

## Cours 2

### Premiers pas avec OSGi

La plate-forme dynamique orientée composants

Bachir Djafri

## Outils nécessaires

---

- ▶ **Framework Apache Felix**

- ▶ Implémentation de la version R4 du modèle OSGi

- ▶ <https://felix.apache.org/>

- ▶ **Eclipse**

- ▶ <http://www.eclipse.org>

- ▶ **Références**

- ▶ <https://felix.apache.org/documentation/tutorials-examples-and-presentations/apache-felix-osgi-tutorial.html>

- ▶ <https://felix.apache.org/documentation/subprojects/apache-felix-framework/apache-felix-framework-usage-documentation.html>

- ▶ <https://www.manning.com/books/osgi-in-action>

# OSGi

---

## Contexte initial

- ▶ OSGi = *Open Service Gateway Initiative* (nom obsolète)
- ▶ Augmentation du logiciel embarqué pour les véhicules
  - ▶ Hausse de 2.6 à 9 milliards d'euros en 2010
  - ▶ 90% des innovations reposent sur l'électronique
  - ▶ 80% des fonctionnalités de l'automobile sont logicielles
  - ▶ La part du logiciel dans l'automobile passe de 22% (2003) à 35% (2010)
- ▶ De plus en plus de périphériques sont intelligents et adaptables
  - ▶ Automobile
  - ▶ Téléphonie mobile
  - ▶ Home cinéma
  - ▶ Domotique
- ▶ Les réseaux de données sont omniprésents
- ▶ Tous ces composants nécessitent des couches logicielles de plus en plus complexes

→ Il faut en faciliter la gestion

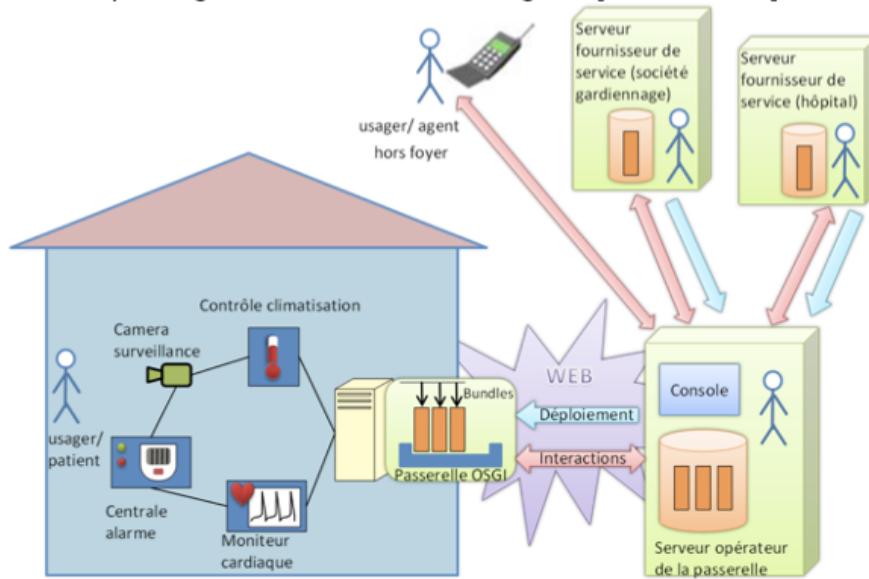
# OSGi

---

- ▶ **La finalité initiale d'OSGi était donc de**
  - ▶ Rendre les **composants logiciels** faciles à gérer (installation, suppression, mises à jour) et cela à distance
  - ▶ Permettre la portabilité des services à valeur ajoutée à travers les marchés et les équipements
  - ▶ Permettre la mise en place de nouveaux business-models
  
- ▶ **Avec pour cibles privilégiées**
  - ▶ Les équipements mobiles (premier équipement OSGi en 2005 Motorola)
  - ▶ L'équipement automobile (déjà BMW)
  - ▶ Les réseaux résidentiels
    - ▶ Via ADSL et autres

# OSGi

## Exemple de gestion d'une maison intelligente [cours Donsez]



# OSGi

---

## Etat des lieux

- ▶ Supporté par l'**OSGi Alliance**, organisation internationale à but non lucratif formée pour développer et promouvoir des spécifications ouvertes pour la livraison par le réseau de services administrés vers des périphériques dans la maison, la voiture ou d'autres environnements
  - ▶ Corporation indépendante
  - ▶ Soutenus par les acteurs majeurs des IT, home/building automation, telematics (car automation), ...
  - ▶ de la téléphonie mobiles (Nokia et Motorola au début)
  
- ▶ et Eclipse pour les plugins de son IDE !
- ▶ et maintenant Apache pour ses serveurs

voir [www.osgi.org](http://www.osgi.org)

## OSGi – domaine d'application

---

### ▶ Initialement, Systèmes embarqués

- ▶ Véhicule de transport (automotive)
- ▶ Passerelle résidentielle / domotique / immotique
- ▶ Contrôle industriel
- ▶ Téléphonie mobile

### ▶ Cependant

- ▶ Tout concepteur d'application est gagnant à distribuer son application sous forme de **plugins** conditionnés dans des **bundles** OSGi
- ▶ Cela évite l'enfer du CLASSPATH
  - ▶ CLASSPATH, lib/ext du JRE ou JavaEE, ...
  - ▶ Voir les chargeurs de classes en Java

### ▶ Maintenant

- ▶ Eclipse RCP, JavaEE, Harmony JRE pieces, .

# OSGi

---

## ► Spécification OSGi

- définit un canevas de déploiement et d'exécution de services Java
- multi-fournisseur, télé-administré
- Cible initiale : set top box, modem cable, ou une passerelle résidentielle dédiée.
- Pour le moment 7 releases, la première en 2000 la dernière en avril 2018 (release 6 pour ce cours)

## ► Caractéristiques principales

- Modularité des applications
  - Chargement/Déchargement de code dynamique
  - Langage d'implémentation Java
  - Déploiement dynamique d'applications sans interruption de la plateforme
  - Installation, Lancement (activation), Mise à jour, Arrêt, Retrait (à distance)
  - « No reboot »
  - Résolution des dépendances versionnées de code
- Architecture orientée composants/services
  - Couplage faible, late-binding (très important)
  - Reconfiguration dynamique des applications (plugins, services techniques)



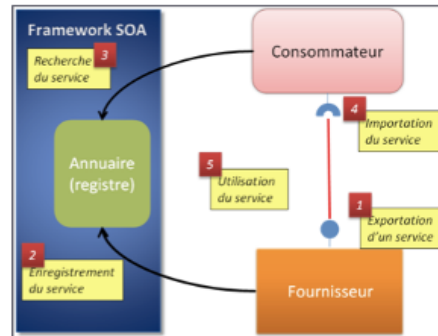
# Points clés d'OSGi

Les fonctionnalités principales d'OSGi sont donc

- ▶ La mise à jour dynamique de logiciels
- ▶ Contrôle à distance
- ▶ Maintenance à distance
- ▶ Diagnostic distant
- ▶ Échange de données

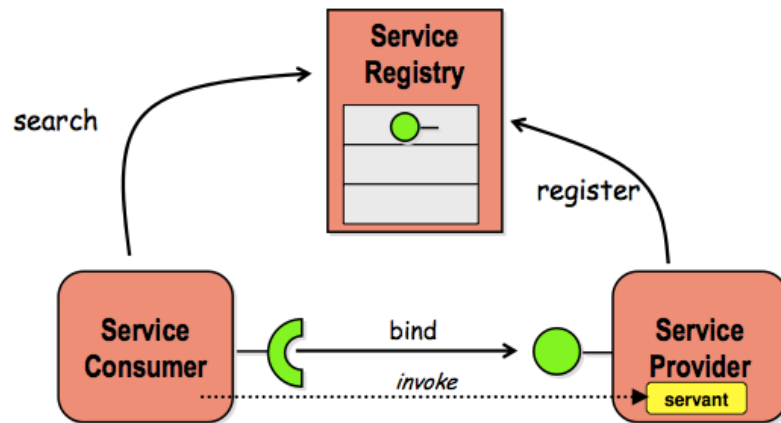
## Techniquement

- ▶ OSGi repose sur
  - ▶ Le langage Java (implémentation)
  - ▶ La programmation orientée composants/services
  - ▶ 3 artefacts: le bundle, le package et le service
- ▶ Mais OSGi n'est pas
  - ▶ Un système distribué (pas d'invocation distante)
  - ▶ Seul le changement de bundle peut se faire à distance



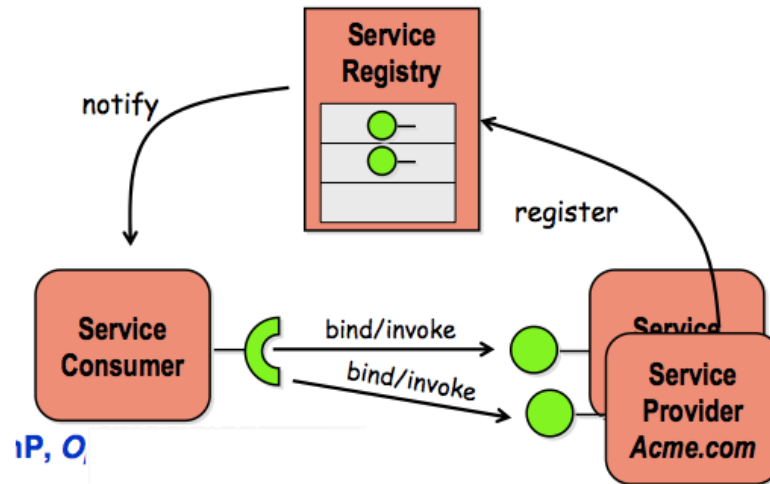
## Approche orientée services dynamiques

- ▶ OSGi suit l'approche orientée service (SOA) ... **Dynamique**
- ▶ Un service (contrat) est invariant



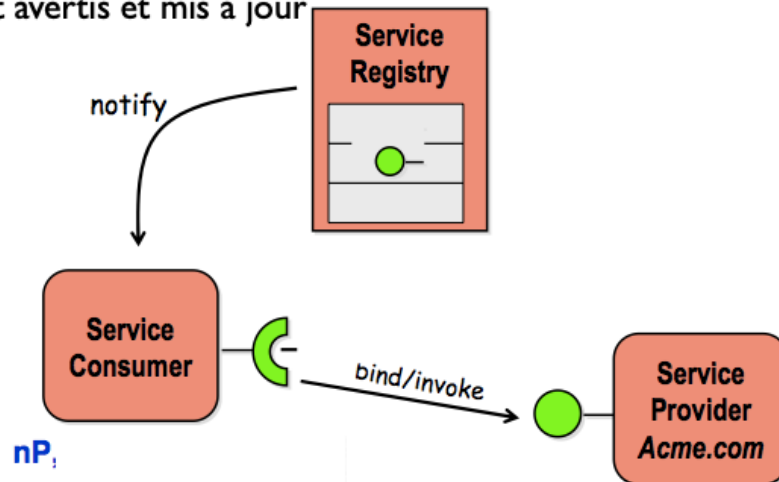
## Approche orientée service dynamique

- L'arrivée dynamique de nouveaux services est possible
- Un client d'un service existant peut alors dynamiquement changer d'implémentation de ce service



## Approche orientée service dynamique

- ▶ De même, une implémentation d'un service peut disparaître dynamiquement ou être modifiée dynamiquement
- ▶ Ceci induit que les composants utilisateurs de ce service (clients) soient avertis et mis à jour



## Structure d'une application basée sur OSGi

---

- ▶ **Rappel – une application Java non modulaire**
  - ▶ Un ensemble de fichiers .jar
  - ▶ Fichiers placés statiquement dans le CLASSPATH ou \$JRE\_HOME/lib/ext



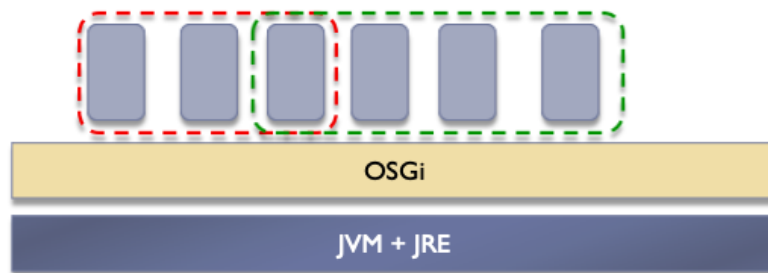
## Structure d'une application basée sur OSGi

### ► Bundle

- Le **composant** OSGi (plugin)
- Unité de livraison et de déploiement sous forme d'archives .jar
- Unité fonctionnelle (offre/utilise des services)

### ► Application

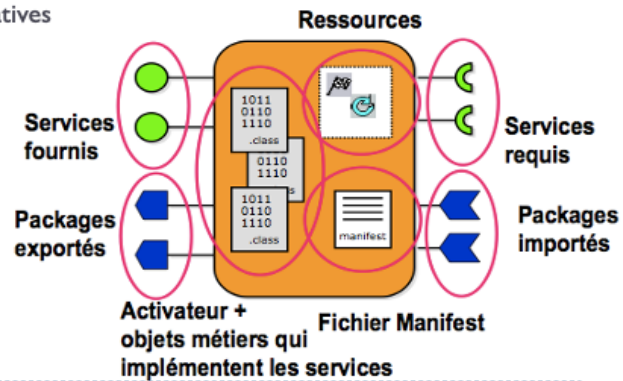
- « Ensemble » de bundles (composants, plugins)
  - Livrés dynamiquement et
  - Éventuellement partagés avec d'autres applications



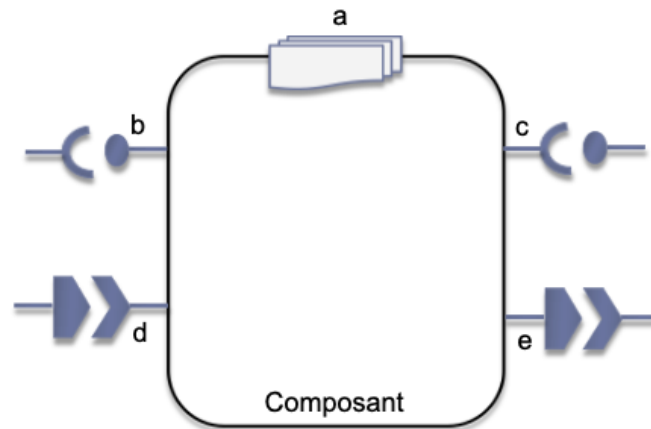
# La notion de bundle

- ▶ Le bundle (baluchon en français) est l'unité de déploiement versionnée dans OSGi, autrement dit le composant.
- ▶ Il est conditionné sous forme d'un fichier JAR contenant
  - ▶ **Un fichier Manifest décrivant le bundle** (point d'entrée pour l'environnement)
  - ▶ Le code binaire des classes d'implémentation
  - ▶ Des ressources utiles pour le bundle (fichiers de configuration, images, etc.)
  - ▶ Des bibliothèques de code natives

+ un fichier Component.xml  
À partir de la release 4  
(à voir aux cours suivants)



## La notion de bundle (Composant OSGI)



Un composant OSGI (un bundle) peut offrir des services via ses interfaces fournies (synchrones) comme l'interface fournie b.

Il peut aussi avoir besoin de services. Ces besoins sont exprimés via des interfaces requises (synchrones) comme l'interface c.

Un composant OSGI peut aussi avoir des propriétés (Une liste de couples clé-valeur). Ces propriétés sont exprimées par une liste de couples clé-valeur associée au composant (a=propriétés configurables).

Un composant OSGI peut avoir besoin (requière) d'autres ressources comme des packages ou d'autres composants (bundles). Ce besoin de ressources est exprimé par des interfaces de ressources requises comme l'interface d (import de package ou besoin de bundle).

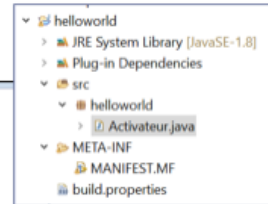
Un composant OSGI peut aussi fournir des ressources (offre de ressources) pour d'autres composants (bundles). Ces ressources sont souvent des packages (export de packages). Cette offre est exprimée par une interface de ressources offerte comme e.



## Illustration – Un premier exemple

- ▶ Contient seulement une classe d'activation

```
1 package helloworld;
2
3 import org.osgi.framework.BundleActivator;
4 import org.osgi.framework.BundleContext;
5
6 public class Activateur implements BundleActivator {
7
8     public void start(BundleContext bundleContext) throws Exception {
9         System.out.println("Hello World !");
10    }
11
12    public void stop(BundleContext bundleContext) throws Exception {
13        System.out.println("Hello World stop...");
14    }
15 }
16
```



MANIFEST.MF

```
1 Manifest-Version: 1.0
2 Bundle-ManifestVersion: 2
3 Bundle-Name: Helloworld
4 Bundle-SymbolicName: helloworld
5 Bundle-Version: 1.0.0
6 Bundle-Activator: helloworld.Activateur
7 Bundle-Vendor: Bachir Djafri
8 Bundle-RequiredExecutionEnvironment: JavaSE-1.8
9 Import-Package: org.osgi.framework;version="1.3.0"
10 Bundle-ActivationPolicy: lazy
11
```

## Illustration – Un premier exemple

---

### ► DEMO

```
java -jar bin/felix.jar
```

### ► Commandes Felix

- `lb` liste des bundles disponibles + activité
- `install file:plugins/helloworld_1.0.0.jar`  
installe le bundle passé en paramètre
- `start id` lance le bundle numéro ID
- `stop id` stoppe le bundle numéro ID
- `update id` met à jour le bundle numéro ID
- `uninstall id` supprime le bundle numéro ID
- `CTRL + D` quitte le framework (stop 0)

# Déploiement d'un bundle

► Le fichier MANIFEST.MF est le descripteur de déploiement associé à un bundle

► Voir <https://www.osgi.org/release-6-1/> pour plus d'informations

► Quelques exemples

1. Bundle-name	nom du bundle
2. Bundle-Description	description textuelle du bundle
3. Import-Package	packages importés (au chargement)
4. Dynamic-Import-Package	packages importés (au fur et à mesure des besoins)
5. Export-Package	packages exportés
6. Bundle-Activator	nom de la classe qui active le bundle (point d'entrée et de sortie)
7. Bundle-ClassPath	à l'intérieur du bundle
8. Bundle-UpdateLocation	url pour mise à jour du bundle
9. Bundle-Version	la version du bundle
10. Bundle-Vendor	le vendeur du bundle
11. Bundle-ContactAddress	adresse mail de contact
12. Bundle-Copyright	chaîne de caractères

## L'implémentation de la classe d'activation

---

- ▶ **Classe publique**
  - ▶ Implémente les 2 méthodes `start()` et `stop()` de l'interface **BundleActivator** qui reçoivent une référence du contexte
- ▶ **La méthode `start(BundleContext context)`**
  - ▶ recherche et obtient des services requis auprès du contexte et/ou positionne des écouteurs (listeners/trackers) sur des événements
  - ▶ enregistre les services fournis auprès du contexte
- ▶ **La méthode `stop(BundleContext context)`**
  - ▶ désenregistre les services fournis
  - ▶ relâche les services requis
  - ▶ *Cependant le framework « devrait » faire ces opérations si `stop()` est oublié*

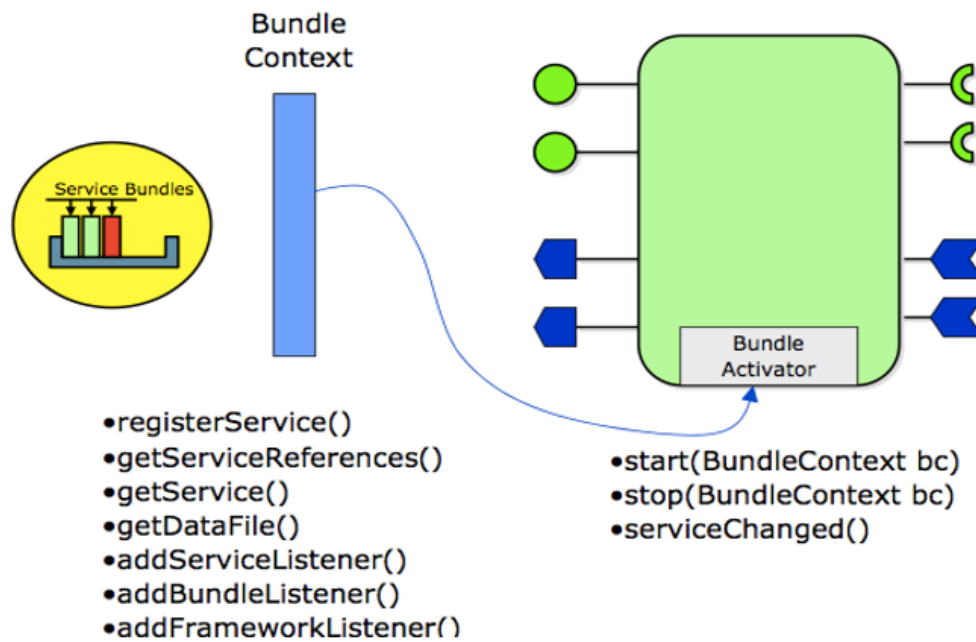
**C'est via l'enregistrement et la résiliation des services auprès du contexte que l'annuaire des services est maintenu à jour !!**

## Le BundleContext

---

- ▶ Référence vers le framework (interface Java)
- ▶ Passé lors des invocations de start() et stop() de l'activateur (interface BundleActivator)
- ▶ Permet
  - ▶ L'enregistrement de services
  - ▶ Le courtage de services (selon des propriétés)
  - ▶ L'obtention et la libération de services
  - ▶ La souscription aux événements du Framework (ou autres)
  - ▶ L'accès aux ressources du bundle
  - ▶ L'accès aux propriétés du framework
  - ▶ L'installation de nouveaux bundles
  - ▶ L'accès à la liste des bundles

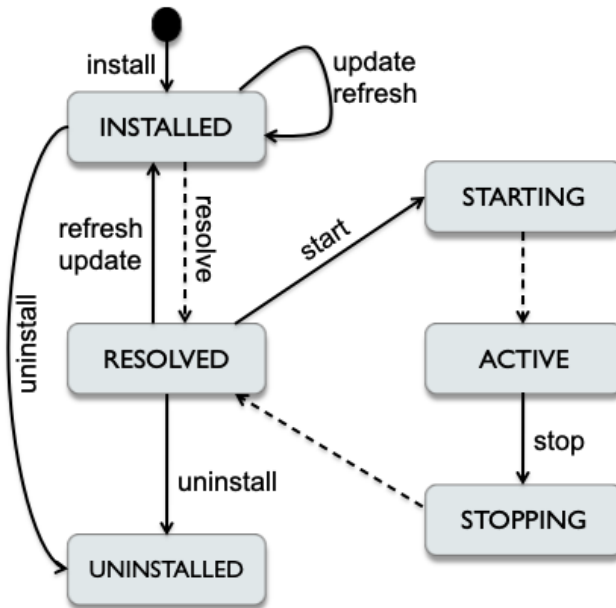
# BundleContext et Activator



# Cycle de vie d'un bundle

- Sept événements possibles modifient l'état d'un bundle

- 1 – **install** charge le bundle dans le système de fichiers
- 2 – **resolve** charge le bundle dans la machine virtuelle
- 3 – **start** invoque la méthode `start()` de l'activateur
- 4 – **stop** invoque la méthode `stop()` de l'activateur
- 5 – **uninstall** Supprime le bundle du système de fichiers
- 6 – **update** réinstalle le bundle
- 7 – **refresh** reconstruit les liaisons du bundle

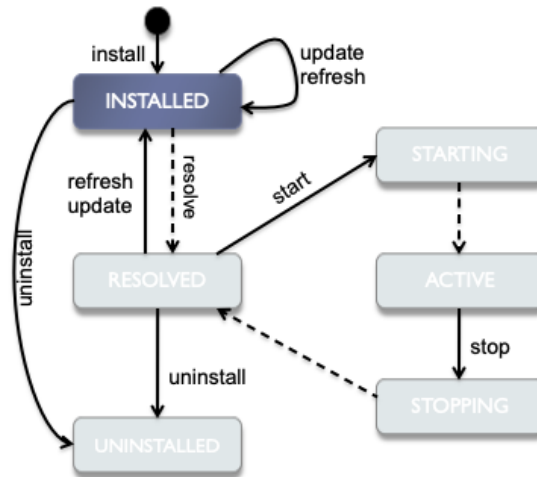


# Cycle de vie d'un bundle

## ► L'installation

- Téléchargement et stockage du bundle (un fichier JAR) dans le système de fichier de la plateforme, le *bundle cache*

- L'activation du bundle devient alors possible

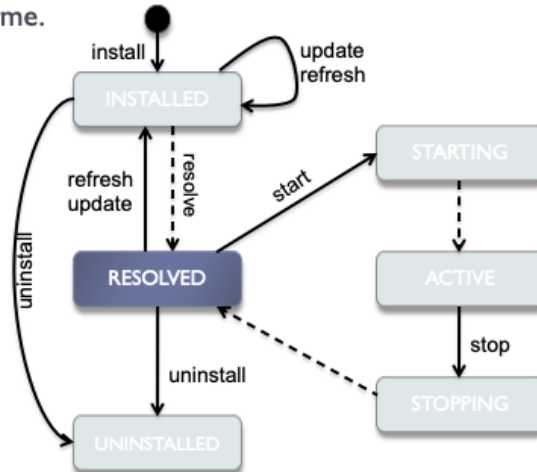




# Cycle de vie d'un bundle

## ► La résolution de dépendances

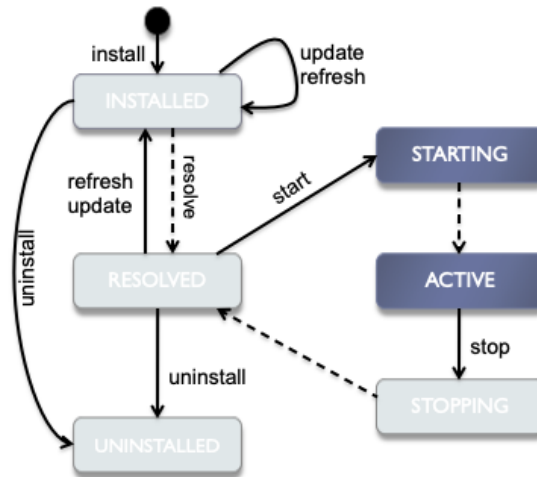
- Le bundle passe à l'état RESOLVED lorsque tous les packages qu'il importe (voir Import-Package) sont tous exportés par des bundles installés et résolus sur la plateforme.
- Une fois dans cet état, il peut exporter les packages listés dans Export-Package
- Quand le même package est exporté par plusieurs bundles, un seul de ces bundles l'exporte



# Cycle de vie d'un bundle

## ► L'activation

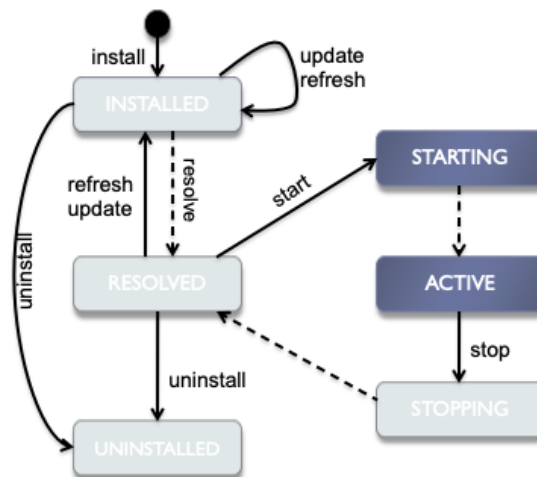
- La plateforme instancie un seul objet de la classe décrite dans le manifeste comme étant le Bundle-Activator
- Cette classe doit implémenter l'interface BundleActivator qui régit le cycle de vie d'un bundle
- La méthode start() est appelée avec en paramètre une instance de BundleContext qui décrit le contexte du bundle et de la plateforme
- Cette méthode peut rechercher des services, en enregistrer et activer différents threads.



# Cycle de vie d'un bundle

## ► L'activation

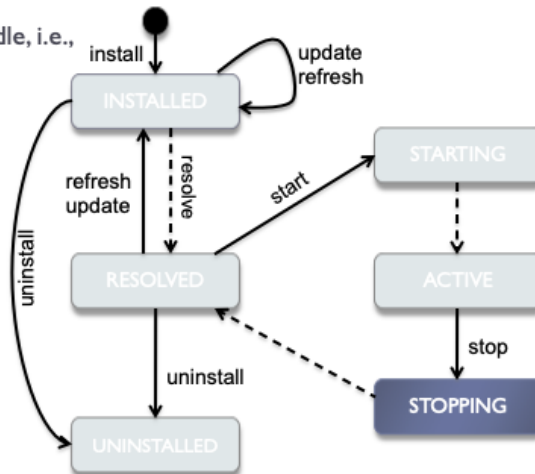
- Si une exception est levée, le bundle revient à l'état RESOLVED
- Sinon il passe à ACTIVE
- Durant l'exécution de la méthode start() il est dans l'état STARTING



# Cycle de vie d'un bundle

## ► L'arrêt

- L'arrêt d'un bundle déclenche la méthode `stop()` de l'interface du `BundleActivator`.
- Cette méthode doit nettoyer le bundle, i.e., dé-enregistrer les services fournis, relâcher les services utilisés et arrêter les threads démarrés.
- Le bundle repasse à l'état **RESOLVED**
- L'objet d'activation est récupérable par le ramasse-miettes.



# Cycle de vie d'un bundle

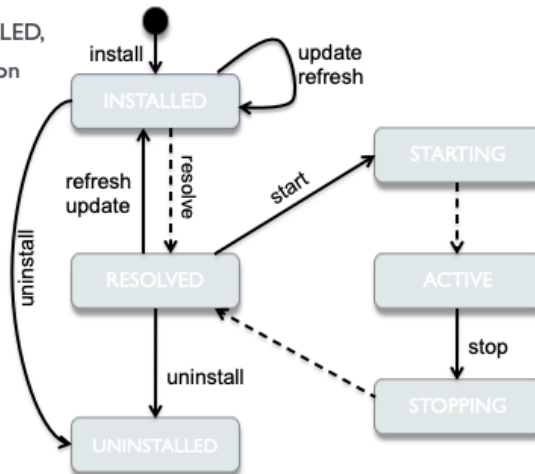
## ► La mise à jour

- Elle provoque l'enchaînement

*Arrêt → ré installation → résolution → ré activation*

- Le bundle se retrouve dans l'état INSTALLED, si de nouveaux packages sont requis et non disponibles sur la plateforme

- Le bundle se retrouve dans l'état RESOLVED si la réactivation du bundle a échouée (exception propagée par la méthode start() suite à l'absence d'un service obligatoire, non optionnel)





## Exemple 2

---

- ▶ Mise en œuvre de l'approche orientée composants avec OSGi
- ▶ Nous sommes dans une approche orientée composants dynamique avec quatre types d'artefacts
  - ▶ *Le fournisseur*
    - ▶ Il offre un service d'un certain type (type référence)
    - ▶ Il possède des propriétés : des couples clés/valeurs
    - ▶ Les types des services sont définis (implémentés) par des interfaces java
  - ▶ *Le client*
    - ▶ Il recherche un service d'un certain type
    - ▶ Il filtre sur les propriétés s'il y en a
    - ▶ Il sélectionne un service qui correspond à ses besoins (type référence)
    - ▶ Appel (utilise) directement l'instance du service
  - ▶ *Le contrat*
    - ▶ L'interface du contrat (type de service) est une interface Java
    - ▶ C'est un fichier de classe déployé dans un bundle
  - ▶ *Annuaire OSGi*
    - ▶ Associe des contrats et des instances de services
    - ▶ Renvoie des références OSGi vers les services (ServiceReference)

## Exemple 2

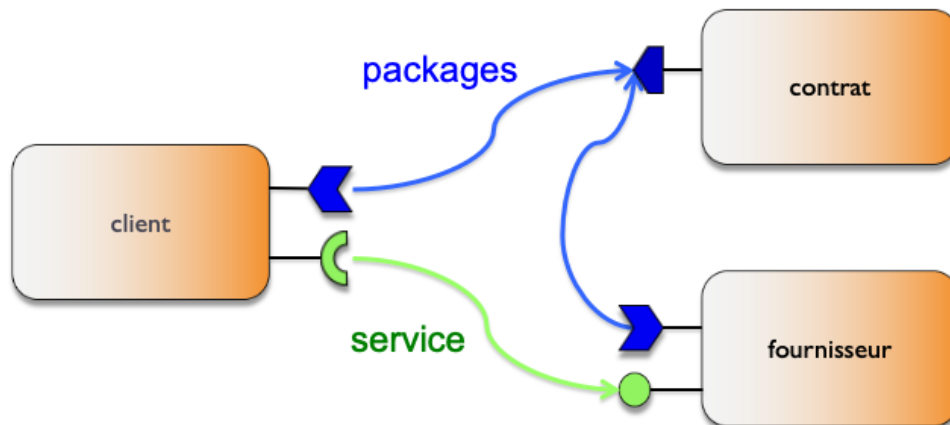
---

- ▶ Mise en œuvre de l'approche orientée composant avec OSGi
  - ▶ Des bundles de contrats
    - ▶ Manifest pas d'activateur, exporte l'interface du service (Type)
    - ▶ Code une interface de service + classes d'exceptions
  - ▶ Des bundles fournisseurs
    - ▶ Manifest activateur + import de la plateforme + import de l'interface du service (contrat avec les types de services)
    - ▶ Activateur alloue et enregistre une instance du service
    - ▶ Code l'implémentation du service
  - ▶ Des bundles clients
    - ▶ Manifest activateur + import de la plateforme + import de l'interface du service (contrat avec les types de services)
    - ▶ Activateur cherche, sélectionne et utilise un service d'un fournisseur
    - ▶ Code utiliser l'implémentation du service « sélectionné »



## Exemple 2

- Mise en œuvre d'un premier exemple de composition
  - Un bundle contrat définit une interface (type de service) pour dire bonjour et au revoir
  - Un bundle fournisseur implémente ce service (offre)
  - Un bundle client l'utilise (besoin)



## Exemple 2

### ► Bundle contrat

- Code – Juste une interface décrivant les services à fournir

```
1 package exemple.contrat;  
2  
3 public interface HelloWorld {  
4     void hello();  
5     void goodbye();  
6 }  
7 |
```

### ► Manifest – Il n'y a pas de classe d'activation !

- Conséquence : le démarrage et l'arrêt du bundle n'ont aucun sens

```
1 Manifest-Version: 1.0  
2 Bundle-ManifestVersion: 2  
3 Bundle-Name: HelloWorldContrat  
4 Bundle-SymbolicName: ContratID  
5 Bundle-Version: 1.0.0  
6 Bundle-Vendor: Bachir Djafri  
7 Automatic-Module-Name: Contrat  
8 Bundle-RequiredExecutionEnvironment: JavaSE-13  
9 Bundle-ActivationPolicy: lazy  
10 Export-Package: exemple.contrat  
11
```

Le composant de contrat fournit une ressource, un package, qui définit les types de services utilisés et implémentés par d'autres composants.

Un composant de contrat exporte un ou plusieurs packages, chacun définissant un ou plusieurs types sous forme d'interfaces Java.

Dans cet exemple, le type HelloWorld est exporté (offert) par le composant de contrat Contrat.

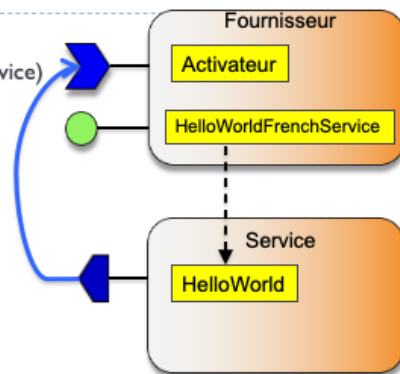
## Exemple 2

### ► Bundle fournisseur

- Implémente l'interface HelloWorld (type de service) du bundle du service précédent
  - Plus précisément une de ses classes implémente le service
- Il doit donc connaître l'existence de cette interface (**indiqué dans le manifeste**)

```
1 Manifest-Version: 1.0
2 Bundle-ManifestVersion: 2
3 Bundle-Name: ProviderFr
4 Bundle-SymbolicName: ProviderFr
5 Bundle-Version: 1.0.0
6 Bundle-Activator: providerfr.Activateur
7 Bundle-Vendor: Bachir Djafri
8 Bundle-RequiredExecutionEnvironment: JavaSE-13
9 Import-Package: exemple.contrat,
10 org.osgi.framework;version="1.3.0"
11 Automatic-Module-Name: ProviderFr
12 Bundle-ActivationPolicy: lazy
13
```

- Il doit en outre fournir le service implémenté à l'extérieur
  - Ceci est fait dans la classe d'activation



## Exemple 2

- ▶ La classe implémentant l'interface et donc le service désiré
- ▶ Service = objet d'une classe implémentant le type de service (interface Java)
- ▶ L'activateur peut aussi implémenter cette interface

**HelloWorldFrenchService**

```
1 package providerfr;  
2  
3 import exemple.contrat>HelloWorld;  
4  
5 public class HelloWorldFrenchService implements HelloWorld {  
6  
7     public void hello() {  
8         System.out.println("Bonjour !");  
9     }  
10  
11     public void goodbye() {  
12         System.out.println("Au revoir !");  
13     }  
14 }  
15
```

▶ 36

Bachir Djafri

M1 Miage – IDC

La classe activateur peut aussi implémenter le type de service. Elle peut même implémenter plusieurs types de services.

Exemple :

```
public class Activateur implements BundleActivator, HelloWorld {  
private ServiceRegistration<HelloWorld> sr;
```

```
public void start(BundleContext context) throws Exception {  
    Dictionary<String, String> props = new Hashtable<>();  
    props.put("Langue", "En");  
    sr = context.registerService(HelloWorld.class, this, props);  
    System.out.println("Un service anglais de type HelloWorld vient d'être  
    enregistré.");  
}
```

```
public void stop(BundleContext context) throws Exception {  
    sr.unregister();  
    System.out.println("Un service anglais de type HelloWorld vient d'être retiré.");  
}
```

```
}  
public void hello() {  
    System.out.println("Hello!");  
}  
public void goodbye() {  
    System.out.println("Good bye!");  
}  
}
```

## Exemple 2

### ► La classe d'activation

- Dans la méthode `start`, on indique au contexte d'enregistrer un service que le bundle fournit

Activateur

Des propriétés sont attachées à ce service (Langue=Fr)

```
1 package providerfr;
2
3 import java.util.Dictionary;
4 import java.util.Hashtable;
5
6 import org.osgi.framework.BundleActivator;
7 import org.osgi.framework.BundleContext;
8 import org.osgi.framework.ServiceRegistration;
9
10 import exemple.contrat.HelloWorld;
11
12 public class Activateur implements BundleActivator {
13
14     private static BundleContext context;
15     private ServiceRegistration<HelloWorld> sr;
16
17     static BundleContext getContext() {
18         return context;
19     }
20
21     public void start(BundleContext bundleContext) throws Exception {
22         Activateur.context = bundleContext;
23         Dictionary<String, String> props = new Hashtable<>();
24         props.put("Langue", "Fr");
25         sr = context.registerService(HelloWorld.class, new HelloWorldFrenchService(), props);
26         System.out.println("Un service de type HelloWorld vient d'être enregistré.");
27     }
28
29     public void stop(BundleContext bundleContext) throws Exception {
30         Activateur.context = null;
31         sr.unregister();
32     }
33 }
34
```

L'objet `ServiceRegistration<TypeService>` est important pour dé-enregistrer le service lors de la désactivation du composant (`sr.unregister()`).

Il peut aussi être utilisé pour récupérer la référence du service enregistré (`ServiceReference getReference()`).

Il peut aussi être utilisé pour changer, dynamiquement, les propriétés du service associé (`void setProperties(Dictionary props)`). C'est le composant fournisseur qui détient cet objet après avoir enregistré son service avec ses propriétés et son type.  
`sr = context.registerService(TypeService.class, new ObjetService(), propriétés);`

La classe activateur peut aussi implémenter le type de service. Elle peut même implémenter plusieurs types de services.

Exemple :

```
public class Activateur implements BundleActivator, HelloWorld {
    private ServiceRegistration<HelloWorld> sr;
```

```
public void start(BundleContext context) throws Exception {
    Dictionary<String, String> props = new Hashtable<>();
```

```
props.put("Langue", "En");  
sr = context.registerService>HelloWorld.class, this, props);  
System.out.println("Un service anglais de type HelloWorld vient d'être enregistré.");  
}
```

```
public void stop(BundleContext context) throws Exception {  
    sr.unregister();  
    System.out.println("Un service anglais de type HelloWorld vient d'être retiré.");  
}  
public void hello() {  
    System.out.println("Hello!");  
}  
public void goodbye() {  
    System.out.println("Good bye!");  
}  
}
```

## Exemple 2

### ► Bundle client

- Il a besoin du service pour fonctionner. Donc à l'exécution, il devra le trouver
- À la compilation, il doit aussi connaître le type « HelloWorld », d'où

```
1 Manifest-Version: 1.0
2 Bundle-ManifestVersion: 2
3 Bundle-Name: Client
4 Bundle-SymbolicName: Client
5 Bundle-Version: 1.0.0
6 Bundle-Activator: client.Activateur
7 Bundle-Vendor: Bachir Djafri
8 Bundle-RequiredExecutionEnvironment: JavaSE-13
9 Import-Package: exemple.contrat,
10 org.osgi.framework;version="1.3.0"
11 Automatic-Module-Name: Client
12 Bundle-ActivationPolicy: lazy
13
```

un import du package adéquat



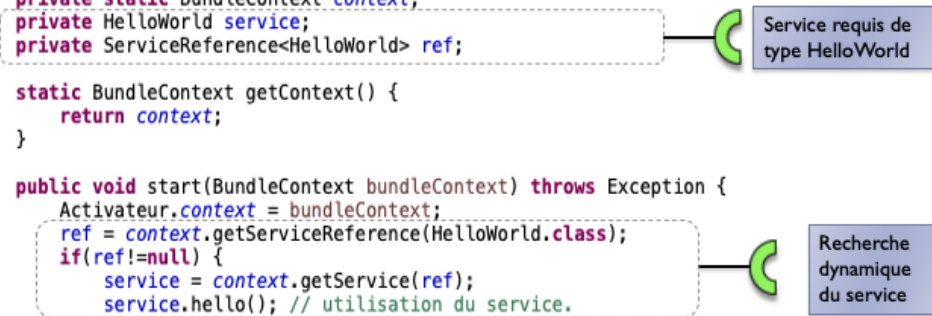


## Exemple 2

### ► Bundle client

- Dans la méthode start() de la classe d'activation, le bundle indique les services qu'il recherche (dont il a besoin)

```
9 public class Activateur implements BundleActivator {
10
11     private static BundleContext context;
12     private HelloWorld service;
13     private ServiceReference<HelloWorld> ref;
14
15     static BundleContext getContext() {
16         return context;
17     }
18
19     public void start(BundleContext bundleContext) throws Exception {
20         Activateur.context = bundleContext;
21         ref = context.getServiceReference(HelloWorld.class);
22         if(ref!=null) {
23             service = context.getService(ref);
24             service.hello(); // utilisation du service.
25         } else { // aucun service de type HelloWorld.
26             service = null;
27             System.out.println("aucun service HelloWorld n'a été trouvé.");
28         }
29     }
30 }
```



L'objet ServiceReference contient le type du service, l'objet de service et les propriétés du service. 3 éléments essentiels pour manipuler un service.

Dans cet exemple, le service requis est optionnel. Si on ne trouve pas de service, le composant est activé sans service. Le service est donc optionnel.

Si le service était obligatoire, alors le composant ne devrait pas être activé. Il doit, soit attendre la disponibilité de ce service (boucle), soit interrompre l'activation en levant une exception dans la méthode start().

Exemple 1 :

```
if(ref!=null) {
    service = context.getService(ref);
    service.hello(); // utilisation du service.
} else { // aucun service de type HelloWorld.
    service = null;
    throw new Exception("Exception : aucun service trouve");
}
```

Exemple 2 :

```
ref=null;
while(ref==null) {
    ref = context.getServiceReference>HelloWorld.class);
    if(ref!=null) { service = context.getService(ref); }
}
service.hello(); // utilisation du service.
```

## Exemple 2

### ► Bundle client

- Dans la méthode stop() de la classe d'activation, le bundle indique les services qu'il n'utilise plus au contexte.

```
34
35 public void stop(BundleContext bundleContext) throws Exception {
36     if(service!=null) {
37         service.goodbye();
38         service=null;
39         context.ungetService(ref);
40         ref=null;
41     }
42     Activateur.context = null;
43 }
44 }
45 |
```

Dans la méthode stop(), on doit remettre la référence du service ServiceReference ainsi que le service de type HelloWorld à null (après les avoir utilisé pour indiquer l'arrêt du composant).