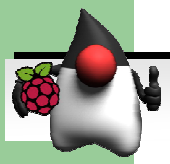
Java pour l'embarqué. Application pour l'Internet des objets

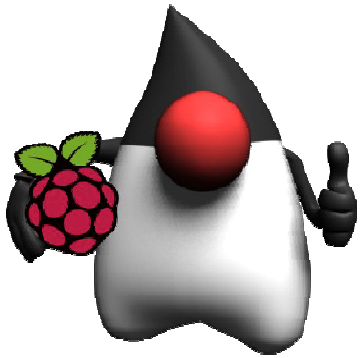
Java ME sur objet connecté RPi

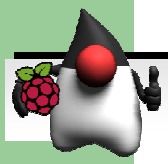
email : kadionik@enseirb-matmeca.fr
web : kadionik.enseirb-matmeca.fr

Patrice KADIONIK
ENSEIRB-MATMECA



PARTIE 1
INTRODUCTION





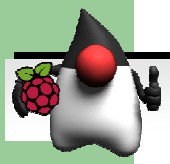
INTRODUCTION

- Ce cours présente Java ME ou J2ME, (*Java Micro Edition*), une version de Java pour les petits systèmes embarqués et l'Internet des objets *Internet of Things* (IoT).

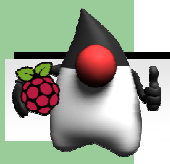
- Nous étudierons ainsi le développement d'applications Java ME ou *MIDlets*.

- La mise en œuvre se fera sur une carte cible Raspberry Pi (RPi) avec l'IDE *Netbeans*. La carte RPi sera présentée ainsi que l'environnement de développement.

- Comme toujours savoir et savoir faire : des travaux pratiques seront proposés à l'issue du cours…

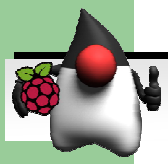


PARTIE 2
JAVA ET JAVA ME



JAVA EVERYWHERE ?

- Java apporte le « *write once, run everywhere!* ».
- Les processeurs d'aujourd'hui même dans l'embarqué supportent un langage interprété. Un bon exemple : Android sur *smartphone*.
- La tendance est donc « *Java everywhere!* ».
- Il faut trouver néanmoins un compromis entre portabilité et performances.



JAVA EE, JAVA SE, JAVA ME

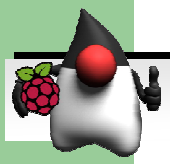
- Java inclut 3 différentes éditions :
  - J2EE (*Java 2 Enterprise Edition*). Appelé maintenant Java EE.
  - J2SE (*Java 2 Standard Edition*). Appelé maintenant Java SE.
  - J2ME (*Java 2 Micro Edition*). Appelé maintenant Java ME.
- Ces différentes éditions visent différents types d'appareils (*devices)* ou de systèmes.

JAVA EE, JAVA SE, JAVA ME

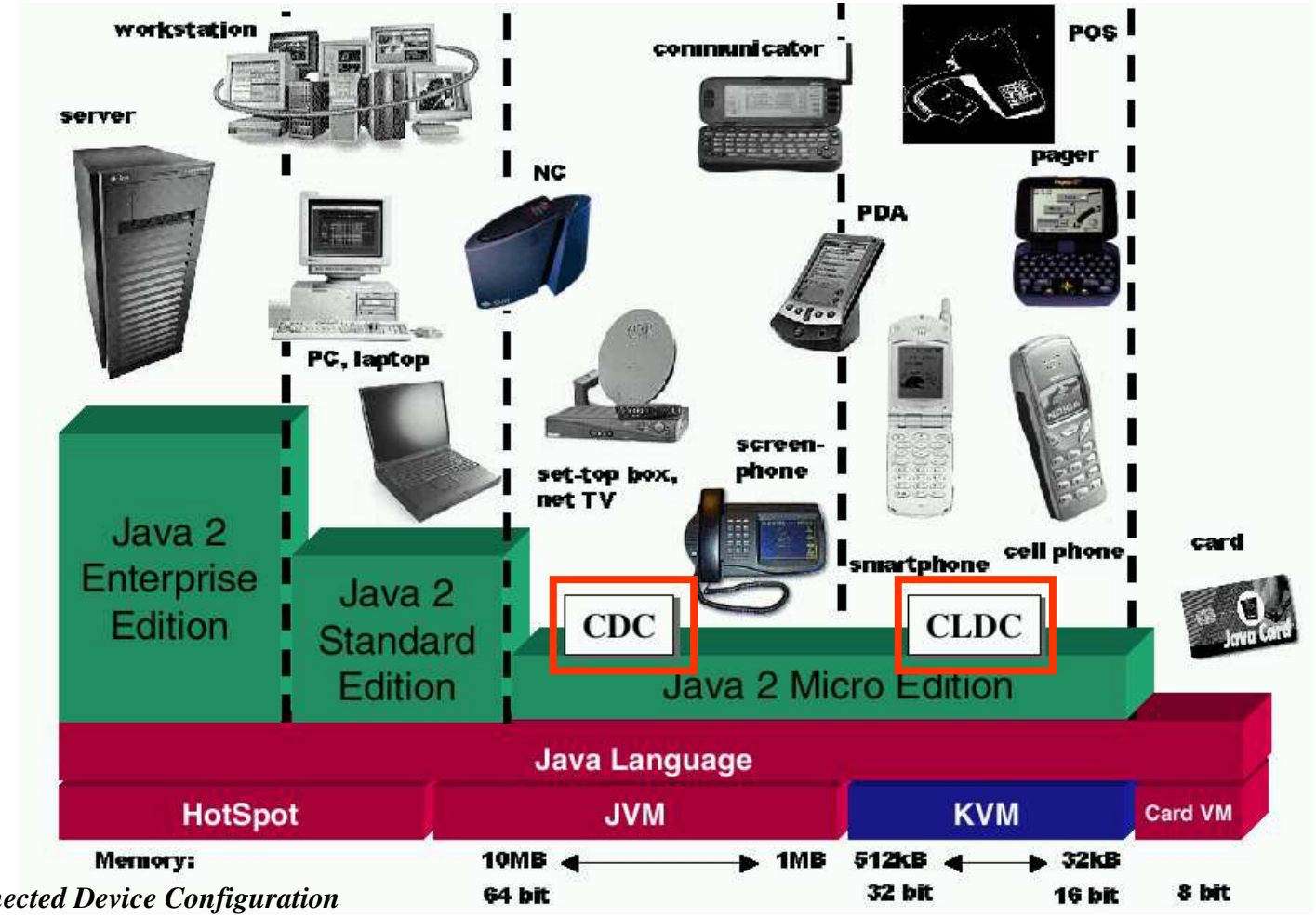
- Java EE (J2EE) vise un usage professionnel pour une large gamme de systèmes composés de quelques dizaines à quelques millions d'utilisateurs. Java EE est mis en œuvre pour des services Web. Les équipements utilisés sont très performants.

- Java SE (J2SE) fournit un environnement complet de développement d'applications pour les *desktops* ou les serveurs. J2SE compose le socle de Java EE.

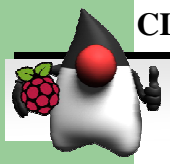
- Java ME (J2ME) propose lui un environnement de développement qui s'adresse aux besoins de l'embarqué : *smartphone*, PDA, *set-top* box, imprimante, objet connecté, IoT…



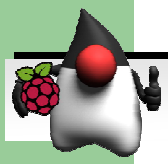
JAVA EE, JAVA SE, JAVA ME



CDC : ***Connected Device Configuration***
CLDC : ***Connected Limited Device Configuration***



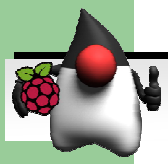
PARTIE 3
PRESENTATION DE JAVA ME



JAVA ME

- *« Java ME fournit un environnement robuste et flexible pour les applications qui s'exécutent sur des périphériques intégrés et* ***mobiles*** *: téléphones mobiles, décodeurs, lecteurs Blu-ray, appareils multimédia numériques, modules M2M, imprimantes… »*

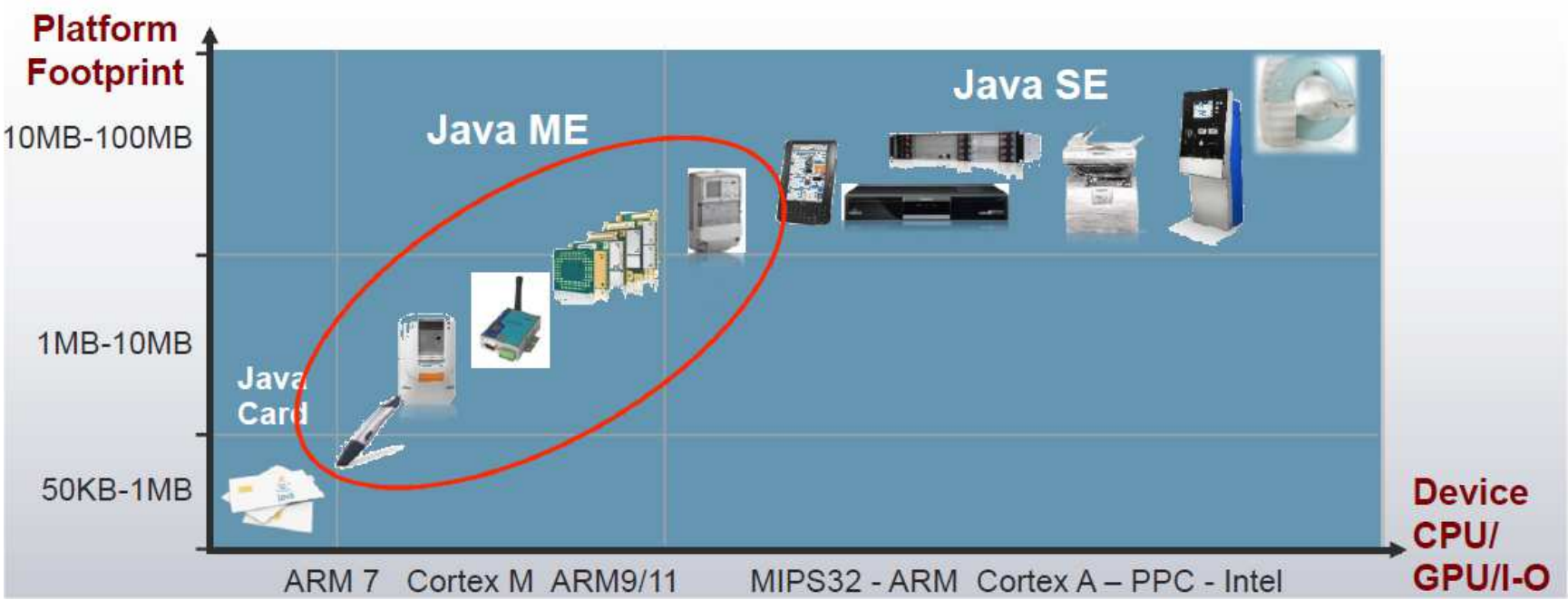
- *« Java ME s'adapte à un environnement limité et permet de créer des applications Java qui s'exécutent sur de petits périphériques dont les capacités d'alimentation, d'affichage et de mémoire sont restreintes. »*



JAVA ME

- Java ME vise donc des équipements (*device)* avec une quantité mémoire limitée, un *display* limité, une puissance limitée :
  - Processeurs 16, 32 bits.
  - Processeurs ARM, Cortex.
  - Quelques dizaines de Ko à quelques Mo de mémoire.

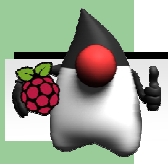
JAVA ME


Platform
Footprint
10MB-100MB
1MB-10MB
50KB-1MB
Java ME
Java SE
Java
Card
ARM 7
Cortex M
ARM9/11
MIPS32 - ARM
Cortex A – PPC - Intel
Device
CPU/
GPU/I-O

JAVA ME

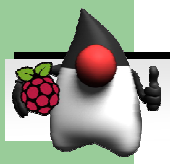
- Java ME retient les bases de Java et permet de développer des applications sur des appareils légers et mobiles.

- Java ME utilise le même langage et les mêmes outils que les versions Java SE ou Java EE mais avec une API réduite :
  - Analogie avec le VHDL de simulation et le VHDL de synthèse !

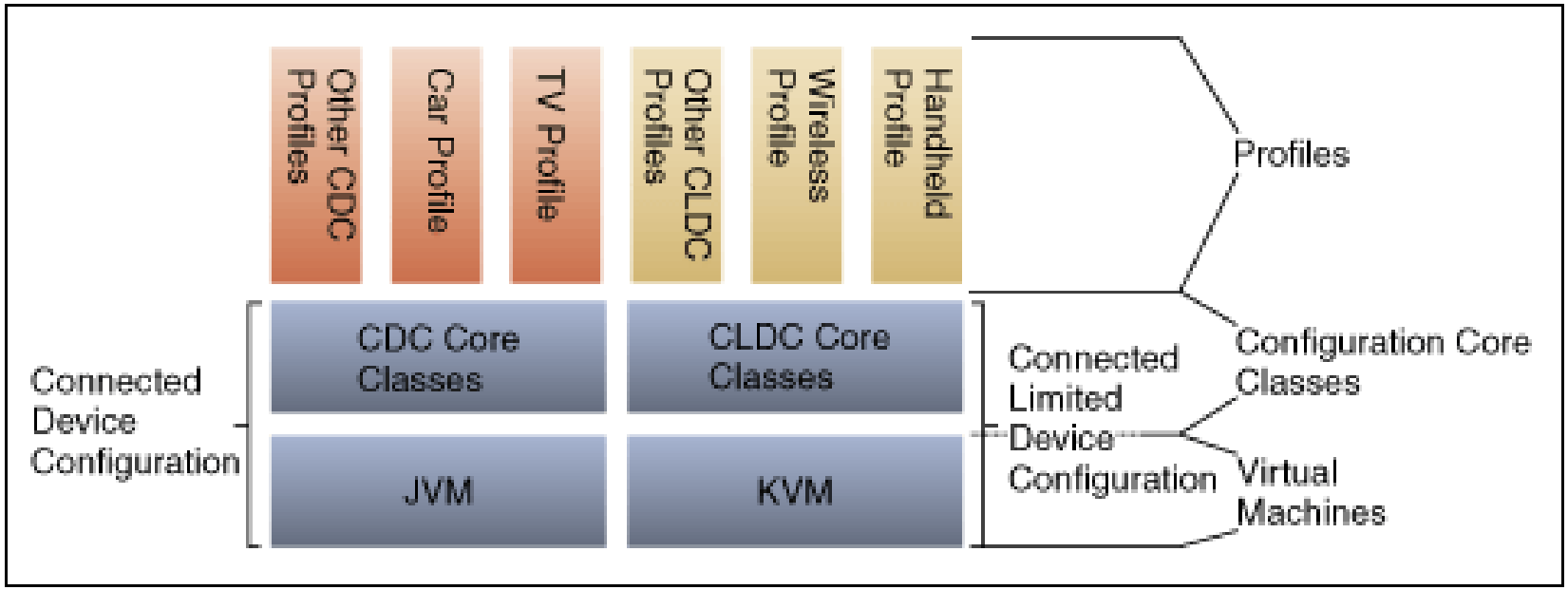


JAVA ME

- La plateforme Java ME est divisée en trois parties :
  - Les configurations. Une configuration offre un ensemble de base de bibliothèques (API) et de capacités de machine virtuelle pour une large gamme d'appareils (*devices*).
  - Les profils. Un profil est un ensemble d'API qui s'applique à une gamme d'appareils particulier.
  - Les paquetages (*packages*) optionnels. Un paquetage optionnel regroupe un ensemble d'API spécifiques à une technologie donnée.



JAVA ME

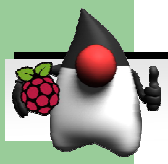

Other CDC Profiles
Car Profile
TV Profile
Other CLDC Profiles
Wireless Profile
Handheld Profile
Profiles
CDC Core Classes
CLDC Core Classes
Connected Device Configuration
Connected Limited Device Configuration
Configuration Core Classes
JVM
KVM
Virtual Machines

CONFIGURATION

- La configuration représente le socle de Java ME. Elle est constituée de la machine virtuelle et des API bas niveau pouvant être utilisées sur un certain type de matériel (sous-ensemble des API de Java SE).

- La configuration garantit la portabilité et l'interopérabilité du code entre les différents types de *devices*.

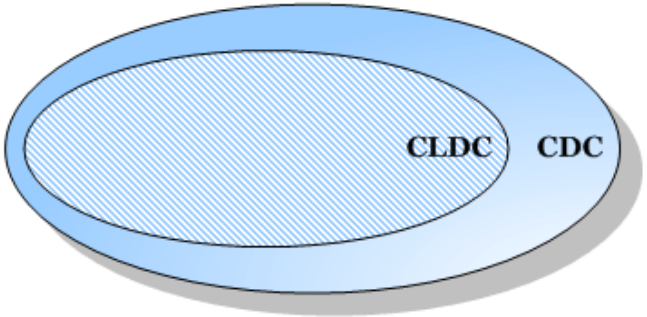
- Il existe actuellement deux types de configuration :
  - CLDC : *Connected Limited Device Configuration*.
  - CDC : *Connected Device Configuration*.

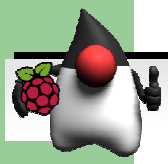


CONFIGURATION

- CLDC est la configuration utilisée sur les petits *devices* avec une connexion réseau intermittente et une alimentation sur batterie (téléphones mobiles, PDA, organiseurs personnels de base, équipements M2M…).
  - KVM (*Kilobyte Virtual Machine*). 40 à 80 Ko.
  - De 160 Ko à 512 Ko de mémoire.

- CDC est la configuration utilisée sur les *devices* plus évolués comme les *smartphones* et les *set-top box*.
  - JVM (*Java Virtual Machine*).
  - Plus de 512 Ko de mémoire.

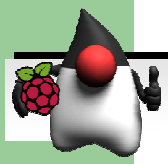




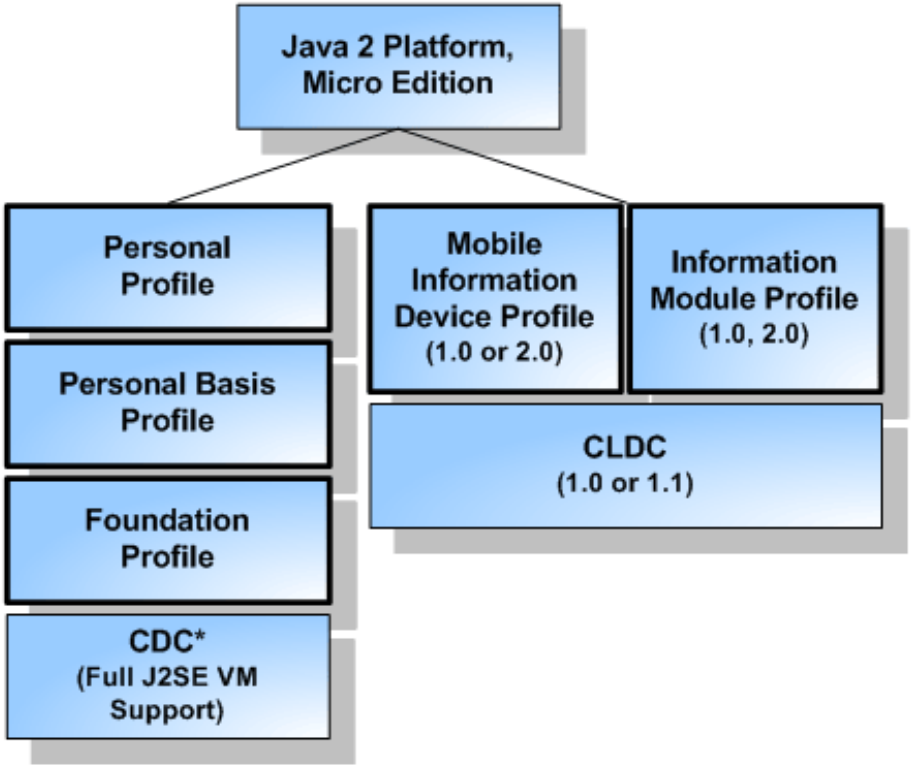
PROFIL

- Le profil est un ensemble d'API particulières à un type de machine ou à une fonctionnalité spécifique. Il prend en charge les fonctionnalités de plus haut niveau.

- Le profil permet de contrôler le cycle de vie d'une application, de construire une interface graphique…



PROFIL

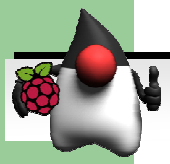


*CDC 1.0 is based on J2SE 1.3
*CDC 1.1 is based on J2SE 1.4

PACKAGES OPTIONNELS

- Les *packages* optionnels fournissent des fonctionnalités supplémentaires non associées à une configuration ou à un profil.

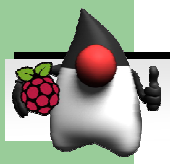
- Par exemple, l'API Bluetooth (JSR 82) fournit des méthodes pour utiliser le réseau Bluetooth et peut être implémentée sur n'importe quelle combinaison de configuration et de profil.



JAVA SPECIFICATION REQUEST

- Les spécifications de Java ME sont définies par l'organisation JCP (*Java Community Process*).

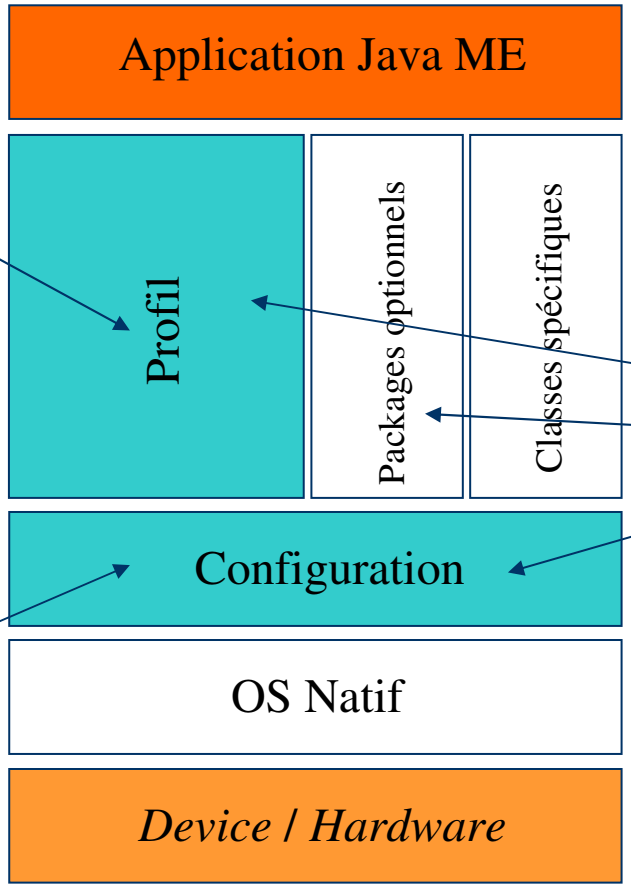
- Chaque spécification est définie par un numéro de JSR (*Java Specification Request*) :
  - API Bluetooth : JSR 82.
  - CLDC 1.1 : JSR 139.
  - MIDP 2.0 : JSR 118.



CONFIGURATION, PROFIL, JSR…

Un profil définit un ensemble d'API pour une famille de *devices*

Une configuration définit une API de base en fonction des ressources matérielles disponibles

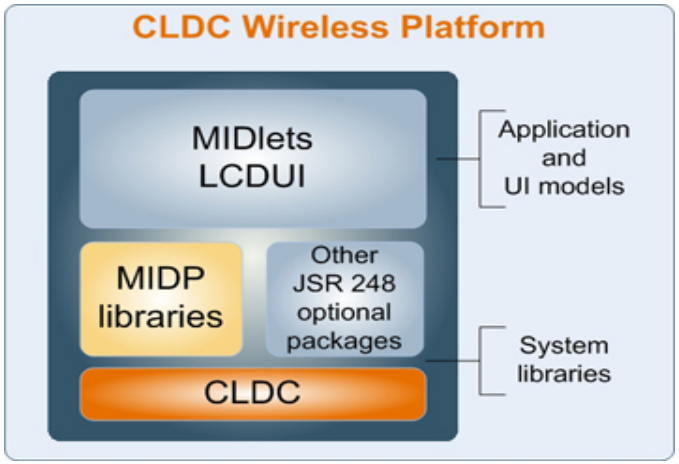


Java Specific Request (JSR)

CONFIGURATION CLDC

- La configuration CLDC est conçue pour remplir les besoins d'exécution d'applications sur les appareils limités en mémoire, capacité de calcul et capacité graphique.

- On associe généralement le profil MIDP (*Mobile Information Device Profil*) qui fournit un environnement complet de développement d'applications pour téléphones mobiles par exemple.

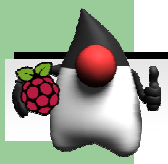


CONFIGURATION CLDC

- CLDC et MIDP représentent ainsi l'environnement Java ME le plus courant. La spécification « *Java Technology for the Wireless Industry* » (JSR 185) associe CLDC 1.0 (ou 1.1) et MIDP 2.0 pour créer un environnement de développement J2ME.

- Les applications de cet environnement sont appelées *MIDlets*.

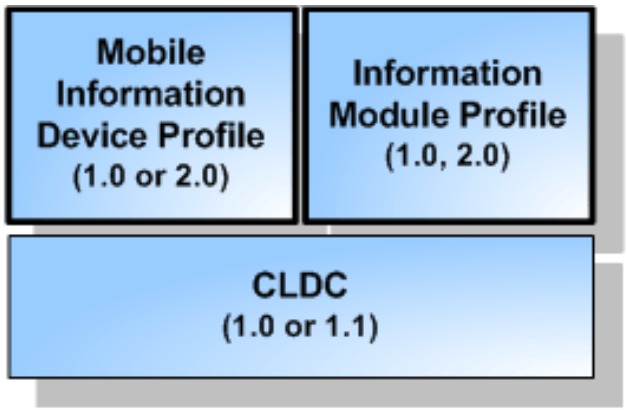
- Une *MIDlet* est donc une application Java. Elle consiste en au moins une classe Java.

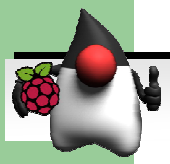


CONFIGURATION CLDC

- Le profil MIDP 1.0 (JSR 37) apporte les fonctionnalités centrales requises par les applications sur téléphone mobile avec une interface utilisateur et une sécurité réseau de base.

- Le profil MIDP 2.0 (JSR 118) étend MIDP 1.0 avec une interface utilisateur avec des fonctionnalités multimédia, une connectivité réseau plus riche, le téléchargement d'applications en ligne et une sécurité complète.

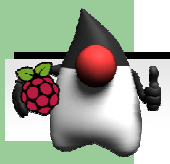




CONFIGURATION CLDC

- Le profil MIDP 2.0 apporte les connectivités réseau suivantes :
  - HTTP.
  - HTTPS.
  - Datagrammes IP.
  - *Sockets* côté client et côté serveur.
  - Communication par une liaison série RS-232.

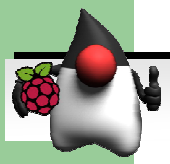
- Les accès aux données, applications et autres ressources du réseau et du terminal sont protégés. Une *MIDlet* n'a aucun privilège et doit être signée pour obtenir un accès à une ressource (comme une *applet* Java).



CONFIGURATION CLDC

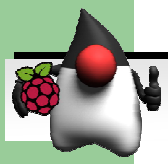
- La configuration CLDC supporte un certain nombre de *packages* optionnels qui permettent au concepteur d'adapter précisément les besoins de son application aux ressources physiques disponibles et qui donnent un accès facilité aux composants spécifiques d'un *device*.

- Quelques exemples :
  - API *Wireless Messaging* (WMA).
  - API *Mobile Media* (MMAPI).



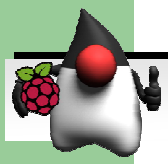
CONFIGURATION CLDC

- *Packages* supportés par la configuration CLDC 1.1 :
  - java.io : E/S par *streams*.
  - java.lang : sous-ensemble des classes de base de Java SE.
  - java.util : classes utilitaires (*Calendar, Date, Vector*…).
  - javax.microedition.io : classes générales pour les fonctions réseau.



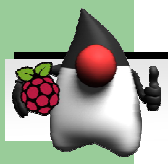
CONFIGURATION CLDC

- *Packages* supportés par la configuration CLDC 1.1 :
  - java.io.* : *ByteArrayInputStream, ByteArrayOutputStream, DataInput, DataOutput, DataInputStream, DataOutputStream, InputStream, OutputStream, InputStreamReader, OutputStreamWriter, PrintStream, Reader, Writer.*
  - java.lang.* : *Boolean, Byte, Character, Class, Integer, Long, Math, Object, Runnable, Runtime, Short, String, StringBuffer, System, Thread, Throwable.*
  - java.util.* : *Calendar, Date, Enumeration, Hashtable, Random, Stack, TimeZone, Vector.*



CONFIGURATION CLDC

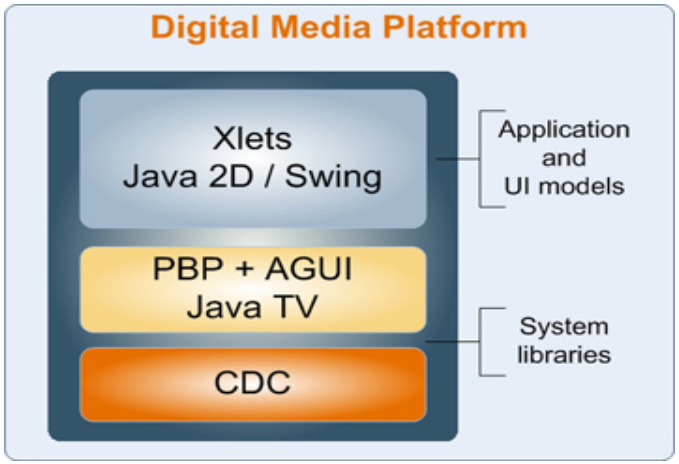
- *Packages* supportés par le profil MIDP 2.0 :
  - java.lang : extension de CLDC 1.1.
  - java.util : extension de CLDC 1.1.
  - javax.microedition.io : extension communications réseau (HTTP, UDP, *socket*…).
  - javax.microedition.lcdui : interface utilisateur.
  - javax.microedition.media : gestion du son.
  - javax.microedition.midlet : définition d'une application et de son cycle de vie.
  - javax.microedition.pki : certificats pour l'authentification des connexions sécurisées.
  - …

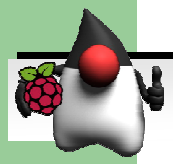


CONFIGURATION CDC

- La configuration CDC concerne des *devices* ayant plus de capacités avec une connexion réseau en se rapprochant des possibilités de Java SE (implémentation complète de la JVM) et tout en respectant les contraintes imposées par leurs ressources.

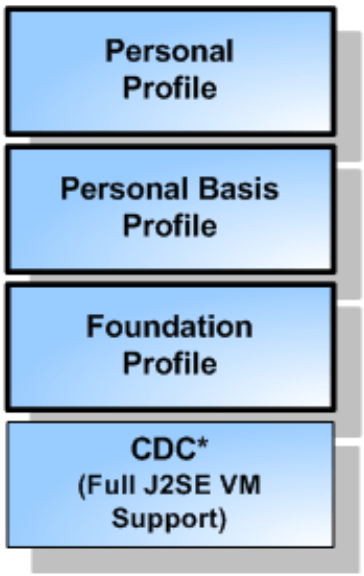
- La configuration CDC apporte :
  - La compatibilité et la sécurité de Java.
  - Les API Java traditionnels.



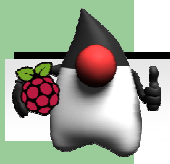


CONFIGURATION CDC

- La configuration CDC définit 3 profils :
  - *Foundation Profile* (JSR 219).
  - *Personal Basis Profile* (JSR 217).
  - *Personal Profile* (JSR 216).



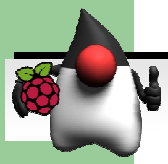
*CDC 1.0 is based on J2SE 1.3
*CDC 1.1 is based on J2SE 1.4



CONFIGURATION CDC

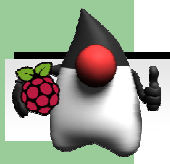
- Le profil *Foundation Profile* :
  - Ne permet pas de développer d'IHM.
  - Il doit être associé à l'un des deux profils suivants : *Personal Basic Profile* ou *Personal Profile*.

- Le profil *Personal Basis Profile* :
  - Permet le développement d'application réseau.
  - Fournit l'environnement graphique léger style AWT.



CONFIGURATION CDC

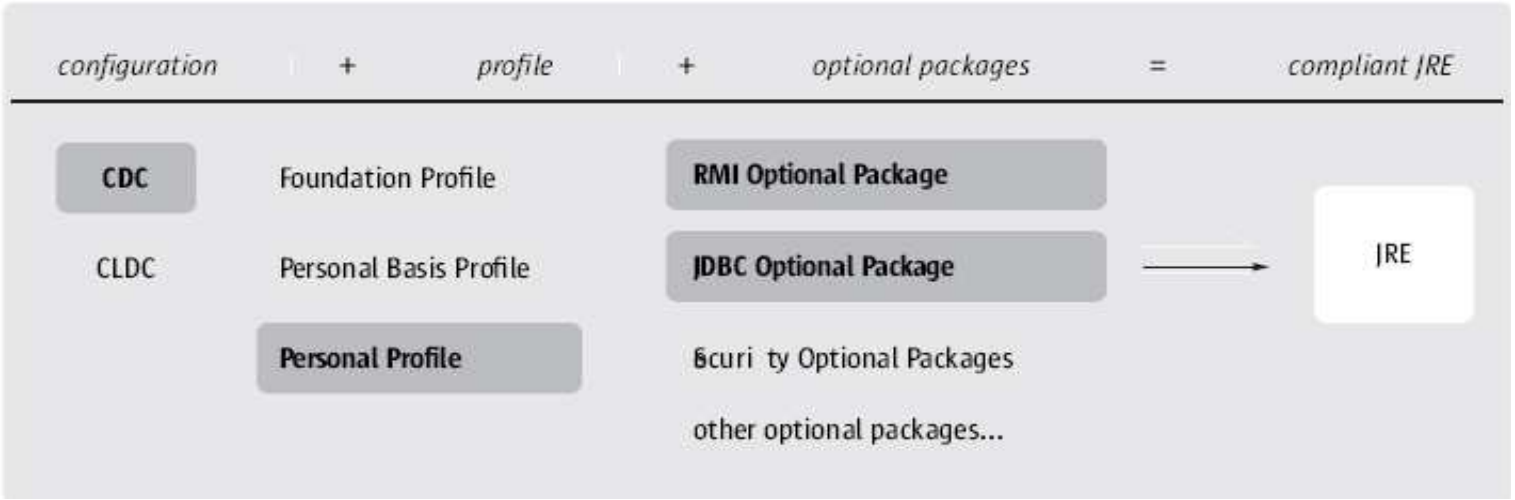
- Le profil *Personal Profile* :
  - Permet le développement complet d'une IHM et d'applications Java AWT.
  - Supporte AWT comme dans le JDK 1.1.
  - Est adapté pour les *applets* Web.

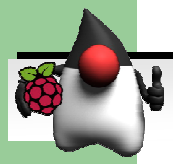


CONFIGURATION CDC

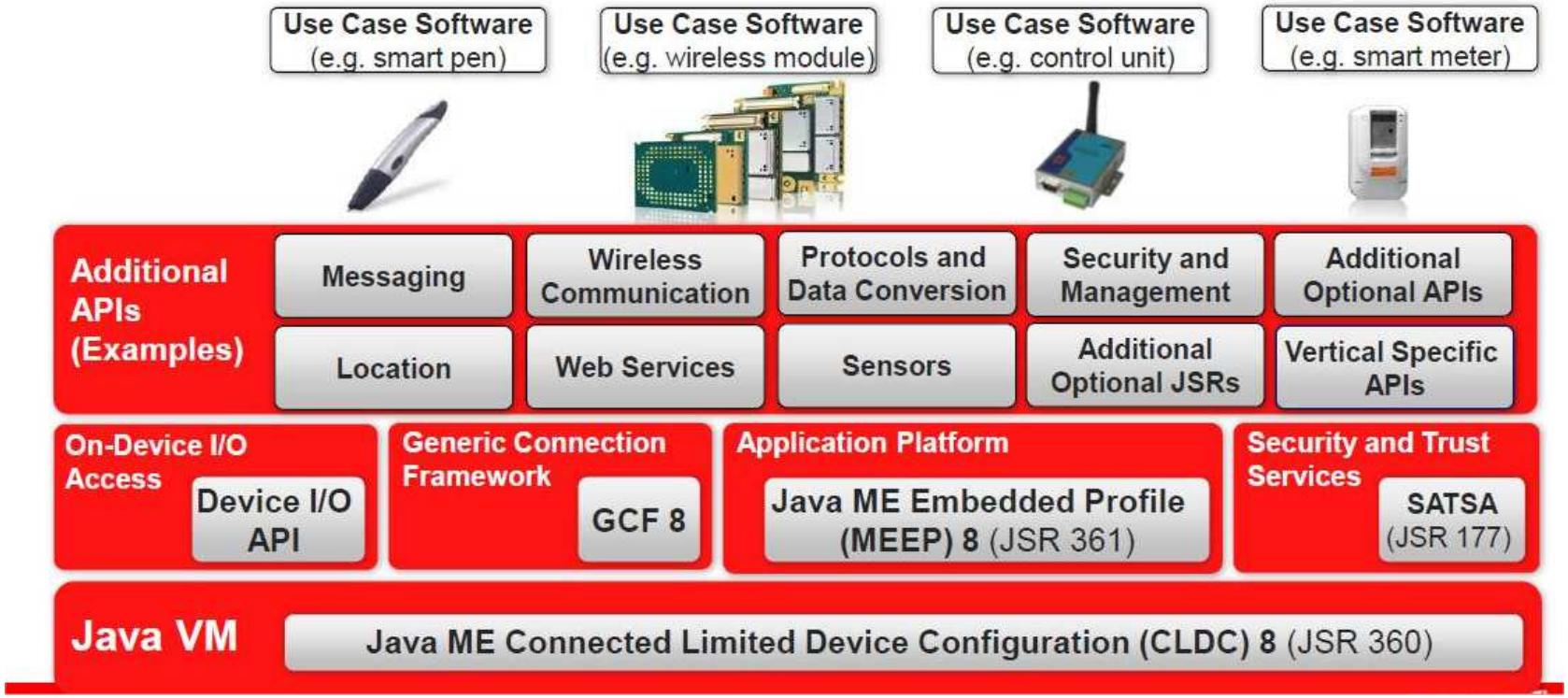
- La configuration CDC supporte aussi un certain nombre de *packages* optionnels.

- Quelques exemples :
  - API *Remote Method Invocation* (RMI).
  - API *Java Database Connectivity* (JDBC).



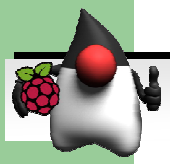


PLATEFORME ORACLE JAVA ME



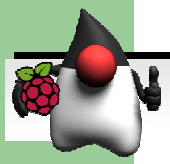
PARTIE 4

DEVELOPPEMENT ET MISE EN OEUVRE D'UNE APPLICATION JAVA ME



ETAT D'UNE MIDLET

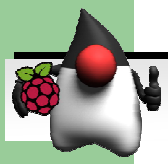
- La *MIDlet* est l'application Java ME à développer.
- Elle est développée de façon classique comme une application autonome Java SE.
- Pour rappel :
  - Application Java autonome : application Java.
  - Application Java côté client Web : *applet*.
  - Application Java côté serveur Web : *servlet*.
  - Application Java ME : *MIDlet*.
  - Application Java Card : *cardlet*.



ETAT D'UNE MIDLET

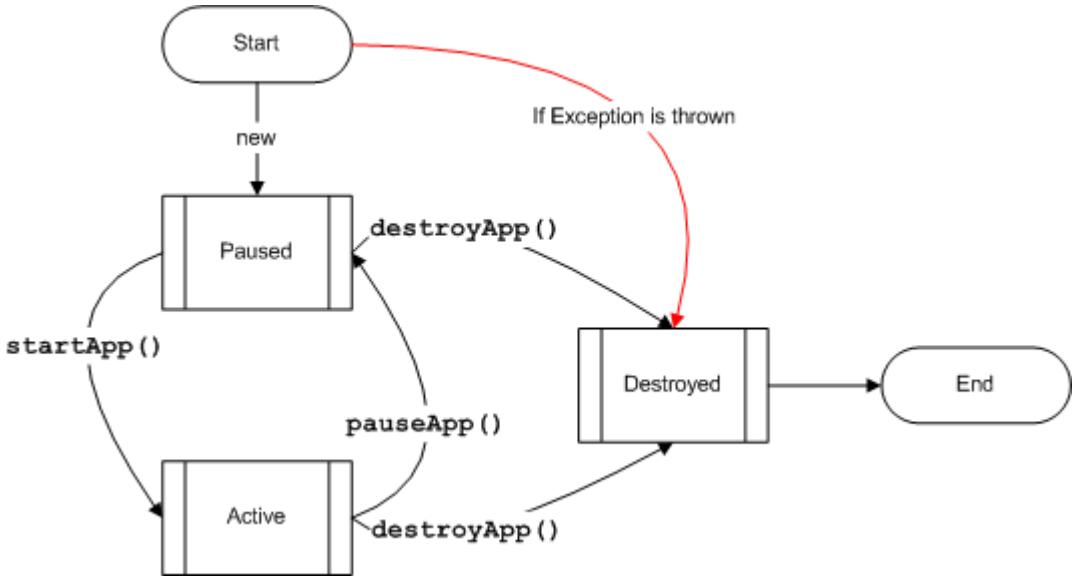
- La classe principale d'une *MIDlet* doit être une sous-classe de javax.microedition.midlet.MIDlet.

- Cette sous-classe définit 3 méthodes de notification liées au cycle de vie de la *MIDlet* :
  - *startApp()*.
  - *pauseApp()*.
  - *destroyApp()*.



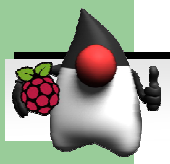
ETAT D'UNE MIDLET

- Il y a 3 états possibles liés au cycle de vie d'une *MIDlet* :
  - *Paused* : l'instance de la *MIDlet* a été construite et est inactive.
  - *Active* : l'instance de la *MIDlet* est active.
  - *Destroyed* : l'instance de la *MIDlet* est terminée et est prête à être éliminée par le ramasse-miettes Java.



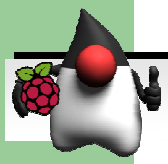
ETAT D'UNE MIDLET

- Les *MIDlets* sont créés par l'AMS (*Application Management System*) suite à une requête de l'usager.
- L'état initial d'une *MIDlet* est *Paused*.
- Après la construction de la *MIDlet*, l'AMS l'active et invoque sa méthode *startApp()*. Après que la méthode *startApp()* ait rendu le contrôle, l'état de la *MIDlet* passe de *Paused* à *Active*.



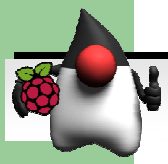
ETAT D'UNE MIDLET

- La destruction de la *MIDlet* intervient lorsque la *MIDlet* passe dans l'état *Destroyed*.
  - Si la destruction est déclenchée par l'AMS, la méthode *destroyApp()* de la *MIDlet* est invoquée.
  - Si la *MIDlet* se suicide, la méthode *destroyApp()* de la *MIDlet* n'est pas invoquée.



ETAT D'UNE MIDLET

```
import javax.microedition.midlet.*;

public class BasicMIDlet extends MIDlet {
   public BasicMIDlet(){
      // constructor
   }
   destroyApp( boolean unconditional ) throws
   MIDletStateChangeException {
      // called when the system destroys the MIDlet
   }
   pauseApp(){
      // called when the system pauses the MIDlet
   }
   startApp() throws MIDletStateChangeException {
     // called when the system activates the MIDlet
   }
}
```

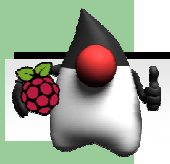


STRUCTURE D'UNE MIDLET

- Une *MIDlet* (ou plusieurs) est empaquetée dans un fichier JAR (*Java Archive*) (fichier *.jar*).

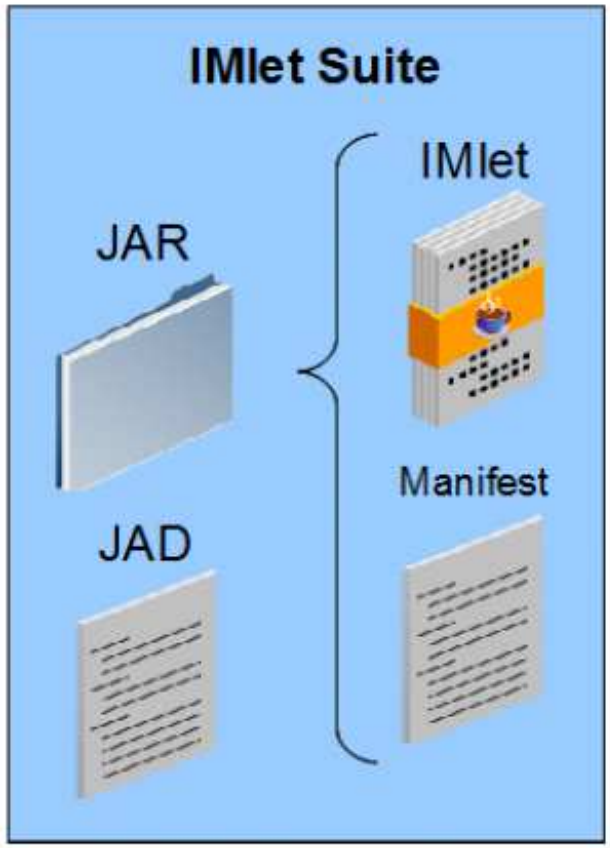
- Le fichier JAR contient les éléments suivants :
  - Les fichiers *.class* de la *MIDlet*.
  - Les fichiers ressource utilisés par la *MIDlet*.
  - Un fichier *Manifest* qui décrit le contenu du fichier JAR.

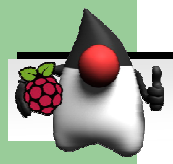
- Un fichier JAD (*Java Application Descriptor*) (fichier *.jad*) accompagne aussi le fichier JAR et fournit des informations sur le déploiement de la *MIDlet* (nom, version, taille…).



STRUCTURE D'UNE MIDLET

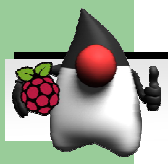
- Le tout forme une suite *MIDlet*. Par abus de langage, la suite *MIDlet* est aussi appelée *MIDlet*…





STRUCTURE D'UNE MIDLET

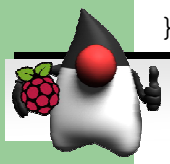
- Exemple : suite *MIDlet Hello* :
  - Fichier *Hello.jar*.
  - Fichier *Hello.jad*.



STRUCTURE D'UNE MIDLET

- Code source *Hello.java* :

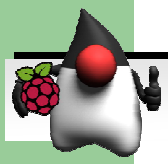
```
package hello;
import javax.microedition.midlet.MIDlet;

public class Hello extends MIDlet {
    public void startApp() {
        System.out.println("Starting...");
        int i;
        i = 0;
        while (true) {
            try {
                Thread.sleep(1000);
              i++;
                System.out.println("Hello World: " + i);
            } catch(InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }
    public void destroyApp(boolean unconditional) {
        System.out.println("Destroyed...");
    }
}
```



STRUCTURE D'UNE MIDLET

- Contenu du fichier *Hello.jad* :

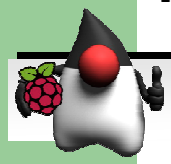
```
MIDlet-1: Hello,,hello.Hello
MIDlet-Jar-Size: 1895
MIDlet-Jar-URL: Hello.jar
MIDlet-Name: Hello
MIDlet-Vendor: kadionik
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.8
MicroEdition-Profile: MEEP-8.0
```



STRUCTURE D'UNE MIDLET

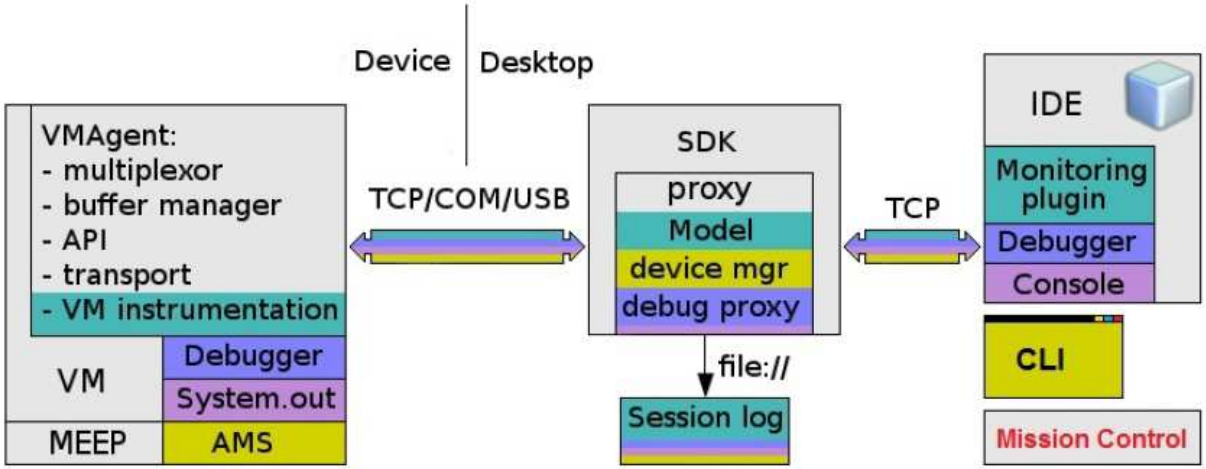
- Contenu du fichier *Hello.jar* :

```
$ jar -tvf Hello.jar
     0 Wed May 25 16:53:02 CEST 2016 META-INF/
   269 Wed May 25 16:53:00 CEST 2016 META-INF/MANIFEST.MF
     0 Wed May 25 16:53:00 CEST 2016 hello/
  1188 Wed May 25 16:53:00 CEST 2016 hello/Hello.class

$ jar -xvf Hello.jar
$ cat META-INF/MANIFEST.MF
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.9.4
Created-By: 1.8.0_65-b17 (Oracle Corporation)
MicroEdition-Profile: MEEP-8.0
MicroEdition-Configuration: CLDC-1.8
MIDlet-1: Hello,,hello.Hello
MIDlet-Vendor: kadionik
MIDlet-Name: Hello
MIDlet-Version: 1.0
```



INTERACTION AVEC L'AMS

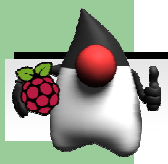
- Il y a différentes possibilités d'interaction avec l'AMS de Java ME .
  - En local sur la carte cible.
  - A distance par le réseau.
  - En ligne de commande ou avec l'IDE (*Integrated Design Entry*).



INTERACTION AVEC L'AMS

- Interaction locale avec l'AMS en ligne de commande (CLI) :

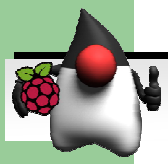
```
rpi-java# installMidlet.sh miniprojet.jar
Java is starting. Press Ctrl-C to exit
Frame buffer device detected: BCM2708
Frame buffer '/dev/fb0' detected: 656x416, depth=16
Failed to open /dev/input/by-id
The suite was successfully installed, ID: 2
[ERROR] [AMS] iso=1:Sending the notification:
   com.oracle.midp.proxy.InstallCommd
```



INTERACTION AVEC L'AMS

- Interaction locale avec l'AMS en ligne de commande (CLI) :

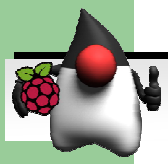
```
rpi-java# listMidlets.sh
Java is starting. Press Ctrl-C to exit
Frame buffer device detected: BCM2708
Frame buffer '/dev/fb0' detected: 656x416, depth=16
Failed to open /dev/input/by-id
Suite: 2
  Name: miniprojet
  Version: 1.0
  Vendor: kadionik
  MIDlets:
    miniprojet: miniprojet.Miniprojet
```



INTERACTION AVEC L'AMS

- Interaction locale avec l'AMS en ligne de commande (CLI) :

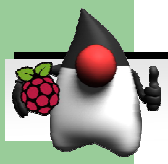
```
rpi-java# runSuite.sh 2
Java is starting. Press Ctrl-C to exit
Frame buffer device detected: BCM2708
Frame buffer '/dev/fb0' detected: 656x416, depth=16
Failed to open /dev/input/by-id
[ERROR] [AMS] iso=1:Sending the notification:
   com.oracle.midp.proxy.RunCommand@d
```



INTERACTION AVEC L'AMS

- Interaction à distance avec l'AMS sur le port d'écoute (65002) :

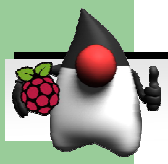
```
oracle>> ams-install file:///C:/some/directory/hello.jar
   hostdownload
<<ams-install,start install,file:///C:/some/directory/hello.jar
<<ams-install,install status: stage DONE, 0%
<<ams-install,install status: stage DONE, 100%
<<ams-install,OK,Install success

oracle>> ams-install http://www.example.com/netdemo.jar
<<ams-install,start install,http://www.example.com/netdemo.jar
<<ams-install,install status: stage DONE, 0%
<<ams-install,install status: stage DONE, 100%
<<ams-install,OK,Install success
```



INTERACTION AVEC L'AMS

- Interaction à distance avec l'AMS sur le port d'écoute (65002) :

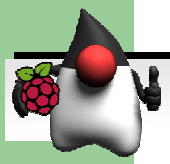
```
oracle>> ams-list
<<ams-list,0.hello|Oracle,STOPPED
<<ams-list,1.netdemo|Oracle,STOPPED
<<ams-list,2.rs232dem|Oracle,RUNNING
<<ams-list,OK,3 suites are installed

oracle>> ams-remove 0
<<ams-remove,OK,hello removed

oracle>> ams-list
<<ams-list,1.netdemo|Oracle,STOPPED
<<ams-list,2.rs232dem|Oracle,RUNNING
<<ams-list,OK,2 suites are installed
```



CREATION D'UNE SUITE MIDLET

- La procédure générale pour créer une suite *MIDlet* à la main inclut les étapes suivantes :
  - 1. Création du code source Java de la *MIDlet*.
  - 2. Compilation de la *MIDlet* pour créer le(s) fichier(s) *.class*.
  - 3. Création du fichier *Manifest*.
  - 4. Empaquetage de(s) fichier(s) *.class* et du fichier *Manifest* dans un fichier JAR *.jar*.
  - 5. Création du fichier JAD *.jad*.

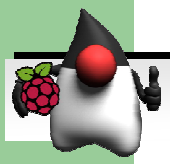
- Le fichier JAR et le fichier JAD forme une suite *MIDlet* prête à être déployée sur le *device*.



CREATION D'UNE SUITE MIDLET

- La procédure générale pour créer une suite *MIDlet* avec un IDE inclut les étapes suivantes :
  - 1. Création d'un projet Java ME avec l'IDE.
  - 2. Ajout du code source d'une ou plusieurs *MIDlet* au projet.
  - 3. Construction du projet.

- L'IDE compile automatiquement les fichiers sources, crée le fichier *Manifest* ainsi que le fichier JAR et le fichier JAD. La suite *MIDlet* est alors prête à être déployée sur le *device*.

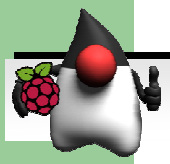


MISE EN ŒUVRE DE JAVA ME

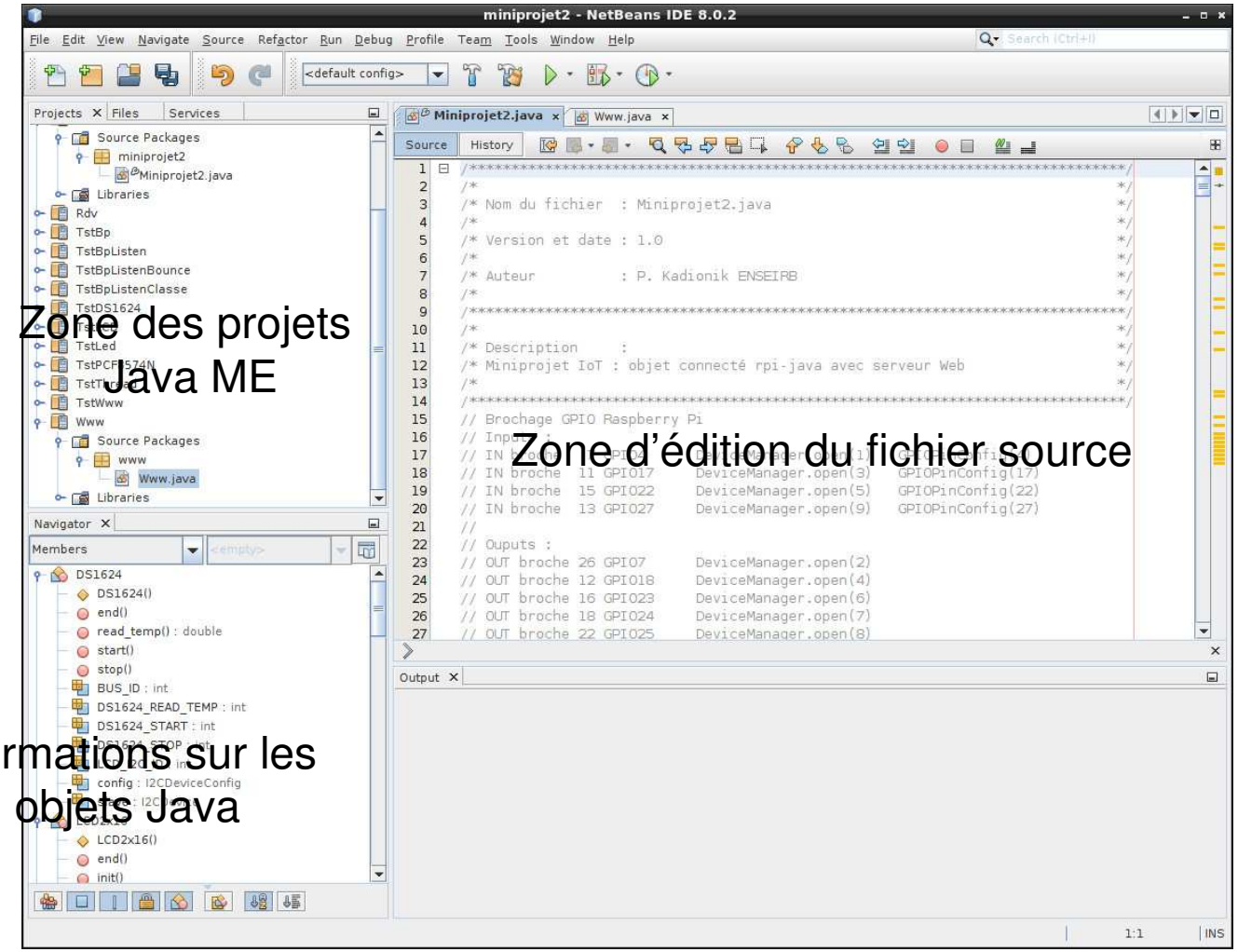
- L'IDE mis en œuvre est NetBeans dans l'environnement Linux.

- NetBeans est un IDE pour développer entre autre des applications Java mais un *plugin* Java ME permet de développer des *MIDlets* Java ME.

- L'application Java ME sera créée en langage Java puis compilée et empaquetée pour créer une suite *MIDlet*. Un agent s'exécutant sur le PC de développement permet d'envoyer la suite *MIDlet* à un autre agent s'exécutant sur la carte cible (carte rpi-java) qui la fera alors exécuter par la machine virtuelle Java ME.



NETBEANS



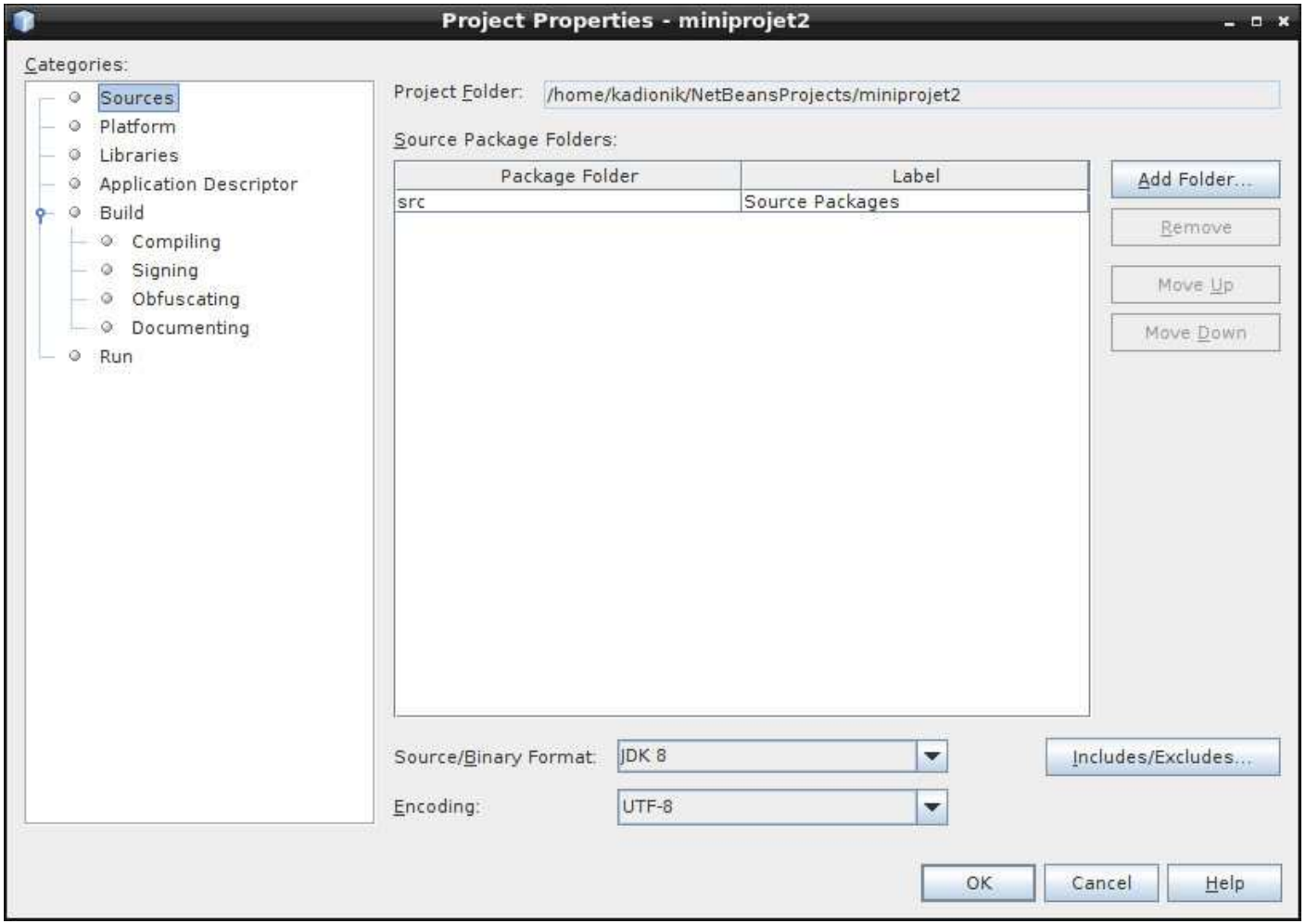
Zone des projets Java ME

Zone d'édition du fichier source

Informations sur les objets Java

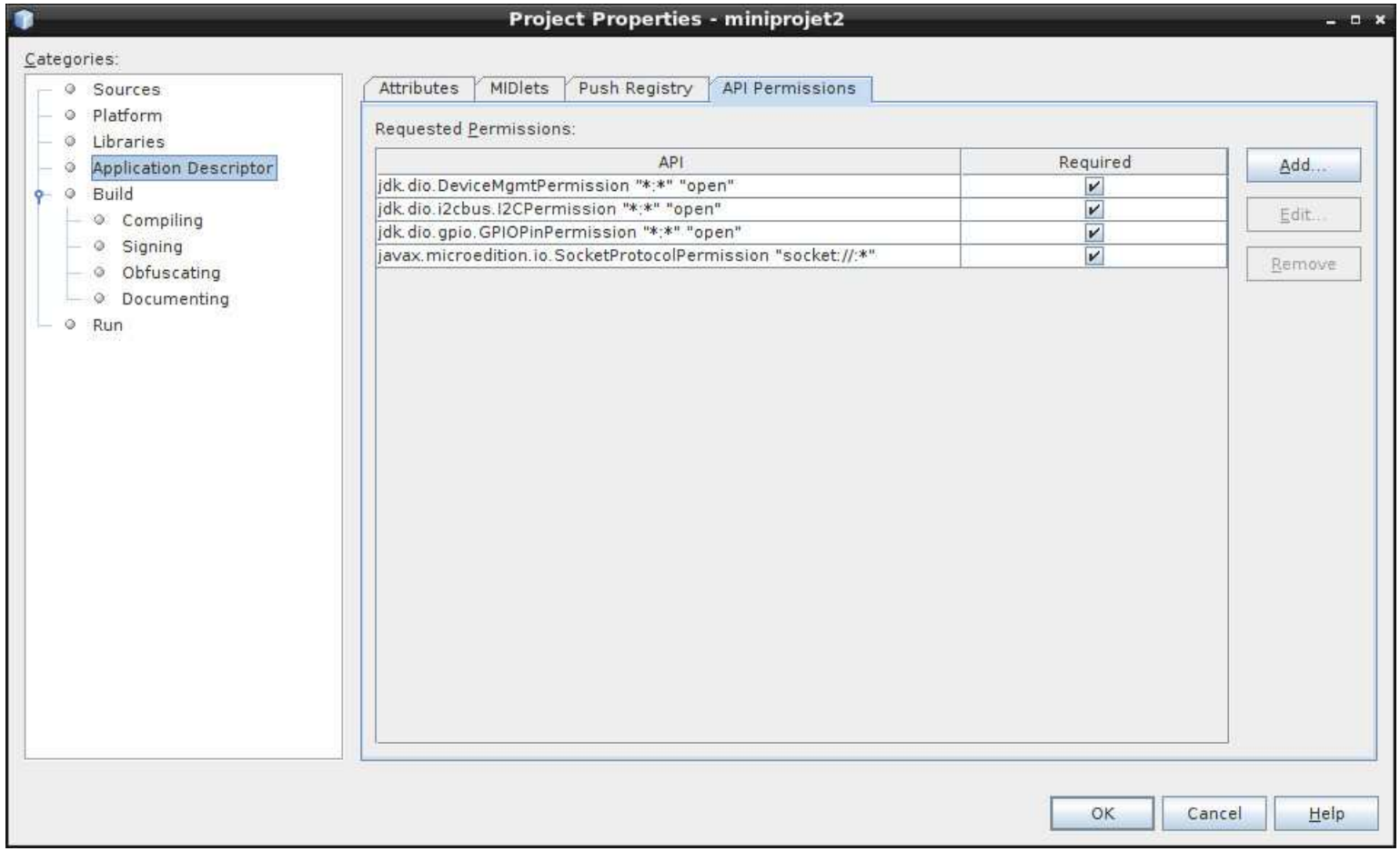
Interface graphique de NetBeans

NETBEANS



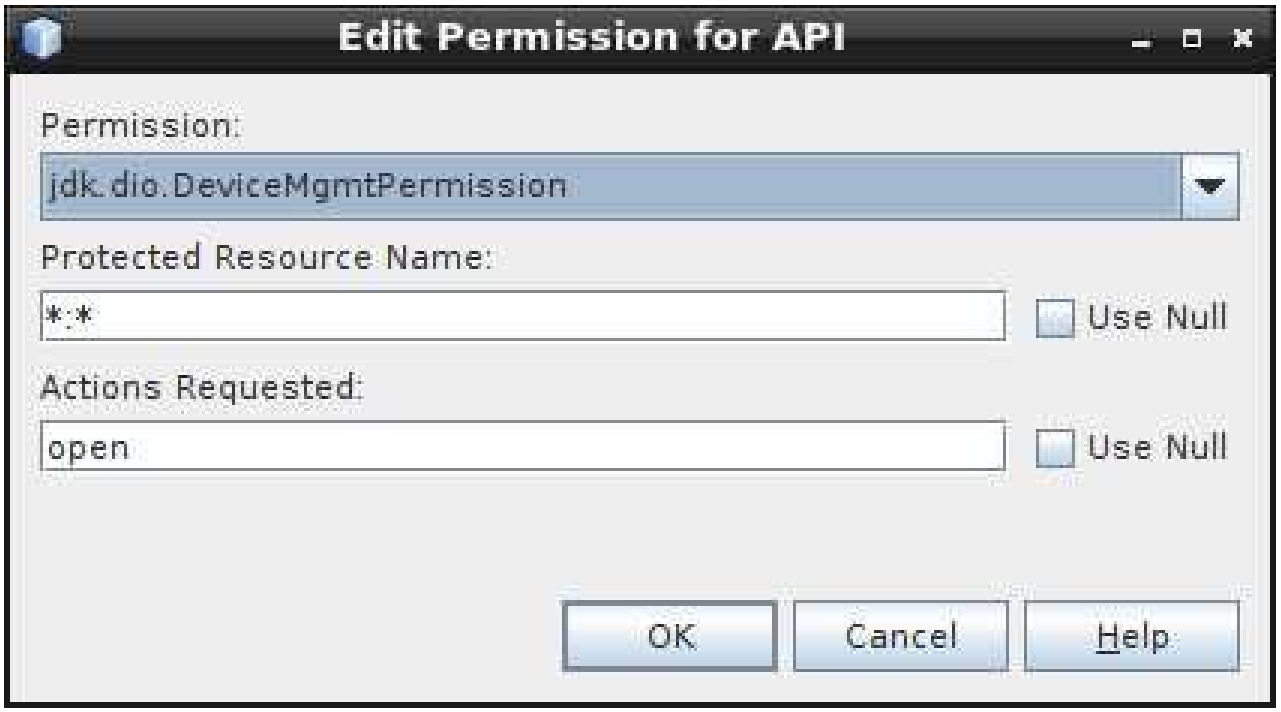
Permissions de la *MIDlet* pour accéder aux E/S

NETBEANS



Permissions de la *MIDlet* pour accéder aux E/S

NETBEANS



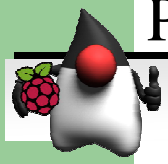
Permissions de la *MIDlet* pour accéder aux E/S

NETBEANS

- Une *MIDlet* doit avoir les droits autorisés pour accéder aux E/S et aux ressources qu'elle désire utiliser.

E/S	Permission	Valeur 1	Valeur 2
BP et leds	jdk.dio.DeviceMgmtPermission	*:*	open
BP et leds	jdk.dio.gpio.GPIOPinPermission	*:*	open
Afficheur et capteur I2C	jdk.dio.i2cbus.I2CPermission	*:*	open
Serveur *socket*	javax.microedition.io.SocketProtocolPermission	socket://:*	

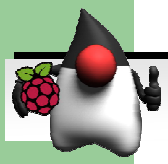
Permissions de la *MIDlet* pour accéder aux E/S et aux ressources



CARTE CIBLE RPI-JAVA

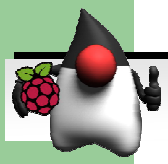
- La carte cible rpi-java est une carte « maison » construite autour d'une carte Raspberry RPi 3B.

- Elle est complétée d'une carte d'entrées/sorties (E/S) connectée à l'aide d'un connecteur 2x20 broches au connecteur d'E/S de la carte RPi.

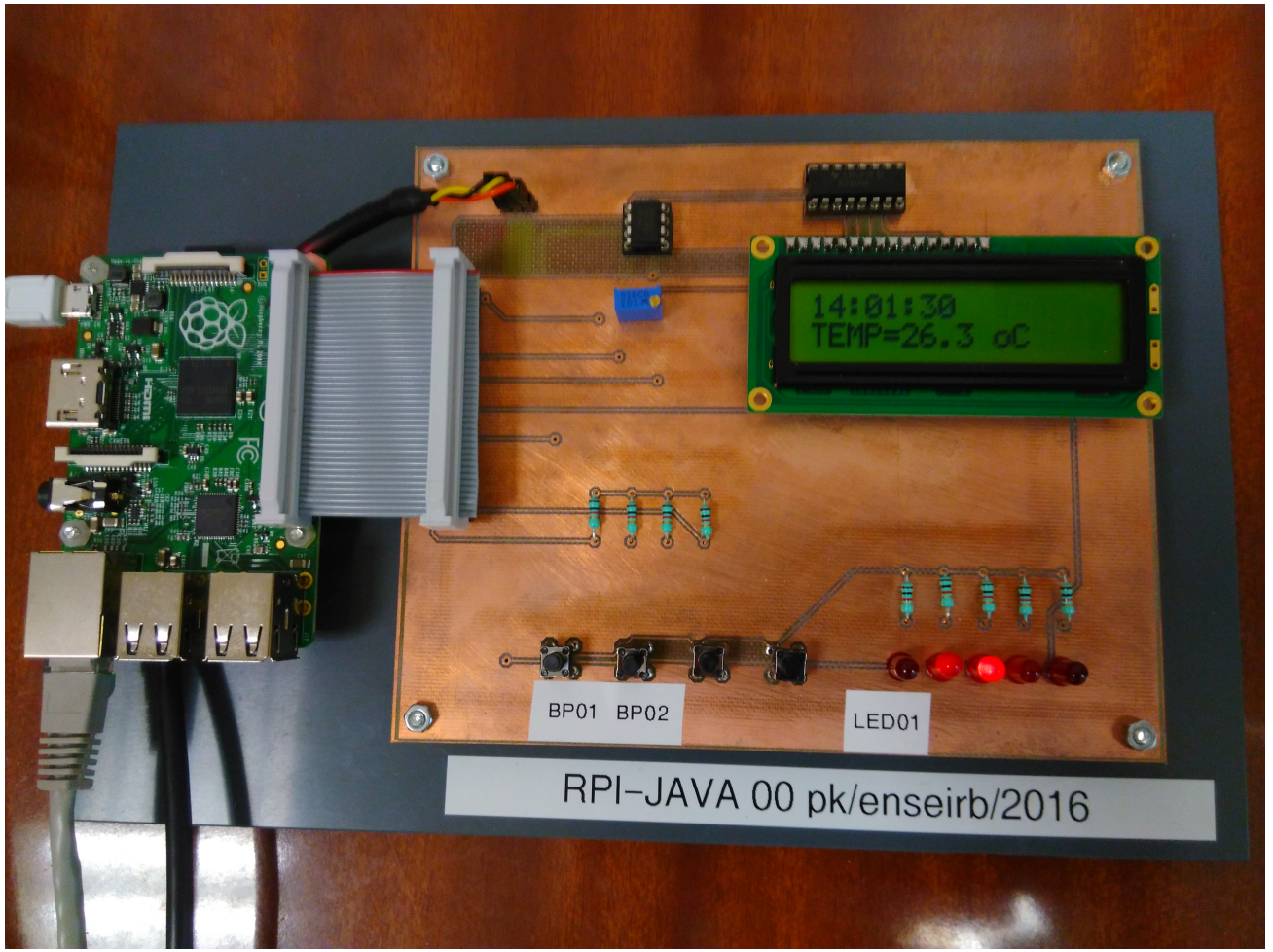


CARTE CIBLE RPI-JAVA

- La carte d'E/S de le carte cible rpi-java possède :
  - 5 leds : LED1 à LED5.
  - 4 boutons poussoirs : BP1 à BP4.
  - Un afficheur LCD 2x16 caractères interfacé sur le bus I2C de la carte RPi.
  - Un capteur de température numérique DS1624 interfacé sur le bus I2C de la carte RPi.



CARTE CIBLE RPI-JAVA



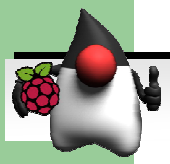
Carte cible rpi-java

CARTE CIBLE RPI-JAVA

- En février 2016, pour le 4e anniversaire de la commercialisation du premier modèle, la fondation Raspberry Pi annonce la sortie de la Raspberry Pi 3.

- Comparé à la RPi 2, elle dispose d'un processeur Broadcom BCM2837 64 bits à quatre cœurs ARM Cortex-A53 à 1,2 GHz et d'une puce Wifi 802.11n et Bluetooth 4.1 intégrée.

- La vitesse d'horloge est 33 % plus rapide que la RPi 2, ce qui permet d'avoir un gain d'environ 50 à 60 % de performances en plus en mode 32 bits.



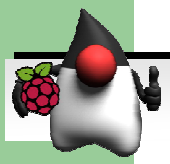
CARTE CIBLE RPI-JAVA

Carte Raspberry Pi 3 modèle B

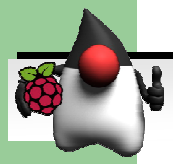
CARTE CIBLE RPI-JAVA

Modèle A	Modèle A+	Modèle 1 B	Modèle 1 B+	Modèle Zero	Modèle 2 B	Modèle 3 B	Modèle 3 B+	Modèle 4 B
25 $	20 $	35 $	35 $	5 $	35 $	35 $	35 $	Jusqu'à 75 $
700 MHz ARM1176JZF-S (ARM11)				1 GHz ARM1176JZF-S	900 MHz quadricœur ARM Cortex-A7	1,2 GHz quadricœur ARM Cortex-A53	1,4 GHz quadricœur ARM Cortex-A53	1,5 GHz quadricœur ARM Cortex-A72
256 Mo		512 Mo			1 Go			Jusqu'à 8 Go

Modèles de cartes Raspberry Pi

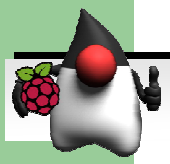


PARTIE 5
API JAVA ME POUR LA CARTE RPI-JAVA



API JAVA ME POUR CARTE RPI-JAVA

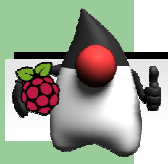
- La carte rpi-java supporte Java ME.
- Java ME a été étendu avec une API Java spécifique pour piloter les E/S de la carte Raspberry Pi.
- Actuellement, Java ME supporte les cartes suivantes :
  - Carte RPi (modèle B et supérieur).
  - Carte STMicroelectronics STM32429I-EVAL (CortexM4/RTX).
  - Carte STMicroelectronics 32F746GDISCOVERY (Cortex-M7/RTX).
  - Carte Intel Galileo Gen. 2.



API JAVA ME POUR CARTE RPI-JAVA

- Il convient d'étudier l'API Java ME pour contrôler les E/S de la carte rpi-java.

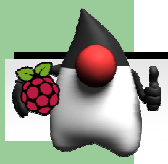
- L'API permet :
  - D'accéder et contrôler les GPIO.
  - D'accéder et contrôler le bus I2C.
  - D'accéder et contrôler le bus SPI.
  - D'accéder et contrôler la liaison série RS-232.



API JAVA ME ET GPIO

- Il convient d'étudier l'API Java ME pour contrôler les GPIO de la carte rpi-java.

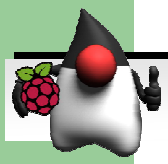
- On utilisera les méthodes suivantes pour accéder aux BP et aux leds de la carte :
  - `DeviceManager.open()`.
  - `GPIOPinConfig()`.



LEDS

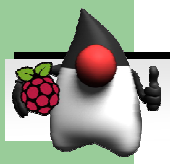
- On utilisera le tableau suivant pour accéder aux leds LED1 à LED5 de la carte rpi-java.

Broche	Numéro de broche connecteur RPi	Fonction	Nom logique	Méthodes Java d'initialisation
OUT	26	LED1	GPIO7	DeviceManager.open(2)
OUT	12	LED2	GPIO18	DeviceManager.open(4)
OUT	16	LED3	GPIO23	DeviceManager.open(6)
OUT	18	LED4	GPIO24	DeviceManager.open(7)
OUT	22	LED5	GPIO25	DeviceManager.open(8)



LEDS

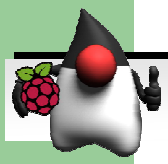
- On utilisera les méthodes suivantes pour allumer/éteindre une led de la carte :
  - Allumer : `setValue(true).`
  - Eteindre : `setValue(false).`



LEDS

- Exemple LED1 :

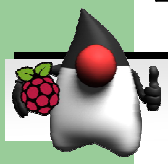
```
private final int LED1 = 2;

GPIOPin led1;

try {
    led1 = (GPIOPin) DeviceManager.open(LED1);

    while (true) {
        led1.setValue(true);
        Thread.sleep(100);
        led1.setValue(false);
    }
} catch (Exception ex) {
    ex.printStackTrace();
}
```



BOUTONS POUSSOIRS

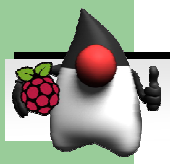
- On utilisera le tableau suivant pour accéder aux boutons poussoirs BP1 à BP4 de la carte rpi-java.

Broche	Numéro de broche connecteur RPi	Fonction	Nom logique	Méthodes Java d'initialisation
IN	7	BP1	GPIO4	DeviceManager.open(1) GPIOPinConfig(4)
IN	11	BP2	GPIO17	DeviceManager.open(3) GPIOPinConfig(17)
IN	15	BP3	GPIO22	DeviceManager.open(5) GPIOPinConfig(22)
IN	13	BP4	GPIO27	DeviceManager.open(9) GPIOPinConfig(27)



BOUTONS POUSSOIRS

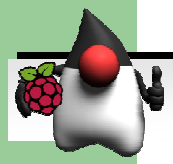
- On utilisera la méthode suivante connaître l'état d'un bouton poussoir de la carte :
  - `getValue()` renvoie *false* si appui et *true* si pas appui sur le bouton poussoir.



BOUTONS POUSSOIRS

- Exemple BP1 :

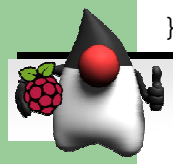
```
private final int BP1 = 1;
final boolean PRESSED = false;

boolean state1;
GPIOPin bp1;
try {
    bp1 = (GPIOPin)DeviceManager.open(BP1);

    while(true) {
        Thread.sleep(500);

        state1 = bp1.getValue();

        if(state1 == PRESSED)
                System.out.println("BP1 PRESSED");
    }
} catch (Exception ex) {
    ex.printStackTrace();
}
```



BOUTONS POUSSOIRS

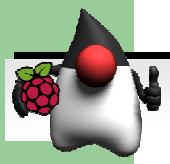
- Exemple BP1 :

```
private final int BP1 = 1;
final boolean PRESSED = false;

boolean state1;
GPIOPin bp1;
try {
     bp1 = (GPIOPin) DeviceManager.open(new GPIOPinConfig(
                        0,
                        BP1,
                        GPIOPinConfig.DIR_INPUT_ONLY,
                        GPIOPinConfig.DEFAULT,
                        GPIOPinConfig.TRIGGER_FALLING_EDGE,
                  false));
    while(true) {
         Thread.sleep(500);
         state1 = bp1.getValue();
         if(state1 == PRESSED)
                  System.out.println("BP1 PRESSED");
    }
} catch (Exception ex) {
    ex.printStackTrace();
}
```



BOUTONS POUSSOIRS

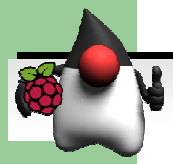
- Les exemples précédents permettent de connaître l'état d'un bouton poussoir de façon synchrone par scrutation.
- On peut connaître l'état d'un bouton poussoir de façon asynchrone en mettant en place un *listener*.
- On installera le *listener* avec la méthode `setInputListener()`. La méthode `valueChanged()` du *listener* sera exécutée s'il y a un changement d'état du bouton poussoir.
- La *MIDlet* doit implémenter l'interface `PinListener`.



BOUTONS POUSSOIRS

- Exemple *listener* BP1 :

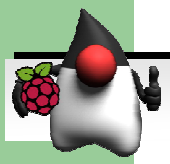
```
public class TstBpListen extends MIDlet implements PinListener {
    private final int BP1 = 1;
    final boolean PRESSED = false;
    GPIOPin bp1;

    public void startApp()
        try {
                bp1 = (GPIOPin) DeviceManager.open(new GPIOPinConfig(
                        0,
                        BP1,
                        GPIOPinConfig.DIR_INPUT_ONLY,
                        GPIOPinConfig.DEFAULT,
                        GPIOPinConfig.TRIGGER_FALLING_EDGE,
                        false));

                bp1.setInputListener(this);
        } catch (Exception ex) {
                ex.printStackTrace();
        }
    }
```



BOUTONS POUSSOIRS

- Exemple *listener* BP1 :

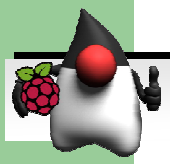
```
public void valueChanged(PinEvent event) {
    try {
        if(bp1.getValue() == PRESSED)
            System.out.println("BP1 PRESSED");
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
}
```



BUS I2C

- Deux périphériques I2C sont connectés sur le bus I2C de la carte rpi-java :
  - Afficheur LCD 2x16 caractères en mode 4 bits par l'intermédiaire d'un circuit adaptateur I2C/GPIO (I2C_ID=0x20).
  - Capteur de température DS1624 (I2C_ID=0x48).

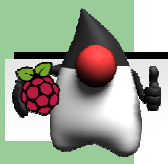
- On accède à ces périphériques par l'intermédiaire de 2 classes Java :
  - Classe *LCD2x16* pour l'afficheur LCD.
  - Classe *DS1624* pour le capteur de température.



BUS I2C

- Le bus I2C étant partagé, on accède successivement à ces 2 périphériques…

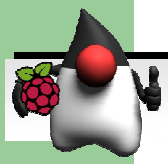
- Les classes Java *LCD2x16* et *DS1624* utilisent les méthodes suivantes :
  - `I2CDeviceConfig()`.
  - `DeviceManager.open()`.
  - `write()`.
  - `read()`.



BUS I2C

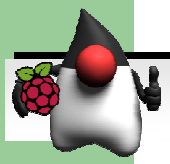
- Exemple : classe *LCD2x16* :

```
class LCD2x16 {
    final int LCD_I2C_ID = 0x20;
    final int BUS_ID = 1;

    I2CDeviceConfig config;
    I2CDevice slave;

    public LCD2x16() throws InterruptedException, IOException {

        config = new I2CDeviceConfig(BUS_ID, LCD_I2C_ID, 7, 100000);
        slave = DeviceManager.open(config);
    }
   public void init() throws InterruptedException, IOException {

        slave.write(0x03);
        lcd_strobe();
        Thread.sleep(20);
. . .
```



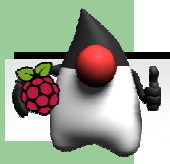
AFFICHEUR LCD

- L'afficheur LCD est accessible par la classe Java *LCD2x16* et les méthodes suivantes :
  - `LCD2x16()` : constructeur.
  - `init()` : initialisation.
  - `lcd_clear()` : effacement de l'afficheur.
  - `lcd_home()` : positionnement en ligne 0, colonne 0.
  - `lcd_pos(int li, int co)` : positionnement en ligne `li` et colonne `co`.
  - `lcd_write_char(byte charvalue)` : écriture d'un caractère à la position courante.



AFFICHEUR LCD

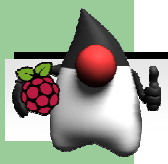
- L'afficheur LCD est accessible par la classe Java *LCD2x16* et les méthodes suivantes :
  - `lcd_puts(String string)` : écriture d'une chaîne de caractères à la position courante.
  - `lcd_line_puts(int line, String string)` : écriture d'une chaîne de caractères à la ligne `line` (0 ou 1).
  - `lcd_strobe()` : fonction utilitaire.
  - `lcd_write_inst(byte value)` : fonction utilitaire.



AFFICHEUR LCD

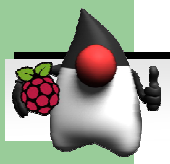
- Exemple d'affichage :

```
LCD2x16 display;
Date now;

try {
     display = new LCD2x16();

     display.init();
     display.lcd_clear();
     display.lcd_home();

     while (true) {
        now = new Date();
        display.lcd_line_puts(0, ""+ getDateString(now));
        now = null;
        Thread.sleep(100);
     }
} catch (InterruptedException ex) {
     ex.printStackTrace();
} catch (IOException ex) {
     ex.printStackTrace();
}
```



CAPTEUR DE TEMPERATURE

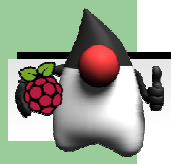
- Le capteur de température DS1624 est accessible par la classe Java *DS1624* et les méthodes suivantes :
  - `DS1624()` : constructeur.
  - `start()` : lancement de l'acquisition de la température.
  - `stop ()` : fin de l'acquisition de la température.
  - `double read_temp()` : lecture de la température.



CAPTEUR DE TEMPERATURE

- Exemple de lecture de la température :

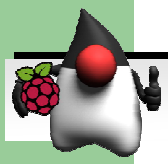
```
DS1624 ds;
double temp;

try {
     ds = new DS1624();

     while (true) {

        ds.start();
        ds.stop();

        temp = ds.read_temp();
        System.out.format("%02.1f oC\n", temp);

        Thread.sleep(1000);
     }
} catch (InterruptedException ex) {
     ex.printStackTrace();
} catch (IOException ex) {
     ex.printStackTrace();
}
```



THREAD JAVA ME

- Une machine virtuelle Java est un OS multitâche. Il est possible de créer des *threads* Java (tâche) qui interagissent ou qui sont en compétition.

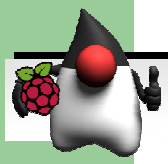
- On peut créer un *thread* pour une classe donnée :
  - En étendant par héritage de la classe *java.lang.Thread*. La méthode `run()` de la classe *Thread* permet d'exécuter le *thread*.
  - En implémentant l'interface *java.lang.Runnable* et en définissant la méthode abstraite `run()` de l'interface.



THREAD JAVA ME

- Exemple par extension :

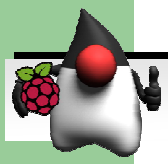
```
public class DoAnotherThing extends Thread {
    public void run(){
        // here is where you do something
    }
}

...

DoAnotherThing doIt = new DoAnotherThing();
doIt.start();

...
```



THREAD JAVA ME

- Exemple par implémentation de l'interface *Runnable* :

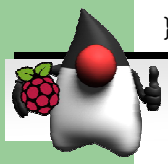
```
public class DoSomething implements Runnable {
    public void run(){
        // here is where you do something
    }
}

...

DoSomething doIt = new DoSomething();
Thread myThread = new Thread( doIt );

myThread.start();

...
```



THREAD JAVA ME

- Exemple par implémentation de l'interface *Runnable* :

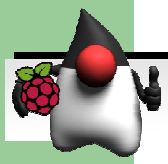
```
task1 t1;
Thread th1;
try {
    t1 = new task1();
    th1 = new Thread(t1);
    th1.start();
    Thread.sleep(10000);
    t1.quit();
} catch (Exception ex) {
    ex.printStackTrace();
}

class task1 implements Runnable {
    private boolean quit = false;

    public void run() {
        ...
    }
    public void quit() {
        quit = true;
    }
}
```



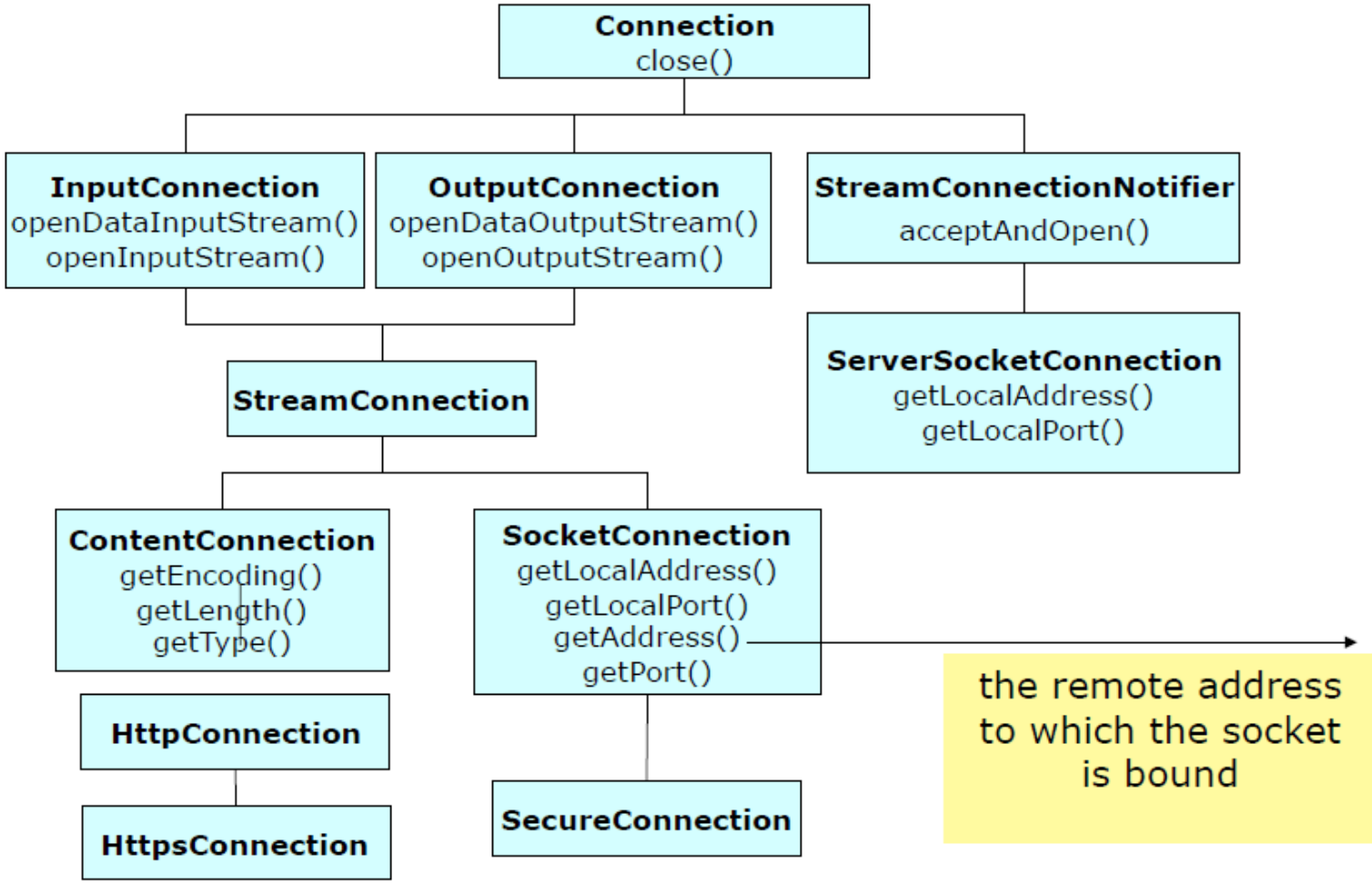
SOCKET JAVA ME

- Le profil MIDP exige le support de HTTP et HTTPS.
- D'autres types de connexion sont aussi supportés :

Type	Interface	Exemple
Socket	SocketConnection	socket://localhost:80
Server socket	ServerSocketConnection	socket://:8080
TLS or SSL socket	SecureConnection	ssl://localhost:80
Serial port	CommConnection	comm:com0;baudrate=9600



SOCKET JAVA ME

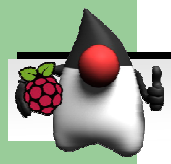


SOCKET JAVA ME

- `Connector` est une classe qui permet d'accéder à une connexion réseau ou à un canal de communication.

- Sa méthode `open()` permet d'ouvrir une URL `name` :
  - `public static Connection open(String name) throws IOException.`

- Autres méthodes de `Connector` :
  - `public static DataInputStream openDataInputStream(String name) throws IOException.`
  - `public static InputStream openInputStream(String name) throws IOException.`
  - `public static DataOutputStream openDataOutputStream(String name) throws IOException.`
  - `public static OutputStream openOutputStream(String name) throws IOException.`

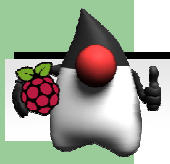


SOCKET JAVA ME

- En général, on essaie d'avoir une seule connexion qui permet de lire et écrire les données.

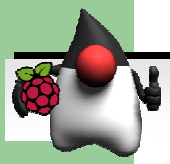
- Ceci est obtenu par une `StreamConnection` qui est une interface qui hérite de `InputConnection` et `OutputConnection`.

- Les méthodes de ces interfaces vont retourner des `DataInputStream` et `DataOuputStream` et donc on peut ainsi utiliser les méthodes pour `DataInputStream` et pour `DataOuputStream`.



SOCKET JAVA ME

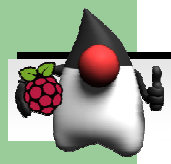
- Pour implémenter un serveur *socket*, il faut :
  - Créer une `ServerSocketConnection` par la méthode `Connector.open()`.
  - Attendre une connexion entrante qui crée une nouvelle *socket* `SocketConnection` par la méthode `acceptAndOpen()`.
  - Récupérer les canaux d'écriture et de lecture sur cette connexion.
  - Récupérer la requête et renvoyer la réponse.
  - Fermer les canaux d'écriture et de lecture par la méthode `close()`.
  - Fermer la connexion établie par la méthode `close()`.



SOCKET JAVA ME

- Exemple d'un miniserveur Web :

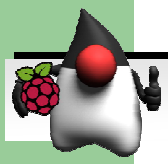
```
public class TstWww extends MIDlet {

    public void startApp() {
        ServerSocketConnection ssc;
        SocketConnection sc;
        InputStream is;
        OutputStream os;

        StringBuffer results = new StringBuffer();

        try {
            ssc = (ServerSocketConnection) Connector.open("socket://:8080");

            while (true) {
                sc = (SocketConnection) ssc.acceptAndOpen();
```



SOCKET JAVA ME

- Exemple d'un miniserveur Web :

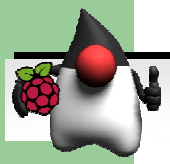
```
is = sc.openInputStream();
os = sc.openOutputStream();

Reader in = new InputStreamReader(is);

String line;
while ((line = readLine(in)) != null)
    ;

PrintStream out = new PrintStream(os);

out.print("HTTP/1.1 200 OK\n");
out.print("Content-Type: text/html\n");
out.print("\n");

out.print("<H1><CENTER> Welcome to the rpi-java Web Server
</CENTER></H2>");
out.print("Hello world!");
```



SOCKET JAVA ME

- Exemple d'un miniserveur Web :

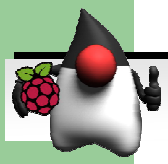
```
                out.close();
                in.close();
                sc.close();
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    public void destroyApp(boolean unconditional) {
    }
```



SOCKET JAVA ME

- Exemple d'un miniserveur Web :

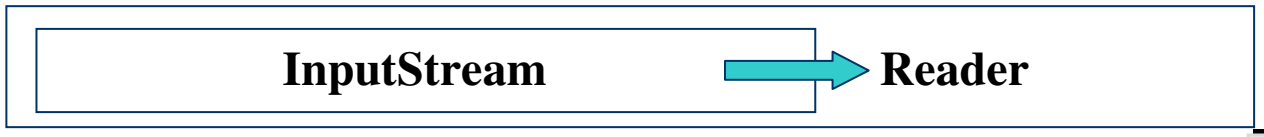
```
public String readLine(Reader in) throws IOException {
    StringBuffer line = new StringBuffer();
    int i;

    while ((i = in.read()) != -1) {
        char c = (char)i;

        if (c == '\n')
            break;
        if (c == '\r')
            ;
        else
            line.append(c);
    }
    if (line.length() == 0)
        return null;

    return line.toString();
}
}
```

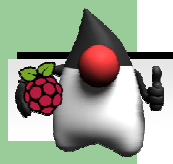


SOCKET JAVA ME

- Un `InputStream` est un flux d'octets entrant non structurés. La méthode `read()` permet de lire ce flux d'octets.
- Un `Reader` est un pont depuis ce flux d'octets vers un flux de caractères. La méthode `read()` permet de lire le flux de caractères.

```
InputStream is;
is = sc.openInputStream();

Reader in;
in = new InputStreamReader(is);

int i;
i = in.read();
```

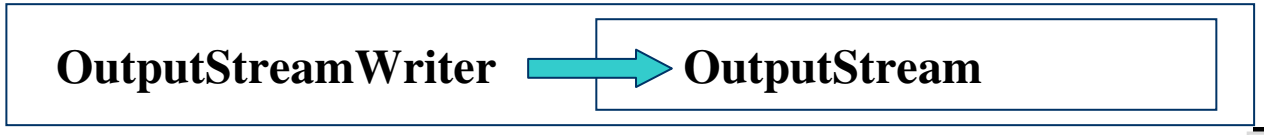


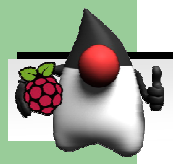


SOCKET JAVA ME

- Un `OutputStream` est un flux d'octets sortant non structurés. La méthode `write()` permet d'écrire ce flux d'octet.
- Un `OuputStreamWriter` est un pont depuis un flux de caractères vers ce flux d'octets non structurés. La méthode `write()` permet d'écrire ce flux de caractères.

```
OutputStream os;
os = sc.openOutputStream();

OutputStreamWriter out;
out = new OutputStreamWriter(os);

os.write("12345678");
```

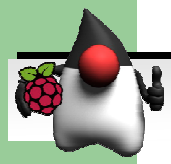




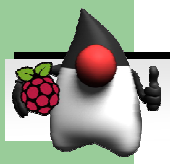
SOCKET JAVA ME

- Un `OutputStream` est un flux d'octets sortant non structurés. La méthode `write()` permet d'écrire ce flux d'octet.
- Un `PrintStream` rajoute la fonctionnalité d'écrire différents types de données. La méthode `print()` permet d'écrire ces données (`boolean, char, char[], double, float, int, long, Object, String`).

```
OutputStream os
os = sc.openOutputStream();

PrintStream out;
out = new PrintStream(os);

out.print("HTTP/1.1 200 OK\n");
```

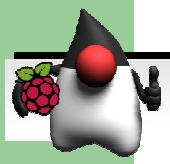


PARTIE 6
CONCLUSION

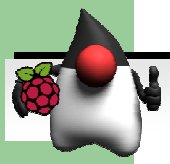


Conclusion

- Java ME permet de mettre en œuvre Java embarqué dans les systèmes embarqués et les objets communicants.
- Java ME introduit les notions de profil et de configuration.
- Avec Java ME, on développe une application Java embarqué appelée *MIDlet*.
- La mise en œuvre de Java ME sur une carte cible à base de Raspberry Pi (carte rpi-java) a été décrite pas à pas. Cette mise en œuvre sera vue en TP pour construire un objet connecté communicant sous Java ME…

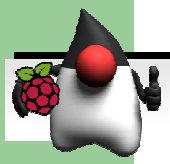


BIBLIOGRAPHIE



Bibliographie

- Java Mobile. H. Sansen. Ressource Internet
- Java Mobile. J2ME. Java Micro-Edition. Inconnu. Ressource Internet
- Java 2 Micro Edition Sockets and SMS. F. Ricci. Ressource Internet
- Java ME : une présentation. Jean Marc Farinone. Ressource Internet
- Java in the Internet of Things: small, smart, connected. T. Barr, A. Caidedo. Oracle
- Java ME Embedded 8. K. Shen. Oracle
- Raspberry Pi with Java: Programming the Internet of Things. S. Chin et J. Weaver. Oracle



Bibliographie

- Site de NetBeans : https://netbeans.org
- Site de Java ME : http://www.oracle.com/technetwork/java/embedded/javame/index.html
- Site Wikipédia : https://fr.wikipedia.org/wiki/Raspberry_Pi