EGE ÜNİVERSİTESİ

BİLGİSAYAR MÜHENDİSLİĞİ

SUNUCU YAZILIM TEKNOLOJİLERİ

Java Persistence API

Hakan KURŞUN

Ahmet ALİŞAN

Murat BOZKURT

Kassım KINGALU

13 Mayıs 2014

İzmir

İÇİNDEKİLER

1. JPA

Java Persistence API, ilişkisel verilerin Java sınıfları ile tutulmasına ve uygulamanın sonlanmasından sonra verinin kalıcı olmasını sağlayan teknolojidir. JPA ile geliştiriciler SQL komutları yerine direk olarak nesneler üzerinde çalışabilir. SQL sorgularını JPA kendi içerisinde barındırır. Veritabanında verileri saklayabilir, güncelleyebilir veya veritabanından verileri çekebilir veya map edebilirsiniz. JPA, verilerin veritabanı üzerinde nasıl eşleşeceği, verinin güvenliği ve performansı ile ilgilenir. Uygulama ile veritabanı arasında arabirim noktasını oluşturur.

JPA teknolojisi ile ORM üzerinde belli bir standartın oluşturulması hedeflenmiştir. Peki ORM nedir?

ORM

İlişkisel veritabanı ve nesneye yönelik programlama dili arasında ilişki kurabildiğimiz programlama tekniğidir . Veritabanı tarafında bulunan tablolar ile sınıflar map edilir . ORM sayesinde INSERT , UPDATE , SELECT ve DELETE işlemleri çok rahat bir şekilde yapılabilir .

ORM mantığının kafamızda daha çok canlanması için şöyle bir örnek verebiliriz . Veritabanında KISI tablosunun olduğunu düşünelim

```
CREATE TABLE KISI(
   KIMLIK_NO INTEGER PRIMARY KEY,
   AD  VARCHAR2(50),
   SOYAD VARCHAR2(50)
);
```

ORM kullanabilmemiz için vertabanındaki tabloya karşılık gelen bir nesne oluşturmamız gerekmektedir . Alt tarafta java dilinde örnek bir sınıf oluşturulmuştur

```
public class Kisi {
	private Integer kimlikNo;
	private String ad;
	private String soyad;
}
```

Daha sonra bu tablo üzerinde select,insert,update,delete işlemleri için mapping(haritalama) yapmalıyız. Mapping in amacı tablo ve kolon bilgileriyle nesne arasında eşleme yapmaktır . Altta verilen kod örneğinde Hibernate kütüphanesi kullanılarak yapılmış Java sınıfında örnek bir mapping görülmektedir

```
@Table(name = "KISI")
public class Kisi {

	@Column(name = "KIMLIK_NO")
	private Integer kimlikNo;

	@Column(name = "AD")
	private String ad;

	@Column(name = "SOYAD")
	private String soyad;
}
```

Daha sonra KISI tablosunda bir veri insert etmek istediğimizde alttaki gibi bir kod yapısı kullanabiliriz

```
Kisi kisi = new Kisi();

kisi.setAd('Serkan');
```

```
kisi.setSoyad('Sakınmaz');
kisi.setKimlikNo(1234567);

entityManager.persist(kisi);
```

Görüldüğü gibi çok basit bir yöntemle insert işlemi yapılmıştır . Eğer bunu JDBC ile yapmak istersek daha uzun bir kod bloğu yazmamız gerekir . Ayrıca yazdığımız kodlar veritabanı bağımlı hale gelecektir .

ORM araçları ile veritabanı bagımsız bir şekilde kod yazabiliriz . Kullandığımız veritabanını bir dosyada belirtiriz . Örnek verirsek projemizi Oracle dan MSSQL le geçiriyorsak sadece bu dosyayı güncellememiz yeterlidir . Eğer JDBC ile yazsaydık bir çok yerde kod değişikliği gerekecekti .

Avantajları

- Yazılan kodun veritabanı ile bağımlılığını azaltır . Bu yüzden veritabanı değişmesi durumunda bağımlılık az olduğu için kod değişimide az olur
- Nesneye yönelik programlama metodolojisi sunar
- ORM ile daha az kod yazılır
- Çoğu ORM araçları bedava
- Development zamanını azaltır

Dezavantajları

- Performansı azaltabilir
- ORM araçlarını öğrenmek için ekstra efor harcanır
- ORM araçlarını projeye entegre etmek için ekstra efor harcanır

Populer ORM Araçları

- Hibernate : Java programlama dilinde en çok kullanılan ORM aracıdır. Şu an geliştirimi JBoss tarafından yapılmaktadır
- OpenJPA : Açık kaynak kodlu ORM aracıdır . Geliştirimi Apache tarafından yapılmaktadır. Hibernate kadar populer olmasada birçok şirket tarafından kullanılmaktadır
- Toplink : Oracle tarafından geliştirilen ORM aracıdır . Açık kaynak kodlu değildir.
- EclipseLink : Açık kaynak kodlu Eclipse şirketi tarafından geliştirimi yapılan ORM aracıdır

İşte bu kütüphanelerin hepsini kullanmak ve hepsini bilmek tabi tahmin edeceğiniz gibi zor ve zahmetli bir iş. Zaten JPA ile buna gerek kalmıyor şöyle ki JPA bu ürünlerin hepsininin çalışmasını tek çatı altında toplayan bir spesifikasyondur.

Yani JPA bir Hibernate veya Eclipse-Link gibi bir kütüphane değil, bunların kullanılmasını sağlayan bir spesifikasyondur. Yani biz JPA sayesinde Hibernate kütüphanesini veya diğer kütüphaneleri bilmek zorunda değiliz. JPA bu kütüphaneleri kullanmamızda bize ara bir katman oluyor. Git şu aracı kullanarak şu veritabanına şu veriyi kaydet diyorsunuz, o da gidip yapıyor. Geri kalanı da sizi ilgilendirmiyor.

2. JPA‘nın Avantajları Nelerdir?

Java Persistence API, Hibernate, TopLink, JDO gibi kalıcılık teknolojilerinin en iyi yönleri ve son zamanlardaki EJB kap(container) yönetimine dayalı kalıcılık üzerine kurulmuştur. Java Persistence API varolan tek bir yapı iskeletine dayanmaz aksine var olan pek çok güncel yapı iskeletinin desteklediği fikirleri içine alır,beraber çalışır ve geliştirir. Bu sayede uygulama geliştiriciler artık nesne ilişkisel modeli gerçekleştirmede uyumsuz ve standart olmayan kalıcılık modelleriyle yüzleşmek durumunda kalmamaktadır. Bununla beraber Java Persistence API daha çok uygulama geliştiriciye standart bir persistence API’ye sahip olma şansını vererek hem Java SE hem de Java EE içinde kullanıla- bilmektedir.

Java Persistence API uygulama geliştiricilere aşağıda açıklandığı üzere önemli avantajlar sağlamaktadır.

- JPA ile çalışırsak ORM implementasyonlarına gerek kalmaz!
- JPA, birçok ORM framework tarafından desteklenmektedir.
- JPA,uygulamayı ORM üreticisinden bağımsız kılar.
- XML ve açıklama tabanlı yapılandırmayı destekler.
- Security: Bir çok hack saldırısına karşı önlem barındırır.
- JPA kendine ait bir sorgu dili JPQL kullandığı için veritabanlarına özel sorgu kelimelerinden bağımsızdır.Yani uygulamanın bilgilerini depolayan veritabanınızı istediğiniz an sorunsuzca değiştirebilirsiniz.

- Yüksek performans; Java Persistence API, taşınabilir aygıtların sınırlı kaynakları ile baş edebilmek için yüksek performans sağlamaya odaklanmıştır. Uygulamalar SAP NetWeaver Application Server ’ın deneyim ve yatırımından Persistence API ile kar sağlamaktadır.
- İş mantığı; Bir işte veriyi değiştirme amaçlı bir çok erişim gerçekleşmektedir. Yapı iskeleti tüm veri erişimlerinin birbirine kenetlenmiş bir şekilde işletildiğinden yada geri alındığından emindir.
- Veri depolama araçlarından bağımsızlık; The Persistence API veri tabanında kütük sistemi gibi veri tutabilmektedir. Java Persistence API kullanan uygulamalar, aktifleştirilecek depolama türü MI tarafından otomatik olarak belirlendiği için o anki aygıt sürücü özelliklerinin en iyi şekilde kullanılmasını sağlamakla beraber başka özelliklere sahip aygıtlara tamamen taşınabilirliğine imkan vermektedir.Eğer uygulamanın J2ME CLDC aygıtına gelecekteki bağlantısı göz önünde bulundurulursa, depolama türü MI’ın bu çevrede doğru veri tutma yöntemini kullanmaya dikkat etmesiyle veri erişim katmanı tamamen yeniden kullanılabilir.
- Çoklu kullanıcı desteği; The Persistence API, MI’ın çoklu kullanıcı kavramında anahtar rolü oynar. Verinin ‘kullanıcı tarafından’ (diğer kullanıcılar için görünmez ve erişime kapalı) ve ‘Paylaşılan’ (aygıtın tüm kullanıcıları tarafından görülen ve erişime açık) olarak depolanmasına izin verir.
- Gelecekteki gelişimlere katılım SAP , MI istemcisinin ve Persistence API’yi içeren API’nin işlevselliğini arttırmaya devam edecektir. Akabinde Persistence API’yi

kullanan uygulamalar performans gelişimlerinden yararlanabilecek ve yeni özellikleri kolaylıkla kullanabileceklerdir

- Garantilenen geliştirme, yükseltme desteği; Persistence API kullanımıyla, MI herhangi bir uygulamanın yükseltme stratejisinde zorunlu olan uygulama verisi üzerinden kontrol kazanır. MI uygulama verisi hakkında bilgi sahibi değilse uygulamanın yükseltilmesi sırasında verinin tutulmasına önlem alamayacağı açıktır. Uygulmalar bu nedenle yükseltilmelerinde önlemlerini kendileri almalıdırlar ki bu da karmaşık ve hataya eğimli bir durum yaratır. Persistence API kullanımıyla uygulama veri kaybı olmadan kolaylıkla yükseltilebileceğinden emin olur.
- Test etmede daha az güç harcama; Özellikle JDBC kullanılırken, veri kalıcılığının kapsamlı test edilmesi oluşturulan SQL ifadelerinin sadece çalışma zamanında ortaya çıkan hatalarını bulmayı gerektirir. Bu nedenle her SQL ifade oluşturucusu güçlülüğünü kesinleştirmek için eksiksiz biçimde test edilmeye ihtiyaç duymaktadır. Java Persistence API ile uygulamalar sonuç almak için SAP'ın test etme gücünü kullanır ve alanındaki ilave yatırımlardan kaçınmış olur.
- Veri kalıcılığının teknik geçmişinden bağımsızlık; Güçlü, gerçekleştirilen, çok kullanıcılı veri kalıcılığı büyük ustalık istemenin yanında veri tabanındaki etkinlik alanları, kütük sistemleri, performans ve Java'nın özü hakkında temel bilgi gerektirmektedir. Java Persistence API kullanarak uygulama geliştiricileri tekniksel ayrıntılar yerine iş senaryosuna yoğunlaşa- bilmektedir.
- Classlar ile çalişmak; %100 Object Oriented olan bir programlama (OOP) dilinde siz veritabanı ile alakalı bir işlem yaptığınızda artık String datalarla değilde OOP

nin en temel yapısı olan Class ve Objectlerle uğraşıyor oluyorsunuz. Bu da veritabanının, bildiğiniz ve iyi yaptığınız bir şeye dönüşmesini sağlıyor.

- Java Persistence API diğer önemli bir yeteneği de EJB 2.1'de olmayan kalıtım ve çokbiçimlilik desteğidir. İlişkisel veritabanı yapısında bir varlığın diğerinden türeyen bir altsınıf olduğu ve atasınıfa karşı sorgu kabul ettiği bir varlık hiyerarşisi eşlemesi yapılabilmektedir. Sorgular çok biçimli olarak tüm hiyerarşiye uygulanır.

3. JPA'nın Bazı Temel Özellikleri Neledir?

Yazımızın başında JPA'nın ilişkisel verilerin Java sınıfları ile tutulmasına ve uygulamanın sonlanmasından sonra verinin kalıcı olmasını sağlayan teknoloji olduğunu belirtmiştik.

Bu teknolojiyi kullanarak veri tabanında kalıcı kılınacak olan nesneye *entity* denir. Java nesneleri ve ilişkisel veri tabanı arasındaki eşlemeyi tanımlamak için Java dili ara veri (metadata) için kod içine yazılan özel açıklamaları (annotations) ve /veya XML tanımlayıcılarının kullanımını destekleyen eksiksiz bir nesne/ilişkisel eşleme belirtimi içermektedir.

```
@Entity
public class Customer1 {
    @Id
    private long custId;
    private String firstName;
```

```
        private String lastName;
        public Customer1(){},

        public Customer1(long custId, String firstName, String lastName) {
                this.custId = custId;
                this.firstName = firstName;
                this.lastName = lastName;
        }
        // set ve get metotları
}
```

Bir nesnenin entity olması için gerek ve yeter şart o nesnenin kendisinden üretileceği sınıfın ya *@Entity* notuyla notlandırılması ya da XML ayarlarında entity olarak ifade edilmesidir. Entitynin veri tabanında saklanancak olan kalıcı durumu (persistent state), tamamen nesne değişkenlerinden oluşacaktır. Entity bir iç sınıf (inner class), bir interface ya da enum tipi veya final olmamalıdır ve public ya da protected olan bir argumansız yapılandırıcısı olmalıdır. Nesne eğişkenleri private olabilir hatta olmalıdır ve kalıcı olan nesnenin istemcileri (client), nesnenin değişkenlerine get/set metotları ile ulaşmalıdır. Entitylerin kalıcılığının minimum sayıda alandan (nesne değişkeni) oluşması gerektiği düşünüldüğünde, veri tabanında kalıcı olması gerekmeyen değişkenler *@Transient* ile notlandırılmasının gerekir.

JPA çalışma zamanı yapıları, veri tabanında *@Entity* ile notlandırılmış sınıfla aynı isme sahip bir tablo arayacak ve tablonun sütunlarının da sınıfın kalıcı alanlarıyla aynı isme sahip olmasını bekleyecektir.

Yani JPA' nın özelliklerini maddelersek :

- Entity öncelikle bir sınıf olmalıdır, enum ya da interface entity olamaz.
- Sınıf, en üst seviyede bir sınıf olmalıdır. Yani entity olacak sınıf, bir iç sınıf (inner class) olmamalıdır.
- Benzer şekilde entity sınıfı final olmamalı ve herhangi bir final kalıcı alan ve metoda sahip olmamalıdır. final anahtar kelimesinin kullanımı ile ilgili bu kısıtlamanın nedenini ileride göreceğiz.
- Sınıf, muhakkak public ya da protected olan bir varsayılan (default) yani argümansız (no-arg) kurucuya (constructor) sahip olmalıdır. Entity sınıfı, tabi olarak, varsayılan kurucu yanında, argüman alan pek çok farklı kurucuya sahip olabilir.
- Eğer entity uzak ara yüzlere (remote interface) geçilecekse, sınıfınjava.io.Serializable ara yüzünü gerçekleştirmesi de gereklidir.
- Geliştirme ve test amaçlı olarak hızlıca yazılan uygulamalarda JPA'in veri tabanında şema (schema) oluşturma yeteneğinden faydalanılır
- Entity soyut bir sınıf olabilir. Bu durumda belli ki bir kalıtım hiyerarşisi vardır ve soyut olan bu sınıftan miras devralan tüm sınıflar da doğrudan entity olurlar.
- Entity sınıfı, entity olmayan bir başka sınıftan miras devir alabileceği gibi, entity olmayan bir sınıf da entity olan bir başka sınıftan miras devir alabilir. İlerideki bölümlerde, miras devir alma ile ilgili ayrıntıları ele alacağız.
- Entitynin durumunu oluşturan nesne değişkenleri daima sarmalama (encapsulation) prensibine uygun bir şekilde yani bean özellikleri olarak ifade edilmelidir. Dolayısıyla entitynin müşterileri ya da istemcileri (clients), entitynin

kalıcı olan nesne değişkenlerine ancak *getter-setter* metotları ya da diğer iş metotları (business methods) üzerinden erişebilmelidir. Bu anlamda kalıcı olan nesne değişkenleri private ya da devir alınma durumu göz önüne alınarak protected olarak nitelenmeli ve bu değişkenlerin, erişim ve değişme durumları göz önüne alınarak, ilgili set ve get metotları oluşturulmalıdır.

Sonuç olarak Java Persistence üç alan içerir;

- Java kalıcılık uygulama programı arayüzü.(Java Persistence API)
- Sordu dili (query language)
- Nesne/ ilişkisel eşlemele ara verisi. (Object/relational mapping metadata)

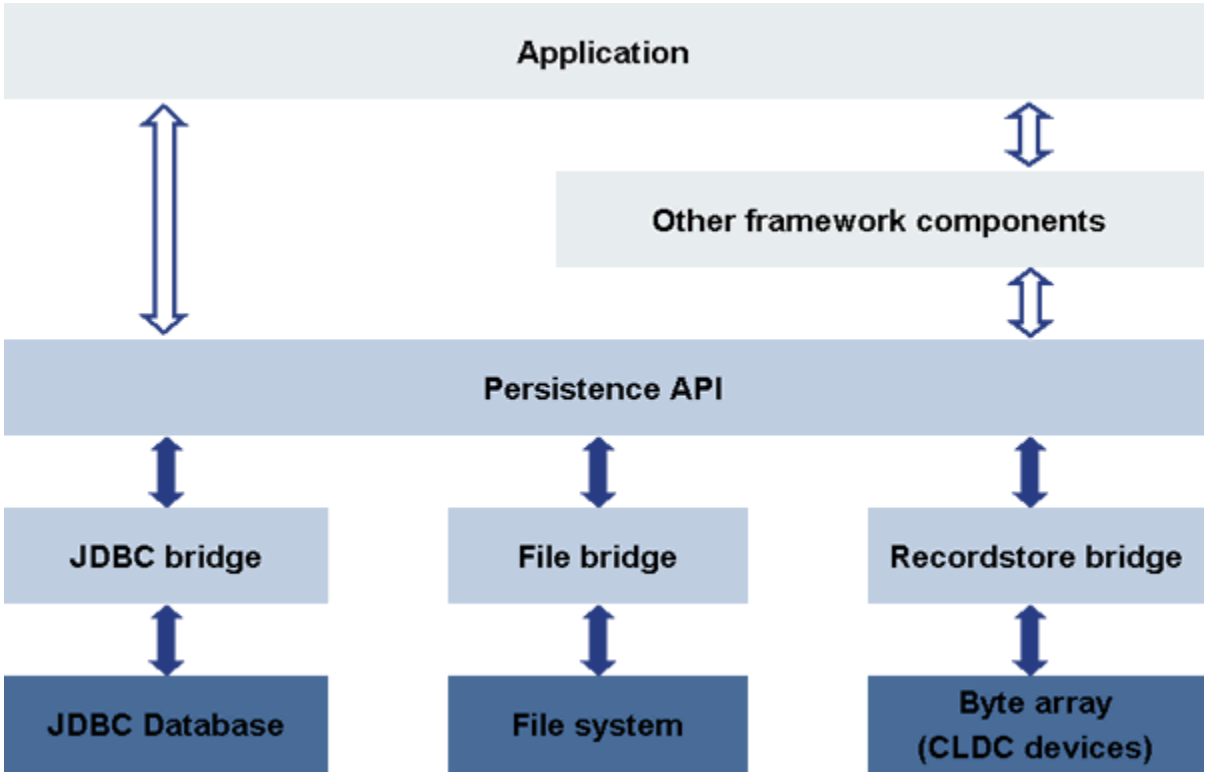
API, Java sınıflarını tanımlar.

The Persistence API amaçlarını aşağıdaki başlıklar altında toplayabiliriz:

- İlişkisel veri tabanı(relational database) ve nesne uyumsağlaması (object orientation)'nın gücünü birleştirmek.
- Nesnelerin birçoğunun minumum biçim dönüştürmeyle kalıcılığını sağlamak.
- Kalıcılığı sağlanan verinin bütün uygulamalarca saydam olmasını sağlamak.
- Veriler üzerinde sorguları desteklemek
- Platform / depolama yönteminden bağımsızlığı başarmak.
    - Depolama gerçekleştirimi uygulama için bir kara kutu olmalıdır.
    - Depolama gerçekleştirimi;
        - Kütük serileştirme
        - JDBC destekleyen herhangi bir veri tabanı
        - Kayıt depolama(J2ME CLDC) 'yı temel almalıdır.

- Taşınabilir uygulamalar ve MI yapı iskeleti için bir tane standart API sağlamak.
- Geliştirme,yükseltme (upgrade) strajesini desteklemek.
- Çoklu kullanıcı (multiple-user) desteği sağlamak.

Aşağıdaki grafik genel mimariyi açıklamaktadır:



JPA map işlemini belli kurallar çerçevesinde yapar. Bu işlemi yaparken kendi metadatalarını kullanır. Bağlantı ve kuralları oluşturduğu persistence.xml dosyası ile tutar. Bu dosya içerisinde veritabanı bağlantısı bilgileri, tablo oluşturma metodları gibi bilgiler bulunur.

JPA da Güvenlik

- İlişkisel veri tabanları, adından da anlaşılacağı üzere, uygulamalarda nesnelerde ifade ettiğimiz durum bilgilerini, bu bilgiler arasındaki ortak alanlardan yola

çıkarak, tablolar ve tablolar arası ilişkilerle tanımlamayı hedef edinmiştir. Bu şekilde, bilginin en az yer kaplayarak, sağlıklı bir şekilde saklanması, korunması ve gerektiğinde tekrar sunulması amaçlanmıştır.

Yüksek Performanslı Sorgulama : JPA güvenliği ile erişim kontrolünüz veritabanında gerçekleştirilir. Belirli tipteki tüm nesneler için veritabanına sorgu göndermelisiniz ve veritabanından sadece erişmek istediğiniz nesnelere ulaşabilirsiniz. Bu filtreleme işlemi veritabanında gerçekleşir. Erişilemeyen sorgu nesneleri hafızaya yüklenmeyecektir.

Erişim Kontrolü :

-Gösterime(Presentation) Bırakılan Erişim Kontrolü : Gösterime bırakılan erişim kontrolünde kullanıcıya sadece görmesine müsade edilen veriler ekranda gösterilir, kullanıcıya sadece erişmesine izin verilen bağlantılar(button dan gelen bağlantılar) ve eylem(actions)ler gösterilir ve kullanmasına izin verilir. Böylece kullanıcının hangi verileri görmesinin filtrelemesi yapılmış olur.

-Servis Seviyesinde Erişim Kontrolü : Metota dayalı erişim kontrolünde kullanılır. Hangi kullanıcının/rolün bu servise erişebileceği belirlenmiştir.

-Sınıf Seviyesinde Erişim Kontrolü : Hangi sınıfın/rolün hangi veriye erişebileceği tanımlanarak veriye erişim filtrelenmiş olur.

JPA da erişim Kontrolü :

JPA güvenliğin de entitilere ve embeddables (içine gömülebilirlere ?) erişim erişim kuralları tarafından belirlenmiştir. Her güvenlik ünitesi için bir erişim kural dizisi belirlenmiştir. Güvenlik ünitesinde karşılığı olan bu erişim kuralları kalıcı bölümdeki

entitiy manager dan almak istediğiniz tüm sorgulara, entitilere ve embeddable (gömülebilirlere) uygulanır.(Orjinal hali : *This set of access rules applies to every JPA query and every entity and embeddable you get out of anEntityManager of the persistence unit that corresponds to that security unit.*)

Örnek erişim kuralı :

GRANT READ ACCESS TO Account account WHERE account.owner = URRENT_PRINCIPAL

Bu kural hesap okumayı kısıtlar, her kullanıcının sadece kendi hesabını okumasını sağlar.

4. JPA da Varlılıklar ve Varlıkların Yönetimi

JPA Entity classes are user defined classes whose instances can be stored in a database.

Requirements for Entity Classes

An entity class must follow these requirements:

- The class must be annotated with the javax.persistence.Entity annotation.
- The class must have a public or protected, no-argument constructor. The class may have other constructors.
- The class must not be declared final. No methods or persistent instance variables must be declared final.
- If an entity instance be passed by value as a detached object, such as through a session bean’s remote business interface, the class must implement the Serializable interface.

- Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
- Persistent instance variables must be declared private, protected, or package-private, and can only be accessed directly by the entity class's methods. Clients must access the entity's state through accessor or business methods.

JPA Entity Fields

Fields of persistable user defined classes (entity classes, embeddable classes and mapped superclasses) can be classified into the following five groups:

- Persistent Identifiers fields
- Transient fields
- Persistent fields
- Inverse (Mapped By) fields
- Version field

The first three groups (transient, persistent and inverse fields) can be used in both entity classes and embeddable classes. However, the last two groups (primary key and version fields) can only be used in entity classes.

Persistent Identifiers fields

Entities must define an id field/fields corresponding the the database primary key.
The id can either be simple or composite value

Strategies:

- @Id: single valued type - most common
- @IdClass: map multiple fields to table PK
- @EmbeddedId map PK class to table PK

Automatic Primary Key

By default the primary key is a sequential 64 bit number (long) that is set automatically by for every new entity object that is stored in the database. The primary key of the first entity object in the database is 1, the primary key of the second entity object is 2, etc. Primary key values are not recycled when entity objects are deleted from the database.

The primary key value of an entity can be accessed by declaring a primary key field:

```
@Entity
public class Project {
    @Id @GeneratedValue long id; // still set automatically
     :
}
```

The @Id annotation marks a field as a primary key field. When a primary key field is defined the primary key value is automatically injected into that field.
The @GeneratedValue annotation specifies that the primary key is automatically allocated.

Application Set Primary Key

If an entity has a primary key field that is not marked with @GeneratedValue, automatic primary key value is not generated and the application is responsible to set a primary key by initializing the primary key field. That must be done before any attempt to persist the entity object:

```
@Entity
public class Project {
    @Id long id; // must be initialized by the application
     :
}
```

A primary key field that is set by the application can have one of the following types:

- Primitive types: boolean, byte, short, char, int, long, float, double.
- Equivalent wrapper classes from package java.lang:

- Byte, Short, Character, Integer, Long, Float, Double.
- java.math.BigInteger, java.math.BigDecimal.
- java.lang.String.
- java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp.
- Any enum type.
- Reference to an entity object.

Composite Primary Key

A composite primary key consist of multiple primary key fields. Each primary key field must be one of the supported types listed above.

For example, the primary key of the following Project entity class consists of two fields:

```
@Entity @IdClass(ProjectId.class)
public class Project {
    @Id int departmentId;
    @Id long projectId;
     :
}
```

When an entity has multiple primary key fields, JPA requires defining a special ID class that is attached to the entity class using the @IdClass annotation. The ID class reflects the primary key fields and its objects can represent primary key values:

Composite PK classes must :
-implement Serializable
-override equals() and hashCode().

```
Class ProjectId  implements Serializable {
    int departmentId;
    long projectId;
    public boolean equals(Object obj);
    public int hashCode();
}
```

Embedded Primary Key

An alternate way to represent a composite primary key is to use an embeddable class:

```
@Entity
public class Project {
    @EmbeddedId ProjectId id;
     :
}

@Embeddable
Class ProjectId {
    int departmentId;
    long projectId;
}
```

The primary key fields are defined in an embeddable class. The entity contains a single primary key field that is annotated with @EmbeddedId and contains an instance of that embeddable class. When using this form a separate ID class is not defined because the embeddable class itself can represent complete primary key values.

Obtaining the Primary Key

JPA 2 provides a generic method for getting the object ID (primary key) of a specified managed entity object. For example:

```
PersistenceUnitUtil util = emf.getPersistenceUnitUtil();
Object projectId = util.getIdentifier(project);
```

A PersistenceUnitUtil instance is obtained from the EntityManagerFactory. The getIdentifier method takes one argument, a managed entity object, and returns the

primary key. In case of a composite primary key - an instance of the ID class or the embeddable class is returned

Persistent Fields

Every non-static non-final entity field is persistent by default unless explicitly specified otherwise (e.g. by using the @Transient annotation).
Storing an entity object in the database does not store methods or code. Only the persistent state of the entity object, as reflected by its persistent fields (including persistent fields that are inherited from ancestor classes), is stored.
When an entity object is stored in the database every persistent field must contain either null or a value of one of the supported persistable types.

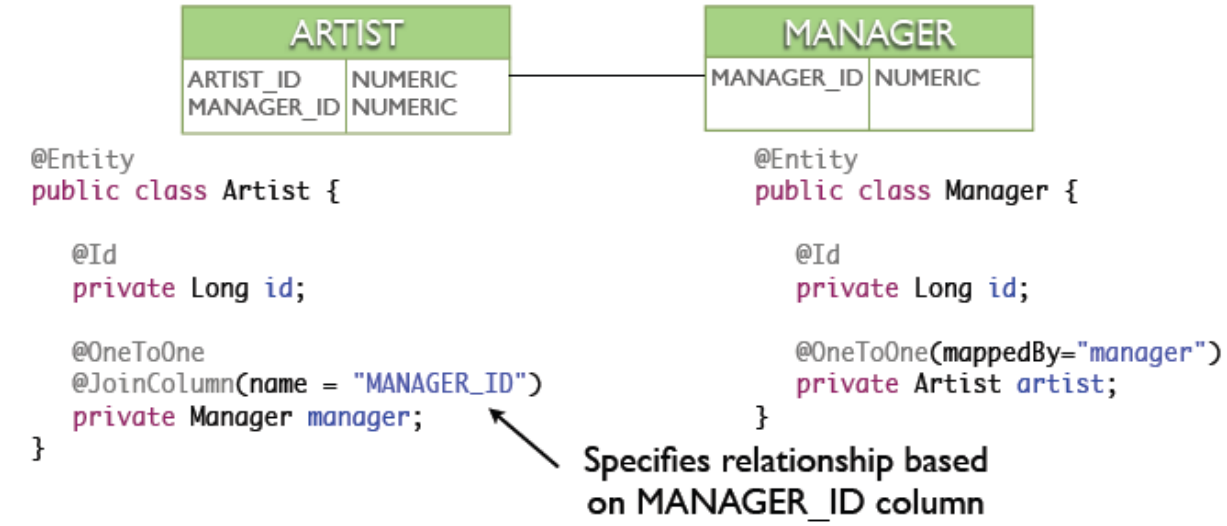
Every persistent field can be marked with one of the following annotations:

- OneToOne, ManyToOne - for references of entity types.
- OneToMany, ManyToMany - for collections and maps of entity types.
- Basic - for any other persistable type.

In JPA only Basic is optional while the other annotations above are required when applicable.

@OneToOne

Can be based on shared primary key or foreign key relationship using either @PrimaryKeyJoinColumn or @JoinColumn



}

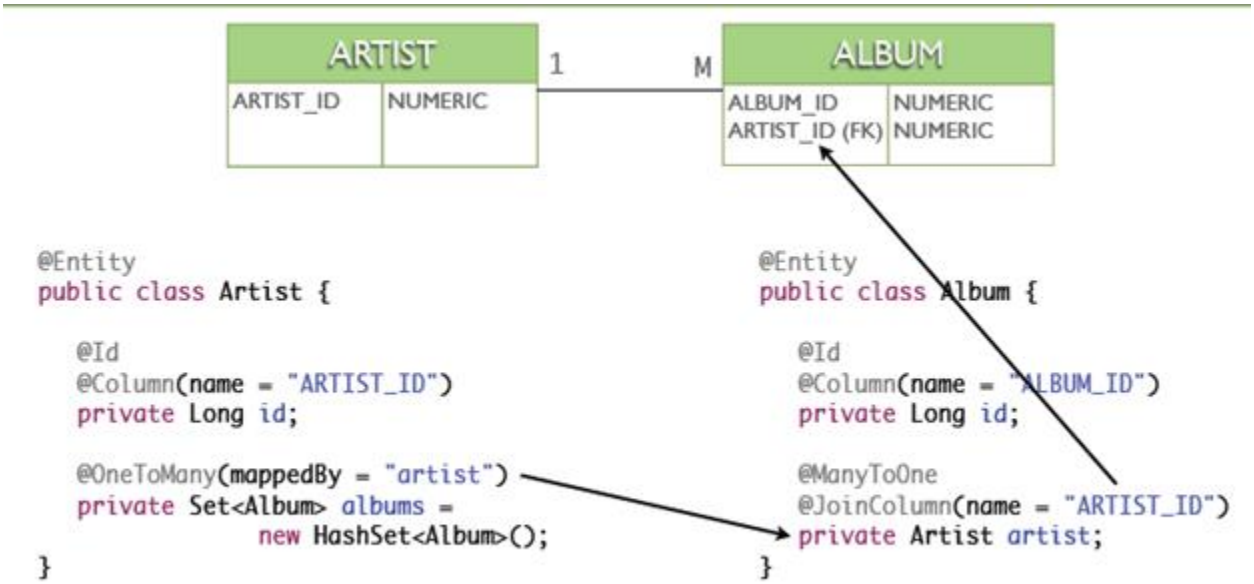
@OneToMany

@OneToMany defines the one side of a one-tomany relationship.
The *mappedBy* element of the annotation defines the object reference used by the child entity.
@OrderBy defines an collection ordering required when relationship is retrieved.
The *child* (many) side will be represented using an implementation of the java.util.Collection interface

@ManyToOne

@ManyToOne defines the many side of a one-tomany relationship
@JoinColumn defines foreign key reference
The *many* side is considered to be the owning side of the relationship

One-To-Many Example



Transient Fields

Transient entity fields are fields that do not participate in persistence and their values are never stored in the database (similar to transient fields in Java that do not participate in serialization). Static and final entity fields are always considered to be transient. Other fields can be declared explicitly as transient using either the Java transient modifier (which also affects serialization) or the JPA @Transient annotation (which only affects persistence):

```
@Entity
public class EntityWithTransientFields {
   static int transient1; // not persistent because of static
   final int transient2 = 0;  // not persistent because of final
   transient int transient3; // not persistent because of transient
   @Transient int transient4; // not persistent because of @Transient
}
```

The above entity class contains only transient (non persistent) entity fields with no real content to be stored in the database.

Inverse Fields

Inverse (or mapped by) fields contain data that is not stored as part of the entity in the database, but is still available after retrieval by a special automatic query.

The following entity classes demonstrate a bidirectional relationship:

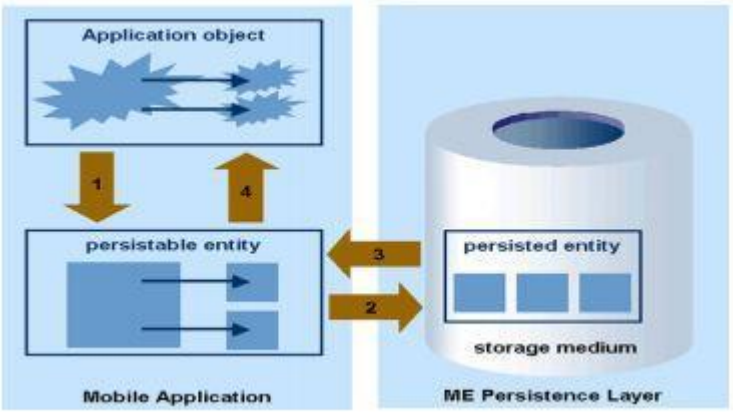
```
@Entity
public class Employee {
    String name;
    @ManyToOne Department department;
}

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

The mappedBy element (above) specifies that the employees field is an inverse field rather than a persistent field. The content of the employees set is not stored as part of a Department entity. Instead, employees is automatically populated when a Department entity is retrieved from the database.

Varlıkların Yönetimi

Varlıklar varlık yönetici (entity manager) tarafından yönetilir. Varlık Yöneticisi javax.persistence.EntityManager olguları ile temsil edilir. Her EntityManager olgusu bir kalıcılık bağlamı ile ilişkilidir. Bir kalıcılık bağlamı, belirli bir varlık olgusunun hangi etkinlik alanında yaratılıp, devam ettirilip, kaldırıldığını tanımlar.



Kalıcılık Bağlamı(Persistence Contex)

Bir kalıcılık bağlamı belirli bir veri deposunda bulunan yönetilen varlık olgularının kümesidir.

EntityManager arayüzü kalıcılık bağlamı ile iletişim kurmak için gerekli metodları tanımlar.

Varlık Yönetici(EntityManager)

EntityManager API kalıcı varlık olgularını yaratıp kaldırır, varlıkların birincil anahtarlarıyla varlıkları bulur, ve sorguların varlıklar üzerinde işletilmesine izin verir.

2 farklı tip varlık yöneticisi bulunmaktadır bunlar;

a. Kap Tabanlı(Container-Managed) Varlık Yöneticiler

Bir kap tabanlı varlık yönetici ile, bir EntityManager olgusunun kalıcılık bağlamı, otomatik olarak, EntityManager olgusunu tek bir Java Transaction Architecture (JTA) hareketi ile kullanan uygulama bilenşenlerinin container'ı tarafından üretilir.

JTA hareketleri çoğunlukla uygulama bileşenlerinin karşılıklı çağrılarını kapsar. Bir JTA hareketini tamamlamak için, bu bileşenler çoğunlukla tek bir kalıcılık bağlamına erişmeye ihtiyaç duyarlar. Bu durum uygulama bileşenlerine javax.persistence.PersistenceContext açıklamaları aracılığı ile bir EntityManager atandığında oluşur. Kalıcılık bağlamı o anki JTA hareketi ile otomatik olarak üretilir ve aynı kalıcılık birimine eşlenen EntityManager referansları, o hareket içinde kalıcılık bağlamına erişim sağlar.

Kalıcılık bağlamının otomatik oluşturulması sayesinde uygulama bileşenleri tek bir hareket içinde sırayla değişiklik yapmak için EntityManager olgularını birbirlerine referans geçmek zorunda kalmazlar Java EE container'ı kap tabanlı varlık yöneticilerinin yaşam devirlerini yönetir EntityManager olgusu elde etmek için ,uygulama bileşenine varlık yöneticiyi bağlarız.

```
@PersistenceContext
EntityManager em;
```

b. Uygulama Tabanlı(Application-Managed) Varlık Yöneticiler

Bir uygulama tabanlı varlık yönetici ile, kalıcılık bağlamı uygulama bileşenlerine üretilmez, ve

EntityManager olgularının yaşam döngüsü uygulama tarafından yönetilir. Uygulama tabanlı varlık yöneticiler, uygulama belirli bir kalıcılık birimindeki EntityManager olgusu içindeki JTA hareketi ile üretilmeyen bir kalıcılık bağlamına erişmek istediğinde kullanılır. Bu durumda her EntityManager yeni,ayrı bir kalıcılık bağlamı yaratır.

EntityManager ve onunla ilişkili kalıcılık bağlamı, uygulama tarafından yaratılıp yok edilebilir

Uygulamalar bu durumda EntityManager olgularını javax.persistence.EntityManagerFactory'nin createEntityManager metodu ile yaratırlar.

EntityManager olgusu elde etmek için, öncelikle javax.persistence.PersistenceUnit açıklamaları aracılığı ile bir EntityManager- Factory olgusu elde edilir ve uygulama bileşenine eklenir.

```
@PersistenceUnit
EntityManagerFactory emf;
EntityManager em = emf.createEntityManager();
```

Varlık Yönetici Kullanarak Varlıkların Bulunması

EntityManager.find metodu kullanılarak veri deposundaki olgulara olguların birincil anahtarları vasıtasıyla bakılır.

```
@PersistenceContext
EntityManager em;
public void enterOrder(int custID, Order newOrder) {
   Customer cust = em.find(Customer.class, custID);
   cust.getOrders().add(newOrder);
   newOrder.setCustomer(cust);
}
```

Bir varlık olgusunun yaşam döngüsünü yönetme

Varlık olgularını Entity-Manager olgusu aracılığı ile varlıklar üzerinde işlemler gerçekleştirerek yönetiriz. Varlık olguları dört durumdan birinde olabilir : yeni, yönetilen, bağlantısız yada kaldırılmış. Yeni varlık olgularının kalıcı kimlikleri yoktur ve henüz bir

kalıcılık bağlamı ile ilişkilendirilmemiştir. Yönetilen varlık olguları kalıcı kimliğe sahiptir ve bir bir kalıcılık bağlamı ile ilişkilendirilmiştir. Bağlantısız varlık olguları kalıcı kimliğe sahiptir ve henüz bir kalıcılık bağlamı ile ilişkilendirilmemiştir. Kaldırılmış varlık olguları kalıcı kimliğe sahiptir ,bir kalıcılık bağlamı ile ilişkilendirilmiştir ve veri deposundan kaldırılmak üzere listelenmiştir.

Varlık Olgularının Kalıcılaştırılması

Yeni varlık olguları persist metodları çağırlarak kalıcı ve yönetilen hale getirilir. Bu, varlığın verisinin veritabanında ,persist işlemi ile ilişkili hareket tamamlandığında saklandığı anlamına gelir.

Eğer varlık zaten yönetiliyorsa, persist işlemi gözardı edilir. Eğer persist kaldırılmış bir varlık olgusu için çağırılırsa, o olgu yönetilen hale gelir. Eğer varlık bağlantısız ise persist metodu bir IllegalArgumentException aykırı durumu fırlatır, yada hareket işlemi başarısızlıkla sonuçlanır.

```
@PersistenceContext
EntityManager em;
...
public LineItem createLineItem(Order order, Product product, int quantity) {
   LineItem li = new LineItem(order, product,   quantity);
   order.getLineItems().add(li);
   try {
            em.getTransaction().begin();
            // TODO:
            em.persist(li);
            em.getTransaction().commit();
  } catch(Exception e) {
          Logger.getLogger(getClass().getName()).log(Level.SEVERE,"exception
caught", e);
            em.getTransaction().rollback();
  } finally {
            em.close();
  }

   return li;
}
```

Varlık Olgusunu Kaldırma

Yönetilen varlık olguları kendisinin remove metodu çagırılarak kaldırılır, ya da bu yönetilen varlıkla başka bir varlık arasındaki bağıntı belirtiminde cascade=REMOVE yada cascade=ALL elemanları kümesi varsa bu ilişkili varlık tarafından içiçe remove işlemi çağırılarak da kaldırılabilir. Eğer remove metodu yeni bir varlığa uygulanmış ise, remove işlemi gözardı edilir. Eğer remove işlemi bağlantısız bir varlığa uygulanmış ise işlev bir IllegalArgumentException aykırı durumu fırlatır yada hareket başarısızlıkla sonuçlanır. Eğer remove İşlemi zaten kaldırılmış bir varlığa uygulanmak istenirse de , bu da göz ardı edilir.

Varlığa ait veri, hareket tamamlanmışsa yada flush işlemi uygulanmış ise veri deposundan kaldırılır. Bu örnekte, Order.getLineItems ilişki belirtiminde cascade=ALL kümesine sahip olduğu için order ile ilişkili tüm LineItem varlıkları da kaldırılır.

```
public void removeOrder(Integer orderId) {
  try {
            Order order = em.find(Order.class, orderId);
            em.getTransaction().begin();
            // TODO:
            em.remove(li);
            em.getTransaction().commit();
  } catch(Exception e) {
           Logger.getLogger(getClass().getName()).log(Level.SEVERE,"exception
caught", e);
            em.getTransaction().rollback();
  } finally {
            em.close();
  }
 }
```

Varlığa ilişkin Verilerinin Veri Tabanında Senkronizasyonu

Kalıcı varlıkların durumu varlıkla ilişkili bir hareket yapıldığı zaman veri tabanına eş zamanlı işlenir. Eğer yönetilen varlık başka bir yönetilen varlıkla iki yönlü bağıntıya

sahipse, veri bağıntının sahiplenen tarafı taban alınarak kalıcılaştırılır. Yönetilen bir varlığın veri deposu ile senkronizasyonunu sağlamak için varlığa ait flush Metodu işletilir. Eğer varlıkla başka bir varlık arasında bağıntı varsa, ve bağıntı belirtimi PERSIST yada ALL için cascade eleman kümesine sahipse,ilişkili varlığa ait veri flush çağırıldında veri deposu ile senkronize
edilir. Eğer varlık kaldırılmışsa, flush ın çağırılması varlığa ait verinin veri deposundan kaldırılmasını sağlar.

Sorgu Oluşturma

EntityManager.createQuery ve EntityManager.createNamedQuery metodları Java Kalıcılık Sorgu dili sorgularıyla veri deposunu sorgulamak için kullanılır.
createQuery metodu dinamik sorgular oluşturmak için kullanılır. Dinamik sorgular uygulamanın iş mantığında direk tanımlanan sorgulardır. createNamedQuery metodu statik sorgular oluşturmak için kullanılır. Statik sorgular javax.persistence.NamedQuery açıklamaları kullanılarak meta veride tanımlanır. @NamedQuery'nin name elemanı createNamedQuery metodu ile kullanılacak metodun adını belirler. @NamedQuery'nin query elemanı sorgudur. Yaratılan NamedQuery varlık sınıfının @NamedQueries alanına da
eklenmelidir.

Örnek olarak:

```
@PersistenceContext
public EntityManager em;
...
customers = em.createNamedQuery("findAllCustomersWithName")
.setParameter("custName", "Smith")
.getResultList();
```

Kalıcılık Birimi (Persistence Unit) ve Ayarları

JPA ile ilgili bir diğer önemli yapı ise kalıcılık birimi'dir (persistence unit). Kalıcılık birimi, projenin JPA ayarları ile bu ayarları paylaşan kalıcı sınıfların oluşturduğu mantıksal yapının ismidir.Kalıcılık birimi, JPA projesinin kökünde olan `META-INF` klasöründeki `persistence.xml` dosyası içinde tanımlanır. `META-INF` klasörü ile `persistence.xml` dosyasının isimleri ve yerlerinin bu şekilde olması bir zorunludur, değişemez. Kalıcılık birimi istenirse program içinde de ilgili API kullanılarak programatik bir şekilde tanımlanabilir. Bunu nasıl yapabileceğimizi bu bölümde ileride göreceğiz. Kalıcılık birimi, projenin JPA ile yönetilecek olan kalıcılık yapısıyla ilgili ayarlarını içerir ve projenin kurulumunun da (deployment) parçasıdır.

persistence.xml Dosyası

```
<persistence>
   <persistence-unit name="OrderManagement">
       <description>This unit manages orders and customers.
            It does not rely on any vendor-specific features and can
            therefore be deployed to any persistence provider.
       </description>
       <jta-data-source>jdbc/MyOrderDB</jta-data-source>
       <jar-file>MyOrderApp.jar</jar-file>
       <class>com.widgets.Order</class>
       <class>com.widgets.Customer</class>
   </persistence-unit>
</persistence>
```

Bu dosya OrderManagement adında bir kalıcılık birmi tanımlar. Bu birim JTAaware jdbc/MyOrderDB veri kaynağını kullanır.

JAR dosyası ve sınıf elemanları yönetilen kalıcılık sınıflarını belirtir : varlık sınıfları, gömülebilir sınıflar(embeddable calsses), ve eşlenmiş atasınıflar(mapped superclass).

Sınıf elemanı yönetilen kalıcılık sınıflarını adlandırırken, JAR dosyası elemanı yönetilen kalıcılık sınıflarını içeren paketlenmiş kalıcılık birimi için anlamlı olan JAR dosyalarını belirler.

jta-data-source (for JTA-aware data sources) ve non-jta-data-source (non-JTA-aware data sources) elemanları ise kap(container) tarafından kullanılmak üzere veri kaynağının global JNDI adını belirler.

JAVA KALICILIK SORGU DİLİ

Java kalıcılık sorgu dili varlıklar ve onların kalıcılık durumları ile ilgili sorgular tanımlar. Sorgu dili belirtilen veri deposundan bağımsız olarak çalışabilien taşınabilir sorgular yazma imkanı verir. Sorgu dili veri modeli için varlıkların kendi ilişkilerini de içeren soyut kalıcılık şemalarını kullanırn ve bu veri modelini taban alan işleç ve ifadelerler tanımlar. Bir sorgunun faaliyet alanı aynı kalıcılık ünitesinde paketlenmiş ilgili varlıkların soyut şemalarını kapsar. Sorgu dili, nesneleri yada varlık soyut şema türleri ve birbirleri arasındaki ilişkileri taban alan değerleri seçmek için SQL benzeri bir dil kullanır. JPA kullanan tüm sorgular çokbiçimlidir (polymorphic). Yani bir sınıf sorgulandığı zaman bununla beraber sorgu kriterini karşılayan tüm alt sınıflar döndürülür. Java kalıcılık sorgu dili geliştirilmiş ve güçlendirilmiş bir sorgu dilidir.

Terimler Dizgesi

Aşağıda bu konuyla ilgili terimlerin açıklaması liste halinde verilmiştir.

• Soyut Şema(Abstract schema): Sorguların üzerinde işlem yaptığı kalıcılık şema soyutlaması (kalıcılık varlıkları, onların durumları ve ilişkileri). Sorgu dili bu kalıcı şema soyutlaması üzerinden sorguları, varlıkların eşlendiği veri tabanı şeması üzerinde çalıştırılan sorgulara dönüştürür.

• Soyut Şema Türü(Abstract schema type): Tüm ifadeler belli bir türü ilerler. Bir varlığın soyut şema türü varlık sınıfı ve Java dili gösterimlerince sağlanan metadata bilgisinden türetilir.

• Backus-Naur Form (BNF): Üstdüzey dillerin sözdizimlerini tanımlayan bir gösterimdir. Bu bölümde açıklana sözdizim çizenekleri BNF gösterimleridir.

• Yol İfedesi(path expression): Bir varlığın durum yada ilişkisinin yönünü belirten ifade .

• Durum Alanı(State field): Bir varlığın kalıcı alanı

• İlişki Alanı(Relationship field): Bir varlığın, türü ilgili varlığın soyut şema türü olan kalıcı ilişki alanı

Söz dizim

Bu bölüm Java kalıcılık belirtimlerinde tanımlanan sorgu dili sözdizimini örneklerle açıklamaktadır.

FROM Deyimi

From deyimi sorgunun etki alanını tanıtma değişkenleri (identification variables) bildirerek belirler.

Tanıtıcılar (Identifiers)

Bir tanıtıcı bir dizi damganın peşpeşe gelmesiyle oluşur. İlk damga, Java programlama dilinin bir tanıtıcısındaki geçerli bir ilk damga olmalıdır (harf,$,_). Bunun ardından gelecek olan her damga bir Java tanıtıcısında geçerli ve ilk damga olmayan bir damga olmalıdır (harf,$,_).(?) damgasını sorgu diline ayrılmış bir karakter olduğundan tanıtıcı olarak kullanılamaz. Bir sorgu dili tanıtıcısı iki istisna hariç büyük harf küçük harf duyarlıdır(case-sensitive).

• anahtar kelimeler

• tanıtma değişkenleri

Bir tanıtıcı, bir sorgu dili anahtar kelimesiyle aynı olamaz.SQL keyword listesi:

ALL	BETWEEN	DELETE	FETCH	IS
AND	BY	DESC	FROM	JOIN
ANY	COUNT	DISTINCT	GROUP	LEFT
AS	CURRENT_DATE	EMPTY	HAVING	LIKE
ASC	CURRENT_TIME	EXISTS	IN	MAX
AVG	CURRENT_TIME STAMP	FALSE	INNER	MEMBER
MIN	NEW	NULL	OF	OR
MOD	NOT	OBJECT	OUTER	ORDER
SELECT	SUM	TRIM	TRUE	WHERE
SOME	UNKNOWN	UPDATE	UPPER	

Tanıtma Değişkenleri(Identification Variables)

Bir tanıtma değişkeni FROM deyiminde bildirilen bir tanıtıcıdır. SELECT ve WHERE deyimleri tanıtma değişkenlerini referans gösterebilirler ancak onları bildiremezler. Bütün tanıtma değişkenleri FROM deyinde bildirilmelidir. Tanıtma değişkeni de bir tanıtıcı olduğu için tanıtıcı için geçerli kurallara uymak zorundadır. Bununla beraber verilen kalıcılık ünitesi içinde tanıtma değişkeni herhangi bir varlık yada soyut şemaya eşleşmemelidir. FROM deyimi virgüllerle ayrılmış çoklu bildirimler içerebilir. Bir bildirim daha önce bildirilmiş bir tanıtma değişkenini referns gösterebilir. Aşağıdaki deyimde t değişkeni daha önce bildirilmiş p değişkenini referans göstremektedir:

```
FROM Player p, IN (p.teams) AS
```

Aşağıdaki sorgu bir takıma ait olup olmadığına bakılmaksızın tüm oyuncuları (player) döndürür.

```
SELECT p
FROM Player p
```

Bu örneğin aksine aşağıdaki sorgu tanıtma değişkeni kullanır ve sadece bir takıma ait olan oyuncuları döndürür:

```
SELECT p
FROM Player p, IN (p.teams) AS t
```

Takip eden sorgu bir öncekiyle aynı sonucu döndürür ancak WHERE deyimi okunurluğu arttırmaktadır:

```
SELECT p
```

```
FROM Player p
WHERE p.teams IS NOT EMPTY
```

Alan Değişkeni Bildirimleri

Bir tanıtma değişkenini bir soyut şema türü olarak bildirmek için bir alan değişkeni bildirimi

belirtilmelidir. Diğer bir deyişle bir tanıtma değişkeni, bir varlığın soyut şeması üzerinde değişim gösterir.Aşağıdaki örnekte tanıtma değişkeni p, Player adlı soyut şemayı temsil eder:

```
FROM Player p
```

Bir aralık değişkeni isteğe bağlı olarak AS de içerebilir.

```
FROM Player AS p
```

Eğer sorgu aynı soyut şema üzerinde bider çok değeri karşılaştırıyorsa FROM deyimi bu soyut şema için birden çok tanıtma değişkeni bildirmelidir.

```
FROM Player p1, Player p2
```

Koleksiyon Üyesi Bildirimleri

Bire- birçok ilişkide çoklu taraf bir varlık koleksiyonu içerir. Tanıtma değişkeni bu koleksiyonun bir üyesini temsil eder. Bir koleksiyon üyesi bildirimi IN işleci içermek zorundadır ama AS işlecini atabilir.

Aşağıdaki örnekte Player adlı soyut şemayla temsil dilen varlık teams adlı bir ilişki alanına sahiptir. Tanıtma değişkeni t, teams koleksiyonunun tek bir üyesini temsil eder :

```
FROM Player p, IN (p.teams) t
```

Join

Join operatoru varlıklar araındaki ilişkiler arasında dolanmak için kullanılır ve fonksiyonle olarak IN işlecine benzer. Aşağıdakide örnekte sorgu, müşteriler (customers) ve siparişler (orders) arasındaki ilişki üzerinde JOIN işlecini kullanır:

```
SELECT c
FROM Customer c JOIN c.orders o
WHERE c.status = 1 AND o.totalPrice > 10000
The INNER keyword is optional:
SELECT c
FROM Customer c INNER JOIN c.orders o
WHERE c.status = 1 AND o.totalPrice > 10000
```

Aynı örnek IN işleciyle yapılmak istenirse:

```
SELECT c FROM Customer c, IN(c.orders) o WHERE c.status = 1
```

Tek değerli bir ilişkide de JOIN işleci kullanılabilir

```
SELECT t FROM Team t JOIN t.league l WHERE l.sport = :sport
```

LEFT JOIN yada LEFT OUTER JOIN , join koşulu içinde boş olabilecek değerlerle eşleşen varlık kümesini döndürür. OUTER anahtar kelimesi isteğe bağlıdır:

```
SELECT c.name, o.totalPrice FROM Order o LEFT JOIN o.customer c
```

WHERE Deyimi

Where deyimi sorgu tarafından döndürülecek değerleri sınırlayan koşul ifadeini belirler. Sorguda WHERE kullanımı isteğe bağlıdır; kullanıldığı durumda sadece koşulu sağlayan değerler aksi halde tüm değerler döner.

Girdi Parametreleri

Bir girdi parametresi adlandırılmış(named) veya konumsal (positional) parametre olabilir.

Adlandırılmış girdi parametresi biz dizgi tarafından takip edilen bir iki nokta üstüste ile ifade edilir. Örneğin, :isim.

Konumsal input parametresi ise ardından bir tamsayı gelen bir soru işareti ile ifade edilir.Örneğin ilk

parametre ?1, ikinci ?2 gibi.

Girdi parametreleri için uyulması gereken kurallar :

• Sadece WHERE yada HAVING deyimi içinde kullanılbilirler..

• Konumsal parametreler 1 den başlayarak numaralandırılmalıdır.

• Adlandırılmış ve konumsl parametreler tek bir sorgu içinde bulunmamalıdır

• Adlandırılmış parametreler büyük harf –küçük harf duyarlıdır.

Koşullu İfadeler (Conditional Expression)

Where deyimi, soldan sağa öncelik sırasına göre işlettilen bir koşullu ifade içerir. İşletim sırasını parantezler kullanarak değiştirmek mümkündür.

BETWEEN İfadesi

BETWEEN ifadesi aritmetik ifadelerde değer aralığına karar verir. Aşağıdaki ikili ifadeler birbirleriyle eşdeğerdir.

```
p.age BETWEEN 15 AND 19
p.age >= 15 AND p.age <= 19
p.age NOT BETWEEN 15 AND 19
p.age < 15 OR p.age > 19
```

Aritmetik ifade NULL değere sahipse BETWEEN'den dönecek sonuç belirsizdir.

IN İfadesi

IN ifadesi bir dizginin belli bir dizgi kümesine yada bir tamsayının belli bir tamsayı kümesine ait olup olmadığına karar verir.

Sorgu Dili Öncelik Sırası Tablosu:

1	=	9	[NOT] IN
2	>	10	IS [NOT] NULL
3	>=	11	IS [NOT] EMPTY
4	<	12	[NOT] MEMBER OF
5	<=	13	Logical NOT
6	<> (not equal)	14	AND
7	[NOT] BETWEEN	15	OR
8	[NOT] LIKE		

Verilen yol ifadesi bir dizgi yada syaısal bir ifade olmalı, NULL değere sahip olursa IN ifadesinden dönecek değer belirsizdir. Aşağıdaki örnekte ülke adını verilen alanda arama yapılmaktadır eğer bu listede varsa TRUE aksi halde FALSE döner.

```
o.country IN ('JAPAN', 'USA', 'France')
```

LIKE İfadesi

LIKE ifadesi yerine herhangi bir dizgi gelebilecek örüntüleri ilgili dizgilere eşler. Bu pattern de (_) damgası herhangi bir damgayı temsil eder, (%) damgası ise bir veya birden çok karakteri temsil eder.

ESCAPE ifadesi örüntüdeki bahsedilen bu damgalar için bi çıkış damgası belirler. LIKE için örnekler:

```
address.phone LIKE '12%3'
'123'
'12993'
'1234'
asentence.word LIKE 'l_se'
'lose' 'loose'
```

NULL Karşılaştırma İfadesi

Null karşılaştırma ifadesi tek değerli bir ifade yada bir girdi değerinin null olup olmadığını kontrol eder. Bu sorgu lig(league) ilişkisi kurulmamış tüm takımları (team) seçer.

```
SELECT t FROM Team t WHERE t.league IS NULL
```

Bu sorgu yukardakine eşdeğer değildir.(=) işleci kullanılarak çalıştırılan NULL ifadesi ilişki kurulu olmasada herzaman bilinmeyen bir değer döndürür. Bu sorgu herzaman boş bir sonuç döndürür.

```
SELECT t FROM Team t WHERE t.league = NULL
```

Empty Koleksiyon Karşılaştırma İfadesi

IS [NOT] EMPTY karşılaştırma ifadesi verilen koleksiyon değerli bir ifdenin boş olup olmadığını test eder.Başka bie deyişle koleksiyon değerli ilişkinin kurulup kurulmadığını sınar. Eğer sınanan değer NULLsa ifadeden de NULL döner. Aşağıdaki örnek hiç line item içermeyen order'ları döndürür:

```
SELECT o FROM Order o WHERE o.lineItems IS EMPTY
```

Koleksiyon Üyesi İfadeleri

[NOT] MEMBER [OF] koleksiyon üyesi ifadeler bir değerin bir koleksiyonun üyesi olup olmadığını sorgular. Değer ve koleksiyon üyeleri aynı türde olmalıdırlar. OF anahtar kelimesi isteğe bağlıdır.

Aşağıdaki örnek line item'ın, order'ın bir parçası olup olmadığını sınar:

```
SELECT o FROM Order o WHERE :lineItem MEMBER OF o.lineItems
```

Alt Sorgular

Alt sorgular; sorgunun WHERE ve HAVING deyimlerinde parantezlerle çevrelenmiş olarak

kullanılabilir. Aşağıdaki örnek ondan fazla sipariş(order) veren müşterileri(customer) bulmaktadır. Aşağıdaki örnek 10 dan fazla sipariş veren müşterileri bulur:

```
SELECT c FROM Customer c WHERE (SELECT COUNT(o) FROM c.orders o) > 10
```

Exists İfadesi

[NOT] EXISTS ifadesi bi alt sorguyla kurulan bağda kullanılır ve sadece alt sorgudan dönen sonuç bir yada daha çok değer içeriyorsa doğru (true) aksi halde yanlıştır(false).Aşağıdaki örnek eşi de çalışan(employee) olan çalışanları bulmaktadır:

```
SELECT DISTINCT emp FROM Employee emp
WHERE EXISTS (
SELECT spouseEmp FROM Employee spouseEmp WHERE spouseEmp =
emp.spouse)
```

All ve Any İfadeleri

ALL ifadesi altsorguyla bağlantılı kullanılır ve eğer alt sorgudan dönen tüm değerler doğruysa yada altsorgu boşsa doğrudur. ANY ifadesi de aynı şekilde kullanılır ve altsorgudan dönen değerlerin bazıları doğruysa ifade doğrudur. Eğer altsorgu sonucu boşsa yada dönen bütün değerler yanlışsa ifade yanlıştır. SOME anahtar kelimesi ANY ile eş anlamlıdır.

ALL ve ANY ifadeleri =, <, <=, >, >=, <> bu karşılaştırma işleçleriyle birlikte kullanılır. Aşağıdaki örnekte çalışanın departmanındaki yöneticiden(manager) daha yüksek maaş(salary) alan çalışanları(employee) bulur.

```
SELECT emp FROM Employee emp
WHERE emp.salary > ALL ( SELECT m.salary
FROM Manager m WHERE m.department = emp.department)
```

SELECT Deyimi

SELECT deyimi nesnelerin türlerini ya da sorgudan dönen değerlerini tanımlar.

Dönüş Türleri

SELECT deyiminin dönüş türü select ifadelerinin içindeki dönüş türleri ile tanımlanır. Eğer çoklu ifadeler kullanılıyorsa,sorgunun sonucu bir Object dizisidir (Object[]) ve dizideki elemanlar SELECT deyimindeki ifadelerin sırasına ve tür açısından da her ifadenin dönüş türüne uyar.Bir SELECT deyimi koleksiyon değerli(collection-valued) ifadeleri belirtemez. Örneğin, SELECT deyimp.teams geçersizdir, çünkü teams bir koleksiyondur. Oysa ki aşagıdaki sorgudali deyim geçerlidir çünkü t , teams kolaksiyonunun bir elemanıdır:

```
SELECT t FROM Player p, IN (p.teams) t
```

Aşağıdaki sorgu örneğinde ise select deyiminde çoklu ifadeler kullanılmıştır:

```
SELECT c.name, c.country.name FROM customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

Bu sorgu Object[] elemanlarının bir listesini döndürür. İlk dizi elemanı müşteri adının belirtildiği bir dizgidir, ikinci dizi elemanı ise müsterinin ülkesinin adının belirtildiği bir dizgidir

SELECT Deyiminde Tümleşik Fonksiyonlar (Aggregate Functions)

Bir sorgunun sonucu tümleşik bir fonksiyonun sonucu da olabilir.

Name Return Type Description

AVG Double Returns the mean average of the fields.

COUNT Long Returns the total number of results.

MAX the type of the field Returns the highest value in the result set.

MIN the type of the field Returns the lowest value in the result set.

SUM

Long (for integral fields)

Double (for floating point fields)

BigInteger (for BigInteger fields)

BigDecimal (for BigDecimal fields)

Returns the sum of all the values in the result set.

SELECT deyiminde tümleşik fonksiyonlar (AVG, COUNT, MAX, MIN, or SUM) su kurallar

cercevesinde kullanılır:

• AVG, MAX, MIN, ve SUM fonksiyonları için eğer fonksiyonun uygulanacağı hiç bir değer yoksa fonksiyonlar null döndürür.

• COUNT fonksiyonu için, eğer fonksiyonun uygulanacağı hiç bir değer yoksa COUNT fonksiyonu 0 döndürür.

Aşağıdaki örnek sipariş(order) miktarının ortalamasını döndürür.

```
SELECT AVG(o.quantity) FROM Order o
```

Aşagıdaki örnek Roxane Coss tarafından sipariş edilen ürünlerin toplam miktarını döndürür:

```
SELECT SUM(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

Aşagıdaki örnek toplam sipariş sayısını döndürür:

```
SELECT COUNT(o) FROM Order o
```

Aşagıdaki örnek Hal Incandenza ‘ nın siparisinde fiyatı olan ürünlerin toplam sayısını döndürür:

```
SELECT COUNT(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Incandenza' AND c.firstname = 'Hal'
```

DISTINCT Anahtar Kelimesi

DISTINCT anahtar kelimesi aynı dönen değerleri eler. Eğer bir sorgu aynı dönüş değerlerine izin veren java.util.Collection döndürüyorsa, sorguda DISTINCT anahtar kelimesi kullanılarak aynı değerler elenmelidir.

Yapılandırıcı İfadeleri(Constructor Expressions)

Yapılandırıcı ifadeleri , sorgunun sonucu olarak Object[] dizisi döndürmek yerine , java örnekleri(java instances) döndürmeye olanak sağlar.

Aşağıdaki sorgu WHERE deyimi ile eşleşen her Customer için bir CustomerDetail örneği yaratır. CustomerDetail müşteri adını ve müşterinin ülkesinin adını tutar. Dolayısıyla sorgu CustomerDetail örneklerinin bir listesini döndürür :

```
SELECT NEW com.xyz.CustomerDetail(c.name, c.country.name)
FROM customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

ORDER BY Deyimi

Adından da anlaşılacağı üzere ORDER BY deyimi sorgudan dönen değerleri ya da nesneleri sıralar. Eğer ORDER BY deyimi çoklu elemanlar içeriyorsa, elemanların soldan sağa dizisi yüksek öncelikliden düşük öncelikliye doğrudur.

ASC anahtar kelimesi artan sırada sıralamayı(varsayılan) ifade eder, DESC anahtar kelimesi azalan sırada sıralama yapılacak analamına gelir.

ORDER BY deyimi kullanılırken, SELECT deyimi sıralanabilir bir nesneler ya da değerler kümesidöndürmelidir. SELECT deyiminden dönmeyen değer yada nesneleri sıralayamazsınız. Örneğin, aşağıdaki sorgu geçerlidir çünkü ORDER BY deyimi SELECT deyiminden dönen nesneleri kullanır:

```
SELECT o
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
```

```
ORDER BY o.quantity, o.totalcost
```

Ancak aşağıdaki örnek geçerli değildir çünkü ORDER BY deyimi SELECT deyiminden dönmeyen bir değeri kullanır:

```
SELECT p.product_name
FROM Order o, IN(o.lineItems) l JOIN o.customer c
WHERE c.lastname = 'Faehmel' AND c.firstname = 'Robert'
ORDER BY o.quantity
```

GROUP BY Deyimi

GROUP BY deyimi bir özellikler kümesine göre değerleri gruplamaya olanak verir: Aşağıdaki sorgu müşterileri ülkelerine göre gruplar ve her ülkedeki müşteri sayısını döndürür:

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

HAVING Deyimi

HAVING deyimi GROUP BY deyimi ile birlikte sorgudan dönen sonucu daha da sınırlamak için kullanılır.

Aşağıdaki sorgu siparişleri müşterilerinin statüsüne göre gruplar, müşteri statüsünü ve aynı statüdeki müştelerin bütün siparişlerinin (totalPrice) ortalamasını döndürür. Buna ek olarak statüsü yalnız 1,2 ya da 3 olan müşteri dikate alır, dolayısıyla diğer müşterilerin siparişleri hesaba alınmaz:

```
SELECT c.status, AVG(o.totalPrice)
FROM Order o JOIN o.customer c GROUP BY c.status HAVING c.status IN (1, 2, 3)
```

Kaynakça

http://serkansakinmaz.blogspot.com.tr/2013/10/orm-nedir-object-relational-mapping.html

http://safakunel.blogspot.com.tr/2009/12/java-jpa-java-persistence-api-nedir.html

http://www.yunusmete.com/2014/01/23/java-persistence-api-jpa-nedir/

http://www.aliboyraz.com/jpa-nedir-1/

http://www.javaturk.org/?p=1119

http://www.javaturk.org/?tag=jpa

http://java.sun.com/developer/technicalArticles/J2EE/jpa

http://belgeler.cs.hacettepe.edu.tr/yayinlar/eski/JPA.pdf

http://www.intertech.com/resource/usergroup/Exploring%20the%20JPA.pdf

http://www.objectdb.com/api/java/jpa