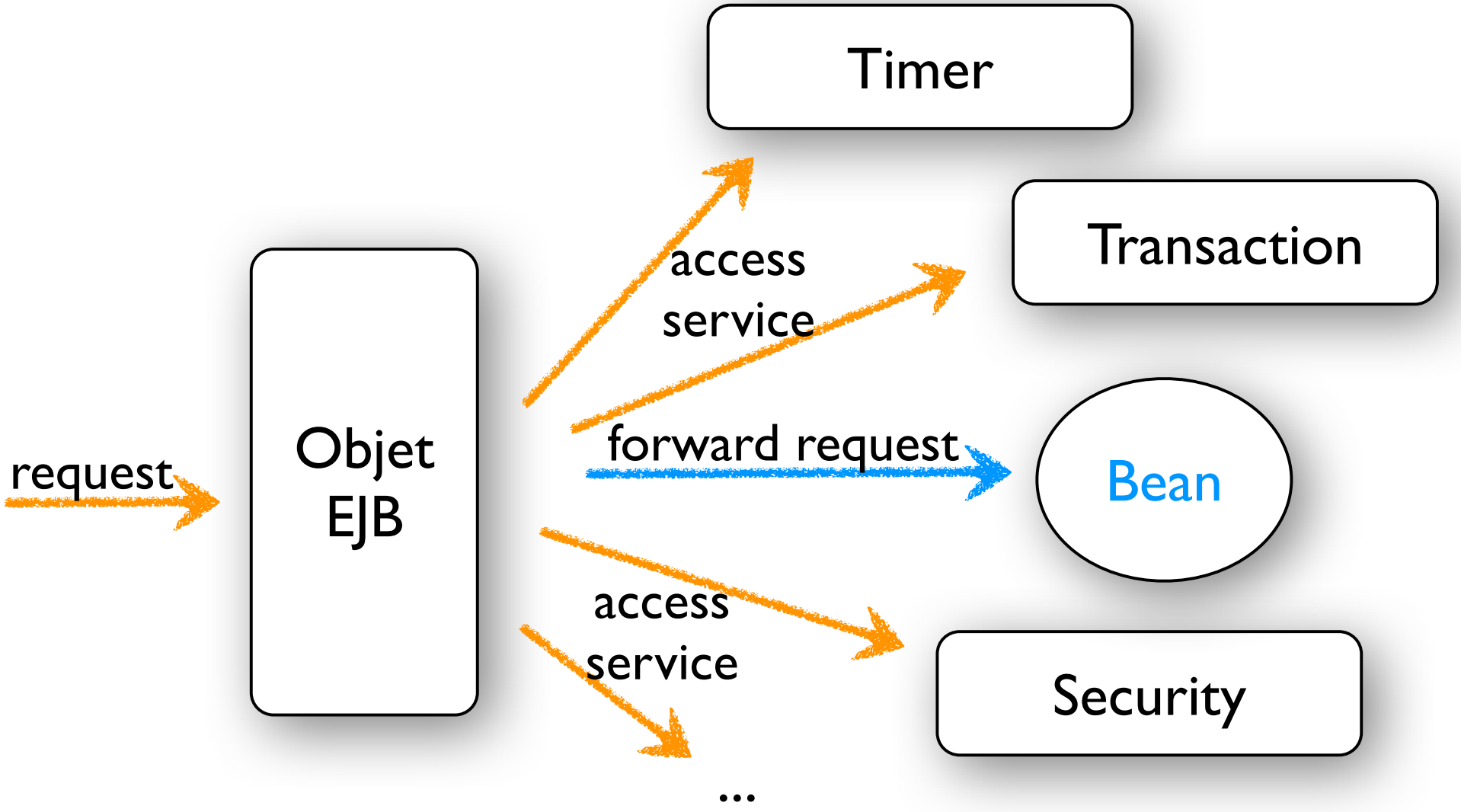
Beans
- Aspects avancés -

Objet EJB


Timer
Transaction
access
service
request
Objet
EJB
forward request
Bean
access
service
Security
...

Accéder a l'environnement

- en général, les EJBs n’accèdent pas au container (et aux services fournis) directement
- la porte d’accès

```
javax.ejb.EJBContext
javax.ejb.SessionContext
javax.ejb.MessageDrivenContext
```

- permet un accès programmé aux services

EJBContext

```
public interface EJBContext {
  public Principal getCallerPrincipal();
  public boolean isCallerInRole(String roleName);
  public EJBHome getEJBHome();
  public EJBLocalHome getEJBLocalHome();

  public boolean getRollbackOnly();
  public UserTransaction getUserTransaction();
  public void setRollbackOnly();

  public TimerService getTimerService();

  public Object lookup(String name);
}
```

sécurité

transactions

timer

JNDI lookup

Utiliser (les sous-classes) EJBContext

- Session Bean

```
@Stateless
public class HelloWorldBean implements HelloWorld {
   @Resource
   SessionContext context;
   ...
}
```

- MDB

```
@MessageDriven
public class TicketMDB {
   @Resource
   MessageDrivenContext context;
   ...
}
```

Timers

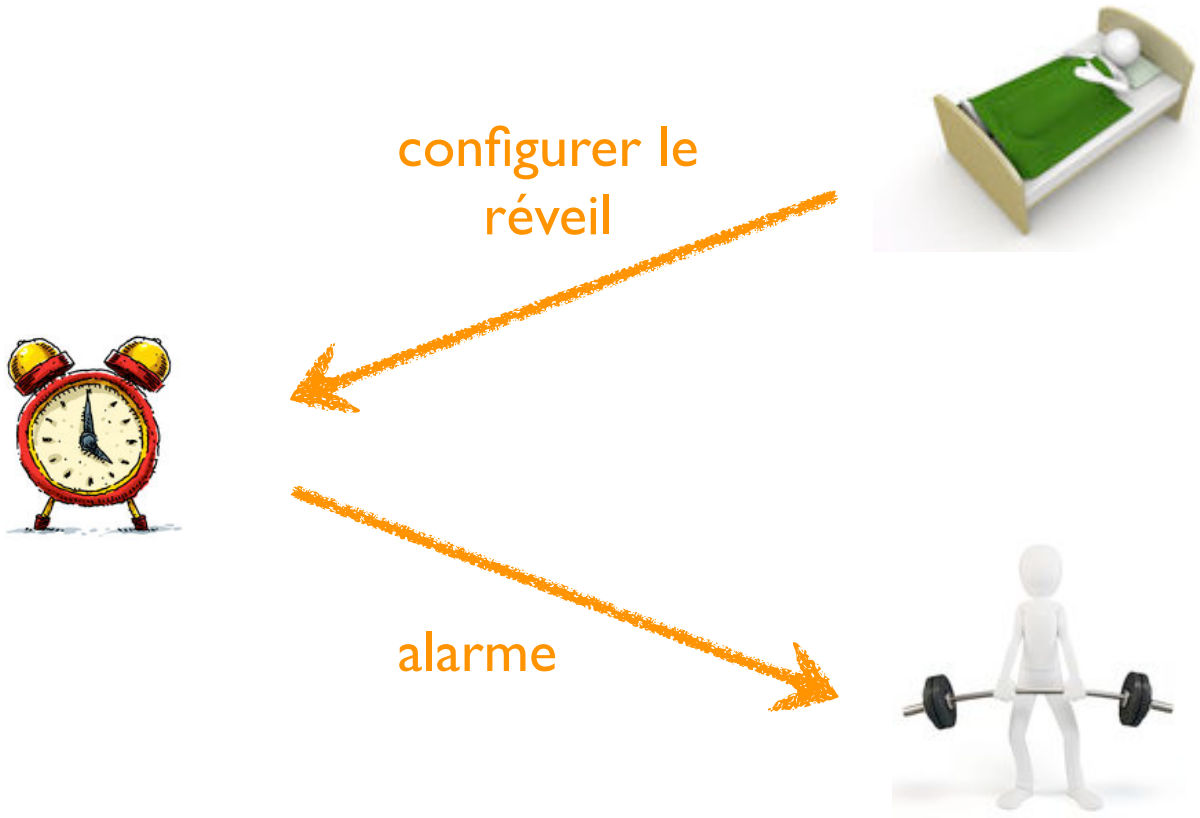
Scheduling

- déclencher des actions à des instants particuliers
  - batch processing (ex: génération rapports)
  - maintenance (ex: nettoyage table temporaires)
  - date limites (ex: pour des enchères)
- utilitaires de scheduling dans les serveurs Java EE ou packages dédiés (Flux, Quartz, ...)

EJB Timer service

- **appels automatique des méthodes** (timeout methods) **après un certain intervalle de temps**
- **utilisable dans stateless beans et MDB**
- **les timers sont *persistants* :** ils survivent au crash et redémarrages des serveurs
- **les timers sont *transactionnels* :** un échec de la transaction conduit à un rollback des actions effectuées par le timer

Fonctionnement

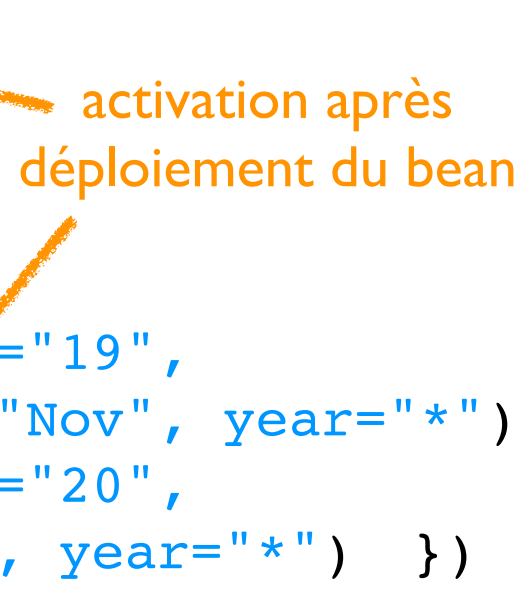


Timer

- **types**
  - *time-delayed* timers
  - *calendar-based* timers
- **création/configuration**
  - déclarative (annotations, fichiers configuration)
  - programmation

Timer *calendar-based* déclaratif

```
@Stateless
public class MealsBean {
    @Schedule(second="0", minute="0", hour="12",
              dayOfMonth="*", month="*", year="*")
    public void sendLunchRecall() {
        System.out.println("Lunch time !");
    }

    @Schedules({
      @Schedule(second="0", minute="0", hour="19",
                dayOfMonth="4th Thu", month="Nov", year="*"),
      @Schedule(second="0", minute="0", hour="20",
                dayOfMonth="25", month="Dec", year="*")  })
      public void sendSpecialLunch() {
           System.out.println("Festive Lunch !!!!");
      }
}
```



Interface Schedule(s)

```
@Target(value=METHOD)
@Retention(value=RUNTIME)
public @interface Schedule {
   String dayOfMonth() default "*";
   String dayOfWeek() default "*";
   String hour() default "0";
   String info() default "";
   String minute() default "0";
   String month() default "*";
   boolean persistent() default true;
   String second() default "0";
   String timezone() default "";
   String year() default "*";
}
```

```
@Target({METHOD})
@Retention(RUNTIME)
public @interface Schedules {
  javax.ejb.Schedule[] value();
}
```

survivre restarts serveur

- applicable à des méthodes

```
void <METHOD>()
void <METHOD>(Timer timer)
```

pas de correspondance entre timer et bean

Syntaxe @Schedule

- Valeur

```
@Schedule(dayOfMonth="1", month="Apr")
```

- Wildcard

```
@Schedule(second="*", minute="*")
```

- Liste

```
@Schedule(second="0", minute="0,30")
```

- Domaine

```
@Schedule(hour="0-6",dayOfWeek="Mon-Fri")
```

- Incréments

```
@Schedule(minute="30/10", hour="*/6")
```

Programmation Timer *time-delayed*

injection Timer service

création Timer

méthode exécutée au timeout

info attachée au timer

```
@Stateless
public class HelloWorldBean implements HelloWorld  {
    @Resource
    TimerService timerService;

   public void sayHello(String name) {
        System.out.println("Hello "+name);
        Timer timer = timerService.createTimer(1*60*1000, name);
   }

   ...
   @Timeout
   public void sayAgain(Timer timer) {
       System.out.println("Hello " + timer.getInfo());
       System.out.println("Recall at " + timer.getNextTimeout());
       return;
   }
}
```

Accéder le TimerService

- par injection

```
@Resource
TimerService timerService;
```

- via le contexte EJB

```
@Resource
SessionContext sc;
...
TimerService timerService = sc.getTimerService();
```

création d'un timer déclenché une fois

création d'un timer récurent

création d'un timer récurent

rice

```
public interface javax.ejb.TimerService {
    Timer createTimer(long duration,Serializable info);
    Timer createTimer(long initialDuration, long intervalDuration,
                      java.io.Serializable info);
    Timer createTimer(Date expiration, Serializable info);
    Timer createTimer(Date initialExpiration, long intervalDuration,
                      java.io.Serializable info);
    Timer createSingleActionTimer(long duration, TimerConfig conf);
    ...
    Timer createIntervalTimer(long initialDuration, long interval,
                              TimerConfig conf)
    ...
    Timer  createCalendarTimer(ScheduleExpression schedule)
    Timer  createCalendarTimer(ScheduleExpression schedule,
                               TimerConfig timerConfig)

    public Collection getTimers();
}
```

configuration additionnelle timer

Timers associés à l'EJB courant

Méthodes `@Timeout`

- appelées quand le timer créer pour le bean expire
- un bean peut avoir au plus une méthode timeout
- format

```
void <METHOD>(Timer timer)
```

- **`timer`** : timer pour lequel la méthode a été invoquée (plusieurs timers peuvent invoquer la même méthode) → on peut donc obtenir les informations transmises à la création du timer

Interface `Timer`

```
public interface javax.ejb.Timer {

    void cancel();

    long getTimeRemaining();

    java.util.Date getNextTimeout();

    java.io.Serializable getInfo();

    ScheduleExpression getSchedule();

    boolean isCalendarTimer();

    boolean isPersistent();
    ...
}
```

résilier le timer et les notifications associées

le temps restant jusqu'à la notification suivante

la date de la notification suivante

information associée à la création

Résumé Timers

- dans la spécification EJB → portabilité
- pas d'installation ou configuration supplémentaires
- persistants (après crash et restart)
- timers déclaratifs → *paramètres connus a priori*
- timers ad-hoc → *paramètres dynamiques*
- *pas la meilleure précision* → pour des processus long et non pas pour du temps réel
- *pas de GUI*

Interceptors

Problème

- ajouter des fonctionnalités manquantes dans une application (finalisée ou en cours de finalisation)
- l'implantation des fonctionnalités implique l'ajout du (même) code correspondant dans plusieurs modules
- **exemples :** *logging, auditing, profiling, statistics, ...*
  - *crosscuting concerns* (préoccupations transverses)

Aspect oriented programming

- **séparer les *préoccupations métier* et les *préoccupations transverses***
  - identification et implantations de préoccupations transverses dans des modules dédiés
  - application de ces modules aux (sections de) modules métier concernés
- **outils : AspectJ, JBoss, ...**

JEE interceptors

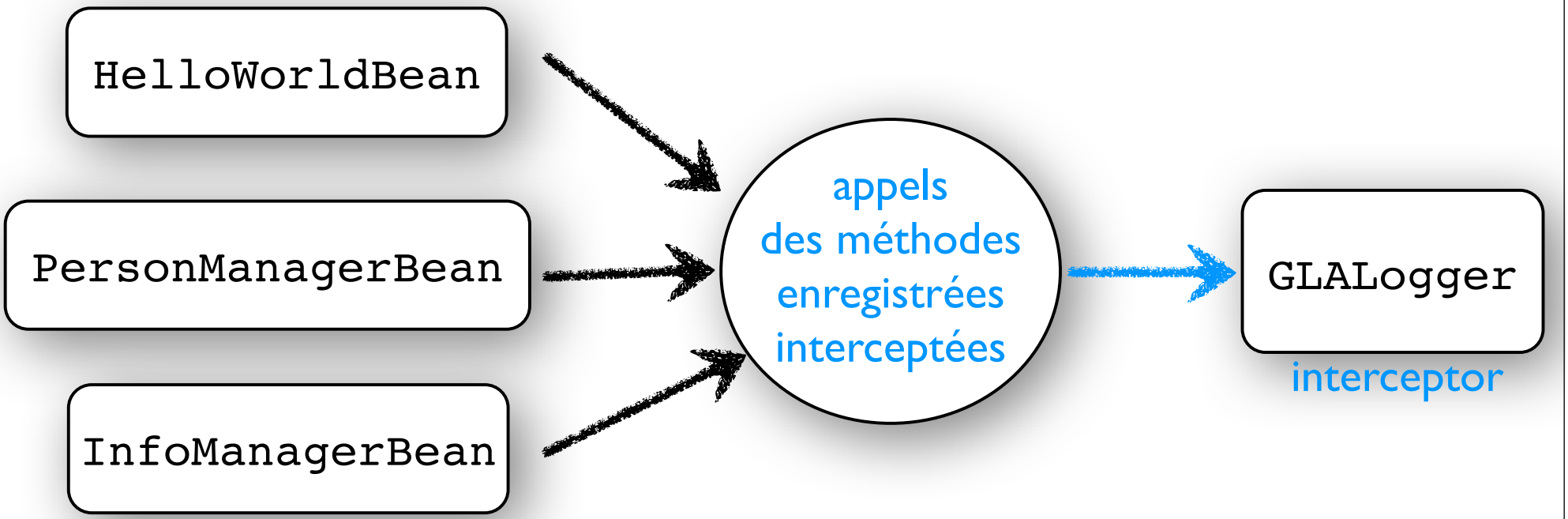
- objets déclenchés automatiquement quand une méthode EJB est appelée
- interception de type *around invoke advice*
  - déclenchés au début de la méthode
  - peuvent inspecter la valeur de retour
  - peuvent intercepter les éventuelles exceptions

Utilisations habituelles

- **avant l'exécution de la méthode**
  - auditing
  - (+) vérification paramètres avant exécution
  - (+) traitement résultat avant retour
- **interception exceptions**
  - auditing
  - (+) re-appel de la méthode ou autres chemins d'exécution

Business interceptors

- les intercepteurs pour la logique métier utilisés généralement pour implanter du code commun



Exemple

```
public class ApplicationLogger {
    @AroundInvoke
    public Object logMethodCall(InvocationContext ic)
      throws Exception {
        System.out.println("Call method "
                + ic.getMethod().getName());
        return ic.proceed();          }
    }
}
```

spécifier la méthode d'interception

```
@Stateless
public class HelloWorldBean implements HelloWorld  {
    @Interceptors({ buslogic.ApplicationLogger.class })
    @Override
    public void sayHello(String name) {
        System.out.println("Hello "+name);
    }
...
}
```

attacher l'intercepteur

Spécifier les intercepteurs

- un intercepteur pour une *méthode* ou une ***classe***

```
@Interceptors({ buslogic.ApplicationLogger.class })
@Stateless
public class HelloWorldBean implements HelloWorld  {
   public void sayHello(String name) {...   }
...
}
```

- ou ***plusieurs*** intercepteurs

```
@Interceptors({ buslogic.ApplicationLogger.class,
                buslogic.ApplicationTracker.class})
@Stateless
public class HelloWorldBean implements HelloWorld  {
   ...
}
```

Intercepteur par défaut

```
<ejb-jar>

    <interceptors>
        <interceptor>
            <interceptor-class>buslogic.ProfilingInterceptor
            </interceptor-class>
        </interceptor>
    </interceptors>

    <assembly-descriptor>
        <interceptor-binding>
            <ejb-name>*</ejb-name>
            <interceptor-class>
                buslogic.ProfilingInterceptor
            </interceptor-class>
        </interceptor-binding>
    </assembly-descriptor>

</ejb-jar>
```

Intercepteur spécifique

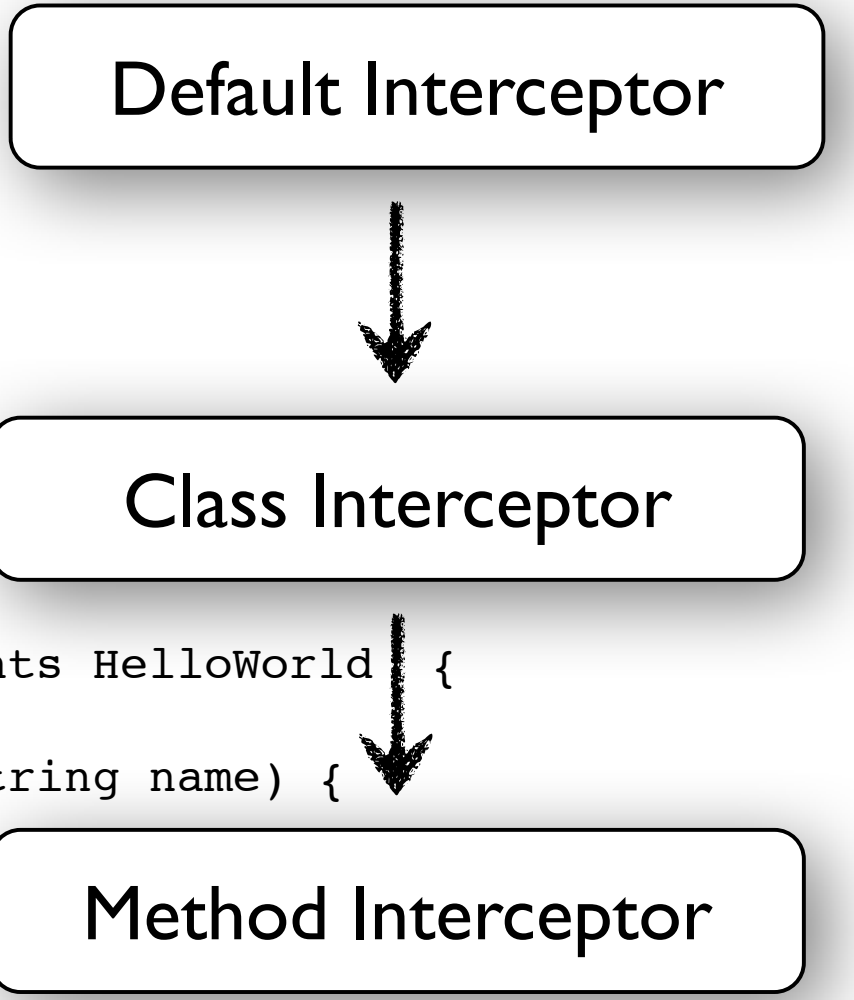
```
<ejb-jar>

    <interceptors>
        <interceptor>
            <interceptor-class>buslogic.ProfilingInterceptor
            </interceptor-class>
        </interceptor>
    </interceptors>

    <assembly-descriptor>
        <interceptor-binding>
            <ejb-name>HelloWorldBean</ejb-name>
            <interceptor-class>
                buslogic.ProfilingInterceptor
            </interceptor-class>
        </interceptor-binding>
    </assembly-descriptor>

</ejb-jar>
```

Ordre d'application

- **plusieurs intercepteurs à un niveau** ➙ appliquer dans l'ordre de spécification

- **exclure des intercepteurs**

```
@Stateless
@Interceptors({ buslogic.Application
@ExcludeDefaultInterceptors
public class HelloWorldBean implements HelloWorld {
    @ExcludeClassInterceptors
    public String greetingsMessage(String name) {
        ...
    }
...
}
```



Méthodes `AroundInvoke`

- peuvent être implantées dans le bean ou dans une classe séparée
- chaque intercepteur doit avoir une seule méthode `AroundInvoke`
- ne doivent pas être des méthodes métier, c.a.d. pas de méthode public dans l'interface du bean
- pattern:

```
public Object <METHOD>(InvocationContext ic)
                                    throws Exception
```

Interface `InvocationContext`

```
public interface InvocationContext {
  Object getTarget();

  Method getMethod();

  Object[] getParameters();

  void setParameters(Object[]);

  Map<String,Object> getContextData();

  Object proceed() throws Exception;
}
```

l'instance du bean propriétaire de la méthode interceptée

méthode pour laquelle l'intercepteur a été déclenché

ses paramètres

changer les paramètres avant l'appel

contexte partagé dans la chaîne des intercepteurs

continuer avec l'intercepteur suivant dans la chaîne ou avec la méthode interceptée

Exceptions

- si la méthode `proceed` n'est pas appelée, les autres intercepteurs dans la chaîne et la méthode métier ne seront pas appelés
- exemple : vérifications sécurité

```
@AroundInvoke
public Object securityCheck(InvocationContext ic)
  throws Exception {
    if (...) { // security check
      throw new AuthorisationException("Not authorised");
    }

    return ic.proceed();
}
```

Context data

class interceptor

```
public class VerifyInterceptor {
  @AroundInvoke
  public Object securityCheck(InvocationContext ic)
                                          throws Exception{
      ic.getContextData().put("Verified", "OK");
      return ic.proceed();
  }
}
```

class interceptor

```
public class VerifyInterceptor {
  @AroundInvoke
  public Object securityCheck(InvocationContext ic) throws Exception{
      ic.getContextData().put("Verified", "OK");
      return ic.proceed();
  }
}
```

method interceptor

```
public class ProfilingInterceptor {
 @AroundInvoke
 public Object profile(InvocationContext ic) throws Exception {
   System.out.println("Method " + ic.getMethod() + " started");
   try {
     if (((String)ic.getContextData().get("Verified")).equals("OK")){
                Object[] param = ic.getParameters();
                param[0] = ((String) param[0]).toUpperCase();
                ic.setParameters(param);
     }
     return ic.proceed();
   } finally {
     System.out.println("Method " + ic.getMethod() + " finished");
   }
 }
}
```

Lifecycle callbacks interceptors

- mêmes (comportements) que pour les beans : `@PostConstruct, @PrePassivate, @PostActivate, @PreDestroy`

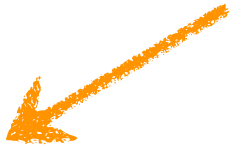
```
public class RessourceLogger {
  @PostConstruct
  public void setMessage(InvocationContext ic) {
       System.out.println("This takes some time!");
       ic.proceed();
  }

  @PreDestroy
  public void cleanup(InvocationContext ic) {
       System.out.println("Ouf, I feel much lighter!");
       ic.proceed();
  }
}
```

CDI Interceptors

- **propose « type-safe interceptor bindings » qui peuvent être combinés pour décrire des comportements (plus) complexes**
  - déclaration (Interceptor) Binding
  - création Interceptor pour (Interceptor) Binding
  - lier Interceptor au Binding

Déclaration Binding

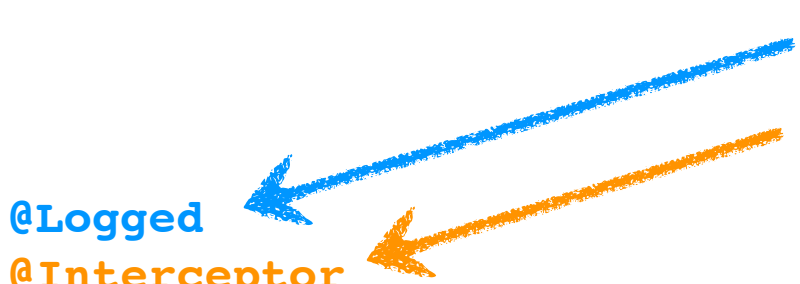
peut être déclaré seulement au niveau de la classe/méthode



```
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Logged {}
```

le nom reflète l'objectif du binding

Création Interceptor



pas besoin de spécifier les annotations avec une approche Interceptor EJB

```
@Logged
@Interceptor
public class LogInterceptor {
    @AroundInvoke
    public Object logMethodCall(InvocationContext ic)
      throws Exception {
        System.out.println("Call method "
                + ic.getMethod().getName());
        return ic.proceed();          }
    }
}
```

- tous les *Interceptors* nécessite un binding

Lier Interceptor au Binding

- au niveau des classes

```
@Stateless
@Logged
public class HelloWorldBean implements HelloWorld  {
   public void sayHello(String name) {...   }
...
}
```

- au niveau des méthodes

```
@Stateless
public class HelloWorldBean implements HelloWorld  {
   @Logged
   public void sayHello(String name) {...   }
...
}
```

Lister les Interceptors

- déclarer tous les intercepteurs à activer dans beans.xml

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="...”>
  <interceptors>
     <class> buslogic.LogInterceptor </class>
  </interceptors>
</beans>
```

Extra power

- binding multiples

```
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Logged
public @interface Secured {}
```

  - les Interceptors liés à `Logged` et à `Secured` s'appliquent à tout bean annoté avec `Secured`

- liaisons multiples

```
@Logged @Profiled
@Interceptor
public class LogProfileInterceptor {
    @AroundInvoke
    public Object logMethodCall(InvocationContext ic) ...
}}
```

  - l'Interceptor s'applique à toute méthode concernée par `Profiled` et `Logged` simultanément

Ordre d'application

- intercepteurs déclarés avec `Interceptors`
- intercepteurs dans ejb-jar.xml
- intercepteurs CDI dans beans.xml

Transactions

Transactions

- concept fondamental dans les applications distribuées
- concerne des aspects transverses (niveau système) plus que les aspects métier
- un ensemble de tâches qui doivent être exécutés de manière atomique
  - une transaction réussie est ***committed*** (les résultats sont rendus permanents)
  - une transaction échouée est ***rolled back*** (comme si rien ne s'était passé)

Vente en ligne

valider commande

Traitement commande

débiter carte

demande livraison

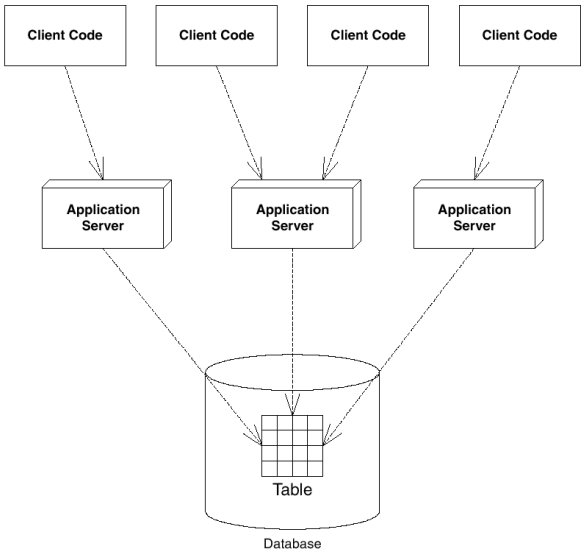
problème livraison

notification echec commande

envoi facture

Concurrence d'accès

- **Plusieurs clients consultent et modifient les mêmes données…**
  - Comment garantir l'intégrité des données ?
  - Comment garantir des performances acceptables ?



Propriétés : ACID

- **Atomicity**
  - toutes les actions d'une transaction réussissent (et les résultats sont permanents) ou échouent (et il n'y a pas d'effet sur le système)
- **Consistency**
  - le système demeure consistant (par rapport aux règles métier) après l'exécution (avec succès ou non) d'une transaction (si consistant avant)
  - responsabilité du développeur

Propriétés - ACID

- **Isolation**
  - les transactions concurrentes ne se marchent pas sur les pieds : chaque transaction est isolée et ne vois pas des résultats partiels (d'autres transactions)
  - le degré d'isolation a un impact direct sur les performances
- **Durability**
  - les mises à jour sur une BD peuvent survivre à un crash (BD, machine, réseau)
  - utilisation de fichiers de log qui permet de revenir dans l'état avant le crash

Violations/Niveaux isolation

Violations possibles:

- «dirty reads», «non-repeatable reads», «phantoms»

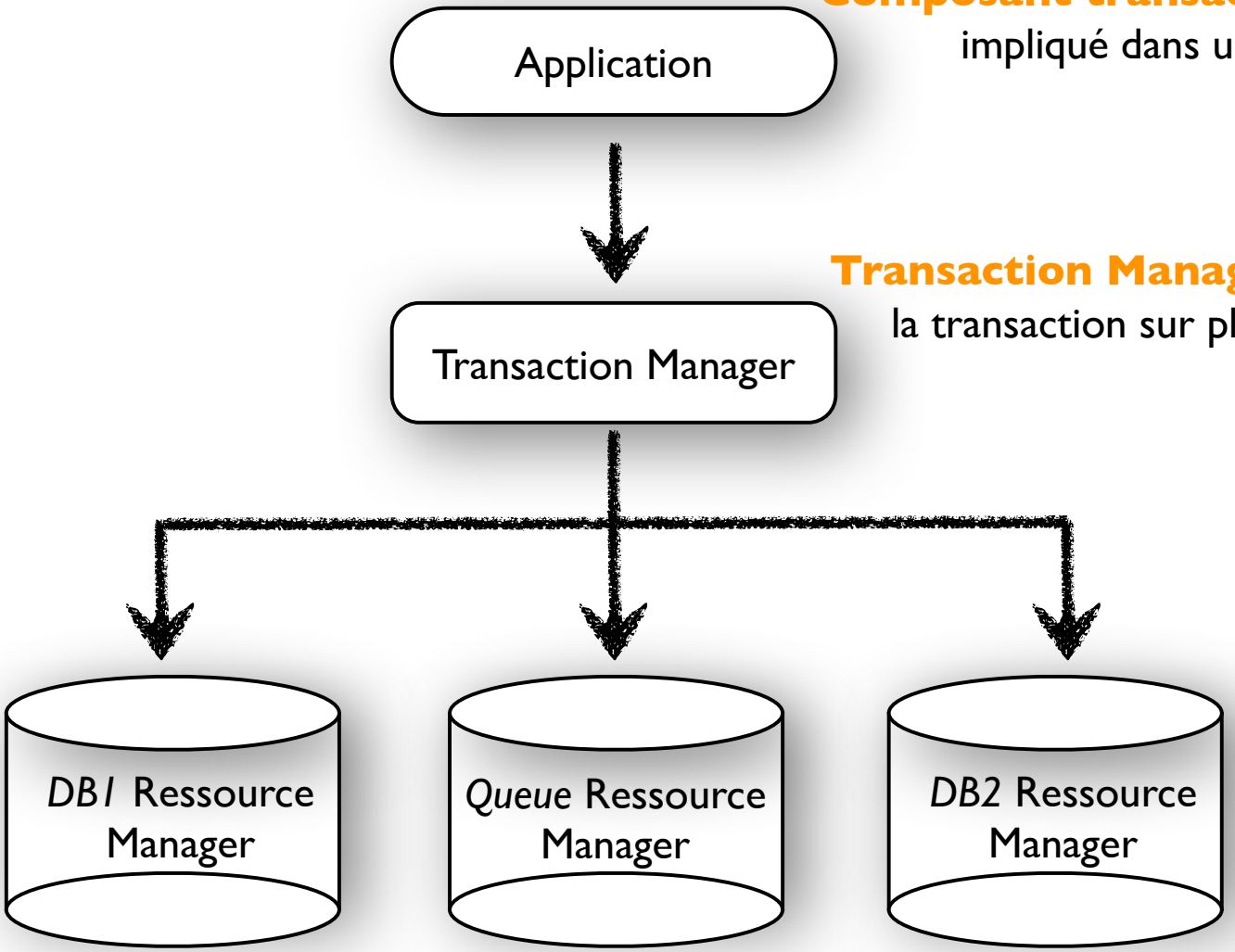
- READ UNCOMMITED
  - la transaction peut voire des données non-commitées
- READ COMMITED
  - la transaction ne lit que des résultats commités
  - évite «dirty reads»

Niveaux isolation

- REPEATABLE READ
  - la transaction ne voit pas les modifications (sur les données lues) faites dans d'autres transactions
  - on obtient les mêmes données à chaque lecture
  - évite «dirty reads», «non-repeatable reads»
- SERIALIZABLE
  - aucune table ne change pendant la transaction
  - évite «dirty reads», «non-repeatable reads», «phantoms»

Transaction management

Composant transactionnel : composant impliqué dans une transaction



Transaction Manager : gère (en coulisse) la transaction sur plusieurs ressources

Ressource Manager : gère les transaction pour une ressource particulière

Modèles de transactions

- Flat
- Nested
- Distributed
- ...

Transactions et EJB

- **Container-managed transactions (CMT)**
  - le container se charge de tout
  - approche déclarative (annotations et descripteur)
- **Bean-managed transactions (BMT)**
  - transaction gérée par les beans
  - code dans les beans
- ***Client Controlled***
  - *transactions programmées du coté client*

Container-managed transactions

- le container lance et exécute le commit et le roll-back
  - lance une transaction JTA avant l'appel
  - appelle la méthode
  - exécute commit/roll-back
- il faut préciser comment gérer les transaction avec des annotations ou des deployement descriptor
- il faut préciser les roll-backs si nécessaire

Exemple

utiliser CMT

définir les attributs de transaction pour la méthode

actions de la transaction

roll-back en cas d'erreur

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class OrderManagerBean {
  @Resource
  private SessionContext context;
  ...
  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void orderItem(Person customer, Item item){
    try {
      if (available(item)){
          chargeCreditCard(customer, item);
          sendOrder(item);
      }
    } catch (CreditValidationException cve) {
      context.setRollbackOnly();
    }
  }
}
```

@TransactionManagement

- permet de spécifier CMT ou BMT
  - `TransactionManagementType.CONTAINER`
  - `TransactionManagementType.BEAN`
- par défaut, CMT

@TransactionAttribute

- utilisé pour préciser au container comment gérer (l'enchainement) des transactions
- on peut spécifier des attributs pour le bean entier ou méthode par méthode,
  - le plus restrictif est appliqué
- chaque méthode métier doit être traitée (globalement ou individuellement)
- `REQUIRED`, `REQUIRES_NEW`, `SUPPORTS`, `MANDATORY`, `NOT_SUPPORTED`, `NEVER`

`Required`

- le bean est toujours dans une transaction
- si une transaction pour ce bean existe, alors le bean la rejoint, sinon, le container crée une nouvelle transaction
- un roll-back éventuel est réalisé par rapport à la transaction entière et `RollbackException` est levée
- utilisé quand la méthode modifie des données et on ne sais pas si le client va démarrer une transaction avant l'appel

RequiresNew

- le bean est toujours dans une nouvelle transaction
- si une transaction existe, elle est suspendue
- lorsque la nouvelle transaction se termine (abort ou commit), l'ancienne transaction est résumée
- utilisé quand on veut respecter les propriétés ACID dans l'unité du bean sans faire intervenir une logique externe

Supports

- **le bean hérite l'environnement transactionnel du client**
- utilisé pour des méthodes qui accèdent des données seulement en mode lecture

Mandatory

- une transaction doit exister avant d'appeler la méthode du bean
- `javax.ejb.TransactionRequiredException` est levée sinon
- utilisé quand on veut être sur que le client échoue si on demande un roll-back

NotSupported

- le Bean ne supporte pas les transactions
- si une transaction existe lors de l'appel du bean, la transaction est suspendue, le bean s'exécute, puis la transaction est résumée
- utilisé typiquement pour des MDB et un provider JMS en mode non-transactionnel, `AUTO_ACKNOWLEDGE`

Never

- **le Bean ne supporte pas les transactions,**
- **une exception `javax.ejb.EJBException` est levée si une transaction existe au moment de l'appel du bean**
- utilisé lors du développement d'une logique non transactionnelle (ex : modification d'un fichier texte) quand on veut être sur que le client est au courant de cette nature non-transactionnelle

Effets des attributs

Attribut	Transaction client	Transaction bean	Effet
REQUIRED	-	T2	nouvelle transaction
	T1	T1	rejoint T1
REQUIRES_NEW	-	T2	nouvelle transaction
	T1	T2	T1 suspendue
SUPPORTS	-	-	pas de transaction
	T1	T1	rejoint T1
MANDATORY	-	Erreur	EJBTransactionRequiredException
	T1	T1	rejoint T1
NOT_SUPPORTED	-	-	pas de transaction
	T1	-	T1 suspendue
NEVER	-	-	pas de transaction
	T1	Erreur	EJBException

Roll-back

- sous certaines conditions une méthode CMT peut demander au container d'effectuer un roll-back sur une transaction (dès que possible)

```
context.setRollbackOnly()
```

- le container vérifie à la fin de la méthode si le flag a été positionné
- il faut s'assurer qu'une transaction existe avant d'appeler `setRollbackOnly()` et `getRollbackOnly()`
  - `java.lang.IllegalStateException`

Traitement exceptions

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void orderItem(Person customer, Item item){
  try {
      if (available(item)){
         chargeCreditCard(customer, item);
         sendOrder(item);
      }
  } catch (CreditValidationException cve) {
    context.setRollbackOnly();
  } catch (CreditProcessingException cpe) {
    context.setRollbackOnly();
  } catch (DatabaseException dbe) {
    context.setRollbackOnly();
  }
}
```

Traitement exceptions

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void orderItem(Person customer, Item item)
    throws CreditValidationException, CreditProcessingException,
           DatabaseException{
      if (available(item)){
           chargeCreditCard(customer, item);
           sendOrder(item);
      }
}
```

roll-back avant de la passer au client

spécifier exceptions application

```
@ApplicationException(rollback=true)
public class CreditValidationException extends Exception { ... }

@ApplicationException(rollback=true)
public class CreditProcessingException extends Exception {... }

@ApplicationException(rollback=false)
public class DatabaseException extends RuntimeException {... }
```

default

Avantages CMT

- **le container s'occupe de tout**
  - les limites de la transaction sont données par le début et la fin d'une méthode
  - le container décide du lancement, du commit et du roll-back
- *pas de flexibilité*

Bean-managed transactions

- spécification explicite (dans le code du bean) des transactions
- granularité plus fine qu'en CMT
- réaliser en utilisant Java Transaction API (JTA)
  - JTA utilisé essentiellement par les développeurs d'applications
  - Java Transaction Service (JTS) utilisé essentiellement par les développeurs d'outils capables d'assurer un service de transaction

Exemple

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class OrderManagerBean {
  @Resource
  private UserTransaction userTransaction;
  ...
  public void orderItem(Person customer, Item item){
    try {
      userTransaction.begin();
      if (available(item)){
          chargeCreditCard(customer, item);
          sendOrder(item);
      }
      userTransaction.commit();
    } catch (CreditValidationException cve) {
      userTransaction.rollback();
    ...
    } catch (Exception e) {
      userTransaction.rollback();
    }
} }
```

utiliser BMT

encapsule les fonctionnalités de base du manager de transactions JEE

start transaction

actions de la transaction

commit transaction

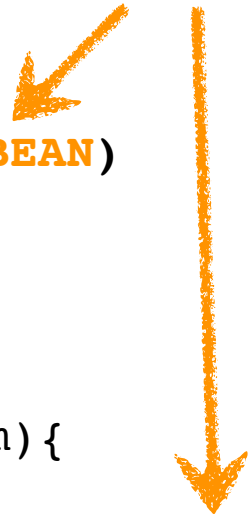
roll-back en cas d'erreur

UserTransaction

- encapsule les fonctionnalités de base fournies par un manager de transactions JEE
- pour obtenir une référence
  - injection
  - à partir du EJBContext
  - JNDI lookup

EJBContext

sinon, `IllegalStateException`



```
@TransactionManagement(TransactionManagementType.BEAN)
public class OrderManagerBean {
  @Resource
  private SessionContext sc;
  ...
  public void orderItem(Person customer, Item item){
     ...
     UserTransaction userTransaction = sc.getUserTransaction();
     ...
  }
}
```

- appels `sc.setRollbackOnly`, `sc.getRollbackOnly`
  - `IllegalStateException` (si dans BMT)

JNDI lookup

```
public class HelloWorldJNDIClient {
 public static void main(String [] args) {
  try {
    Context ctx = new InitialContext();
    HelloWorld hello = (HelloWorld) ctx.lookup(
         "java:global/HelloWorld/HelloWorld-ejb/HelloWorldBean");

    UserTransaction userTransaction = (UserTransaction) ctx.lookup(
         "java:comp/UserTransaction");
    userTransaction.begin();
    ...
    userTransaction.commit();
  } catch (Exception ex) {
    ex.printStackTrace();
  }
 }
}
```

UserTransaction

```
public interface UserTransaction {

  void begin() throws NotSupportedException, SystemException;

  void commit() throws ...;

  void rollback() throws ...;

  void setRollbackOnly() throws ...;

  int getStatus() throws ...;

  void setTransactionTimeout(int seconds) throws ...;

}
```

créer transaction et l'associer au thread

transmet "succès" à la transaction

abandonne la transaction

marque la transaction pour roll-back

`javax.transaction.Status` de la transaction

temps en secondes avant la fin de la transaction

Status

STATUS_ACTIVE	Associated transaction is in an active state.
STATUS_MARKED_ROLLBACK	Associated transaction is marked for rollback, possibly due to `setRollbackOnly`.
STATUS_PREPARED	Associated transaction is in the prepared state because all resources have agreed to commit.
STATUS_COMMITTED	Associated transaction has been committed.
STATUS_ROLLEDBACK	Associated transaction has been rolled back.
STATUS_UNKNOWN	Status for associated transaction is not known.
STATUS_NO_TRANSACTION	There is no associated transaction in the current thread.
STATUS_PREPARING	Associated transaction is preparing to be committed and awaiting response from subordinate resources.
STATUS_COMMITTING	The transaction is in the process of committing.
STATUS_ROLLING_BACK	The transaction is in the process of rolling back.

CMT vs. BMT

- BMT
  - *verbose, complexe, difficile à maintenir*
  - *ne peut pas rejoindre une transaction*
  - **début/fin transaction pas forcement délimités par une méthode** → données bloquées par le code le moins possible

Questions ?