Java EE 7 アプリケーション設計ガイド
- CDI 1.2 編
CDI: Contexts and Dependency Injection for the Java EE platform

日本アイ・ビー・エム　システムズ・エンジニアリング株式会社

© 2015 IBM Corporation

Disclaimer

- この資料は日本アイ・ビー・エム株式会社ならびに日本アイ・ビー・エム　システムズ・エンジニアリング株式会社の正式なレビューを受けておりません。
- 当資料は、資料内で説明されている製品の仕様を保証するものではありません。
- 資料の内容には正確を期するよう注意しておりますが、この資料の内容は2015年6月現在の情報であり、製品の新しいリリース、PTFなどによって動作、仕様が変わる可能性があるのでご注意下さい。
- 今後国内で提供されるリリース情報は、対応する発表レターなどでご確認ください。
- ＩＢＭ、ＩＢＭロゴおよびibm.comは、世界の多くの国で登録されたInternational Business Machines Corporationの商標です。他の製品名およびサービス名等は、それぞれＩＢＭまたは各社の商標である場合があります。現時点でのＩＢＭの商標リストについては、www.ibm.com/legal/copytrade.shtmlをご覧ください。
- 当資料をコピー等で複製することは、日本アイ・ビー・エム株式会社ならびに日本アイ・ビー・エム　システムズ・エンジニアリング株式会社の承諾なしではできません。
- 当資料に記載された製品名または会社名はそれぞれの各社の商標または登録商標です。
- JavaおよびすべてのJava関連の商標およびロゴは Oracleやその関連会社の米国およびその他の国における商標または登録商標です。
- Microsoft, Windows および Windowsロゴは、Microsoft Corporationの米国およびその他の国における商標です。
- Linuxは、Linus Torvaldsの米国およびその他の国における登録商標です。
- UNIXはThe Open Groupの米国およびその他の国における登録商標です。

© 2015 IBM Corporation

目次

1. はじめに
2. 概要
3. インジェクション(注入)
4. 生成・初期化とクリーンナップ
5. プロデューサーとディスポーザー
6. 代替実装と特殊化
7. スコープ
8. インターセプターとデコレーター
9. イベント通知
10. ステレオタイプ
11. 検出
12. まとめ

参考資料

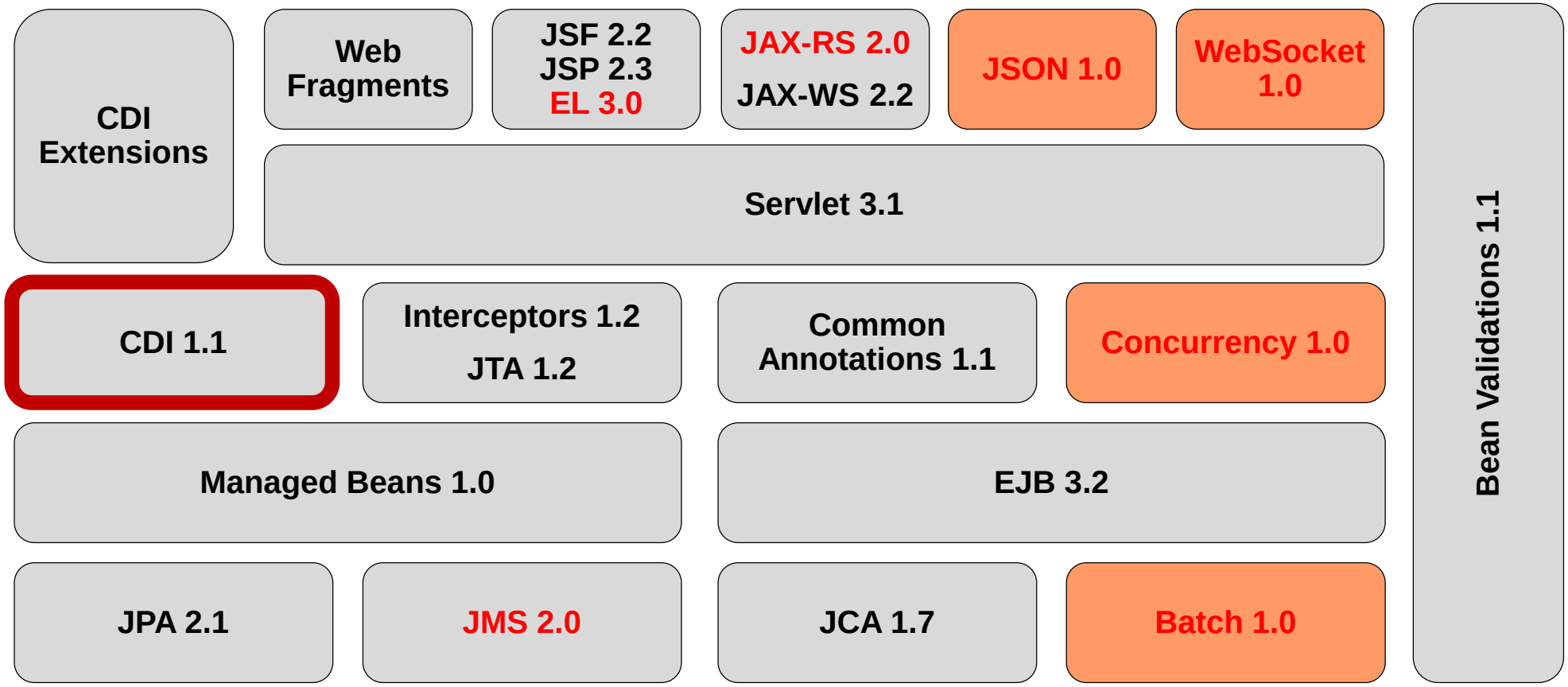
© 2015 IBM Corporation

1. はじめに

© 2015 IBM Corporation

Java EE 7 に含まれる仕様 (JSR一覧)

- Java EE 7 では、４つの仕様が新規に追加され、３つの仕様がバージョンアップされている
- 当ガイドでは、機能改善された CDI 1.1 のメンテナンス・リリースである CDI 1.2 を対象とする



© 2015 IBM Corporation

WebSphere Application Server のサポート状況 (2015年6月現在)

- Full Profile 版
  - CDI 1.0 をサポート

- Liberty Profile 版
  - V8.5.5.5 までは CDI 1.0 をサポート
  - V8.5.5.6 から CDI 1.2 に対応

© 2015 IBM Corporation

2. 概要

© 2015 IBM Corporation

CDI (Contexts and Dependency Injection) とは

- DI (Dependency injection, 依存性の注入) とは
  - コンポーネント間の依存関係をプログラムのソースコードから排除し、外部の設定ファイルなどで注入できるようにするソフトウェアパターンである
  - 依存関係がプログラムから外部に取り除かれることで、以下のようなメリットが発生する
    - 結合度の低下によるコンポーネント化の促進
    - 単体テストの効率化
    - 特定のフレームワークへの依存度低下
  - 出典: Wikipedia

- CDI は、DI の機能に加えて、コンポーネントである bean のライフサイクル・コンテキストの管理機能を提供
  - Contexts　　bean のライフサイクルをライフサイクル・コンテキストに関連付ける
  - Dependency Injection　　bean のインスタンスをインジェクション(注入)する

- CDI では、アノテーションと Bean の型に従った、タイプ・セーフな DI を実現

© 2015 IBM Corporation

bean と CDI 管理 Bean (CDI-style Managed Bean)

- CDI は以下のものを bean として扱う (CDI の bean の定義)
  - CDI 管理 Bean (CDI-style Managed Bean)
  - EJB セッション Bean
  - プロデューサー・メソッドが返すもの
  - プロデューサー・フィールドが示すもの
  - リソース
    - Java EE リソース (データソース、接続ファクトリーなど)
    - パーシスタンス・コンテキスト
    - パーシスタンス・ユニット
    - リモート EJB
    - Web サービス

- CDI 管理 Bean (CDI-style Managed Bean) は、以下の全て条件を満たす Java クラスである
  - コンクリートな Java クラスであるか、または、@Decorator が付与されている
  - パラメーターなしのコンストラクターを持つ、または、@Inject が付与されたコンストラクターを持つ
  - non-static なインナー・クラスでない
  - EJB として定義されていない
  - javax.enterprise.inject.spi.Extension を実装していない
  - @Vetoed が付与されていない、@Vetoed が付与されたパッケージ内に存在しない
- CDI 管理 Bean は、特殊な Java のクラスではない

© 2015 IBM Corporation

bean の例

- bean の定義例を示す（詳細は次章を参照）

```
@Named
@RequestScoped
public class UserInfo {
    private String username;
    private String address;
    ......
    getter/setter メソッド
}
```

- Servlet や JSF で使用する CDI 管理 Bean の例
- 定義としては、JavaBean と同様の内容
- アノテーション「@Named」でデフォルトの bean 名を付与
- アノテーション「@RequestScoped」でスコープを定義
- スコープを指定するアノテーションが付与されているので、bean discovery mode が annotated でも bean として検出される

```
@Stateless
public class BookShopBean extends    Business
                          implements BookShop {

    public void audit( ··· ) {
        ......
    }
    public void sell( ··· ) {
        ......
    }
    ......
}
```

- セッション Bean の例
- bean discovery mode が none 以外であれば bean として検出される
- インターフェース BookShop を実装し、Business のサブクラスとして定義されている

```
public abstract class Business {
    public abstract void audit( ··· );
    ......
}
```

```
public interface BookShop {
    public void sell( ··· );
    ......
}
```


© 2015 IBM Corporation

インジェクション(注入)の利用例

- インジェクション(注入)の利用例を示す（詳細は次章を参照）

Bean の型に従った、
タイプ・セーフな注入が行われる!!!

CDI 管理 Bean

```
@Named
@RequestScoped
public class OrderOperation {
    ......
    @Inject
    private BookShop shop;
    ......
}
```

- JSF で使用する CDI 管理 Bean などを想定
- アノテーション「@Inject」で BookShop 型の bean を注入

- アノテーション「@Inject」でデータソースを注入
- 限定子(Qualifie)で目的のデータソースを指定

```
@Stateless
public class BookShopBean extends    Business
                          implements ShopBook {
    ......
    @Inject
    @ShopDb private DataSource shopDb;
    ......
}
```

セッション Bean

- データソースを提供
- 限定子(Qualifie)を付与

```
@Dependent
public class Resources {

    @Produces
    @Resource(lookup="jdbc/ShopDB")
    @ShopDb public DataSource ds;

}
```


© 2015 IBM Corporation

bean discovery mode （1）

- CDI がアーカイブ(jar, war)内の bean を検出するモード(bean discovery mode)は 3 種類
  - beans.xml は、CDI のデプロイメント・デスクリプター用の XML ファイル

- annotated モード（デフォルト、推奨）
    - アーカイブ内に beans.xml が存在しない場合、または、
    - アーカイブ内にバージョン 1.1 の beans.xml が存在し、bean-discovery-mode が annotated の場合
  - セッション Bean と Bean 定義アノテーションが付与された CDI 管理 Beanが bean として認識される
    - Bean 定義アノテーション：
      スコープを指定するアノテーション、@Dependent、@Interceptor、@Decorator、ステレオタイプのアノテーション
- all モード
    - アーカイブ内に空の beans.xml が存在する場合、または、
    - アーカイブ内にバージョン指定のない beans.xml ファイルが存在する場合、または、
    - アーカイブ内にバージョン 1.1 の beans.xml が存在し、bean-discovery-mode が all の場合
  - セッション Bean と全ての CDI 管理 Bean が bean として認識される
- none モード
    - アーカイブ内にバージョン 1.1 の beans.xml が存在し、bean-discovery-mode が none の場合
  - 全て Bean として認識されない

© 2015 IBM Corporation

bean discovery mode （2）

- beans.xml の配置場所
  - jarファイルの場合：　META-INF/beans.xml
  - warファイルの場合：　WEB-INF/beans.xml または WEB-INF/classes/META-INF/beans.xml

- beans.xml の例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">
</beans>
```

- CDI 1.0 との互換性維持
  - CDI 1.0 との互換性を維持するために、beans.xml が存在しないアーカイブ(jar, war)を無視する設定を提供することになっている
  - WebSphere Liberty Profile では、server.xm 内の以下のタグで設定できる

```
<cdi12Container disableImplicitBeanArchives="true"></cdi12Container>
```


© 2015 IBM Corporation

CDI のその他の機能

- EL (Expression Language) との連携
  - JSF facelets や JSP から bean にアクセスする機能(EL から bean にアクセスする機能)を提供
- CDI インターセプター
  - メソッド呼び出しやライフサイクル・イベントが発生した時に、クロスカティング(横断的)タスクを差し込むための機能を提供
- デコレーター
  - メソッド呼び出しが発生した時に、ビジネス・ロジックを差し込んだりカスタマイズするための機能を提供
- イベント通知
  - bean 間でイベントの送信と受信を行う機能を提供
- 機能追加用の SPI (Service Provider Interface)
  - CDI の機能を拡張するためのインターフェースを提供
  - 本資料では取り扱いません

© 2015 IBM Corporation

CDI 1.1 (1.2) の主な新機能

- CDI 1.1 (1.2) から追加された主な新機能は以下の通りです
  - @Priority を使用したグローバル・イネイブルメント
    - インターセプター(Interceptor)
    - デコレーター(Decorator)
    - 代替(alternatives)実装
  - @AroundConstruct のサポート
  - コンストラクターへのインターセプターのバインドのサポート
  - 組み込み Bean に対するデコレーターのサポート
  - イベントのメタデータ検査用に EventMetadata を追加
  - @Vetoed による CDI 無効化のサポート
  - CDI の自動有効化
  - beans.xml へのクラス除外フィルターの追加

© 2015 IBM Corporation

3. インジェクション(注入)

© 2015 IBM Corporation

Bean Type

- bean は複数の Bean Type を持つことがある
  - CDI 管理 Bean のデフォルトの Bean Type
    - 自身のタイプ
    - 親クラスのタイプ
    - 実装したインターフェースのタイプ
    - Object タイプ
  - セッション Bean のデフォルトの Bean Type
    - 実装したインターフェースのタイプ
    - Object タイプ
- @Type アノテーションで Bean Type を限定できる
  - 複数 Bean Type の指定も可能
    - 例：@Typed({ BookShop.class, Business.class })

CDI 管理 Bean

```
@Dependent
public class BookShopImpl extends    Business
                          implements BookShop {
    ......
}
```

・CDI 管理 Bean なので、Bean Type は、BookShopImpl, Business, BookShop および Object になる

セッション Bean

```
@Stateless
public class BookShopBean extends    Business
                          implements ShopBook {
    ......
}
```

・セッション Bean なので、Bean Type は、BookShop と Object だけになる

```
@Dependent @Typed(BookShop.class)
public class BookShopImpl extends    Business
                          implements BookShop {
    ......
}
```

・Bean Type を BookShop に限定

© 2015 IBM Corporation

Bean Type を使用した注入

- bean は @Inject で注入する
  - javax.inject パッケージ
  - Bean Type に従って、タイプ・セーフな注入が実行される
  - インジェクション先は non-static で non-final なフィールド
  - インジェクションをサポートするコンポーネントは次ページを参照

```
@Named
@RequestScoped
public class OrderOperation {
    ......
    @Inject
    private BookShop shop;
    ......
}
```

・Bean Type が BookShop の bean を注入

```
@Stateless
public class BookShopBean extends    Business
                          implements ShopBook {
    ......
    @Inject
    @ShopDb private DataSource shopDb;
    ......
}
```

・Bean Type が DataSource の bean を注入

【注意】bean のコンストラクターを直接呼び出してインスタンスを作成してしまうと、CDI コンテナーの機能は享受できない

- bean はコンテキストに関連付けられず、CDI によるライフサイクル管理が行われない
- 依存関係の注入が行われない
- インターセプター、デコレーターなどが関連付けられない

© 2015 IBM Corporation

インジェクションをサポートするコンポーネント

- CDI の bean をインジェクションできるコンポーネント(インジェクション先のコンポーネント)は、Java EE 7 の仕様で以下のように定められている

仕様	コンポーネント
Servlet	• servlet • servlet filter • event listener • HTTP upgrade handler
JSP	• tag handler • tag library event listener
JSF	• managed class (詳細は JSF の仕様の "JSF Managed Classes and Java EE Annotations" を参照)
JAX-WS	• service endpoint • handler
JAX-RS	• JAX-RS component (CDI管理の場合)

仕様	コンポーネント
WebSocket	• endpoint
EJB	• bean
Interceptor	• interceptor
Java Persistence	• entity listener
Managed Beans	• managed bean
CDI	• CDI-style managed bean • decorator
Java EE Platform	• main class (static) • login callback handler

© 2015 IBM Corporation

組み込みの Bean

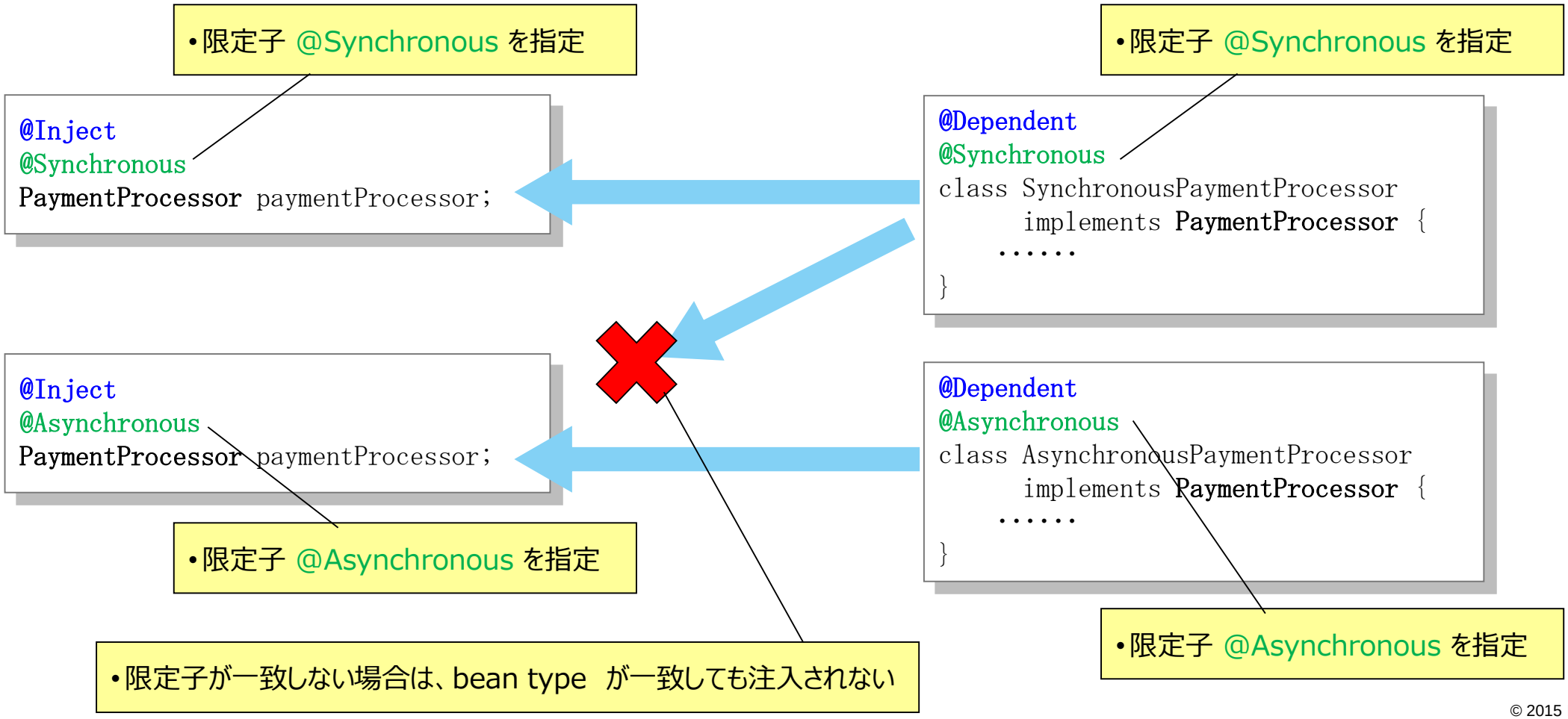
- 組み込みの Bean として以下のものが使用できる
  - 限定子 @Default が付与されている

組み込みの Bean	Bean Type
EJB コンテナー	
JTA ユーザー・トランザクション	javax.transaction.UserTransaction
呼び出し元のプリンシパル	javax.security.Principal
Web コンテナー	
HTTP サーブレット・リクエスト	javax.servlet.http.HttpServletRequest
HTTP セッション	javax.servlet.http.HttpSession
サーブレット・コンテキスト	javax.servlet.ServletContext

© 2015 IBM Corporation

限定子 (Qualifier)

- 複数の bean が同じ bean type を持つ場合、bean type だけでは bean を特定できない
- 限定子を定義し、限定子を付与することで、どの bean を注入するか指定できる



© 2015 IBM Corporation

限定子 (Qualifier) の定義

- 限定子は @Qualifier アノテーションを使用して定義する
  - javax.inject パッケージ

限定子の定義例

```
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Synchronous {}
```

- 限定子 Synchronous を定義している例
- @Qualifier で、限定子であることを宣言
- @Target で、METHOD, FIELD, PARAMETER および TYPE に付与できるアノテーションであることを宣言
- @Retention(RUNTIME) で、実行時までアノテーション情報が残ることを宣言
- @Inherited は指定していない（サブクラスには継承されないように宣言）

© 2015 IBM Corporation

メンバーを持つ限定子 (Qualifier) の定義

- メンバー(プロパティー)を持つ限定子も定義できる

メンバーを持つ限定子の定義例

```
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface PayBy {
    PaymentMethod value();
}
```

```
public enum PaymentMethod {
    CASH, CREDIT_CARD, CHEQUE
}
```

メンバーを持つ限定子の付与例

```
@Dependent
@Asynchronous
@PayBy(PaymentMethod.CASH)
class AsynchronousCachPaymentProcessor
      implements PaymentProcessor {
    ......
}
```

- 限定子 PayBy を定義している例
- メンバー(プロパティー)にアクセスするためのメソッド value() を追加
- この例では、メンバーの型を enum を利用して定義
- その他はメンバーを持たない限定子と同様

© 2015 IBM Corporation

組み込みの限定子 (Qualifier) : @Any, @Default, @New

- @Any
  - javax.enterprise.inject パッケージ
  - 全ての bean に暗黙的に付与される限定子
  - @New 限定子が付与された場合のみ消える
- @Default
  - javax.enterprise.inject パッケージ
  - 全ての bean に暗黙的に付与される限定子 (デフォルト限定子)
  - @Named 以外の限定子が付与されると消える
- @New
  - javax.enterprise.inject パッケージ
  - CDI 1.1 以降では非推奨(deprecated)
  - CDI 1.1 以降では @Dependent が付与されていると解釈される

```
@Dependent
public class Order {
      ......
}
```

- @Any と @Default の２つの限定子が付与されていることになる

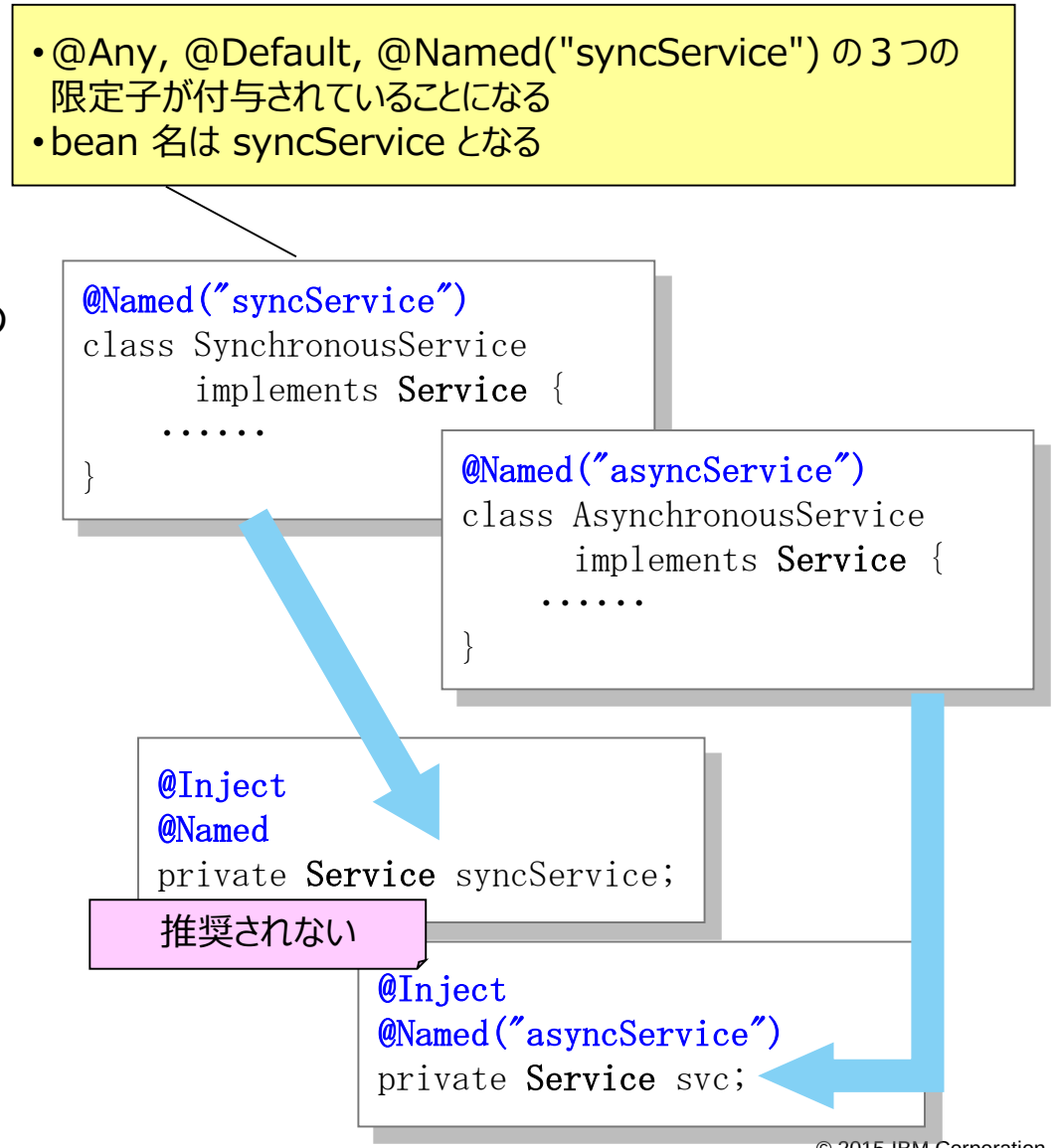
```
@Dependent
@Asynchronous
class AsynchronousPaymentProcessor
      implements PaymentProcessor {
    ......
}
```

- @Any と @Asynchronous の２つの限定子が付与されていることになる
- @Default は除去される

© 2015 IBM Corporation

組み込みの限定子 (Qualifier) : @Named

- @Named
  - javax.inject パッケージ
  - bean 名を指定する限定子 (bean に名前を付ける限定子)
    - 例 : @Named("userInfo")
  - bean 名は、JSF facelets や JSP から bean にアクセスするために使用（EL から bean にアクセスするために使用）
  - @Inject と @Named を組み合わせて注入することもできるが、タイプ・セーフで無くなるので注意が必要（推奨されない）
- デフォルトの bean 名
  - @Named のみを指定した場合の bean 名
  - CDI 管理 Bean　最初を小文字にしたクラス名
  - セッション Bean　最初を小文字にしたクラス名
  - プロデューサー・メソッド　メソッド名 (getter の場合はプロパティー名)
  - プロデューサー・フィールド　フィールド名
  - 注入操作の場合　注入先のフィールド名、パラメーター名



© 2015 IBM Corporation

bean の Programmatic Lookup（1）

- Bean Type と限定子を指定して bean を注入できるが、場合によっては困ることもある
  - どの bean が必要になるかが、実行時に決まる場合
    - 但し、Programmatic Lookup を使用しなくても、後述のプロデューサー・メソッドで対応できるケースもある
  - 特定の Bean Type の全てのインスタンスをイテレート操作したい場合

- bean の Programmatic Lookup を使用すると、ダイナミックに bean を取得し操作することが可能

- Programmatic Lookup の方法
  - @Inject で javax.enterprise.inject.Instance<T> を注入する
    - T には、必要とする bean の Bean Type を指定
  - Instance<T> が提供するメソッドを使用して、bean を取得する

Instance<T> の定義

```
public interface Instance<T> extends Iterable<T>, Provider<T> {
    public Instance<T> select(Annotation... qualifiers);
    public <U extends T> Instance<U> select(Class<U> subtype, Annotation... qualifiers);
    public <U extends T> Instance<U> select(TypeLiteral<U> subtype, Annotation... qualifiers);
    public boolean isUnsatisfied();
    public boolean isAmbiguous();
    public void destroy(T instance);
}
```


© 2015 IBM Corporation

bean の Programmatic Lookup（2）

- 単純な Programmatic Lookup
  - 動的な操作を行っていないので、Bean Type と限定子を使用した通常の注入でも実現できる例

```
@Inject
@Synchronous
Instance<PaymentProcessor> paymentProcessor;
```

- @Inject と限定子を指定して Instance オブジェクトを注入
- この例では、Bean Type が PaymentProcessor の bean を取得するための、Instance オブジェクトが注入される

```
PaymentProcessor pp = paymentProcessor.get();
```

- 注入された Instance<PaymentProcessor> の get() メソッドを呼び出し、bean のインスタンスを取得

© 2015 IBM Corporation

bean の Programmatic Lookup（3）

- Programmatic Lookup の例
  - 使用する bean を変数の値の応じて動的に切替える例

```
@Inject @Any
Instance<PaymentProcessor> anyPaymentProcessor;
```

- @Inject と @Any を指定して Instance オブジェクトを注入
- @Any を指定しているので、限定子が付与されている bean も Lookup の対象となる

```
class SynchronousQualifier  extends AnnotationLiteral<Synchronous>
                                    implements Synchronous {}
class AsynchronousQualifier extends AnnotationLiteral<Asynchronous>
                                    implements Asynchronous {}

Annotation qualifier = synchronously ? new SynchronousQualifier() : new AsynchronousQualifier();
PaymentProcessor pp = anyPaymentProcessor.select(qualifier).get();
```

- javax.enterprise.util.AnnotationLiteral<T> を使用して、限定子用の java.lang.annotation.Annotation をインナー・クラスとして宣言
- 変数の値に応じて、どちらのアノテーションを指定するか判定
- Instance<T> の select() メソッドで bean を絞り込んでから、get() メソッドで bean のインスタンスを取得

  - 全てのインスタンスをイテレート操作する例

```
for (PaymentProcessor pp: anyPaymentProcessor) pp.test();
```

- Instance<T> は Iterable<T> を実装しているので、全ての bean のインスタンスをイテレート操作することが可能

© 2015 IBM Corporation

注入ポイントのメタデータ (Injection Point Metadata)

- CDI コンテナーは、bean が注入されたポイントのメタデーターを提供
  - メタデータの Bean Type : InjectionPoint
    - javax.enterprise.inject.spi パッケージ

- メタデータの注入
  - 通常の bean と同様に、 @Inject で注入できる
    - bean のフィールド、コンストラクターのパラメーター、初期化メソッド、プロデューサー・メソッドなどに注入できる
  - 但し、注入先となる bean のスコープは、@Dependent でなければならない

- 用途
  - ログ出力用の bean などが、注入先のクラスを認識するために使用する

InjectionPoint の定義

```
public interface InjectionPoint {
    public Type getType();
    public Set<Annotation> getQualifiers();
    public Bean<?> getBean();
    public Member getMember();
    public Annotated getAnnotated();
    public boolean isDelegate();
    public boolean isTransient();
}
```

ログ出力用の bean を返す、プロデューサー・メソッドでの利用例

```
@Produces
Logger createLogger( InjectionPoint injectionPoint ) {
    return Logger.getLogger( injectionPoint.getMember().getDeclaringClass().getName() );
}
```


© 2015 IBM Corporation

4. 生成・初期化とクリーンナップ

© 2015 IBM Corporation

bean のインスタンス生成

- CDI コンテナーは管理 Bean のコンストラクターを呼び出して bean のインスタンスを作成する

- 管理 Bean のコンストラクター
  - default-access, public, protected または private なコンストラクターが使用される
  - @Inject アノテーションが付与されたコンストラクターが優先的に使用される
  - 存在しない場合は、デフォルト・コンストラクター(パラメーターなしのコンストラクター)が使用される

- パラメーターの注入
  - パラメーターを持つ場合は CDI コンテナーが呼び出し時に注入する
  - パラメーターには限定子を付与できる

```
@ConversationScoped
public class Order {

    private Product product;
    private User customer;

    @Inject
    public Order( @Selected Product product,
                              User customer ) {
        this.product  = product;
        this.customer = customer;
    }

    public Order( Order original ) {
        this.product  = original.product;
        this.customer = original.customer;
    }

    Order() {
    }
    ......
}
```

- この例では 3 つのコンストラクターが定義されているが、CDI コンテナーが使用するのは @Inject が付与されたもの
- パラメーター product と customer が注入される

© 2015 IBM Corporation

初期化メソッド (Initializer Methods)

- CDI コンテナーは bean のインスタンスを作成し、フィールドに注入を行った後に、初期化メソッドを呼び出す
  - 初期化メソッド内では、注入されたフィールドを使用できる
- 初期化メソッド
  - @Inject を付与して定義
  - 複数個定義できる （定義しなくても良い）
  - default-access, public, protected または private で、non-abstract, non-static, non-generic なメソッド
  - セッション Bean の場合は、ビジネス・メソッドである必要はない
- パラメーターの注入
  - パラメーターを持つ場合は CDI コンテナーが呼び出し時に注入する
  - パラメーターには限定子を付与できる

```
@ConversationScoped
public class Order {

    private Product product;
    private User customer;

    @Inject
    void setProduct( @Selected Product product ) {
        this.product = product;
    }

    @Inject
    public void setCustomer( User customer ) {
        this.customer = customer;
    }
}
```

- この例では 2 つの初期化メソッドが定義されており、両方とも使用される
- パラメーター product と customer が注入される

© 2015 IBM Corporation

コールバック・メソッド：PostConstruct

- 「Dependency Injection for Java」で規定
- @PostConstruct アノテーションを付与して定義
  - javax.annotation パッケージ
- CDI コンテナーは bean の初期化メソッドの実行後に、PostConstruct メソッドを呼び出す
- PostConstruct メソッド
  - default-access, public, protected または private で、non-abstract, non-static なメソッド
  - 「void <メソッド名>() 」の形式
  - @PostConstruct を付与できるメソッドは 1 つのみ
- パラメーターの注入は行われない

```
@ConversationScoped
public class Order {

    private Product product;
    private User customer;

    @Inject
    void setProduct( @Selected Product product ) {
        this.product = product;
    }

    @Inject
    public void setCustomer( User customer ) {
        this.customer = customer;
    }

    @PostConstruct
    void init() {
        ......
    }

}
```

- この例では init() が PostConstruct メソッド

© 2015 IBM Corporation

コールバック・メソッド：PreDestroy

- 「Dependency Injection for Java」で規定
- @PreDestroyアノテーションを付与して定義
  - javax.annotation パッケージ
- CDI コンテナーは bean を破棄する直前に、PreDestroy メソッドを呼び出す
- PreDestroy メソッド
  - default-access, public, protected または private で、non-abstract, non-static なメソッド
  - 「void <メソッド名>() 」の形式
  - @PreDestroy を付与できるメソッドは１つのみ
- パラメーターの注入は行われない

```
@ConversationScoped
public class Order {

    private Product product;
    private User customer;

    @Inject
    void setProduct( @Selected Product product ) {
        this.product = product;
    }

    @Inject
    public void setCustomer( User customer ) {
        this.customer = customer;
    }

    @PreDestroy
    void clear() {
        ......
    }

}
```

- この例では clear() が PreDestroy メソッド

© 2015 IBM Corporation

5. プロデューサーとディスポーザー

© 2015 IBM Corporation

プロデューサーとディスポーザー

- プロデューサー
  - コンストラクターで生成できないオブジェクトなどを CDI の bean として扱いたい時に使用する
    - オブジェクトは CDI 管理 Bean で無くても良い
    - オブジェクトの実装型が動的に変わっても良い
    - 特殊な初期化処理が必要なオブジェクトでも良い
  - bean のファクトリーとしての役割を持つ
  - Java EE のリソースやコンポーネントを @Inject で注入する場合に使用する

- ディスポーザー
  - プロデューサーが返したオブジェクトのクリーンナップ処理を行いたい時に使用する

© 2015 IBM Corporation

プロデューサー・メソッド

- 注入されるオブジェクトを返すメソッド

- プロデューサー・メソッドの定義
  - @Produces アノテーションを付与
    - javax.enterprise.inject パッケージ
  - スコープを定義するアノテーションや、@Named などの限定子を付与可能
  - CDI 管理 Bean または セッション Bean のメソッド
    - default-access, public, protected または private で、static または non-static なメソッド
  - メソッドの戻り値の型で Bean Type が決まる

- プロデューサー・メソッドのパラメーターの注入
  - パラメーターを持つ場合は CDI コンテナーが呼び出し時に注入する
  - パラメーターには限定子を付与できる

```
public class OrderFactory {
    @Produces @ConversationScoped
    public Order createCurrentOrder( Shop shop,
                                     @Selected Product product ) {
        Order order = new Order( product, shop );
        return order;
    }
}
```

- スコープが定義されたオブジェクトを返す例
- パラメーター shop と product が注入される
- Bean Type は Order となる

© 2015 IBM Corporation

プロデューサー・フィールド

- 注入されるオブジェクトを示すフィールド
  - プロデューサー・メソッドの簡易版

- プロデューサー・フィールドの定義
  - @Produces アノテーションを付与
    - javax.enterprise.inject パッケージ
  - スコープを定義するアノテーションや、@Named などの限定子を付与可能
  - CDI 管理 Bean または セッション Bean のフィールド
    - default-access, public, protected または private フィールド
    - セッション Bean の場合は、static なフィールドでなければならない
  - フィールドの型で Bean Type が決定

```
public class Shop {

    @Produces @ApplicationScoped @Catalog @Named("catalog")
    List<Product> products = ....;

}
```

- スコープが定義された例
- 限定子として、@Named とユーザー定義の @Catalog が付与されている
- Bean Type は List<Product> となる

© 2015 IBM Corporation

ディスポーザー・メソッド

- プロデューサー・メソッドやフィールドが返したオブジェクトのクリーンナップ処理を実装できるメソッド

- ディスポーザー・メソッドの定義
  - CDI 管理 Bean または セッション Bean のメソッド
    - default-access, public, protected または private で、static または non-static なメソッド
  - パラメーターを 1 つだけ持つ
    - クリーンナップするオブジェクトが渡される
    - プロデューサー・メソッドの戻り値の型やプロデューサー・フィールドの型に合致する必要がある
    - 限定子も加味される
  - パラメーターに @Disposes アノテーションを付与
    - javax.enterprise.inject パッケージ

```
public class UserDatabaseEntityManager {

    @Produces @ConversationScoped @UserDatabase
    public EntityManager create( EntityManagerFactory emf ) {
        return emf.createEntityManager();
    }

    public void close( @Disposes @UserDatabase EntityManager em ) {
        em.close();
    }
}
```

- プロデューサー・メソッドとディスポーザー・メソッドの例
- 両者とも、EntityManager 型のオブジェクトを処理する
- 両者とも、ユーザー定義の限定子 @UserDatabaseが付与されている

© 2015 IBM Corporation

Java EE リソースやコンポーネントの注入

- Java EE リソースやコンポーネントを注入する場合
  - プロデューサー・フィールドに Java EE リソースやコンポーネントを注入するためのアノテーションを付与する
  - 同じ型のリソースなどは、限定子を付与して区別する

- Java EE リソースやコンポーネントを注入するためのアノテーション
  - ✓ Java EE リソース： @Resource
  - ✓ パーシスタンス・コンテキスト： @PersistenceContext
  - ✓ パーシスタンス・ユニット： @PersistenceUnit
  - ✓ リモート EJB： @EJB
  - ✓ Web サービス： @WebServiceRef

```
@Inject
PaymentWebService paymentWeb;

@Inject
PaymentEjbService paymentEjb;

@Inject
@CustomerDatabase
Datasource custDb;

@Inject
@CustomerDatabase
EntityManager custEm;

@Inject
@CustomerDatabase
EntityManagerFactory custEmf;
```

```
public class AppResources {

    @Produces @WebServiceRef( lookup="java:app/service/PaymentService" )
    PaymentWebService paymentWebService;

    @Produces @EJB( ejbLink="../their.jar#PaymentService" )
    PaymentEjbService paymentEjbService;

    @Produces @Resource( lookup="jdbc/CustomerDatasource" )
    @CustomerDatabase Datasource customerDatabase;

    @Produces @PersistenceContext( unitName="CustomerDatabase" )
    @CustomerDatabase EntityManager customerPersistenceContext;

    @Produces @PersistenceUnit( unitName="CustomerDatabase" )
    @CustomerDatabase EntityManagerFactory customerPersistenceUnit;

}
```


© 2015 IBM Corporation

6. 代替実装と特殊化

© 2015 IBM Corporation

Bean の代替(Alternative)実装の利用 （１）

- 複数の bean を状況に応じて使い分ける場合に役立つ
  - 例-1. テスト用にダミー・コードを埋め込んだ bean を使いたい場合
  - 例-2. 要件に応じてアプリの動作を切り替えたい場合

- 代替(Alternative)実装
  - @Alternative アノテーションを付与して定義
    - javax.enterprise.inject パッケージ
  - beans.xml の alternatives タグを使用して有効化
    - アノテーションを付与しただけでは代替として使用されない
    - 複数の代替実装のうち、有効にできるのは１つだけ

```
<beans ......>
    <alternatives>
        <class>javaeetutorial.encoder.TestCoderImpl</class>
    </alternatives>
</beans>
```

• beans.xml で使用する代替実装を指定

```
@Dependent
public class CoderImpl implements Coder {
    .........
}
```

• プロダクション用の実装コード

```
@Dependent
@Alternative
public class TestCoderImpl implements Coder {
    .........
}
```

• テスト用に作成した代替実装
• @Alternative アノテーションを付与

```
@Inject
Coder coder;
```

• 注入される側では、
共通の Bean Type で注入

© 2015 IBM Corporation

Bean の代替(Alternative)実装の利用 （２）

- beans.xml で指定した代替実装が使用されるのは、同じアーカイブ内のみ

- 複数アーカイブで構成されるアプリにおいて、アプリ全体で代替(Alternative)実装を切り替えたい場合は、@Priority アノテーションを付与する
  - javax.annotation パッケージ
  - 値が最も大きい代替実装が有効になる
    - 「インターセプターとデコレーター」の章の「補足：@Priority で指定する値」を参照
  - beans.xml での指定は不要になるが、切り替えにはソース・ファイルの修正が必要となる

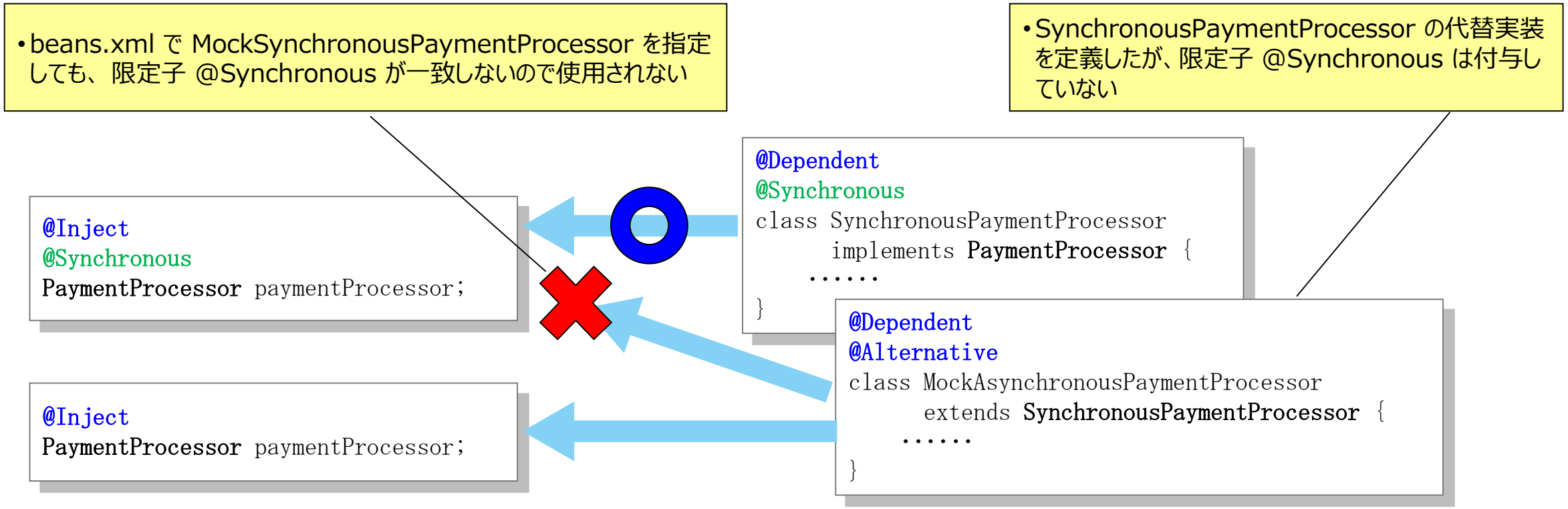
```
@Dependent
@Alternative
@Priority(Interceptor.Priority.APPLICATION+10)
public class TestCoderImpl implements Coder {
    .........
}
```

- @Alternative アノテーションに加え、@Priority で値を指定
- 値が最も大きい代替実装が使用される

© 2015 IBM Corporation

Bean の代替(Alternative)実装の利用 （３）

- 限定子が付与/指定されている場合は、限定子も同じでないと、代替実装として使用されない
  - 限定子が付与されていない場合だけ代替実装を使いたい場合や、特定の限定子が付与されている場合だけ代替実装を使いたい場合には便利な動作となる
  - しかし、限定子の指定に関係なく全てを代替実装に置き換えたい場合には不便な動作となる



© 2015 IBM Corporation

Bean の特殊化(Specialization)の利用

- 全てを代替実装に置き換えたい場合は、特殊化が便利
- 特殊化(Specialization)の実装
  - 元の bean を**直接サブクラス化**し、@Specializes アノテーション(javax.enterprise.inject パッケージ)を付与する
    - 元の bean が持つ全ての限定子(bean名を含む)が継承される
  - beans.xml での指定は不要

```
@Dependent
public class LoginAction {
    ......
```

CDI 管理 Bean の例

```
@Dependent
@Specializes
public class MockLoginAction extends LoginAction {
    ......
```

```
@Stateless
public class LoginActionBean implements LoginAction {
    ......
```

セッション Bean の例

```
@Stateless
@Specializes
public class MockLoginActionBean extends LoginActionBean
                                 implements LoginAction {
    ......
```

```
@Dependent
public class Shop {
    @Produces
    PaymentProcessor getPaymentProcessor() {
        return new PaymentProcessor();
    }
    ......
```

プロデューサー・メソッドの例

```
@Dependent
public class MockShop extends Shop {
    @Override
    @Specializes
    @Produces
    PaymentProcessor getPaymentProcessor() {
        return new MockPaymentProcessor();
    }
    ......
```


© 2015 IBM Corporation

7. スコープ

© 2015 IBM Corporation

スコープ（１）

- bean のスコープを定義することで、bean をコンテキストに関連付け、bean のライフサイクルを管理できる
  - 必要な時に自動的に bean を作成し、コンテキストが終了する時に自動的に bean を破棄する
  - 同じコンテキスト内では、同じ bean が共有される
  - コンテクスチュアル(Contextual)な動作が実現できる

- 以下の４つのスコープが定義されている
  - javax.enterprise.context パッケージ内のアノテーションで指定する

アノテーション	スコープ
@RequestScoped	• １回の HTTP リクエストの範囲
@ConversationScoped	• 複数回の HTTP リクエストの範囲 • 開発者が開始と終了を制御
@SessionScoped	• １つの HTTP セッションの範囲
@ApplicationScoped	• １つのアプリケーションの範囲
@Dependent	• 注入先ごとにインスタンスが作成され、注入先と同じスコープを持つ

• @RequestScoped, @SessionScoped, @ApplicationScoped は、JSF が定義している同名のスコープと同義
• JSF 側のアノテーションは、javax.faces.bean パッケージ内に存在

• java.io.Serializable を実装する必要がある

• java.io.Serializable を実装する必要がある

• @Dependent は疑似スコープ (pseudo scope)
• コンテクスチュアルな動作とならない

© 2015 IBM Corporation

スコープ（２）

- セッション Bean の場合は、使用できるスコープが限定される
  - Stateful セッション Bean　　任意のスコープ
  - Stateless セッション Bean　　@Dependent のみ
  - Singleton セッション Bean　　@ApplicationScoped、または、@Dependent

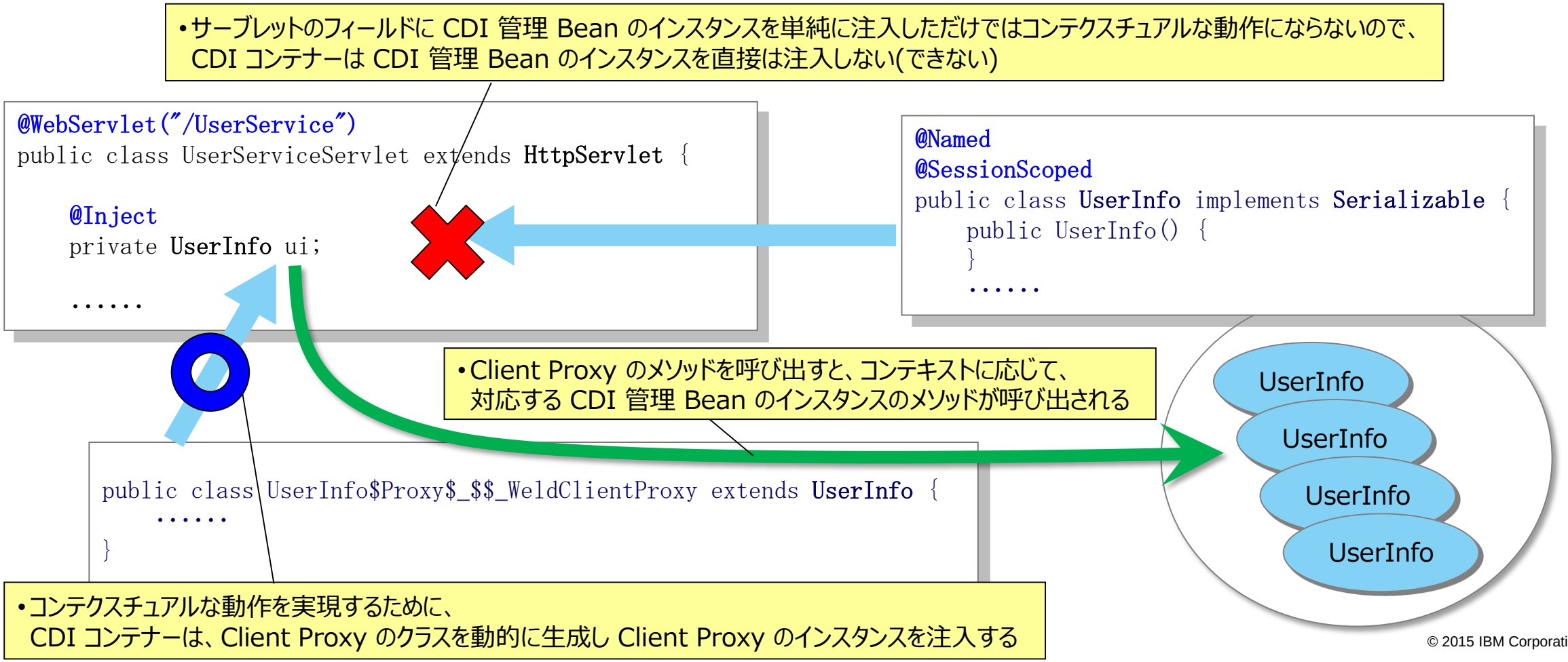
- スコープを持つ CDI 管理 Bean の実装では、以下の制約が追加される
  - パラメーターを持たない、private 以外のデフォルト・コンストラクターを持つ
  - クラスが final でない (サブクラスが作成できる)
  - public, protected または default-access の 非 static なメソッドが final でない (メソッドがオーバーライドできる)
  - public なフィールドが存在しない (フィールド値の変更はメソッド経由でのみ実行できる)

  - この制約は、コンテクスチュアル(Contextual)な動作を実現するための Client Proxy を生成するために必要となる

© 2015 IBM Corporation

スコープ（3）

- CDI コンテナーは、コンテクスチュアル(Contextual)な動作を実現するために、Client Proxy を生成
- 例：サーブレットにセッション・スコープの CDI 管理 Bean を注入した場合



© 2015 IBM Corporation

スコープ（４）

- Conversation スコープの制御
  - Conversation (CDI コンテナーが提供する組み込み Bean の１つ)を注入
    - javax.enterprise.context パッケージ
  - Conversation の begin()/end() メソッドを呼び出して、Conversation スコープを開始/停止

```
@Inject
private Conversation cnv;

public String execute() {
    if ( cnv.isTransient() ) cnv.begin();
    ......
    return "nextPage?faces-redirect=true";
}
```

```
@Inject
private Conversation cnv;

public String execute() {
    if ( ! cnv.isTransient() ) cnv.end();
    ......
    return "menuPage?faces-redirect=true";
}
```

利用例

```
package javax.enterprise.context;

public interface Conversation {
    public void begin();
    public void begin(String id);
    public void end();
    public String getId();
    public long getTimeout();
    public void setTimeout(long milliseconds);
    public boolean isTransient();
}
```

参考：Conversation インターフェースの定義

© 2015 IBM Corporation

8. インターセプターとデコレーター

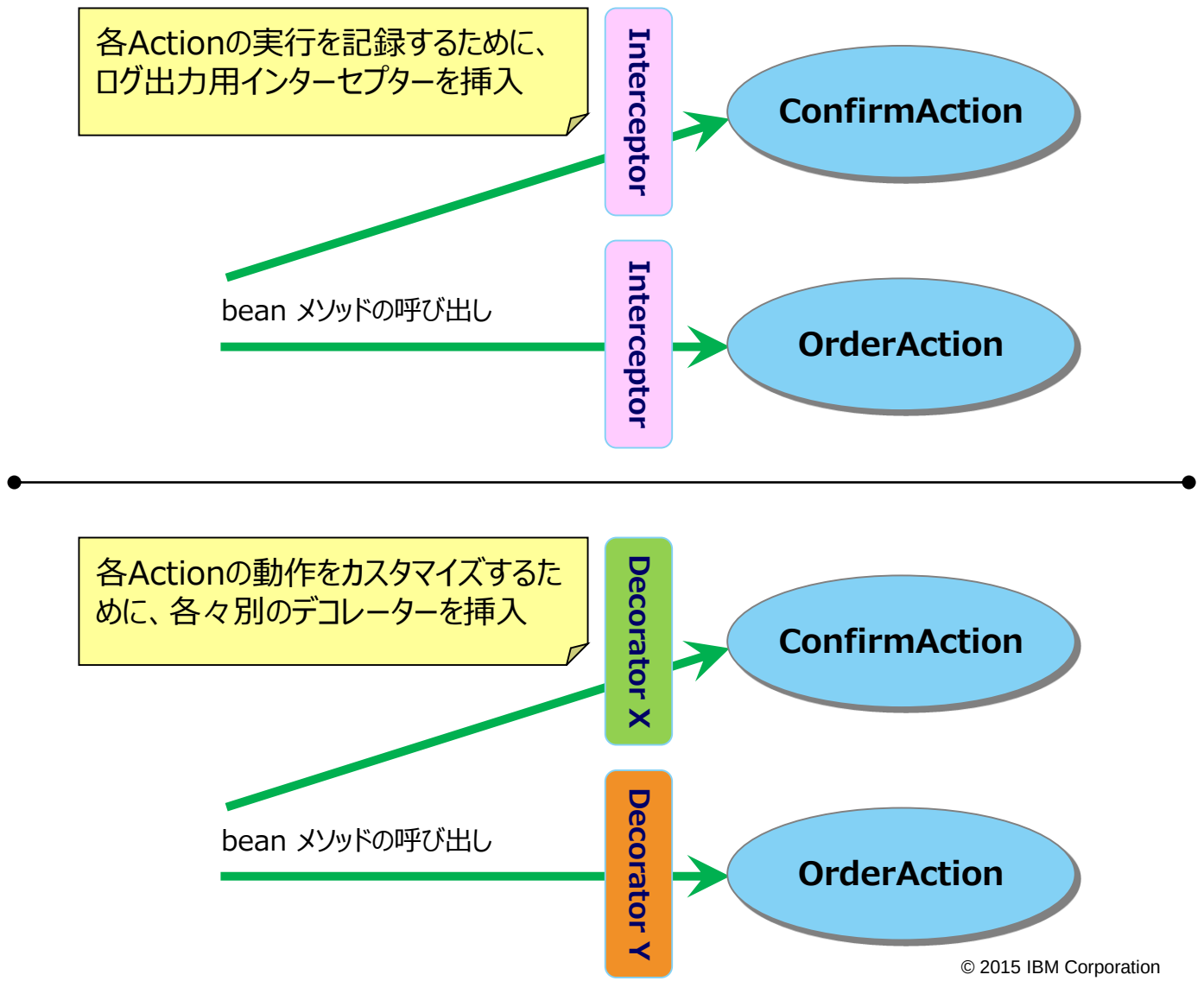
© 2015 IBM Corporation

インターセプター (Interceptor) と デコレーター(Decorator)

- 両者とも処理を差し込むために使用できるが、用途が異なる

- インターセプター (Interceptor)
  - メソッド呼び出しやライフサイクル・イベントが発生した時に、クロスカティング(横断的)タスクを差し込むための機能
  - AOP (Aspect Oriented Programming) で使用し、ビジネス・ロジックの実装には使用しない

- デコレーター(Decorator)
  - メソッド呼び出しが発生した時に、ビジネス・ロジックを差し込んだりカスタマイズするための機能
  - ビジネス・ロジックの実装に使用する



© 2015 IBM Corporation

インターセプター (Interceptor)

- メソッド呼び出しやライフサイクル・イベントが発生した時に、クロスカティング(横断的)タスクを差し込むための機能
  - AOP (Aspect Oriented Programming) で使用し、ビジネス・ロジックの実装には使用しない
  - ログ出力、監査、プロファイリングなど、アプリケーション内で繰り返し現れる処理を実装するために使用する

- ライフサイクルとコンテキスト
  - インターセプターは、関連付けられたオブジェクトと同じライフサイクルを持つ
  - メソッド呼び出しのインターセプターは、関連付けられたメソッドと同じコンテキストで呼び出される
  - ライフサイクル・イベントのインターセプターが呼び出されるコンテキストは規定されていない

- インターセプターも CDI 管理 Bean なので、@Inject で他の bean を注入できる

- CDI 1.1 以降では、ライフサイクル・イベントもインターセプター・クラスで処理できるように拡張された
  - CDI 1.0 では、前述のコールバック・メソッド(@PostConstruct/@PreDestroy)しかなかった

© 2015 IBM Corporation

メソッド呼び出しのインターセプト（１）

- インターセプターを使用する場合は、以下の流れで開発を行う
  1. インターセプター・バインディング・タイプを定義する
  2. インターセプターを実装する
  3. インターセプターを対象に関連付ける
  4. インターセプターを有効にする

- メソッド呼び出しのインターセプターを開発する流れを説明する

1. インターセプター・バインディング・タイプの定義
   - インターセプターと、その対象となるクラスやメソッドを関連付けるために使用するアノテーションを定義する
   - @InterceptorBinding を使用して定義

```
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.interceptor.InterceptorBinding;

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

インターセプター・バインディング・タイプの定義例

- インターセプター・バインディング・タイプ TimeLogging を定義している例
- @InterceptorBinding で、インターセプター・バインディングであることを宣言
- @Target で、TYPE と METHOD に付与できるアノテーションであることを宣言
- @Retention(RUNTIME) で、実行時までアノテーション情報が残ることを宣言
- @Inherited で、サブクラスにも継承されるように宣言

© 2015 IBM Corporation

メソッド呼び出しのインターセプト（２）

2. インターセプターの実装
   - インターセプター・バインディング・タイプのアノテーションと @Interceptor を付与
   - パラメーターなしの、public なコンストラクターを持つ必要がある
   - java.io.Serializable を実装 (*)

     (*) パッシベーション可能な bean (例:SessionScopedのbean) にバインドする場合に必要
   - インターセプター・メソッドを実装
     - @AroundInvoke を付与したメソッドを１つだけ実装
     - メソッドのシグネチャー：　Object メソッド名(InvocationContext) throws Exception

インターセプターの実装例

```
import javax.interceptor.*;
@Interceptor
@TimeLogging
public class TimeLoggingInterceptor implements java.io.Serializable {
    @AroundInvoke
    public Object writeLog( InvocationContext ctx ) throws Exception {
        System.out.println( new java.util.Date() );
        return ctx.proceed();
    }
}
```

・InvocationContext の proceed() を呼び出して処理を進め、その結果を返す

© 2015 IBM Corporation

メソッド呼び出しのインターセプト（３）

3. インターセプターを対象に関連付ける

– 対象とするクラスやメソッドにもインターセプター・バインディング・タイプのアノテーションを付与し、インターセプターと関連付ける

• クラスにインターセプター・バインディング・タイプのアノテーションを付与し、全てのメソッドにインターセプターをバインドする例

```
@TimeLogging
@SessionScoped
public class PaymentHandler implements Serializable {

    public String pay() {
        ......
    }

    public void reset() {
        ......
    }
}
```

```
@SessionScoped
public class PaymentHandler implements Serializable {

    @TimeLogging
    public String pay() {
        ......
    }

    public void reset() {
        ......
    }
}
```

• 特定のメソッドにだけインターセプターをバインドする例

© 2015 IBM Corporation

メソッド呼び出しのインターセプト（４）

4. インターセプターを有効にする

- beans.xml にインターセプターのクラスを登録
  - この方法で有効化した場合は、同じアーカイブ内の bean に対してのみインターセプターが有効になる
  - 複数のインターセプターがメソッドに関連付けられている場合は、beans.xml に登録されている順番で呼び出される

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...... >
    <interceptors>
        <class>jp.hoge.TimeLoggingInterceptor</class>
    </interceptors>
</beans>
```

- または、インターセプター・クラスに @Priority アノテーションを付与し、優先順位を指定
  - 複数アーカイブで構成されるアプリにおいて、アプリ全体でインターセプターが有効になる
  - 複数のインターセプターがメソッドに関連付けられている場合は、優先順位が小さいものから順番に呼び出される

```
import javax.interceptor.*;
import javax.annotation.Priority;
@Interceptor
@Priority(Interceptor.Priority.APPLICATION+10)
@TimeLogging
public class TimeLoggingInterceptor implements java.io.Serializable {
    ......
```


© 2015 IBM Corporation

補足：@Priority で指定する値

- インターセプターの優先順位は、Interceptor.Priority が提供する定数を利用して、以下のルールに従って指定する。値が小さいインターセプターから順番に呼び出される。

定数	値	意味
PLATFORM_BEFORE	0	• Java EE プラットフォームが定義するインターセプターで、呼び出しチェーンの早い段階で実行させたいものは、PLATFORM_BEFORE から LIBRARY_BEFORE の範囲のプライオリティーを使用する • 最も高いプライオリティーを持つ
LIBRARY_BEFORE	1000	• 拡張ライブラリーが定義するインターセプターで、呼び出しチェーンの早い段階で実行させたいものは、LIBRARY_BEFORE から APPLICATION の範囲のプライオリティーを使用する
APPLICATION	2000	• アプリケーションで定義するインターセプターは、APPLICATION から LIBRARY_AFTER の範囲のプライオリティーを使用する
LIBRARY_AFTER	3000	• 拡張ライブラリーが定義するインターセプターで、低い優先順位のものは、LIBRARY_AFTER から PLATFORM_AFTER の範囲のプライオリティーを使用する
PLATFORM_AFTER	4000	• Java EE プラットフォームが定義するインターセプターで、低い優先順位のものは、PLATFORM_AFTER より大きい値のプライオリティーを使用する

- 代替(Alternative)実装の場合は、値が最も大きい代替実装が使用される。

© 2015 IBM Corporation

@Interceptors を使用したインターセプターの関連付けと有効化

- @Interceptors アノテーションを使用すると、
  対象のクラスやメソッドにインターセプターを直接関連付け、有効化することができる
  - 複数のインターセプター・クラスを指定すると、指定した順番でインターセプターが呼び出される

```
@Interceptors( { TimeLoggingInterceptor.class, PerfLoggingInterceptor.class } )
@SessionScoped
public class PaymentHandler implements Serializable {
    ......
```

クラスに対して、２つのインターセプターを直接バインドする例

- 注意点
  - 簡便な方法だが、インターセプター・クラスの名前を指定するので柔軟性が低下する
  - beans.xml にインターセプターを登録すると、 @Interceptors で指定した順番は無視される

© 2015 IBM Corporation

ライフサイクル・コールバック・イベントのインターセプト

- 以下のライフサイクル・イベントをインターセプトできる
  - AroundConstruct
    - 対象のオブジェクトが作成され、注入が行われる前にインターセプターが呼び出される
    - 全てのインターセプターが InvocationContext の proceed() を呼び出した時点で、注入が行われる
  - PostConstruct および PreDestroy
    - 対応するライフサイクル・イベントが発生した時にインターセプターが呼び出される
- バインディング
  - クラスに対してバインドする必要がある
  - インターセプター・バインディング・タイプの定義では、@Target を TYPE のみにする
- ライフサイクル・イベントを処理するインターセプター・メソッドの実装
  - javax.interceptor パッケージ内の該当するアノテーションを付与する
  - ライフサイクル・イベントごとに１つだけメソッドを定義できる
    (同じイベントを処理するメソッドは複数定義できな)
  - メソッドのシグネチャー：
    void メソッド名(InvocationContext) または
    Object メソッド名(InvocationContext) throws Exception

```
@AroundConstruct
public Object init( InvocationContext ctx )
                                throws Exception {
    ......
    return ctx.proceed();
}
```


© 2015 IBM Corporation

InvocationContext

- インターセプター・メソッドに渡される InvocationContext を利用すると、より高度な処理を行える
  - proceed() メソッド
    - インターセプター・チェーンに登録されている次のインターセプターを呼び出す
  - contextData プロパティー
    - java.util.Map<String,Object> 型のオブジェクトで、同じメソッド呼び出しをインターセプトするインターセプター間で情報を共有する為に使用できる
  - parameters プロパティー
    - Object の配列で、メソッドに渡されるパラメーターに対応する
    - 必要に応じて、インターセプターが参照・変更できる
  - target プロパティー
    - インターセプターの対象となっているオブジェクトが取得できる

© 2015 IBM Corporation

デコレーター(Decorator)

- メソッド呼び出しが発生した時に、ビジネス・ロジックを差し込んだりカスタマイズするための機能

- デコレーターの実装
  - @Decorator アノテーションを付与
  - デコレートする bean と同じ Bean Type を持つ
    - デリゲーション・オブジェクトと同じ Bean Type を持つ
  - @Inject と @Delegate でデコレートする bean を注入
    - デリゲーション・オブジェクトを 1 つだけ注入
    - コンストラクターや初期化メソッドで注入することも可能
  - 全てのビジネス・ロジックを実装しなくても良い
    - abstract なクラスでも良い
  - デコレーターも CDI 管理 Bean なので、他の必要な bean を注入できる

デコレートされる bean

```
@Dependent
public class CoderImpl implements Coder {
    @Override
    public String codeString( String s, int v ) {
        .........
    }
    .........
}
```

- Bean Type が Coder の bean
- デリゲーション・オブジェクトはこの bean になる

デコレーター

```
import javax.decorator.*;

@Decorator
public abstract class CoderDecorator implements Coder {

    @Inject @Delegate
    Coder delegate;

    @Override
    public String codeString( String s, int v ) {
        String r = delegate.codeString( s, v );
        if ( r != null ) r = r.toUpperCase();
        return r;
    }
}
```

- ビジネス・メソッドをオーバーライド
- デリゲーション・オブジェクトのメソッドを流用

- Bean Type が Coder 用のデコレーターを定義

- @Inject @Delegate で Bean Type が Coder のデリゲーション・オブジェクトを注入

© 2015 IBM Corporation

デコレーター(Decorator)の有効化

- デコレーターはインターセプターと同様の方法で有効化できる

  – beans.xml にデコレーターのクラスを登録
    - この方法で有効化した場合は、同じアーカイブ内の bean に対してのみでデコレーターが有効になる
    - デコレーターは、beans.xml に登録されている順番で呼び出される

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ・・・・・・ >
    <decorators>
        <class>jp.hoge.CoderDecorator</class>
    </decorators>
</beans>
```

  – または、デコレーター・クラスに @Priority アノテーションを付与し、優先順位を指定
    - 複数アーカイブで構成されるアプリにおいて、アプリ全体でデコレーターが有効になる
    - デコレーターは、優先順位が小さいものから順番に呼び出される

  – インターセプターとデコレーターがメソッドにバインドされている場合、インターセプターが先に呼び出される

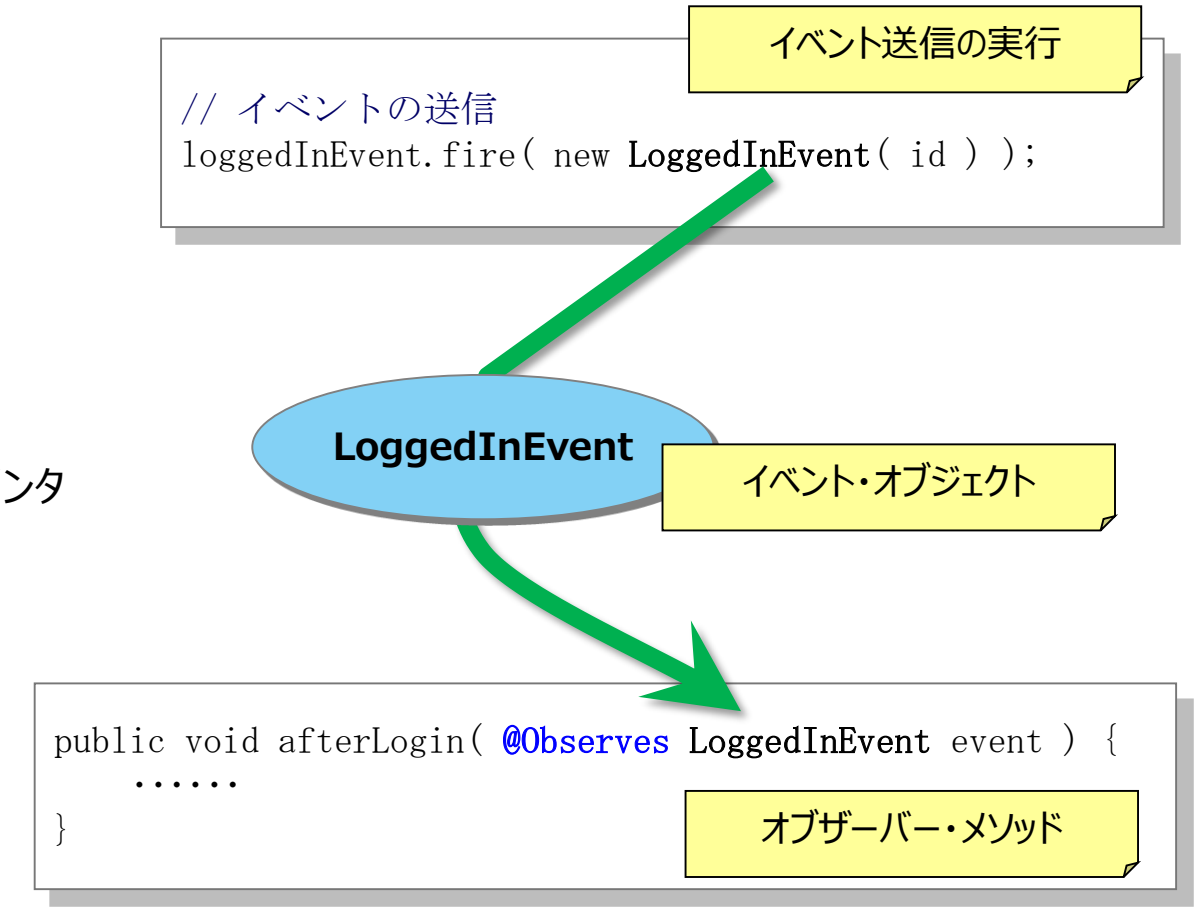
© 2015 IBM Corporation

9. イベント通知

© 2015 IBM Corporation

イベント通知

- bean 間でイベントの送信と受信を行う機能
  - 異なる層に配置された bean 間でも連携が可能となる
- 構成要素
  - イベント・オブジェクト
    - 通知される情報を含むオブジェクト
  - イベント・タイプ
    - イベントの型を表す
    - イベント・オブジェクトのクラス、スーパー・クラス、および、実装するインターフェースがイベント・タイプとなる
  - イベント限定子
    - イベントを限定するために使用される限定子
    - 同じタイプのイベントを区別するために使用できる
  - オブザーバー・メソッド
    - イベントを受け取るメソッド
    - 監視対象とするイベント・タイプのパラメータを持つ
    - このパラメーターには、必要に応じてイベント限定子を付与できる



© 2015 IBM Corporation

イベントの送信処理（１）

- CDI が提供する Event<?> を使用して送信を行う
  - Event<イベント・タイプ> のインスタンスを注入
    - javax.enterprise.event パッケージ
  - イベント・オブジェクトを引数に fire() メソッドを実行

イベント限定子なしの送信

```
@Inject
Event<LoggedInEvent> loggedInEvent;

public void login() {
    ......
    LoggedInEvent event = new LoggedInEvent( id );
    loggedInEvent.fire( event );
}
```

- イベント限定子を付与したい場合は、
  Evnent<?> の注入時に限定子を付与する

アノテーションを使用して２つのイベント限定子を付与する例

```
@Inject
@Admin @Updated
Event<LoggedInEvent> adminLoggedInEvent;

public void adminLogin() {
    ......
    LoggedInEvent event = new LoggedInEvent( id );
    adminLoggedInEvent.fire( event );
}
```


© 2015 IBM Corporation

イベントの送信処理（２）

- イベント限定子は、送信処理を行うコード内で動的に付与することも可能
  - イベントに付与する限定子を状況によって切替えたい場合などに使用

送信処理時に２つのイベント限定子を動的に付与する例

```
@Inject
Event<LoggedInEvent> loggedInEvent;

public void adminLogin() {
    ......
    class UpdatedQualifier extends AnnotationLiteral<Updated>  implements Updated {}
    class AdminQualifier   extends AnnotationLiteral<Admin>    implements Admin   {}
    ......
    LoggedInEvent event = new LoggedInEvent( id );
    loggedInEvent.select( new UpdatedQualifier(), new AdminQualifier() ).fire( event );
}
```

- UpdatedQualifier と AdminQualifier は、
  javax.enterprise.util.AnnotationLiteral<T> を使用して、
  限定子用の java.lang.annotation.Annotation の実装クラスとして宣言

© 2015 IBM Corporation

イベントの受信処理

- イベントの受信はオブザーバー・メソッドで行う
- オブサーバー・メソッド
  - bean の default-access, public, protected または private なメソッド (static メソッドでも良い)
  - イベント・パラメーター(@Observes アノテーションが付与されたパラメータ)を 1 つだけ持つ
    - パッケージ javax.enterprise.event
  - イベント・パラメーターのタイプが監視対象のイベント・タイプとなる
  - イベント・パラメーターにイベント限定子を付与できる
  - その他のパラメーターを持つ場合は、呼び出し時に CDI コンテナーが注入する
    - javax.enterprise.inject.spi.EventMetadata を注入すると、通知されたイベントのメタデータにアクセスできる
    - メタデータからイベント・オブジェクトの型、イベントに付与されている限定子、イベントの送信元の情報を取得できる

```
public void afterLogin( @Observes @Admin @Updated LoggedInEvent event, EventMetadata metadata ) {
    ......
}
```

2 つのイベント限定子が付与された LoggedInEvent を監視対象とし、イベント・メタデータの注入を受けるオブザーバー・メソッドの例

- 以下の条件を満たすオブザーバー・メソッドにイベントが通知される
  - イベント・パラメーターのタイプが、送信されたイベントに適合する
  - イベント・パラメーターに付与された限定子が、送信されたイベントに付与されている限定子のサブセットである

© 2015 IBM Corporation

Conditional なオブザーバー・メソッドと Transactional なオブザーバー・メソッド

- 通常のオブザーバー・メソッド
  - @Observes、または、@Observes(notifyObserver=Reception.ALWAYS) が付与された オブザーバー・メソッド
  - bean のコンテクスチュアルなインスタンスが存在しない場合は、インスタンスが作成され、オブザーバー・メソッドが呼び出される
    - スコープが @Dependent の bean の場合は、受信の都度、インスタンスが作成される

- Conditional なオブザーバー・メソッド
  - @Observes(notifyObserver=Reception.IF_EXISTS) が付与されたオブザーバー・メソッド
  - bean のコンテクスチュアルなインスタンスが存在する場合にだけ、オブザーバー・メソッドが呼び出される
    - スコープが @Dependent の bean は、Conditional なオブザーバー・メソッドを持てない

- Transactional なオブザーバー・メソッド
  - @Observes(during=TransactionPhase.??????) で以下のフェーズを指定したオブザーバー・メソッド
    - BEFORE_COMPLETION, AFTER_COMPLETION, AFTER_FAILURE, AFTER_SUCCESS
  - イベントが送信された時点ではオブザーバー・メソッドは呼び出されず、during で指定した JTA トランザクションの終了フェーズでオブザーバー・メソッドが呼び出される
  - 但し、JTA トランザクションの外でイベントが送信された場合は、即座にオブザーバー・メソッドが呼び出される

© 2015 IBM Corporation

オブザーバー・メソッドの呼び出し

- オブザーバー・メソッドの呼び出しは、実装方法やスコープなどに依存し、以下のようになる
  - オブザーバー・メソッドが static メソッドの場合は、それが呼び出される
  - オブザーバー・メソッドが static メソッドではない場合は、bean のコンテクスチュアルなインスタンスのオブザーバー・メソッドが呼び出される
  - 但し、 Conditional なオブザーバー・メソッドの場合は、該当のスコープがアクティブで、コンテクスチュアルなインスタンスが存在する場合にだけ、呼び出される

- 複数のオブザーバー・メソッドが呼び出される場合、それらが呼び出される順序は規定されていない

- オブザーバー・メソッドが例外をスローした場合
  - Transactional なオブザーバー・メソッド　　イベントの通知処理は継続、コンテナーによってキャッチされログ出力される
  - その他のオブザーバー・メソッド　　イベントの通知処理は中止、fire() メソッドが例外をリスローする

- オブザーバー・メソッドが呼び出されるコンテキスト
  - イベント送信が行われた(fire()メソッドが実行された)コンテキストで実行される
  - 但し、BEFORE_COMPLETION 以外が指定された Transactional なオブザーバー・メソッドでは、トランザクションのコンテキストは規定されていない

© 2015 IBM Corporation

10. ステレオタイプ

© 2015 IBM Corporation

ステレオタイプ (Stereotypes)

- アプリの開発では、同じ特性や役割(ロール)を持つ bean を多量に開発することになる
  - 特に、アーキテクチャー・パターンを適用すると、特性や役割に応じて bean が何種類かに大別され、種類ごとに同じメタデータを付与することになる
- ステレオタイプを使用すると、共通のメタデータをカプセル化することができる

- ステレオタイプで定義できるメタデータ
  - スコープ（デフォルトのスコープ）
  - インターセプターのバインド(関連付け)
  - デフォルトの bean 名
  - 代替(alternatives)実装の宣言
  - 他のステレオタイプ

- 禁止事項
  - 名前を明示的に指定した @Named はステレオタイプに指定しない
  - @Named 以外の限定子はステレオタイプに指定しない
  - @Typed はステレオタイプに指定しない

© 2015 IBM Corporation

ステレオタイプ (Stereotypes) の定義と利用

- ステレオタイプは @Stereotype アノテーションを使用して定義
  - javax.enterprise.inject パッケージ
  - @Target で TYPE, METHOD, FIELD など指定

- bean へのステレオタイプの付与は他のアノテーションと同様

```
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.enterprise.inject.Stereotype;

@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

ステレオタイプの定義例

- ステレオタイプ Action を定義している例
- @Stereotype で、ステレオタイプであることを宣言
- @Target で、TYPE に付与できるステレオタイプであることを宣言
- @Retention(RUNTIME) で、実行時まで情報が残ることを宣言
- @Inherited は指定していない（サブクラスには継承されないように宣言）

ステレオタイプを付与した例

```
@Action
public class LoginAction {
    ......
}
```


© 2015 IBM Corporation

ステレオタイプ (Stereotypes) の定義例と利用例

- 定義例

スコープ、インターセプター・バインディング、および、デフォルトの名前を指定したステレオタイプの例

```
@RequestScoped
@TimeLogging
@Named
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

代替実装を指定したステレオタイプの例

```
@Alternative
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Mock {}
```

- 利用例
  - bean には複数のステレオタイプを付与できる
  - ステレオタイプで指定されているスコープや bean 名は、上書きできる

```
@Action
public class LoginAction {
    ......
}
```

- リクエスト・スコープを持つ
- インターセプター TimeLogging をバインド
- デフォルトの bean 名 loginAction を付与

```
@Action @Mock
public class MockAction {
    ......
}
```

- @Action に加えて @Mock を付与
- 更に代替実装として扱われる

```
@Action
@SessionScoped
@Named("login")
public class LoginAction {
    ......
}
```

- スコープと bean 名を上書きする例

© 2015 IBM Corporation

組み込みのステレオタイプ (Stereotypes)： @Model

- @Model
  - javax.enterprise.inject パッケージ
  - MVC 構造のアプリケーションにおいてモデルの役割を担う bean に付与するために組み込みで提供されている

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Model {}
```

@Model の定義内容

© 2015 IBM Corporation

11. 検出

© 2015 IBM Corporation

CDI コンテナーによる bean の検出

- アーカイブ(jar, war)内の bean は、bean discovery mode に従って検出される
  - 詳細は、概要の「bean discovery mode」を参照
- 特定のクラスや特定のパッケージ内のクラスを検出対象から除外することが可能
  - bean discovery mode が all の時に、特定のクラスやパッケージを bean として検出させたくない場合などに使用
  - 他のコンテナーが管理するオブジェクト(例:JPAのエンティティーなど)を CDI に検出させたくない場合などに使用
- 以下の２つの方法で除外できる
  - @Vetoed アノテーション
    - javax.enterprise.inject パッケージ
  - 除外フィルター(Exclude Filter)
    - beans.xml で指定

© 2015 IBM Corporation

@Vetoed アノテーション

- @Vetoed をクラスやパッケージに付与することで除外する

- bean クラスに付与する例
  - 付与されたクラスのみが検出されなくなる

```
@Vetoed
@Entity
public class Empolyee {
    ......
```

bean クラスのソース・ファイル

- パッケージに付与する例
  - 付与されたパッケージ直下の全てのクラスが検出されなくなる
  - bean の検出対象となるパッケージを限定することで、アプリの起動時間を短縮したい場合も有用

```
@javax.enterprise.inject.Vetoed
package hoge.noncdi;
```

package-info.java ファイル

© 2015 IBM Corporation

除外フィルター (Exclude Filter)

- beans.xml に除外フィルターを追加することでも除外できる

beans.xml の例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ··· bean-discovery-mode="all">
    <scan>
        <exclude name="com.acme.rest.*" />
        <exclude name="com.acme.faces.**">
            <if-class-not-available name="javax.faces.context.FacesContext"/>
        </exclude>
        <exclude name="com.acme.verbose.*">
            <if-system-property name="verbosity" value="low"/>
        </exclude>
        <exclude name="com.acme.ejb.**">
            <if-class-available name="javax.enterprise.inject.Model"/>
            <if-system-property name="exclude-ejbs"/>
        </exclude>
    </scan>
</beans>
```

- 指定したパッケージ内の全クラスを除外する例
- 指定したクラスが存在しない場合にのみ、指定したパッケージとそのサブ・パッケージ内の全てのクラスを除外する例
- 指定したシステム・プロパティーが指定した値の場合にのみ、指定したパッケージ内の全クラスを除外する例
- 指定したクラスが存在し、かつ、指定したシステム・プロパティーが定義されている場合にのみ、指定したパッケージとそのサブ・パッケージ内の全てのクラスを除外する例
- name にクラス名を指定し、特定のクラスのみ除外することも可能

12. まとめ

© 2015 IBM Corporation

まとめ

当ガイドでは Java EE 7 アプリケーション設計ガイドとして、CDI 1.2 について解説を行いました

CDI 1.2

- タイプ・セーフなインジェクション(注入)機能を提供します
- ライフサイクルを持つ bean の管理を CDI コンテナーに任せることができます
- 代替実装や特殊化の機能を使用することで、ソースを(ほとんど)修正せずに、bean を差し替えることができます
- インターセプターを使用すると、クロスカティング(横断的)タスクを簡単に差し込むことができます
- ビジネス・ロジックを差し込みたい場合は、デコレーターが利用できます
- bean 間でのイベント通知機能が提供されています
- アーカイブ内のクラスは、bean discovery mode に従って検出されます

© 2015 IBM Corporation

参考資料

- JSR 346: Contexts and Dependency Injection for Java EE 1.1
  - https://jcp.org/en/jsr/detail?id=346
- JSR 346: Contexts and Dependency Injection for Java EE 1.2
  - https://jcp.org/aboutJava/communityprocess/mrel/jsr346/index.html
- JSR 330: Dependency Injection for Java
  - https://jcp.org/en/jsr/detail?id=330
- Java EE 7 Specification APIs
  - http://docs.oracle.com/javaee/7/api/
- The Java EE 7 Tutorial
  - http://docs.oracle.com/javaee/7/tutorial/index.html
- IBM Knowledge Center - WebSphere Application Server Liberty Core 8.5.5
  - http://www-01.ibm.com/support/knowledgecenter/SSD28V/welcome

© 2015 IBM Corporation