Java Platform Enterprise Edition

JTP: Ing. Felipe Steffolani
fsteffolani@sistemas.frc.utn.edu.ar

Capa de Acceso a Datos

- Objetivos
  - Transparencia
  - Flexibilidad
  - Independencia
- Diferentes opciones
  - Escribir el código
  - Genera el código
  - Herramienta ORM

Escribir el código

- Es la opción más básica pero más flexible
- Podemos volcar allí todas las particularidades que necesitemos
- En la mayoría de los sistemas al menos es necesario en una porción de la DAL
- Implementamos un patrón
- El patrón con mayor cantidad de implementaciones es el patrón DAO

Ventajas y Desventajas

- Ventajas
  - Flexibilidad
  - Conocimiento del código
  - Libertad de acción ante casos particulares
- Desventajas
  - Código muy tedioso y repetitivo
  - Natural generación de errores
  - Tiempo de desarrollo

Generadores de código

- Es muy común utilizar generadores de código para evitar programar la capa de acceso a datos
- Estos trabajan en base a un patrón
- El código generado es Datacéntrico
- Existen muchos productos la mayoría pagos y algunos gratuitos
- Además existen productos genéricos basados en templates

Capa Generada DAO

- En general la capa generada implementa el patrón DAO
- Algunos generadores nos permiten elegir el nivel de abstración
- Otros genera directamente el máximo nivel de abstracción

Ventajas y Desventajas

- Ventajas
  - Evitamos escribir el código a mano
  - Menor inserción de errores en la capa
  - Dependiendo del patrón, gran transparencia.
- Desventajas
  - En general código no manejable
  - Tratamiento especial de casos particulares
  - Dependencia de librerías

Mapeo Objeto – Relacional

- Herramienta más conocida tradicionalmente Hibernate, otras Eclipselink, Toplink
- Delegamos la responsabilidad de la capa en un framework
- Podemos manejarnos en un plano de Objetos
- El framework se encarga de persistir y recuperar datos
- Las relaciones se almacenan en archivos de configuración XML

Framework de Persistencia de Objetos de Java

JPA – JAVA PERSISTANCE API

¿Qué es JPA?

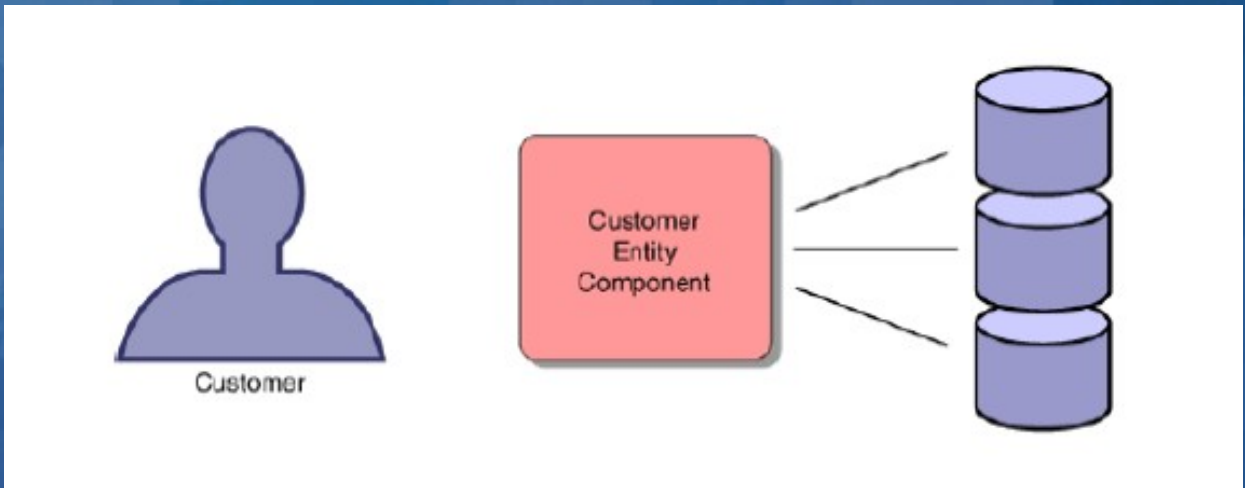
- JPA por Java Persistance API
  - `javax.persistance.*`
- Básicamente es una especificación en lenguaje java que implementa los mecanismos para acceder a datos en una base de datos relacional.
- Fue incluida inicialmente como parte de la especificación de EJB3.0 , donde reemplaza al EJB2.0 – CMP.

La API de Persistencia de Java.

- Reemplaza los EJBs llamados EntityBeans con clases Entidad que no son EJBs.
- Es una API standard para especificar información de mapeo de Objetos a Tablas Relacionales.
- Se puede usar con o sin un Application Server de Java EE.
- Hay dos tipos de persistencia:
  - Container-Managed Persistence
  - Application-Managed Persistence

Software de Mapeo Objeto-Relacional (ORM).

- Proporciona la conversión automática de tablas relacionales a objetos en memoria.



Java Persistence API y ORM.

- La API de Persistencia de Java especifica las características de tecnologías existentes de ORM en la tecnología Java EE.
- Es solamente una especificación, no un software ORM utilizable.
- Se requiere una implementación particular.
- Sun Application Server (GlassFish) utiliza Toplink Essentials (Implementación de Referencia) como default.

JPA en 3 palabras

- Entities
  - POJOs: No hacen falta interfaces ni clases base
  - Constructor sin argumentos
  - Implementación Serializable (opcional)
- Configuración
  - /META-INF/persistence.xml
    - JTA o RESOURCE_LOCAL
    - Utilización de un DataSource remoto o un Pool de Conecciones
  - Mapeos: usando Anotaciones o XML
- API
  - EntityManager y EntityManagerFactory
  - Consultas con JP QL o Native SQL
  - EntityTransaction
  - @PersistenceContext, @PersistenceUnit

¿Qué es una entidad?
Un poco de historia

- Plain Old Java Object(POJO)
  - Se crean por medio de la palabra reservada new
- No se requieren interfaces
- Tiene una identidad de persistencia
- Puede tener estados tanto persistentes como no persistentes
  - Tipos Simples (ej. primitivos, enums ..)
  - Estados no persistentes (transient o @Transient)
- Permite herencia entre entidades
- *Serializable*; usable como objetos separados en otras capas
  - No necesitan objetos de transferencia de datos

- Entity == Plain Old Java Object (POJO) + Información de Mapeo
  - Especificada por @Entity o a través de configuración XML
  - Clave Primaria: @Id o @IdClass (composite PK)
- Información de Mapeo o Metadata
  - Anotaciones
  - ORM XML

Ejemplo de Entidad de Persistencia

```
@Entity
public class Customer {
    private Long id;
    private String name;
    private Address address;
    private Collection<Order> orders = new HashSet();
    public Customer() {}

    @Id public Long getID() {
        return id;
    }

    protected void setID (Long id) {
        this.id = id;
     }
...
```

Annotation ✉ denota que es Entidad

@Id denota la clave primaria

Métodos accesores para acceder el estado

Usando ORM XML

```
<entity
   class="model.Employee">
   <attributes>
      <id name="id">
         <column name="EMP_ID"/>
      </id>
   </attributes>
</entity>
```

- Nota: La configuración XML personaliza e incluso pisa la configuración mediante Anotaciones.

Declaración de las Clases Entidad.

- Se codifican como clases standard de Java con las siguientes características:
  - Marcadas con la anotación javax.persistence.Entity en la clase o declaradas como "Entities" en un Deployment Descriptor.
  - Deben ser clases públicas.
  - Tener un constructor sin argumentos con acceso public o protected.
  - Que se puedan extender, es decir, que no sean finales.
  - Que implementen la interfaz serializable si van a ser entregadas vía return por EJBs de sesión remotos.
  - No pueden ser clases internas (Inner Classes).

Mapeo de Default.

Elemento en Memoria	Elemento en Base de Datos
Clase Entidad	Tabla
Atributo en Clase Entidad	Columna de la Tabla
Instancia de la Clase Entidad	Fila de la Tabla

- El nombre de la tabla es por default igual a la de la Clase Entidad.
- Se puede cambiar con la anotación @Table.
  - @Entity
  - @Table(name = "Customer")
  - public class Cliente { L }

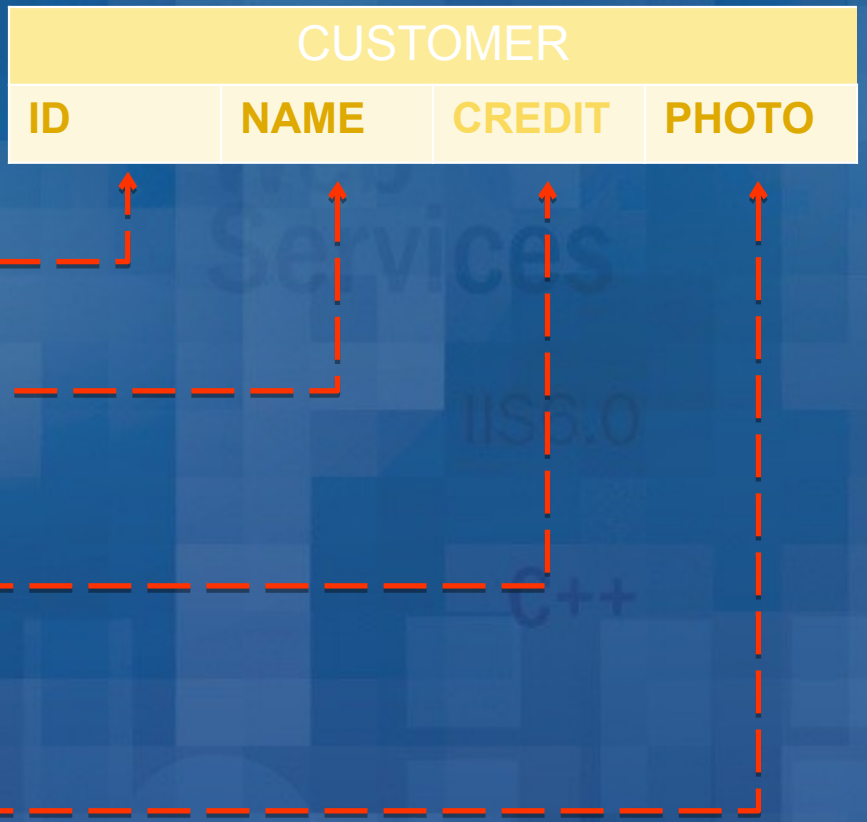
Mapeos Simples

```
@Entity(access=FIELD)
public class Customer {
        @Id
        int id;

        String name;

        @Column(name="CREDIT")
        int c_rating;

        @Lob
        Image photo;
}
```



Campos Persistentes vs. Propiedades Persistentes.

- El estado de las Clases Entidad se sincroniza con la base de datos.
- El estado de una Clase Entidad se obtiene de sus variables de instancia (campos) o de sus propiedades (variables más métodos getters y setters).
- La obtención del estado se determina por la colocación de las anotaciones.
- No se pueden tener ambos casos.

Obtención de Estado vía Campos Persistentes.

Cuando el acceso es vía Campos Persistentes, el contenedor obtiene el estado leyendo directamente las variables de la clase.

- Las variables no pueden ser públicas.
- No deben ser leídas directamente por el cliente.
- Todas las variables son "persistidas" a menos que se declaren como transient de java o se les ponga la anotación @Transient.
- Por ejemplo:

```
@Id @Column(name = "ID") private int id;
@Column(name = "MSG") private String message;
public int getId() { return id; }
public void setId(int id) { this.id = id; }
public String getMessage() { return message; }
public void setMessage(String message) { this.message = message; }
```

Obtención de Estado vía Propiedades Persistentes.

Cuando el acceso es vía Propidades Persistentes, el contenedor obtiene el estado usando los métodos getter.

- Los métodos deben ser públicos o portected.
- Deben seguir las convenciones de nombres de los Java Beans.
- Las anotaciones deben ir en los métodos getters.
- Por ejemplo:

```
private int id;
private String message;
@Id @Column(name = "ID") public int getId() { return id; }
public void setId(int id) { this.id = id; }
@Column(name = "MSG")
public String getMessage() { return message; }
public void setMessage(String message) { this.message =
    message; }
```

Tipos de Datos Persistentes.

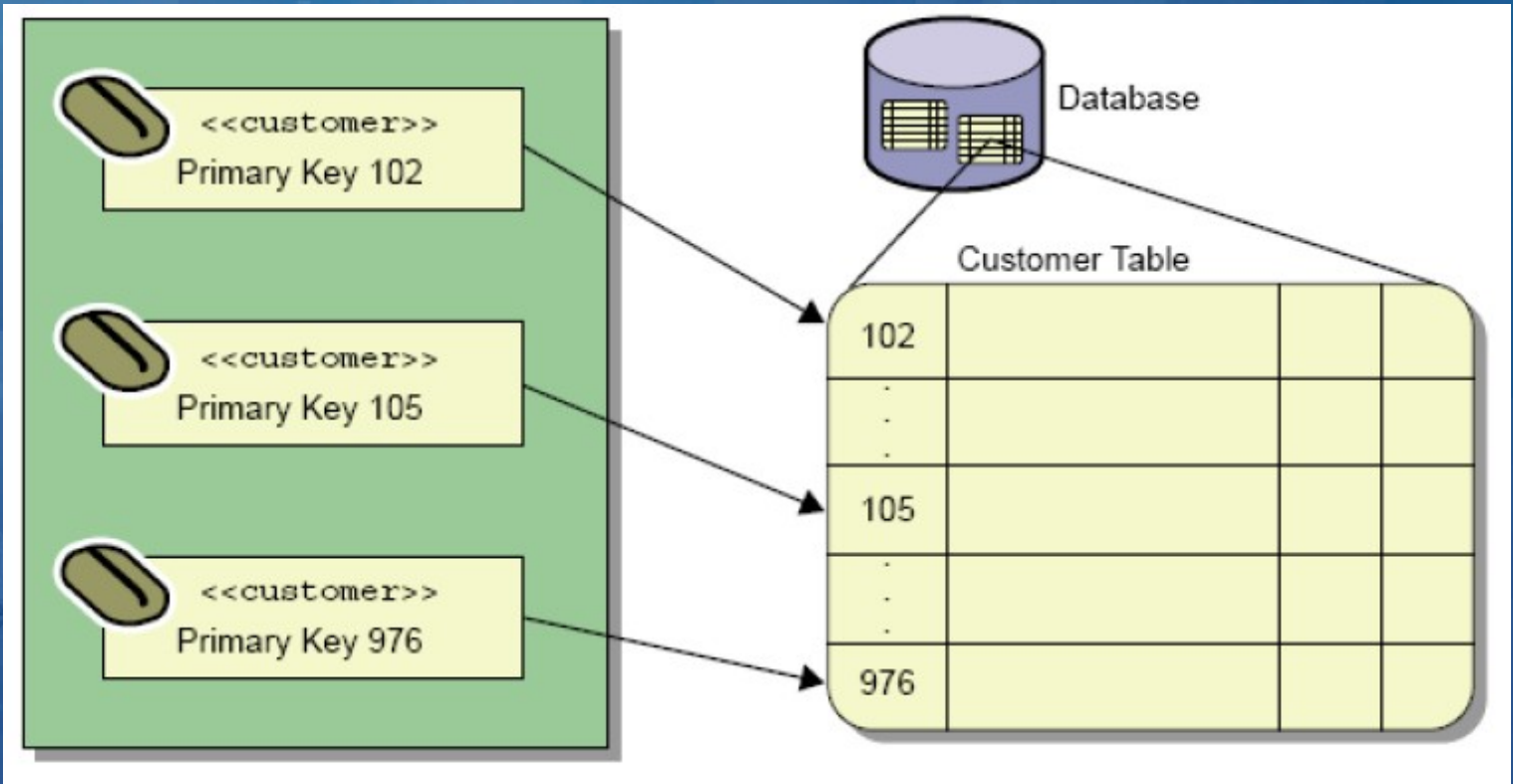
- Primitivos de Java.
- Objetos de clases Wrapper de Java.
- java.lang.String
- byte[ ] y Byte[ ]
- char[ ] y Character[ ]
- Cualquier tipo serializable como:
  - java.util.Date
  - java.sql.Date
  - java.sql.TimeStamp

Concepto de Clave Primaria.

- Cada objeto de Clase Entidad se distingue por tener una clave primaria.
- La clave primaria proporciona identidad a la Clase Entidad.
- Típicamente son Strings o enteros, pero pueden también ser objetos de otras clases que corresponden a varias columnas de la tabla que representan.
- Cada Clase Entidad debe tener una clave primaria.
- Se marcan con la anotación @Id.
- Pueden ser autoincrementables.
- Por ejemplo:

```
@Id @GeneratedValue(strategy = GenerationType.AUTO)
@Column(name = "ID", nullable = false)
private int id;
```

Relación entre Entidades y Filas de Tabla.



Mapeos Simples

- Se mapea directamente el nombre del atributo al de la columna de la tabla mediante @Basic
- Este mapeo por defecto es inferido para los atributos no mapeados.
- Puede se usado junto con @Column (mapeo físico)
- Puede ser especializado por:
  - @Lob
  - @Enumeration
  - @Temporal

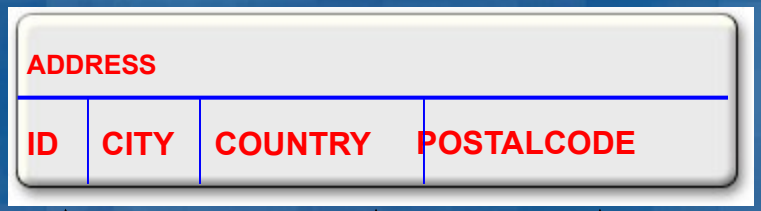
Mapeos por defecto

```
@Entity
public class Address {
    @Id
    private Long id;

    private String city;

    private String country;

    private String postalCode;
```



Redefiniendo Mapeos

```
@Entity
public class Address {
    @Id
    private Long id;

    private String city;

    private String country;
    @Column(name="P_CODE")
    private String postalCode;
```

Mepeo de Relaciones

- Mapeos soportados
  - @ManyToOne, @OneToOne – una entidad
  - @OneToMany, @ManyToMany – coleccion de entidades
- Unidireccional o bidireccional
- Especificación de la relación de conocimiento
  - @JoinColumn columna clave
  - @JoinTable para muchos a muchos

Mapeo muchos a uno

```
@Entity
public class Customer {
    @Id
    int id;
    @ManyToOne
    Address addr;
}
```

Mapeo uno a muchos

```
@Entity
public class Customer {
   @Id
   int id;
   ...

   @OneToMany(mappedBy="cust")
   Set<Order> orders;
}
```

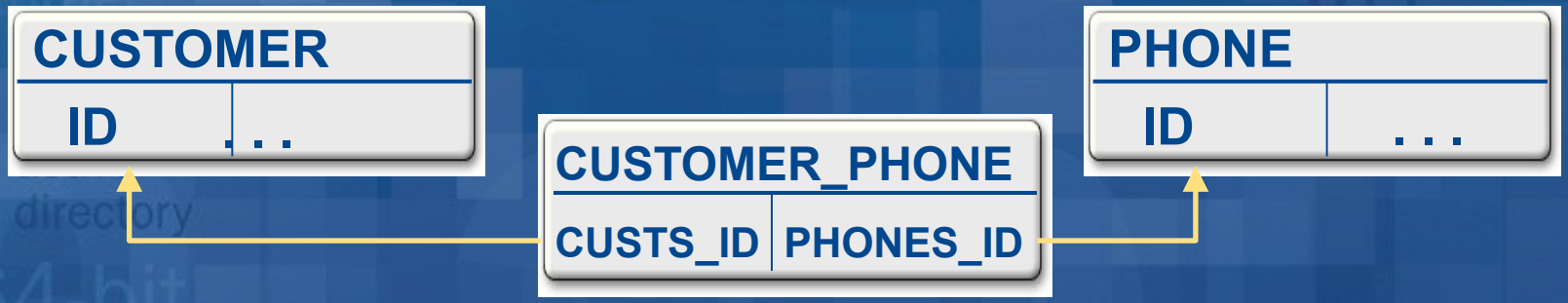
```
@Entity
public class Order {
   @Id
   int id;
   ...

   @ManyToOne
   Customer cust;
}
```

Mapeo Muchos a muchos

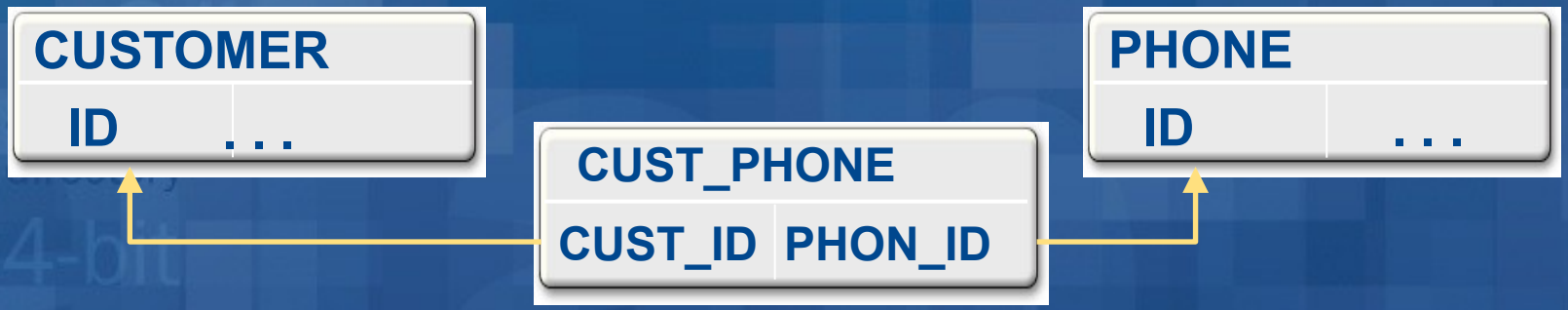
```
@Entity
public class Customer {
  @Id
  int id;
  ...
  @ManyToMany
  Collection<Phone> phones;
}
```

```
@Entity
public class Phone {
  @Id
  int id;
  ...
 @ManyToMany(mappedBy="phones")
  Collection<Customer> custs;
}
```



Mapeo Muchos a Muchos

```
@Entity
public class Customer {
   ...
   @ManyToMany
   @JoinTable(table=@Table(name="CUST_PHONE"),
     joinColumns=@JoinColumn(name="CUST_ID"),
     inverseJoinColumns=@JoinColumn(name="PHON_ID"))
   Collection<Phone> phones;
}
```



Herencia

- Las entidades pueden heredar de:
  - Otras entidades
  - Clases comunes
- La herencia se puede mapear en tres formas
  - Single table — Todas las clases a una tabla
  - Joined — Cada clase almacenada en una tabla diferente
  - Table per concrete class — Cada clase concreta es mapeada a una tabla

Modelo

```
public abstract class Animal {
   int id;
   String name;
```

```
public class LandAnimal extends Animal {
   int legCount;
}
```

```
public class AirAnimal extends Animal {
   short wingSpan;
}
```

Mapeos

- **Single table:**

- **Joined:**

- **Table per Class:**

Otros Conceptos de la API de Persistencia.

- Unidad de Persistencia (Persistence Unit).
- Administrador de Entidad (Entity Manager).
- Contexto de Persistencia (Persistence Context).
- Identidad de Persistencia (Persistence Identity).

  Es la clave primaria.

Persistence Unit.

- Una Unidad de Persistencia en una colección de Clases Entidad representadas por un archivo llamado persistence.xml contenido dentro de un archivo EAR, JAR o WAR de una aplicación Java EE.
- Una Unidad de Persistencia define que Clases Entidad son administradas por un Entity Manager.
- Está limitada a un solo DataSource.

El Archivo persistence.xml.

- Configura que clases constituyen una Unidad de Persistencia.
- Define la base de la Unidad de Persistencia, es decir el archivo al que pertenece.
- Especifica el Data Source utilizado.
- Ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence">
<persistence-unit name="BrokerTool-ejb" transaction-
    type="JTA">
<jta-data-source>StockMarket</jta-data-source>
<jar-file>BrokerLibrary.jar</jar-file>
<properties/>
</persistence-unit>
</persistence>
```

El Persistence Context.

- Se interpreta como una copia activa de una Unidad de Persistencia.
- Se pueden tener varios Contextos de Persistencia que usan la misma Unidad de Persistencia activos al mismo tiempo.
- El Contexto de Persistencia:
  - Típicamente dura lo mismo que una transacción.
  - Limita las instancias de Entidades a una sola instancia por Identidad de Persistencia.
  - Tiene una API para su manejo, conocida como el Entity Manager.

El Entity Manager.

- Proporciona los métodos para controlar eventos en un Contexto de Persistencia y el ciclo de vida de las instancias de Clases Entidad en un Contexto de Persistencia.
- El Entity Manager:
  - Proporciona operaciones como flush(), find() y createQuery() para controlar un Contexo de Persistencia.
  - Reemplaza la funcionalidad de las interfaces home de los EJBs de entidad anteriores.
  - Se obtiene usando anotaciones en clases manejadas por el contenedor:

    @PersistenceContext private EntityManager em;

Operaciones con entidades

- EntityManager (javax.persistence)
  - persist()- inserta una instancia de la entidad en la bd
  - remove()- borra una instancia de la bd
  - refresh()- recarga los datos de la db para una ent.
  - merge()- sincroniza el estado de una entidad en la bd
  - find()- busca una entidad por la clave
  - createQuery()- crea una consulta de JPQL
  - createNamedQuery()- crea una consulta nomenclada
  - createNativeQuery()- crea una consulta SQL
  - contains()- determina si una entidad está persistida
  - flush()- fuerza la sincronización de todas las entidades

Método Persist

```
public Order createNewOrder(Customer customer) {
   Order order = new Order(customer);
   entityManager.persist(order);
   return order;
}
```

- Solo puedo enviar instancias nuevas.

Métodos Find y Remove

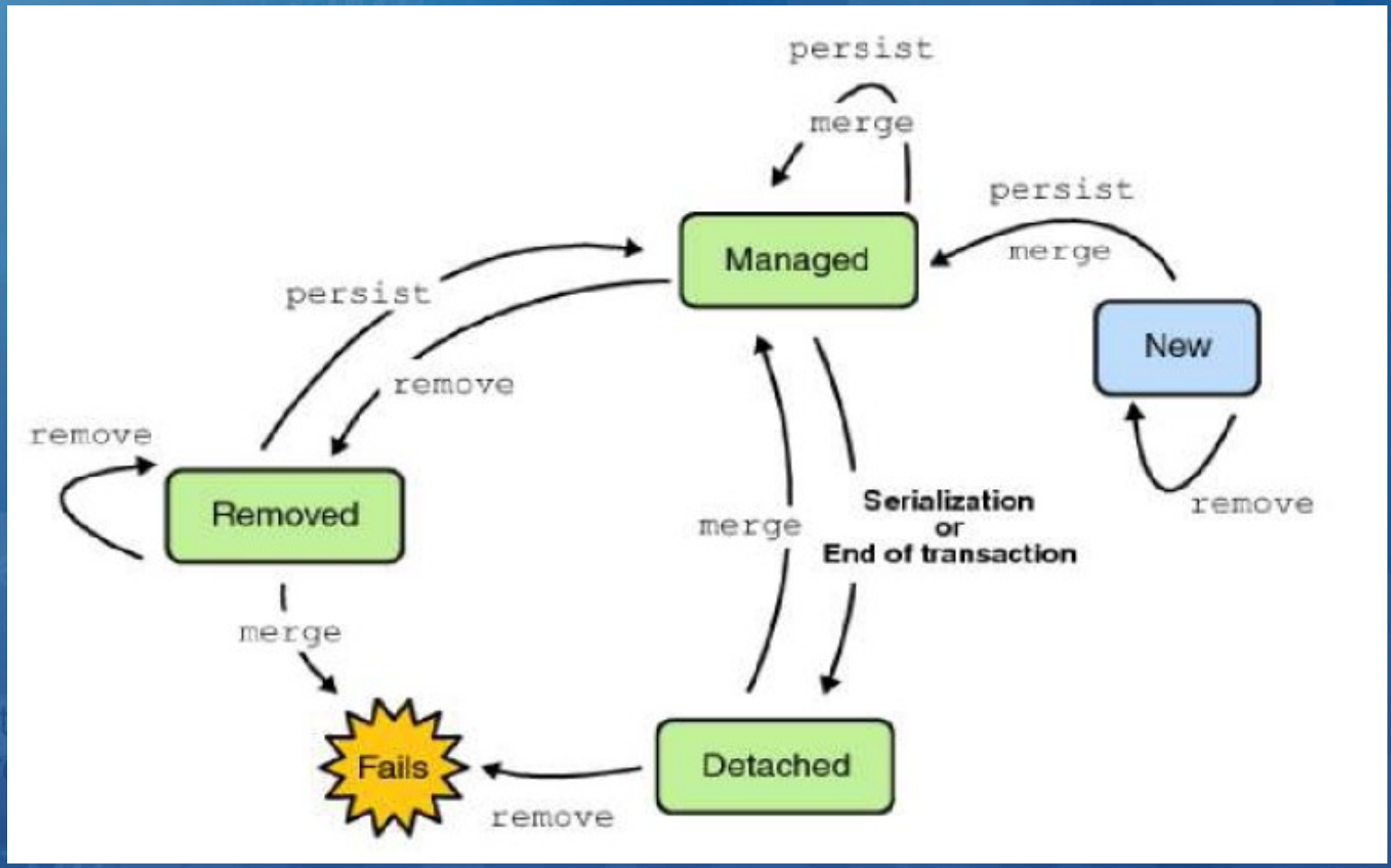
```
public void removeOrder(Long orderId) {
  Order order = entityManager.find(Order.class,
  orderId);
  entityManager.remove(order);
}
```

- Solo puede enviar instancias manejadas al métod *remove()*

Método Merge

```
public OrderLine updateOrderLine(OrderLine
  orderLine) {
  return entityManager.merge(orderLine);
}
```

Ciclo de Vida de las Entidades.



Entidades persistentes:
Ciclo de vida

- Las entidades persistentes pueden estar desacopladas de la base de datos o acopladas a ella a través de un EntityManager. En este caso están en el estado Managed (gestionadas).
- Por ejemplo, una entidad persistente creada a través de una consulta a la base de datos normalmente está gestionada por el EntityManager que hace la consulta.

Entidades persistentes:
Ciclo de vida, II

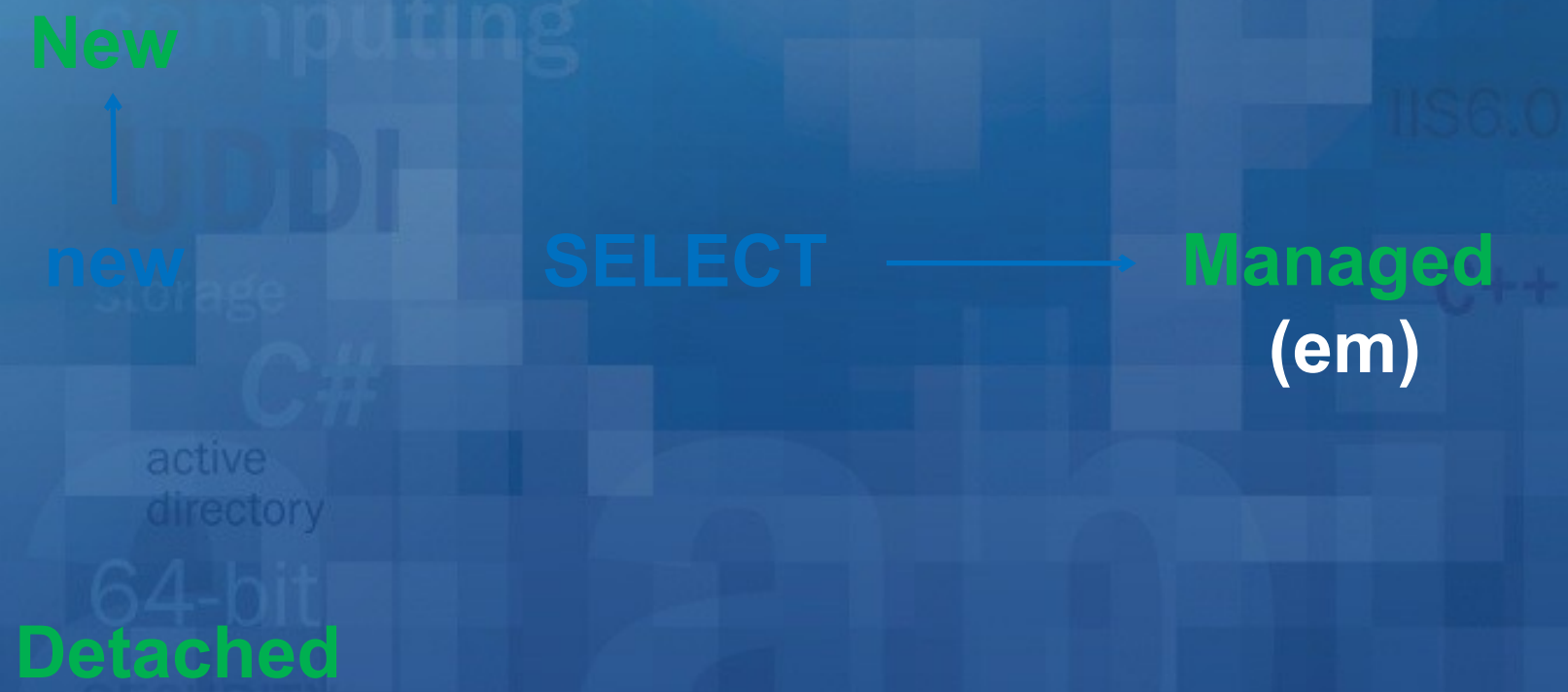
SELECT → Managed
(em)

Detached

Entidades persistentes:
Ciclo de vida, III

- Una entidad persistente creada mediante *new* está desacoplada de la base de datos y se encuentra inicialmente en el estado New.
- Si una entidad está en el estado New, en la base de datos puede haber un registro cuya clave primaria sea la misma de la entidad.

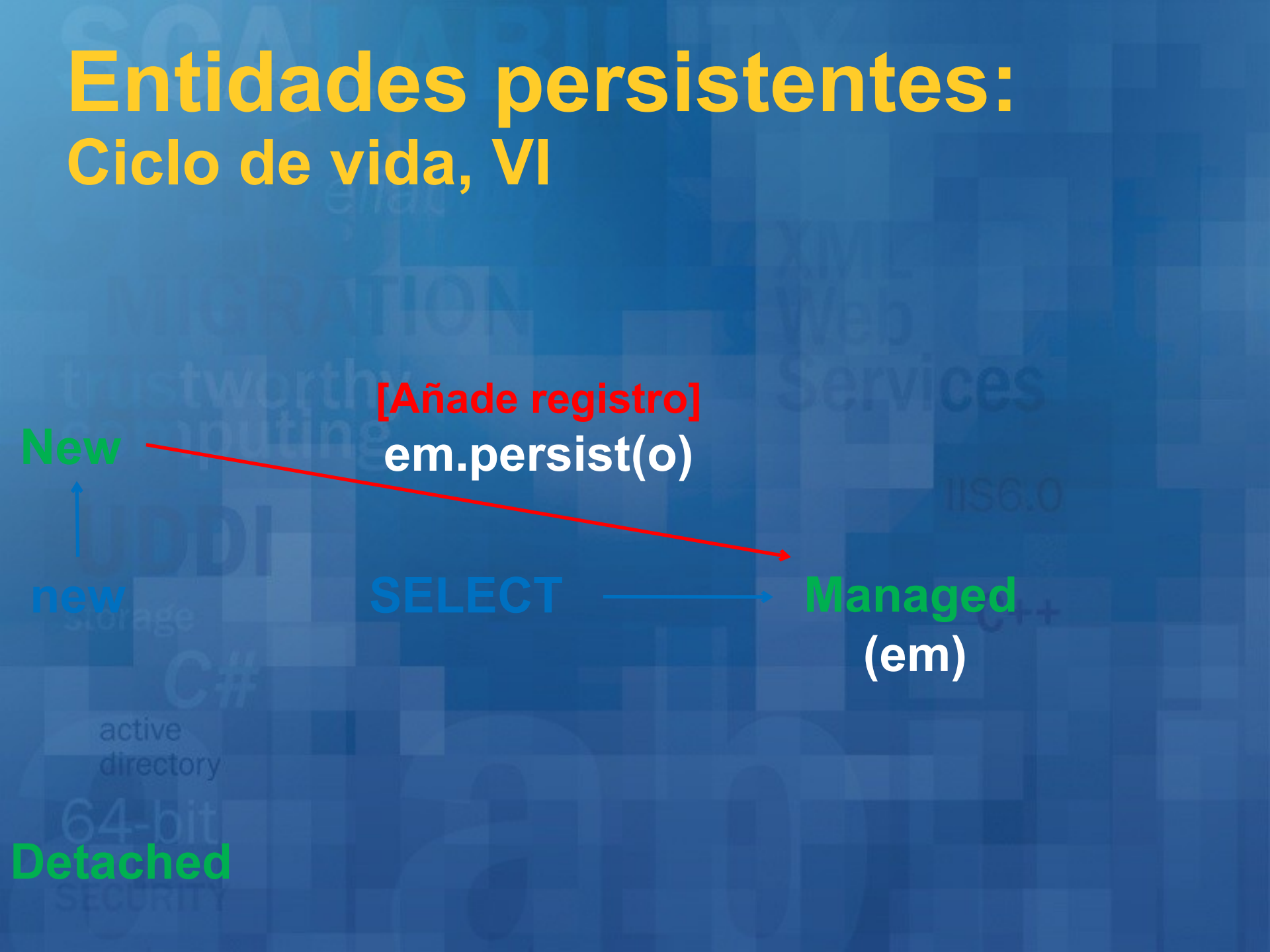
Entidades persistentes: Ciclo de vida, IV



Entidades persistentes:
Ciclo de vida, V

- Las entidades persistentes que están en el estado New pueden acoplarse a la base de datos mediante el método persist(Object) de la clase EntityManager, que las gestiona a partir de ese momento tras insertar los registros necesarios en la base de datos.
- Si en la base de datos ya hay un registro con la misma clave primaria, se lanza una excepción.

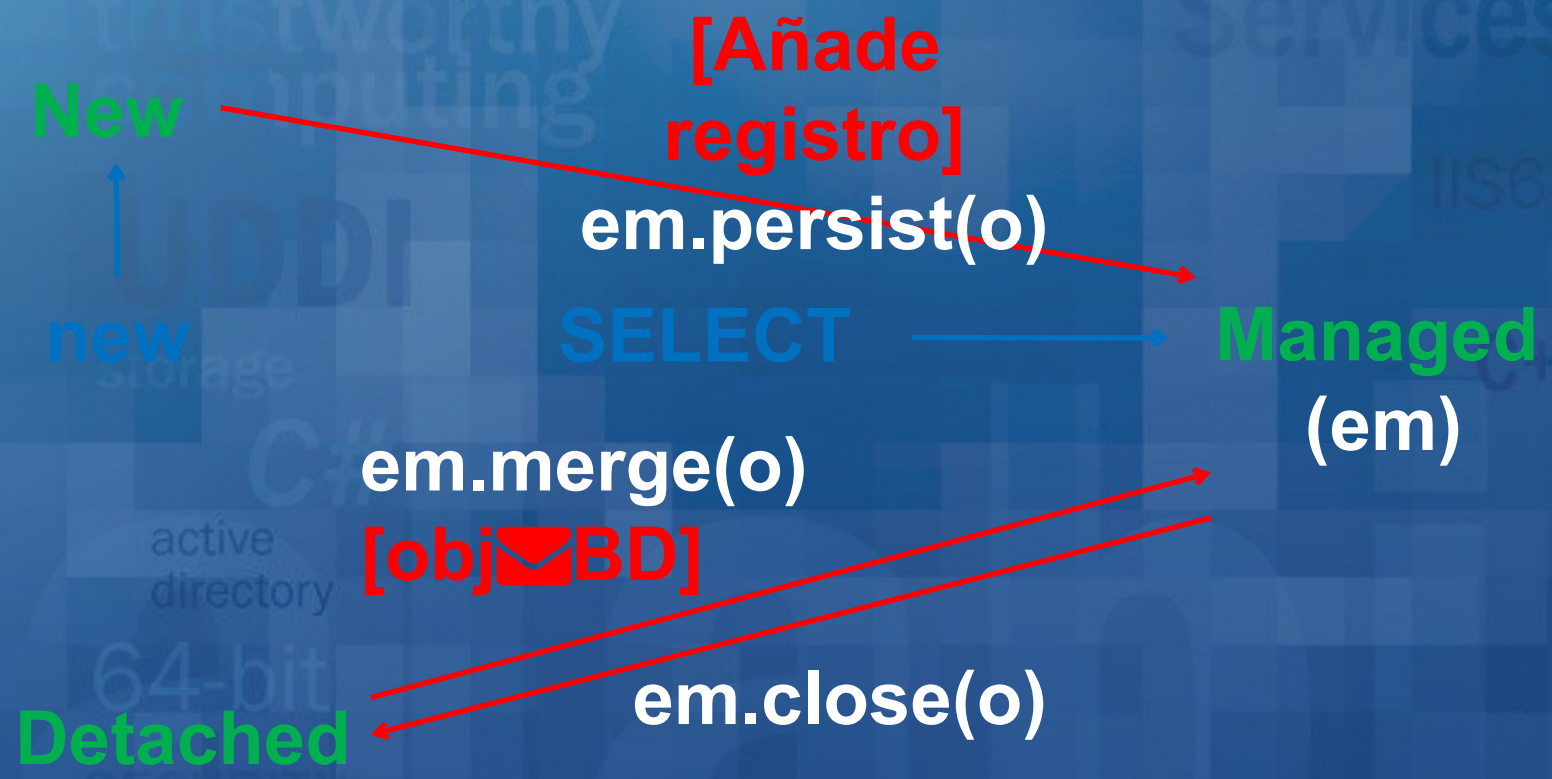
Entidades persistentes:
Ciclo de vida, VI



Entidades persistentes:
Ciclo de vida, VII

- Las entidades persistentes gestionadas se pueden desacoplar de la base de datos mediante el método close(Object) del EntityManager que las gestiona, pasando a estar en el estado Detached.
- El método merge(Object) de la clase EntityManager permite volver a gestionar una entidad persistente que está en el estado Detached. Tras su ejecución se actualizará la base de datos con su contenido.

Entidades persistentes:
Ciclo de vida, VIII



Entidades persistentes:
Ciclo de vida, IX

- Tanto en el estado New como en Detached los cambios que origina el cambio de estado de una entidad persistente no se trasladan necesariamente a la base de datos de inmediato (se hace cuando termina la transacción que se está ejecutando).
- Lo mismo ocurre con los cambios en los valores de los atributos persistentes de las entidades gestionadas.

Entidades persistentes:
Ciclo de vida, X

- Se puede forzar la realización inmediata en la base de datos de los cambios pendientes de realizar correspondientes a todas las entidades gestionadas por un EntityManager mediante el método flush() del mismo.
- Esto permite que cualquier otra consulta que se haga refleje los cambios realizados en las entidades gestionadas por el EM.

Entidades persistentes:
Ciclo de vida, XI

Entidades persistentes:
Ciclo de vida, XII

- Las consultas tienen asociado un modo de *flush*, que permite hacer que siempre que se ejecuten haya garantías de que la base de datos está actualizada con respecto a cambios en las entidades gestionadas.
- El modo de *flush* puede ser AUTO o COMMIT.
- Los EntityManagers también tienen un modo de *flush*.

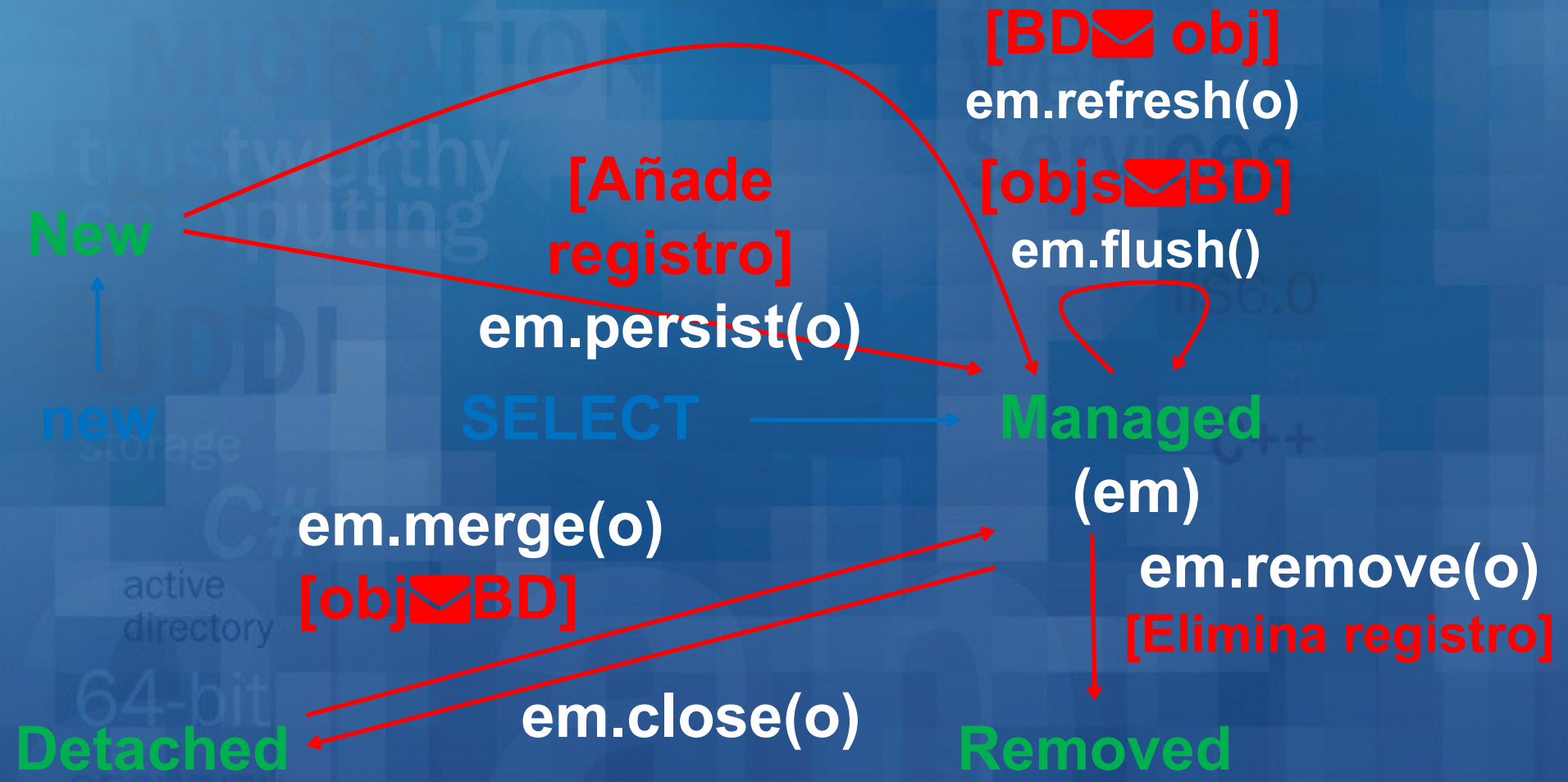
Entidades persistentes:
Ciclo de vida, XIII

- La actualización de los registros correspondientes a las entidades gestionadas por un EntityManager se puede hacer mediante una llamada explícita al método flush() en el programa o lo puede hacer el EntityManager al ejecutar acciones previstas en su funcionamiento (por ejemplo, cuando se termina una transacción).

Entidades persistentes:
Ciclo de vida, XIV

- También se puede actualizar una entidad acoplada a una base de datos con los valores que ésta contiene. Esto se hace mediante el método EntityManager.refresh(Object).
- Se puede borrar de una base de datos la información correspondiente a una entidad acoplada a ella, mediante el método remove de la clase EntityManager. La entidad pasa entonces al estado Removed.

Entidades persistentes:
Ciclo de vida, XV



Estados de las Entidades.

Estado	Significado
New	Una nueva instancia de la Entidad se creó usando new de Java. No hay un registro asociado a la llave primaria en la base de datos y la entidad no está conectada a un contexto de persistencia.
Managed	Hay un registro de la base de datos correspondiente y los datos se mantienen sincronizados. La entidad está conectada a un contexto de persistencia y tiene un identidad única. Solamente una instancia de una entidad "managed" puede existir en un contexto de persistencia.
Detached	Hay un registro de la base de datos correspondiente pero los datos no se sincronizan y la instancia de la entidad no está conectada a un contexto de persistencia.
Removed	Este estado representa que está pendiente el borrado del registro correspondiente en la base de datos.

Anotaciones Especiales.

Una Entidad puede ser notificada antes o después de cambios en su ciclo de vida, utilizando métodos con anotaciones especiales:

- @PrePersist
- @PostPersist
- @PreRemove
- @PostRemove
- @PreUpdate
- @PostUpdate
- @PostLoad

Ejemplo de un Query Nativo.

```
Query query = em.createNativeQuery("SELECT * FROM
      Customer",Customer.class);
List customers = query.getResultList();
Query query = em.createNativeQuery("SELECT * FROM
      SHARESWHERE
            SSN = '" + customerId + "'", CustomerShare.class);
List shares = query.getResultList();
```

- Usan SQL standard.
- No se recomiendan porque pueden hacer la aplicación poco portable.

Queries

- Queries Estáticos
  - Definidos con metadatos de lenguaje de java o XML
    - Annotations: @NamedQuery, @NamedNativeQuery
- Queries Dinámicos
  - El query se especifica en tiempo de ejecución
- Utiliza Lenguaje de Query de Persistencia de Java o SQL
- Parámetros nombrados o posicionados
- El EM es fábrica de objetos de query
  - createNamedQuery, createQuery, createNativeQuery
- Métodos de query para contolar resultados máximos, paginación, modo flush

Queries Dinámicos

```
// Build and execute queries dynamically
    at runtime.
public List findWithName (String name) {
    return em.CreateQuery (
    “SELECT c FROM Customer c ” +
    “WHERE c.name LIKE :custName”)
    .setParameter(“custName”, name)
    .setMaxResults(10)
    .getResultList();
}
```

Named Query

```
// Named queries are a useful way to create reusable
    queries
@NamedQuery(
    name="findCustomersByName",
    queryString="SELECT c FROM Customer c " +
    "WHERE c.name LIKE :custName"
)

@PersistenceContext public EntityManager em;
List customers =
em.createNamedQuery("findCustomersByName").
    setParameter("custName", "smith").getResultList();
```

Ejemplo de un Query de Persitence Query Language.

```
@Entity
@NamedQuery(name="FindAllOpenAuctions",
query=" SELECT OBJECT(a) FROM Auction AS a WHERE a.status=
'OPEN'")
```

- Usan el Persistence Query Language en vez de SQL.
- Hacen la aplicación altamente portable.
- No pueden utilizar características especiales de SQL.

Preguntas?

Felipe Steffolani
fsteffolani@sistemas.frc.utn.edu.ar