Comparison between Java EE 5 and J2EE 1.4

Applies to:

Java EE 5.0 & J2EE 1.4

For more information, visit the Java homepage.

Summary

J2EE is enormously powerful. Java EE 5 makes getting started easier to develop enterprise Java applications. This document compares between Java EE 5 and J2EE 1.4. This document introduces the new features of Java EE 5.

Author: Biplab Ray

Company: IBM India Pvt Ltd

Created on: 13 October 2008

Author Bio

Biplab Ray is working for IBM India Pvt Ltd as System Engineer and development of composite applications using CAF, GP, WebDynpro. He also works in technologies like Java/J2EE and has depth understanding on SOA, Webservice, Enterprise service, XML.

© 2008 SAP AG

Table of Contents

Java EE5 – What is it? 5
Java EE5 5
From J2EE 1.3 to Java EE5 5
Java EE5 Overview 6
Java EE 5 is based on Java SE 5.0 6
J2SE 1.4 – Boxing / Unboxing 7
  Boxing 7
  Unboxing 7
  Why wrapper classes? 7
J2SE 5.0 – Autoboxing / Unboxing 8
J2SE 5.0 – Enhanced for Loop 9
  J2SE 1.4 9
  J2SE 5.0 11
  When to use the for-each loop? 11
  Limits 11
J2SE 5.0 – Static Imports 12
  J2SE 1.4 12
  J2SE 5.0 13
J2SE 5.0 – Typesafe Enums 14
  J2SE 1.4 14
  J2SE 5.0 14
  Things to know about enums 15
J2SE 5.0 – Varargs 15
  J2SE 1.4 15
  J2SE 5.0 16
J2SE 5.0 – Generics Introduction 17
  J2SE 1.4 17
  J2SE 5.0 17
  What is the Compiler doing? 18
J2SE 5.0 – Generics – Inheritance and Compatibility 18
J2SE 5.0 – Generics – Casting 19
J2SE 5.0 – Generics – Raw Types 19
J2SE 5.0 – Generics – Wildcards 20
J2SE 5.0 – Generics – Bounded Type Parameter 21
J2SE 5.0 – Generics and Arrays 22
J2SE 5.0 – Generic Methods 23
J2SE 5.0 – Annotations – Introduction 23
J2SE 5.0 – Annotations – Usage 24
J2SE 5.0 – Meta Annotations 24

© 2008 SAP AG

Java EE 5 – Ease of development 25
J2EE 1.4: JNDI call, object creation, XML 25
Java EE 5: Annotation 26
Example: Create a Web Service from a session bean 26
J2EE 1.4 26
The .xml files 27
Java EE5 27
EJB 2.x – Drawbacks 28
EJB 3.0 – Overview 29
EJB 3.0 – Annotations 30
EJB 3.0 - POJI & POJO 31
EJB 3.0 - Dependency Injection 31
EJB 3.0 – Interceptors 32
EJB 3.0 – Default 33
EJB 3.0 - Message-Driven Beans 33
EJB 3.0 – Java Persistence API 1.0 34
New Features of the EJB Container 36
EJB Fundamentals 37
Resource management 37
Life Cycle of a Stateless Session Bean 37
Life Cycle of a Stateful Session Bean 38
Primary Services: Concurrency 38
Primary Services: Transactions 38
Primary Services: Transactions 39
Java Persistence API (JPA 1.0) 40
JPA – The New Java Persistence API 40
JPA – Key Features 40
JPA – Entity Example 41
JPA: How Applications Control the State of Entities 42
JPA: Entity Manager 42
JPA: Persistence Context 43
JPA: Persistence Context Types 43
JPA: Detaching and Merging Entities 43
JPA: Queries 44
JPA: Other Powerful Features for Entities 45
JPA: Concurrency Control 45
JPA: Some Elementary Schema Mappings 46
JPA: Mapping example 47
Entity Relationships 48
One to one unidirectional example 48
One to one bidirectional 49
One to many unidirectional 50
One to many bidirectional 51

© 2008 SAP AG

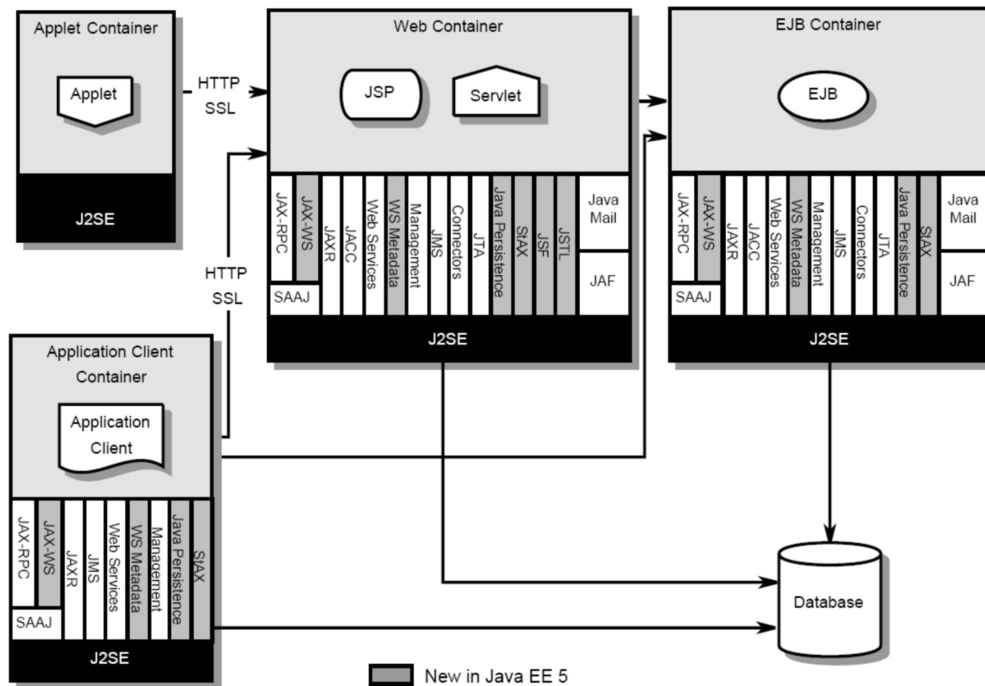
Many to one unidirectional 52
Many to many unidirectional 53
Many to many bidirectional 54
Cascading 54
Related Content 55
Disclaimer and Liability Notice 56

© 2008 SAP AG

Java EE5 – What is it?

Java EE 5 = THE standard for enterprise applications in Java

It's part of ISV's expectations towards a Java Server



Java EE5

- Build on Java SE 5.0
- Ease of development
- Usage of annotations
- EJB 3.0 simplification
- Java Persistence API
- JavaServer Faces
- Web Services simplified

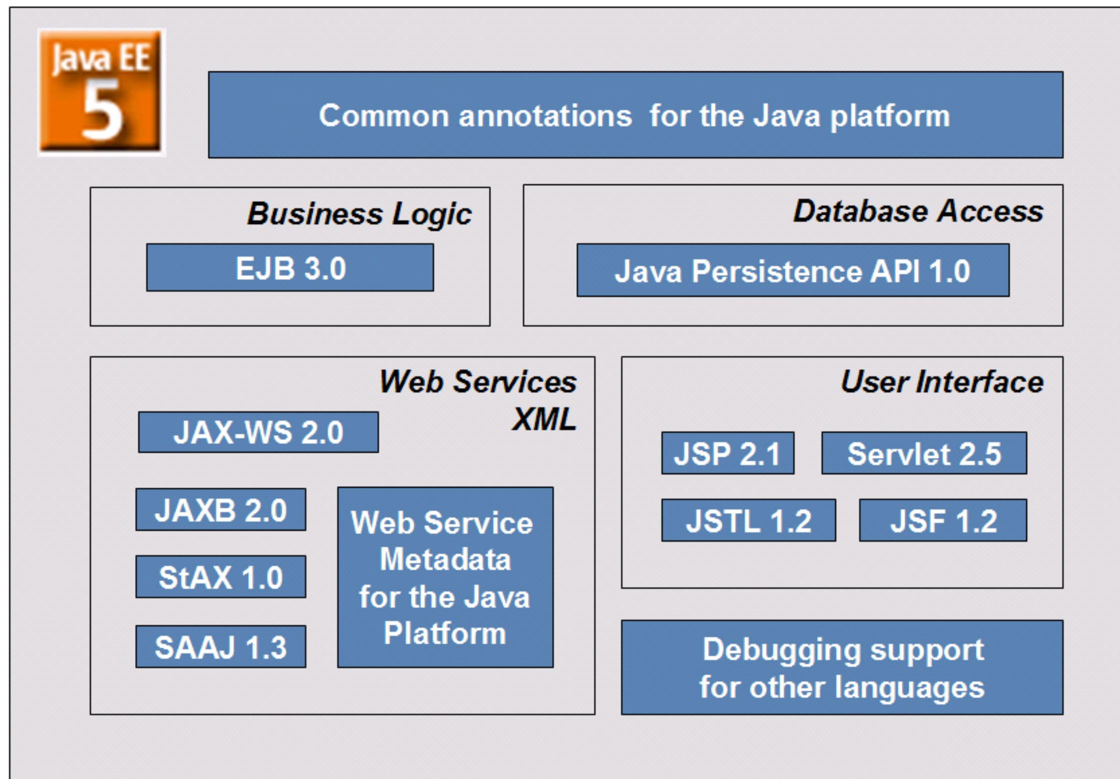
From J2EE 1.3 to Java EE5

Remark:

- J2EE was renamed to Java EE
- Correct terminology ➔ J2EE 1.3 J2EE 1.4 Java EE 5
- Wrong terminology ➔ J2EE 1.5 J2EE 5 JEE 5 JEE5 Java EE 5.0

© 2008 SAP AG

Java EE5 Overview



Java EE 5 is based on Java SE 5.0

- Autoboxing / Unboxing
  - Eliminates manual conversion between primitive types and wrapper types
- Enhanced for Loop
  - Eliminates iterators and index variables when iterating over collections and arrays
- Typesafe Enums
  - Real typesafe enumerations
- Static Imports
  - Avoid qualifying static members / methods with class names
- Varargs
  - Variable-length argument lists
- Generics
  - Compile-time type safety to the Collections framework.
- Annotations (Metadata)
  - Metadata in the source-code allowing tools (Compiler) to generate code

© 2008 SAP AG

J2SE 1.4 – Boxing / Unboxing

Boxing

There is a wrapper class for every primitive in Java. Primitive values can not be treated as objects, they have to be boxed into wrapper classes.

Examples:

```
Integer iWrapper = new Integer(4711);

Integer iWrapper2 = new Integer("101011", 2);
```

Unboxing

To get back the primitve value a method of the wrapper class must be called.

Examples:

xxxValue() - convert the value of a wrapped numeric to a primitive

```
int iValue = iWrapper.intValue();

byte bValue = iWrapper.bytValue();

short sValue = iWrapper.shortValue();
```

Why wrapper classes?

Header The primitives can be included in activities reserved for objects, like as being added to Collections, or returned from a method with an object return value.

To provide an assortment of utility functions for primitives.

Example: Add and retrieve int values from a collection

```
collection.add(new Integer(4711));                 Boxing

int i = ((Integer)collection.get(0)).intValue;     Unboxing
```


© 2008 SAP AG

J2SE 5.0 – Autoboxing / Unboxing

With J2SE 5.0 the compiler automatically does boxing and unboxing

```
Integer intObject = 4711;

int i = intObject;
```

The above code will be rewritten from the compiler into the following code

```
Integer intObject = Integer.valueOf(4711);

int i = intObject.intValue();
```

Example: Add and retrieve int values from a collection

Java 1.4.2

```
import java.util.ArrayList;

public class ManualBoxing {

public static void main(String[] args)

{

ArrayList list = new ArrayList();

for (int i = 0; i < 10; i++){

list.add(new Integer(i));

}

}

}
```

Java 1.5

```
import java.util.ArrayList;

public class ManualBoxing {

public static void main(String[] args) {

ArrayList<Integer> list = new ArrayList<Integer>();

for (int i = 0; i < 10; i++){

list.add(i);

}

}

}
```

Do not use autoboxing for scientific or performance-sensitive numerical code. An Integer is not a substitute for an int!

© 2008 SAP AG

J2SE 5.0 – Enhanced for Loop

J2SE 1.4

Iterating over an array or a collection is done using an index variable or an Iterator object even the Iterator or the index are most of the time not really needed !

```
public class OldForArray {

public static void process(int[] mArray){

for (int i=0; i< mArray.length; i++){

 System.out.printf("%d squared is %d.\n",i, mArray[i]);

}

}

}


public class OldIteratorUsage {

 public static void process(List mList) {

Iterator iterator = mList.iterator();

while (iterator.hasNext()) {

System.out.printf("%d squared is %d.\n", j++, iterator.next());

}

}

}
```


© 2008 SAP AG

The iterator is just clutter. Furthermore, it is an opportunity for ERROR !!!

```
List suits = ...;
List ranks = ...;
List sortedDeck = new ArrayList();
for (Iterator i = suits.iterator(); i.hasNext(); )
for (Iterator j = ranks.iterator(); j.hasNext(); )
sortedDeck.add(new Card(i.next(), j.next()));
```

throws NoSuchElementException!

```
List suits = ...;
List ranks = ...;
List sortedDeck = new ArrayList();
for (Iterator i = suits.iterator(); i.hasNext(); ) {
Suit suit = (Suit) i.next();
for (Iterator j = ranks.iterator(); j.hasNext(); )
sortedDeck.add(new Card(suit, j.next()));
}
```


© 2008 SAP AG

J2SE 5.0

- Inside the for loop a variable is defined, which automatically contains the collection (array) values for each step.
- You don't have to declare the iterator, you don't have to provide a generic declaration for it.
- The compiler does this for you behind your back, but you need not concern yourself with it.

```
public class OldForArray {
public static void process(int[] mArray){
for (int element : mArray){
System.out.println("Element: ", element);
}
}
}
```

```
public class OldIteratorUsage {
 public static void process(List mList) {
for ( Element element : mList) {
 System.out.println("Element: ", element);
}
}
}
```

When to use the for-each loop?

Any time you can.

Limits

The for-each loop is not usable for filtering because it hides the iterator, so you cannot call remove.

© 2008 SAP AG

J2SE 5.0 – Static Imports

J2SE 1.4

- Standard way of represent constant types was:

```
public class/interface Season{
public static final int WINTER = 0;
public static final int SPRING = 1;
public static final int SUMMER = 2;
public static final int FALL   = 3;
}
```

In order to access static members, it is necessary to qualify references with the class they came from.

```
int season = Season.SUMMER;
switch( season ){
case Season.WINTER: ....
}
```

In order to get around this, people sometimes put static members into an interface and inherit from that interface.

BAD IDEA!!! Constant Interface Antipattern

Constant Interface Antipattern

Sometimes, people place static members into an interface and inherit from it looking for avoiding prefixing static members with class names.

```
public class Order extends Values implements Constants {
double price() {
return basePrice() + shipping() + TAX;
  }
double shipping() {
return Math.min(basePrice(), SOME_VALUE);
}
}
```

Interfaces are intended for defining types, not providing access to static members. When a class implements an interface, it becomes part of the class's public API, creating an undesirable connection between classes and confusing clients.

© 2008 SAP AG

J2SE 5.0

The static import construct allows unqualified access to static members without inheriting from the type containing the static members. Instead, the program imports the members, either individually:

```
import static com.sap.myApp.util.Season.WINTER;
```

or

```
import static com.sap.myApp.util.Season.*;
```

Once the static members have been imported, they may be used without qualification:

```
public String getSeasonAsString(){

switch (this.season) {

case WINTER: return “winter”;

case SPRING: return “spring;

….

}

}
```

Use it only when you require frequent access to static members from one or two classes. If you overuse the static import feature, it can make your program unreadable and unmaintainable!

© 2008 SAP AG

J2SE 5.0 – Typesafe Enums

J2SE 1.4

Standard way of represent enumerated types was the "int Enum pattern". However this pattern has major disadvantages.

```
public class Season{

public static final int WINTER = 0;

public static final int SPRING = 1;

public static final int SUMMER = 2;

public static final int FALL   = 3;

}

int season = Season.SUMMER;          //ok

switch(season){

case Season.WINTER:       //ok

....

}

season = 4711;       //ok but useless
```

J2SE 5.0

Enumerations use the keyword enum. Enums are rewritten by the compiler into normal classes.

```
public enum Season{

WINTER, SPRING, SUMMER, FALL;

}

Season seasons = Season.SUMMER;    //ok

switch(seasons){

case WINTER:                              //ok, qualifier not allowed

....

}
```

The enum values (WINTER, SPRING...) are static references pointing to objects of type Season

```
public static final Season SPRING = new Season();
```


© 2008 SAP AG

Things to know about enums

- You can use the toString() method of enum objects. It prints out the name of the static reference (WINTER, SUMMER...)
- Generated class implements interface Comparable, so you can compare for order.

```
Season.WINTER.compareTo(Season.SUMMER));   //-2
```

- Generated class implements the static method values(), which returns an array of references to all enum objects.

```
Season[] seasonsArray = Season.values();
```

Since enums are rewiritten into classes, you can add arbitrary methods and fields to an enum type.

J2SE 5.0 – Varargs

J2SE 1.4

A method that takes an arbitrary number of parameters requires to create an array prior to invoking the method.

```
class Computer{
public static int sum (int[] v){
int sum = 0;
for(int i=0; i < v.length; i++)
sum += v[i];
return sum;
}
}
System.out.println(Computer.sum(new int[]{1,2,3}));
```


© 2008 SAP AG

J2SE 5.0

- Using ... after the type indicates, that an arbitrary number of parameters can be passed

```
class Computer{
public static int sum (int... v){
int sum = 0;
for(int i=0; i < v.length; i++)
sum += v[i];
return sum;
}
}
```

- The compiler just rewrites the method into

```
public static int sum(int[] v)
```

Therefore you can still use v.length and v[i].

- Varargs can be used only in the final argument position !

© 2008 SAP AG

J2SE 5.0 – Generics Introduction

J2SE 1.4

```
ArrayList myList = new ArrayList();
myList.add(new Integer(4711));
Integer i = (Integer)myList.get(0);
```

J2SE 5.0

```
ArrayList<Integer> myList = new ArrayList<Integer>();
myList.add(4711);
Integer i = myList.get(0);
```

- Generics provides a way for you to communicate the type of an interface or class to the compiler : -> no cast necessary anymore
- ArrayList<Integer> is a generic class, that takes a type parameter
- The compiler can check the type correctness at compile-time

  myList.add("4711"); will be rejected at compile-time :-> more type safe

Excerpt of the interface definition List

```
public interface List<E>{
void add(E x);
Iterator<E> iterator();
}
```

- You can see <E> as a placeholder. It will be uniformly replaced by Integer
- Using List<Integer> you can add Integers or subtypes of Integer to this collection
- Whereas List defines only one type, List<E> is the basis for defining an arbitrary number of types !
- Difference to C++ Templates: Although multiple types can be generated with List<E>, there are NO multiple copies of the code (only one .class file at runtime)

© 2008 SAP AG

What is the Compiler doing?

- The compiler translates a generic class into a normal class, removing the type information. This is called erasure
- After erasure, the compiler adds the necessary casts into the code
- It is only marked in the meta-data, that this class is a generic type

  The following code therefore returns true:

```
ArrayList<Integer> li = new ArrayList<Integer>();

ArrayList<String> ls = new ArrayList<String>();

System.out.println(li.getClass() == ls.getClass());
```

J2SE 5.0 – Generics – Inheritance and Compatibility

- Without using generics, the following code is ok

```
ArrayList l = new ArrayList();

Object  o = l;
```

- Using Generics, the following code is rejected by the compiler

```
ArrayList<String> l = new ArrayList<String>;

l.add("Hello");

ArrayList<Object> o = l; //compiler error, because...

o.add(new Object());     // list o has now different types
```

- The following code however is possible...

```
ArrayList<String> al = new ArrayList<String>();

List<String> li = al;
```

Inheritance for generic types is only possible, if the generic type parameter is the same and the base classes have an inheritance relationship

© 2008 SAP AG

J2SE 5.0 – Generics – Casting

- Casts are only allowed, if the generic type parameter is the same

```
ArrayList<String> al = new ArrayList<String>();

Collection<String> c = al;

ArrayList<String> li = (ArrayList<String>)c;  // Works OK

//not allowed

ArrayList<String> al = new ArrayList<String>();

Collection<Object> c = (Collection<Object>)al; // Error
```

J2SE 5.0 – Generics – Raw Types

- The "Raw Type" is the not generic type which is generated after compiling the generic type. It can also be used and makes J2SE 5.0 code compatible with old (pre 5.0) code.
- Compatibility with Raw Types:

```
List l = new ArrayList<String>();

l.add(new Object()); // Should Not Be Possible. Compiler Warning.

String s = l.get(0); // Compile Error

String s  = (String)l.get(0); // Runtime Error
```


© 2008 SAP AG

J2SE 5.0 – Generics – Wildcards

- Write a method that prints out all elements in a list?

```
void printCollection(List<Object> c){
for(Object o: c)
System.out.println(o);
}
…
List<String> l = new ArrayList<String>();
l.add("Hello");
someVariable.printCollection(l); // Error
```

- Compiler error, because List<String> is not type compatible with List<Object>

- <?> is used as a wildcard and is equal to "any type", a type which is not known so far

```
void printCollection(List<?> c){
for(Object o: c)
System.out.println(o.toString());
}
…
List<String> l = new ArrayList<String>();
l.add("Hello");
someVariable.printCollection(l); // Ok
```

- You can see List<?> as a super-type of all other List types like List<String>, List<Integer> .....

© 2008 SAP AG

- Even if List<?> can be seen as the super-type for List<String>, List<Integer> the following code does not work

```
List<?> l = new ArrayList<String>();

l.add("Hello"); //Error

l.add(new Object()); // Error

l.add(new Integer(4711)); // Ok

Object o =l.get(0);
```

- Since we do not know the type of List, we cannot add something to it, because it could violate the integrity of the list.
- But we can read something which is not known from the List and assign it to Object!

J2SE 5.0 – Generics – Bounded Type Parameter

- If we want to set constraints on a type parameter, we use a bounded type variable e.g. <E extends Vector>

```
class MyClass<E extends Vector>{}

MyClass<String > my = new MyClass<String>(); // Error

MyClass<Vector> my = new MyClass<Vector>(); //Ok

MyClass<Stack> my = new MyClass<Stack>(); // Ok
```

- Using extends, you enforce that the typed parameter implements one or more interfaces, or that it is a subclass of a specified class.

- Syntax of multiple types and interfaces

```
class C<T1 extends A & I1 & I2, T2 extends B> {   }
```

Typed parameter T1 must be of type A or a subclass of type A and must implement the interfaces I1 and I2

© 2008 SAP AG

Additional types are separated with commas

- Example of a bounded type parameter:

```
class A{
public void m(){}
}
class C<T extends A >{
public void f(T t){
t.m() ; // only possible as T extends A
}
}
Using Class C :
C<A> c1 = new C<A>();
C1.f(new A());
```

- If T would not have a constraint, it would not be possible to call method m inside f. It would only be possible to call the methods defined for Object.
- <T super A> : T must be of type A or a parent of A

J2SE 5.0 – Generics and Arrays

- Using Arrays, the VM makes a runtime check and an ArrayStoreException is thrown, if a different type is inserted

```
Object[] objects = new String[10];
objects[0] = "7"; // Ok
objects[1] = new Integer(4711);  // Compile Time ok but throws an exception at runtime
```

- Because the compiler erases all generic information from the generated, class, the VM is not able to make an insert check at runtime. Type-safety could not be guaranteed during runtime !
- For that reason it is not possible to create generic array objects (reference variables are possible).

© 2008 SAP AG

J2SE 5.0 – Generic Methods

- Non generic classes can still have generic methods. The generic type parameter is introduced before the return type of the method
- Example: Write a utility function, which randomly returns either parameter 1 or parameter 2. The method should be type independent

```
class Util{
public static <T> T randomParam(T p1, T p2){
return Math.random() > 0.5 ? p1 : p2;
}
}
//same method-call with different parameter sets
String s  = Util.randomParam("Hello", "World");
Integer i = Util.randomParam(47, 11);
```

J2SE 5.0 – Annotations – Introduction

- Manny API's need additional code or additional files, which must be maintained in parallel to the program code. For example JavaBeans require an additional BeanInfo class, EJB's require a deployment descriptor etc.
- It would be more convenient and less error-prone, if the information in these side files were maintained as annotations (meta data) in the program itself.
- Java already has some sort of annotations like the transient modifier or the @deprecated javadoc tag.
- With J2SE 5.0 there is now a general mechanism for defining and use of your own annotations
- An annotation is always an instance of a specific annotation type
- An annotation type is defined as a special interface

```
public @interface RequestForEnhancement {
int id();
String topic();
String responsible() default "[unassigned]";
}
```

- Each method defines an element of the annotation type
- Method declarations must not have any parameters or a throws clause
- Return types are restricted to primitives, String, Class, enums, annotations, and arrays of the preceding types.
- An annotation type is stored in a .java file and will be translated into a .class file e.g. RequestForEnhancement.class

© 2008 SAP AG

J2SE 5.0 – Annotations – Usage

- An annotation is a special kind of modifier, and can be used anywhere that other modifiers can be used.

```
@RequestForEnhancement(
id = 276423,
topic = "Performance optimization",
responsible = "Mr. Smith"
public static void handleEnhancement() {....}
```

- An annotation type with no elements is called a marker annotation. In this case, it is possible to omit the parentheses.

```
public @interface Preliminary { }
@Preliminary public class MyClass{...}
```

- In annotations with a single element, the element should be named value. In this case it is possible to omit the element name and the equals sign.

```
public @interface Copyright {
String value();
}
@Copyright("SAP AG") public class MyClass {...}
```

J2SE 5.0 – Meta Annotations

- There are some standard annotations which can be used when defining an annotation type. These are called meta annotations
- @Retention – Defines when the annotation is available (runtime, compile-time)
- @Target – Defines the program part, where the annotaion is valid (class, method or attribute?)

© 2008 SAP AG

Java EE 5 – Ease of development

Leverage new JDK 5.0 language features

➔ Annotations

Let the container do more work

➔ Dependency injection

Configuration by exception

➔ Defaulting for typical cases

Example: Access to session bean from a servlet

J2EE 1.4: JNDI call, object creation, XML

```
try{
Context ctx = new InitialContext();
MyBeanHome home = (MyBeanHome)ctx.lookup("java:comp/env/ejb/myBean");
MyBean myBean =  home.create();
myBean.method();
}catch(){
…
}
…
```

```
<ejb-local-ref>
   <ejb-ref-name>ejb/myBean</ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <local-home>test.MyBeanHome</local-home>
   <local>test.MyBean</local>
</ejb-local-ref>
...
```


© 2008 SAP AG

Java EE 5: Annotation

```
@EJB

private MyBean myBean;

...
myBean.method();

...
```

- The annotation @EJB indicates, that the container should inject an instance of the Session Bean into the variable „myBean“

Example: Create a Web Service from a session bean

J2EE 1.4

```
package endpoint;

import java.rmi.*;

public class HelloServiceImpl
            implements HelloServiceSEI {

    public String sayHello(String param)
        throws java.rmi.RemoteException {
        return "Hello " + param;
    }
}

package endpoint;

import java.rmi.*;

public interface HelloServiceSEI
            extends java.rmi.Remote {
    public String sayHello(String param)
        throws java.rmi.RemoteException;
}
```


© 2008 SAP AG

The .xml files

```
<?xml version='1.0' encoding='UTF-8' ?>
<webservices xmlns='http://java.sun.com/xml/ns/j2ee'
version='1.1'>
  <webservice-description>
    <webservice-description-name>
     HelloService</webservice-description-name>
    <wsdl-file>
     WEB-INF/wsdl/HelloService.wsdl</wsdl-file>
    <jaxrpc-mapping-file>
     WEB-INF/HelloService-mapping.xml
     </jaxrpc-mapping-file>
    <port-component xmlns:wsdl-
port_ns='urn:HelloService/wsdl'>
      <port-component-name>HelloService</port-component-name>
      <wsdl-port>wsdl-port_ns:HelloServiceSEIPort</wsdl-port>
      <service-endpoint-interface>
       endpoint.HelloServiceSEI</service-endpoint-interface>
      <service-impl-bean>
        <servlet-link>WSServlet_HelloService</servlet-link>
      </service-impl-bean>
    </port-component>
  </webservice-description>
</webservices>
```

```
<?xml version='1.0' encoding='UTF-8' ?>
<configuration
    xmlns='http://java.sun.com/xml/ns/jax-rpc/ri/config'>
  <service name='HelloService'
      targetNamespace='urn:HelloService/wsdl'
      typeNamespace='urn:HelloService/types'
      packageName='endpoint'>
    <interface name='endpoint.HelloServiceSEI'
        servantName='endpoint.HelloServiceImpl'>
    </interface>
  </service>
</configuration>
```

Java EE5

```
@Stateless

@WebService

public class HelloServiceBean{

public String sayHello(String param){

return "Hello "+param;

}

}
```


© 2008 SAP AG

EJB 2.x – Drawbacks

Too many software artifacts to implement

- bean class
- local / remote component interfaces
- local / remote component intefaces
- XML deployment descriptors

Callback methods

- ejbPassivate, ejbActivate, ejbLoad, ...
- Have to be implemented even when not used

XML hell

- Complex XML descriptors
- with almost no defaults

JNDI lookups everywhere

Entity beans limitations

- Heavyweight
- Not really object-oriented
- No transient instances ➔ separate DTO classes needed

Query Language limitations

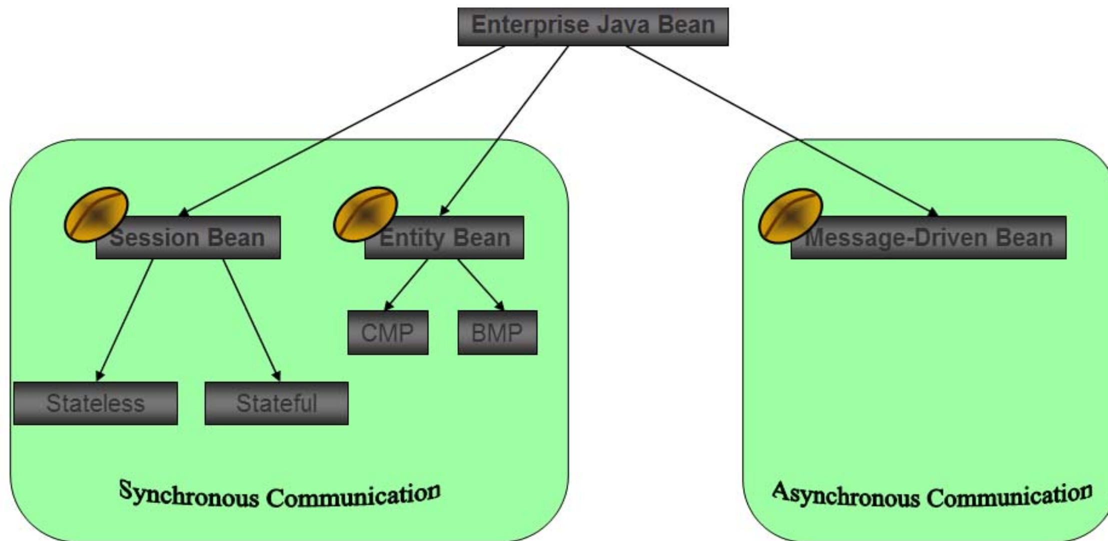
- Joins, sub queries, mass updates / deletes

Alternatives of EJB available

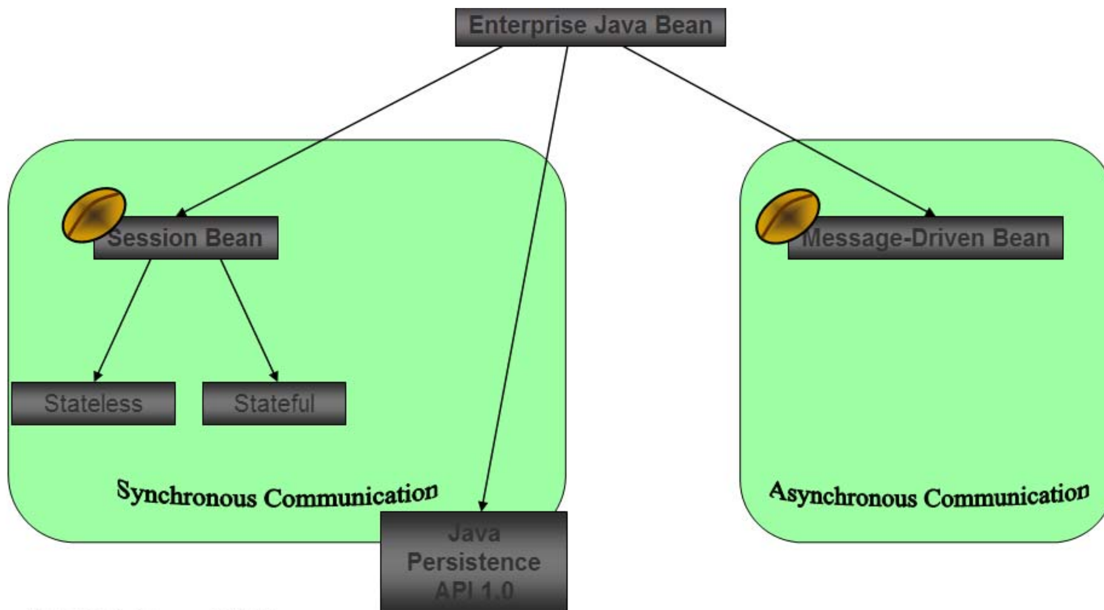
- Spring + Hibernate

© 2008 SAP AG

EJB 3.0 – Overview

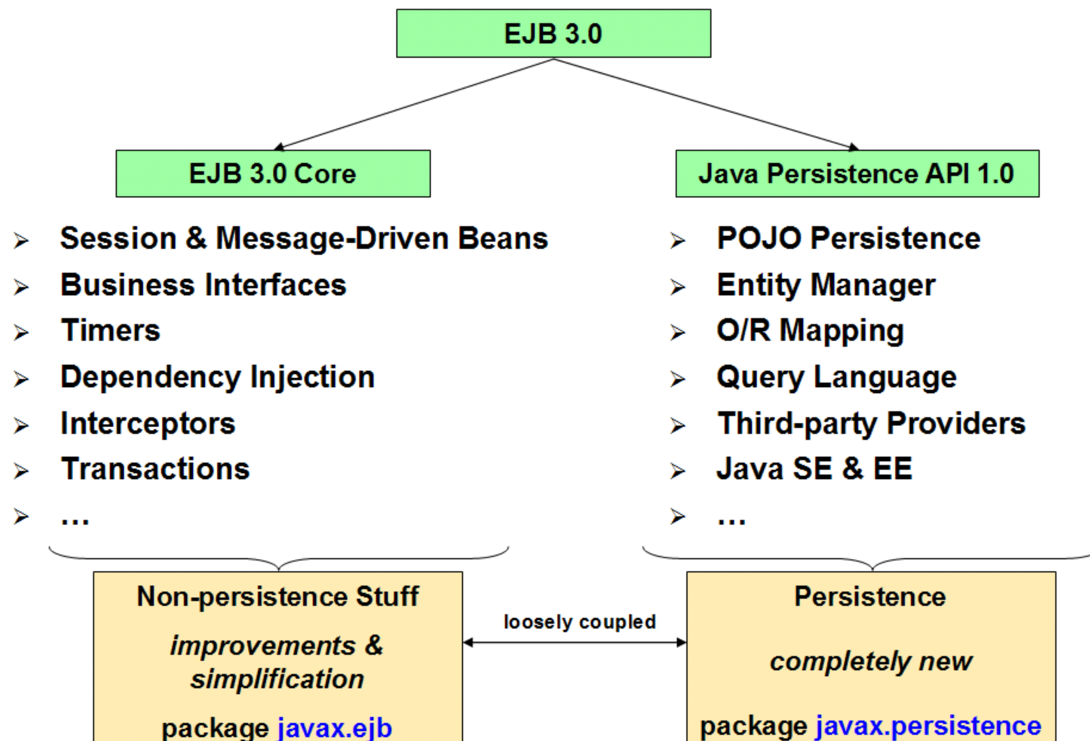


EJB 2.x (J2EE 1.3 & 1.4)



EJB 3.0 (Java EE 5)

© 2008 SAP AG



EJB 3.0 – Annotations

Annotations are a better alternative to deployment descriptors

Anything can be configured with annotations

However, deployment descriptors are still allowed

Below is an example of annotation

```
@Stateless public class HelloWorldBean implements HelloWord {

@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW

public String sayHello(String name) {

return "Hello " + name;

}

@AroundInvoke

public void profile(InvocationContext ctx) {

...
}

@PreDestroy

public void removing() {

...

}
}
```


© 2008 SAP AG

EJB 3.0 - POJI & POJO

POJO + Annotation == Session Bean

POJI Business Interface ➔ no need to extend EJB(Local)Object

```
@Local public interface HelloWorld {

   public String sayHello(String name);

}
```

POJO Bean Class ➔ no need to implement SessionBean

```
@Stateless public class HelloWorldBean implements HelloWorld {

   public String sayHello(String name) {

      return "Hello " + name;

}
```

No Home interfaces anymore!

EJB 3.0 - Dependency Injection

Let the container do more work!

```
@Stateless public class CartBean implements ShoppingCart {

@Resource javax.sql.DataSource orderDS;

public String makeOrder(String product) {

Connection c = orderDS.getConnection();

…

}

}
```

‘orderDS’ is already set by the container.

© 2008 SAP AG

EJB 3.0 – Interceptors

Interceptor = method intercepting a business method call

Implements a common logic for all business methods

```
@Stateless public class HelloWorldBean implements HelloWord {
public String sayHello(String name) {
return "Hello " + name;
}
@AroundInvoke
public void profile(InvocationContext ctx) {
long begin = System.currentTimeMillis();
try {
ctx.proceed();
} finally {
long time = System.currentTimeMillis() – begin;
log(“Executed for ” + time + “ ms”);
}
}
}
```


© 2008 SAP AG

EJB 3.0 – Default

Configuration by exception is the main motto of EJB 3.0!

In most cases, you don't need to configure anything!

```
@Stateless public class CartBean implements ShoppingCart {
@Resource javax.sql.DataSource orderDS;
public String makeOrder(String product) {
Connection c = orderDS.getConnection();
…
}
}
```

You get the following configuration for the bean above:

- Container-managed transactions; Required transaction attribute
- Business interface ShoppingCart; EJB name CartBean

Permit All security permissions

EJB 3.0 - Message-Driven Beans

No new functionality after EJB 2.1

Ease of development

```
@MessageDriven(activationConfig = {
@ActivationConfigProperty(propertyName = "destinationType",
propertyValue = "javax.jms.Queue")
@ActivationConfigProperty(propertyName = "destination",
propertyValue = "MyQueue")})
public class ReceiverBean implements MessageListener {
public void onMessage(javax.jms.Message msg) {
...
}
}
```

- No need to implement MessageDrivenBean

© 2008 SAP AG

EJB 3.0 – Java Persistence API 1.0

- POJO Persistence
- No Interfaces Required
- Inheritance
- Polymorphism
- Implicit O/R Mapping
- Can be Data Transfer Objects
- Java Persistence Query Language
- Can be Used in Java SE & EE

```
@Stateless public class ProductsBean {
@PersistenceContext(name="Orders")
EntityManager em;
public void createNewOrder(String id) {
Order order = new Order(id, name);
em.persist(order);
}
}
```


© 2008 SAP AG

```
@Entity
public class Order {
private String id;
private String name;
@Id
public String getId() {
return id;
}
public void setId(String _id) {
id = _id;
}
public String getName() {
        return name;
    }
public void setName(String _name) {
        name = _name;
    }
}
```


© 2008 SAP AG

New Features of the EJB Container

Lazy start of EJB modules

- No resources are acquired at server start time
- EJBs are not started until they are accessed
  - For instance: implicit / explicit JNDI lookups

EJB metadata model built and stored during deployment

- Minimize application start time
- By shifting processing from startup to deployment time
- Application start ➔ Just load pre-processed EJB model

Improved session management for stateful session beans

- Seamlessly integrated into “Solid Rock” session management
- Unique robustness / failover capabilities based on shared Java objects

Management of EJBs

- Compliant with Java management specification
- Integrated into SAP management infrastructure
- NW-Admin UI.

Stateful Web services through stateful session beans

- The bean class itself can be exposed as a web service
- No business interfaces or web service interface are required
- Each business interface can be exposed as a web service
- The stateful bean instance is shared in the session
  ➔ stateful Web services

© 2008 SAP AG

EJB Fundamentals

Resource management

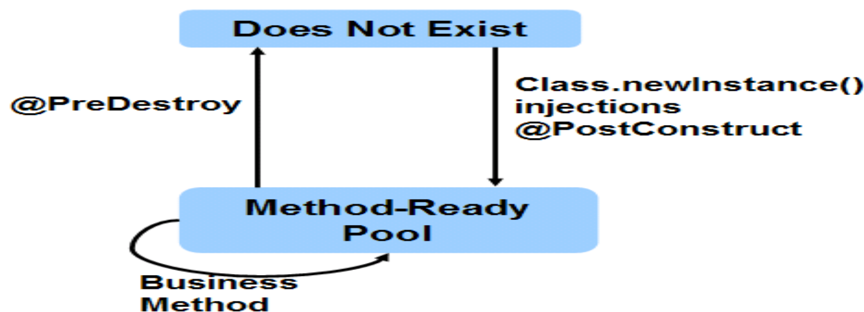
EJB supports two mechanisms to handle large numbers of clients and reduce resource consumption

- Instance pooling for Stateless Session Beans / Message Driven Beans
  - Subsequent client requests are served by different beans from the pool.
  - After executing a business method, the bean is put back into the pool.
  - No state is kept between client calls
  - Pool size is managed by the container

- Activation Mechanism for Stateful Session Beans
  - If the container needs resources, beans are removed from memory
  - The state of the bean is serialized to secondary memory

If a client invokes a method on the bean, a new instance is created and populated with the serialized state

Life Cycle of a Stateless Session Bean



Does Not Exist state: The bean has not been instantiated (is not in memory)

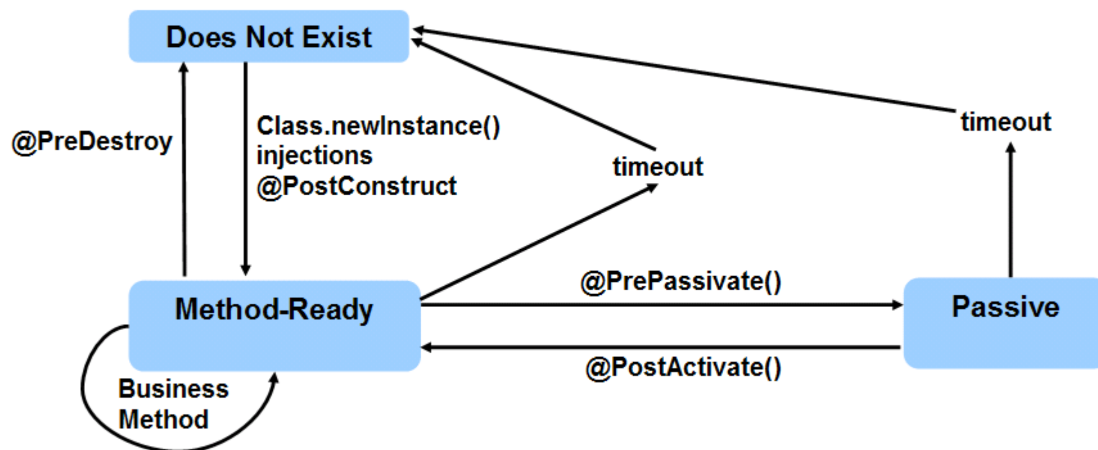
Method-Ready Pool state: A bean instance is created in memory and is ready to serve client requests to business methods.

During the transition from “Does not exist” to “Method-ready” the container creates a new instance, inject resources into the bean and calls a method which is annotated with @PostConstruct.

Before removing a bean from memory, a method with the annotation @PreDestroy is called by the container.

© 2008 SAP AG

Life Cycle of a Stateful Session Bean



Transition to Passive state: Beans get into the Passive state, if the container resources get limited.

Transition to Does Not Exist state: In case of timeouts or in case the client calls a method with the annotation @Remove, the bean instance is removed and goes into the Does Not Exist state.

Primary Services: Concurrency

Session Beans: Session Beans do not need to be thread-save (synchronized keyword is forbidden by the EJB spec).

Stateless Session Beans do not need to be concurrent, because simultaneous client calls are served with different bean instances (taken from the pool).

Stateful Session Beans do not need to be concurrent, because each client gets his own instance and is bound to that single instance.

Message Driven Beans do not need to be concurrent, because simultaneous messages are served with different bean instances from the pool.

Primary Services: Transactions

Declarative (implicit) transaction management

- Transactional behavior can be controlled on method level with the annotation @TransactionAttribute
- By default the transaction attribute on a method is set to “REQUIRED”. This means, that a new transaction is created if the client doesn’t already have one
- Transaction Propagation is supported (call of other operations inside a method)

```
// this annotation is redundant, since the default is REQUIRED

   @TransactionAttribute(value=TransactionAttributeType.REQUIRED)

public Customer createCustomer(String name) {
    ...

   }
```


© 2008 SAP AG

Primary Services: Transactions

Explicit (bean managed) transaction Management

- Can be switched on by the annotation
  @TransactionManagement (TransactionManagerType.BEAN)
- JTA can be used to define transactions explicitly in the sourcecode

```
@TransactionManagement(TransactionManagementType.BEAN)

public class CustomerManager {

 @Resource UserTransaction ut;

   public Customer createCustomer(String name){

   try{
     ut.begin();
       ...
     ut.commit();
   }
 ...
```

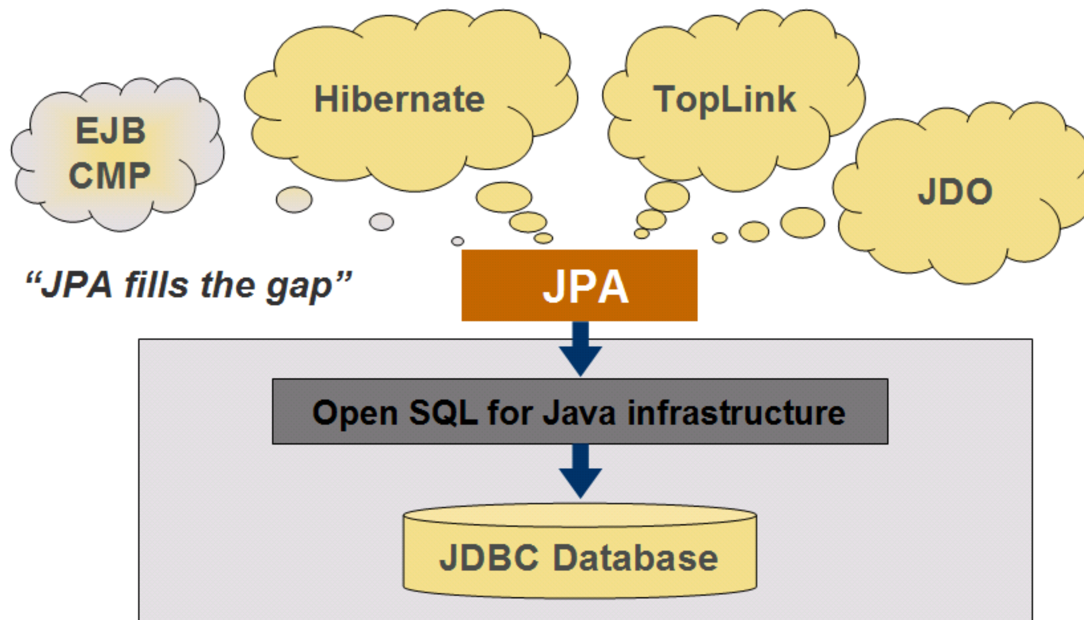
EJB provides a comprehensive and simple mechanism for delimiting transactions at the method level and propagating transactions automatically in a declarative way.

© 2008 SAP AG

Java Persistence API (JPA 1.0)

JPA – The New Java Persistence API

- Unified object persistence for Java
- Standardized object-relational mapping
- Based on ideas of existing persistence frameworks



JPA – Key Features

POJO Persistence

- Plain Java classes
- No required interfaces
- Supports inheritance of entities

Standardized object-relational mapping

- Simple configuration
- Annotation- or XML-based
- Default rules → Configuration by exception

Application makes explicit API calls

Entities themselves can be used for data transfer

- Detach/merge mechanism

© 2008 SAP AG

JPA – Entity Example

```
@Table(name="TMP_EMP") //point : 4
@Entity //point : 2
public class Employee { // point : 1
@Id // point : 3
private int id;
private String name;
public int getId() {
        return id;
    }
public void setId(int theId) {
        id = theId;
    }
public String getName() {
        return name;
    }
public void setName(String theName) {
        name = theName;
    }
}
```

1. Start with a regular Java class (JavaBean conventions)
2. Declare as entity
3. Declare primary key
4. Override defaults (Optional)

© 2008 SAP AG

JPA: How Applications Control the State of Entities

Applications use the EntityManager API to handle entities

```
@Stateless public class HRSesBean {
  @PersistenceContext(unitName="HR")  //Injection of a container-managed entity manager
  EntityManager em;
  public void createEmployee(String name, Long id) {
   Employee employee = new Employee(name, id);
    em.persist(employee);
  }
}
```

Main API for synchronizing entities and their persistent state

- life cycle methods (persist, find, update, remove, flush, merge, …)
- factory for queries

Integrated with EJB 3.0 ➔ Container-managed entity managers

JPA: Entity Manager

Main API for application programmers / clients

- create, update, delete entities
- force database update
- read entities
- (...)

```
package javax.persistence;
public interface EntityManager {
   public void persist(Object entity);
   public <T> T merge(T entity);
   public void remove(Object entity);

   public void flush();
   public Query createQuery(String qlString);
   public Query createNamedQuery(String name);
   public <T> T find(Class<T> entityClass, Object primaryKey);
   // ...
}
```


© 2008 SAP AG

JPA: Persistence Context

```
@PersistenceContext(unitname="HR") private EntityManager em;
```

Set of managed entity instances within an entity manager

Entities become managed by

- passing them explicitly to the entity manager
- reading them from the database via entity manager or query API

Entities are unique within persistence context

Only managed entities are synchronized with DB

JPA: Persistence Context Types

```
@PersistenceContext(unitName="GuestBook",   type=PersistenceContextType.TRANSACTION
| EXTENDED)

EntityManager em;
```

Default Value = TRANSACTION

- Persistence contexts live as long as the transaction exists.
- If the transaction completes (commit), the transaction-scoped persistence context will be destroyed and all managed entities will be detached and synchronized with the DB

Extended Persistence Context

- It can live longer than a transaction
- Managed objects remain managed after method call.
- Is used for stateful session beans

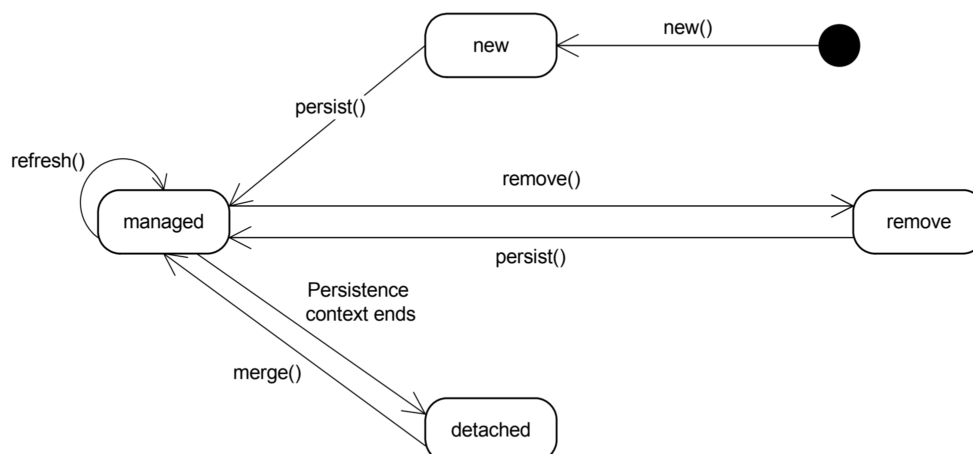
JPA: Detaching and Merging Entities

Entity instances are called detached if

- they have a representation in the database and
- are not managed, i.e. not contained in the persistence context

Persistence context close → entities detached

Entities can be updated and merged back into persistence context



© 2008 SAP AG

- Refresh: read entity from database, overwrite changes
- New: the entity instance was created in memory, it is not yet associated with the persistence context or an instance in the DB
- Managed: the entity has a persistent identity in the DB and is currently associated with the persistence context. Entity changes will be synchronized with the database when the transaction ends, or the flush() operation is called.
- Detached: the entity is stored in the DB, but is not associated with the persistence context
- Removed: the entity is associated with a persistence context, but is scheduled for removal from the database

JPA: Queries

- Named queries (static → recommended)

```
@NamedQuery(name = "findAllEmployees", query = "SELECT e FROM Employee e");

...

List<Employee> employees = em.createNamedQuery ("findAllEmployees").getResultList();
```

- Dynamic queries

```
Query query = em.createQuery(queryText.toString());

...

List<Employee> employees = query.getResultList();
```

Java Persistence Query Language (JPQL)

- SQL-like, rich in features (JOIN, GROUP BY, HAVING, bulk update/delete, parameters, …)
- as an alternative, JPA also supports native SQL statements

© 2008 SAP AG

JPA: Other Powerful Features for Entities

Relationships between entities by annotations to appropriate fields

- @OneToOne, @ManyToOne, @OneToMany, @ManyToMany
- bidirectional or unidirectional, lazy or eager fetching, cascading, ...

```
@OneToMany(cascade=ALL, fetch=FetchType.EAGER)
public List<Employee> getEmployees() {
return employees;
}
```

Automatic primary key generation

```
@Id
@GeneratedValue(strategy=AUTO)
Long id;
```

Optimistic verification to detect update conflicts

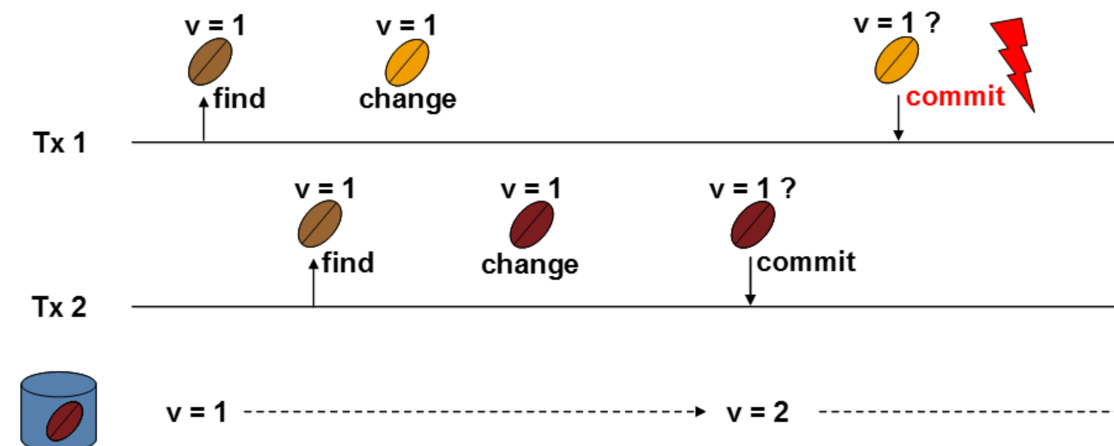
```
@Version
protected int version;
```

JPA: Concurrency Control

Optimistic Locking

- A transaction fails if a concurrent transaction has changed the same data
- Requires a version attribute

```
@Version
private int version;
```



© 2008 SAP AG

JPA: ID Generation

Identifier values (primary keys) can be generated automatically

- Integral primary keys
- Clusterwide unique
- Different generation types
  - TABLE

```
@TableGenerator(name="ID_GEN", table="TMP_ID_TABLE",
pkColumnName="GEN_NAME",

valueColumnName="GEN_VAL")

@Id

@GeneratedValue(generator="ID_GEN")

private int id;
```

TMP_ID_TABLE

GEN_NAME	GEN_VAL

- AUTO (default, for rapid prototyping – in SAP implementation: table based)

- SEQUENCE, IDENTITY (not available in SAP implementation)

JPA: Some Elementary Schema Mappings

- @Table: Defines the table where the bean class maps to.
- @Column: Defines the column where the bean property is stored.
- @ Id: The primary key your bean property maps to (primary key classes and composite keys are also possible)
- @GeneratedValue: Defines that this bean property has its value generated using an generator
- @Transient: Bean properties that should not be persistent.
- @Basic: Defines this field as beeing persistent. Used mostly to define the fetch type of the bean property.
- @Temporal: Additional information about the mapping of Date or Calendar (mapping to DB types date, time, timestamp)
- @Lob: Defines the bean property as being stored as large object (mapped to BLOB or CLOB depending on the Java type)

© 2008 SAP AG

JPA: Mapping example

```
@Entity

@Table(name="CSC_CUSTOMER")

public class Customer implements Serializable {

@Id

@GeneratedValue(strategy=GenerationType.TABLE, generator = "CSCGenerator")

@TableGenerator(name="CSCGenerator", table="CSC_GENKEY")

private int id;

@Column(name="CUSTOMER_NAME")

private String name;

@Basic(fetch=FetchType.LAZY)
  private Address address;

@Transient
  private int internalNumber;

@Lob
  private byte[] picture;

@Temporal(TemporalType.TIME)
  private Date lastContact;

...
```


© 2008 SAP AG

Entity Relationships

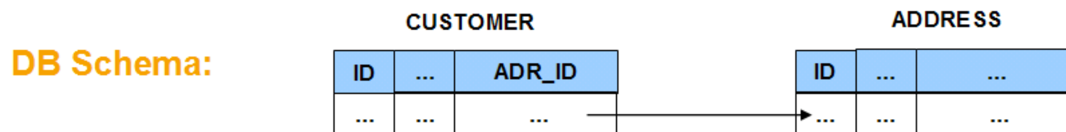
There are seven relationship types:

- One-to-one unidirectional
- One-to-one bidirectional
- One-to-many unidirectional
- One-to-many bidirectional / Many-to-one bidirectional
- Many-to-one unidirectional
- Many-to-many unidirectional
- Many-to-many bidirectional

One to one unidirectional example

A Customer has one address:

DB Schema:



Programming model

```
@Entity
public class Customer {
//...
@OneToOne
@JoinColumn(name="ADR_ID")
private Address address;
//...
}
```

The address entity has no relation back to the customer, since this is the unidirectional case

© 2008 SAP AG

One to one bidirectional

An Employee has one parking space

DB Schema:

Programming model

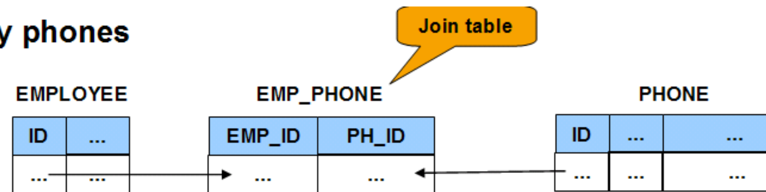
```
@Entity

public class Employee {

//...

@OneToOne

@JoinColumn(name="PARK_ID")

private ParkingSpace parkSpace;

//...

}


@Entity

public class ParkingSpace {

//...

@OneToOne(mappedBy="parkSpace")

private Employee employee ;

//...

}
```


© 2008 SAP AG

One to many unidirectional

An Employee has many phones

DB Schema:



Programming model

```
@Entity
public class Employee {
//...
@OneToMany
@JoinTable(name="EMP_PHONE",joinColumns=@JoinColumn(name="EMP_ID"),
inverseJoinColumns=@JoinColumn(name="PH_ID"))
private List<Phone> phones;
//...
}
```

The phone entity has no relation back to the employee, since this is the unidirectional case

Note: Some vendor may provide support for an unidirectional one-to-many foreign key mapping, where a FK column in the PHONE table exist. This is not supported in the current version of JPA !

© 2008 SAP AG

One to many bidirectional

A Department has many Employees

DB Schema:

Programming model

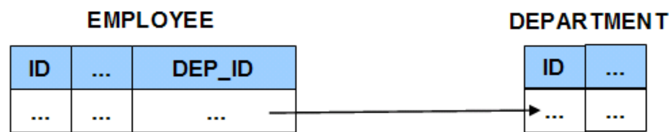
```
@Entity
public class Department {
//...
@OneToMany(mappedBy="department")
private List<Employee> employees;
//...
}

@Entity
public class Employee {
//...
@ManyToOne
@JoinColumn(name="DEP_ID")
private Department department ;
//...
}
```


© 2008 SAP AG

Many to one unidirectional

Many Employees belong to one Department (but the department does not know its employees)

DB Schema:



Programming model

```
@Entity
public class Employee {
//...
@ManyToOne
@JoinColumn(name="DEP_ID")
private Department department ;
//...
}
```

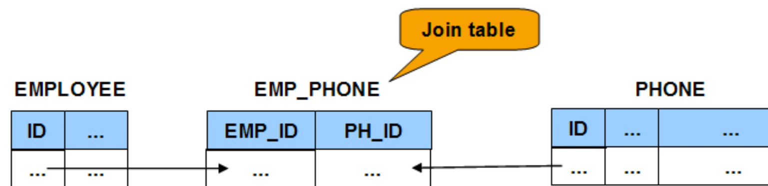
The Department entity has no relation back to the Employee, since this is the unidirectional case.

© 2008 SAP AG

Many to many unidirectional

Many Employees can have many Phones (the phone entities do not know to which Employee they belong

DB Schema:



Programming model

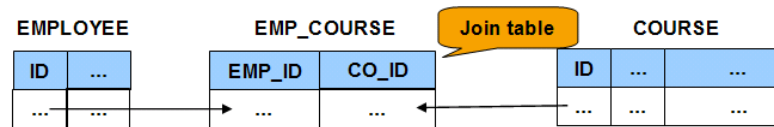
```
@Entity
public class Employee {
//...
@ManyToMany
@JoinTable(name="EMP_PHONE",joinColumns=@JoinColumn(name="EMP_ID"),
inverseJoinColumns=@JoinColumn(name="PH_ID"))
private List<Phone> phones;
//...
}
```

The Phone entities do have no relation back to the Employee, since this is the unidirectional case.

© 2008 SAP AG

Many to many bidirectional

Many Employees can book many Courses



Programming model

```
@Entity
public class Employee {
//...
@ManyToMany
@JoinTable(name="EMP_COURSE",joinColumns=@JoinColumn(name="EMP_ID"),
inverseJoinColumns=@JoinColumn(name="CO_ID"))
private List<Course> courses;
//...
}

@Entity
public class Course {
//...
@ManyToMany(mappedBy="courses")
private List<Employee> employees;
//...
}
```

Cascading

@OneToOne(cascade={CascadeType.PERSIST})

- No cascading by default
- Types:
  - PERSIST, MERGE, REMOVE, REFRESH, ALL

© 2008 SAP AG

Related Content

http://java.sun.com/javaee/5/docs/tutorial/doc/

For more information, visit the Java homepage.

© 2008 SAP AG

Disclaimer and Liability Notice

This document may discuss sample coding or other information that does not include SAP official interfaces and therefore is not supported by SAP. Changes made based on this information are not supported and can be overwritten during an upgrade.

SAP will not be held liable for any damages caused by using or misusing the information, code or methods suggested in this document, and anyone using these methods does so at his/her own risk.

SAP offers no guarantees and assumes no responsibility or liability of any type with respect to the content of this technical article or code sample, including any liability resulting from incompatibility between the content within this document and the materials and services offered by SAP. You agree that you will not hold, or seek to hold, SAP responsible or liable with respect to the content of this document.

© 2008 SAP AG