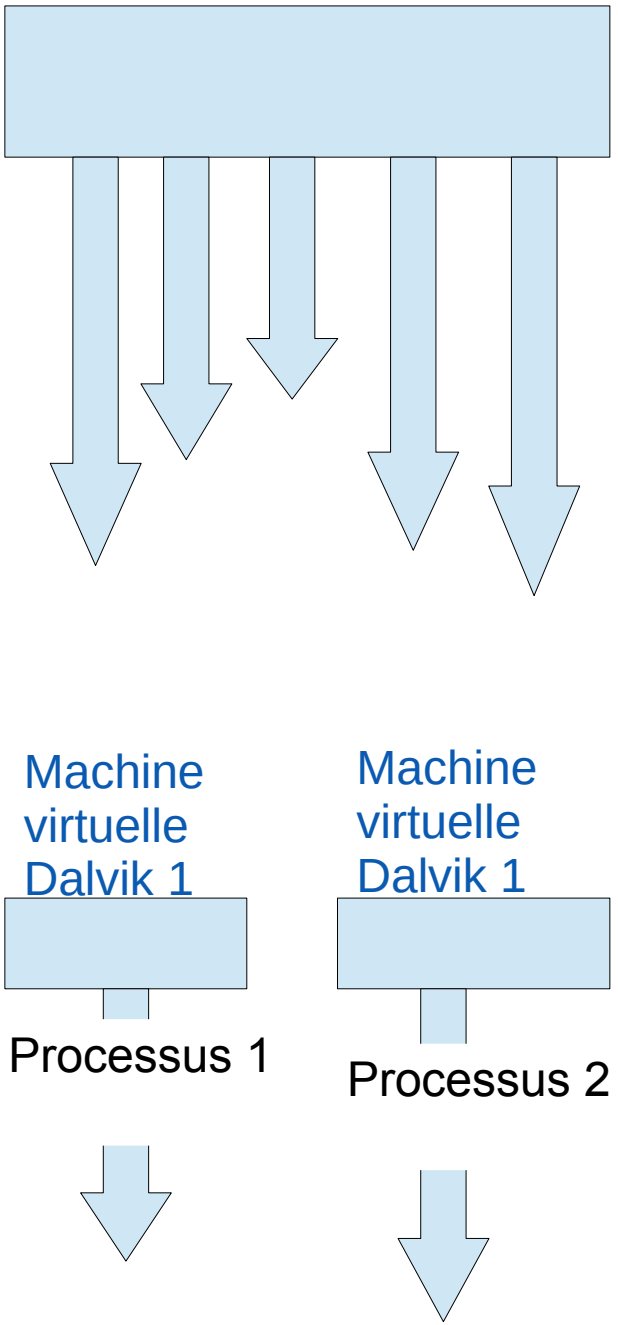
Processus et Threads

Thread versus Processus

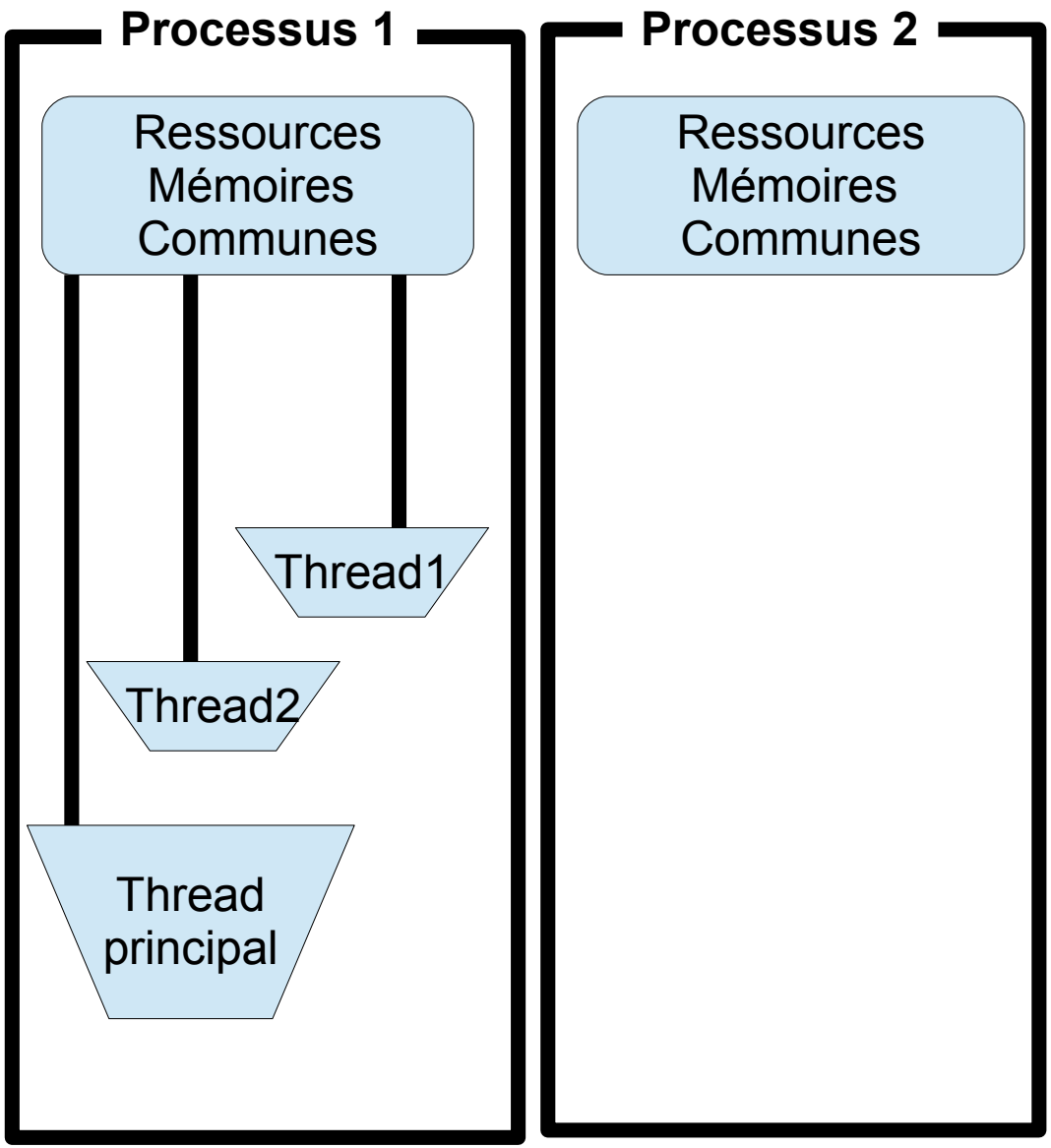
- Les threads et les processus sont deux méthodes pour l'exécution parallèle des applications

- Les processus sont des unités d'exécution indépendantes qui :
  - Contiennent leurs propres états
  - Utilisent leurs propres espaces de nommage
  - Communiquent via le mécanisme de communication inter-processus
  - Apparaissent au niveau architectural



Thread versus Processus

- Les threads sont des unités de construction au niveau du code qui n'affectent pas l'architecture des applications
  - Un processus peut contenir plusieurs threads
    - Tous les threads dans un processus partagent le même état et le même espace de nommage
    - Peuvent communiquer ensemble directement parce que ils partagent les mêmes variables



Thread et processus en Android

- Une application → un processus
  - Au lancement du premier composant d'une application (activité, service, etc.), Android lance un nouveau processus Linux pour l'application avec un seul thread d'exécution
- Un processus pour tous les composants d'une application
  - Par défaut, tous les composants d'une même application sont exécutés dans le même processus et thread (appelé "main" thread)
  - Si un composant d'une application est lancé et il existe déjà un processus créé pour cette application (parce que un autre composant de cette application existe déjà), ce composant est lancé dans ce même processus et utilise le même thread d'exécution
- Multi processus et multi threads
  - Il est possible de créer des processus différents pour différents composants d'une même application
  - Il est possible de créer différents threads dans un même processus

Processus

- Par défaut, tous les composants d'une même application s'exécutent dans le même processus : Situation satisfaisante (adaptée) à la plupart des applications
- Il est possible de définir quel processus doit contrôler quel composant
  - Déclaration à faire dans le fichier manifest
    - Utilisation de l'attribut android:process associé à un composant (activité, service, receiver, provider)
      - Doit mentionner le processus où ce composant doit s'exécuter
    - Utilisation de l'attribut android:process associé à une application
      - Peut spécifier le processus par défaut de tous les composants de cette application

Cycle de vie de processus

- Android peut arrêter un processus à un point donné, quand la mémoire disponible, nécessaires pour d'autres processus qui sont en interaction avec l'utilisateur, est insuffisante
  - Tous les composants de ce processus sont ainsi détruits
  - Un nouveau processus est relancée pour ces composants quand il y aura à nouveau des tâches à exécuter par ces composants
- Pour décider quels processus tuer, le système Android mesure leur importance relative pour l'utilisateur.
  - Par exemple, il arrête en priorité un processus d'hébergement des activités qui ne sont plus visibles à l'écran, par rapport à un processus d'hébergement des activités visibles

- Il y a cinq niveaux dans la hiérarchie de priorité

Règles de gestion du cycle de vie des processus

- Processus au premier plan (Foreground process)
  - Un processus nécessaire pour les tâches que l'utilisateur est entrain de faire
  - Un processus est considéré de cet catégorie si une des conditions suivantes est vérifiée :
    - Le processus héberge une activité d'interaction avec l'utilisateur (la méthode onResume() a été appelée)
    - Le processus héberge un service relié (bound) à une activité d'interaction avec l'utilisateur
    - Le processus héberge un service exécuté au premier plan (foreground) – le service a été appelé avec startForeground()
    - Le processus héberge un service qui est en train d'exécuter une de ses méthodes de rappels/callbacks - onCreate(), onStart(), onDestroy())
    - ...

Règles de gestion du cycle de vie des processus

- Processus visibles
  - Un processus est considéré comme visible s'il vérifie une des conditions suivantes :
    - Le processus héberge une activité qui n'est pas au premier plan mais qui est encore visible à l'utilisateur (sa méthode onPause() a été appelée).
    - Le processus héberge un service relié à une activité visible ou du premier plan
    - Un processus visible est considéré comme extrêmement important et ne peut être tué que si ceci permet de préserver les processus au premier plan

Règles de gestion du cycle de vie des processus

- Processus de gestion de services
  - Un processus qui héberge un service qui a été lancé avec la méthode startService() et qui ne fait pas partie des deux catégories (premier plan et visible)
  - Les processus de gestion de service ne sont pas directement liés à l'interface utilisateur.
    - Il s’exécutent tant qu'il y assez de mémoire pour exécuter les processus « premier plan » et « visible »

Règles de gestion du cycle de vie des processus

- Processus arrière plan (Background process)
  - Un processus hébergeant une activité qui n'est pas (actuellement) visible à l'utilisateur (la méthode onStop() a été appelée)
  - Ce processus n'a pas d'impact direct sur l'utilisation encours de l'utilisateur
  - Le système peut les tuer à tout moment pour récupérer de la mémoire pour des processus au premier plan, visibles ou de gestion de service
  - Sont enregistrés dans une liste LRU (least recently used) qui indexe les processus dont l'une de leur activité a été récemment visualisée.
    - Ces processus sont tués selon l'ordre chronologique de visualisation de leurs activités

Règles de gestion du cycle de vie des processus

- Les processus vides (**Empty process)**
  - Un processus qui n'héberge aucun composant active de l'application
  - La seule raison de laisser ses activités en vie et pour ds raison de « cache » : améliorer le temps de re-lancement d'un composant
  - Le système tue ces processus en priorité

Règles de gestion du cycle de vie des processus

- Android classe un processus dans l'ordre de priorité la plus élevé possible
  - Se basant sur les composants qui sont actifs dans le processus
  - Exemple :
    - Si un processus héberge en même temps un service et une activité visible, ce processus sera classé comme un processus visible et non pas comme un processus de gestion de service

- Le rang d'un processus pourrait être augmenté en fonction des processus qui dépendent du processus en question

- Un processus servant un autre processus ne peut être jamais classé moins prioritaire que le processus dont il serve

Thread

Les threads

- Un thread est une unité d'exécution parallèle
  - Chaque thread a son propre pile de tâches (pile d'appels)
    - Cette pile d'appel est utilisée pour la gestion des appels de méthodes, le passage de paramètres et sauvegarde des variables locales de la méthode appelante

- Chaque machine virtuelle instancie au moins un thread principal
  - Appelé plus communément User Interface thread (UI thread)
  - Exécute les méthodes onCreate(), onStart(), OnPause(), onResume(), onStop(), onDestroy() de l'Activité/service
  - Gère les interactions de l'utilisateur

- Les threads dans la même VM interagissent et se synchronisent par l'utilisation d'objets communs

Avantages/inconvénients du multi-threading

- Avantages
  - Les threads d'un même processus partagent les mêmes ressources mémoires mais ont des états d'exécutions indépendantes
  - Permettent de séparer différentes tâches d'une application
    - Le thread principal est responsable de l'interface utilisateur
    - Les tâches lentes sont exécutées par des threads de fond (arrière plan)
  - Le mécanisme du « Threading » fournit une abstraction utile pour l'exécution concurrente

- Inconvénients
  - Le code de l'application est plus complexe
  - Besoin d'éviter, déterminer et résoudre les interblocages entre threads

Création de threads

- Dans une grande majorité des cas, l'UI thread permet de réaliser toutes les opérations nécessaires au fonctionnement d'une application

- Cependant, certaines actions sont consommatrices en temps
  - Il est intéressant d'envisager la création de threads
    - Exemple
      - Création d'un thread pour récupérer des informations d'Internet, consulter la base de données des contacts, etc.

Création de threads

- Deux façons pour créer un thread

  1) Création d'une nouvelle classe qui étend la classe Thread et « override » la méthode run()

```
MyThread t = new MyThread();
t.start();
```

  2) Création d'une nouvelle instance de Thread en lui passant un objet Runnable

```
Runnable myRunnable1 = new MyRunnableClass();
Thread t1 = new Thread(myRunnable1);
t1.start();
```

- Dans les deux cas, la méthode Start() doit être appelée pour lancer le thread

Création de threads

- Exemple : création de threads suivant deux styles différents

```
public class MainActivity extends Activity {
@Override
public void onCreate(Bundle savedInstanceState)
        {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
            Runnable myRunnable1 = new MyRunnableClass();
            Thread t1 = new Thread(myRunnable1);
            t1.start();

            MyThread t2 = new MyThread();
            t2.start();

        }//onCreate
```

Création de threads

- Exemple : création de threads suivant deux styles différents
  - La classe MyRunnableClass

```
public class MyRunnableClass implements Runnable {
    @Override
    public void run() {
        try {
            for (int i = 100; i < 105; i++)
                {
                    Thread.sleep(1000);
                    Log.e("<<runnable>>", “runnable talking: " + i);
                }

            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
    }//run

}//class
```

Création de threads

- Exemple : création de threads suivant deux styles différents
  - La classe MyThread

```
public class MyThread extends Thread{
    @Override
    public void run() {
        super.run();
        try {
                for(int i=0; i<5; i++){
                    Thread.sleep(1000);
                    Log.e("[[thread]]", "Thread talking: " + i);
                }
            }
        catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
         }
  }//run

}//class
```

Communication entre threads

- Les threads de fond ne sont pas autorisés à interagir avec l'interface utilisateur
  - Le thread principal est le seul autorisé à accéder à la vue de l'activité principale
  - Les variables globales peuvent être accédées (consultées et mises à jour) dans les threads

Communication entre threads

- Une application peut inclure des opérations lentes (qui consomment du temps)
  - Le thread principal doit rester en charge uniquement des interactions avec l'utilisateur

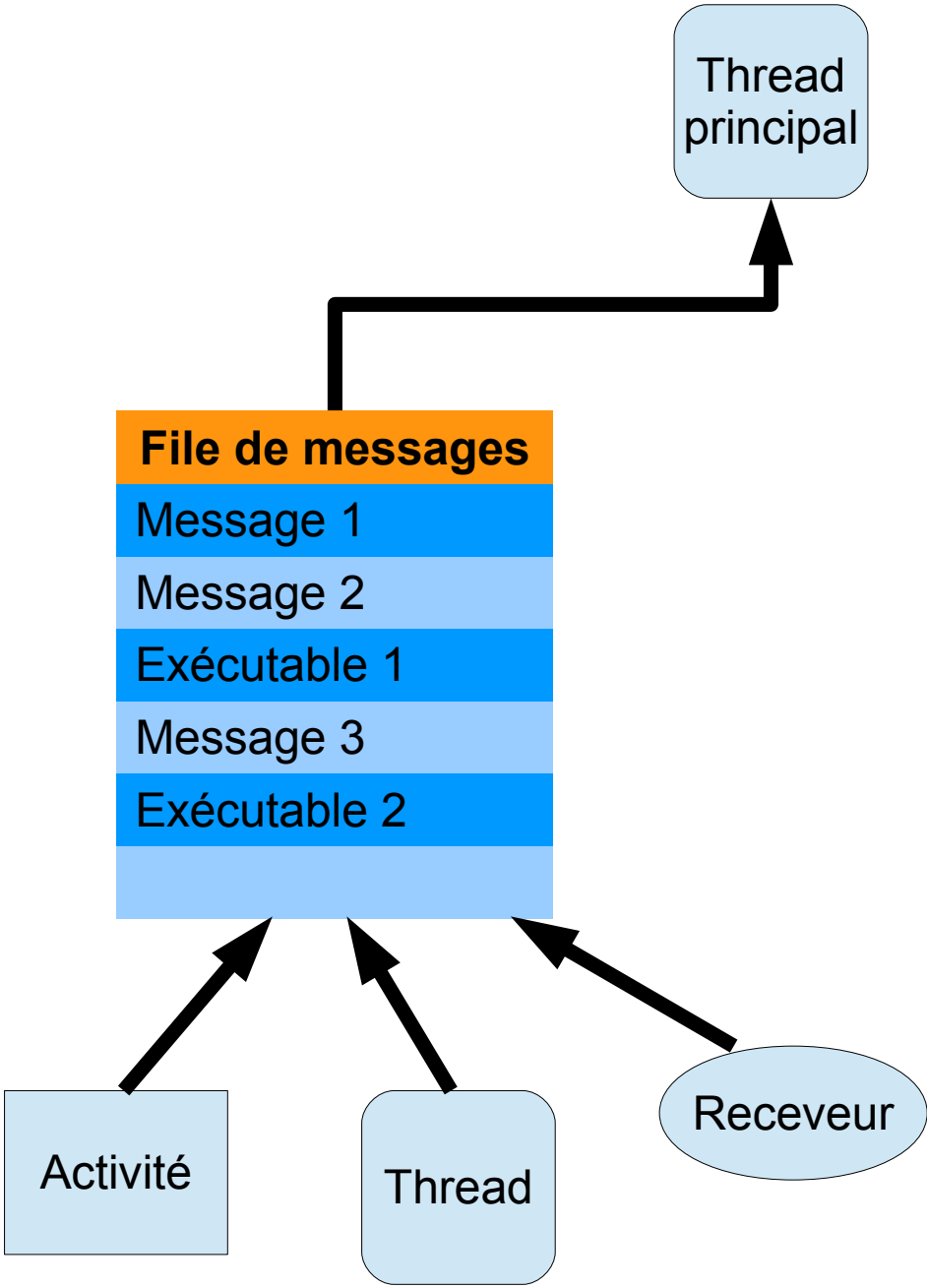
    - Deux solutions

      - Exécuter les opérations lentes comme un service de fond et utilisation des notifications pour informer l'utilisateur de l'évolution de ces opérations

      - Exécuter les opérations lentes dans un thread de fond
        - Les interactions entre les threads Android sont réalisées par l'utilisation de :
          - Un objet Handler du thread principal
          - Envoi (poster) des objets Runnable à la vue principale

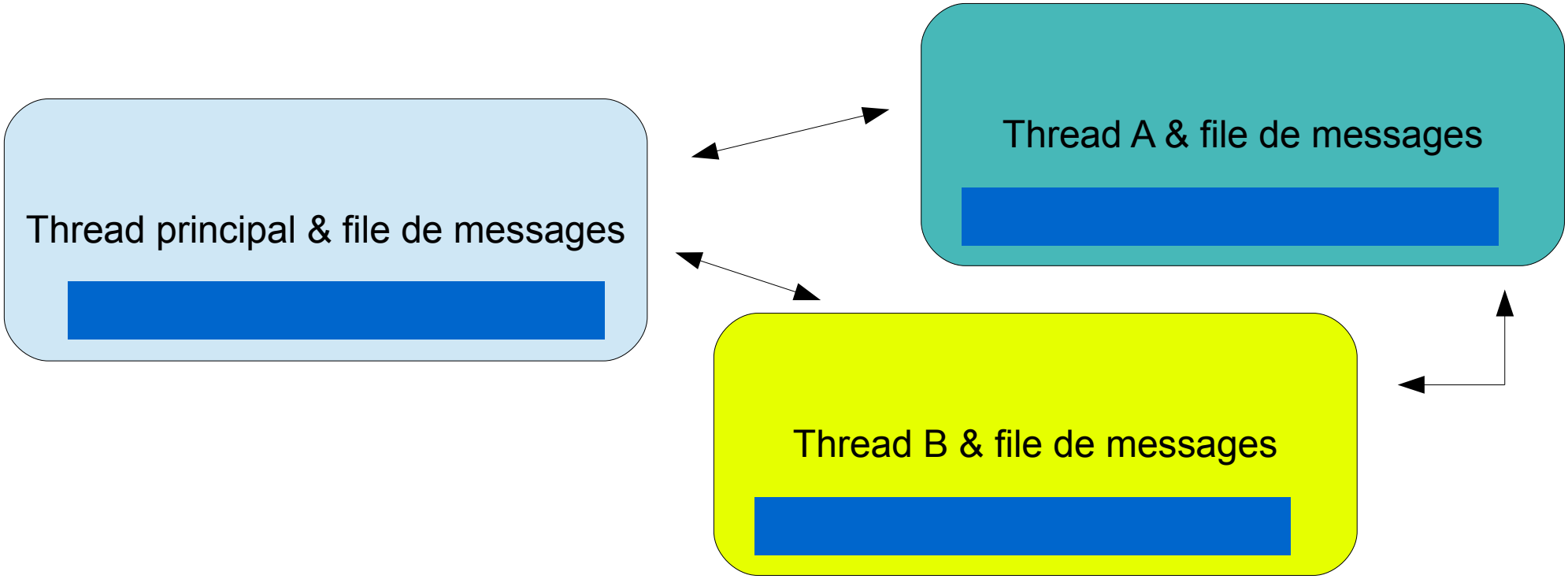
Communication entre threads

- La classe Handler
  - Deux utilisations principales d'un objet Handler
    - De programmer des messages et des exécutables (Runnables) pour être traités
    - Ajouter des actions qui doivent être exécutées par un autre thread



Communication entre threads

- Le thread principal est associé à un Handler pour recevoir les messages envoyés par les autres threads
- Chaque thread peut définir son propre Handler
  - Un Handler dans un thread secondaire crée une file locale de massage qui est utilisée pour recevoir les messages envoyés par d'autres threads (incluant le thread principal)



Communication entre threads

- La file de messages/Handler
  - Pour communiquer avec le thread principal, un thread secondaire doit demander un jeton. Il utilise la méthode obtainMessage()
  - Une fois le jeton obtenu, le thread secondaire attache les données à transmettre au jeton et ajoute le jeton à la file des messages associée au Handler. Utilisation de la méthode sendMessage()
  - Le Handler attends la réception de nouveaux messages au thread principal. Utilisation de la méthode handleMessage()
  - Un message extrait de la file peut soit retourner certaines données au thread principal ou demander l'exécution d'objets exécutables (runnable). Utilisation de la méthode post()

Communication entre threads

- Utilisation de messages

Thread principal	Thread secondaire

...
Handler myHandler = new Handler() {
 	
@Override
public void handleMessage(Message msg) {
// do something with the message...
// update GUI if needed!
...
}//handleMessage
 	
};//myHandler
...
	...
	Thread backgJob = new Thread (new Runnable (){
	
	@Override
	public void run() {
	//...do some busy work here ...
	//get a token to be added to the main's message
	//queue
	Message msg = myHandler.obtainMessage();
	...
	//deliver message to the //main's message-queue
	myHandler.sendMessage(msg);
	}//run
	
	});//Thread
	
	//this call executes the parallel thread
	backgroundJob.start();
	...

Communication entre threads

- Utilisation de Post

Thread principal	Thread secondaire

```
…

Handler myHandler = new Handler();
@Override
  public void onCreate(
                    Bundle savedInstanceState) {
  ...
 Thread myThread1 =
                new Thread(backgroundTask,
                                "backAlias1");
 myThread1.start();
}//onCreate
…
// this is the foreground runnable
Private Runnable foregroundTask
        = new Runnable() {
@Override
Public void run () {
// work on the UI if needed
}
...
```

```
// this is the "Runnable" object
// that executes the background thread


 private Runnable backgroundTask
                        = new Runnable () {

  @Override
  public void run() {
  … Do some background work here

   myHandler.post(foregroundTask);

  }//run

 }; //backgroundTask
```

Communication entre threads

- Messages
  - Pour envoyer un message au Handler, le thread doit invoquer obtainMessage() pour obtenir un objet Message
  - Exemple

```
// thread 1 produces some local data
String localData = “Greetings from thread 1”;

// thread 1 requests a message & adds localData to it
Message mgs = myHandler.obtainMessage (1, localData);
```

Communication entre threads

- Les méthodes sendMessage()
  - sendMessage() puts the message at the end of the queue immediately
  - sendMessageAtFrontOfQueue() puts the message at the front of the queue immediately (versus the back, as is the default), so your message takes priority over all others
  - sendMessageAtTime() puts the message on the queue at the stated time, expressed in the form of milliseconds based on system uptime (SystemClock.uptimeMillis())
  - sendMessageDelayed() puts the message on the queue after a delay, expressed in milliseconds

Communication entre threads

- Traitement des Messages
  - Pour traiter les messages envoyés par les threads secondaires l'objet Handler doit implémenter un listener
    - handleMessage(Message msg)
  - Appelé par chaque message (msg) qui apparaît dans la file des messages
  - Le Handler met à jour la vue principale si nécessaire.

Communication entre threads

- Example 2 : Progress Bar
  - Le thread principal affiche un widget représentant une barre de progression horizontale et circulaire montrant la progression d'une opération exécuté comme une tâche de fond. Certaines données aléatoires sont périodiquement envoyées par le thread de fond et des messages sont affichés dans la vue principale.

Communication entre threads

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#44ffff00"
    android:orientation="vertical"
    android:padding="4dp" >
<TextView android:id="@+id/txtWorkProgress"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:padding="10dp"
            android:text="Working ...."
            android:textSize="18sp"
            android:textStyle="bold" />
<ProgressBar
            android:id="@+id/progress1"
            style="?android:attr/progressBarStyleHorizontal"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
<ProgressBar
            android:id="@+id/progress2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
```

Communication entre threads

```
<ScrollView
    android:layout_width="match_parent"
    android:layout_height="wrap_content" >

<TextView
    android:id="@+id/txtReturnedValues"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="7dp"
    android:background="#ff0000ff"
    android:padding="4dp "
    android:text="returned from thread..."
    android:textColor="@android:color/white"
    android:textSize="14sp" />

</ScrollView>

</LinearLayout>
```

Communication entre threads

```
Public class ThreadsMessages extends Activity {
ProgressBar bar1;
ProgressBar bar2;

TextView msgWorking;
TextView msgReturned;

// this is a control var used by backg. threads
boolean isRunning = false;

// lifetime (in seconds) for background thread
final int MAX_SEC = 30;

//String globalStrTest = "global value seen by all threads ";
int globalIntTest = 0;
```

Communication entre threads

```
Handler handler = new Handler() {
@Override
public void handleMessage(Message msg) {

String returnedValue = (String) msg.obj;
//do something with the value sent by the background thread here
msgReturned.append("\n returned value: " + returnedValue );
bar1.incrementProgressBy(2);
//testing early termination
if (bar1.getProgress() == MAX_SEC){
msgReturned.append(" \nDone \n back thread has been stopped");
isRunning = false;
}
if (bar1.getProgress() == bar1.getMax()){
msgWorking.setText("Done");
bar1.setVisibility(View.INVISIBLE);
bar2.setVisibility(View.INVISIBLE);
}
else {
msgWorking.setText("Working..." + bar1.getProgress() );
} }
}; //handler
```

Communication entre threads

```
@Override
public void onCreate(Bundle icicle) {
      super.onCreate(icicle);
      setContentView(R.layout.main);
      bar1 = (ProgressBar) findViewById(R.id.progress1);
      bar1.setProgress(0);
      bar1.setMax(MAX_SEC);
      bar2 = (ProgressBar) findViewById(R.id.progress2);
      msgWorking = (TextView)findViewById(R.id.txtWorkProgress);
      msgReturned = (TextView)findViewById(R.id.txtReturnedValues);

      //globalStrTest += "XXX"; // slightly change the global string
      globalIntTest = 1;

}//onCreate
```

Communication entre threads

```
public void onStart() {
    super.onStart();

    // this code creates the background activity where busy work is done
    Thread background = new Thread(new Runnable() {
        public void run() {
            try {
                for (int i = 0; i < MAX_SEC && isRunning; i++) {
                    //try a Toast method here (it will not work!)
                    //fake busy work here
                Thread.sleep(1000); //one second at a time
                // this is a locally generated value between 0-100
                Random rnd = new Random();
                int localData = (int) rnd.nextInt(101);
                //we can see and change (global) class variables
                String data = "Data-" + globalIntTest + "-" + localData; globalIntTest++;
                //request a message token and put some data in it
                Message msg = handler.obtainMessage(1, (String) data);
                // if thread is still alive send the message
                if (isRunning) { handler.sendMessage(msg);
                }
            }
    }
```

Communication entre threads

```
catch (Throwable t) {
// just end the background thread
isRunning = false;
} }
}); // Thread


isRunning = true;
background.start(); } //onStart


public void onStop() {
super.onStop();

isRunning = false;

}//onStop

}//class
```