Génie Logiciel Avancé

Horatiu CIRSTEA

Sujets abordés

- Développement avec composants - EJB
  - Session Beans
  - Message Driven Beans
- Modélisation du domaine - JPA
- Aspects transversaux: intercepteurs
- Timers & schedulers

Sujets abordés

- Développement avec composants - EJB
  - Session Beans
  - Message Driven Beans
- Modélisation du domaine - JPA
- Aspects transversaux: intercepteurs
- Timers & schedulers

http://arche.univ-lorraine.fr/course/view.php?id=314

Sources et lectures supplémentaires

- Beginning Java EE 6 with GlassFish 3, A. Goncalves
- EJB 3 in Action, D.Panda, R.Rahman, D.Lane, *Manning*
- The Java EE Tutorial (oracle.com)
- The NetBeans E-commerce Tutorial (netbeans.org)

Développement avec composants

- Entreprise Java Beans -

Généralités

Applications entreprise

- commerce électronique
- domaine bancaire et assurance
- gestion/planification ressources
- ...

Applications entreprise

- Distribuées
- Scalables (supportant la montée en charge)
- Disponibles
- Supportant la réutilisation
- Maintenables et extensibles
- Sûres
- Sécurisées

Applications entreprise

- nombreuses sources d'informations potentielles
- interactions entre des systèmes hétérogènes
- grand nombre de transactions
- évolutives

Choses à résoudre

- invocations de méthodes (distantes)
- persistance et intégration avec le back-end
- gestion de transactions
- multithreading
- gestion des ressources
- gestion des messages

Choses à résoudre

- redéploiement (à chaud)
- arrêt de serveurs sans interrompre l'application
- clustering (transparence des défaillances)
- gestion de la charge
- sécurité

Qui s'occupe de tout ceci ?

Middleware

- Composants ou systèmes permettant de rendre transparents les problèmes de distribution
- **Middleware développé en entreprise**
  - adresse rarement tous les problèmes
  - couts de développement/maintenance élevés
  - orthogonal au secteur d'activité de l'entreprise

Serveurs d'application

- prise en charge des aspects middleware par un système spécialisé
- utilise une architecture distribuée
- fournit une solution aux différents problèmes précédents

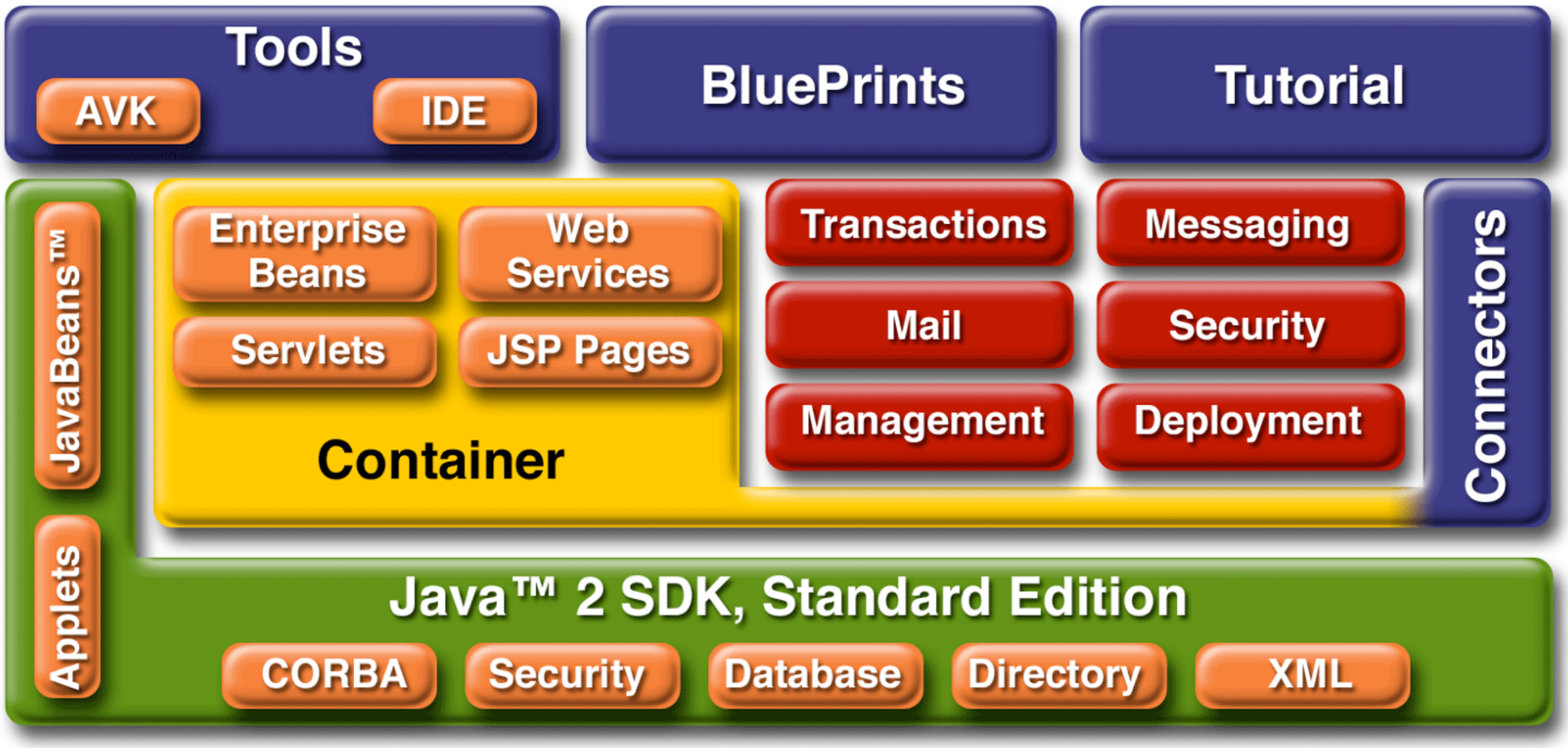
Séparation des métiers et des spécificités : d'un côté la logique métier, de l'autre la logique middleware.

Introduction
Java EE

Java EE

- un modèle de programmation (application multi-tiers, client légers)
- une plate-forme (ensemble de spécifications et de politiques requises)
- un ensemble de tests de compatibilité
- une implantation de référence
- des patrons de conception (blueprints)

Java EE



java.sun.com

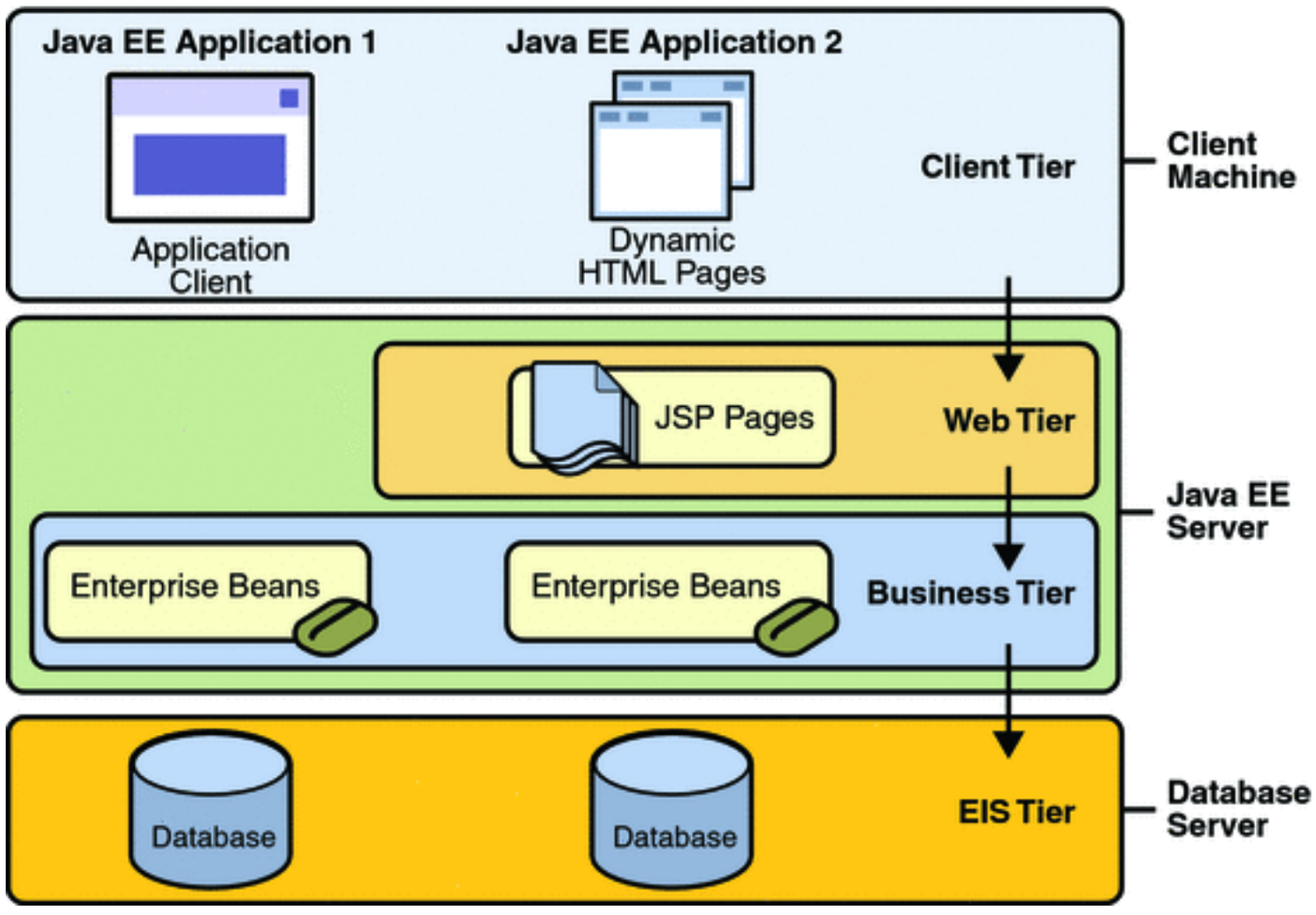
Développement avec composants

Un composant

- implante des interfaces prédéfinies
- logique facilement réutilisable
- intégrable dans le middleware

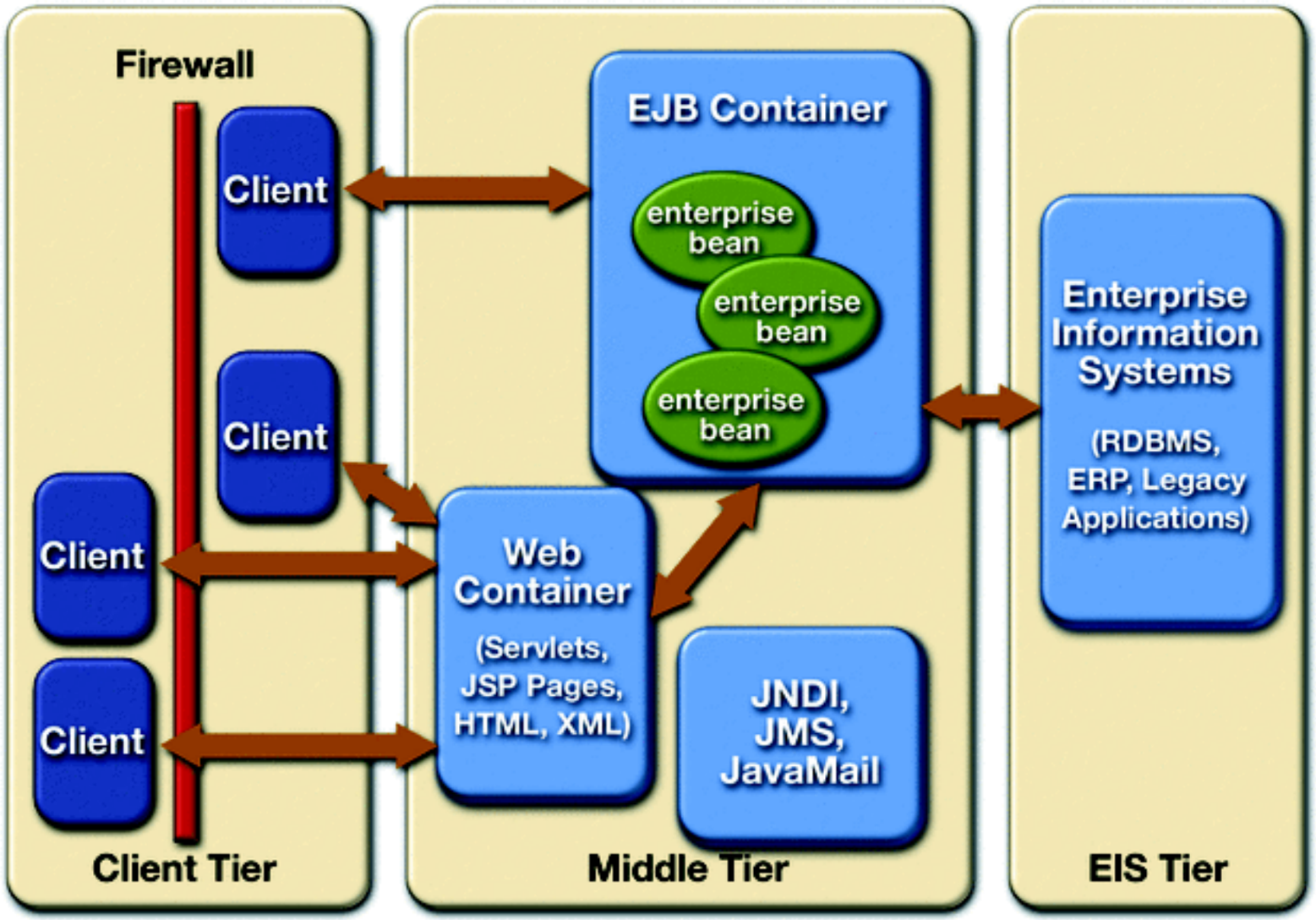
On assemble les composants comme un puzzle, afin de résoudre des problèmes importants.

Architecture multi-tiers



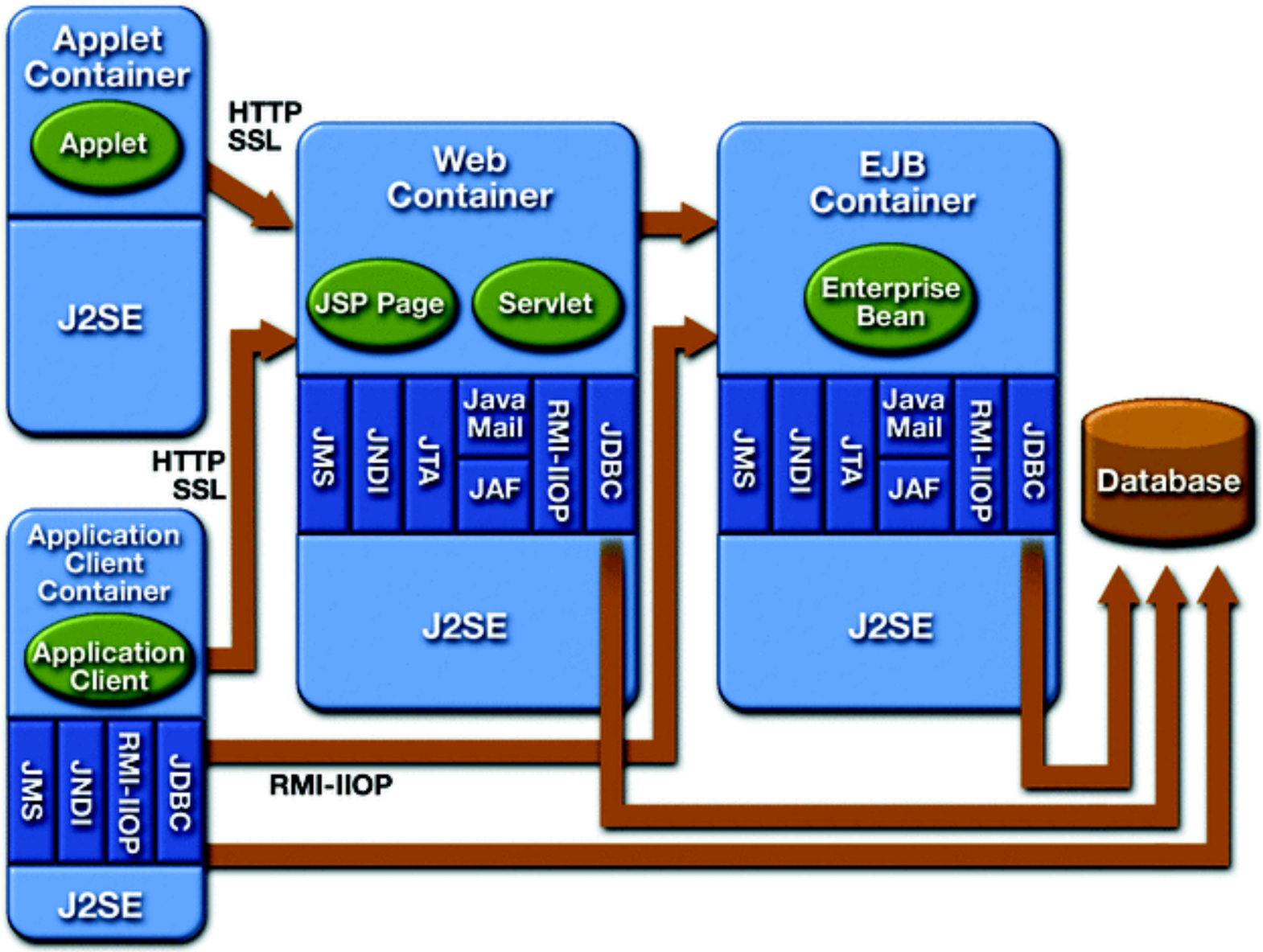
oracle.com

Application JEE



java.sun.com

Technologies



java.sun.com

APIs

- EJB (Enterprise Java Beans)
- Java RMI/RMI-IIOP (Remote Method Invocation)
- JNDI (Java Naming and Directory Interface)
- JDBC (Java Data Base Connectivity)
- JTA (Java Transaction API)
- JMS (Java Messaging Service)

APIs

- Java Servlets and Java Pages (JSP)
- Java IDL (Corba)
- JavaMail
- JCA (J2EE Connector Architecture)
- JAXP (Java API for XML Parsing)
- JAAS (Java Authentification and Authorization Service)

Entreprise Java Beans

EJB

Framework pour des ***composants***

EJB

Framework pour des ***composants***

- logiciels
- écrits en Java
- exposant une interface
- déployables dans un container d'EJB
- configurables de façon externe

EJB

Framework pour des ***composants***

- logiciels
- écrits en Java
- exposant une interface
- déployables dans un container d'EJB
- configurables de façon externe

Spécification API publique (Java Community Process).

Le framework EJB

Le framework EJB

- Dependency Injection, thread-safety, pooling
- Transactions
- Persistance
- Sécurité
- Interceptors
- Timers
- Messaging...

Composant EJB

- Composant
  - conforme à la spécification EJB
- Les clients sont
  - des servlets
  - des applets
  - des EJBs
  - des applications classiques ...

Métiers

- Le fournisseur d'EJBs
- L'assembleur d'application
- Le déployeur d'EJBs
- L'administrateur système
- Le fournisseur de conteneur EJB (JBoss, JRun,...)
- Le fournisseur de serveur d'application
- Le fournisseur de persistance

EJB

Prendre une classe Java et la rendre ***distribuée***,
sécurisée et ***transactionnelle***

EJB

Prendre une classe Java et la rendre ***distribuée***,
sécurisée et ***transactionnelle***

POJO + @ =

meta-data annotations

EJB

Prendre une classe Java et la rendre ***distribuée***,
sécurisée et ***transactionnelle***

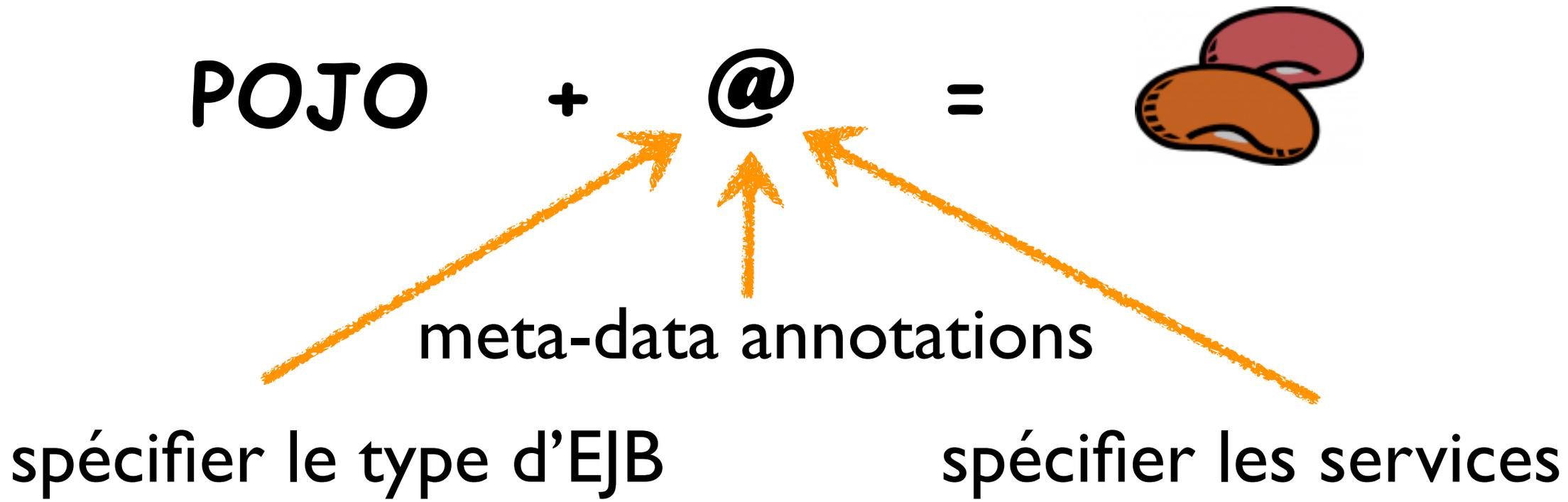
POJO + **@** =

meta-data annotations

spécifier le type d'EJB

EJB

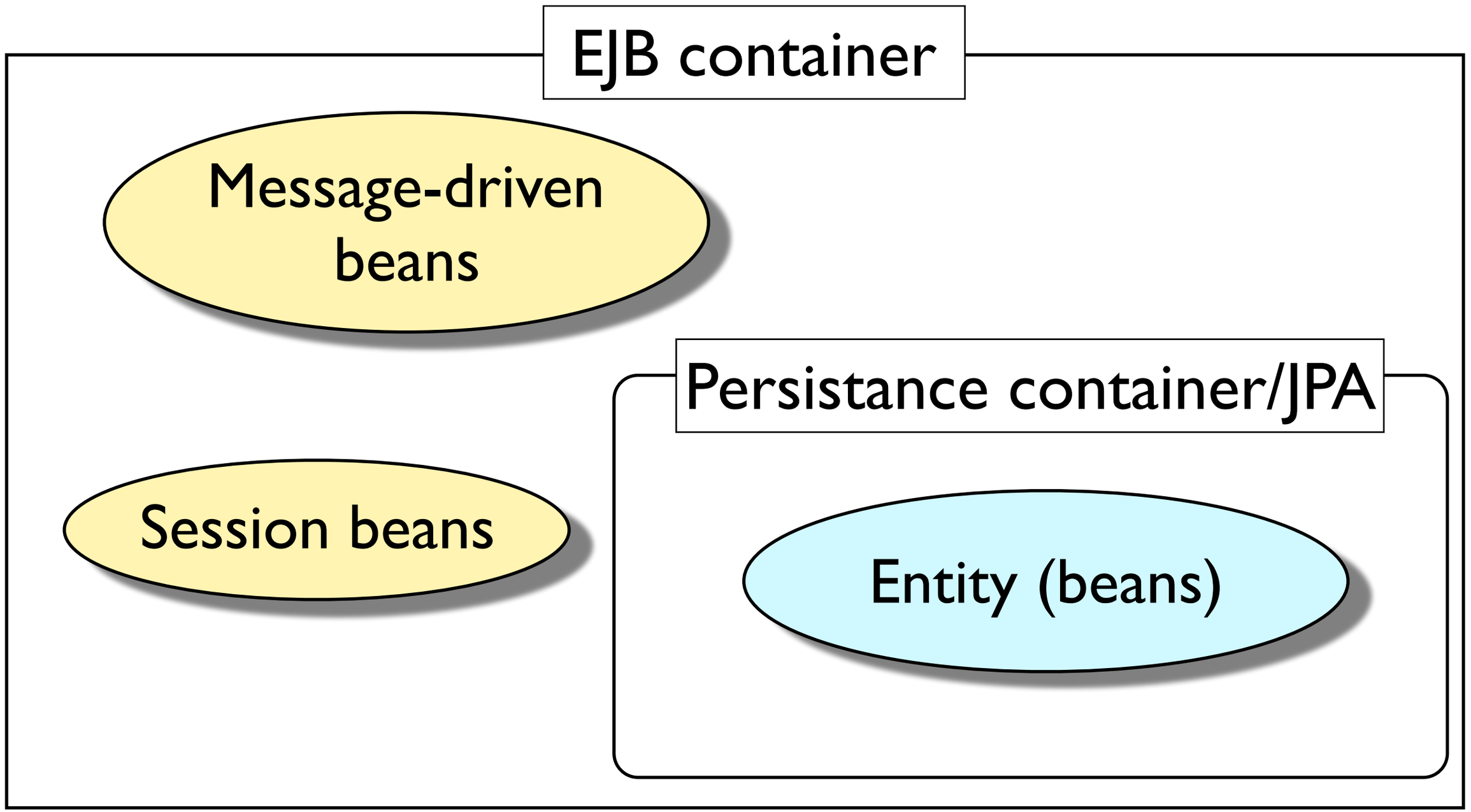
Prendre une classe Java et la rendre ***distribuée***,
sécurisée et ***transactionnelle***



Les types de beans

- Session beans
- Message-Driven beans
- Entity (beans)

Les types de beans



Session Beans

- contiennent la logique métier de l'application
  - appelé pour effectuer un traitement
  - *exemple* : vérifier des données bancaires
- **session** ⇨ durée de vie limitée à la session d'un client (ne survit pas au shutdown du serveur)

Message-Driven Beans

- contiennent de la logique métier
- invoqués en envoyant des messages à un *serveur** de messagerie (un client n'invoque jamais explicitement un MDB)
- *exemple* : demande d'inventaire hebdomadaire

* *IBM WebSphere MQ, SonicMQ, Oracle Advanced Queueing, TIBCO*

Entities

- la représentation OO des données (stockées dans une BDD) de l'application
- on peut ajouter de la logique métier
- *exemple*: User, Item, ...
- persistance gérée par un « Entity Manager »
- pas vraiment des beans en JEE (≥6)

Session Beans

Session Bean

- traite une requête d'un client
- représente une action, un verbe, une logique métier
- *exemples*
  - rechercher un article
  - envoyer une commande
  - ajouter des items dans un caddy
  - ...

Interfaces possibles

- **`@Local`** : depuis un client local (même JVM)
- **`@Remote`** : depuis un client distant (RMI/IIOP)
- **`@WebService`** : service web
- pas d'interface

Types

Types

- un EJB traite une requête d'un client

Types

- un EJB traite une requête d'un client

  ⇨ suites d'appels de méthodes

Types

- un EJB traite une requête d'un client

  ⇨ suites d'appels de méthodes

- **3 types de Session Beans**
  - **Stateless :** l'état n'est pas conservé entre deux appels
  - **Stateful :** l'état est conservé entre deux appels
  - **Singleton :** partagé par les clients durant la durée de vie de l'application

HelloWorld

```
package m2nancy;

@Local
public interface HelloWorld {

    public void sayHello(String name) ;

}
```

HelloWorld

```
package m2nancy;

@Local
public interface HelloWorld {

    public void sayHello(String name) ;

}
```

POJI

HelloWorld

```
package m2nancy;

@Local
public interface HelloWorld {

    public void sayHello(String name) ;

}
```

interface locale

POJI

HelloWorld

```
package m2nancy;

@Local
public interface HelloWorld {

    public void sayHello(String name) ;

}
```

```java
package m2nancy;

@Local
public interface HelloWorld {

    public void sayHello(String name) ;

}
```

```java
package m2nancy;
import javax.ejb.Stateless;

@Stateless
public class HelloWorldBean
        implements HelloWorld {
    public HelloWorldBean() {    }

    public void sayHello(String name) {
        System.out.println("Salut "+name);
    }
}
```

```java
package m2nancy;

@Local
public interface HelloWorld {

    public void sayHello(String name) ;

}
```

```java
package m2nancy;
import javax.ejb.Stateless;

@Stateless
public class HelloWorldBean
        implements HelloWorld {
    public HelloWorldBean() {    }

    public void sayHello(String name) {
        System.out.println("Salut "+name);
    }
}
```

POJO

```java
package m2nancy;

@Local
public interface HelloWorld {

    public void sayHello(String name) ;

}
```

```java
package m2nancy;
import javax.ejb.Stateless;

@Stateless
public class HelloWorldBean
        implements HelloWorld {
    public HelloWorldBean() {    }

    public void sayHello(String name) {
        System.out.println("Salut "+name);
    }
}
```

stateless bean

POJO

Bean interface/classe

- **un bean peut avoir plusieurs interfaces**
  - plusieurs clients peuvent utiliser le bean de différentes manières
- un bean ne peut pas être `abstract`
- un bean peut avoir des méthodes non-privées non-accessible par l'interface (test, lifecycle callbacks)

Règles EJB

- au moins une interface (plus le cas depuis EJB 3.1)
- classe concrète - pas `final`, pas `abstract`
- un constructeur sans arguments
- une classe EJB peut hériter un autre EJB/POJO

```
@Stateless
public class HelloBean extends HelloWorldBean implements Hello
```

- les annotations `@Stateless/@Stateful` ignorées dans l'héritage
- nom de méthodes ne commencent pas avec `ejb`

Conversational state

- **Stateful session bean**
  - l'état préservé entre 2 appels de méthode
  - en général, pour modéliser workflows multistep
    - ex : registration, shopping cart
- **Stateless session beans**
  - aucune information état préservée
  - en général, pour modéliser des services utilitaires
  - ex : view item, post review

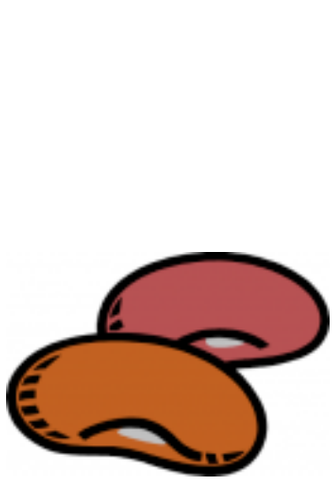
Comment utiliser les beans?

- recherche explicite de ressources
  - JNDI lookup
- via la configuration
  - dependency injection

JNDI lookup

EJB

JNDI lookup



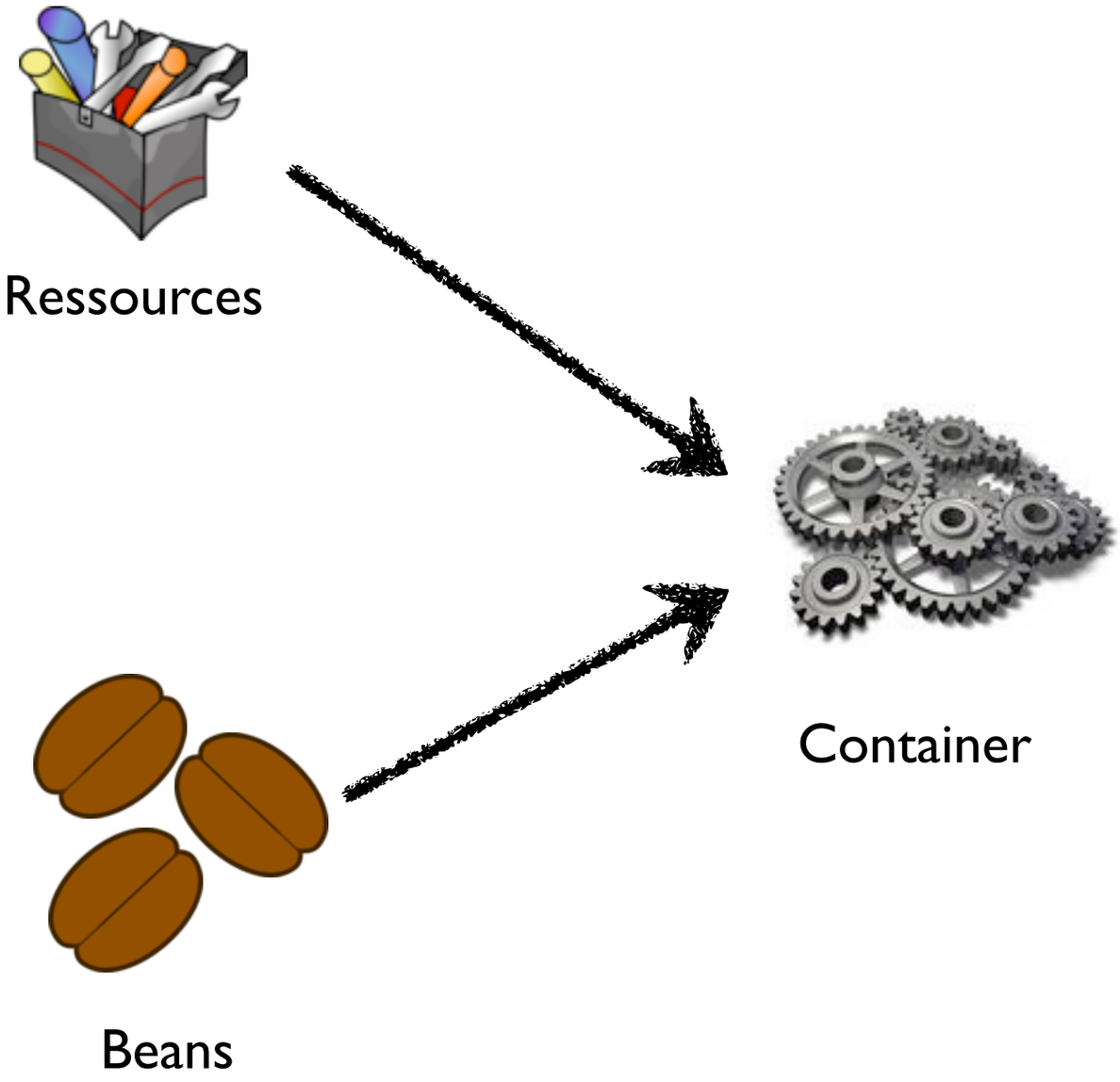
EJB

Lookup

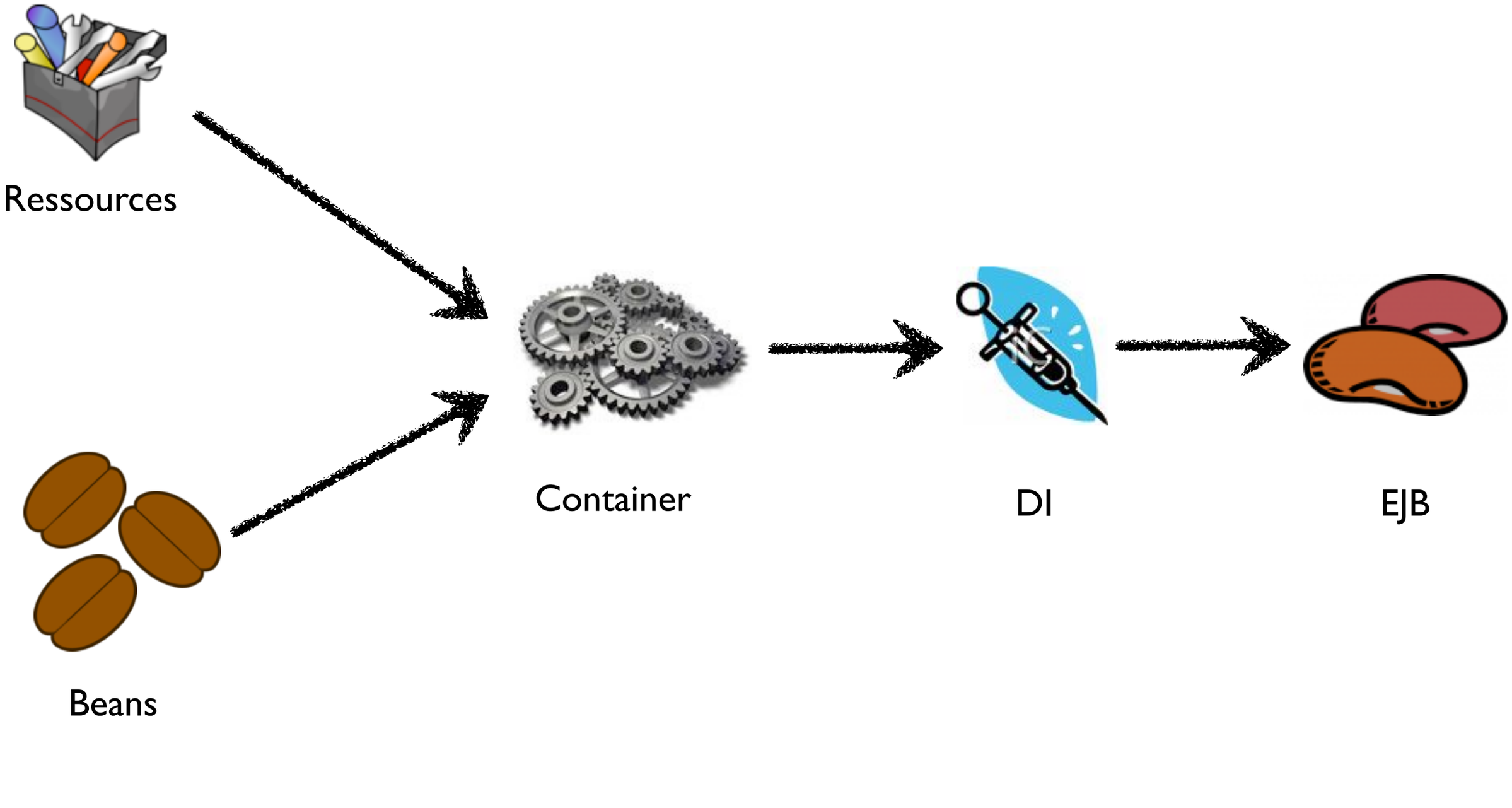
Ressources

Beans

Dependency injection



Dependency injection



Client

```
package m2nancy;

@Local
public interface HelloWorld {
    public void sayHelloTo(String name);
}
```

```
import javax.ejb.EJB;
import m2nancy.HelloWorld;

public class HelloWorldClient {
   @EJB
   private static HelloWorld hello;

   public static void main(String [] args) {
      try {
         hello.sayHello("James Bond");
      } catch (Exception ex) {
         ex.printStackTrace();
      }
} }
```

Client

```
package m2nancy;

@Local
public interface HelloWorld {
    public void sayHelloTo(String name);
}
```

```
import javax.ejb.EJB;
import m2nancy.HelloWorld;

public class HelloWorldClient {
  @EJB
  private static HelloWorld hello;
  public static void main(String [] args) {
    try {
      hello.sayHello("James Bond");
    } catch (Exception ex) {
      ex.printStackTrace();
    }
} }
```

injection EJB

Client avec lookup

```
import javax.naming.Context;
import javax.naming.InitialContext;
import m2nancy.HelloWorld;

public class HelloWorldJNDIClient {
 public static void main(String [] args) {
   try {
     Context context = new InitialContext();
     HelloWorld hello = (HelloWorld)context.lookup(
                "java:global/HelloWorld/HelloWorld-ejb/
                 HelloWorldBean!buslogic.HelloWorld");
     hello.sayHello("James Bond");
   } catch (Exception ex) {
     ex.printStackTrace();
   }
} }
```

Client avec lookup

```
import javax.naming.Context;
import javax.naming.InitialContext;
import m2nancy.HelloWorld;

public class HelloWorldJNDIClient {
 public static void main(String [] args) {
  try {
    Context context = new InitialContext();
    HelloWorld hello = (HelloWorld)context.lookup(
                "java:global/HelloWorld/HelloWorld-ejb/
                 HelloWorldBean!buslogic.HelloWorld");
    hello.sayHello("James Bond");
  } catch (Exception ex) {
    ex.printStackTrace();
  }
} }
```

lookup

Client avec lookup

```
import javax.naming.Context;
import javax.naming.InitialContext;
import m2nancy.HelloWorld;

public class HelloWorldJNDIClient {
 public static void main(String [] args) {
  try {
    Context context = new InitialContext();
    HelloWorld hello = (HelloWorld)context.lookup(
              "java:global/HelloWorld/HelloWorld-ejb/
               HelloWorldBean!buslogic.HelloWorld");
    hello.sayHello("James Bond");
  } catch (Exception ex) {
    ex.printStackTrace();
  }
} }
```

lookup

EAR

EJB-JAR

bean

interface

Interfaces possibles

- **`@Local`** : depuis un client local (même JVM)
- **`@Remote`** : depuis un client distant (RMI/IIOP)
- **`@WebService`** : service web
- pas d'interface

Interface locale

```
package m2nancy;

@Local
public interface HelloWorld {

    public void sayHelloTo(String name) ;

}
```

Interface Remote

```
package m2nancy;

@Remote
public interface HelloWorld extends Remote {
    public void sayHelloTo(String name) ;

}
```

Interface Remote

`java.rmi.Remote`
optionnelle

```
package m2nancy;

@Remote
public interface HelloWorld extends Remote {
    public void sayHelloTo(String name) ;
}
```

`throws java.rmi.RemoteException`
implicite

Interface Remote

`java.rmi.Remote`
optionnelle

```
package m2nancy;

@Remote
public interface HelloWorld extends Remote {
    public void sayHelloTo(String name) ;
}
```

`throws java.rmi.RemoteException`
implicite

Tous les paramètres et types de retour doivent être `Serializable`.

Interfaces multiples

```
public interface HelloWorld {
    public void sayHelloTo(String name) ;
}

@Local
public interface HelloWorldLocal
                              extends HelloWorld{
   ...
}

@Remote
public interface HelloWorldRemote
                               extends HelloWorldLocal {
   ...
}
```

Cycle de vie

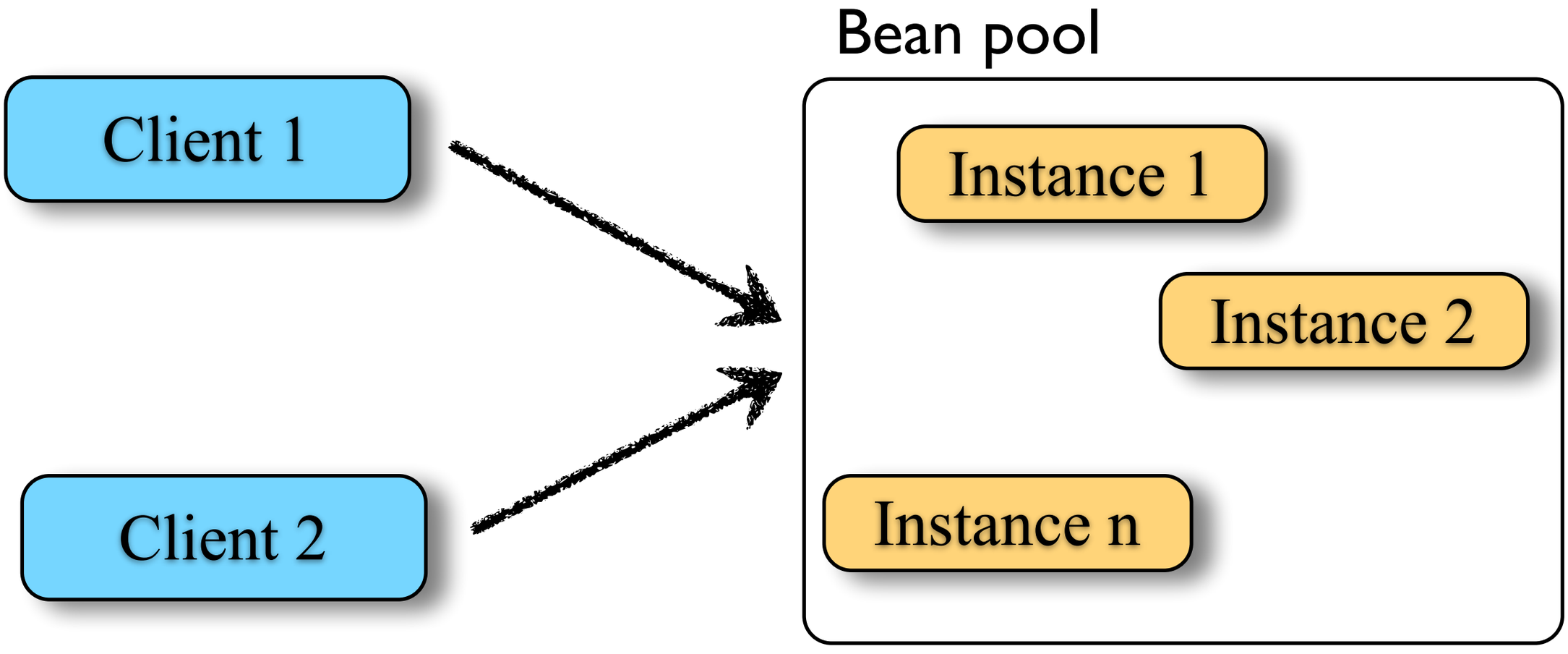
- le *container* gère les session beans
  - création instances
  - injection dépendances
  - destruction instances
  - optimisations

Stateless Session Beans

Stateles session beans

- modélisent des taches réalisables avec un seul appel de méthode
- réalisent en général plusieurs tâches (implante plusieurs méthodes)
- les plus populaires
- les plus efficaces (bean pools)

Bean pools



Une instance de Stateless Session Bean n'est pas propre à un client donné, elle peut être partagée entre chaque appel de méthode.

Stateless interface

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Stateless {
    String name() default "";
    String mappedName() default "";
    String description() default "";
}
```

Stateless interface

nom
(utiliser pour bind JNDI)

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Stateless {
    String name() default "";
    String mappedName() default "";
    String description() default "";
}
```

- si le nom est omis le container utilise le nom de la classe

Stateless interface

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Stateless {
    String name() default "";
    String mappedName() default "";
    String description() default "";
}
```

nom
(utiliser pour bind JNDI)

vendor-specific

- si le nom est omis le container utilise le nom de la classe

HelloWorld

```
package m2nancy;
import javax.ejb.Stateless;

@Stateless (name = "HelloWorld")
public class HelloWorldBean
                    implements HelloWorld {
    public HelloWorldBean() {    }

    public void sayHello(String name) {
        System.out.println("Salut "+name);
    }
}
```

Cycle de vie - Session beans

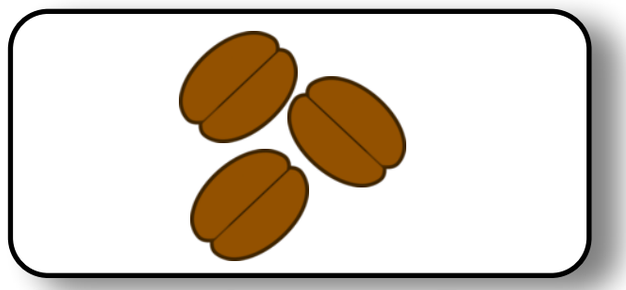
Bean n’existe pas

Cycle de vie - Session beans

container décide
que de nouvelles instances
sont nécessaires

Bean n'existe pas

création instance 
injection dépendances
@PostConstruct



Bean dans le Pool

Cycle de vie - Session beans

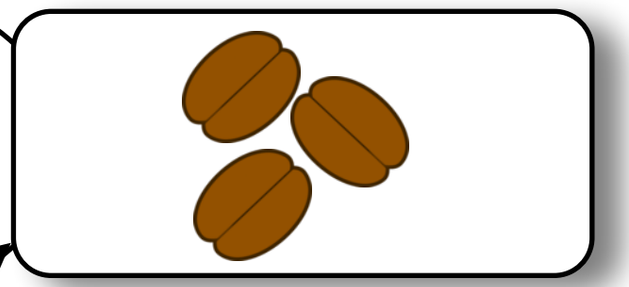
container décide
que de nouvelles instances
sont nécessaires

Bean n'existe pas

création instance

injection dépendances
@PostConstruct

appels méthodes
par client



Bean dans le Pool

Cycle de vie - Session beans

container décide
que de nouvelles instances
sont nécessaires

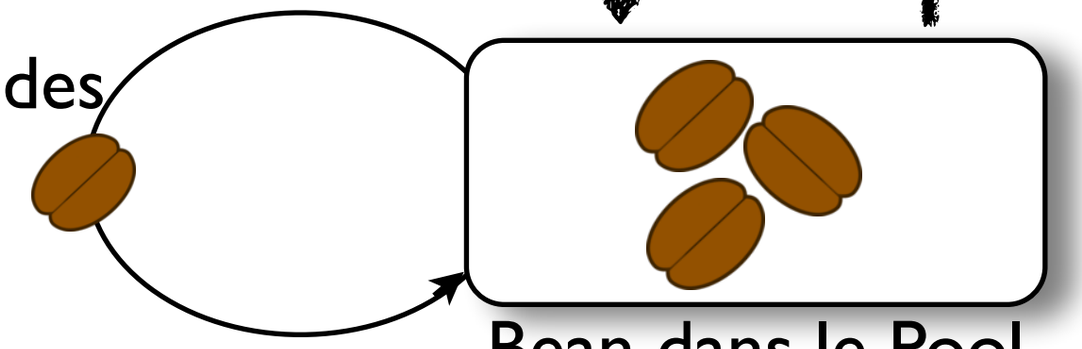
Bean n'existe pas

container décide que
moins d'instances sont
nécessaires

création instance
injection dépendances
@PostConstruct

@PreDestroy
destruction instance

appels méthodes
par client



Bean dans le Pool

Callbacks

- @PostConstruct
  - après la création, le setup et les injections
- @PreDestroy
  - avant la destruction et l'élimination du pool
- pattern
  - `void METHOD()`
  - ne peut pas lever des exceptions (non `Runtime`)
- utilisation : allocation/libération ressources

Exemple

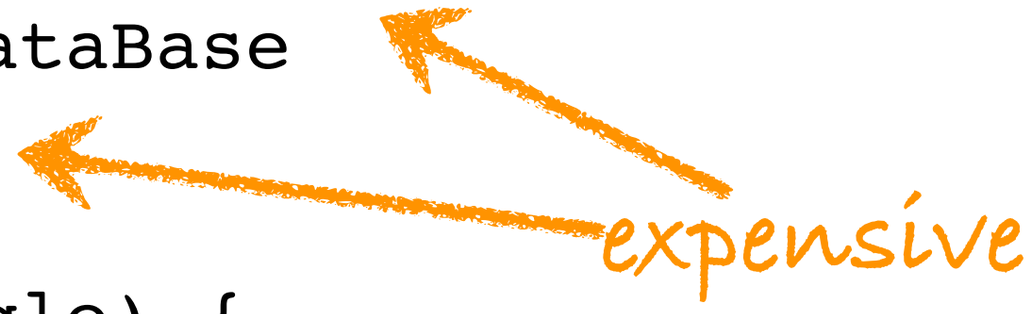
```
@Stateless(name = "PersonManager")
public class PersonManagerBean implements PersonManager
{
    @Resource(lookup = "jdbc/gla")
    private DataSource dataSource;
    private Connection connection;

    public void addPerson(Person person) {
        try {
            connection = dataSource.getConnection();
            // save person to DataBase
            connection.close();
            connection = null;
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
} }
```

Exemple

```
@Stateless(name = "PersonManager")
public class PersonManagerBean implements PersonManager
{
    @Resource(lookup = "jdbc/gla")
    private DataSource dataSource;
    private Connection connection;

    public void addPerson(Person person) {
        try {
            connection = dataSource.getConnection();
            // save person to DataBase
            connection.close();
            connection = null;
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
} }
```



Utiliser des callbacks

```
@Stateless(name = "PersonManager")
public class PersonManagerBean implements PersonManager
{
    @Resource(lookup = "jdbc/gla")
    private DataSource dataSource;
    private Connection connection;

    @PostConstruct
    public void initialize() {
        try {
            connection = dataSource.getConnection();
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
    }
    ...
}
```

Utiliser des callbacks

```
@Stateless(name = "PersonManager")
public class PersonManagerBean implements PersonManager
{
    @Resource(lookup = "jdbc/gla")
    private DataSource dataSource;
    private Connection connection;
    ...
    @PreDestroy
    public void cleanup() {
        try {
            connection.close();
            connection = null;
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
    }
}
```

Exemple

```
@Stateless(name = "PersonManager")
public class PersonManagerBean implements PersonManager
{
    @Resource(lookup = "jdbc/gla")
    private DataSource dataSource;
    private Connection connection;

    public void addPerson(Person person) {
        try {
            connection = dataSource.getConnection();
            // save person to DataBase
            connection.close();
            connection = null;
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
} }
```

Exemple

```
@Stateless(name = "PersonManager")
public class PersonManagerBean implements PersonManager
{
    @Resource(lookup = "jdbc/gla")
    private DataSource dataSource;
    private Connection connection;

    public void addPerson(Person person) {
        try {

            // save person to DataBase


        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
}  }
```

Utiliser des callbacks

```
@Stateless(name = "PersonManager")
public class PersonManagerBean implements PersonManager
{
    @Resource(lookup = "jdbc/gla")
    private DataSource dataSource;
    private Connection connection;

    public void addPerson(Person person) {
        try {

          Statement smt = connection.createStatement();
          smt.execute("INSERT INTO PERSONS(...) “);

        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
}  }
```

Stateful Session Beans

Stateful Session Beans

- l'état est conservé tout au long de la session
  - au cours d'appels de méthodes successifs
  - au cours de transactions successives
  - si un appel de méthode change l'état du Bean, lors d'un autre appel de méthode l'état sera disponible

Conséquence : une instance de Stateful Session Bean par client.

Performance

- une instance de Stateful Session Bean par client
  - un stateful bean ne peut pas être retourné dans le Pool avant la fin d'une session client
  - potentiellement un impact important sur la mémoire utilisée

Règles Stateful Beans

- variables d'instance utilisées pour sauvegarder l'état de type primitif ou `Serializable`
- une méthode qui permet la destruction explicite de l'objet (`@Remove`)
- des méthodes de passivation/ activation (`@PrePassivate`/`@PostActivate`)
- pas d'interface `@WebService`

Cycle de vie - Stateful beans

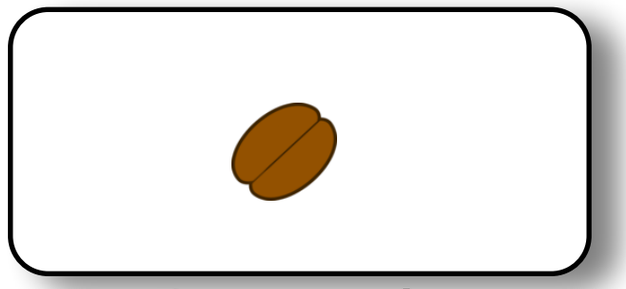
Bean n’existe pas

Cycle de vie - Stateful beans

Bean n'existe pas

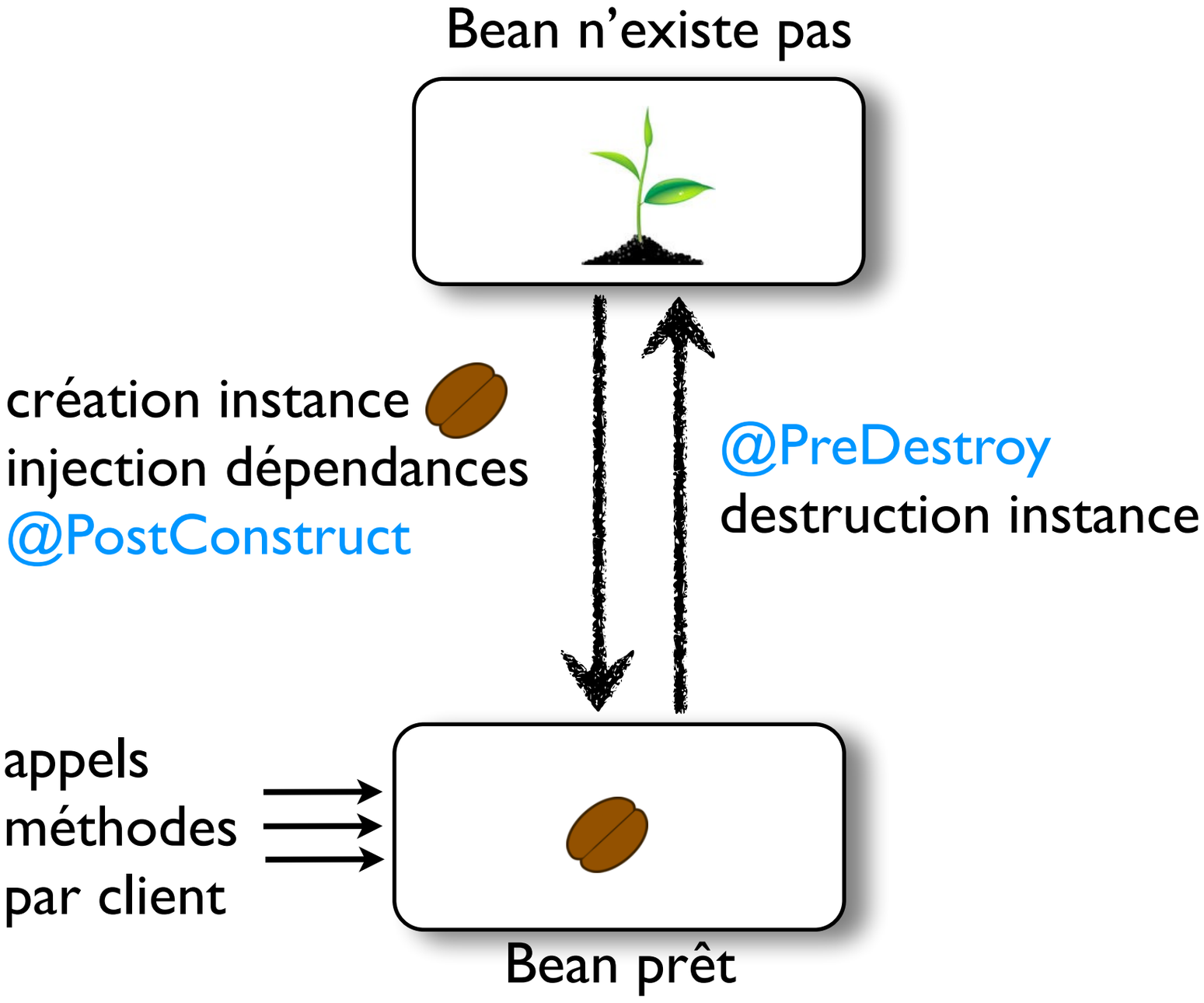
création instance 
injection dépendances
@PostConstruct

appels
méthodes
par client

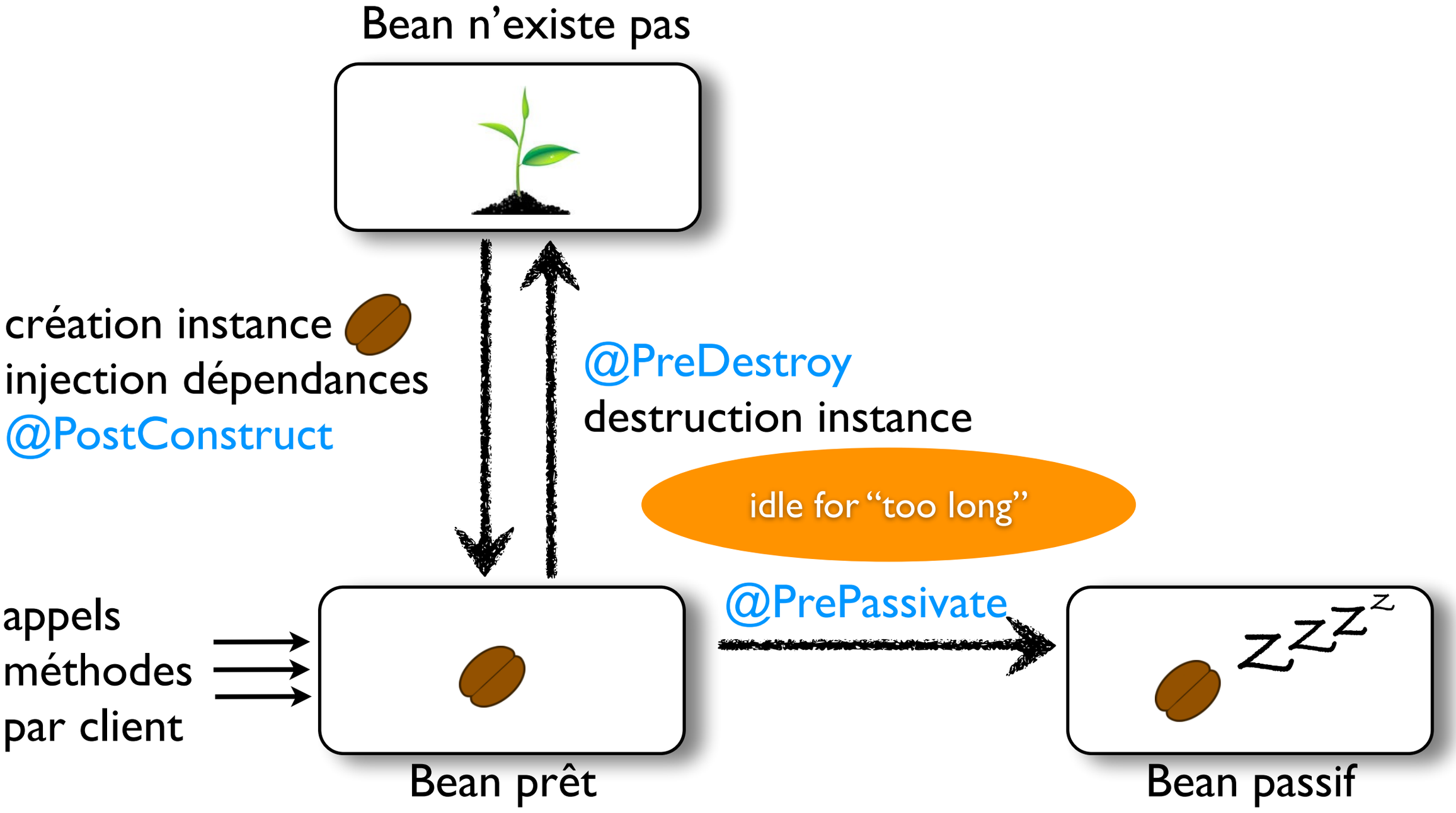


Bean prêt

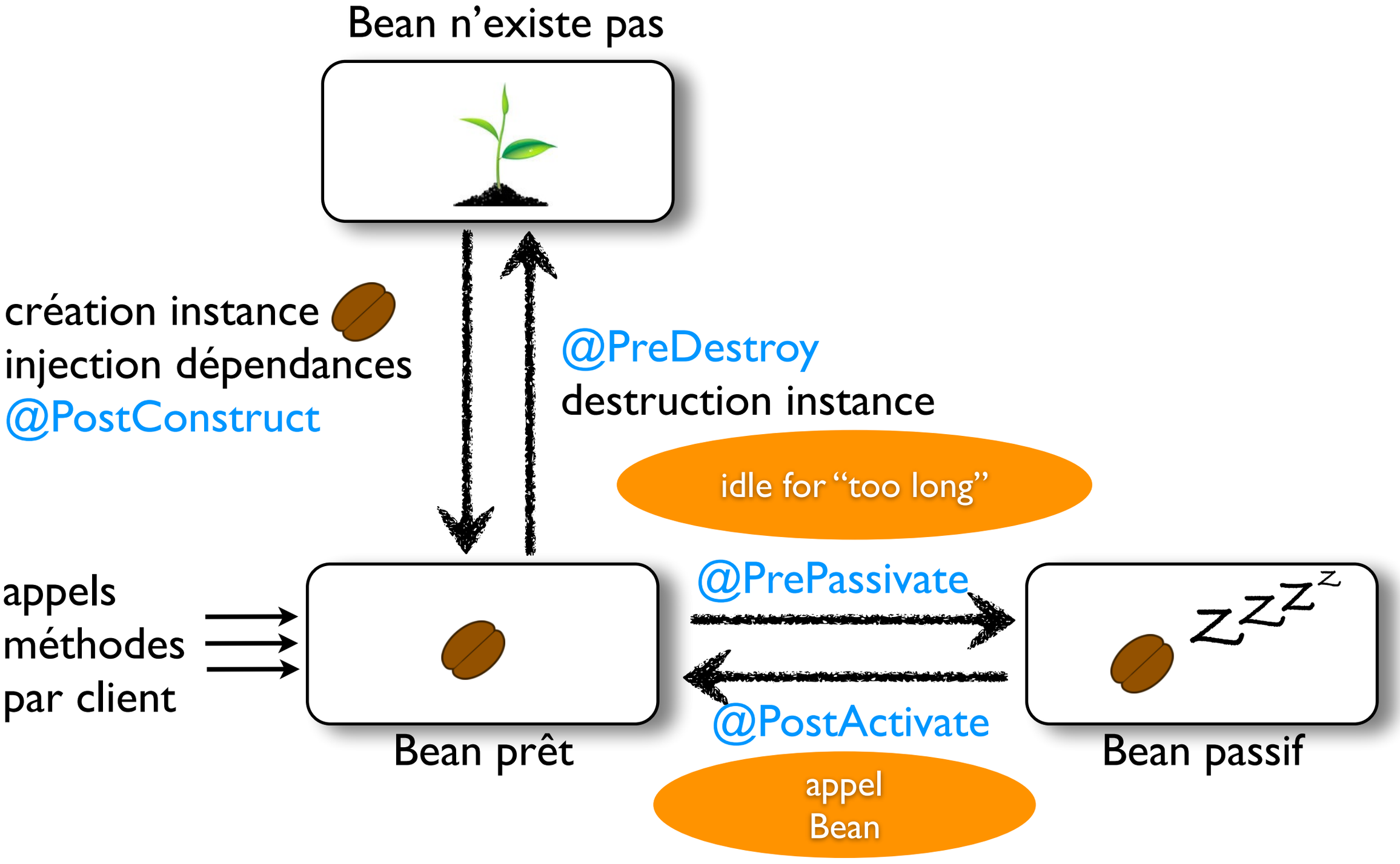
Cycle de vie - Stateful beans



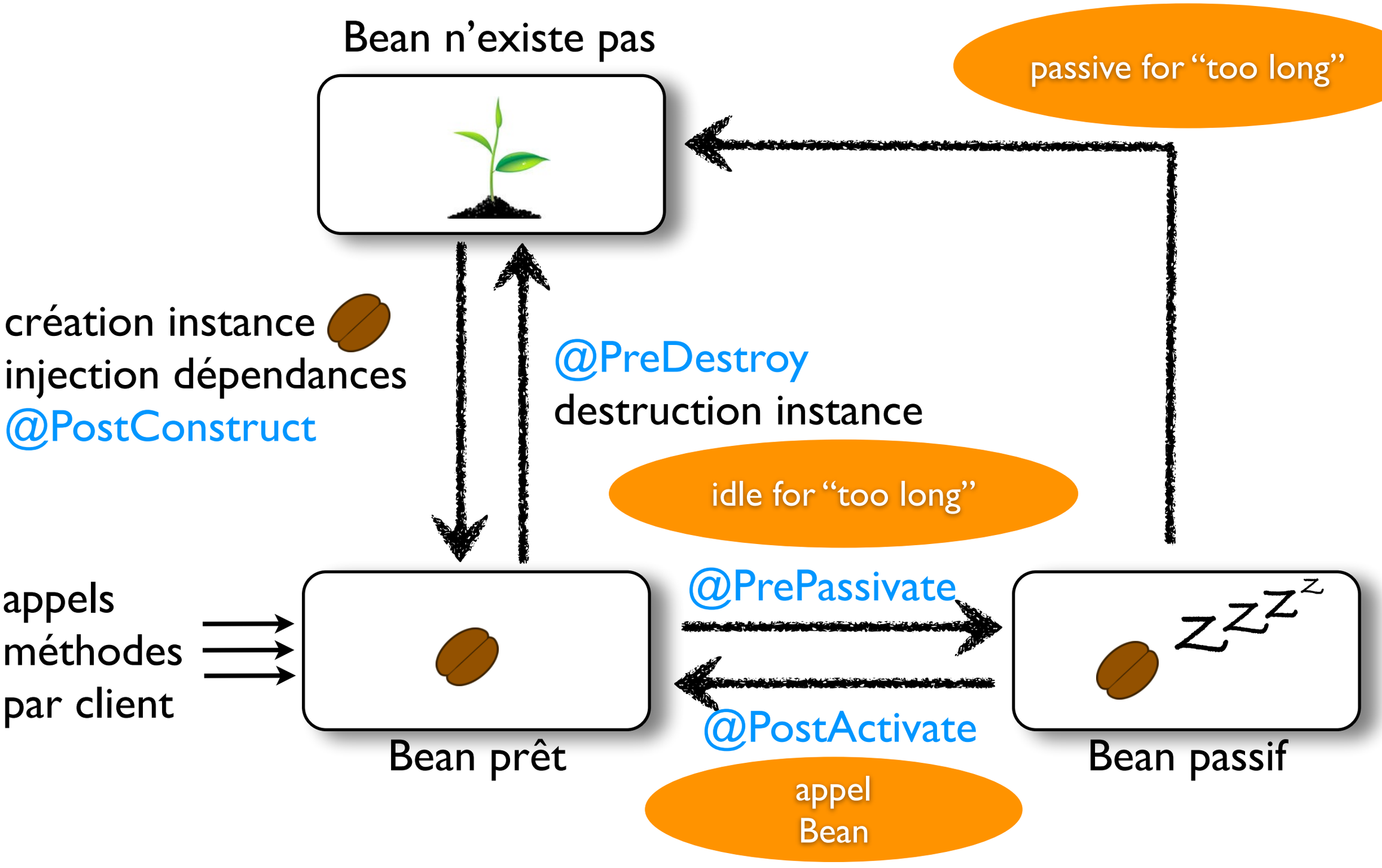
Cycle de vie - Stateful beans



Cycle de vie - Stateful beans



Cycle de vie - Stateful beans



```
@Stateful (name = "InfoManager")
public class InfoManagerBean implements InfoManager {
    private int ticketNumber;
    private String description;
    private String message;

    public void setDescription(String description) {
        this.description = description;
    }

    public void setTicketNumber(int ticketNumber) {
        this.ticketNumber = ticketNumber;
    }

    public String completeSubmission() {
        return message + " " +
               ticketNumber + " : " + description;
    }
}
```

```
@Stateful (name = "InfoManager")
public class InfoManagerBean implements InfoManager {
    private int ticketNumber;
    private String description;
    private String message;

    public void setDescription(String description) {
        this.description = description;
    }

    public void setTicketNumber(int ticketNumber) {
        this.ticketNumber = ticketNumber;
    }

    public String completeSubmission() {
        return message + " " +
               ticketNumber + " : " + description;
    }
}
```

informations

état

```
@Stateful (name = "InfoManager")
public class InfoManagerBean implements InfoManager {
    private int ticketNumber;
    private String description;
    private String message;

    @PostConstruct
    @PostActivate
    public void setMessage() {
       this.message = "Description of the ticket no ";
       System.out.println("This takes some time!");
    }

    @PrePassivate
    @PreDestroy
    public void cleanup() {
       this.message = null;
       System.out.println("Ouf, I feel much lighter!");
    }
```

```
@Stateful (name = "InfoManager")
public class InfoManagerBean implements InfoManager {
    private int ticketNumber;
    private String description;
    private String message;
    }

    @Remove
    public void cancelTicket() {
        this.ticketNumber = -1;
        this.description = null;
    }

    @Remove
    public String completeSubmission() {
        return message + " " +
                ticketNumber + " : " + description;
    }
}
```

```
public class HelloWorldClient {
   @EJB
   private static InfoManager info;

   public static void main(String [] args) {
      try {
         info.setTicketNumber(1);
         info.setDescription("Ca marche bien");
         String result = info.completeSubmission();
         System.out.println(result);
      } catch (Exception ex) {
         ex.printStackTrace();
} } }
```

```
public class HelloWorldClient {
   @EJB
   private static InfoManager info;

   public static void main(String [] args) {
      try {
         info.setTicketNumber(1);
         info.setDescription("Ca marche bien");
         String result = info.completeSubmission();
         System.out.println(result);
      } catch (Exception ex) {
         ex.printStackTrace();
} } }
```

Client

```
Description of the ticket no 1 :
Ca marche bien
```

```
public class HelloWorldClient {
   @EJB
   private static InfoManager info;

   public static void main(String [] args) {
      try {
         info.setTicketNumber(1);
         info.setDescription("Ca marche bien");
         String result = info.completeSubmission();
         System.out.println(result);
      } catch (Exception ex) {
         ex.printStackTrace();
} } }
```

Client

```
Description of the ticket no 1 :
Ca marche bien
```

Server

```
This takes some time!

Ouf, I feel much lighter!
```

```
public class HelloWorldClient {
   @EJB
   private static InfoManager info;

   public static void main(String [] args) {
      try {
         info.setTicketNumber(1);
         info.setDescription("Ca marche bien");
         String result = info.completeSubmission();
         System.out.println(result);
      } catch (Exception ex) {
         ex.printStackTrace();
} } }
```

état préservé pendant la session

Client

```
Description of the ticket no 1 :
Ca marche bien
```

Server

```
This takes some time!

Ouf, I feel much lighter!
```

@EJB interface

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface EJB {
  String name() default "";
  Class beanInterface() default Object.class;
  String beanName() default "";
}
```

nom
(utiliser pour bind JNDI)

si plusieurs Beans pour la même interface

@EJB interface

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface EJB {
  String name() default "";
  Class beanInterface() default Object.class;
  String beanName() default "";
}
```

nom
(utiliser pour bind JNDI)

si plusieurs Beans pour la même interface

- si le nom est omis le container utilise le nom de l'interface pour le dériver

DI et Stateful Beans

- on peut injecter Stateful Beans dans d'autres Stateful Beans
- on ne peut pas injecter Stateful Beans dans des objets sans état (Stateless Bean, Servlet, ...)
- on peut injecter Stateless Beans dans des Stateful Beans

Keypoints

- Choisir avec attention les types des beans
  - statefull → stateless + persistance
- Choisir avec attention les types d'interfaces
- Séparer les objectifs transversaux
  - interceptors
- Choisir avec attention les types de données stockées dans un état conversationnel
- Ne pas injecter Stateful EJB dans des Servlets

Singleton Beans

Singleton Beans

- une seule instance est créée pour la durée de vie de l'application
- permettent
  - des initialisations au démarrage du serveur
  - la création d'une seule instance d'un bean
- similaire au stateless session beans
  - contrôle accès concurrents
  - enchainement d'initialisations de singletons

Utilisation Singletons

- taches au démarrage
  - vérification consistance BDD, vérifications systèmes externes (ex: LDAP), ...
- point de distribution
  - cache et contrôle valeurs spéciales, ...
- anti-pattern
  - services stateless et/ou qui n'utilisent pas le container

Médecin de garde

```
@Singleton
@Startup
@DependsOn("MiseEnPlaceSingleton")
public class MedecinGardeSingleton {
    private String medecin;
    @PostConstruct
    public void init() {
         loadMedecinGarde();
    }
    @Schedule(...)
    private void loadMedecinGarde() { ... medecin = ...}

    public String getMedecinGarde() {
        return medecin;
    }
}
```

Médecin de garde

singleton

démarrage au startup

dépendances

```
@Singleton
@Startup
@DependsOn("MiseEnPlaceSingleton")
public class MedecinGardeSingleton {
    private String medecin;
    @PostConstruct
    public void init() {
        loadMedecinGarde();
    }
    @Schedule(...)
    private void loadMedecinGarde() { ... medecin = ...}

    public String getMedecinGarde() {
        return medecin;
    }
}
```

tous les jours

Singleton interface

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Singleton {
    String name() default "";
    String mappedName() default "";
    String description() default "";
}
```

Singleton interface

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Singleton {
    String name() default "";
    String mappedName() default "";
    String description() default "";
}
```

- la même que `Stateless`

Cycle de vie - Singleton beans

Bean n’existe pas

Cycle de vie - Singleton beans

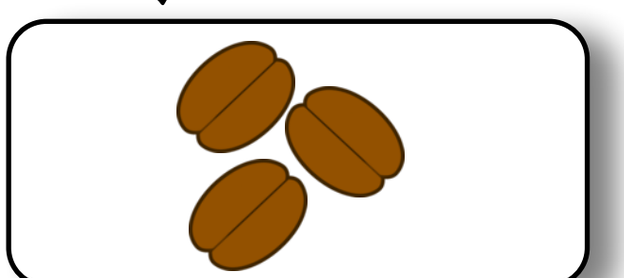
- lancement application & **Startup**
- lancement bean dépendant & **DependsOn**
- paresseuse (premier accès)

Bean n'existe pas

création instance

injection dépendances

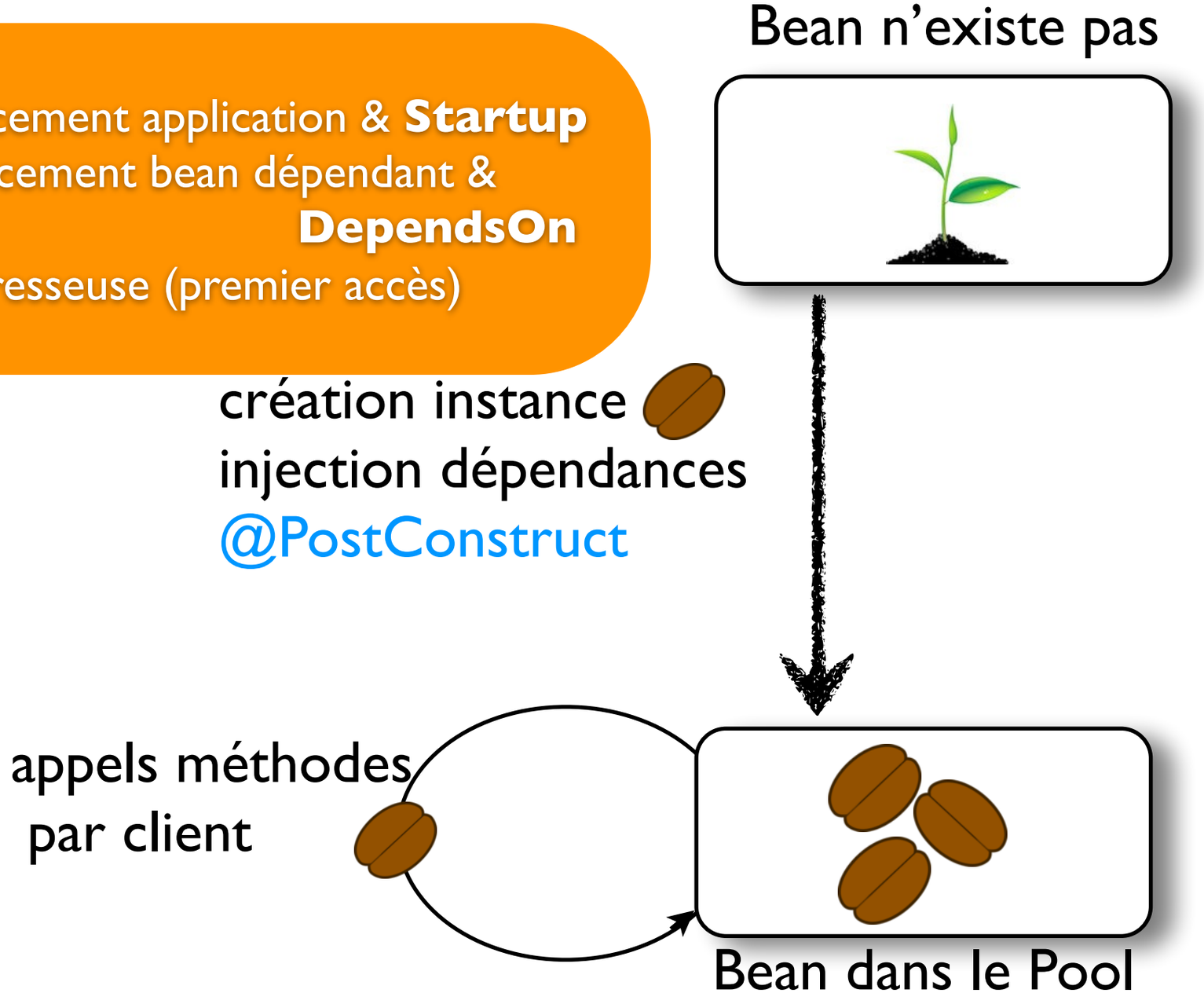
@PostConstruct



Bean dans le Pool

Cycle de vie - Singleton beans

- lancement application & **Startup**
- lancement bean dépendant & **DependsOn**
- paresseuse (premier accès)



Cycle de vie - Singleton beans

- lancement application & **Startup**
- lancement bean dépendant & **DependsOn**
- paresseuse (premier accès)

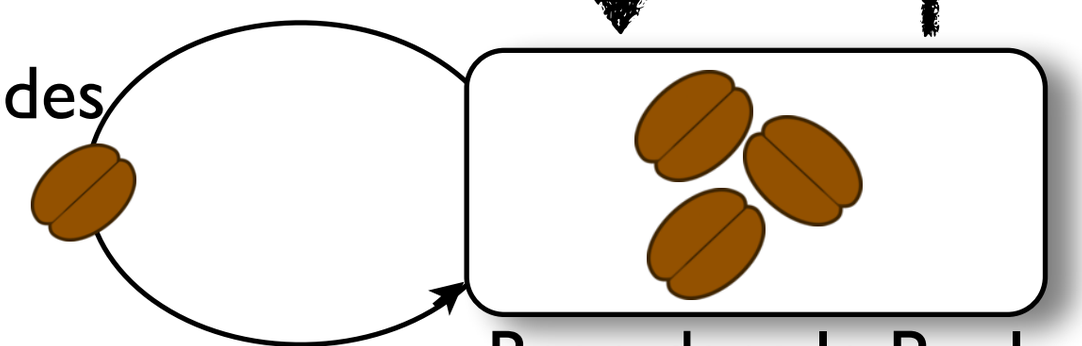
Bean n'existe pas

shutdown application

création instance
injection dépendances
@PostConstruct

@PreDestroy
garbage collection

appels méthodes
par client



Bean dans le Pool

Concurrence

- gérée par le container
  - par défaut
  - `@ConcurrencyManagement( ConcurrencyManagementType.CONTAINER)`
- gérée par le bean
  - utilisation explicite des techniques Java (`synchronized`/`volatile`)
  - `@ConcurrencyManagement( ConcurrencyManagementType.BEAN)`

Concurrence container

`(ConcurrencyManagementType.CONTAINER)`

- `@Lock(LockType.READ)`
  - la méthode peut être utilisée par plusieurs clients en même temps si pas de lock en write
- `@Lock(LockType.WRITE)`
  - placer un lock write quand la méthode est appelée
  - valeur par défaut

Time-out

```
@Target(value = {ElementType.METHOD, ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface AccessTimeout {
  public long value();
  public TimeUnit unit() default TimeUnit.MILLISECONDS;
}
```

Time-out

- on dispose d'un mécanisme de time-out pour éviter le blocage des client

```
@Target(value = {ElementType.METHOD, ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface AccessTimeout {
  public long value();
  public TimeUnit unit() default TimeUnit.MILLISECONDS;
}
```

- une exception `ConcurrentAccessTimeoutException` est levée si pas de réponse après `value units`

Médecin de garde

```
@Local
public interface MedecinGarde {
    @Lock(LockType.READ)
    @AccessTimeout(value = 2, unit = TimeUnit.SECONDS)
    String getMedecinGarde();
}
```

sur le bean ou l'interface

sur la classe ou la méthode

Utilisation Singletons

- attention aux bottlenecks possibles
- choisir le bon type de concurrence
- configurer la concurrence
  - utiliser `LockType.READ` si possible
  - utiliser `@AccessTimeout` pour éviter les attentes trop longues

Questions ?