ORACLE®

# Java EE 6 の詳細について

日本オラクル株式会社　Fusion Middleware 事業統括本部
シニア Java エバンジェリスト
寺田　佳央
ブログ：http://yoshio3.com

以下の事項は、弊社の一般的な製品の方向性に関する概要を説明するものです。また、情報提供を唯一の目的とするものであり、いかなる契約にも組み込むことはできません。以下の事項は、マテリアルやコード、機能を提供することをコミットメント（確約）するものではないため、購買決定を行う際の判断材料になさらないで下さい。オラクル製品に関して記載されている機能の開発、リリースおよび時期については、弊社の裁量により決定されます。

# Java EE 6の概要

## 2009年12月正式リリース

# Java EE 6
# 今後は本番環境へ

**WebLogic**

# Java EE 6 のメインテーマ
## 開発生産性の大幅な向上

- 拡張性
- プロファイルの提供
- 仕様の削減
- 進化するかんたん開発


# 拡張性

- Java EE 以外のフレームワークも容易に利用可能
  - Spring, Struts, Wicket 等 3rd パーティフレームワークを利用可能
  - 複雑な設定は不要
    - 追加するフレームワークの Servlet、Servlet フィルタ、コンテキストリスナーは自動検知、自動登録
  - フレームワーク毎の設定項目は web fragment 設定ファイルに集約

# プロファイル

- Java EEの技術を用途毎に分割して提供
  - Java EEのサブセットを提供
- 独自プロファイルの開発が可能
  - 例：電話会社向けプロファイル
- Java EE 6で最初に提供されるプロファイル
  - Webプロファイル(Webの開発に特化)
  - Enterprise Platform(フルJava EE)


# Web プロファイル

## Webアプリケーションの開発に特化した軽量プロファイル

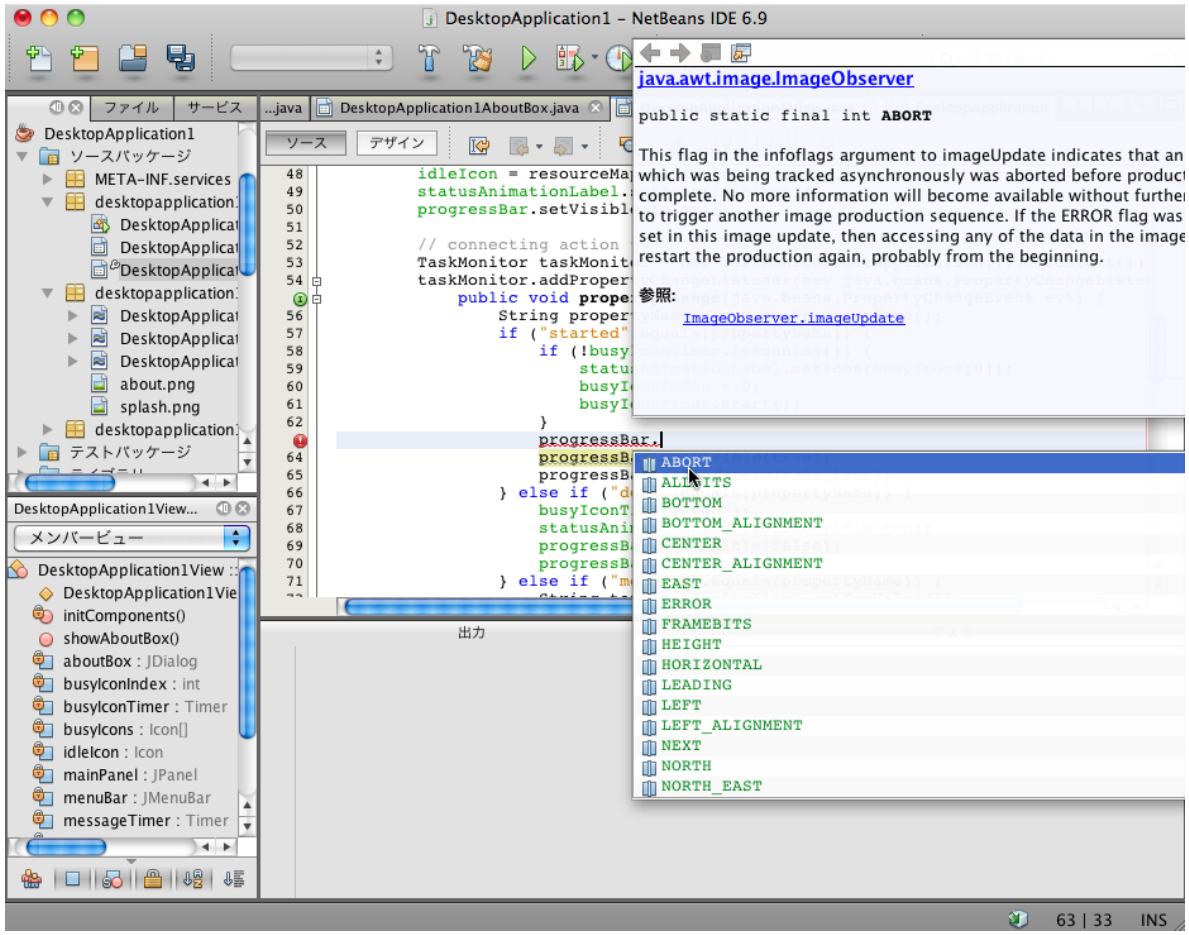
- Webプロファイルに含まれる技術
  - Servlet 3.0
  - JSP 2.2 / EL
  - JSTL
  - JSF 2.0
  - Bean Validation 1.0
  - EJB 3.1 Lite
  - JPA 2.0
  - JTA 1.1
  - DI 1.0/CDI 1.0
  - Managed Beans 1.0
  - Interceptors 1.1
  - Common Annotations

# 仕様の削減
## Pruning

- 2段階プロセス
  - 古く使われなくなったAPIの整理
  - コンポーネントのオプション化
  - 次期バージョン (Java EE 7) で削除される可能性
    - JAX-RPC(->JAX-WS)
    - EJB Entity Beans(->JPA)
    - JAXR
    - JSR-88

# かんたん開発

## 開発を容易にする様々な統合開発環境




ORACLE JDEVELOPER

NetBeans

eclipse

JBuilder

IntelliJIDEA
The most intelligent Java IDE around

# Managed Bean 1.0

# Interceptor 1.1

# Managed Bean 1.0

- Java EE の Managed Bean
  - JSF/JMX の Managed Bean とは異なる
  - @javax.annotation.ManagedBean を指定するだけ (JSR-250)
- コンテナによって管理されるオブジェクト
  - 最低限必要な要件を満たす
  - アプリケーションクライアントコンテナ/Web コンテナ/EJB コンテナ
- POJO による実装
  - 軽量なコンポーネントモデル
- 基本的なサービスを提供
  - リソースの注入 (@Resourece, @Inject …)
  - ライフサイクル管理 (@PostConstruct, @PreDestroy)
  - Interceptors による AOP 開発 (@Interceptor, @AroundInvoke)

# Managed Bean 1.0

- 他の Java EE コンポーネントのベースコンポーネント
  - EJB, CDI 等は Managed Bean として定義されており、Managed Bean の機能を利用可能
- 呼び出し元と同一スレッドで実行
- 代表的なアノテーション
  - @Resource：リソースの宣言
  - @PostConstruct : DI 完了後、初期化を実装する際に利用
  - @PreDestroy：コンテナからインスタンスが破棄される際に実行
  - …
  - 詳細は JSR-250 をご参照

# Managed Bean 1.0 の例

JSR-250 Java EE
Common Annotation

```
@javax.annotation.ManagedBean
@Stateless
public class SomeBean {
    @Resourece
    private Detasouce ds;

    @PostConstruct
    public void init(){ … }

    @PreDestroy
    public void destroy(){ … }
}
```

EJB,CDIではアノテーションを宣言しなくても利用可能

リソースの注入

ライフサイクル（初期化時）

ライフサイクル（破棄時）

他のコードから上記 ManagedBean を利用する場合 @Resource で注入

```
@Resource SomeBean sbean
```

```
@Inject SomeBean sbean
```

# Interceptors 1.1 の使用
# AOP、ログ、監査、プロファイルに便利

- 実際の処理を行うメソッドの呼び出しをインターセプトし別の処理を挿入
- ターゲットクラスに対し1つ以上のバインディングが必要

```
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.METHOD;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Inherited
@javax.interceptor.InterceptorBinding
@Retention(java.lang.annotation.RetentionPolicy.RUNTIME)
@Target({METHOD, TYPE})
public @interface MyInterceptorBinding {
}
```

# Interceptors 1.1 の使用
# 自身のインターセプタクラスの作成

```
import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;

@Interceptor
@MyInterceptorBinding
public class MyInterceptor {

    @AroundInvoke
    public Object intercept(InvocationContext context) throws
Exception {
        System.out.println("before interception”);
        Object result = context.proceed();//次処理の呼び出し
        System.out.println("after interception”);
        return result;
    }
}
```

# 実装したインタセプタの適用

```
import javax.interceptor.Interceptors;

@javax.annotation.ManagedBean
@Stateless
@Interceptors(MyInterceptor.class)
public class SomeBean {
    @Resourece
    private Detasouce ds;

    @PostConstruct
    public void init(){ … }

    @PreDestroy
    public void destroy(){ … }
}
```

実行例：

```
情報: INIT was called
情報: before interception
情報: EJB INIT
情報: after interception
情報: DESTROY was called
```

# Servlet 3.0

ORACLE

# Servlet 3.0
## JSR-315

- 特徴
  - 設定ファイル(web.xml)のオプション化
  - アノテーションベースの設定
  - 拡張性
    - web-fragments.xml の提供
  - マルチパート対応
    - ファイルアップロード
  - 非同期 Servlet のサポート
  - セキュリティ(login/logout処理に対応)

Java SE 5の言語仕様で新たに追加されたアノテーションを使用し宣言的プログラミングモデルを採用。またジェネリクスの利用も可能

# Servlet 3.0
# かんたん開発

- アノテーションを利用した宣言的プログラミングモデル
  - web.xml のオプション化

| アノテーション | 用途 |
|---|---|
| @WebServlet | Servletの定義 |
| @WebFilter | フィルタの定義 |
| @WebListener | リスナの定義 |
| @WebInitParam | パラメータの定義 |
| @ServletSecurity | セキュリティの制約 |
| @MultipartConfig | ファイルアップロード |

アノテーションの設定はweb.xmlで上書き可能

# Servlet 3.0 のサンプル
## アノテーションを利用したプログラミング

```
package hello;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet(name="Hello", urlPatterns={"/Hello"})
public class Hello extends HttpServlet {

    @Override
    protected void doGet(HttpSe
                         HttpS
throws ServletException, IOExcep
        .......
    }
}
```

```
<web-app>
 <servlet>
   <servlet-name>Hello</servlet-name>
   <servlet-class>hello.Hello</servlet-class>
 </servlet>
 <servlet-mapping>
   <servlet-name>Hello</servlet-name>
   <url-pattern>/Hello/* </url-pattern>
 </servlet-mapping>
</web-app>
```

**web.xml は不要**

# Servlet 3.0
## 拡張性

- モジュール型開発
  - 静的コンテンツ・イメージ・JSP 等のリソースを外出し可能
  - WEB-INF/lib 配下の jar ファイルに META-INF/resources を作成
  - コンテキストルートでアクセス可能

例：WEB-INF/lib/resources.jar

```
> jar cvf resources.jar META-INF/
        META-INF/MANIFEST.MF
        META-INF/resources/
        META-INF/resources/test.html
```

例：http://www.oracle.com/test.html

# Servlet 3.0
## 拡張性

- 3rd パーティフレームワークの追加が容易
  - 設定ファイルによる拡張
  - web-fragments.xml
  - 外部ライブラリ、フレームワークをweb.xmlとは別の設定ファイルで管理
  - ライブラリの呼び出し順を設定可能
- プログラムによる拡張
  - サーブレット、フィルタ等をサーブレット初期化時に設定可能

# Servlet 3.0
## 設定ファイルによる Web コンテナの拡張

- 単一の web.xml ファイルに記載すると可読性の低下、運用保守性の低下
- フレームワーク毎に設定を別ファイルで管理可能
- web-fragment.xmlで外部フレームワークの設定を管理

```
<web-app version="3.0"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
metadata-complete="false">
```

web.xml の設定

metadata-complete の設定が true の場合、アノテーションが利用できなくなる他、web-fragment.xmlに記載したリソースの自動検出ができなくなるので注意

# Servlet 3.0
# web-fragment.xml の設定例

```
<?xml version="1.0" encoding="UTF-8"?>
<web-fragment xmlns=http://java.sun.com/xml/ns/javaee
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
 http://java.sun.com/xml/ns/javaee/web-fragment_3_0.xsd"
 version="3.0">
    <context-param>
    </context-param>
    <servlet>
    </servlet>
    <servlet-mapping>
    </servlet-mapping>
    <session-config>
    </session-config>
    <welcome-file-list>
    </welcome-file-list>

</web-fragment>
```

記載する内容は、web.xml に記載する内容と同様

# Servlet 3.0
# プログラムによる Web コンテナの拡張

- ServletContainerInitializer
  - インタフェースを実装しコンテナを拡張
  - 起動時に1度だけ実装クラスを検知
- @HandlesTypes
  - ServletContainerInitializerがハンドル可能な型を宣言
- ServletContext
  - 動的にservlet/filterの登録等が可能

# Servlet 3.0
# プログラムによる Web コンテナの拡張例

- コンテナ拡張コードは META-INF/services へ配置

```
@HandlesTypes(WebService.class)

class JAXWSServletContainerInitializer implements
ServletContainerInitializer{

  public void onStartup(Set<Class<?>> c, ServletContext ctx)
throws ServletException {
    // 実行時の初期化やマッピング等の JAX-WS 固有の設定等をここで実装
    ServletRegistration reg = ctx.addServlet("JAXWSServlet",
                                "com.sun.webservice.JAXWSServlet");
    reg.addServletMapping("/foo");
   }
}
```

# Servlet 3.0
# マルチパート対応

```
 <FORM action="/MyFileUpload" enctype="multipart/form-data" method="POST">
ファイル名: <INPUT type="file" name="content">
          <INPUT type="submit" value="Submit">
 </FORM>
```

- <INPUT TYPE="file" name="fname"> でマルチパートフォームを記載
- HttpServletRequest#getPart("fname")でマルチパートデータを取得
  - フォーム名に一致するデータを取得
- HTTPヘッダよりファイル名を取得
  - Content-Disposition:
  - form-data; name="content"; filename="FILE_NAME"

# Servlet 3.0
## マルチパート対応

```
@WebServlet(name = "MyFileUpload", urlPatterns ={"/MyFileUpload"})
@MultipartConfig(fileSizeThreshold=5000000,
                 MaxFileSize=10000000, location="/tmp")
public class MyFileUpload extends HttpServlet {
   protected void processRequest(HttpServletRequest
request, HttpServletResponse response)
   throws ServletException, IOException {
   // <INPUT type="file" name="content">より取得
   Part part = request.getPart("content");
   // Content-Disposition ヘッダを解析し
   // ファイル名の取得
   String fname = getFilename(part);
   // @MultipartConfig(location="/tmp") で設定したディレクトリ配下に保存
   part.write(fname);
```

# Servlet 3.0
# 非同期処理のサポート

- 非同期リクエスト処理用のプログラムモデル
  - 例：Comet, chat, push apps
- 長時間処理が必要なリクエストに有効
  - スレッドを別処理に再利用可能
  - スレッドを1クライアント毎に保持する必要無し
- Async用アノテーションを追加
  - @WebServlet( asyncSupported=true )

# Servlet 3.0
# 非同期サーブレット

```
@WebServlet(name="CalculatorServlet", asyncSupported=true,
                          urlPatterns={"/calc", "/getVal"})
public class CalculatorServlet extends HttpServlet{
    public void doGet(HttpServletRequest req,
HttpServletResponse res) {
    ...
          AsyncContext aCtx = req.startAsync(req, res);
      }
   }
```

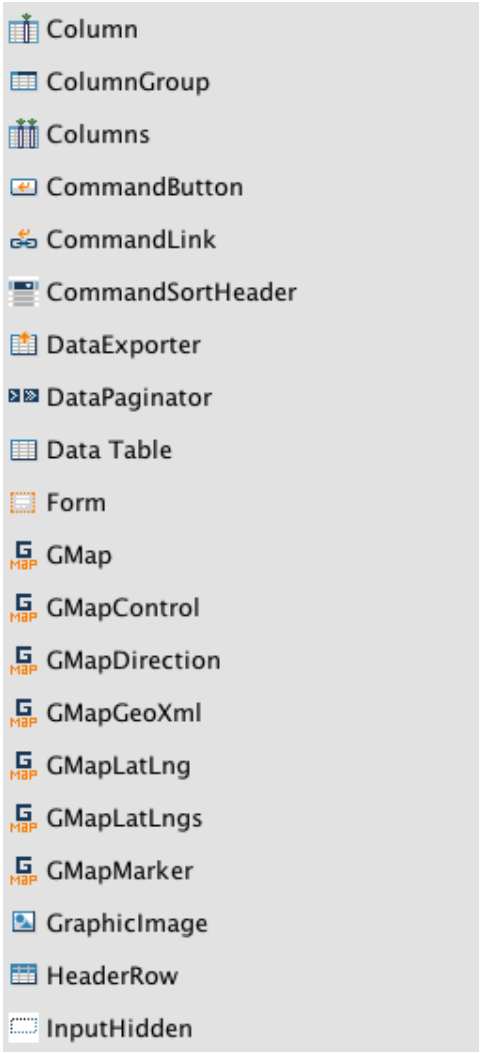
# Servlet 3.0
# セキュリティの拡張

- 新しい認証方式を提供
  - <form action="j_security_check" method="post">の代わりに使用
- 認証/ログイン/ログアウト機能の提供
  - HttpServletRequest.authenticate
  - HttpServletRequest.login
  - HttpServletRequest.logout
- アノテーションによる宣言
  - @ServletSecurity

# JSF 2.0

# JavaServer Faces 2.0


- コンポーネントベース開発 (MVC)
  - Facelets による実装
  - テンプレート機能の提供
  - カスタム複合コンポーネント
  - Ajax 対応
  - ブックマーク可能なページ
- 設定項目の簡略化
  - ページナビゲーションの改良
  - faces-config.xml のオプション化
- Bean Validation のサポート
- Servlet コンテナのバージョンに非依存
  - Servlet 3.0 コンテナの他 2.5 上でも動作可能



# JSF 2.0
# テンプレート機能

- 共通構成要素の再利用
  - 開発効率の向上
  - 再利用性の向上
  - 可読性の向上
  - 運用・保守性の向上


ご参照： http://yoshio3.com/2011/01/14/jsf20-new-with-facelets-template/

# JSF 2.0
# テンプレート機能

- テンプレート
  - 変更可能箇所を <ui:insert name=“”> で記載

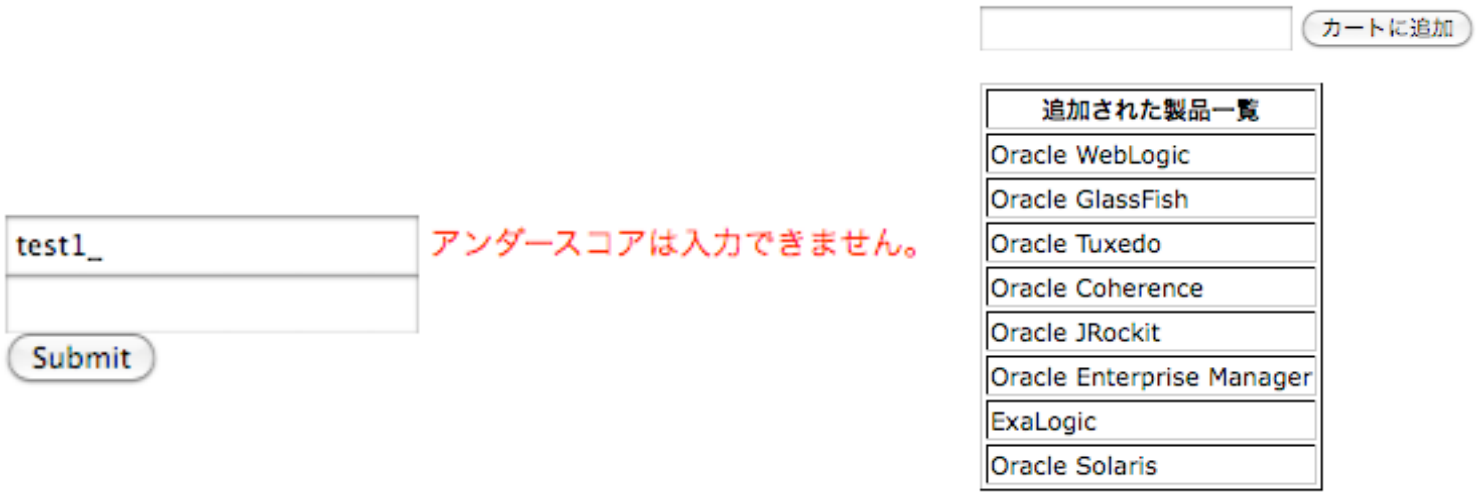
```
<ui:insert name="content">Content</ui:insert>
```

- テンプレートクライアント（テンプレートを利用する側）
  - <ui:composition> で該当のテンプレートの読み込み
  - 変更箇所を <ui:define name=“”> で上書き

```
<ui:composition template="./templates/tableTemplate.xhtml">
<ui:define name="content">
 ここにページ毎の内容を記述
</ui:define>
</ui:composition>
```

# JSF 2.0
# Ajax 対応

- JSF の仕様中で JavaScript API が定義
  - jsf.ajax.request …
  - JSF から Ajax 呼び出しがかんたん
  - コンポーネントのサーバ側の処理と、クライアント側のレンダリング処理の制御




ご参照： http://yoshio3.com/2011/01/18/jsf-20-ajax-support/

# JSF 2.0
# Ajax 対応

- <f:ajax> タグを使用したかんたん開発
  - JavaScript の知識不要で Ajax 化を実現
  - 既存の JSF コードに対して数行で Ajax を実現

```
01 <h:form prependId="false">
02     <h:inputText id="productname" value="#
   {cart.product.productname}">
03     </h:inputText>
04     <h:commandBut
05         <f:ajax execute="productname" render="tables"/>
06     </h:commandButton>
07     <br/>
08     <br/>
09
10     <h:dataTable border="1" id="tables" value="#
   {cart.selectedItems}" var="item">
11         <f:facet name="header">
12             <h:outputText value="追加された製品一覧" />
13         </f:facet>
14         <h:column>
15             <h:outputText value="#{item.productname}" />
16         </h:column>
17     </h:dataTable>
18 </h:form>
```

たった１行のコード追加で Ajax を実現(JavaScriptの知識不要)

# JSF 2.0
# ブックマーク可能な URL

- <h:link> <h:button>
- リクエスト URL に対するパラメータの受け渡し
  - outcome 属性に直接記載（画面遷移時に使用）
    - <h:link outcome="viewEntry?query=10&result=5" value="リンク" />
  - View Parameters の使用（アクセス時に使用）
    - http://localhost:8080/pathToWebapp/view.xhtml?data=foobar
    - <f:viewParam id="data" name="data" value="#{someBean.data}" />

# JSF 2.0
# ブックマーク可能な URL

- リクエスト URL に対するパラメータの受け渡し
  - f:param タグの使用（画面遷移時に使用）
  - 画面遷移時にパラメータを付加し遷移したい場合
  - 例：

    http://localhost:8080/JSF-Bookmark/faces/viewEntry.xhtml?getParam=getValue

```
<h:link outcome="viewEntry" value="リンク">
    <f:param name="getParam" value="getValue" />
</h:link>
```

ご参照：　http://yoshio3.com/2011/02/16/jsf-bookmarkable-url-suppor/

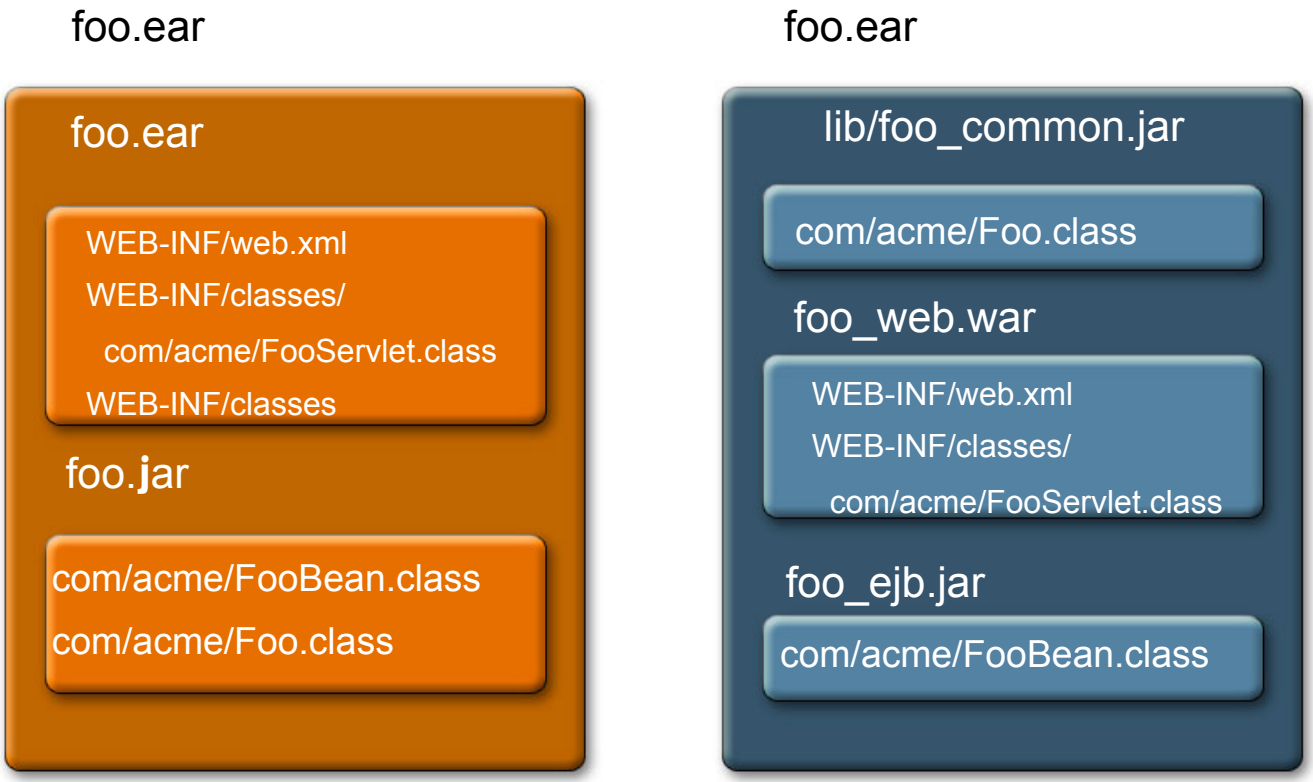
# EJB 3.1

# EJB 3.1
## JSR-318

- 特徴
  - パッケージの簡略化
  - EJB 3.1 “Lite” の提供
  - ローカルビジネスインタフェースのオプション化
  - 標準化された Global JNDI名
  - Java SEに組み込み可能なEJBコンテナ
  - その他の新機能

# EJB 3.1 –パッケージングの簡略化

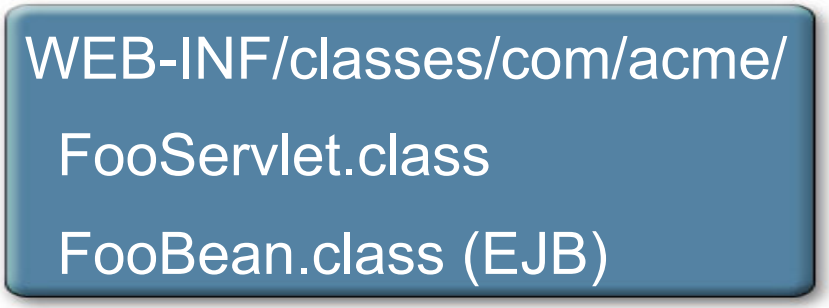
## Java EE 5 までのパッケージング




適切なアーカイブファイル（ear,war,jar)へパッケージ化が必要

面倒なパッケージング/作業負担が大

# EJB 3.1 –パッケージングの簡略化
## Java EE 6 のパッケージング




- かんたんなパッケージング
  - EJBをwarファイルへ含める事が可能
  - WEB-INF/classes:ファイルとして
  - WEB-INF/lib: 分割jarファイルとして
- 今まで同様のパッケージ化も可能
  - ejb-jarファイル
- 配備記述子はオプション
  - 記載する必要がある場合別途 WEB-INF/ejb-jar.xmlへ記述可能

# EJB Lite
## 軽量版の提供(Web Profileで利用可)

Full EJB 3.1機能のサブセットを提供

- Lite
  - ローカルセッションBeans
  - CMT/BMT
  - Declarative Security
  - Interceptors
- Full = Lite +
  - Message-Driven Beans
  - Web Service Endpoint
  - 2.x/3.x Remote view
  - RMI-IIOP Interoperability
  - Timer Service
  - Async method call
  - 2.x Local view
  - CMP/BMP Entity

# EJB 3.1
## Local Business Interface のオプション化

EJB の定義と実装

```
public interface Hello {
    public String sayHello();
}
```

クライアントの実装

```
@EJB
private Hello h;

...

h.sayHello();
```

```
@Stateless
public class HelloBean implements
                                Hello {
    public String sayHello()
      { return "hello"; }}
```

インタフェース定義が必須だった

# EJB 3.1
# 標準化された Global JNDI 名

- Java EE 5 まで
  - Global JNDI名はアプリケーションサーバ提供ベンダー独自に設定
  - Global JNDI名を直接記載したコードは移植性が損なわれていた
  - 他製品へ移行する際JNDI名の変更が必要
- Java EE 6 から
  - JNDI 名が標準化
  - Global JNDI 名
    - java:global[/<app-name>]/<module-name>/<ejb-name>
  - アプリケーション内の JNDI 名
    - java:app/<module-name>/<ejb-name>
  - モジュール定義内の JNDI 名
    - java:module/<ejb-name>

# EJB 3.1
## 標準化された Global JNDI 名

```
@Stateless
public class HelloBean implements Hello {
       public String sayHello(String msg) {
         return “Hello “ + msg;
  }
}
```

**JNDI NAME:**

***java:global/hello/HelloBean***

***java:app/hello/HelloBean***

***java:module/HelloBean***

# EJB 3.1
# 組み込み可能な EJB コンテナ

- EJB 3.0まで
  - EJBコンポーネントの単体テストは困難
  - Remote Facade/Web Tierで強制実行
  - サーバ／クライアントで別プロセスの稼働
- EJB 3.1から
  - Java SEにEJBコンテナを組み込むことが可能
  - JUnit等で同一Javaプロセス内でテストが可能

# EJB 3.1
## 組み込み可能 EJB コンテナ

```
package test;
import javax.ejb.Stateless;
@Stateless
public class Hello {
    public String sayHello(){
        return "Hello Embedded TEST";
    }
}
```

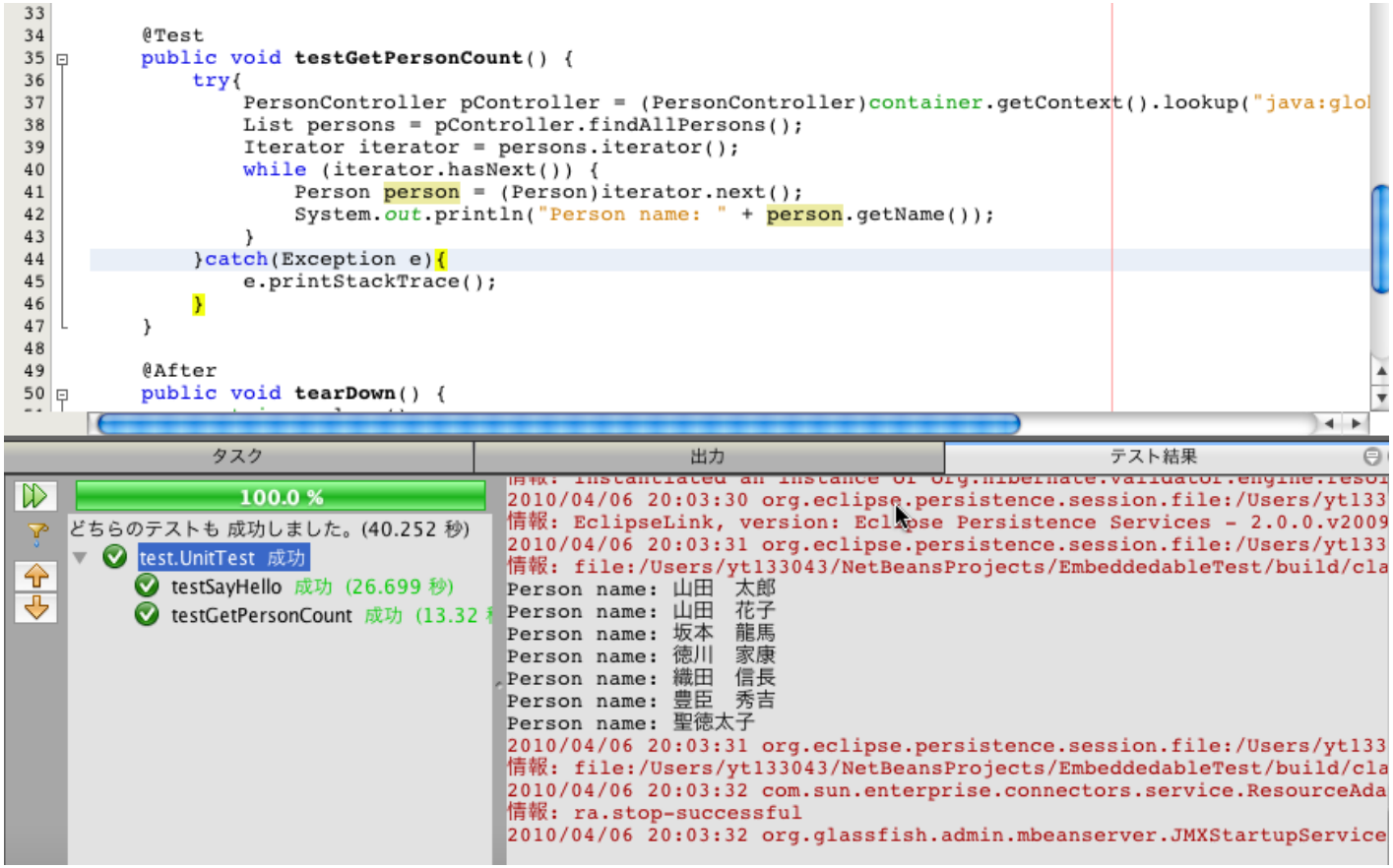
# EJB 3.1
# 組み込み可能 EJB コンテナ

```
@Test
public void testSayHello() {
    Map p = new HashMap();
    p.put("org.glassfish.ejb.embedded.glassfish.instance.root",
    "/Applications/GlassFish/glassfishv3-webprofile/glassfish/
    domains/domain1");
    EJBContainer container = EJBContainer.createEJBContainer(p);
    try{
        Hello hello = (Hello)container.getContext().lookup
            ("java:global/classes/Hello");
        System.out.println(hello.sayHello());
    }catch(Exception e){
        e.printStackTrace();
 }}
```

ご参照： http://bit.ly/hzzjR3

# EJB 3.1
# 組み込み可能 EJB コンテナ

- JUnit の単体テスト



# EJB 3.1
# 組み込み可能 EJB コンテナ

- GlassFish を使用する場合に必要なライブラリ

- javax.ejb.jar
  - EJBContainerクラスを含むライブラリ
- glassfish-embedded-static-shell.jar
  - 組み込み可能なGlassFishライブラリ

# EJB 3.1
## その他の新機能

- Singleton Session Beansの追加
- 並列アクセス処理対応
- Startup / Shutdownコールバック機能の追加
- タイマーサービス
- 自動タイマー生成/カレンダーベースタイマー
- 非同期処理

# EJB 3.1

## Singleton Session Bean

- アプリケーションで唯一のインスタンス
- コンテナの停止時にインスタンスは破棄
- 並列アクセスをサポート
- Singleton Session Beansの初期化
- 複数のSingletonコンポーネントが存在する場合、コンポーネントの初期化順を規定したい場合、@DependsOnを使用

# EJB 3.1

## Sigleton Session Bean

```
@Singleton
public class SharedBean {
  private SharedData
shared;
  @PostConstruct
  private void init() {
    shared = ...;
  }
  public int getXYZ() {
    return shared.xyz;
  }
```

```
@Stateless
public class FooBean {
  @EJB
  private SharedBean
shared;

  public void foo() {
    int xyz =
shared.getXYZ();
    …
  }
}
```

# EJB 3.1
## Singleton Session Bean

- コンテナ管理(デフォルト)
  - @ConcurrencyManagement(CONTAINER)
  - メソッドにメタデータを記述しアクセス制御(ロック)の指定が可能
  - @Lock(READ): 複数アクセスからの読み込みが可能
  - @Lock(WRITE): 書き込みは単一アクセス
- ビーン管理
  - @ConcurrencyManagement(BEAN)
  - 開発者による状態の同期処理、インスタンスのアクセス処理制御が可能
  - synchronized,volatileの利用が可能
- 両方同時の使用は不可能

# EJB 3.1
## Singleton Session Bean

```
Startup/Shutdown コールバック

@Singleton
@Startup (アプリケーションが起動する前に EJB コンテナが実行)
public class StartupBean {

  @PostConstruct
  private void onStartup() { … }

  @PreDestroy
  private void onShutdown() { … }

}
```

# EJB 3.1
# タイマーサービス

- EJBコンテナが提供するサービス
  - アノテーションでスケジュールを定義
    - @Schedule：単一メッソッドに対するタイマースケジュールを定義
    - @Schedules：単一メソッドに対して複数の@Scheduleを定義する場合に使用
- カレンダー表記でタイマー定義可能
  - 毎日、毎時、毎分に実行
    - (minute="*", hour="*", timezone="Asia/Tokyo")
  - 午前9時～午後5時、30分毎
    - (minute="0,30", hour="9-17")

# EJB 3.1
# タイマーサービス

```
@Stateless
public class BankBean {
  @PersistenceContext EntityManager accountDB;
  @Resource javax.mail.Session mailSession;
 //毎月最終日午前8時にsendMonthlyBankStatements()を実行
  @Schedule(hour="8", dayOfMonth="Last")
  void sendMonthlyBankStatements() {
    ...
  }
}
```

※注意

@Scheduleはvoid型のメソッドに適用可能

@Timeout or @Schedule method must return void and be a no-arg method or take a single javax.ejb.Timer param

# EJB 3.1
# 非同期処理

- アノテーションでかんたんに非同期処理を定義可能
  - @Asynchronous：非同期処理の定義
  - クラスレベル：クラスに含まれる全メソッドの非同期を定義
  - メソッドレベル：特定のメソッド単位で非同期を定義
- 非同期メソッド定義における返り値の設定
  - void型：非同期処理で値を返さなくてもよい場合
  - Future<V>：非同期処理結果を返す必要がある場合
    - <V>: 結果の型

# EJB 3.1
# 非同期処理

```
@Stateless public class DocBean {
  @PersistenceContext EntityManager resultsDB;
  @EJB DocBean myself;
  public void processDocument(Document document) {
    myself.doAnalysisA(document);
    myself.doAnalysisB(document);
  }
  @Asynchronous
  public void doAnalysisA(Document d) {...}
  @Asynchronous
  public void doAnalysisB(Document d) {...}
```

# JPA 2.0

# JPA 2.0 : JSR-317
## EJB の仕様から独立

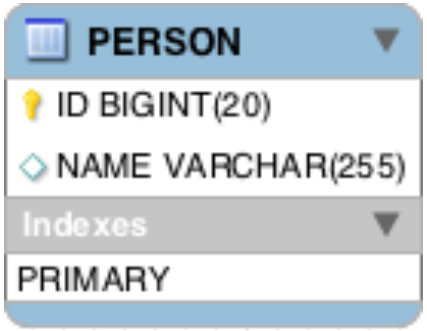
- JPA 2.0 = JPA 1.0 + α
  - モデリングの強化
  - JPQL新しい構文の追加
  - Criteria API の提供
  - メタモデルAPIの提供
  - 悲観的ロックの追加
  - バリデーションのサポート
  - 設定オプションの標準化

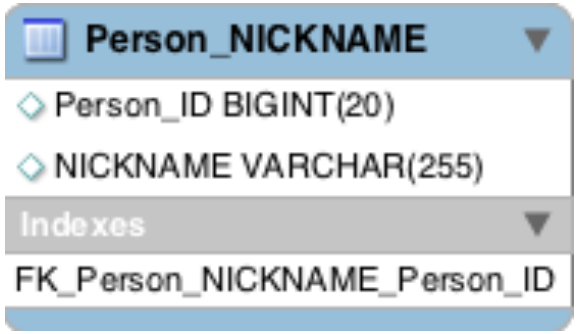
# JPA 2.0
# モデリングの強化

- コレクションのサポート
  - 基本型のコレクション
  - Embeddableのコレクション
  - 多段レベルの組み込み
  - 関連性を持った組み込み
  - Mapサポートの強化
  - 順序付けリストのサポート

# JPA 2.0
## 基本型のコレクション

```
@Entity
public class Person implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    @ElementCollection
    private Set<String> nickname;
```



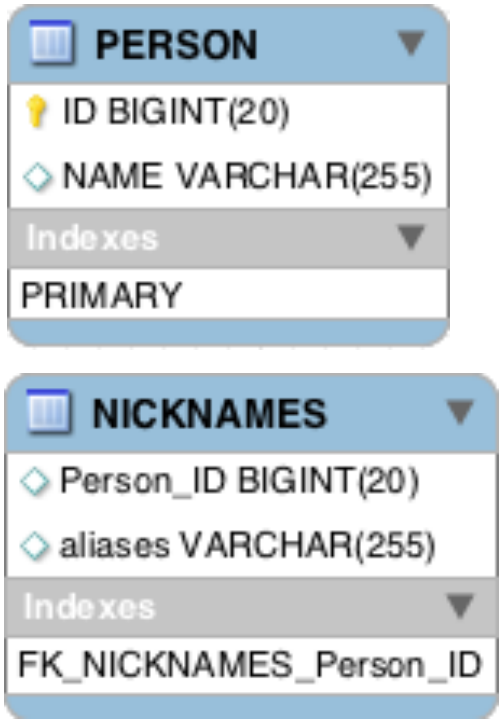




デフォルト:

EntityName_FIELDNAME テーブルとマップ

# JPA 2.0
## 基本型のコレクション

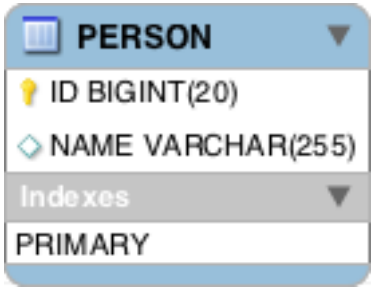
```
@Entity
public class Person implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    @ElementCollection
    @CollectionTable(name="NICKNAMES")
    @Column(name="aliases")
    private Set<String> nickname;
```

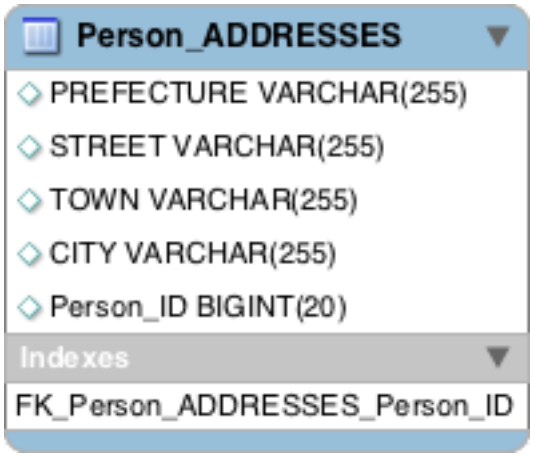



マップするテーブル名とカラム名の指定が可能

# JPA 2.0
## 組み込み可能型コレクション

```
@Embeddable
public class Address{
    private String street;
    private String town;
    private String city;
    private String prefecture;
}
@Entity
public class Person {
    …
    @ElementCollection
    private Set<Address> addresses;
}
```

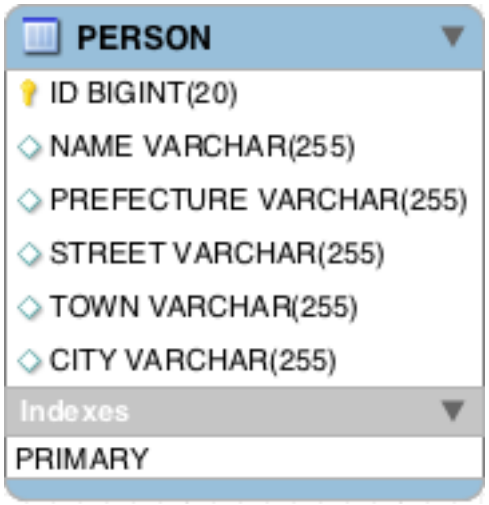



EntityName_FIELDNAME テーブルとマップ(名前は変更可能)

# JPA 2.0
## 多段レベルの組み込み

```
@Embeddable
public class ContactInfo {
    @Embedded
    Address addresses;
}
@Entity
public class Person {
    @Id
    private Long id;
    private String name;
    private ContactInfo contactInfo;
```

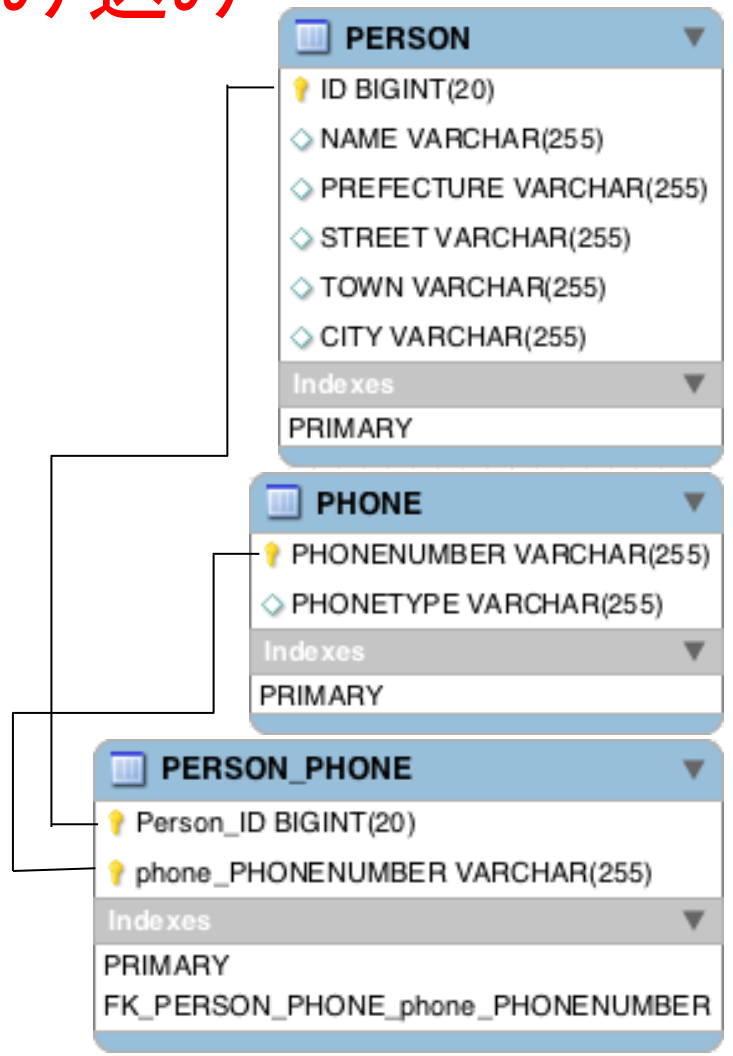



EntityName_FIELDNAME テーブルとマップ(名前は変更可能)

Person_ADDRESS テーブルは未作成

# JPA 2.0
## 関連性を持つテーブルの組み込み

```
@Embeddable public class ContactInfo {
    @Embedded Address addresses;
    @OneToMany
    Set<Phone> phone;
}
@Entity public class Person {
    @Id
    private Long id;
    private String name;
    private ContactInfo contactInfo;
}
@Entity
public class Phone {
    @Id String phonenumber;
    String phonetype;
}
```



# JPA 2.0

## Map の Collection

- JPA1.0のMap
  - 限定的に使用可能
  - Entityの属性をMapのキーとして利用,値はEntity
- JPA 2.0のMap
  - 柔軟性が向上
  - キー、値共にBasic型, Embeddable型, Entity型の使用が可能

# JPA 1.0
## @MapKey のみ利用可能

- Entity 内の属性を Map キーとして利用
- value は Entity オブジェクト

```
@Entity public class Person{
    @OneToMany(mappedBy="owner")
    @MapKey(name="phone_type")
    private Map <String, PhoneNumber> phoneNumbers;
    …
@Entity
public class PhoneNumber{
    @Basic @Id private String phone_type;
    @ManyToOne private Person owner;
```

# JPA 2.0

## JPA 2.0 で利用可能な Map の種類

| Map の種類 | Mapping | Key のアノテーション | Value のアノテーション |
|---|---|---|---|
| Map<Basic,Basic> | @ElementCollection | @MapKeyColumn<br>@MapKeyEnumerated<br>@MapKeyTemporal | @Column |
| Map<Basic, Embeddable> | @ElementCollection | @MapKeyColumn<br>@MapKeyEnumerated<br>@MapKeyTemporal | Embeddable でマップ<br>@AttributeOverride<br>@AssociationOverride |
| Map<Basic, Entity> | @OneToMany,<br>@ManyToOne | @MapKeyColumn<br>@MapKeyEnumerated<br>@MapKeyTemporal | Entity でマップ |
| Map<Embeddable, Basic> | @ElementCollection | Embeddable でマップ<br>@AttributeOverride | @Column |
| Map<Embeddable, Embeddable> | @ElementCollection | Embeddable でマップ | Embeddable でマップ<br>@AttributeOverride<br>@AssociationOverride |

# JPA 2.0

## JPA 2.0 で利用可能な Map の種類

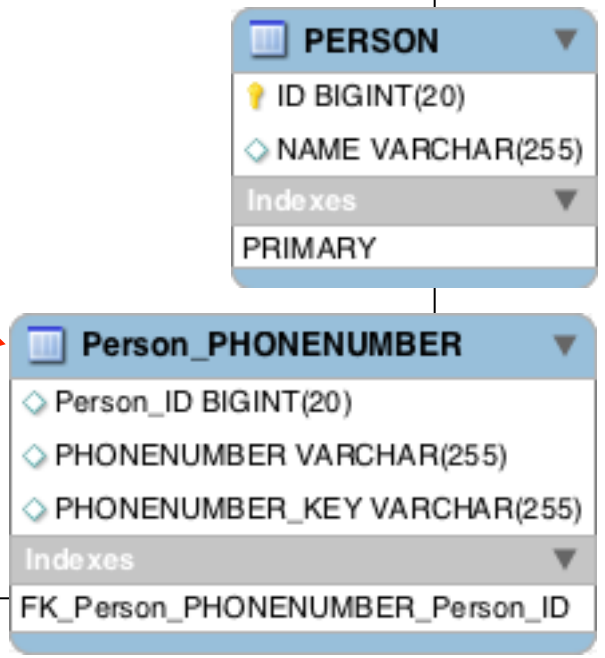
| Map の種類 | Mapping | Key のアノテーション | Value のアノテーション |
|---|---|---|---|
| Map<Embeddable, Entity> | @OneToMany<br>@ManyToOne | Embeddable でマップ | Embeddable でマップ<br>@AttributeOverride<br>@AssociationOverride |
| Map<Entity, Basic> | @ElementCollection | @MapKeyJoinColumn | Entity でマップ |
| Map<Entity, Embeddable> | @ElementCollection | @MapKeyJoinColumn | Embeddable でマップ<br>@AttributeOverride<br>@AssociationOverride |
| Map<Entity, Entity> | @OneToMany<br>@ManyToOne | @MapKeyJoinColumn | Entity でマップ |

# JPA 2.0

## Map のコレクション: Basic 型のキー

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;

    @ElementCollection
    @CollectionTable(name="**")
キー:@MapKeyColumn(name="**")
    @Column(name="**")
    private Map<String,String>
phoneNumber;
```




それぞれ name を指定し名前を変更可能

# JPA 2.0

## Map のコレクション: Enumeration のキー

```
public class Person implements Serializable {
    @Id private Long id;

    @ElementCollection
    @CollectionTable
    @MapKeyEnumerated(EnumType.STRING)
    @MapKeyColumn
    @Column
    private Map<PhoneType,String> phoneNumber;
}
public enum PhoneType{Mobile, Home, Company}
```

キーで扱う文字列の数が少ない場合、

Stringの代わりにEnumeratedを使用し効率化可能

# JPA 2.0
## 検索結果のソート

```
@Entity
Public class Project {
    ....
    @OneToMany
    @OrderBy("lastname DESC(昇順),phonenumber ASC(降順)")
    private List<Person> person;
}
@Entity
public class Person {
    ....
    private String lastname;
    private String phonenumber;
}
```

@OrderBy:検索結果から動的にオーダを生成

# JPA 2.0
# 検索結果のソート

- @OrderColumn
  - インサート、アップデート、デリート時等において恒久的に永続化
  - OneToMany,ManyToManyで利用可能

```
@Entity
public class CreditCard {

    @Id long ccNumber;
    @OneToMany  // unidirectional
    @OrderColumn
    List<CardTransaction> transactionHistory;
    ...
}
```

# JPA 2.0

## Automatic Orphan Deletion

- DBレコードの連鎖削除（カスケード削除）
  - 関連性のあるエンティティが削除された際、孤児となったエンティティを自動削除
  - @OneToOne, @OneToManyに適用可能
  - cascade=CascadeType.REMOVEは不必要

```
@Entity public class Order {
  @Id int orderId;
  ...
  @OneToMany(cascade=PERSIST, orphanRemoval=true )
  Set<Item> lineItems;
```

# JPA 2.0

## Java Persistence Query Language

- 新しく追加されたモデリングとマップ機能に対応
  - Select Listの操作機能
  - CASE,COALESCE,NULLIF表現の追加
  - Restricted polymorphism
  - IN表現におけるコレクション-値パラメータ

# JPA 2.0
## JPQL の追加演算子

- INDEX
  - ORDERED LISTで使用
- KEY, VALUE, ENTRY
  - MAPで使用
- CASE, COALESCE, NULLIF
  - CASE表現で使用
- TYPE
  - Restricted Polymorphismで使用

# JPA 2.0

## OrderList で使用する INDEX

```
SELECT t  FROM CreditCard c JOIN c.transactions t
WHERE c.cardHolder.name = 'John Doe'
  AND INDEX(t) < 10
```

# JPA 2.0
## MAP で使用する KEY, VALUE

```
// Inventory is Map<Movie,Integer>

SELECT v.location.street, KEY(i).title, VALUE(i)
FROM VideoStore v JOIN v.inventory i
WHERE KEY(i).director LIKE '%Hitchcock%'
  AND VALUE(i) > 0
```

# JPA 2.0
# ANSI SQL-92 CASE Expression 対応

- 検索結果を変更したい場合、テーブル定義の変更ができない場合に有効
- CASE
  - CASE{WHEN <cond_exp> THEN <scalar_expr>} + ELSE <scalar_exp> END
- COALESCE
  - COALESCE(<scalar_exp> {,<scalar_expr>} + )
- NULLIF
  - NULLIF(<scalar_expr1>, <scalar_expr2>)

# JPA 2.0
# CASE 式の利用例

## 給料の昇級額の計算

```
UPDATE Employee e
SET e.salary =
  CASE e.rating
     WHEN 1 THEN e.salary * 1.2
     WHEN 2 THEN e.salary * 1.1
     ELSE e.salary * 1.02
  END
```

社員の評価レートが1の場合基本給の1.2倍

社員の評価レートが2の場合基本給の1.1倍

それ以外は、1.02倍

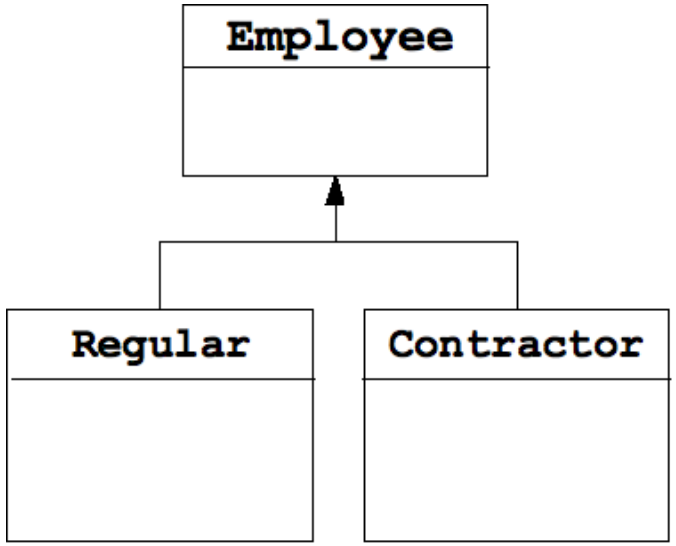
# JPA 2.0

## TYPE Expression

- 継承したEntityの内、
- 特定のEntityの取得が可能

```
SELECT e
FROM Employee e
WHERE TYPE(e) IN (Contractor)
```

TYPEにはJavaのEntityクラスを指定



# JPA 2.0
# Criteria API

- クエリーの制御をJavaオブジェクトで制御
  - JPA1.0ではJPQL/Native SQLでクエリーを記載
  - JPQLで可能なことはプログラム上で実現可能
- Criteria APIはクエリー実行結果の型の安全性が向上
  - クエリー実行におけるランタイムエラーの発生率が減少
  - 統合開発環境の型チェックを利用し安全に実装可能
- JPQLはSQLに精通した開発者に有効
  - 統合開発環境が自動的に雛形を生成（開発効率の向上）
- 開発者に選択肢を提供
  - オブジェクトベース、文字列ベースのクエリーを提供

ORACLE

# JPA 2.0
# Criteria API のインタフェース

- CriteriaBuilder :
  - CriteriaQueryオブジェクトを生成するファクトリ
  - EntityManager#getCriteriaBuilder()より取得
- CriteriaQuery :
  - クエリーの実行結果の型を返す
  - <T> CriteriaQuery<T> createQuery(Class<T> resultClass)
- Root :
  - クエリーの範囲指定
- その他
- Join, ListJoin, MapJoin, Path, Subqueryなど

# JPA 2.0

## Criteria API の使用例と対応するJPQL文

```
EntityManager em = … ;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Person> p = cb.createQuery(Person.class);
 Root<Person> person = p.from(Person.class);
 p.select(person).where(
                cb.equal(person.get(“name”),”Taro Yamada”));
```

対応する JPQL 文

SELECT p FROM Person p WHERE p.name = 'Taro Yamada'

# JPA 2.0
## Metamodel API

- 強力な型チェックを可能
  - ランタイムエラーの排除
    - 文字列によるエンティティの属性へのアクセスを排除
  - コンパイル時に属性の有無、型チェックを実施
- Metamodelクラスの生成
  - _を付加した正規化したMetamodelクラスを生成
  - IDEで自動生成（※NetBeans は Ver 7.0 より）
- プログラム上で動的に生成

# JPA 2.0

## Metamodel API を使用しない場合

```
EntityManager em = … ;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Person> p = cb.createQuery(Person.class);
Root<Person> person = p.from(Person.class);
p.select(person).where(
                cb.equal(person.get(“name”),”Taro Yamada”));
```

“name”が存在しない場合、もしくは型が異なる(String型でない)
場合ランタイムエラーが発生

# JPA 2.0

## Metamodel クラスとは

```
@Entity
public class Person {
    @Id
    @GeneratedValue
(strategy =
     GenerationType.AUTO)
    private Long id;
    private String name;
    Private Integer age;
}
```

Entity クラス

```
import
javax.persistence.metamodel.Singul
arAttribute;
@javax.persistence.metamodel.Stati
cMetaModel(Person.class)
public class Person_ {
  public static volatile
SingularAttribute<Person,Long> id;
  public static volatile
SingularAttribute<Person,String>
name;
  public static volatile
SingularAttribute<Person,Integer>
age;
```

Metamodel クラス

# JPA 2.0
# 動的に Metamodel クラスを取得

プログラム中で動的に Metamodel クラスを生成

```
EntityManager em = ...;
Metamodel meta = em.getMetaModel();
EntityType<Person> Person_ = meta.entity(Person.class);
```

# JPA 2.0
## Metamodel クラスを使用した実装例

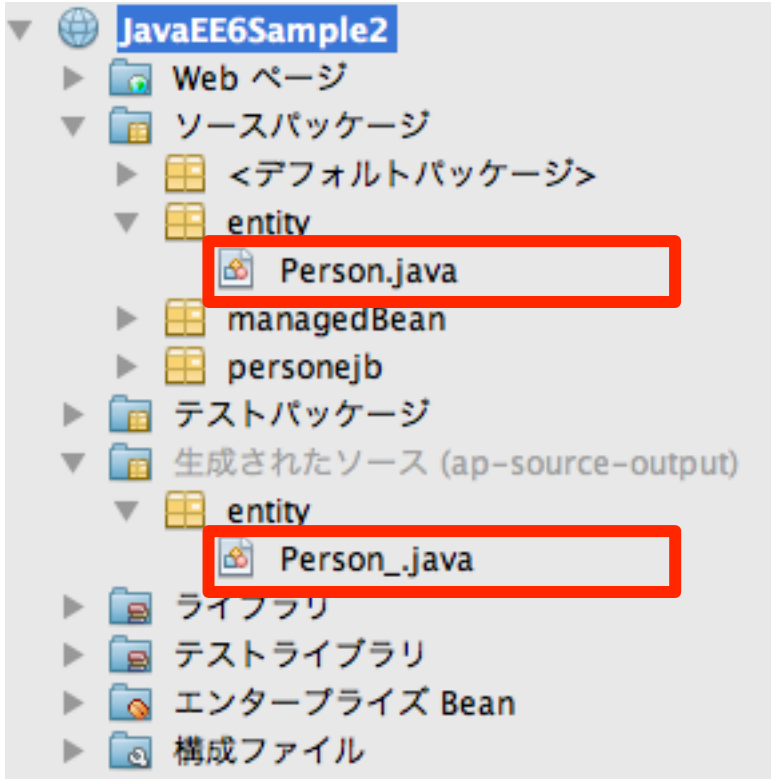
```
EntityManager em = … ;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Person> p = cb.createQuery(Person.class);
Root<Person> person = p.from(Person.class);
p.select(person).
  where(cb.equal(person.get(Person_.name),"Taro Yamada"));
```

解決：Person_.nameの有無、型チェックをコンパイル時に可能

（IDEの強力な型チェックでランタイムエラーを抑制）

# JPA 2.0
## Metamodel クラスの自動生成

- NetBeans 7.0 における自動生成
  - プロジェクトの構築により自動生成

# JPA 2.0
# 悲観的ロックの追加

- JPA1.0
  - 楽観的ロック
  - 操作している情報は他が操作する可能性が少ない場合に利用
- JPA 2.0
  - 悲観的ロック
  - 操作している情報を他が操作する可能性がある場合に利用
  - 用途を明確にして使用しなければパフォーマンスに悪影響

ORACLE

# JPA 2.0
# 悲観的ロックの使用

```
em.lock(person, LockModeType.PESSIMISTIC_WRITE);
```

- LockModeType.PESSIMISTIC_READ
  - 共有ロック
- LockModeType.PESSIMISTIC_WRITE
  - 排他ロック
- トランザクションが存在しない場合
  - TransactionRequiredException をスロー

# JPA 2.0

## JPA に統合された Bean Validation

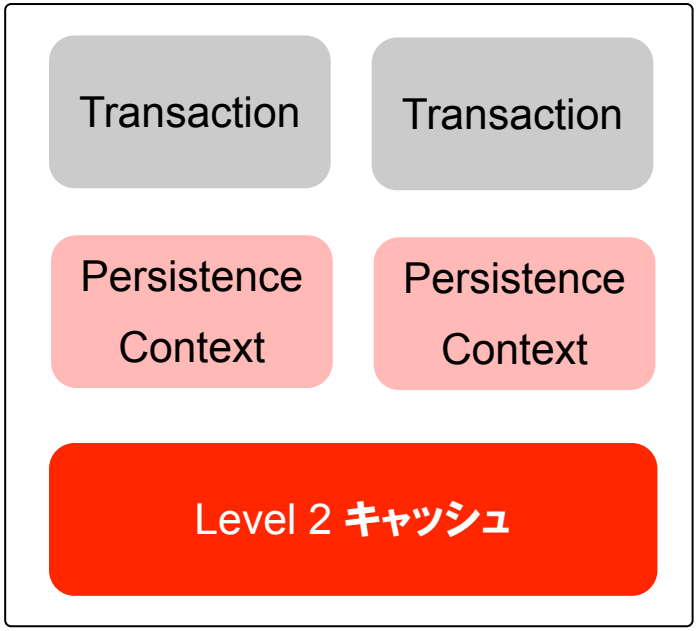
エンティティレベルでバリデーションを実現

```
@Entity public class Employee {
  @Id Integer empId;
  String name;
  @Max(15) Integer vacationDays;
  @Valid Address worksite;
  ...
}
@Embeddable public class Address {
  @Size(max=30) String street;
  @Size(max=20) String city;
  @Zipcode String zipcode;
  ...
}
```

JPA Entity クラレベルでのバリデーションを実現

# JPA 2.0
## 2次(共有)キャッシュのサポート

- JPA 1.0(1次キャッシュ)
  - 単一のPersistence Contextのエンティティに対するキャッシュを利用可能
  - EntityManagerがエンティティを管理
- JPA 2.0(2次キャッシュ)
  - JPA 1.0ではベンダが独自に提供していたがJPA 2.0で標準化
  - 複数のPersistence Contextのエンティティのキャッシュを共有し利用可能
  - EntityManagerFactoryからCache APIを利用可能



# JPA 2.0
## 2次(共有)キャッシュのサポート

- 利点
  - 一度読み込んだエンティティへのアクセスが高速/DB負荷軽減
  - 読み込みが多く、修正が頻繁ではないエンティティへのアクセスに有効
- 欠点
  - 大量のオブジェクトを扱う場合、メモリを大量に消費
  - 更新されたデータに対して古いデータを参照する場合がある
  - 並列書き込み時に考慮が必要
  - スケーラビリティが低下する可能性があり

# JPA 2.0
# 2次(共有)キャッシュのサポート

- キャッシュ動作設定
  - persistence.xml の <shared-cache-mode> の設定

| 設定 | 動作 |
|---|---|
| ALL | 全てキャッシュ<br>(EclipseLink の場合デフォルト値) |
| NONE | キャッシュの無効 |
| ENABLE_SELECTIVE | @Cacheable(true) のエンティティのみキャッシュ |
| DISABLE_SELECTIVE | @Cacheable(false) のエンティティ以外をキャッシュ |

# JPA 2.0
# 2次(共有)キャッシュのサポート

- EclipseLink における L2 キャッシュの設定

```
@Entity
@Cache(type=CacheType.WEAK,
    isolated=false,
    expiry=60000,
    alwaysRefresh=true,
    disableHits=true,
    coordinationType=INVALIDATE_CHANGE_OBJECTS)
@Cacheable(true)
public class Person implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
```

TYPE=FULL: 削除されるまでフラッシュされない

TYPE=WEAK: 参照がなくなると GC 実行

Isolated=true に設定した場合 L2 キャッシュを無効化

# JPA 2.0
## 2次(共有)キャッシュのサポート

### クエリにおけるキャッシュ動作設定

- Cache Retrive (検索)モード
  - javax.persistence.cache.retrieveMode
  - BYPASS :キャッシュを無視しDBより結果を取得
  - USE :キャッシュを利用(既にキャッシュ内に存在する場合)
- Cache Store (保存)モード
  - javax.persistence.cache.storeMode
  - BYPASS : キャッシュを更新せず、DBに直接保存
  - REFRESH : 既存キャッシュをリフレッシュ/置換
  - USE : キャッシュを更新しコミット時にDBに反映

# JPA 2.0
# 2次(共有)キャッシュのサポート

## クエリにおけるキャッシュ動作設定

```
public List<Person> findPersons(){
    TypedQuery<Person> query =
                                em.createQuery("SELECT p ...");
    query.setProperty(
            "javax.persistence.cache.retrieveMode",
            CacheRetrieveMode.BYPASS);
    query.setProperty(
            "javax.persistence.cache.storeMode",
            CacheStoreMode.REFRESH);
    return query.getResultList();
}
```

※1:REFRESH はデフォルトの設定のため記載不要

※2:全ての検索で BYPASS するとキャッシュの意味無し(用途の検討が必要)

# JPA 2.0
# 設定項目の標準化（JPA 1.0 の場合）

- ベンダー固有の設定が必要
  - JDBCドライバ、URL、ユーザ名、パスワード等
- 移植性が低下

```
<property name="eclipselink.jdbc.driver"
    value="org.apache.derby.jdbc.ClientDriver"/>
<property name="eclipselink.jdbc.url"
    value="jdbc:derby://localhost:1527/Sample"/>
<property name="eclipselink.jdbc.user" value="APP"/>
<property name="eclipselink.jdbc.password" value="APP"/>
```

# JPA 2.0
# 設定項目の標準化（JPA 2.0 の場合）

- persistence.xml の設定項目の標準化
- JPA 2.0 環境での移植性が向上
  - EclipseLink, Hibernate, OpenJPA 間で移植性が向上

```
<property name="javax.persistence.jdbc.driver"
        value="org.apache.derby.jdbc.ClientDriver"/>
<property name="javax.persistence.jdbc.url"
        value="jdbc:derby://localhost:1527/Sample>
<property name="javax.persistence.jdbc.user" value="APP"/>
<property name="javax.persistence.jdbc.password" value="APP"/>
```

# JAX-RS 1.1

ORACLE

# JAX-RS 1.1

## http://jsr311.java.net/

- RESTful サービス用の高レベル HTTP API
- POJO とアノテーションベース
  - web.xml の設定は不要
  - API の利用が可能
- HTTP メソッドとのマッピング
  - GET, POST, PUT, DELETE 等
- EJB, CDI, Servlet 等と統合
  - ステートレス、シングルトン Bean と統合
  - Managed Bean 1.0 と統合
  - CDI と統合

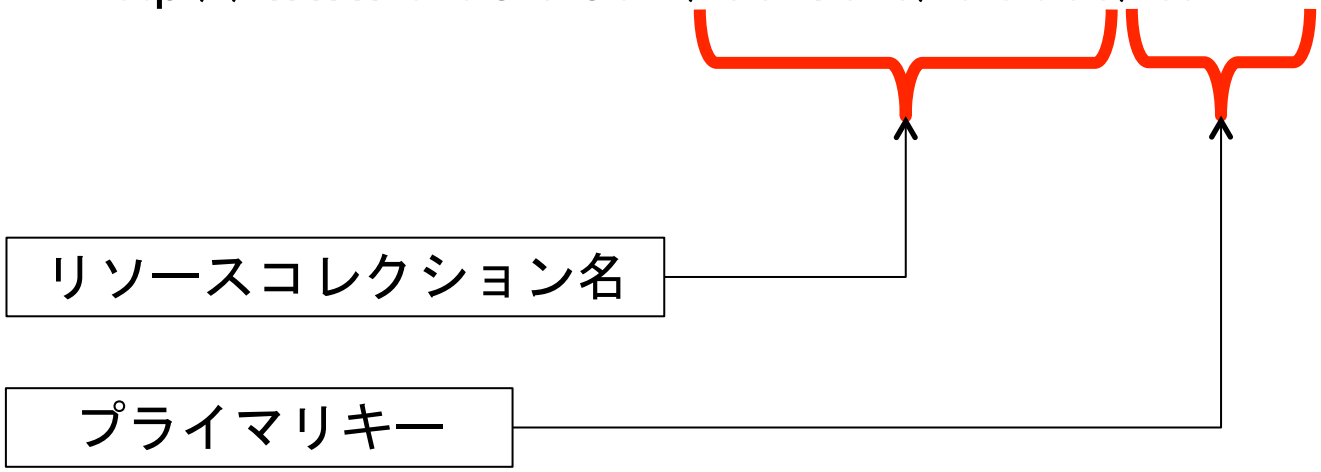
ORACLE

# JAX-RS 1.1
# REpresentational State Transfer

- 全ての HTTP リソースは ID を持つ
  - ID を URI で表す
  - http://example.com/widgets/foo
  - http://example.com/customers/bar
  - http://example.com/customers/bar/orders/2
  - http://example.com/orders/110421/customer

# JAX-RS 1.1
## URI の例

- http://www.oracle.com/servers/blades/t5440




リソースと URI をどのようにマップするかを定義

# JAX-RS 1.1
## 複数の表現形式

- 複数形式のフォーマットを提供
  - XML
  - JSON
  - (X)HTML
- コンテンツ・ネゴシエーションのサポート
  - Accept header

    GET /foo

    Accept: application/json
  - URI ベース

    GET /foo.json

# JAX-RS 1.1
## REST リクエストの構成

- リクエストのリソース(名詞)
  - URI で識別
  - 例 : http://www.example.com/parts
- メソッド(動詞) – 名詞を操作する為
  - GET, POST, PUT, DELETE 等
- 表現 – 状態確認
  - クライアント – サーバ間で状態を転送するための表現
  - XML, JSON, (X)HTML
- アプリケーションの状態と表現を変換するためメソッドを使用

# JAX-RS 1.1
## REST の HTTP リクエスト/レスポンス

# JAX-RS 1.1
## ルートリソースクラスの例

- 想定するアプリケーションコンテキスト
  - http://example.com/catalogue
  - 一覧取得 – GET http://example.com/catalogue/widgets
  - 特定項目取得 – GET http://example.com/catalogue/widgets/nnn

```
@Path("widgets")
Public class WidgetsResource{
   @GET
   String getList(){…}

   @GET @Path("{id}")
   String getWidget(@PathParam("id") String id) {…}
}
```

# JAX-RS 1.1
## リソースクラスの例

```
@Path("root")
Public class RootResource{ //リクエストスコープ
  @Context UriInfo uri;

  @GET
  public String get (){ return "GET"; }

  @Path("sub-resource")
  public SubResource sub() { return new SubResource();}
}

public class SubResource{
  …
}
```

# JAX-RS 1.1
## URI Path テンプレート

- URI パステンプレート
  - URI 表現中の埋め込み変数の利用
  - リクエストされた特定の変数を取得するため @PathParam を使用

例：http://example.com/users/FooBar

```
@Path("/users/{username}")
Public class UserResource{
  @GET
  @Produces("text/xml")
  String getUser(@PathParam("username")String name){
    …
  }
}
```

# JAX-RS 1.1
## @PathParam, @QueryParam

- ２つのアノテーションを使用しクライアントから送信された情報を抽出

- @PathParam – リクエストから直接情報抽出
  - 例：http://host/catalogue/items/123
- @QueryParam – リクエストの URI クエリーから情報抽出
  - 例：http://host/catalogue/items/?item=123

# JAX-RS 1.1
## @PathParam, @QueryParam の実装例

```
@Path(“/items”)
@Consumes(“application/xml”)
Public class ItemResource{
  //リクエストhttp://host/catalogue/items/?start=123
  @GET
  ItemConverter get (@QueryParam(“start”)int start){
    … }

  //リクエストhttp://host/catalogue/items/123
  @Path(“{id}”)
  ItemResource getItemResource(@PathParam(“id”)Long id){
    …
}
```

# JAX-RS 1.1

## @Produces

- リソース表現のMIME タイプの指定で使用
- MIME タイプに応じたデータを作成しクライアントへ返信
- クラスレベルとメソッドレベルそれぞれで指定可能
  - メソッドレベルで設定した値はクラスレベルで設定した値を上書き

# JAX-RS 1.1
## @Produces の実装例

```
@Path(“/myResource”)
@Produces(“text/plain”)
Public class SomeResource{

  //指定しない場合クラスレベルの設定が有効
  @GET
  public String doGetAsPlainText (){}

  //クラスレベルの設定を上書き
  @GET
  @Produces(“text/xml”)
  public String doGetAsXML () {}
}
```

# JAX-RS 1.1

## Response クラスを利用したレスポンスの生成

```
//新規リソースを作成後、追加リソースを指すURIを返す
@POST
@Consumes("application/xml")
public Response addUser(InputStream userData){
  try{
    User user = getUser(userData);
    long userId = persist(user);   //DB に永続化
    return Response.created(URI.create("/" +
                                       userId)).build();
  } catch (Exception e) {
    throw new MyException(e);
  }
}
```

# JAX-RS 1.1

## @Consumes

- クライアントから送信されたリソースの MIME タイプを指定
- クラスレベル、メソッドレベルそれぞれの設定が可能
  - メソッドレベルの設定はクラスレベルの設定を上書き
- コンテナはクライアントから指定された MIME タイプに応じたメソッドの呼び出しを行う
  - 指定された MIME タイプ用のメソッドが存在しない場合 HTTP のエラーコード “415 Unsupported Media Type” を返信

ORACLE

# JAX-RS 1.1
## @Consumes の実装例

```
//クライアントで指定された MIME タイプに応じた処理の実施
@POST
@Consumes("application/xml")
public Response postXml(String xmlData){
    …
}

@POST
@Consumes("text/plain")
public Response postText(String textData){
    …
}
```

# JAX-RS 1.1

- JAX-RS 1.1
  - Web Profile には含まれない
  - GlassFish Web Profile 版では利用可能
- JCP
  - http://jcp.org/en/jsr/detail?id=311
- API
  - http://jsr311.java.net/nonav/releases/1.1/index.html
- 仕様
  - http://jsr311.java.net/nonav/releases/1.1/spec/spec.html

# Bean Validation

# Bean Validation 1.0

- アプリケーション中で宣言的なバリデーションが可能
- カスタムバリデーションを作成可能
- １度の制限でどこでもバリデート可能
  - Bean 、フィールド、プロパティに制限
  - Null チェック、数値適用範囲チェック、メールアドレスチェック等
- JSF 2.0 に統合
  - f:validateRequired, f:validateRegexp
  - ManagedBean
    - @NotNull, @Size(max=40) String address 等
- JPA 2.0 に統合
  - Entity クラス
    - @NotNull, @Size(max=40) String address 等

ORACLE

# DI/CDI

# Dependency Injection

## DI 1.0/CDI 1.0

- @Inject アノテーション
  - @Inject @LoggedIn User user;
- Injection メタモデル
  - どんなBeanもInject対象
  - EJB session beans
  - Plain classes with @ManagedBean
  - CDIがモジュール内で見つけたクラス
  - デフォルトで無効、有効化する場合は、beans.xmlを配置
  - META-INF/、WEB-INF/に配置

# Java EE 7 の
# テーマ：クラウド

# Java EE 7 に含まれる技術


- JPA 2.1 (JSR-338)
- JAX-RS 2.0 (JSR-339)
- JMS 2.0
- JavaServer Faces 2.2
- WebTier
  - HTML 5対応
  - WebSocket 対応
  - JSON API のサポート

SOFTWARE. HARDWARE. COMPLETE.

ORACLE


ORACLE®