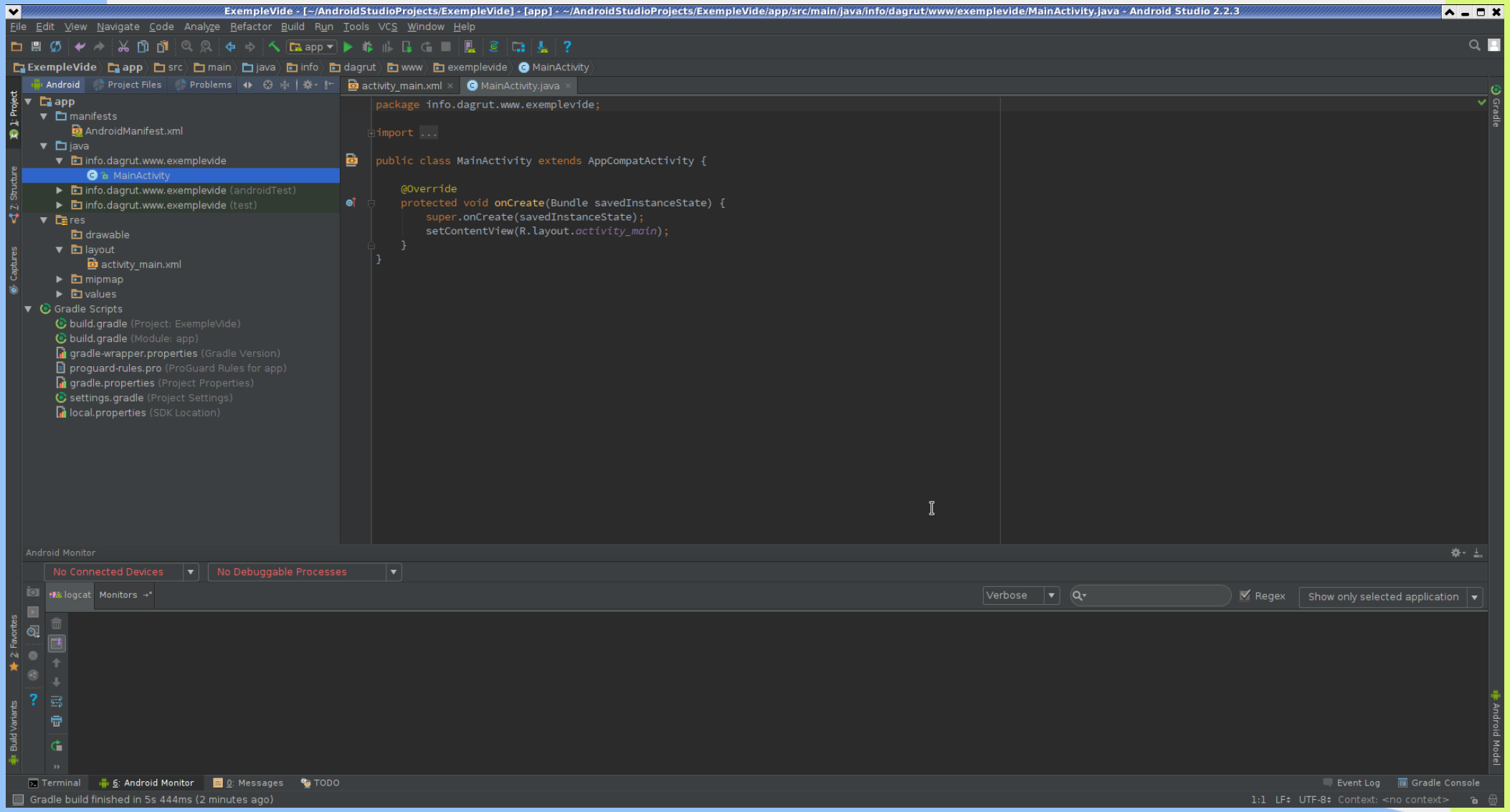
Outils Android

- Google met à disposition :
  - Un IDE nommé **Android Studio**
  - Un SDK et des outils en ligne de commande
- **Android Studio** est récent (première release publique en 2015)
  - Précédemment, il fallait utiliser **Eclipse** (un IDE plus générique, orienté Java et développé en Java) ainsi qu'un plugin ajoutant les fonctionnalités nécessaires au développement sur Android
  - Il est toujours possible de n'utiliser que les outils du SDK si nécessaire, mais cela rendrait le développement plus fastidieux

Installation des outils

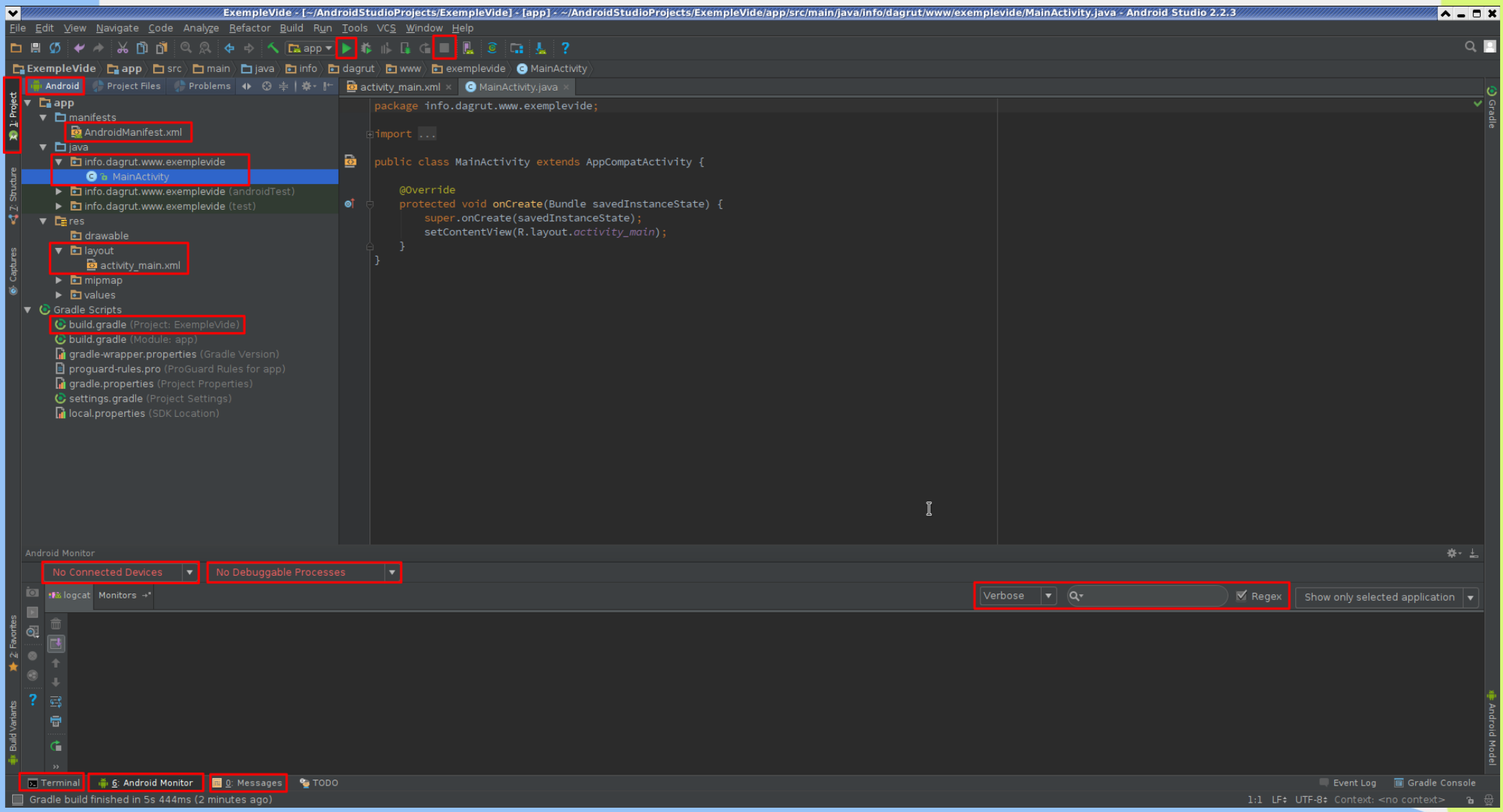
- Installez Java
  http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html
- Rendez vous sur
  https://developer.android.com/sdk/index.html
- Téléchargez le pack **Android Studio** complet
- Les détails de l'installation se trouvent sur
  https://developer.android.com/sdk/installing/index.html?pkg=studio
  - Il faut cliquer sur [Show instructions for all platforms] pour afficher les instructions d'installation

Aperçu de l'interface d'Android Studio



Maxime Ferrino — Denis Payet
01/2018 — IUT de Saint-Pierre

Aperçu de l'interface d'Android Studio



Sources

- Stackoverflow
- Wikipédia
- radleymarx.com
- Oxiane
- Cours de *Denis Payet*
- Cours de *David Turquay*
- vogella.com

Annexes

Annexe 0 : Glossaire d'**Android Studio**

- **adb** : Android Debug Bridge
  Commande permettant la manipulation d'un périphérique Android
- **Gradle** : Suite d'outils indépendante d'Android permettant de simplifier la compilation d'un projet Java
- **Maven** : Outil de gestion de projets développé par Apache, permettant entre autre la gestion automatique des mises à jour de bibliothèques

Annexe 1 : Utilisation de l'objet Log

L'objet Log contient cinq fonctions statiques : **`Log.v()`**, **`Log.d()`**, **`Log.i()`**, **`Log.w()`**, **`Log.e()`**

Il est disponible dans le package **`android.util`**

Permettent de logguer des informations dans les niveaux suivants

- V : Verbose
- D : Debug
- I : Info
- W : Warning
- E : Error

Annexe 1 : Utilisation de l'objet Log

Ces fonctions prennent deux paramètres de type **`String`** :

- Le **TAG**, qui sera ajouté dans la ligne de log, et qui permet de filtrer plus facilement le résultat des logs
- Le **message** en lui même

Utile pour *débugger* une application

Ces niveaux permettent de filtrer les logs d'une application, afin d'afficher plus ou moins d'informations

Note : Les logs d'Android sont très nombreux (plusieurs dizaines lignes de logs par minute alors même que le téléphone est inactif), il est donc important de pouvoir filtrer les logs correctement avec l'utilisation d'un **TAG** explicite et unique.

Annexe 2 : Les unités graphiques sur Android

On en dénombre 6 :

- **px** : Dimensions en **pixels**
- **in** : Dimensions en **inch** (en fonction de la taille réelle de l'écran)
- **mm** : Dimensions en **millimètres** (en fonction de la taille réelle de l'écran)
- **pt** : Dimensions en **points**. 1 point = 1/72eme d'inch
- **dp** ou **dip** :
  - Unité abstraite qui est basée sur la **densité d'affichage** du périphérique
  - Unité relative à un écran de **160dpi**
  - À préférer pour tout les types de dimensions, excepté les polices de caractères
- **sp** :
  - Unité abstraite qui est basée sur la **densité d'affichage** du périphérique ET sur la **taille de police** utilisée sur l'application
  - Unité relative à un écran de **160dpi**
  - À préférer pour les tailles de polices

Annexe 2 : Les unités graphiques sur Android

Les unités **dp/dip/sp** s'adaptent à une densité d'affichage qui n'est pas forcément celle de l'appareil

Android utilise 6 densités d'affichage

- 120dpi
- 160dpi
- 240dpi
- 320dpi
- 480dpi
- 640dpi

Annexe 2 : Les unités graphiques sur Android

Si l'écran de l'appareil ne tombe pas sur une de ces valeurs, il va utiliser **la plus proche**

- Cela peut mener sur des légers problèmes d'affichages, dans le cas ou un appareil se retrouve presque entre deux densités et que l'écart avec la densité utilisée par l'OS est donc assez éloignée de la densité réelle.

Sur des interfaces complexes, c'est un détail à prendre en compte

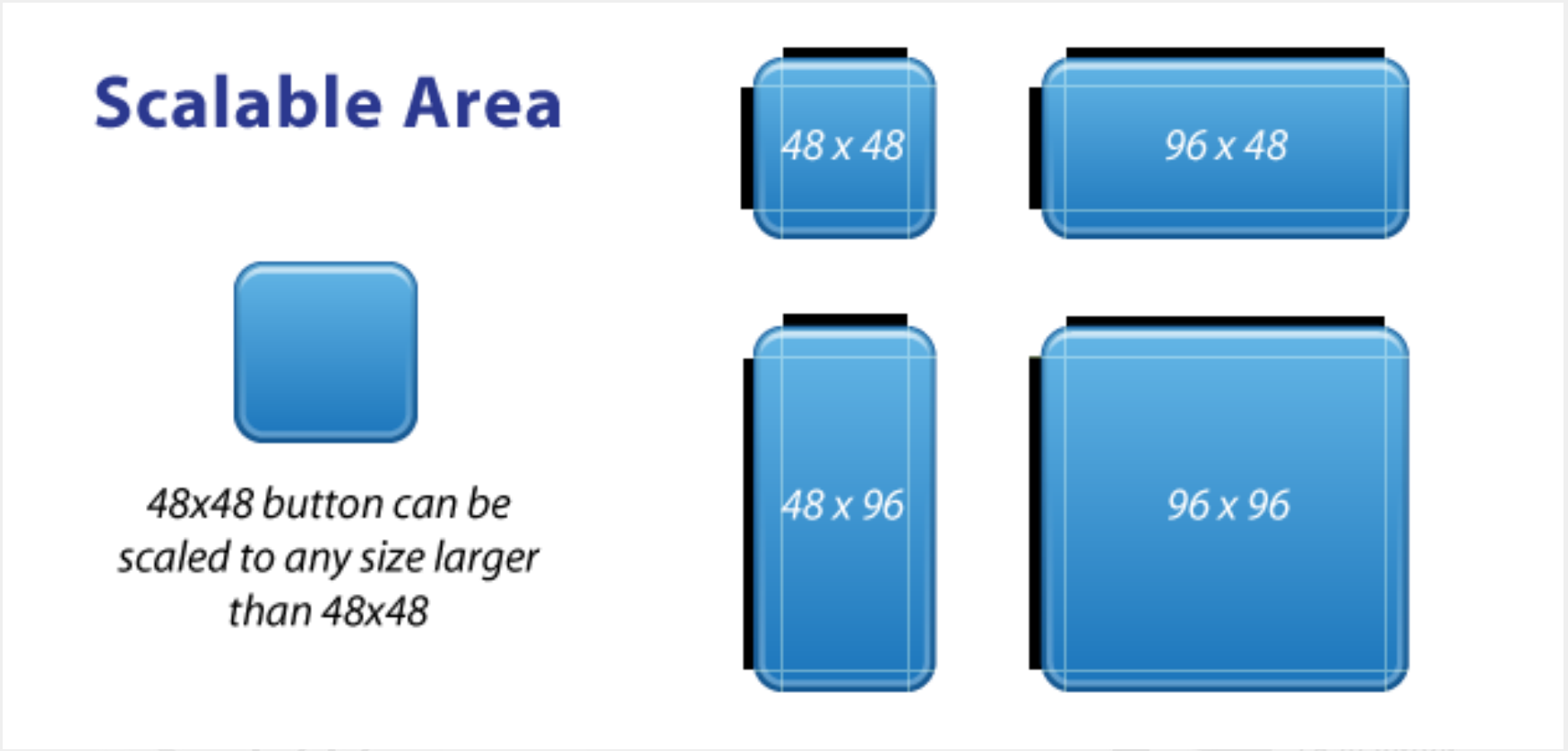
Annexe 2 : Les fichiers nine-patch

Les fichiers nine-patch

- Format d'image particulier, parfois requis pour certaines images par Android
- L'extension de fichier doit être **`.9.png`**
- Le contenu de l'image reste le même, mais une bordure de **1px** est à rajouter **tout autour** de l'image
  - Les bordures du haut et de gauche servent à définir la zone étirable sur l'image
  - Les bordures de droite et du bas servent à définir la zone à remplir (Dans le cas d'un texte par exemple. Cette zone est optionnelle
- Si les pixels de cette bordure sont **noirs**, ils définissent la partie à étendre ou à remplir. Sinon ils doivent être **blancs**.

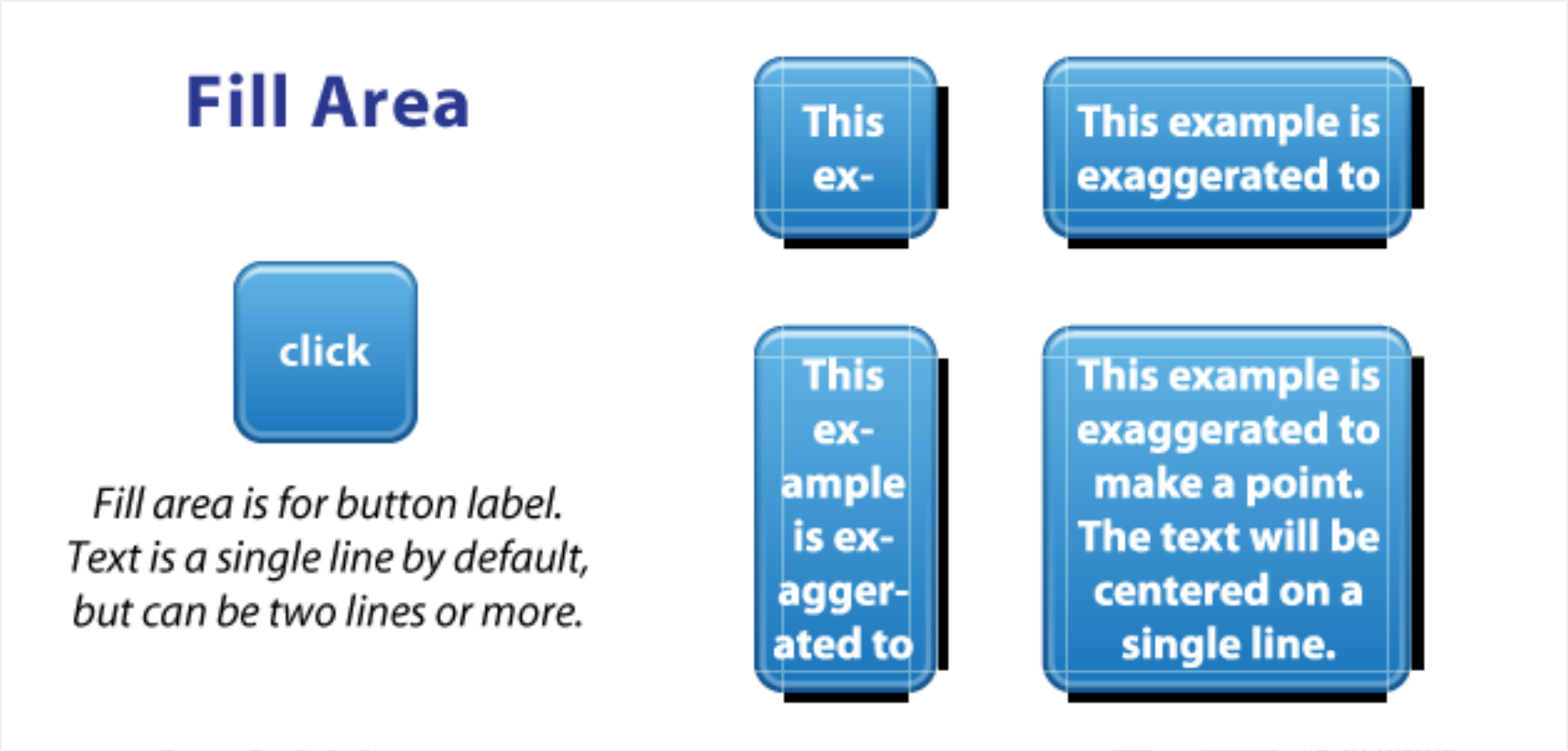
Annexe 2 : Les fichiers nine-patch

Exemples :



Annexe 2 : Les fichiers nine-patch

Exemples :



Annexe 3 : La sécurité des applications Android

Android est un système multi-processus

Cela implique certaines sécurités

- Les applications ne doivent pas pouvoir accéder aux informations d'autres applications
- Cela vaut à la fois pour les données, le code, ou même les fichiers
- Ainsi, même les périphériques amovibles, lorsqu'ils sont utilisés par Android, sont segmentés afin que les applications ne puissent voir que leur propre dossier

Annexe 3 : La sécurité des applications Android

Les applications ne doivent pas non plus pouvoir accéder aux données du système

- Cela est assuré à la fois par la machine virtuelle **Dalvik**, mais aussi et surtout par le système **Linux** présent sur la machine

Toute application **doit être signée**

- Afin d'identifier l'auteur de l'application
- Afin de savoir si elle a été validée ou non par *Google*

Annexe 3 : La sécurité des applications Android

Ainsi, l'application ne peut pas nuir à d'autres éléments qu'elle même, ou tout au plus à d'autres applications, mais possédant le même identifiant

- Cet identifiant étant défini par le certificat qui a signé ces applications
- Ainsi que par l'attribut `SharedUserId`, qui doit être le même

Notez que toutes ces sécurités ne sont plus présentes dés lors qu'un téléphone est *rooté*

Le terme équivalent sur **iOS** est « *Jailbreak* »

Annexe 3 : La sécurité des applications Android

- Les Applications ont parfois besoin de permissions spécifiques pour accéder à des fonctionnalités
- **Exemples :**
  - Accés aux contacts
  - Accés aux paramètres du *wifi*
  - ...
- **Avant *Android* 5.0**, il n'était pas possible d'installer une application si on n'acceptait pas que toutes les permissions qu'elle demandait soient autorisées
- **Avec *Android* 5.0**, il est possible après installation d'autoriser ou non des permissions à la demande pour l'utilisateur

Annexe 3 : La sécurité des applications Android

Il est possible pour une application de définir ses propres permissions

- Cela peut limiter l'exécution de l'application par d'autres applications, par exemples
- La demande d'acceptation peut survenir lors du lancement de l'application même, d'un service, de la réception d'un message *Broadcast*, etc.
- Les permissions se définissent dans le fichier **Manifest**

Annexe 3 : La sécurité des applications Android

Avant d'être ajoutée sur le Google Play Store, chaque application doit :

- Être signée avec un certificat validé par *Google*
- Être analysée par les algorithmes de *Google*

Note : En début d'année 2016 *Google* a supprimé plusieurs application malveillante (émission de spams) sur le *Google Play Store*, qui se faisaient passer pour des tests de QI. Cela montre que malgrés tout le système n'est pas infaillible.

Annexe 4 : Les optimisations

- Plusieurs optimisations sont possibles et parfois requises sur les téléphones *Android*
- La raison est simple : Les ressources sont généralement limitées (mémoire, CPU, réseau peu fiable, peu de batterie), et elles le sont d'autant plus sur les derniers périphériques et objets connectés (montres, lunettes, ...)
- D'autres optimisations sont possibles
  - Utilisation de la classe **`ViewHolder`**
  - Réutiliser les objets *Java* si possible
  - Éviter les allocations dans les *callbacks* pouvant être appelées très souvent
  - Éviter de redimensionner les images *bitmap* trop souvent, mettre en cache ces informations
  - Ne pas faire trop de **`Layouts`** ni trop de **`Views`** sur une même page
  - Créer des composants ou **`Layouts`** réutilisables
  - ...

FIN