Android Performance

Rémi Forax

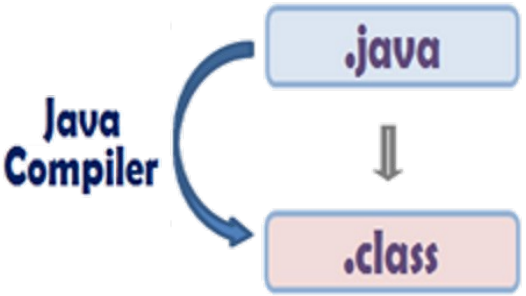
Performance

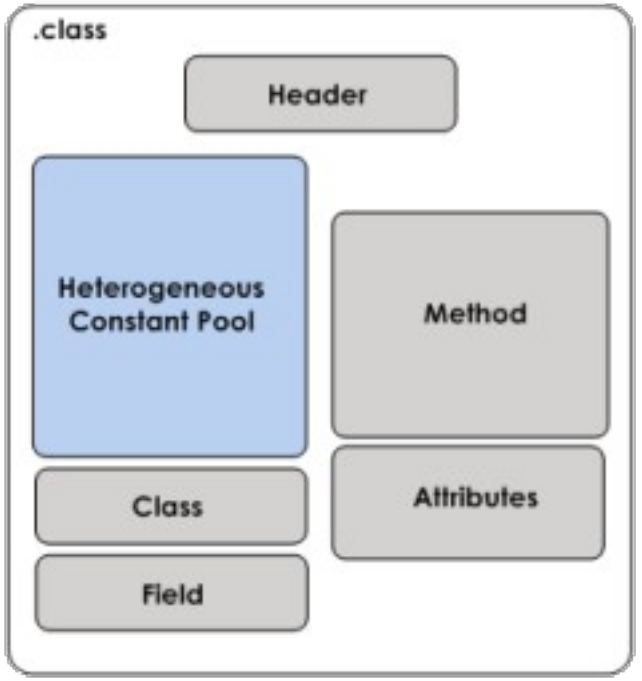
Android n'est pas un environnement d'exécution desktop/server

- Linux + patch wakelock
- Librarie C spécifique (bionic)
- Gestion de process spécifique
  - zigote
- Machine virtuelle spécifique (dalvik)
  - format du bytecode spécifique (DEX)

Rappel: Java VM

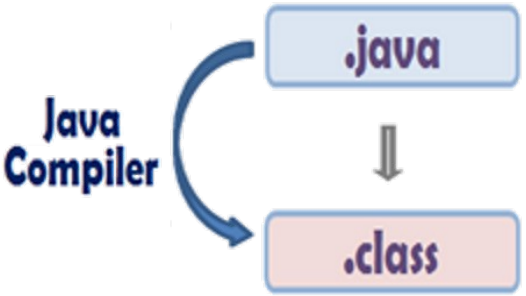
Interpreteur lit un .class et charge dynamiquement les classes dont il a besoin

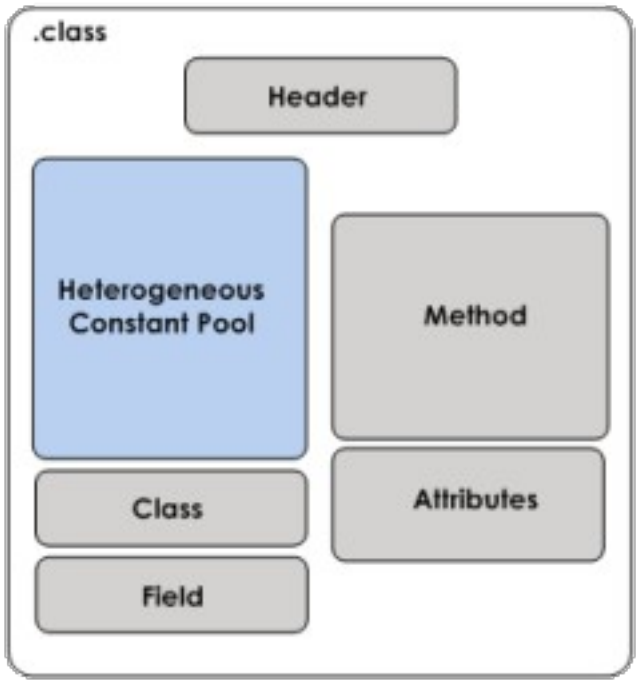


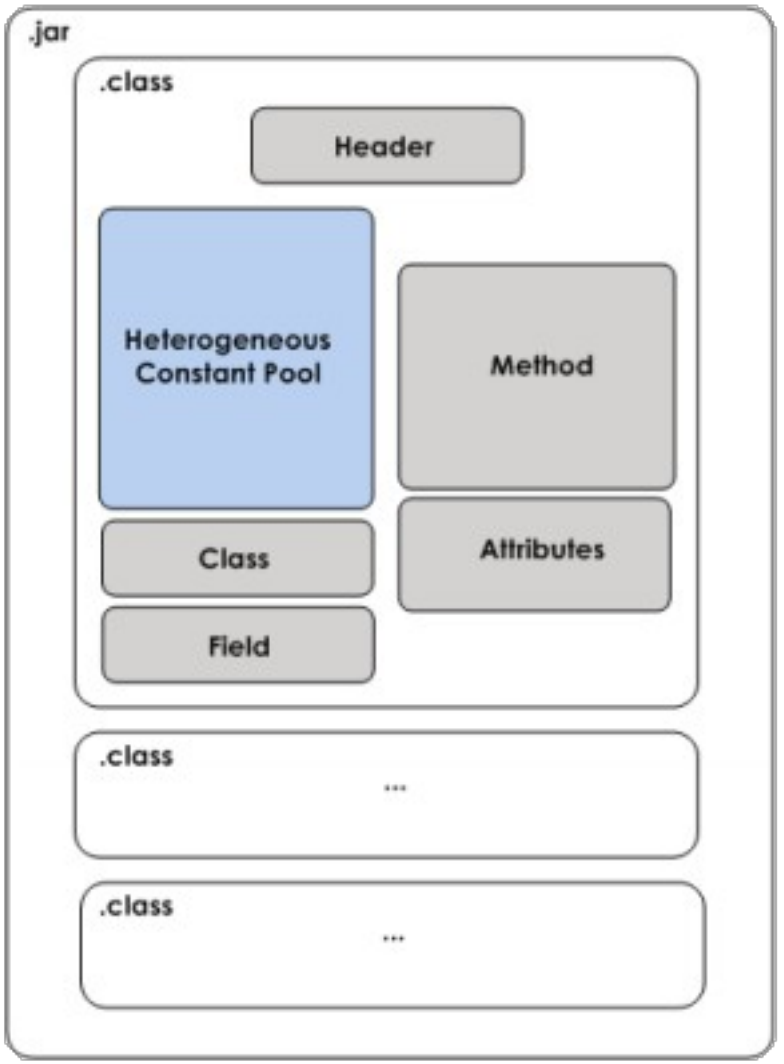


JVM vs DVM

On regroupe les .class dans un jar (compression au format ZIP)



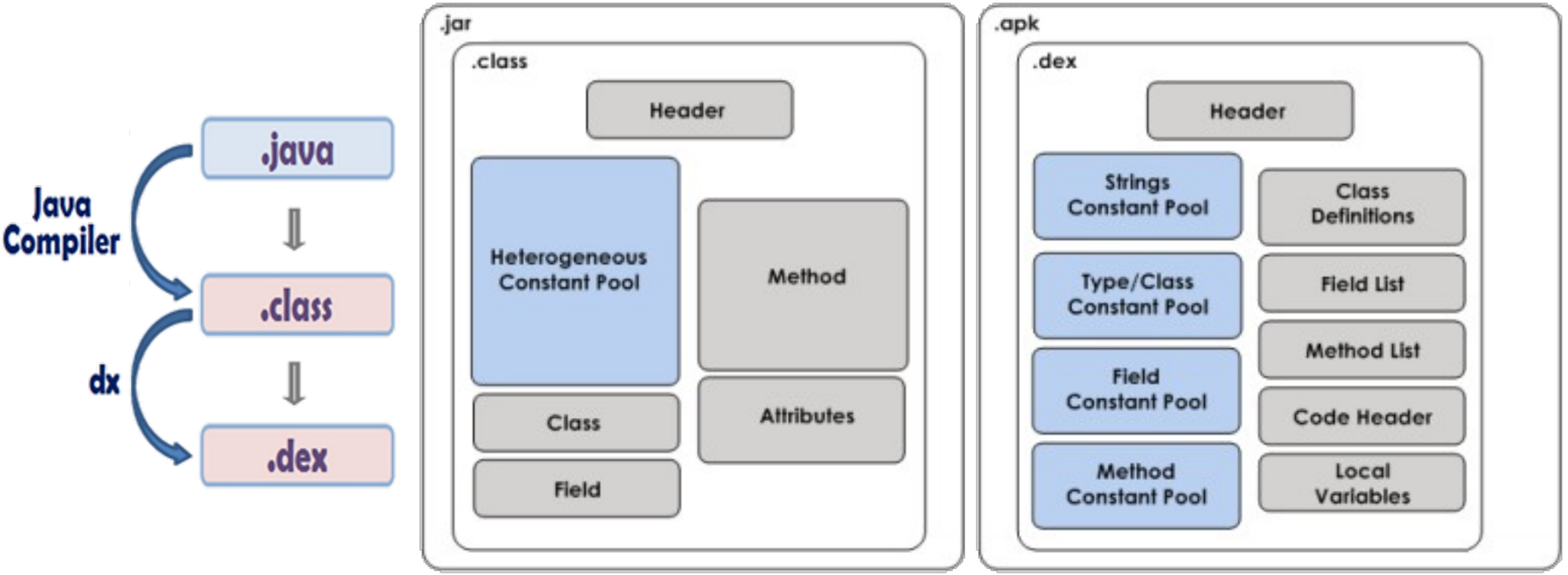




JVM vs DVM

Dalvik lit un fichier DEX

Un DEX contient plusieurs classes compressées



Dalvik / format DEX

Interpreter à registres

Bytecode different des .class habituelles

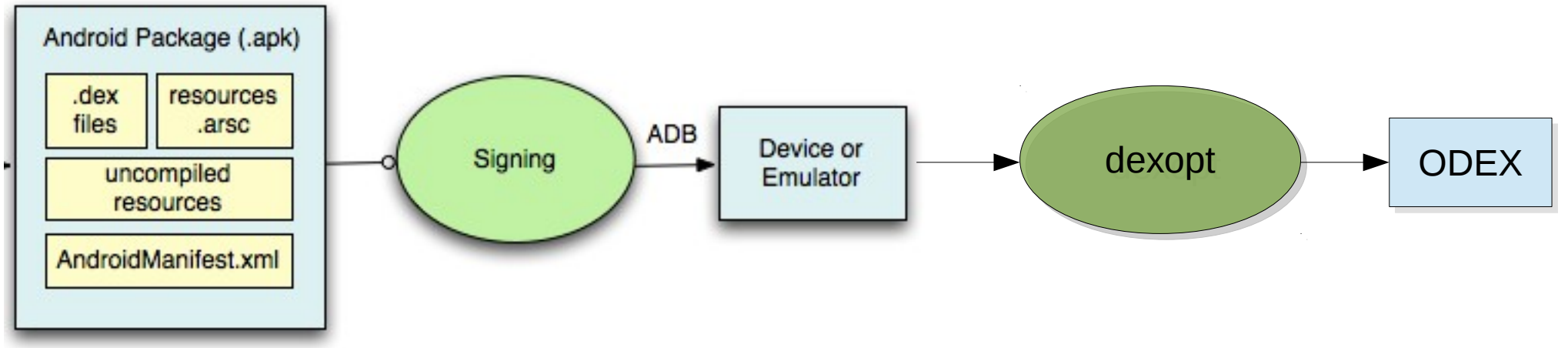
Un .dex stocke plein de .class

- .dex un peu plus gros qu'un .jar
- Les .dex sont read-only
- Les dictionnaires (constant pool) sont globales à un dex

Dexopt

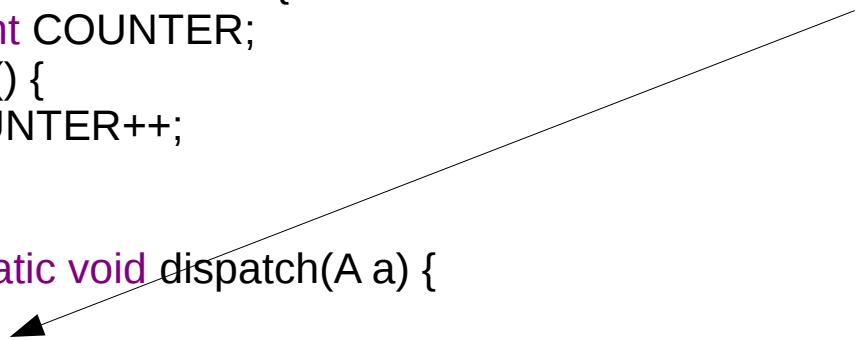
Sur le téléphone Android, lors de l'installation, l'outil dexopt transforme le bytecode

- Vérifie le code
- Optimise le code Ahead Of Time
  (byte order, de-virtualisation, intrinsics/inlining de méthode connue)



Rappel: fonctionnement d'une VM classique

```
static class A {
    static int COUNTER;
    void m() {
        COUNTER++;
    }
}
static class B extends A {
    static int COUNTER;
    void m() {
        COUNTER++;
    }
}
private static void dispatch(A a) {
    a.m();
}
private static void test(A a) {
    for(int i=0; i < 100_000; i++) {
        dispatch(a);
    }
}
public static void main(String[] args) {
    test(new A());
    test(new B());
```



Appel avec 1 classe

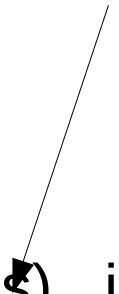
```
public static void main(String[] args) {
    test(new A());
```

```
java -XX:+PrintInlining ...
 58    1      b       Test::dispatch (5 bytes)
                        @ 1   Test$A::m (9 bytes)   inline (hot)

 77    2      b       Test$A::m (9 bytes)
 78    3 %    b       Test::test @ 5 (19 bytes)
                        @ 6   Test::dispatch (5 bytes)   inline (hot)
                          @ 1   Test$A::m (9 bytes)   inline (hot)
```

Appel avec 1 classe + B est chargée

```
public static void main(String[] args) {
    new B();
    test(new A());
```

L'interpreteur profile les appels polymorphe



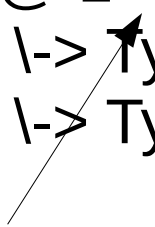
```
java -XX:+PrintInlining ...
55    1      b       Test::dispatch (5 bytes)
                       @ 1   Test$A::m (9 bytes)   inline (hot)
                        \-> TypeProfile (6700/6700 counts) = Test$A

70    2      b       Test$A::m (9 bytes)
71    3 %    b       Test::test @ 5 (19 bytes)
                       @ 6   Test::dispatch (5 bytes)   inline (hot)
                         @ 1   Test$A::m (9 bytes)   inline (hot)
                          \-> TypeProfile (6701/6701 counts) = Test$A
```

Appel avec 2 classes

```
public static void main(String[] args) {
    test(new A());
    test(new B());
```

java -XX:+PrintInlining ...

```
...
74    5 %  b       Test::test @ 5 (19 bytes)
                     @ 6   Test::dispatch (5 bytes)   inline (hot)
                       @ 1   Test$A::m (9 bytes)   inline (hot)
                       @ 1   Test$B::m (9 bytes)   inline (hot)
                        \-> TypeProfile (2047/8748 counts) = Test$B
                        \-> TypeProfile (6701/8748 counts) = Test$A
```



Le JIT inline les 2 appels dans test

donc

L'interpreteur

- Regarde les classes qui sont chargée (CHA)
- Pour un site d'appel donné
  - Profile les classes possible du receveur

Le JIT utilise ces informations

- Pour dé-optimiser le code
- Pour inliner le code

La création du profile et le JIT se font à l'exécution avec le données réels

et Dalvik alors

Dalvik ne profile pas, ne deoptimise pas, n'inline pas

- Dexopt fait cela sur les données statique (AOT), pas à l'exécution

donc il faut écrire le code en conséquence

- Éviter les appels polymorphe si pas nécessaire
- Eviter les getter/setter si pas nécessaire

Les 3 règles de l'optimization

Règle 1: On optimise pas

- On recalcule pas deux fois la même chose
- On choisi le bon algo (la bonne complexité)

Règle 2: On n'optimise pas à l'aveugle

- La majorité du code est pas exécuter assez souvent
- On utilise un profiler (sur le hardware !)

Règle 3: On n'optimise pas pareil chaque langage/plateforme

- On suit les bonnes pratiques

Bonnes pratiques

- Eviter l'allocation d'objets inutiles
- Pas de stockage de reference graphique
- Eviter les appels polymorphes
- Vive la visibilité de paquetage
- Rien dans le bloc statique
- Attention à la boucle for-each
- On utilise pas de double

Eviter l'allocation d'objets inutiles

Le GC de Dalvik est un mark and sweep

- Pas générationnel !, chaque objet compte
- Pas compact, dont les objets ne sont pas bouger en mémoire
- Marche comme un malloc/free avec free automatique
  - Problème de fragmentation (gruyère !!)

donc attention à l'allocation

de plein de petits objets

de gros tableaux/images !

Eviter l'allocation d'objets inutiles

Il y a plusieurs méthodes dans le SDK d'Android qui sont appellées très souvent, il ne faut pas faire d'allocation dans ces méthodes

- Affichage
  - View.measure(), View.layout(), View.onDraw()
- Gestion évènement hardware
  - onTouch(View, MotionEvent)
- Les modèles basés sur ListAdapter
  - getItem(), getView(), etc.

Eviter l'allocation d'objets inutiles

Problème, allocation d'un objet dans onDraw !

```
public class GraphicsView extends View{
    @Override
  protected void onDraw(Canvas canvas) {
    Paint paint = new Paint();
    paint.setARGB(255, 0, 0, 0);
    ...
```

Eviter l'allocation d'objets inutiles

Exemple de recyclage d'objet dans onDraw.

```
public class GraphicsView extends View{
  // faire l'allocation ici
  private final Paint paint = new Paint();

  @Override
  protected void onDraw(Canvas canvas) {
    // ne pas faire l'allocation ici, recycler l'objet
    paint.setARGB(255, 0, 0, 0);
    ...
```

Activité & Etat

Il faut faire **très** attention à

- différentier
  - Les objets Androids (Activity, Context, View)
  - Les objets de l'application (donnée métier, AsyncTask, ...)
- l'endroit où l'on maintient des références sur ces objets
  - On ne stocke pas de référence à un objet Android (pas de memory leak !)
  - Une activité et toutes les vues correspondantes peuvent disparaitres
- respecter le cycle de vie des objets
  - Libérer les ressources visuelles dans onPause
  - Sauvegarder/recharger dans le bundle dans onStop/onCreate

Eviter les appels polymorphes

Manipuler les objets par des références sur leur classe pas leur type

- Si cela ne nuit pas à la ré-utilisation

  ~~List<String>~~ list = new ArrayList<String>()

Mettre les méthode private ou static ou final si possible

- Dexopt évite l'appel polymorphe dans ces cas

Utiliser les bons types !

```
public class BadMovie  {
    private final List<People> cast = new ArrayList<People>();

    public List<People> getFirstFirstCastMember(int limit) {
        List<People> firstList = new ArrayList<People>();

        firstList.addAll(
            cast.subList(0, Math.min(limit, cast.size())));

        return firstList;
    }
}
```

Utiliser les bons types !

```
public class GoodMovie  {
    // ici ArrayList car car elle n'est pas visible de l'extérieur
    private final ArrayList<People> cast = new ArrayList<People>();

    // ici List car c'est un appel publique
    public List<People> getFirstFirstCastMember(int limit) {
        // ici ArrayList car elle n'est pas visible, on donne aussi une taille
        ArrayList<People> firstList = new ArrayList<People>(limit);

        // cool mais pas efficace car crée une liste intermédiaire
        //firstList.addAll(cast.subList(0, Math.min(limit, cast.size())));

        // on sort le calcul de min du test pour l'exécuter qu'une fois
        int min = Math.min(limit, cast.size());
        for(int i=0; i<min i++) {
          // ici on sait que cast est une ArrayList donc get() est en 0(1)
          firstList.add(cast.get(i));
        }
        return firstList;
    }
}
```

Vive la visibilité de paquetage

La visibilité de paquetage est souvent un bon compromis pour les méthodes/champs qui doivent être visible pour une/d'autres classes mais pas forcément à l'extérieur

- On utilise souvent des classes internes statique
  - Attention le constructeur par défaut à la même visibilité que la classe
- Donc **pas de getter**, on accède directement au champ
- On met les méthodes final pour indiquer à dexopt que l'appel n'est pas polymorphe

Rien dans le bloc statique

Même si les classes sont chargées dynamiquement, faire des initialisations dans le bloc statique n'est pas une bonne idée car celles-ci entraine souvent le chargement d'autres classes

C'est souvent la cause #1 de “Java est lent” au démarrage.

Rien dans le bloc statique

Attention !

- Les variables statiques non final
- Les variables statique final dont le type n'est pas un primitif ou String

  sont initialisées dans le bloc statique

se conformer au cycle de vie d'une activité évite les mauvaises surprises

Attention à la boucle foreach !

La boucle for(:)

- Parcours avec un index sur un tableau
  - OK !
- Parcours en utilisant un Iterateur sur une collection
  - Donc création d'un Itérateur à chaque parcours
  - Problème si la boucle est dans une méthode appelée fréquemment
    - En Java classique pas de problème: Escape Analysis
    - Avec Andrdoid, il vaut mieux
      - Ne faire des boucles que sur ArrayList
      - et de façon indexé (arrayList.get(index))

On utilise pas de double !

- Les doubles de Java
  - sont codés sur 64 bits
  - ont leur opération primaire sur 80 bits
- Les téléphones/tablettes
  - ont un CPU et un bus sur 32 bits (par le dernier iphone !)
  - des puces graphiques 32 bits

donc toutes les opérations sur les doubles sont lentes (voir très lente)

android.os.StrictMode

- Permet de détecter des erreurs d'utilisation d'API non voulu (best effort)
  - Permet de spécifier une politique de détection pour les threads et pour la VM

```
public void onCreate() {
  super.onCreate();
  StrictMode.setThreadPolicy(...);
  StrictMode.setVMPolicy(...);
}
```

- Ne doit être activer que lors du développement

android.os.StrictMode

On choisit ce que l'on veut détecter
avec detect*()

- detect[DiskWrites|Network|LeakedClosableObjects|...]

puis de l'action a effectuer avec penalty*()

- penalty[Death|Dialog|Log|...]

```
setThreadPolicy|setVMPolicy(
   new StrictMode.*Policy.Builder()
    .detectAll()
    .penaltyLog()
    .build());
```

Debugger

Android possède un debugger, DDMS, intégré à l'environement Eclipse

permet de

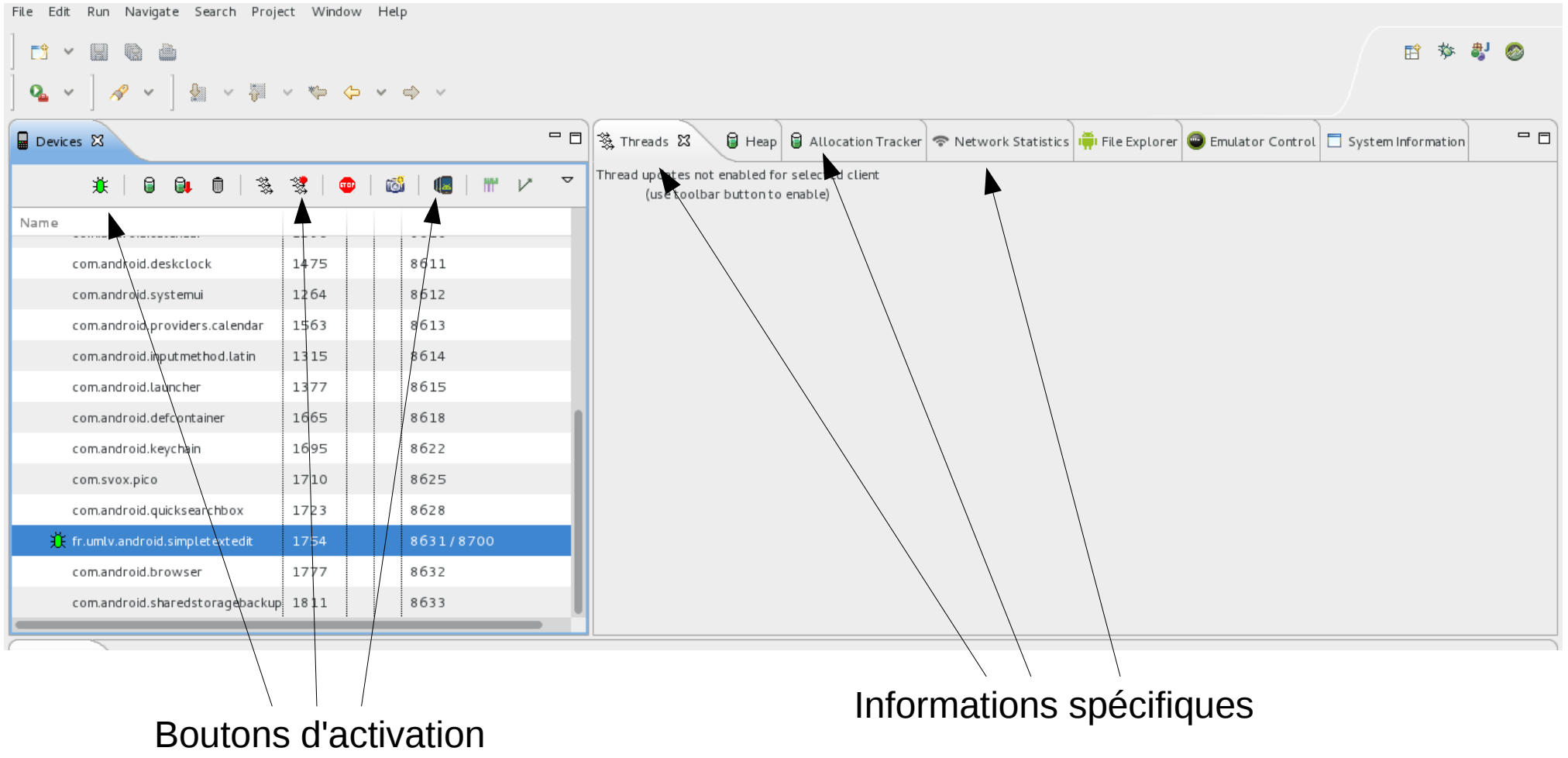
Changer le code de l'application à chaud

Profiler l'application

- Voir les allocations
- Voir le temps passé dans chaque méthode
- Voir l'utilisation du réseau

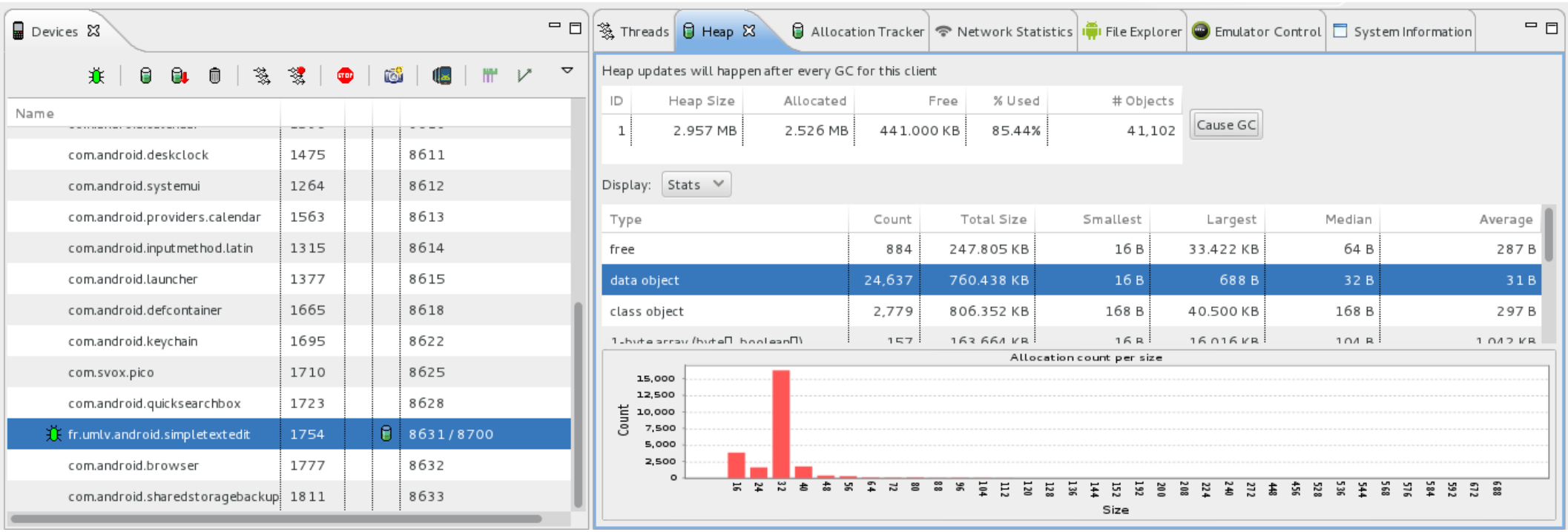
DDMS

Perspective DDMS sous eclipse



DDMS - heap

Allocation par taille d'objet



DDMS - trace

Affiche le stacktrace avec les temps d'exécutions de chaque méthodes