Java Enterprise Edition

IQSOFT-John Bryce Training Center

Dr. Attila Vincze

Aug 21 - Aug 25, 2016

Outline I

JPA I

The Java Persistence API:

- Replaces EJB 2.1 entity beans with non-EJB entity classes
- Is a standard API for specifying object-to-relational mapping information
- Can be used with or without a Java EE Application Server
  - Container-managed persistence
  - Application-managed persistence

Object-relational mapping (ORM) software:

- Provides an object-oriented view of the database
- Examples include EclipseLink and Hibernate

JPA II

JPA is an abstraction above JDBC that makes it possible to be independent of SQL. All classes and annotations of this API are in the javax.persistence package. The main components of JPA are

- ORM, which is the mechanism to map objects to data stored in a relational database.
- An entity manager API to perform database-related operations, such as CRUD.
- The JPQL, which allows you to retrieve data with an object-oriented query language.
- Transactions and locking mechanisms which Java Transaction API (JTA) provides when accessing data concurrently. JPA also supports resource-local (non-JTA) transactions.
- Callbacks and listeners to hook business logic into the life cycle of a persistent object.

JPA III

A Brief History of JPA

- ORM solutions have been around for a long time, even before Java.
- Products such as TopLink originally started with Smalltalk in 1994 before switching to Java. Commercial ORM products like TopLink have been available since the earliest days of the Java language.
- In 1998, EJB 1.0 was created and later shipped with J2EE 1.2. It was a heavyweight, distributed component used for transactional business logic. Entity Bean CMP was introduced as optional in EJB 1.0, became mandatory in EJB 1.1, and was enhanced through versions up to EJB 2.1 (J2EE 1.4). Persistence could only be done inside the container through a complex mechanism of instantiation using home, local, or remote interfaces.
- Parallel to the J2EE world was a popular open source solution that led to some surprising changes in the direction of persistence: Hibernate, which brought back a lightweight, object-oriented persistent model.
- After years of complaints about Entity CMP 2.x components and in acknowledgment of the success and simplicity of open source frameworks such as Hibernate, the persistence model of the Enterprise Edition was completely rearchitected in Java EE 5.
- JPA 1.0 was born with a very lightweight approach that adopted many Hibernate design principles. The JPA 1.0 specification was bundled with EJB 3.0 (JSR 220).

JPA IV

- In 2009 JPA 2.0 (JSR 317) shipped with Java EE 6 and brought new APIs, extended JPQL, and added new functionalities such as second-level cache, pessimistic locking, or the criteria API.
- With Java EE 7, JPA 2.1 follows the path of ease of development and brings new features.

JPA V

What's New in JPA 2.1?

- Schema generation: JPA 2.1 has standardized database schema generation by bringing a new API and a set of properties (defined in the persistence.xml).
- Converters: These are new classes that convert between database and attributes representations.
- CDI support: Injection is now possible into event listeners.
- Support for stored procedures: JPA 2.1 allows now dynamically specified and named stored procedure queries.
- Bulk update and delete criteria queries: Criteria API only had select queries; now update and delete queries are also specified.
- Downcasting: The new TREAT operator allows access to subclass-specific state in queries.

JPA VI

Reference Implementation

- EclipseLink 2.5 is the open source reference implementation of JPA 2.1.
- EclipseLink's origins stem from the Oracle TopLink product given to the Eclipse Foundation in 2006.

JPA VII

The principle of object-relational mapping (ORM) is to bring the world of database and objects together. It involves delegating access to relational databases to external tools or frameworks, which in turn give an object- oriented view of relational data, and vice versa.

- **Persistent data** are everywhere, and most of the time they use relational databases as the underlying persistence engine (as opposed to schemaless databases). **Relational databases** store data in tables made of rows and columns. Data are identified by primary keys, which are special columns with uniqueness constraints and, sometimes, indexes. The relationships between tables use foreign keys and join tables with integrity constraints.
- **In Java**, we manipulate objects that are instances of classes. Objects inherit from others, have references to collections of other objects, and sometimes point to themselves in a recursive manner. We have concrete classes, abstract classes, interfaces, enumerations, annotations, methods, attributes, and so on.

JPA VIII

Entities vs. objects

- Entities are objects that **live shortly in memory and persistently in a database**. They have the ability to be mapped to a database; they can be concrete or abstract; and they support inheritance, relationships, and so on.
- Entities, once mapped, can be managed by JPA.
- You can persist an entity in the database, remove it, and query it using a query language Java Persistence Query Language, or JPQL).

Simple Book entity

```
@Entity
public class Book {
      @Id @GeneratedValue
      private Long id;
      private String title;
      private Float price;
      private String description; private String isbn;
      private Integer nbOfPage; private Boolean illustrations;
      public Book() {
      }
      // Getters, setters
}
```

JPA IX

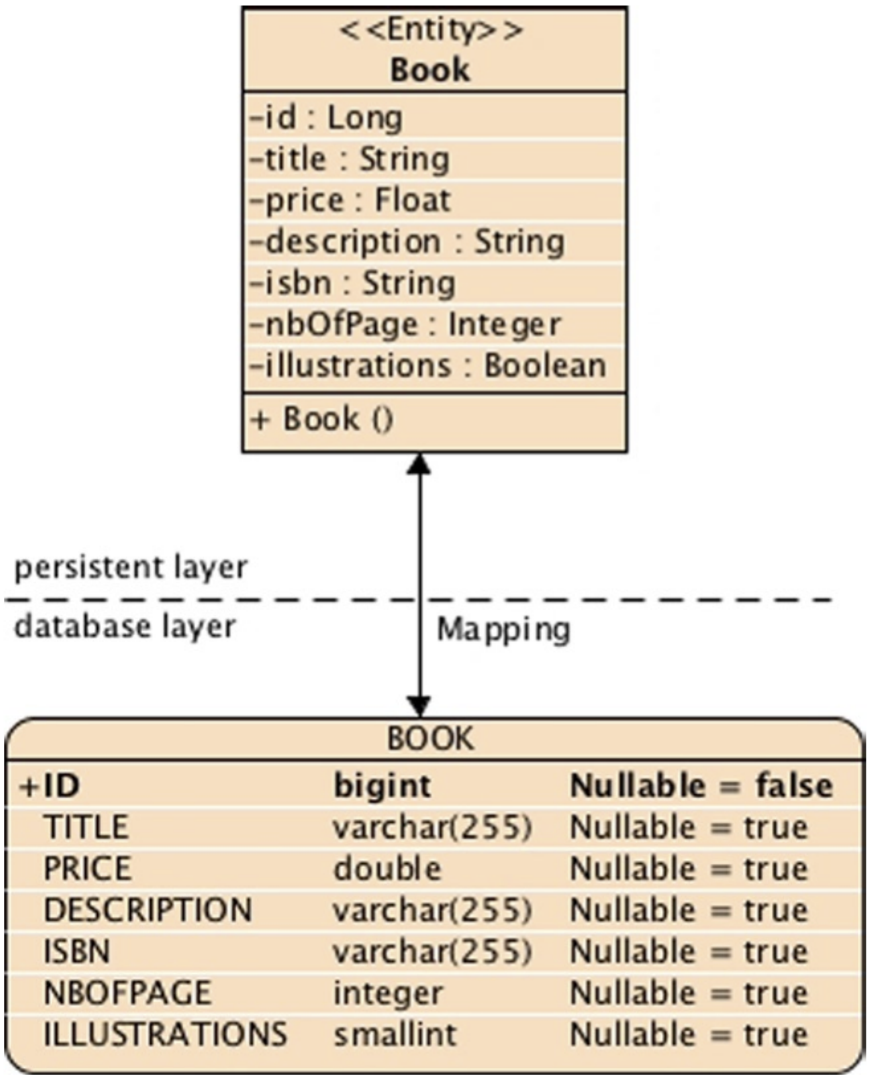
An entity class must follow these rules

- The entity class must be annotated with `@javax.persistence.Entity` (or denoted in the XML descriptor as an entity).
- The `@javax.persistence.Id` annotation must be used to denote a simple primary key.
- The entity class must have a **no-arg constructor** that has to be **public** or **protected**. The entity class may have other constructors as well.
- The entity class must be a top-level class. **An enum or interface cannot be designated as an entity**.
- The entity class **must not be final**. No methods or persistent instance variables of the entity class may be final.
- If an entity instance has to be passed by value as a detached object (e.g., through a remote interface), the entity class must implement the **Serializable** interface.

ORM is done through metadata. It can be written in two different formats.

- **Annotations**: The code of the entity is directly annotated with all sorts of annotations that are described in the javax.persistence package.
- **XML descriptors**: Instead of (or in addition to) annotations, you can use XML descriptors. The mapping is defined in an external XML file that will be deployed with the entities. This can be very useful when database configuration changes depending on the environment, for example.

JPA X


<<Entity>>
Book
-id : Long
-title : String
-price : Float
-description : String
-isbn : String
-nbOfPage : Integer
-illustrations : Boolean
+ Book ()
persistent layer
database layer
Mapping
BOOK
+ID bigint Nullable = false
TITLE varchar(255) Nullable = true
PRICE double Nullable = true
DESCRIPTION varchar(255) Nullable = true
ISBN varchar(255) Nullable = true
NBOFPAGE integer Nullable = true
ILLUSTRATIONS smallint Nullable = true

JPA XI

Java EE 5 introduced the idea of **configuration by exception** (sometimes referred to as programming by exception or convention over configuration) and is still heavily used today in Java EE 7. This means, unless specified differently, the container or **provider should apply the default rules**. In other words, **having to supply a configuration is the exception to the rule**.

- The **entity name** is mapped to a relational table name (e.g., the Book entity is mapped to a BOOK table). If you want to map it to another table, you will need to use the `@Table` annotation.
- **Attribute names** are mapped to a column name (e.g., the id attribute, or the getId() method, is mapped to an ID column). If you want to change this default mapping, you will need to use the `@Column` annotation.
- JDBC rules apply for mapping Java primitives to relational data types. A String will be mapped to VARCHAR, a Long to a BIGINT, a Boolean to a SMALLINT, and so on. The default size of a column mapped from a String is 255 (a String is mapped to a VARCHAR(255)).

```
CREATE TABLE BOOK (
ID BIGINT NOT NULL,
TITLE VARCHAR(255),
PRICE FLOAT,
DESCRIPTION VARCHAR(255),
ISBN VARCHAR(255),
NBOFPAGE INTEGER,
ILLUSTRATIONS SMALLINT DEFAULT 0, PRIMARY KEY (ID)
)
```

JPA XII

Entity manager

- The central piece of the API responsible for orchestrating entities is the `javax.persistence.EntityManager`.
- Its role is to manage entities, read from and write to a given database, and allow simple CRUD (create, read, update, and delete) operations on entities as well as complex queries using JPQL.
- In a technical sense, the entity manager is just an interface whose implementation is done by the persistence provider (e.g., EclipseLink).

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter04PU");
EntityManager em = emf.createEntityManager();
em.persist(book);
```

JPA XIII

```
@Entity
@NamedQuery(name = "findBookH2G2", query = "SELECT b FROM Book b WHERE b.title ='H2G2'")
public class Book {
      @Id @GeneratedValue
      private Long id;
      private String title;
      private Float price;
      private String description;
      private String isbn;
      private Integer nbOfPage;
      private Boolean illustrations;
      // Constructors, getters, setters
}
```

JPA XIV

```
public class Main {
      public static void main(String[] args) {
            // 1-Creates an instance of book
            Book book = new Book("H2G2", "The Hitchhiker's Guide to the Galaxy", 12.5F, "1-84023-742
            // 2-Obtains an entity manager and a transaction
            EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter04PU");
            EntityManager em = emf.createEntityManager();
            // 3-Persists the book to the database
            EntityTransaction tx = em.getTransaction();
            tx.begin();
            em.persist(book);
            tx.commit();
            // 4-Executes the named query
            book = em.createNamedQuery("findBookH2G2", Book.class).getSingleResult();
            // 5-Closes the entity manager and the factory em.close();
            emf.close();
      }
}
```

JPA XV

Persistent Unit

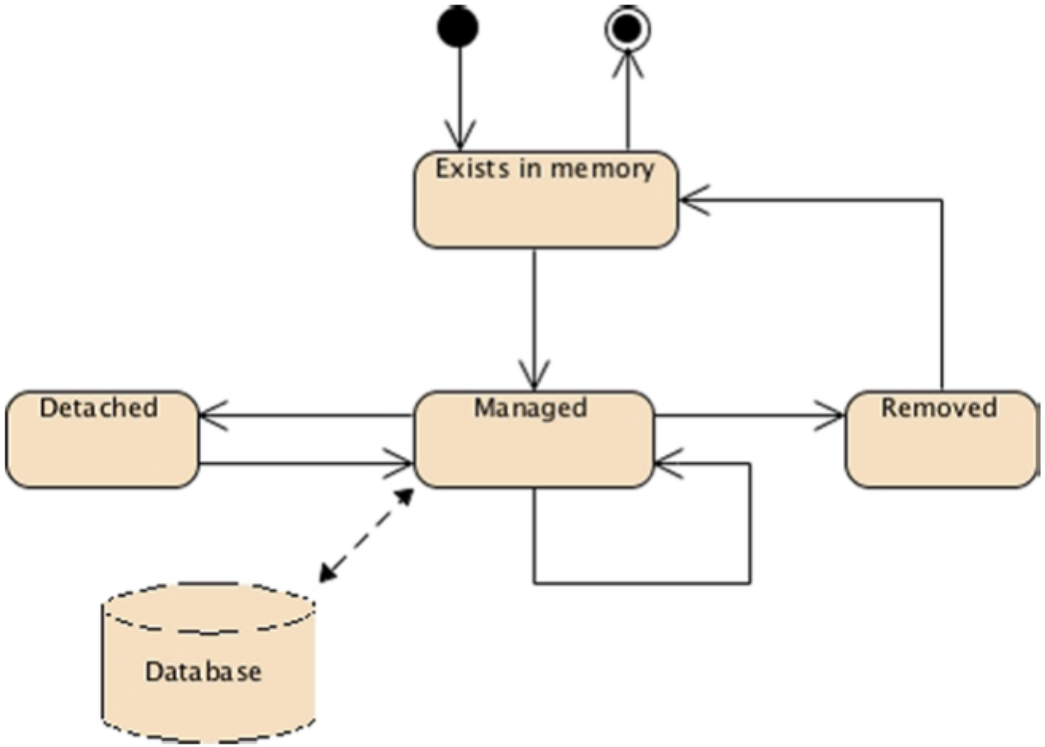
- Which JDBC driver should you use?
- How should you connect to the database?
- What's the database name?
- The persistence unit indicates to the entity manager the type of database to use and the connection parameters, which are defined in the `persistence.xml`.

JPA XVI

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">
    <persistence-unit name="chapter04PU" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>org.agoncal.book.javaee7.chapter04.Book</class>
        <properties>
            <property name="javax.persistence.schema-generation-action" value="drop-and-create"/>
            <property name="javax.persistence.schema-generation-target" value="database"/>
            <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:derby://localhost:1527/chapter04DB;create=true"/>
            <property name="javax.persistence.jdbc.user" value="APP"/>
            <property name="javax.persistence.jdbc.password" value="APP"/>
        </properties>
    </persistence-unit>
</persistence>
```

JPA XVII

Entity Life Cycle and Callbacks



- `@PrePersist`
- `@PostPersist`

JPA XVIII

Integration with Bean Validation

- Entities may include Bean Validation constraints and be automatically validated.
- In fact, automatic validation is achieved because JPA delegates validation to the Bean Validation implementation.

```
@Entity
public class Book {
      @Id @GeneratedValue
      private Long id;
      @NotNull
      private String title;
      private Float price;
      @Size(min = 10, max = 2000)
      private String description;
      private String isbn;
      private Integer nbOfPage;
      private Boolean illustrations;
      // Constructors, getters, setters
}
```

Object-relational mapping I

The configuration by exception allows the JPA provider to map an entity to a database table using all the defaults. But defaults are not always suitable, especially if you map your domain model to an existing database. JPA comes with a rich set of metadata so you can customize the mapping.

Object-relational mapping II

Tables

- Rules for configuration-by-exception mapping state that the entity and the table name are the same (a **Book** entity is mapped to a **BOOK** table).
- The `@javax.persistence.Table` annotation makes it possible to change the default values related to the table.
- You can also define unique constraints to the table using the `@UniqueConstraint` annotation.

```
@Entity
@Table(name = "t_book")
public class Book {
      @Id
      private Long id;
      private String title;
      private Float price;
      private String description;
      private String isbn;
      private Integer nbOfPage;
      private Boolean illustrations;
      // Constructors, getters, setters
}
```

Object-relational mapping III

`@SecondaryTable`

- Up to now, I have assumed that an entity gets mapped to a single table, also known as a primary table.
- But sometimes when you have an existing data model, you need to spread the data across multiple tables, or secondary tables.
- When you use secondary tables, you must consider the issue of performance. Every time you access an entity, the persistence provider accesses several tables and has to join them.
- On the other hand, secondary tables can be a good thing when you have expensive attributes such as binary large objects (BLOBs) that you want to isolate in a different table.

Object-relational mapping IV

```
@Entity
@SecondaryTables({
      @SecondaryTable(name = "city"),
      @SecondaryTable(name = "country")
})
public class Address {
      @Id
      private Long id;
      private String street1;
      private String street2;
      @Column(table = "city")
      private String city;
      @Column(table = "city")
      private String state;
      @Column(table = "city")
      private String zipcode;
      @Column(table = "country")
      private String country;
      // Constructors, getters, setters
}
```

Object-relational mapping V

```
@Entity
@Table(name = "t_address")
@SecondaryTables({
      @SecondaryTable(name = "t_city"),
      @SecondaryTable(name = "t_country")
})
public class Address {
      // Attributes, constructor, getters, setters
}
```

Object-relational mapping VI

Primary Keys

- In relational databases, a primary key **uniquely identifies each row** in a table.
- It comprises either a **single column** or **set of columns**.
- **JPA requires entities to have an identifier mapped to a primary key**, which will follow the same rule: uniquely identify an entity by either a single attribute or a set of attributes (composite key).
- This entity's primary key value **cannot be updated** once it has been assigned.

`@javax.persistence.Id` annotates an attribute as being a unique identifier. It can be one of the following types:

- Primitive Java types: `byte, int, short, long, char`
- Wrapper classes of primitive Java types: `Byte, Integer, Short, Long, Character`
- Arrays of primitive or wrapper types: `int[], Integer[]`, etc.
- Strings, numbers, and dates: `java.lang.String, java.math.BigInteger, java.util.Date, java.sql.Date`

Object-relational mapping VII

When creating an entity, the value of this identifier can be generated either manually by the application or automatically by the persistence provider using the `@GeneratedValue` annotation. This annotation can have four possible values.

- **SEQUENCE** and **IDENTITY** specify use of a database SQL sequence or identity column, respectively.
- **TABLE** instructs the persistence provider to store the sequence name and its current value in a table, increasing the value each time a new instance of the entity is persisted. As an example, EclipseLink creates a SEQUENCE table with two columns.
- The generation of a key is done automatically (**AUTO**) by the underlying persistence provider, which will pick an appropriate strategy for a particular database (EclipseLink will use the TABLE strategy).

Object-relational mapping VIII

```
@Entity
public class Book {
      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      private Long id;
      private String title;
      private Float price;
      private String description;
      private String isbn;
      private Integer nbOfPage;
      private Boolean illustrations;
      // Constructors, getters, setters
}
```

Object-relational mapping IX

Composite Primary Keys:

`@EmbeddedId`

```
@Embeddable
public class NewsId {
      private String title;
      private String language;
      // Constructors, getters, setters,
      //equals, and hashcode
}
```

```
@Entity
public class News {
      @EmbeddedId
      private NewsId id;
      private String content;
      // Constructors, getters, setters
      //equals and hash
```

```
NewsId pk = new NewsId("Richard Wright has died on September 2008", "EN");
News news = em.find(News.class, pk);
```

Object-relational mapping X

@IdClass

```
public class NewsId {
      private String title;
      private String language;
      // Constructors, getters, setters,
      //equals, and hashcode
```

```
@Entity
@IdClass(NewsId.class)
public class News {
      @Id
      private String title;
      @Id
      private String language;
      private String content;
      // Constructors, getters, setters
}
```

Object-relational mapping XI

Both approaches, **@EmbeddedId** and **@IdClass**, will be mapped to the same table structure.

```
create table NEWS (
CONTENT VARCHAR(255),
TITLE VARCHAR(255) not null,
LANGUAGE VARCHAR(255) not null,
primary key (TITLE, LANGUAGE)
);
```

The `@IdClass` approach is more prone to error, as you need to define each primary key attribute in both the `@IdClass` and the entity, taking care to use the same name and Java type.

- `@IdClass`: select n.title from News n
- `@EmbeddedId`: select n.**newsId**.title from News n

Object-relational mapping XII

Attributes

- Java primitive types (`int, double, float`, etc.) and the wrapper classes (`Integer, Double, Float`, etc.)
- Arrays of bytes and characters (`byte[], Byte[], char[], Character[]`)
- String, large numeric, and temporal types (`java.lang.String, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp`)
- Enumerated types and user-defined types that implement the `Serializable` interface
- Collections of basic and embeddable types

Object-relational mapping XIII

@javax.persistence.Basic

```
@Target({
      METHOD, FIELD})
@Retention(RUNTIME)
public @interface Basic {
      FetchType fetch() default EAGER;
      boolean optional() default true;
}
```

```
@Entity
public class Track {
      @Id @GeneratedValue(strategy = GenerationType.AUTO)
      private Long id;
      private String title;
      private Float duration;
      @Basic(fetch = FetchType.LAZY)
      @Lob
      private byte[] wav;
      private String description;
      // Constructors, getters, setters
}
```

Object-relational mapping XIV

@javax.persistence.Column

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Column {
  String name() default "";
  boolean unique() default false;
  boolean nullable() default true;
  boolean insertable() default true;
  boolean updatable() default true;
  String columnDefinition() default "";
  String table() default "";
  int length() default 255;
  int precision() default 0; // decimal precision
  int scale() default 0;     // decimal scale
}
```

Object-relational mapping XV

```
@Entity
public class Book {
      @Id @GeneratedValue(strategy = GenerationType.AUTO)
      private Long id;
      @Column(name = "book_title", nullable = false, updatable = false)
      private String title;
      private Float price;
      @Column(length = 2000)
      private String description;
      private String isbn;
      @Column(name = "nb_of_page", nullable = false)
      private Integer nbOfPage;
      private Boolean illustrations;
      // Constructors, getters, setters
}
```

Note that not all the combinations of attributes on datatypes are valid (e.g., length only applies to string-valued column, scale and precision only to decimal column).

Object-relational mapping XVI

```
create table BOOK (
  ID BIGINT not null,
  BOOK_TITLE VARCHAR(255) not null,
  PRICE DOUBLE(52, 0),
  DESCRIPTION VARCHAR(2000),
  ISBN VARCHAR(255),
  NB_OF_PAGE INTEGER not null,
  ILLUSTRATIONS SMALLINT,
  primary key (ID)
);
```

Most of the elements of the @Column annotation have an influence on the mapping. Others affect the dynamic behavior of the entity manager when accessing the relational data.

Object-relational mapping XVII

@javax.persistence.Temporal

```
@Entity
public class Customer {
      @Id @GeneratedValue
      private Long id;
      private String firstName;
      private String lastName;
      private String email;
      private String phoneNumber;
      @Temporal(TemporalType.DATE)
      private Date dateOfBirth;
      @Temporal(TemporalType.TIME)
      private Date lastAction;
      @Temporal(TemporalType.TIMESTAMP)
      private Date creationDate;
      // Constructors, getters, setters
}
```

Object-relational mapping XVIII

```
create table CUSTOMER (
ID BIGINT not null,
FIRSTNAME VARCHAR(255),
LASTNAME VARCHAR(255),
EMAIL VARCHAR(255),
PHONENUMBER VARCHAR(255),
DATEOFBIRTH DATE,
LASTACTION TIME,
CREATIONDATE TIMESTAMP,
primary key (ID)
);
```

Object-relational mapping XIX

@javax.persistence.Transient

```
@Entity
public class Customer {
      @Id @GeneratedValue
      private Long id;
      private String firstName;
      private String lastName;
      private String email;
      private String phoneNumber;
      @Temporal(TemporalType.DATE)
      private Date dateOfBirth;
      @Transient
      private Integer age;
      @Temporal(TemporalType.TIMESTAMP)
      private Date creationDate;
      // Constructors, getters, setters
}
```

Object-relational mapping XX

@Enumerated

```
public enum CreditCardType {
	VISA,
	MASTER_CARD,
	AMERICAN_EXPRESS
}
```

By default, the persistence providers will map this enumerated type to the database assuming that the column is of type Integer.

```
@Entity
@Table(name = "credit_card")
public class CreditCard {
	@Id
	private String number;
	private String expiryDate;
	private Integer controlNumber;
	private CreditCardType creditCardType;
	// Constructors, getters, setters
}
```

Imagine introducing a new constant to the top of the enumeration. Because ordinal assignment is determined by the order in which values are declared, the values already stored in the database will no longer match the enumeration.

Object-relational mapping XXI

```
@Entity
@Table(name = "credit_card")
public class CreditCard {
	@Id
	private String number;
	private String expiryDate;
	private Integer controlNumber;
	@Enumerated(EnumType.STRING)private CreditCardType creditCardType;// Constructors, getters, setters
```

Object-relational mapping XXII

Access Type

- Annotations can be applied on an **attribute** (or field access).
- Annotations can be applied on **getter** method (or property access).

```
@Entity
public class Customer {
      @Id @GeneratedValue
      private Long id;
      @Column(name = "first_name", nullable = false, length = 50)
      private String firstName;
      @Column(name = "last_name", nullable = false, length = 50)
      private String lastName;
      private String email;
      @Column(name = "phonenumber", length = 15)
      private String phoneNumber;
      // Constructors, getters, setters
}
```

The Customer Entity with Annotated Fields

Object-relational mapping XXIII

```
@Entity
public class Customer {
	private Long id;
	private String firstName;
	private String lastName;
	private String email;
	private String phoneNumber;
	// Constructors
	@Id @GeneratedValue
	public Long getId() {
		return id;
	}
	public void setId(Long id) {
		this.id = id;
	}
	@Column(name = "first_name", nullable = false, length = 50)
	public String getFirstName() {
		return firstName;
	}
	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}
	@Column(name = "last_name", nullable = false, length = 50)
	public String getLastName() {
		return lastName;
	}
	public void setLastName(String lastName) {
		this.lastName = lastName;
```

Object-relational mapping XXIV

```
        }
        public String getEmail() {
                return email;
        }
        public void setEmail(String email) {
                this.email = email;
        }
        @Column(name = "phone_number", length = 15)
        public String getPhoneNumber() {
                return phoneNumber;
        }
        public void setPhoneNumber(String phoneNumber) {
                this.phoneNumber = phoneNumber;
        }
}
```

The Customer Entity with Annotated Properties

Object-relational mapping XXV

```
@Entity
@Access(AccessType.FIELD)
public class Customer {
      @Id @GeneratedValue
      private Long id;
      @Column(name = "first_name", nullable = false, length = 50)
      private String firstName;
      @Column(name = "last_name", nullable = false, length = 50)
      private String lastName;
      private String email;
      @Column(name = "phone_number", length = 15)
      private String phoneNumber;
      // Constructors, getters, setters

      @Access(AccessType.PROPERTY)
      @Column(name = "phone_number", length = 555)
      public String getPhoneNumber() {
            return phoneNumber;
      }
      public void setPhoneNumber(String phoneNumber) {
            this.phoneNumber = phoneNumber;
      }
}
```

The Customer Entity That Explicitly Mixes Access Types

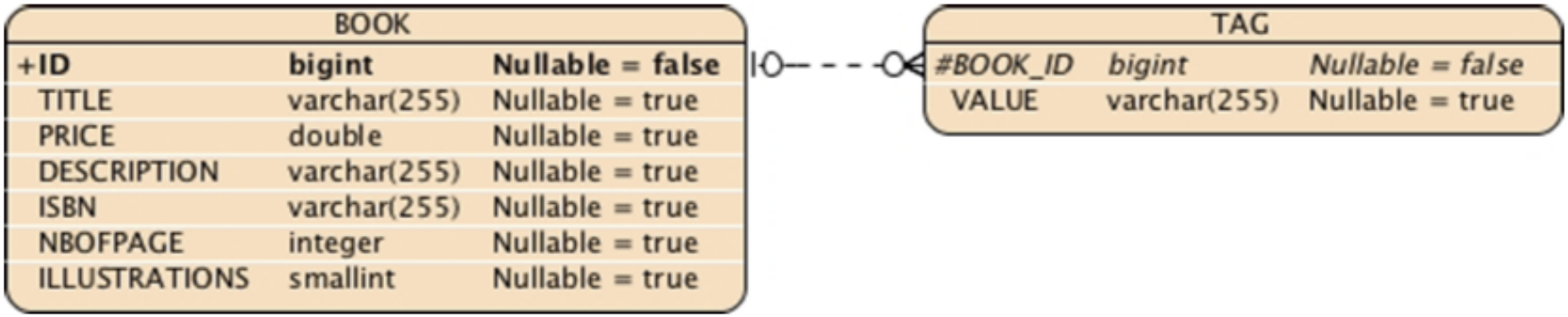
Collection of Basic Types I

@ElementCollection and @CollectionTable annotations

- `@ElementCollection` annotation to indicate that an attribute of type `java.util.Collection` contains a collection of instances of basic types (i.e., nonentities) or embeddables.
- In addition, the `@CollectionTable` annotation allows you to customize details of the collection table (i.e., the table that will join the entity table with the basic types table) such as its name.

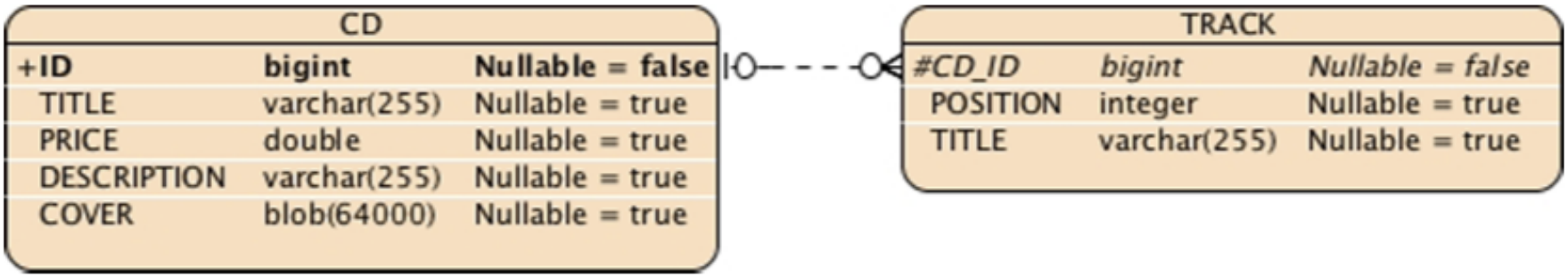
```
@Entity
public class Book {
      @Id @GeneratedValue
      private Long id;
      private String title;
      private Float price;
      private String description;
      private String isbn;
      private Integer nbOfPage;
      private Boolean illustrations;
      @ElementCollection(fetch = FetchType.LAZY)
      @CollectionTable(name = "Tag")
      @Column(name = "Value")
      private List<String> tags = new ArrayList<>();
      // Constructors, getters, setters
}
```

Collection of Basic Types II



Map of Basic Types I

```
@Entity
public class CD {
      @Id @GeneratedValue
      private Long id;
      private String title;
      private Float price;
      private String description;
      @Lob
      private byte[] cover;
      @ElementCollection
      @CollectionTable(name="track")
      @MapKeyColumn (name = "position")
      @Column(name = "title")
      private Map<Integer, String> tracks = new HashMap<>();
      // Constructors, getters, setters
}
```



Mapping with XML I

An important notion to always have in mind is that the XML takes precedence over annotations.

```
@Entity
public class Book {
      @Id @GeneratedValue
      private Long id;
      private String title;
      private Float price;
      @Column(length = 500)
      private String description;
      private String isbn;
      private Integer nbOfPage;
      private Boolean illustrations;
      // Constructors, getters, setters
}
```

```
create table BOOK_XML_MAPPING (
  ID BIGINT not null,
  BOOK_TITLE VARCHAR(255) not null,
  DESCRIPTION VARCHAR(2000),
  NB_OF_PAGE INTEGER not null,
  PRICE DOUBLE(52, 0),
  ISBN VARCHAR(255),
  ILLUSTRATIONS SMALLINT,
  primary key (ID)
);
```

Mapping with XML II

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_1.xsd"
version="2.1">
  <persistence-unit name="chapter05PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>org.agoncal.book.javaee7.chapter05.Book</class>
    <mapping-file>META-INF/book_mapping.xml</mapping-file> <properties>
        <!--Persistence provider properties-->
      </properties>
  </persistence-unit>
</persistence>
```

A persistence.xml File Referring to an External Mapping File

Embeddables I

Embeddables are objects that don't have a persistent identity on their own, they can only be embedded within owning entities. Each attribute of the embedded object is mapped to the table of the entity. It is a strict ownership relationship (a.k.a. composition), so that if the entity is removed, the embedded object is also removed.

```
@Embeddable
public class Address {
      private String street1;
      private String street2;
      private String city;
      private String state;
      private String zipcode;
      private String country;
      // Constructors, getters, setters
}
```

Embeddables II

```
@Entity
public class Customer {
      @Id @GeneratedValue
      private Long id;
      private String firstName;
      private String lastName;
      private String email;
      private String phoneNumber;
      @Embedded
      private Address address;
      // Constructors, getters, setters
}
```

```
create table CUSTOMER (
 ID BIGINT not null,
 LASTNAME VARCHAR(255),
 PHONENUMBER VARCHAR(255),
 EMAIL VARCHAR(255),
 FIRSTNAME VARCHAR(255),
 STREET2 VARCHAR(255),
 STREET1 VARCHAR(255),
 ZIPCODE VARCHAR(255),
 STATE VARCHAR(255),
 COUNTRY VARCHAR(255),
 CITY VARCHAR(255),
 primary key (ID)
);
```

Embeddables III

Access Type of an Embeddable Class

```
@Entity
@Access(AccessType.FIELD)
public class Customer {
      @Id @GeneratedValue
      private Long id;
      @Column(name = "first_name", nullable = false, length = 50)
      private String firstName;
      @Column(name = "last_name", nullable = false, length = 50)
      private String lastName;
      private String email;
      @Column(name = "phonenumber", length = 15)
      private String phoneNumber;
      @Embedded
      private Address address;
      // Constructors, getters, setters
}
```

Embeddables IV

```
@Embeddable
@Access(AccessType.PROPERTY)
public class Address {
      private String street1;
      private String street2;
      private String city;
      private String state;
      private String zipcode;
      private String country;
      // Constructors
      @Column(nullable = false)
      public String getStreet1() {
            return street1;
      }
      public void setStreet1(String street1) {
            this.street1 = street1;
      }
      public String getStreet2() {
            return street2;
      }
      public void setStreet2(String street2) {
            this.street2 = street2;
      }
      @Column(nullable = false, length = 50)
      public String getCity() {
            return city;
      }
      public void setCity(String city) {
```

Embeddables V

```
            this.city = city;
        }
        @Column(length = 3)
        public String getState() {
            return state;
        }
        public void setState(String state) {
            this.state = state;
        }
        @Column(name = "zip_code", length = 10)
        public String getZipcode() {
            return zipcode;
        }
        public void setZipcode(String zipcode) {
            this.zipcode = zipcode;
        }
        public String getCountry() {
            return country;
        }
        public void setCountry(String country) {
            this.country = country;
        }
}
```

Relationship Mapping I

The world of object-oriented programming abounds with classes and associations between classes.

- An association has a **direction**. It can be **unidirectional** (i.e., one object can navigate toward another) or **bidirectional** (i.e., one object can navigate toward another and vice versa).
- An association also has a **multiplicity** (or cardinality). Each end of an association can specify how many referring objects are involved in the association.
- In Java, an association that represents more than one object uses collections of type `java.util.Collection`, `java.util.Set`, `java.util.List`, or even `java.util.Map`.
- A relationship has an ownership (i.e., the owner of the relationship).

Relationship Mapping II

Relationships in Relational Databases

- In JPA when you have an association between one class and another, in the database you will get a **table reference**.
- This reference can be modeled in two different ways: by using a foreign key (a **join column**) or by using a **join table**.
- In database terms, a column that refers to a key of another table is a **foreign key column**.

Entity Relationships

- `@OneToOne, @OneToMany, @ManyToOne, or @ManyToMany`

Table: All Possible Cardinality-Direction Combinations

Cardinality	Direction
One-to-one	Unidirectional
One-to-one	Bidirectional
One-to-many	Unidirectional
Many-to-one	Bidirectional
Many-to-one	Unidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

Relationship Mapping III

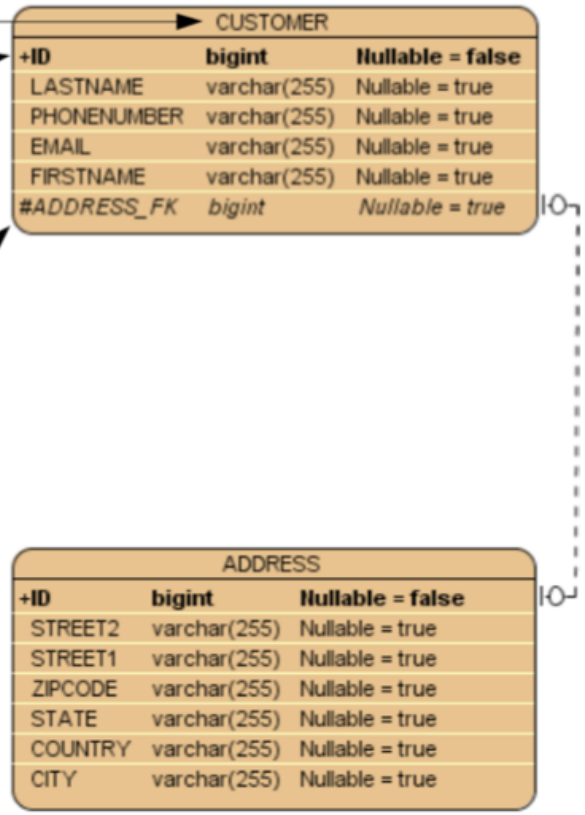
Unidirectional and Bidirectional

```
@Entity
public class Customer

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @OneToOne
    @JoinColumn(name = "address_fk")
    private Address address;
}

@Entity
public class Address {

    @Id @GeneratedValue
    private Long id;
    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipcode;
    private String country;
    @OneToOne(mappedBy = "address")
    private Customer customer;
}
```



Relationship Mapping IV

Notes

- The **mappedBy** element indicates that the join column (address) is specified at the other end of the relationship.
- There is a mappedBy element on the @OneToOne, @OneToMany, and @ManyToMany annotations, but not on the @ManyToOne annotation.
- You cannot have a mappedBy attribute on both sides of a bidirectional association.

Relationship Mapping V

@OneToOne Unidirectional

```
@Entity
public class Customer {
      @Id @GeneratedValue
      private Long id;
      private String firstName;
      private String lastName;
      private String email;
      private String phoneNumber;
      private Address address;
      // Constructors, getters, setters
}
```

```
@Entity
public class Address {
      @Id @GeneratedValue
      private Long id;
      private String street1;
      private String street2;
      private String city;
      private String state;
      private String zipcode;
      private String country;
      // Constructors, getters, setters
}
```

Relationship Mapping VI

```
create table CUSTOMER (
  ID BIGINT not null,
  FIRSTNAME VARCHAR(255),
  LASTNAME VARCHAR(255),
  EMAIL VARCHAR(255),
  PHONENUMBER VARCHAR(255),
  ADDRESS_ID BIGINT,
  primary key (ID),
  foreign key (ADDRESS_ID) references ADDRESS(ID)
);
```

```
create table ADDRESS (
  ID BIGINT not null,
  STREET1 VARCHAR(255),
  STREET2 VARCHAR(255),
  CITY VARCHAR(255),
  STATE VARCHAR(255),
  ZIPCODE VARCHAR(255),
  COUNTRY VARCHAR(255),
  primary key (ID)
);
```

Relationship Mapping VII

```
@Target({
      METHOD, FIELD})
@Retention(RUNTIME)
public @interface OneToOne {
      Class targetEntity() default void.class;
      CascadeType[] cascade() default {
      };
      FetchType fetch() default EAGER;
      boolean optional() default true;
      String mappedBy() default "";
      boolean orphanRemoval() default false;
}
```

```
@Entity
public class Customer {
      @Id @GeneratedValue
      private Long id;
      private String firstName;
      private String lastName;
      private String email;
      private String phoneNumber;
      @OneToOne (fetch = FetchType.LAZY)
      @JoinColumn(name = "add_fk", nullable = false)
      private Address address;
      // Constructors, getters, setters
}
```

Relationship Mapping VIII

In JPA, a foreign key column is called a **join column**. The `@JoinColumn` annotation allows you to customize the mapping of a foreign key.

Relationship Mapping IX

`@OneToMany` Unidirectional

```
@Entity
public class Order {
      @Id @GeneratedValue
      private Long id;
      @Temporal(TemporalType.TIMESTAMP)
      private Date creationDate;
      private List<OrderLine> orderLines;
      // Constructors, getters, setters
}
```

```
@Entity
@Table(name = "order_line")
public class OrderLine {
      @Id @GeneratedValue
      private Long id;
      private String item;
      private Double unitPrice;
      private Integer quantity;
      // Constructors, getters, setters
}
```

Notes

- The above Order doesn't have any special annotation and relies on the configuration-by-exception paradigm.

Relationship Mapping X

- By default, one-to-many unidirectional relationships use a **join table** to keep the relationship information, with two foreign key columns.
- One foreign key column refers to the table ORDER and has the same type as its primary key, and the other refers to ORDER_LINE.
- The name of this joined table is the name of both entities, separated by the _ symbol. The join table is named ORDER_ORDER_LINE.

```
@Target({
      METHOD, FIELD})
@Retention(RUNTIME)
public @interface JoinTable {
      String name() default "";
      String catalog() default "";
      String schema() default "";
      JoinColumn[] joinColumns() default {
      };
      JoinColumn[] inverseJoinColumns() default {
      };
      UniqueConstraint[] uniqueConstraints() default {
      };
      Index[] indexes() default {
      };
}
```

Relationship Mapping XI

```
@Entity
public class Order {
      @Id @GeneratedValue
      private Long id;
      @Temporal(TemporalType.TIMESTAMP)
      private Date creationDate;
      @OneToMany
      @JoinTable(name = "jnd_ord_line",
      joinColumns = @JoinColumn(name = "order_fk"),
      inverseJoinColumns = @JoinColumn(name = "order_line_fk") )
      private List<OrderLine> orderLines;
      // Constructors, getters, setters
}
```

```
create table JND_ORD_LINE (
ORDER_FK BIGINT not null,
ORDER_LINE_FK BIGINT not null,
primary key (ORDER_FK, ORDER_LINE_FK),
foreign key (ORDER_LINE_FK) references ORDER_LINE(ID),
foreign key (ORDER_FK) references ORDER(ID)
);
```

Relationship Mapping XII

The Order Entity with a Join Column

```
@Entity
public class Order {
      @Id @GeneratedValue
      private Long id;
      @Temporal(TemporalType.TIMESTAMP)
      private Date creationDate;
      @OneToMany(fetch = FetchType.EAGER)
      @JoinColumn(name = "order_fk")
      private List<OrderLine> orderLines;
      // Constructors, getters, setters
}
```

Relationship Mapping XIII

@ManyToMany Bidirectional

- In a bidirectional relationship you need to explicitly define the owner (with the **mappedBy** element)

```
@Entity
public class CD {
      @Id @GeneratedValue
      private Long id;
      private String title;
      private Float price;
      private String description;
      @ManyToMany(mappedBy = "appearsOnCDs")
      private List<Artist> createdByArtists;
      // Constructors, getters, setters
}
```

Relationship Mapping XIV

```
@Entity
public class Artist {
      @Id @GeneratedValue
      private Long id;
      private String firstName;
      private String lastName;
      @ManyToMany
      @JoinTable(name = "jnd_art_cd",
      joinColumns = @JoinColumn(name = "artist_fk"),
      inverseJoinColumns = @JoinColumn(name = "cd_fk"))
      private List<CD>appearsOnCDs;
      // Constructors, getters, setters
}
```

Note

- A many-to-many and one-to-one bidirectional relationship, either side may be designated as the owning side.
- No matter which side is designated as the owner, the other side should include the **mappedBy** element.

Fetching Relationships I

Table: Default Fetching Strategies

Annotation	Default Fetching Strategy
@OneToOne	EAGER
@ManyToOne	EAGER
@OneToMany	LAZY
@ManyToMany	LAZY

```
@Entity
public class Order {
      @Id @GeneratedValue
      private Long id;
      @Temporal(TemporalType.TIMESTAMP)
      private Date creationDate;
      @OneToMany(fetch = FetchType.EAGER)
      private List<OrderLine> orderLines;
      // Constructors, getters, setters
}
```

Ordering Relationships I

@OrderBy

```
@Entity
public class Comment {
      @Id @GeneratedValue
      private Long id;
      private String nickname;
      private String content;
      private Integer note;
      @Column(name = "posted_date")
      @Temporal(TemporalType.TIMESTAMP)
      private Date postedDate;
      // Constructors, getters, setters
}
```

```
@Entity
public class News {
      @Id @GeneratedValue
      private Long id;
      @Column(nullable = false)
      private String content;
      @OneToMany(fetch = FetchType.EAGER)
      @OrderBy("postedDate DESC")
      private List<Comment> comments;
      // Constructors, getters, setters
}
```

Ordering Relationships II

@OrderColumn

- This annotation informs the persistence provider that it is required to maintain the ordered list using a separate column where the index is stored. The `@OrderColumn` defines this separate column.

```
@Entity
public class Comment {
      @Id @GeneratedValue
      private Long id;
      private String nickname;
      private String content;
      private Integer note;
      // Constructors, getters, setters
}
```

Ordering Relationships III

```
@Entity
public class News {
      @Id @GeneratedValue
      private Long id;
      @Column(nullable = false)
      private String content;
      @OneToMany(fetch = FetchType.EAGER)
      @OrderColumn(name = "posted_index")
      private List<Comment> comments;
      // Constructors, getters, setters
}
```

```
create table NEWS_COMMENT (
NEWS_ID BIGINT not null,
COMMENTS_ID BIGINT not null,
POSTED_INDEX INTEGER
);
```

Inheritance Mapping I

Inheritance is a completely **unknown** concept and not natively implemented in a relational world. How do you organize a hierarchical model into a flat relational one? JPA has three different strategies you can choose from.

single-table-per-class-hierarchy The sum of the attributes of the entire entity hierarchy is flattened down to a single table (this is the default strategy).

joined-subclass In this approach, each entity in the hierarchy, concrete or abstract, is mapped to **its own dedicated** table.

table-per-concrete-class This strategy maps each concrete entity hierarchy to its own separate table.

Inheritance Mapping II

Single-Table-per-Class Hierarchy Strategy

```
@Entity
public class Item {
      @Id @GeneratedValue
      protected Long id;
      protected String title;
      protected Float price;
      protected String description;
      // Constructors, getters, setters
}
```

```
@Entity
public class Book extends Item {
      private String isbn;
      private String publisher;
      private Integer nbOfPage;
      private Boolean illustrations;
      // Constructors, getters, setters
}
```

Inheritance Mapping III

```
@Entity
public class CD extends Item {
      private String musicCompany;
      private Integer numberOfCDs;
      private Float totalDuration;
      private String genre;
      // Constructors, getters, setters
}
```

Note

- With the single-table-per-class strategy (the default one), they all end up in the same database table, which defaults to the name of the root class: ITEM.
- An additional column that doesn't relate to any of the entities' attributes: it's the discriminator column, **DTYPE**.

Inheritance Mapping IV

ITEM		
+ID	**bigint**	**Nullable = false**
DTYPE	varchar(31)	Nullable = true
TITLE	varchar(255)	Nullable = false
PRICE	double	Nullable = false
DESCRIPTION	varchar(255)	Nullable = true
ILLUSTRATIONS	smallint	Nullable = true
ISBN	varchar(255)	Nullable = true
NBOFPAGE	integer	Nullable = true
PUBLISHER	varchar(255)	Nullable = true
MUSICCOMPANY	varchar(255)	Nullable = true
NUMBEROFCDS	integer	Nullable = true
TOTALDURATION	double	Nullable = true
GENRE	varchar(255)	Nullable = true

Inheritance Mapping V

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)
@DiscriminatorColumn (name="disc",
discriminatorType = DiscriminatorType.CHAR)
@DiscriminatorValue("I")
public class Item {
      @Id @GeneratedValue
      protected Long id;
      protected String title;
      protected Float price;
      protected String description;
      // Constructors, getters, setters
}
```

```
@Entity
@DiscriminatorValue("B")
public class Book extends Item {
      private String isbn;
      private String publisher;
      private Integer nbOfPage;
      private Boolean illustrations;
      // Constructors, getters, setters
}
```

Inheritance Mapping VI

```
@Entity
@DiscriminatorValue("C")
public class CD extends Item {
      private String musicCompany;
      private Integer numberOfCDs;
      private Float totalDuration;
      private String genre;
      // Constructors, getters, setters
}
```

Inheritance Mapping VII

Joined-Subclass Strategy

- In the joined-subclass strategy, each entity in the hierarchy is mapped to its **own** table.
- The root entity maps to a table that defines the primary key to be used by **all** tables in the hierarchy, as well as the **discriminator column**.
- Each subclass is represented by a separate table that contains its own attributes (not inherited from the root class) and a primary key that refers to the root table's primary key.
- The non-root tables do not hold a discriminator column.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Item {
      @Id @GeneratedValue
      protected Long id;
      protected String title;
      protected Float price;
      protected String description;
      // Constructors, getters, setters
}
```

Inheritance Mapping VIII

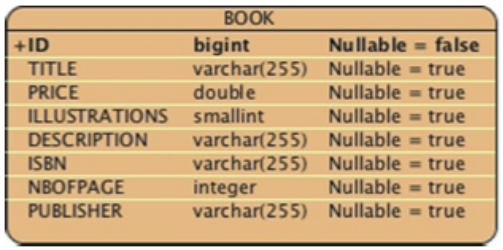
Note

- You can still use @DiscriminatorColumn and @DiscriminatorValue annotations in the root entity to customize the discriminator column and values (the DTYPE column is in the ITEM table).
- The joined-subclass strategy is intuitive and is close to what you know from the object inheritance mechanism.
- But querying can have a performance impact. This strategy is called joined because, to reassemble an instance of a subclass, the subclass table has to be joined with the root class table.
- The deeper the hierarchy, the more joins needed to assemble a leaf entity.
- This strategy provides good support for polymorphic relationships but requires one or more join operations to be performed when instantiating entity subclasses.
- This may result in poor performance for extensive class hierarchies. Similarly, queries that cover the entire class hierarchy require join operations between the subclass tables, resulting in decreased performance.

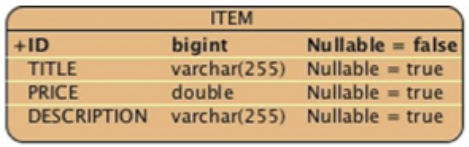
Table-per-Concrete-Class Strategy I

In the table-per-class (or table-per-concrete-class) strategy, each entity is mapped to its own dedicated table like the joined-subclass strategy. The difference is that all attributes of the root entity will also be mapped to columns of the child entity table.

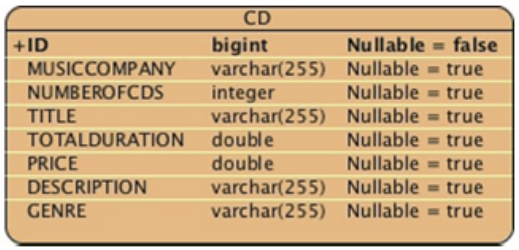
```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Item {
      @Id @GeneratedValue
      protected Long id;
      protected String title;
      protected Float price;
      protected String description;
      // Constructors, getters, setters
}
```



BOOK		
+ID	bigint	Nullable = false
TITLE	varchar(255)	Nullable = true
PRICE	double	Nullable = true
ILLUSTRATIONS	smallint	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true
ISBN	varchar(255)	Nullable = true
NBOFPAGE	integer	Nullable = true
PUBLISHER	varchar(255)	Nullable = true



ITEM		
+ID	bigint	Nullable = false
TITLE	varchar(255)	Nullable = true
PRICE	double	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true



CD		
+ID	bigint	Nullable = false
MUSICCOMPANY	varchar(255)	Nullable = true
NUMBEROFCDS	integer	Nullable = true
TITLE	varchar(255)	Nullable = true
TOTALDURATION	double	Nullable = true
PRICE	double	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true
GENRE	varchar(255)	Nullable = true

Overriding Attributes I

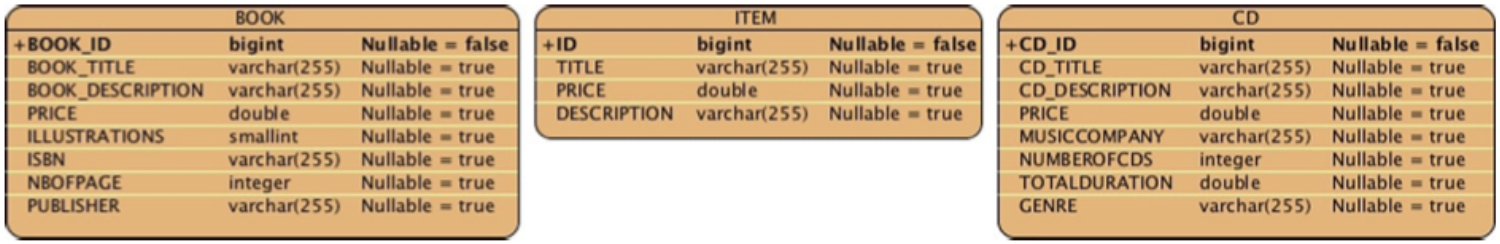
With the table-per-concrete-class strategy, the columns of the root class are duplicated on the leaf tables. They keep the same name.

- But what if a legacy database is being used and the columns have a different name?
- JPA uses the @AttributeOverride annotation to override the column mapping and @AttributeOverrides to override several.

```
@Entity
@AttributeOverrides({
      @AttributeOverride(name = "id",
      column = @Column(name = "book_id")),
      @AttributeOverride(name = "title",
      column = @Column(name = "book_title")),
      @AttributeOverride(name = "description",
      column = @Column(name = "book_description"))
})
public class Book extends Item {
      private String isbn;
      private String publisher;
      private Integer nbOfPage;
      private Boolean illustrations;
      // Constructors, getters, setters
}
```

Overriding Attributes II

```
@Entity
@AttributeOverrides({
      @AttributeOverride(name = "id",
      column = @Column(name = "cd_id")),
      @AttributeOverride(name = "title",
      column = @Column(name = "cd_title")),
      @AttributeOverride(name = "description",
      column = @Column(name = "cd_description"))
})
public class CD extends Item {
      private String musicCompany;
      private Integer numberOfCDs;
      private Float totalDuration;
      private String genre;
      // Constructors, getters, setters
}
```



BOOK		
+BOOK_ID	bigint	Nullable = false
BOOK_TITLE	varchar(255)	Nullable = true
BOOK_DESCRIPTION	varchar(255)	Nullable = true
PRICE	double	Nullable = true
ILLUSTRATIONS	smallint	Nullable = true
ISBN	varchar(255)	Nullable = true
NBOFPAGE	integer	Nullable = true
PUBLISHER	varchar(255)	Nullable = true

ITEM		
+ID	bigint	Nullable = false
TITLE	varchar(255)	Nullable = true
PRICE	double	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true

CD		
+CD_ID	bigint	Nullable = false
CD_TITLE	varchar(255)	Nullable = true
CD_DESCRIPTION	varchar(255)	Nullable = true
PRICE	double	Nullable = true
MUSICCOMPANY	varchar(255)	Nullable = true
NUMBEROFCDS	integer	Nullable = true
TOTALDURATION	double	Nullable = true
GENRE	varchar(255)	Nullable = true

Type of Classes in the Inheritance Hierarchy I

```
public class Item {
      protected String title;
      protected Float price;
      protected String description;
      // Constructors, getters, setters
}
```

```
@Entity
public class Book extends Item {
      @Id @GeneratedValue
      private Long id;
      private String isbn;
      private String publisher;
      private Integer nbOfPage;
      private Boolean illustrations;
      // Constructors, getters, setters
}
```

```
create table BOOK (
ID BIGINT not null,
ILLUSTRATIONS SMALLINT,
ISBN VARCHAR(255),
NBOFPAGE INTEGER,
PUBLISHER VARCHAR(255),
primary key (ID)
);
```

Type of Classes in the Inheritance Hierarchy II

Mapped superclass

- JPA defines a special kind of class, called a mapped superclass, to share state and behavior, as well as mapping information entities inherit from.
- Mapped superclasses are not entities.

```
@MappedSuperclass
@Inheritance(strategy = InheritanceType.JOINED)
public class Item {
      @Id @GeneratedValue
      protected Long id;
      @Column(length = 50, nullable = false)
      protected String title;
      protected Float price;
      @Column(length = 2000)
      protected String description;
      // Constructors, getters, setters
}
```

Type of Classes in the Inheritance Hierarchy III

```
@Entity
public class Book extends Item {
      private String isbn;
      private String publisher;
      private Integer nbOfPage;
      private Boolean illustrations;
      // Constructors, getters, setters
}
```

```
create table BOOK (
ID BIGINT not null,
TITLE VARCHAR(50) not null,
PRICE DOUBLE(52, 0),
DESCRIPTION VARCHAR(2000),
ILLUSTRATIONS SMALLINT,
ISBN VARCHAR(255),
NBOFPAGE INTEGER,
PUBLISHER VARCHAR(255),
primary key (ID)
);
```

Summary I

- Thanks to configuration by exception, not much is required to map entities to tables; inform the persistence provider that a class is actually an entity (using `@Entity`) and an attribute is its identifier (using `@Id`), and JPA does the rest.
- JPA has a very rich set of annotations to customize every little detail of ORM (as well as the equivalent XML mapping).

Managing Persistent Objects I

- In JPA, the centralized service to manipulate instances of entities is the **entity manager**.
- It provides an API to create, find, remove, and synchronize objects with the database.
- It also allows the execution of different sorts of **JPQL queries**, such as dynamic, static, or native queries, against entities.
- **Locking** mechanisms are also possible with the entity manager.
- **JPQL** is the language defined in JPA to query entities stored in a relational database.

Entity Manager I

- When an entity is managed, you can carry out persistence operations, and the entity manager will automatically synchronize the state of the entity with the database.
- When the entity is detached (i.e., not managed), it returns to a simple POJO.
- `EntityManager` is an interface implemented by a persistence provider that will generate and execute SQL statements.

```
public interface EntityManager {
      // Factory to create an entity manager, close it and check if it's open
      EntityManagerFactory getEntityManagerFactory();
      void close();
      boolean isOpen();
      // Returns an entity transaction
      EntityTransaction getTransaction();
      // Persists, merges and removes and entity to/from the database
      void persist(Object entity);
      <T> T merge(T entity);
      void remove(Object entity);
      // Finds an entity based on its primary key (with different lock mecanisms)
      <T> T find(Class<T> entityClass, Object primaryKey);
      <T> T find(Class<T> entityClass, Object primaryKey, LockModeType lockMode);
      <T> T getReference(Class<T> entityClass, Object primaryKey);
      // Locks an entity with the specified lock mode type (optimistic, pessimistic)
      void lock(Object entity, LockModeType lockMode);
      // Synchronizes the persistence context to the underlying database
      void flush();
```

Entity Manager II

```
void setFlushMode(FlushModeType flushMode);
FlushModeType getFlushMode();
// Refreshes the state of the entity from the database, overwriting any changes made
void refresh(Object entity);
void refresh(Object entity, LockModeType lockMode);
// Clears the persistence context and checks if it contains an entity
void clear();
void detach(Object entity);
boolean contains(Object entity);
// Sets and gets an entity manager property or hint
void setProperty(String propertyName, Object value);
Map<String, Object> getProperties();
// Creates an instance of Query or TypedQuery for executing a JPQL statement
Query createQuery(String qlString);
<T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery);
<T> TypedQuery<T> createQuery(String qlString, Class<T> resultClass);
// Creates an instance of Query or TypedQuery for executing a named query
Query createNamedQuery(String name);
<T> TypedQuery<T> createNamedQuery(String name, Class<T> resultClass);
// Creates an instance of Query for executing a native SQL query
Query createNativeQuery(String sqlString);
Query createNativeQuery(String sqlString, Class resultClass);
Query createNativeQuery(String sqlString, String resultSetMapping);
// Creates a StoredProcedureQuery for executing a stored procedure in the database
StoredProcedureQuery createStoredProcedureQuery(String procedureName);
StoredProcedureQuery createNamedStoredProcedureQuery(String name);
// Metamodel and criteria builder for criteria queries (select, update and delete)
CriteriaBuilder getCriteriaBuilder();
Metamodel getMetamodel();
```

Entity Manager III

```
    Query createQuery(CriteriaUpdate updateQuery);
    Query createQuery(CriteriaDelete deleteQuery);
    // Indicates that a JTA transaction is active and joins the persistence context to it
    void joinTransaction();
    boolean isJoinedToTransaction();
    // Return the underlying provider object for the EntityManager
    <T> T unwrap(Class<T> cls);
    Object getDelegate();
    // Returns an entity graph
    <T> EntityGraph<T> createEntityGraph(Class<T> rootType);
    EntityGraph<?> createEntityGraph(String graphName);
    <T> EntityGraph<T> getEntityGraph(String graphName);
    <T> List<EntityGraph<? super T>> getEntityGraphs(Class<T> entityClass);
}
```

Entity Manager IV

Obtaining an Entity Manager

```
public class Main {
      public static void main(String[] args) {
            // Creates an instance of book
            Book book = new Book("H2G2",
            "The Hitchhiker's Guide to the Galaxy", 12.5F,
            "1-84023-742-2", 354, false);
            // Obtains an entity manager and a transaction
            EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter06PU");
            EntityManager em = emf.createEntityManager();
            // Persists the book to the database
            EntityTransaction tx = em.getTransaction(); tx.begin();
            em.persist(book);
            tx.commit();
            // Closes the entity manager and the factory
            em.close();
            emf.close();
      }
}
```

Application managed manager

Note

- reating an application-managed entity manager is simple enough using a factory, but what differentiates application managed from container managed is how the factory is acquired.

Entity Manager V

- A container-managed environment is when the application evolves in a Servlet or an EJB container.
- In a Java EE environment, the most common way to acquire an entity manager is by the @PersistenceContext annotation, or by JNDI lookup.
- The component running in a container (Servlet, EJB, web service, etc.) doesn't need to create or close the entity manager, as its life cycle is managed by the container.

```
@Stateless
public class BookEJB {
      @PersistenceContext(unitName = "chapter06PU")
      private EntityManager em;
      public void createBook() {
            // Creates an instance of book
            Book book = new Book("H2G2", "The Hitchhiker's Guide to the Galaxy", 12.5F,
            "1-84023-742-2", 354, false);
            // Persists the book to the database
            em.persist(book);
      }
}
```

Container managed manager

Entity Manager VI

Persistence Context

- It is a set of managed entity instances at a given time for a given user's transaction.
- Only one entity instance with the same persistent identity can exist in a persistence context.
- The persistence context can be seen as a **first-level cache**.
- It's a short, live space where the entity manager stores entities before flushing the content to the database.
- By default, objects just live in the persistent context for the duration of the transaction.

Entity Manager VII

The configuration for an entity manager is bound to the factory that created it.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd" version="2.1">
  <persistence-unit name="chapter06PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>org.agoncal.book.javaee7.chapter06.Book</class>
    <class>org.agoncal.book.javaee7.chapter06.Customer</class>
    <class>org.agoncal.book.javaee7.chapter06.Address</class>
    <properties>
      <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
      <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:derby:memory:chapter06DB;create=true"/>
      <property name="eclipselink.logging.level" value="INFO"/>
    </properties>
  </persistence-unit>
</persistence>
```

The persistence unit is the **bridge** between the persistence context and the database.

Entity Manager VIII

Table: Basic operations of EntityManager

Method
void persist(Object entity)
<T> T find(Class<T> entityClass, Object primaryKey)
<T> T getReference(Class<T> entityClass, Object primaryKey)
void remove(Object entity)
<T> T merge(T entity)
void refresh(Object entity)
void flush()
void clear()
void detach(Object entity)
boolean contains(Object entity)

Entity Manager IX

Persisting an Entity

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
assertNotNull(customer.getId());
assertNotNull(address.getId());
```

Note

- If you need to pass entities by value (remote invocation, external EJB container, etc.), they must implement the java.io.Serializable marker (no method) interface.

Entity Manager X

Finding by ID

```
Customer customer = em.find(Customer.class, 1234L) ;
if (customer!= null) {
      // Process the object
}
```

getReference()

- It is very similar to the find operation, as it has the same parameters, but it retrieves a reference to an entity (via its primary key) but does not retrieve its data.
- Think of it as a **proxy** to an entity, **not** the entity **itself**.
- It is intended for situations where a managed entity instance is needed, but no data, other than potentially the **entity's primary key**, being accessed.
- With getReference(), the state data are fetched lazily, which means that if you don't access state before the entity is detached, the data might not be there.
- If the entity is not found, an `EntityNotFoundException` is thrown.

```
try {
      Customer customer = em.getReference(Customer.class, 1234L);
      // Process the object
} catch(EntityNotFoundException ex) {
      // Entity not found
}
```

Entity Manager XI

Removing an Entity

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
tx.begin();
em.remove(customer);
tx.commit();
// The data is removed from the database but the object is still accessible
assertNotNull(customer);
```

Entity Manager XII

Orphan Removal

- For data consistency, orphans are not desirable, as they result in having rows in a database that are not referenced by any other table, without means of access.
- With JPA, you can inform the persistence provider to automatically remove orphans or cascade a remove operation

```
@Entity
public class Customer {
      @Id @GeneratedValue
      private Long id;
      private String firstName;
      private String lastName;
      private String email;
      @OneToOne (fetch = FetchType.LAZY, orphanRemoval=true)
      private Address address;
      // Constructors, getters, setters
}
```

Entity Manager XIII

Synchronizing with the Database

- Flushing an Entity
- Refreshing an Entity

Entity Manager XIV

Contains

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
tx.begin();
em.persist(customer);
tx.commit();
assertTrue(em.contains(customer));
tx.begin();
em.remove(customer);
tx.commit();
assertFalse(em.contains(customer));
```

Entity Manager XV

Clear and Detach

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
tx.begin();
em.persist(customer);
tx.commit();
assertTrue(em.contains(customer));
em.detach(customer);
assertFalse(em.contains(customer));
```

Entity Manager XVI

Merging an Entity

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
tx.begin();
em.persist(customer);
tx.commit();
em.clear();
// Sets a new value to a detached entity
customer.setFirstName("William");
tx.begin();
em.merge(customer);
tx.commit();
```

Entity Manager XVII

Updating an Entity

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
tx.begin();
em.persist(customer);
customer.setFirstName("Williman");
tx.commit();
```

Note

- If an entity is currently managed, changes to it will be reflected in the database automatically.

Entity Manager XVIII

Cascading Events

```
@Entity
public class Customer {
      @Id @GeneratedValue
      private Long id;
      private String firstName;
      private String lastName;
      private String email;
      @OneToOne (fetch = FetchType.LAZY, cascade = {
            CascadeType.PERSIST, CascadeType.REMOVE})
      @JoinColumn(name = "address_fk")
      private Address address;
      // Constructors, getters, setters
}
```

JPQL I

- JPQL is used to define searches against persistent entities independent of the underlying database.
- JPQL is a query language that takes its roots in the syntax of SQL, which is the standard language for database interrogation.
- Under the hood, JPQL uses the mechanism of mapping to transform a JPQL query into language comprehensible by an SQL database.

```
SELECT <select clause>
FROM <from clause>
[WHERE <where clause>]
[ORDER BY <order by clause>]
[GROUP BY <group by clause>]
[HAVING <having clause>]
```

JPQL II

```
SELECT c
FROM Customer c

SELECT c.firstName
FROM Customer c

SELECT c.firstName, c.lastName
FROM Customer c

SELECT CASE b.editor WHEN 'Apress'
            THEN b.price * 0.5
            ELSE b.price * 0.8
       END
FROM Book b


SELECT c.address
FROM Customer c

SELECT c.address.country.code
FROM Customer c


SELECT NEW CustomerDTO(c.firstName, c.lastName, c.address.street1)
FROM Customer c

SELECT DISTINCT c FROM Customer c
```

JPQL III

```
SELECT DISTINCT c.firstName FROM Customer c
```

JPQL IV

Binding Parameters

```
SELECT c
FROM Customer c
WHERE c.firstName = ?1 AND c.address.country = ?2

SELECT c
FROM Customer c
WHERE c.firstName = :fname AND c.address.country = :country
```

```
SELECT c
FROM Customer c
WHERE c.age = (SELECT MIN(cust. age) FROM Customer cust))
```

JPQL V

Order By

```
SELECT c
FROM Customer c
WHERE c.age > 18 ORDER BY c.age DESC

SELECT c
FROM Customer c
WHERE c.age > 18
ORDER BY c.age DESC, c.address.country ASC
```

Group By and Having

```
SELECT c.address.country, count(c) FROM Customer c
GROUP BY c.address.country

SELECT c.address.country, count(c) FROM Customer c
GROUP BY c.address.country
HAVING c.address.country <> 'UK'
```

JPQL VI

Bulk Delete

```
DELETE FROM Customer c
WHERE c.age < 18
```

Bulk Update

```
UPDATE Customer c
SET c.firstName = 'TOO YOUNG' WHERE c.age < 18
```

Queries I

Dynamic queries This is the simplest form of query, consisting of nothing more than a JPQL query string dynamically specified at runtime.

Named queries Named queries are static and unchangeable.

Criteria API JPA 2.0 introduced the concept of object-oriented query API.

Native queries This type of query is useful to execute a native SQL statement instead of a JPQL statement.

Stored procedure queries JPA 2.1 brings a new API to call stored procedures.

Queries II

Dynamic Queries

- Dynamic queries are defined on the fly as needed by the application.

```
Query query = em.createQuery("SELECT c FROM Customer c", Customer.class);
query.setMaxResults(10);
List<Customer> customers = query.getResultList();
```

Note

- An issue to consider with dynamic queries is the cost of translating the JPQL string into an SQL statement at runtime.
- Because the query is dynamically created and cannot be predicted, the persistence provider has to parse the JPQL string, get the ORM metadata, and generate the equivalent SQL.
- The performance cost of processing each of these dynamic queries can be an issue.
- If you have static queries that are unchangeable and want to avoid this overhead, then you can use named queries instead.

Queries III

Named Queries

- Named queries are different from dynamic queries in that they are static and unchangeable.
- In addition to their static nature, which does not allow the flexibility of a dynamic query, named queries can be more efficient to execute because the persistence provider can translate the JPQL string to SQL once the application starts, rather than every time the query is executed.

```
@Entity
@NamedQuery(name = "findAll", query="select c from Customer c")
public class Customer {
}
```

```
Query query = em.createNamedQuery("findAll");

TypedQuery<Customer> query = em.createNamedQuery("findAll", Customer.class);

Query query = em.createNamedQuery("findWithParam");
query.setParameter("fname", "Vincent");
query.setMaxResults(3);
List<Customer> customers = query.getResultList();
```

Note

- There is a restriction in that the name of the query is scoped to the persistence unit and must be unique within that scope, meaning that only one findAll method can exist.

Queries IV

```
@Entity
@NamedQuery(name = Customer.FIND_ALL, query="select c from Customer c"),
public class Customer {
      public static final String FIND_ALL = "Customer.findAll";
      // Attributes, constructors, getters, setters
}

TypedQuery<Customer> query = em.createNamedQuery(Customer.FIND_ALL, Customer.class);
```

Queries V

Criteria API (or Object-Oriented Queries)

- JPA 2.0 created a new API, called Criteria API. It allows you to write any query in an object-oriented and syntactically correct way.
- Most of the mistakes that a developer could make writing a statement are found at compile time, not at runtime.
- The idea is that all the JPQL keywords (SELECT, UPDATE, DELETE, WHERE, LIKE, GROUP BY) are defined in this API.
- The Criteria API supports everything JPQL can do but with an object-based syntax.

```
SELECT c FROM Customer c WHERE c.firstName = 'Vincent'
```

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.equal(c.get("firstName"), "Vincent"));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();
```

Queries VI

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.greaterThan(c.get("age").as(Integer.class), 40));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();
```

Note

- Criteria API allows you to write error-free statements.
- But it's not completely true yet. You can still see some strings ("firstName" and "age") that represent the attributes of the Customer entity. So typos can still be made.

Queries VII

Type-Safe Criteria API

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.greaterThan(c.get(Customer_.age), 40));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();
```

To bring type safety, JPA 2.1 can generate a static metamodel class for each entity.

```
@Generated("EclipseLink")
@StaticMetamodel(Customer.class)
public class Customer_{
      public static volatile SingularAttribute<Customer, Long> id;
      public static volatile SingularAttribute<Customer, String> firstName;
      public static volatile SingularAttribute<Customer, String> lastName;
      public static volatile SingularAttribute<Customer, Integer> age;
      public static volatile SingularAttribute<Customer, String> email;
      public static volatile SingularAttribute<Customer, Address> address;
}
```

Queries VIII

Native Queries

```
Query query = em.createNativeQuery("SELECT * FROM t_customer", Customer.class);
List<Customer> customers = query.getResultList();
```

Note

- The main reason to use JPA native queries rather than JDBC calls is because the result of the query will be automatically converted back to entities.

Queries IX

Stored Procedure Queries

```
StoredProcedureQuery query = em.createStoredProcedureQuery("sp_archive_old_books");
query.registerStoredProcedureParameter("archiveDate", Date.class, ParameterMode.IN);
query.registerStoredProcedureParameter("maxBookArchived", Integer.class, ParameterMode.IN);
query.setParameter("archiveDate", new Date());
query.setParameter("maxBookArchived", 1000);
query.execute();
```

Cache API I

- The entity manager is a **first-level cache** used to process data comprehensively for the database and to cache short-lived entities.
- The second-level cache **sits between the entity manager and the database** to reduce database traffic by keeping objects loaded in memory and available to the whole application.

```
public interface Cache {
      // Whether the cache contains the given entity
      public boolean contains(Class cls, Object id);
      // Removes the given entity from the cache
      public void evict(Class cls, Object id);
      // Removes entities of the specified class (and its subclasses) from the cache
      public void evict(Class cls);
      // Clears the cache.
      public void evictAll();
      // Returns the provider-specific cache implementation
      public <T> T unwrap(Class<T> cls);
}
```

Cache API II

```
@Entity
@Cacheable(true)
public class Customer {
      @Id @GeneratedValue
      private Long id;
      private String firstName;
      private String lastName;
      private String email;
      // Constructors, getters, setters
}
```

```
Customer customer = new Customer("Patricia", "Jane", "plecomte@mail.com");
tx.begin();
em.persist(customer);
tx.commit();
/ Uses the EntityManagerFactory to get the Cache
Cache cache = emf.getCache();
// Customer should be in the cache
assertTrue(cache.contains(Customer.class, customer.getId()));
// Removes the Customer entity from the cache
cache.evict(Customer.class);
// Customer should not be in the cache anymore
assertFalse(cache.contains(Customer.class, customer.getId()));
```

Concurrency I

JPA 2.1 uses two different locking mechanisms

- **Optimistic locking** is based on the assumption that most database transactions don't conflict with other transactions.
- Pessimistic locking is based on the opposite assumption, so a lock will be obtained on the resource before operating on it.

```
Book book = em.find(Book.class, 12);
// Lock to raise the price
em.lock(book, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
book.raisePriceByTwoDollars();
```

```
Book book = em.find(Book.class, 12, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
// The book is already locked, raise the price
book.raisePriceByTwoDollars();
```

Versioning I

```
@Entity
public class Book {
      @Id @GeneratedValue
      private Long id;
      @Version
      private Integer version;
      private String title;
      private Float price;
      private String description;
      private String isbn;
      private Integer nbOfPage;
      private Boolean illustrations;
      // Constructors, getters, setters
}
```

Note

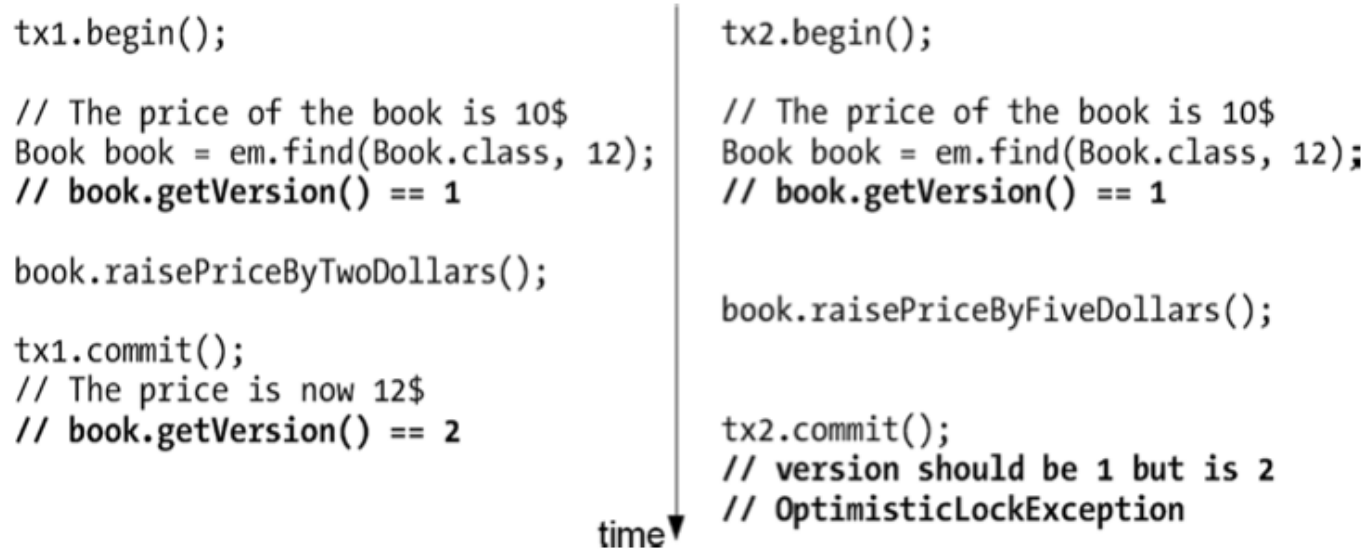
- The entity can access the value of its version property but must not modify it.
- Only the persistence provider is permitted to set or update the value of the version attribute when the object is written or updated to the database.

Versioning II

```
Book book = new Book("H2G2", 21f, "Best IT book", "123-456", 321, false);
tx.begin();
em.persist(book);
tx.commit();
assertEquals(1, book.getVersion());
tx.begin();
book.raisePriceByTwoDollars();
tx.commit();
assertEquals(2, book.getVersion());
```

Versioning III

Optimistic Locking




```
Book book = em.find(Book.class, 12);
// Lock to raise the price
em.lock(book, LockModeType.OPTIMISTIC);
book.raisePriceByTwoDollars();
```

How would you throw an OptimisticLockException?

- Either by explicitly locking the entity (with the lock or the find methods that you saw, passing a LockModeType) or by letting the persistence provider check the attribute annotated with `@Version`.

Versioning IV

- The use of a dedicated `@Version` annotation on an entity allows the EntityManager to perform optimistic locking simply by comparing the value of the version attribute in the entity instance with the value of the column in the database.
- Without an attribute annotated with `@Version`, the entity manager will not be able to do optimistic locking automatically (implicitly).

Note

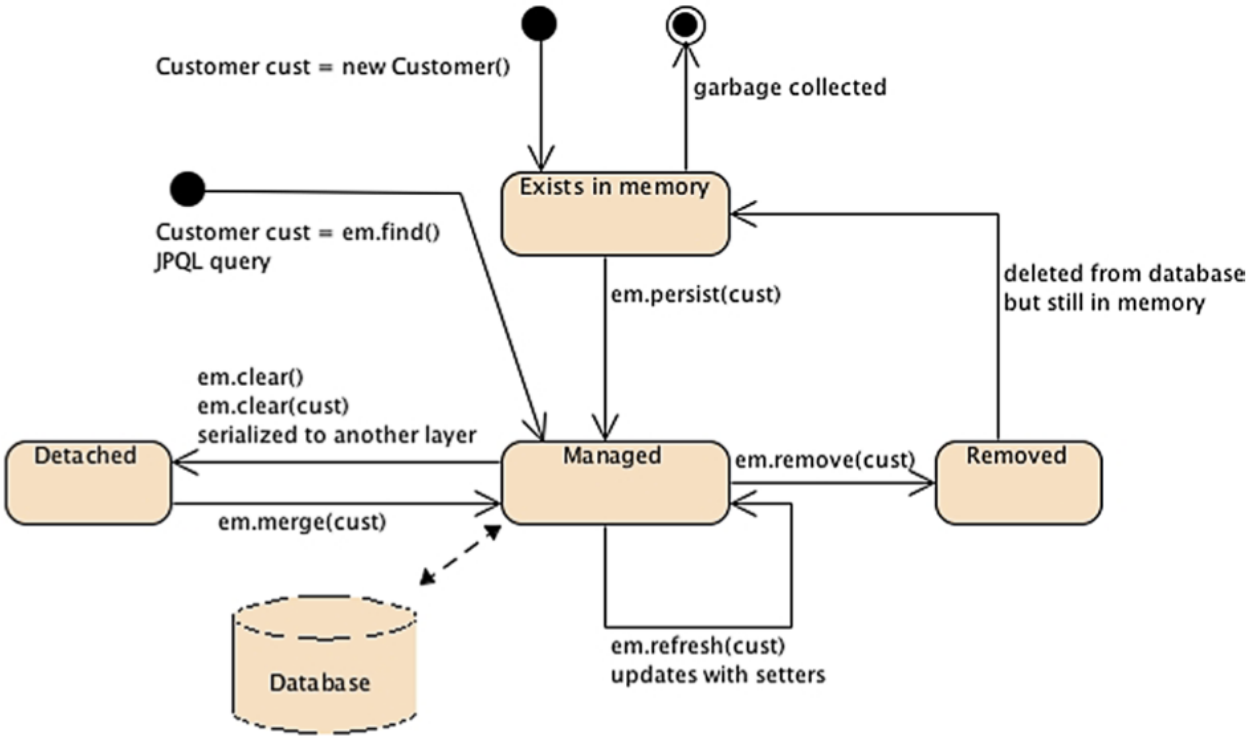
- Applications are strongly encouraged to enable optimistic locking for all entities that may be concurrently accessed

Versioning V

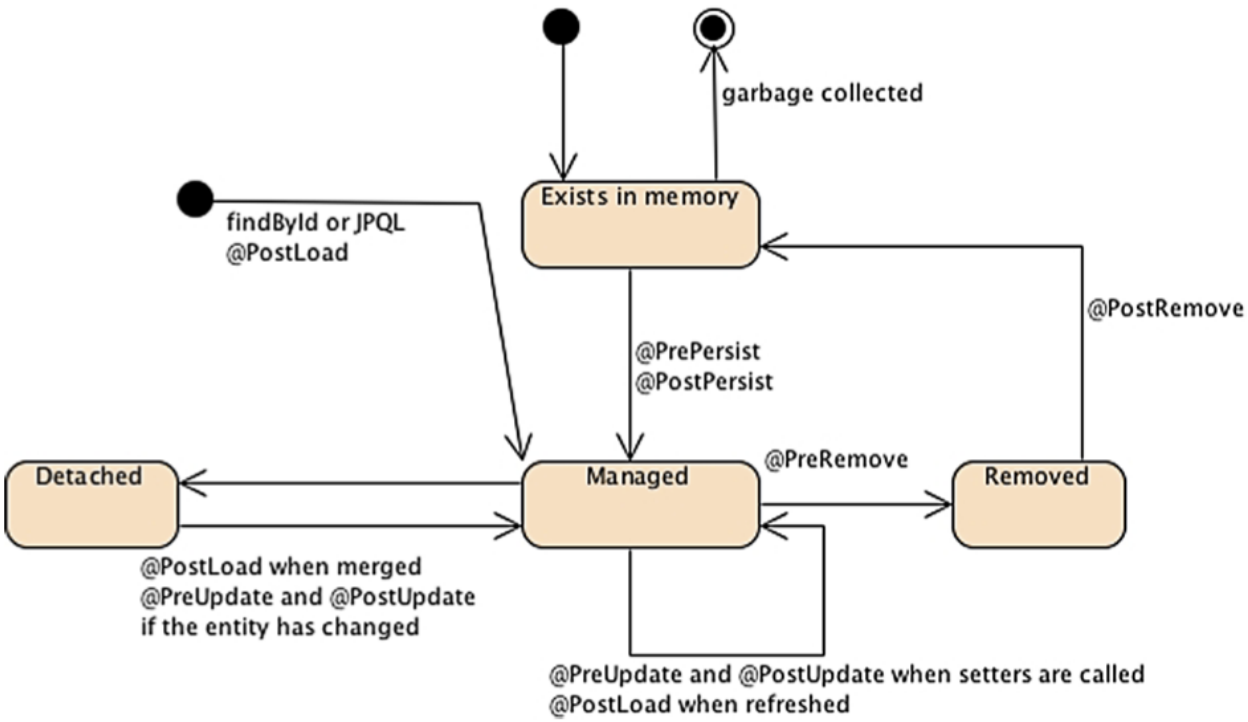
Pessimistic locking

- It is based on the opposite assumption to optimistic locking, because a lock is eagerly obtained on the entity before operating on it.
- This is very resource restrictive and results in significant performance degradation, as a database lock is held using a SELECT ... FOR UPDATE SQL statement to read data.
- Pessimistic locking may be applied to entities that do not contain the annotated `@Version` attribute.

Entity Life Cycle I



Callbacks I


garbage collected
Exists in memory
findById or JPQL
@PostLoad
@PostRemove
@PrePersist
@PostPersist
@PreRemove
Detached
Managed
Removed
@PostLoad when merged
@PreUpdate and @PostUpdate
if the entity has changed
@PreUpdate and @PostUpdate when setters are called
@PostLoad when refreshed

Callbacks II

```
@Entity
public class Customer {
      @Id @GeneratedValue
      private Long id;
      private String firstName;
      private String lastName;
      private String email;
      private String phoneNumber;
      @Temporal(TemporalType.DATE)
      private Date dateOfBirth;
      @Transient
      private Integer age;
      @Temporal(TemporalType.TIMESTAMP)
      private Date creationDate;
      @PrePersist
      @PreUpdate
      private void validate() {
            if (firstName == null || "".equals(firstName))
            throw new IllegalArgumentException("Invalid first name");
            if (lastName == null || "".equals(lastName))
            throw new IllegalArgumentException("Invalid last name");
      }
      @PostLoad
      @PostPersist
      @PostUpdate
      public void calculateAge() {
            if (dateOfBirth == null) {
                  age = null;
```

Callbacks III

```
                return;
            }
            Calendar birth = new GregorianCalendar();
            birth.setTime(dateOfBirth);
            Calendar now = new GregorianCalendar();
            now.setTime(new Date());
            int adjust = 0;
            if (now.get(DAY_OF_YEAR) - birth.get(DAY_OF_YEAR) < 0) {
                adjust = -1; }
            age = now.get(YEAR) - birth.get(YEAR) + adjust;
        }
        // Constructors, getters, setters
}
```

Note

- Methods can have public, private, protected, or package-level access but must not be static or final.
- A method may be annotated with multiple life-cycle event annotations (the validateData() method is annotated with @PrePersist and @PreUpdate). However, only one life-cycle annotation of a given type may be present in an entity class.
- A method can invoke JNDI, JDBC, JMS, and EJBs **but cannot invoke any EntityManager or Query operations**.

Callbacks IV

- With inheritance, if a method is specified on the superclass, it will get invoked **before** the method on the child class. For example, if Customer was inheriting from a Person entity, the Person `@PrePersist` method would be invoked before the Customer `@PrePersist` method.

Listeners I

- Callback methods in an entity work well when you have business logic that is only related to that entity.
- Entity listeners are used to extract the business logic to a separate class and share it between other entities.
- An entity listener is just a POJO on which you can define one or more life-cycle callback methods. To register a listener, the entity needs to use the `@EntityListeners` annotation.

```
public class AgeCalculationListener {
      @PostLoad
      @PostPersist
      @PostUpdate
      public void calculateAge(Customer customer) {
            if (customer.getDateOfBirth() == null) {
                  customer.setAge(null);
                  return;
            }
            Calendar birth = new GregorianCalendar();
            birth.setTime(customer.getDateOfBirth());
            Calendar now = new GregorianCalendar();
            now.setTime(new Date());
            int adjust = 0;     if (now.get(DAY_OF_YEAR) - birth.get(DAY_OF_YEAR) < 0) {
                  adjust = -1;
            }
            customer.setAge(now.get(YEAR) - birth.get(YEAR) + adjust);
      }
}
```

Listeners II

```
public class DataValidationListener {
      @PrePersist
      @PreUpdate
      private void validate(Customer customer) {
            if (customer.getFirstName() == null || "".equals(customer.getFirstName()))
            throw new IllegalArgumentException("Invalid first name");
            if (customer.getLastName() == null || "".equals(customer.getLastName()))
            throw new IllegalArgumentException("Invalid last name");
      }
}
```

A callback method defined on an entity has the following signatures

- `void <METHOD>();`
- `void <METHOD>(Object anyEntity);`
- `void <METHOD>(Customer customerOrSubclasses);`

Note

- When several listeners are defined and the life-cycle event occurs, the persistence provider iterates through each listener **in the order in which they are listed** and will invoke the callback method, passing a reference of the entity to which the event applies.

Listeners III

```
@EntityListeners({
      DataValidationListener.class, AgeCalculationListener.class})
@Entity
public class Customer {
      @Id @GeneratedValue
      private Long id;
      private String firstName;
      private String lastName;
      private String email;
      private String phoneNumber;
      @Temporal(TemporalType.DATE)
      private Date dateOfBirth;
      @Transient
      private Integer age;
      @Temporal(TemporalType.TIMESTAMP)
      private Date creationDate;
      // Constructors, getters, setters
}
```

Listeners IV

Default listeners

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd"
version="2.1">
 <persistence-unit-metadata>
  <persistence-unit-defaults>
   <entity-listeners>
    <entity-listener class="org.DebugListener"/>
   </entity-listeners>
  </persistence-unit-defaults>
 </persistence-unit-metadata>
</entity-mappings>
```

Listeners V

Exclude default listeners

```
@ExcludeDefaultListeners
@Entity
public class Customer {
      @Id @GeneratedValue
      private Long id;
      private String firstName;
      private String lastName;
      private String email;
      private String phoneNumber;
      @Temporal(TemporalType.DATE)
      private Date dateOfBirth;
      @Transient
      private Integer age;
      @Temporal(TemporalType.TIMESTAMP)
      private Date creationDate;
      // Constructors, getters, setters
}
```

When an event is raised, the listeners are executed in the following order:

- `@EntityListeners` for a given entity or superclass in the array order,
- Entity listeners for the superclasses (highest first),
- Entity listeners for the entity,
- Callbacks of the superclasses (highest first)
- Callbacks of the entity.

CDI 1.1 I

Introduction

- Inversion of control (IoC), meaning that the container would take control of your business code and provide technical services (such as transaction or security management).
- Taking control meant managing the life cycle of the components, bringing dependency injection and configuration to your components.
- CDI is built on the concept of "loose coupling, strong typing," meaning that beans are loosely coupled but in a strongly typed way

Understanding Beans

- Java SE has JavaBeans, Java EE has Enterprise JavaBeans.
- Java EE has other sorts of components such as Servlets, SOAP web services, RESTful web services, entities and of course Managed Beans.
- POJOs are just Java classes that run inside the Java Virtual Machine (JVM).
- JavaBeans are just POJOs that follow certain patterns (e.g., a naming convention for accessors/mutators (getters/setters) for a property, a default constructor) and are executed inside the JVM.
- Managed Beans are container-managed objects that support only a small set of basic services: resource injection, life-cycle management, and interception.

CDI 1.1 II

Dependency Injection

- Dependency Injection (DI) is a design pattern that decouples dependent components.
- It is part of inversion of control, where the concern being inverted is the process of obtaining the needed dependency.
- Instead of an object looking up other objects, the container injects those dependent objects for you.
- This is the so-called Hollywood Principle, "Don't call us? "(lookup objects), "we'll call you" (inject objects).
- Java EE 5 introduced a new set of annotations (@Resource, @PersistenceContext, @PersistenceUnit, @EJB, and @WebServiceRef).

Life-Cycle Management

- The life cycle of a POJO is pretty simple: as a Java developer you create an instance of a class using the new keyword and wait for the Garbage Collector to get rid of it and free some memory.
- If you want to run a CDI Bean inside a container, you are not allowed to use the new keyword.
- You need to inject the bean and the container does the rest, meaning, the container is the one responsible for managing the life cycle of the bean: it creates the instance; it gets rid of it.

CDI 1.1 III

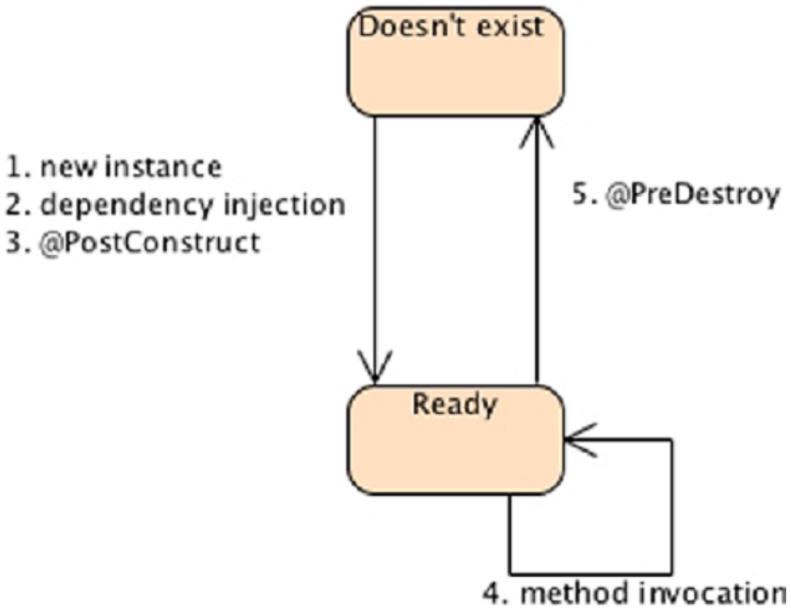


Figure: Managed Bean life cycle

CDI 1.1 IV

Scopes and Context

- CDI Beans may be stateful and are contextual, meaning that they live in a well-defined scope (CDI comes with predefined scopes: request, session, application, and conversation scopes).
- The container manages all beans inside the scope automatically for you and, at the end of the session, automatically destroys them.
- Different clients of a stateful bean see the bean in different states.

CDI 1.1 V

Interception

- Interceptors are used to interpose on business method invocations.
- In this aspect, it is similar to aspect-oriented programming (AOP). AOP is a programming paradigm that separates cross-cutting concerns (concerns that cut across the application) from your business code.
- Managed Beans support AOP-like functionality by providing the ability to intercept method invocation through interceptors.
- When you develop a session bean, you just concentrate on your business code.
- Behind the scenes, when a client invokes a method on your EJB, the container intercepts the invocation and applies different services (life-cycle management, transaction, security, etc.).
- With interceptors, you add your own cross-cutting mechanisms and apply them transparently to your business code.

CDI 1.1 VI

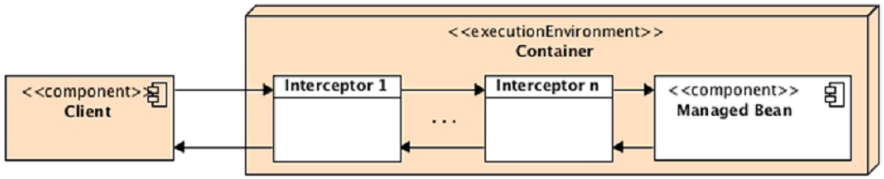


Figure: A container intercepting a call and invoking interceptors

CDI 1.1 VII

Loose Coupling and Strong Typing

- With injection a bean is not aware of the concrete implementation of any bean it interacts with.
- Beans can use event notifications to decouple event producers from event consumers or decorators to decouple business concerns.
-

CDI 1.1 VIII

Deployment Descriptor

- Nearly every Java EE specification has an optional XML deployment descriptor.
- With CDI, the deployment descriptor is called beans.xml and is mandatory.
- At deployment time, CDI checks all of your application's jar and war files and each time it finds a beans.xml deployment descriptor it manages all the POJOs, which then become CDI Beans.
- If your web application contains several jar files and you want to have CDI enabled across the entire application, each jar will need its own beans.xml to trigger CDI and bean discovery for each jar.

CDI 1.1 IX

A CDI Bean can be any kind of class that contains business logic. It may be called directly from Java code via injection, or it may be invoked via EL from a JSF page.

```
public class BookService {
      @Inject
      private NumberGenerator numberGenerator;
      @Inject
      private EntityManager em;
      private Date instanciationDate;
      @PostConstruct
      private void initDate() {
            instanciationDate = new Date();
      }
      @Transactional
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            book.setInstanciationDate(instanciationDate);
            em.persist(book);
            return book;
      }
}
```

A BookService Bean Using Injection Life-Cycle Management and Interception

Anatomy of a CDI Bean

- It is not a non-static inner class

CDI 1.1 X

- It is a concrete class, or is annotated @Decorator
- It has a default constructor with no parameters, or it declares a constructor annotated @Inject.

```
public class BookService {
      private NumberGenerator numberGenerator;
      public BookService() {
            this.numberGenerator = new IsbnGenerator();
      }
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            return book;
      }
}
```

A BookService POJO Creating Dependencies Using the New Keyword

CDI 1.1 XI

```
public class BookService {
      private NumberGenerator numberGenerator;
      public BookService(NumberGenerator numberGenerator) {
            this.numberGenerator = numberGenerator;
      }
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            return book;
      }
}

BookService bookService = new BookService(new IsbnGenerator());
BookService bookService = new BookService(new IssnGenerator());
```

A BookService POJO Choosing Dependencies Using the Constructor

This illustrates what inversion of control is: the control of creating the dependency between BookService and NumberGenerator is inverted because it's given to an external class, not the class itself. Since you end up connecting the dependencies yourself, this technique is referred to as construction by hand. we used the constructor to choose implementation (constructor injection), but another common way is to use setters (setter injection).

CDI 1.1 XII

@Inject

- As Java EE is a managed environment you don't need to construct dependencies by hand but can leave the container to inject a reference for you.
- In a nutshell, CDI dependency injection is the ability to inject beans into others in a typesafe way, which means no XML but annotations.
- Injection already existed in Java EE 5 with the @Resource, @PersistentUnit or @EJB annotations, for example. But it was limited to certain resources (datasource, EJB . . .) and into certain components (Servlets, EJBs, JSF backing bean . . .).
- With CDI you can inject nearly anything anywhere thanks to the @Inject annotation.

CDI 1.1 XIII

```
public class BookService {
      @Inject
      private NumberGenerator numberGenerator;
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            return book;
      }
}

public class IsbnGenerator implements NumberGenerator {
      public String generateNumber() {
            return "13-84356-" + Math.abs(new Random().nextInt());
      }
}
```

BookService Using @Inject to Get a Reference of NumberGenerator

CDI 1.1 XIV

Injection can occur via three different mechanisms: property, setter, or constructor. Notice that it isn't necessary to create a getter and a setter method on an attribute to use injection. CDI can access an injected field directly (even if it's private), which sometimes helps eliminate some wasteful code.

```
@Inject
private NumberGenerator numberGenerator;

@Inject
public BookService (NumberGenerator numberGenerator) {
      this.numberGenerator = numberGenerator;
}

@Inject
public void setNumberGenerator(NumberGenerator numberGenerator) {
      this.numberGenerator = numberGenerator;
}
```

The container is the one doing injection, not you (you can't invoke a constructor in a managed environment); therefore, there is only one bean constructor allowed so that the container can do its work and inject the right references.

CDI 1.1 XV

Default Injection

- Assume that NumberGenerator only has one implementation (IsbnGenerator). CDI will then be able to inject it simply by using @Inject on its own.

Whenever a bean or injection point does not explicitly declare a qualifier, the container assumes the qualifier @javax.enterprise.inject.Default.

```
@Inject
private NumberGenerator numberGenerator;

@Inject @Default
private NumberGenerator numberGenerator;
```

@Default is a built-in qualifier that informs CDI to inject the default bean implementation. If you define a bean with no qualifier, the bean automatically has the qualifier @Default

```
@Default
public class IsbnGenerator implements NumberGenerator {
      public String generateNumber() {
            return "13-84356-" + Math.abs(new Random().nextInt());
      }
}
```

The IsbnGenerator Bean with the @Default Qualifier

CDI 1.1 XVI

If you only have one implementation of a bean to inject, the default behavior applies and a straightforward @Inject will inject the implementation.

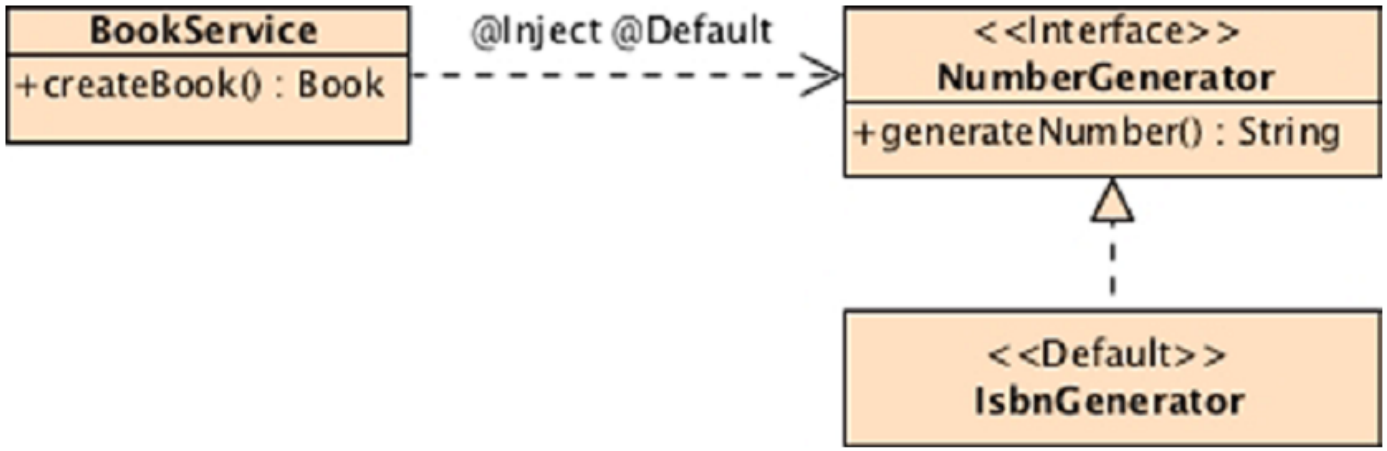


Figure: Class diagram with @Default injection

CDI 1.1 XVII

Qualifiers

- At system initialization time, the container must validate that exactly one bean satisfying each injection point exists. Meaning that if no implementation of NumberGenerator is available, the container would inform you of an unsatisfied dependency and will not deploy the application.
- If there is only one implementation, injection will work using the @Default qualifier.
- If more than one default implementation were available, the container would inform you of an ambiguous dependency and will not deploy the application. That's because the typesafe resolution algorithm fails when the container is unable to identify exactly one bean to inject.

CDI 1.1 XVIII

CDI uses qualifiers, which basically are Java annotations that bring typesafe injection and disambiguate a type without having to fall back on String-based names.

CDI 1.1 XIX

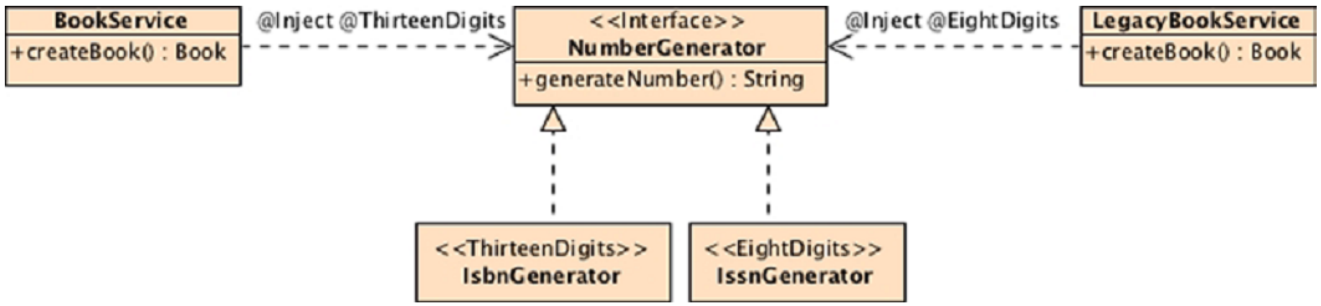


Figure: Services using qualifiers for non-ambiguous injection

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

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

CDI 1.1 XX

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {
      public String generateNumber() {
            return "13-84356-" + Math.abs(new Random().nextInt());
      }
}

@EightDigits
public class IssnGenerator implements NumberGenerator {
      public String generateNumber() {
            return "8-" + Math.abs(new Random().nextInt());
      }
}
```

CDI 1.1 XXI

```
public class BookService {
      @Inject @ThirteenDigits
      private NumberGenerator numberGenerator;
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            return book;
      }
}

public class LegacyBookService {
      @Inject @EightDigits
      private NumberGenerator numberGenerator;
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            return book;
      }
}
```

CDI 1.1 XXII

Qualifiers with Members

```
@Qualifier
@Retention(RUNTIME) @Target({
      FIELD, TYPE, METHOD}) public @interface NumberOfDigits {
      Digits value();
      boolean odd();
}
public enum Digits {
      TWO,
      EIGHT,
      TEN,
      THIRTEEN
}
```

```
@Inject @NumberOfDigits(value = Digits.THIRTEEN, odd = false)
private NumberGenerator numberGenerator;

@NumberOfDigits(value = Digits.THIRTEEN, odd = false)
public class IsbnEvenGenerator implements NumberGenerator {
}
```

Multiple Qualifiers

CDI 1.1 XXIII

```
@ThirteenDigits @Even
public class IsbnEvenGenerator implements NumberGenerator {
}

@Inject @ThirteenDigits @Even
private NumberGenerator numberGenerator;
```

Alternatives

- Qualifiers let you choose between multiple implementations of an interface at development time. But sometimes you want to inject an implementation depending on a particular deployment scenario. For example, you may want to use a mock number generator in a testing environment.
- Alternatives are beans annotated with the special qualifier javax.enterprise.inject.Alternative. By default alternatives are disabled and need to be enabled in the beans.xml descriptor to make them available for instantiation and injection.

CDI 1.1 XXIV

```
@Alternative
public class MockGenerator implements NumberGenerator {
      public String generateNumber() {
            return "MOCK";
      }
}

@Alternative @Default
public class MockGenerator implements NumberGenerator {
}

@Alternative @ThirteenDigits
public class MockGenerator implements NumberGenerator {
}
```

```
<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 ÂŹ
 http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">
    <alternatives>
        <class>org.agoncal.book.javaee7.chapter02. MockGenerator</class>
    </alternatives>
</beans>
```

CDI 1.1 XXV

```
@Inject
private NumberGenerator numberGenerator;
```

You can have several beans.xml files declaring several alternatives depending on your environment (development, production, test).

CDI 1.1 XXVI

Producers

- I've shown you how to inject CDI Beans into other CDI Beans. But you can also inject primitives (e.g., int, long, float), array types and any POJO that is not CDI enabled, thanks to producers. By CDI enabled I mean any class packaged into an archive containing a beans.xml file.
- By default, you cannot inject classes such as a java.util.Date or java.lang.String. That's because all these classes are packaged in the rt.jar file (the Java runtime environment classes) and this archive does not contain a beans.xml deployment descriptor.
- The only way to be able to inject POJOs is to use producer fields or producer methods.

```
public class NumberProducer {
      @Produces @ThirteenDigits
      private String prefix13digits = "13-";

      @Produces @ThirteenDigits
      private int editorNumber = 84356;

      @Produces @Random
      public double random() {
            return Math.abs(new Random().nextInt());
      }
}
```

CDI 1.1 XXVII

```
@ThirteenDigits
public class IsbnGenerator implements NumberGenerator {

      @Inject @ThirteenDigits private String prefix;

      @Inject @ThirteenDigits private int editorNumber;

      @Inject @Random
      private double postfix;
      public String generateNumber() {
            return prefix + editorNumber + postfix;
      }
}
```

CDI 1.1 XXVIII

InjectionPoint API

- There are certain cases where objects need to know something about the injection point into which they are injected.
- How would you produce a Logger that needs to know the class name of the injection point?
- CDI has an InjectionPoint API that provides access to metadata about an injection point.

```
Logger log = Logger.getLogger(BookService.class.getName());
```

```
Logger log = Logger.getLogger(BookService.class.getName());

public class LoginProducer {

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

To use the produced logger in any bean you just inject it and use it. The logger's category class name will then be automatically set

```
@Inject Logger log;
```

CDI 1.1 XXIX

Disposers

- Some producer methods return objects that require explicit destruction such as a Java Database Connectivity (JDBC) connection, JMS session, or entity manager.
- For creation, CDI uses producers, and for destruction, disposers.
- A disposer method allows the application to perform the customized cleanup of an object returned by a producer method.

```
public class JDBCConnectionProducer {
      @Produces
      private Connection createConnection() {
            Connection conn = null;
            try {
                  Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
                  conn = DriverManager.getConnection("jdbc:derby:memory:chapter02DB", "APP", "APP");
            } catch (InstantiationException | IllegalAccessException | ClassNotFoundException) {
                  e.printStackTrace();
            }
            return conn;
      }
      private void closeConnection(@Disposes Connection conn) throws SQLException {
            conn.close();
      }
}
```

CDI 1.1 XXX

- Destruction can be performed by a matching disposer method, defined by the same class as the producer method.
- Each disposer method, annotated with @Disposes, must have exactly one disposed parameter of the same type (here java.sql.Connection) and qualifiers (@Default) as the corresponding producer method return type (annotated @Produces).
- The disposer method (closeConnection()) is called automatically when the client context ends and the parameter receives the object produced by the producer method.

```
@ApplicationScoped
public class DerbyPingService {
      @Inject
      private Connection conn;
      public void ping() throws SQLException {
            conn.createStatement().executeQuery("SELECT 1 FROM SYSIBM.SYSDUMMY1");
      }
}
```

CDI 1.1 XXXI

Scopes

- Every object managed by CDI has a well-defined scope and life cycle that is bound to a specific context.
- In Java, the scope of a POJO is pretty simple: you create an instance of a class using the new keyword and you rely on the garbage collection to get rid of it and free some memory.
- With CDI, a bean is bound to a context and it remains in that context until the bean is destroyed by the container. There is no way to manually remove a bean from a context.

CDI defines the following built-in scopes and even gives you extension points so you can create your own

- Application scope (@ApplicationScoped)
- Session scope (@SessionScoped)
- Request scope (@RequestScoped)
- Conversation scope (@ConversationScoped)
- Dependent pseudo-scope (@Dependent)

CDI 1.1 XXXII

Conversation

- The conversation scope is slightly different than the application, session, or request scope. It holds state associated with a user, spans multiple requests, and is demarcated programmatically by the application.

```
@ConversationScoped
public class CustomerCreatorWizard implements Serializable {
      private Login login;
      private Account account;

      @Inject
      private CustomerService customerService;

      @Inject
      private Conversation conversation;

      public void saveLogin() {
            conversation.begin();
            login = new Login();
            // Sets login properties
      }
      public void saveAccount() {
            account = new Account();
            // Sets account properties
      }
```

CDI 1.1 XXXIII

```
    public void createCustomer() {
        Customer customer = new Customer();
        customer.setLogin(login);
        customer.setAccount(account);
        customerService.createCustomer(customer);
        conversation.end();
    }
}
```

CDI 1.1 XXXIV

Beans in Expression Language

```
@Named
public class BookService {
      private String title, description;
      private Float price;
      private Book book;
      @Inject @ThirteenDigits
      private NumberGenerator numberGenerator;
      public String createBook() {
            book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            return "customer.xhtml";
      }
}
```

```
<h:commandButton value="Send email" action="#{
      bookService.createBook}"/>
```

```
@Named("myService")
public class BookService {
}
```

```
<h:commandButton value="Send email" action="#{
      myService.createBook}"/>
```

CDI 1.1 XXXV

Interceptors

Constructor-level interceptors Interceptor associated with a constructor of the target class (@AroundConstruct)

Method-level interceptors Interceptor associated with a specific business method (@AroundInvoke)

Timeout method interceptors Interceptor that interposes on timeout methods with (@AroundTimeout)

Life-cycle callback interceptors Interceptor that interposes on the target instance life-cycle event callbacks (@PostConstruct and @PreDestroy)

CDI 1.1 XXXVI

```
@Transactional
public class CustomerService {
      @Inject
      private EntityManager em;
      @Inject
      private Logger logger;
      public void createCustomer(Customer customer) {
            em.persist(customer);
      }
      public Customer findCustomerById(Long id) {
            return em.find(Customer.class, id);
      }
      @AroundInvoke
      private Object logMethod(InvocationContext ic) throws Exception {
            logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
            try {
                  return ic.proceed();
            } finally {
                  logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
            }
      }
}
```

A CustomerService Using Around-Invoke Interceptor

CDI 1.1 XXXVII

```
public class LoggingInterceptor {
      @Inject
      private Logger logger;
      @AroundConstruct
      private void init(InvocationContext ic) throws Exception {
            logger.fine("Entering constructor");
            try {
                  ic.proceed();
            } finally {
                  logger.fine("Exiting constructor");
            }
      }
      @AroundInvoke
      public Object logMethod(InvocationContext ic) throws Exception {
            logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
            try {
                  return ic.proceed();
            } finally {
                  logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
            }
      }
}
```

An Interceptor Class with Around-Invoke and Around-Construct

CDI 1.1 XXXVIII

```
@Transactional
public class CustomerService {
      @Inject
      private EntityManager em;
      @Interceptors(LoggingInterceptor.class)
      public void createCustomer(Customer customer) {
            em.persist(customer);
      }
      public Customer findCustomerById(Long id) {
            return em.find(Customer.class, id);
      }
}
```

CustomerService Uses an Interceptor on One Method

```
@Transactional
@Interceptors(LoggingInterceptor.class)
public class CustomerService {
      public void createCustomer(Customer customer) {
            ...}
      public Customer findCustomerById(Long id) {
            ...}
}
```

CDI 1.1 XXXIX

```
@Transactional
@Interceptors(LoggingInterceptor.class)
public class CustomerService {
      public void createCustomer(Customer customer) {
      }
      public Customer findCustomerById(Long id) {
      }
      @ExcludeClassInterceptors
      public Customer updateCustomer(Customer customer) {
      }
}
```

Life-Cycle Interceptor

CDI 1.1 XL

```
public class ProfileInterceptor {
      @Inject
      private Logger logger;
      @PostConstruct
      public void logMethod(InvocationContext ic) throws Exception {
            logger.fine(ic.getTarget().toString());
            try {
                  ic.proceed();
            } finally {
                  logger.fine(ic.getTarget().toString());
            }
      }
      @AroundInvoke
      public Object profile(InvocationContext ic) throws Exception {
            long initTime = System.currentTimeMillis();
            try {
                  return ic.proceed();
            } finally {
                  long diffTime = System.currentTimeMillis() - initTime;
                  logger.fine(ic.getMethod() + " took " + diffTime + " millis");
            }
      }
}
```

An Interceptor with Both Life-Cycle and Around-Invoke

CDI 1.1 XLI

```
@Transactional
@Interceptors(ProfileInterceptor.class)
public class CustomerService {
      @Inject
      private EntityManager em;
      @PostConstruct
      public void init() {
            // ...
      }
      public void createCustomer(Customer customer) {
            em.persist(customer);
      }
      public Customer findCustomerById(Long id) {
            return em.find(Customer.class, id);
      }
}
```

CustomerService Using an Interceptor and a Callback Annotation

Chaining and Excluding Interceptors

CDI 1.1 XLII

```
@Stateless
@Interceptors({
      I1.class, I2.class})
public class CustomerService {
      public void createCustomer(Customer customer) {
            ...}
      @Interceptors({
            I3.class, I4.class})
      public Customer findCustomerById(Long id) {
            ...}
      public void removeCustomer(Customer customer) {
            ...}
      @ExcludeClassInterceptors
      public Customer updateCustomer(Customer customer) {
            ...}
}
```

CustomerService Chaining Serveral Interceptors

CDI 1.1 XLIII

Interceptor Binding

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

```
@Interceptor
@Loggable
public class LoggingInterceptor {
      @Inject
      private Logger logger;
      @AroundInvoke
      public Object logMethod(InvocationContext ic) throws Exception {
            logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
            try {
                  return ic.proceed();
            } finally {
                  logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
            }
      }
}
```

Loggable Interceptor

CDI 1.1 XLIV

```
@Transactional
@Loggable
public class CustomerService {
      @Inject
      private EntityManager em;
      public void createCustomer(Customer customer) {
            em.persist(customer);
      }
      public Customer findCustomerById(Long id) {
            return em.find(Customer.class, id);
      }
}
```

CustomerService using the Interceptor Binding

```
@Transactional
public class CustomerService {
      @Loggable
      public void createCustomer(Customer customer) {
            ...}
      public Customer findCustomerById(Long id) {
            ...}
}
```

CDI 1.1 XLV

Prioritizing Interceptors Binding

- Interceptor binding brings you a level of indirection, but you lose the possibility to order the interceptors (@Interceptors(I1.class, I2.class)).

```
@Interceptor
@Loggable
@Priority(200)
public class LoggingInterceptor {
      @Inject
      private Logger logger;
      @AroundInvoke
      public Object logMethod(InvocationContext ic) throws Exception {
            logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
            try {
                  return ic.proceed();
            } finally {
                  logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
            }
      }
}
```

```
@Interceptor
@Loggable
@Priority(Interceptor.Priority.LIBRARY_BEFORE + 10)
public class LoggingInterceptor {
}
```

CDI 1.1 XLVI

Decorator

```
@Decorator
public class FromEightToThirteenDigitsDecorator implements NumberGenerator {
      @Inject @Delegate
      private NumberGenerator numberGenerator;
      public String generateNumber() {
            String issn = numberGenerator.generateNumber(); String isbn = "13-84356" + issn.substrin
      }
}
```

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-insta
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
version="1.1" bean-discovery-mode="all">

<decorators>
<class>org.agoncal.book.javaee7.chapter02.FromEightToThirteenDigitsDecorator</class>

</decorators>

</beans>
```

CDI 1.1 XLVII

Events

- One bean can define an event, another bean can fire the event, and yet another bean can handle the event.

```
public class BookService {
      @Inject
      private NumberGenerator numberGenerator;
      @Inject
      private Event<Book> bookAddedEvent;
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber());
            bookAddedEvent.fire(book);
            return book;
      }
}
```

```
public class InventoryService {
      @Inject
      private Logger logger;
      List<Book> inventory = new ArrayList<>();
      public void addBook(@Observes Book book) {
            logger.info("Adding book " + book.getTitle() + " to inventory"); inventory.add(book);
      }
}
```

CDI 1.1 XLVIII

```
public class BookService {
      @Inject
      private NumberGenerator numberGenerator;
      @Inject @Added
      private Event<Book> bookAddedEvent;
      @Inject @Removed
      private Event<Book> bookRemovedEvent;
      public Book createBook(String title, Float price, String description) {
            Book book = new Book(title, price, description);
            book.setIsbn(numberGenerator.generateNumber()); bookAddedEvent.fire(book);
            return book;
      }
      public void deleteBook(Book book) {
            bookRemovedEvent.fire(book);
      }
}
```

CDI 1.1 XLIX

```
public class InventoryService {
      @Inject
      private Logger logger;
      List<Book> inventory = new ArrayList<>();
      public void addBook(@Observes @Added Book book) {
            logger.warning("Adding book " + book.getTitle() + " to inventory");
            inventory.add(book);
      }

      public void removeBook(@Observes @Removed Book book) {
            logger.warning("Removing book " + book.getTitle() + " to inventory");
            inventory.remove(book);
      }
}

void addBook(@Observes @Added @Price(greaterThan=100) Book book)
```