ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

# Programación de componentes de acceso a datos

# Concepto y características de un **componente**

Generalmente, un componente es una **unidad de software** que **encapsula** un segmento de **código** con ciertas **funciones**.

Los estilos de estos componentes pueden ser:

- estilos proporcionados por el **entorno de desarrollo** (incluidos en la **interfaz** de usuario)
- o pueden ser **no visuales**, siendo sus funciones similares a las de las bibliotecas remotas.

Los componentes del software tienen las siguientes características principales:

• Un componente es una unidad ejecutable que se puede instalar y utilizar de forma independiente.

• Puede interactuar y operar con otros componentes desarrollados por terceros, es decir, empresas o desarrolladores pueden utilizar componentes y agregarlos a las operaciones que se realizan, lo que significa que pueden estar **compuestos por estos componentes**.

• No tienen estado, al menos su estado no es visible desde 'fuera'.

• La programación orientada a componentes o un componente, se puede asociar a los distintos engranajes electrónicos que, en su **conjunto**, forman un **sistema más grande**.

Podemos afirmar que un componente será formado por los siguientes elementos:

1. Atributos, operaciones y eventos: Es parte de la interfaz del componente. Todo en su junto representaría el **nivel sintáctico**.

2. Comportamiento: Representa la parte **semántica**.

3. Protocolos y escenarios: Representa la **compatibilidad** de las secuencias de los mensajes con otros componentes, y la **forma de actuar** que tendrá el componente en diferentes **escenarios** donde llegará a ser ejecutado.

4. Propiedades: Las diferentes **características** que puede tener el componente.

# Propiedades y atributos de un componente
# Indexadas y Simples

Las propiedades del componente **determinan su estado** y lo distinguen del resto. Se ha adelantado anteriormente que estos componentes **carecen de estado**, pero estos componentes tienen una serie de propiedades a las que se puede **acceder y modificar** de diferentes formas.

Las propiedades de los componentes se dividen en:

1. **simples**,
2. **indexadas**,
3. **compartidas**,
4. y **restringidas**.

## Alcance de las propiedades de los beans de Java

Los atributos y las propiedades de los beans de java deben mantener su scope (alcance) **privado**. De esta forma fomentamos la **seguridad y el acceso** a dichas propiedades y atributos, y sólo se propagarán los cambios **llamando al método** determinado de dicha propiedad.

Se puede verificar y modificar las propiedades del componente accediendo al método del componente o la función de acceso.

Hay dos tipos de estas funciones:

• El método get se utiliza para **consultar** o leer el valor de un atributo.

La sintaxis general es la siguiente:

```
Tipo getNombrePropiedad()
```

• El método set se utiliza para **asignar** o cambiar el valor de un atributo.

Su sintaxis general en Java es:

```
SetNombrePropiedad(Tipo valor)
```

Centrándonos en las propiedades o atributos indexados y simples, podemos decir, que:

- los atributos simples son atributos que **representan un solo valor**.

  Por ejemplo, si hay un botón en la interfaz gráfica, los atributos simples serán atributos relacionados con su **tamaño**, **color** de fondo, **etiqueta**, etc.

- Los atributos o propiedades indexadas: son propiedades más complejas que son muy similares a un conjunto de valores: los índices. Todos los **elementos de este tipo** de atributo comparten el **mismo tipo** y se puede acceder a ellos **por índice**.

  Concretamente, se puede **acceder** mediante el método de acceso que se mencionó anteriormente (**get / set**), aunque la llamada de estos métodos variará, ya que **solo se puede acceder a cada propiedad a través de su índice**.

  La sintaxis que encontramos en Java sería:

```
setPropertyName (int index, PropertyType value)
```

# Propiedades y atributos de un componente

## Compartidas y Restringidas

Además de los atributos simples y los atributos indexados, los componentes también tienen otros dos tipos de atributos:

- Atributos compartidos.
- Atributos restringidos.

Los atributos compartidos:

Se refieren a los datos mediante los que **informa a todos los interesados** sobre el atributo, cuando cambian, por lo que solo se les notifica **cuando cambia el atributo**. El mecanismo de notificación se basa en eventos, es decir, hay un **componente de origen** que notifica al **componente receptor** cuando un atributo compartido cambia a través de un **evento**. Debe quedar claro que estas propiedades no son bidireccionales, por lo que el **componente receptor no puede responder**.

Para que un componente **permita propiedades compartidas**, debe **admitir dos métodos** Java para registrar componentes que estén interesados en cambios de propiedad.

Los métodos y su sintaxis son:

- Para agregar: addPropertyChangeListener (PropertyChangeListener x)
- Para eliminar: removePropertyChangeListener (PropertyChangeListener x)

Las propiedades restringidas:

Buscarán la **aprobación de otros componentes** antes de cambiar su valor.

Al igual que con los atributos compartidos, se deben proporcionar **dos métodos** de registro para el receptor.

Su sintaxis general en Java sería:

- addVetoableChangeListener (VetoableChangeListener x),
- removeVetoableChangeListener (VetoableChangeListener x).

Las **propiedades** son uno de los **aspectos relevantes de la interfaz** del componente porque son elementos que forman parte de la **vista externa** del componente (representada como una vista **pública**). Desde la perspectiva del control y uso de componentes, estos elementos observables son particularmente importantes y son la **base para caracterizar otros aspectos** de los componentes.

## POJOS y beans

Bienvenidos al ejemplo de un POJO en Java. Ahora vamos a revisar algunos detalles y diferencias entre los "beans" y los POJOs. Todos los "beans" son POJOs, pero no todos los POJOs son "beans". Aunque pueda parecer un poco confuso, así es como funciona.

Este es un ejemplo básico de un POJO. Tenemos la clase 'Empleado', que tiene varios atributos y un constructor público que nos permite crear un objeto a partir de cero, introduciendo estos atributos. También tenemos nuestros getters aquí. Podríamos agregar setters también, pero este es un ejemplo básico.

¿Qué tenemos que tener en cuenta acerca de los "beans"? Bueno, en comparación con los POJOs, todos los "beans", como mencionamos antes, son POJOs. Pero no todos los POJOs son "Java beans". Otro punto importante es que **implementan** la interfaz '**Serializable**'. Todos los campos deben ser **privados** en los "beans". Esto se hace para proporcionar un control completo sobre los campos. Mientras que **en los POJOs no es obligatorio**; podemos tener campos privados o públicos.

Además, todos los campos en los "**beans**" deben tener sus **getters y/o setters**. Otra característica es que pueden tener un constructor sin argumentos. Luego, los campos solo pueden ser accedidos mediante el constructor, los getters o los setters.

Es importante mencionar que los **getters y setters** tienen ciertos **nombres especiales**, dependiendo del nombre del campo. Por ejemplo, si el nombre del campo es 'xProperty' o 'someProperty', preferiblemente el getter será 'getSomeProperty'.

Por último, la **visibilidad de los getters y setters** en general es **pública**, precisamente para acceder a los atributos desde esos métodos. Además, la visibilidad de los atributos mismos se establece como pública, para que cualquier objeto pueda usarlos.

Básicamente, estas son algunas de las diferencias entre los "beans" y los POJOs. Los "beans" son objetos más restrictivos en comparación con los POJOs.

Ejemplo de POJO en Java

```java
package pruebas;

public class Employee {
    // campo por defecto
    String name;
    // campo publico
    public String id;
    // salario privado
    private double salary;
    // constructor para inicializar los campos
    public Employee(String name, String id, double salary) {
        this.name = name;
        this.id = id;
        this.salary = salary;
    }
    // getters
    public String getName() {
        return name;
    }
    public Strind getId() {
        return id;
    }
    public Double getSalary() {
        return salary;
    }
}
```

## Ejemplo de Implementación de un componente y propagación

Se requiere la realización de un componente que a través de la clase PropertyChangeSupport ejecute los métodos propagados de:

- addPropertyChangeListener
- removePropertyChangeListener

Tal y como hemos visto en los puntos anteriores de la unidad, nos dispondremos a usar la lógica de componentes EJB para así poder propagar los diferentes cambios.

1. En primer lugar deberemos de **importar** la paquetería java.beans.
2. Luego crearíamos un objeto de la clase mencionada PropertyChangeSupport:

   PropertyChange objeto

   ```
   private PropertyChangeSupport objetoCambios = new PropertyChangeSupport(this);
   ```

   Como podemos observar, el objeto del código anterior mantiene una lista de oyentes y por consiguiente, lanzará los diferentes eventos de cambio de propiedad.
3. A continuación, implementaremos los **métodos que nos ofrece PropertyChangeSupport**, simplemente envolveremos las diferentes llamadas de los métodos del objeto soportado:

   Métodos

   ```
   public void addPropertyChangeListener(
       PropertyChangeListener listener) {
           changes.addPropertyChangeListener(listener);
     }

   public void removePropertyChangeListener(
        PropertyChangeListener listener) {
           changes.removePropertyChangeListener(listener);
        }
   ```

# Eventos

La **iteración entre componentes** se denomina control activo, es decir, el **funcionamiento** de un componente se activa mediante la **llamada de otro componente**.

Además del control activo (la forma más común de invocar operaciones), existe otro método llamado control reactivo, que se refiere a **eventos de componentes**, como los permitidos por el modelo de componentes EJB.

En el control reactivo, los componentes pueden generar eventos correspondientes a **solicitudes de invocación de operaciones**. Posteriormente, otros componentes del sistema recolectarían estas solicitudes, y activarían **llamadas a operaciones** para sus propósitos de procesamiento.

Por ejemplo, cuando el puntero del mouse pasa sobre el ícono del escritorio, si la forma del ícono del escritorio cambia, el componente de icono emitiría eventos relacionados con la operación de cambiar la forma del icono.

# Introspección

Las herramientas de desarrollo **descubren las características** de los componentes (es decir, propiedades de los **componentes, métodos y eventos**) a través de un proceso llamado introspección. La introspección, por tanto, se puede definir como un mecanismo a través del cual se pueden **descubrir las propiedades, métodos y eventos** contenidos en el componente.

Los componentes admiten la introspección de dos formas:

- Mediante el uso de convenciones de nomenclatura específicas, que se conocen al nombrar las características de los componentes. Para EJB, la clase **Introspector** comprueba el EJB para encontrar esos **patrones de diseño** y descubrir sus características.
- Proporcionando información clara sobre atributos, métodos o eventos de clases relacionadas.

En particular, EJB admite **múltiples niveles de introspección**.

En un nivel bajo, esta introspección se puede lograr mediante la posibilidad de **reflexión**. Estas características permiten a los objetos Java descubrir **información** sobre **métodos** públicos, **campos** y **constructores** de clases cargados durante la ejecución del programa. La reflexión permite la introspección de todos los componentes del software, y todo lo que se tiene que hacer es **declarar el método o variable como público** para que pueda ser descubierto a través de la reflexión.

## Persistencia del componente

Para EJB, la persistencia se proporciona con la biblioteca JPA de Java. Hoy en día podemos encontrar tal especificación en **Oracle**. Para usar JPA, se requiere el uso al menos del **JDK 1.5** (conocido como Java 5) en adelante, ya que amplía las nuevas especificaciones del lenguaje Java, como son las **anotaciones**. Para utilizar la biblioteca JPA, es esencial tener un conocimiento sólido de **POO** (programación orientada a objetos), **Java**, y **lenguaje** de **consulta estructurado**.

Las clases que componen JPA son las siguientes:

- Persistence.

  La clase javax.persistence.Persistence contiene un **método auxiliar estático**, el cual usamos para tener un **objeto EntityManagerFactory** de forma **independiente** del que se obtiene a través de JPA.

- EntityManagerFactory.

  La clase javax.persistence.EntityManagerFactory: de la cual **extraemos objetos de tipo EntityManager** usando el conocido patrón de Factory, (o patrón de diseño Factory). (Es una fábrica de objetos EntityManager).

- EntityManager.

  La clase javax.persistence.EntityManager es la **interfaz JPA principal** para la persistencia de aplicaciones. Cada EntityManager puede **crear, leer, modificar y eliminar** (CRUD) un grupo de objetos persistentes.

- Entity.

  La clase Entity es **una anotación** Java. La encontramos al **mismo nivel** que las clases Java **serializables**.

  Cada una de estas entidades las representamos como **registros** diferentes en base de datos.

- EntityTransaction.

  Permite **operaciones conjuntas** con datos persistentes, de tal forma, que pueden crear **grupos para persistir** distintas operaciones. Si todo el grupo realiza las operaciones sin ningún problema se persistirá, de lo contrario, todos los intentos fallarán. En caso de error, la base de datos realizará **rollback** y volverá al estado anterior.

- Query.

  Cada proveedor de JPA ha implementado la interfaz javax.persistence.Query para **encontrar objetos persistentes** procesando ciertas condiciones de búsqueda. JPA utiliza JPQL y SQL para estandarizar el soporte de consultas. Podremos obtener un objeto Query **a través del EntityManager**.

Finalmente, aquellas **anotaciones JPA** (o anotaciones EJB 3.0) las encontraremos en **javax.Persistence**.

Muchos **IDE que admiten JDK5** (como Netbeans, Eclipse, IntelliJ IDEA) poseen **utilidades y complementos** para exportar clases de entidad usando las diferentes anotaciones JPA partiendo de la arquitectura de la base de datos.

## Java EE tutorial

Abordamos los componentes Java, los EJBeans en general, así como el módulo de persistencia y los POJOs. En la página oficial de documentación de Oracle, encontramos un tutorial que nos enseña cómo crear nuestro propio Java Enterprise Edition desde cero. Este tutorial nos proporciona una visión general del tema y nos guía a través de la instalación del entorno, la preparación de la plataforma e incluso el despliegue con Apache.

Además, aborda los capítulos de Enterprise Beans y persistencia, donde podemos revisar los contenedores y profundizar en conceptos clave.

El módulo de persistencia, por ejemplo, explora la API y nos introduce en su funcionamiento. En la documentación, encontramos información sobre el Entity Manager, su interfaz y cómo utilizarlo. Además de estos temas, el tutorial también ofrece detalles sobre web services y otros servicios relacionados. Es una fuente de información muy completa que merece la pena explorar en detalle.

https://docs.oracle.com/javaee/5/tutorial/doc/

# Ejemplo de persistencia de 2 objetos

A modo recordatorio de persistencia se requiere realizar un ejemplo de persistencia de 2 objetos de clase persona cuyos atributos sean según constructor:

- Nombre
- Edad
- Dirección

Se requiere usar la clase EntityManagerFactory.

Crearemos distintos objetos de una clase persona y los persistiremos.

A continuación, mostraremos el código por medio del cual persistiremos 2 objetos de la clase persona.

Primero nos crearemos los diferentes objetos mediante constructor:

Objetos persona

```
Persona persona1 = new Persona("pedro",25, "calle 1");
Persona persona2 = new Persona("pedro",25, "calle 7");
```

Y después, procedemos a crear EntityManagerFactory, y a persistir los diferentes objetos:

Persistiendo

```
EntityManagerFactory emf =
      Persistence.createEntityManagerFactory("FactoryPersonas");
EntityManager em =
      emf.createEntityManager();
try {
    em.getTransaction().begin();
    em.persist(persona1);
    em.persist(persona2);
    em.getTransaction().commit();
} catch (Exception e) {
    e.printStackTrace();
}finally {
    em.close();
}
```

# Empaquetado de componentes

Un módulo J2EE consta de **uno o más componentes** J2EE del **mismo tipo** de contenedor, y **descriptores** de implementación de componentes de este tipo.

> El **descriptor** de despliegue del módulo EJB especifica los **distintos atributos** de nuestra transacción y la **autorización** de seguridad del EJB.
>
> Los módulos JavaEE **sin descriptores** de implementación de aplicaciones se pueden implementar como **módulos separados**.

Antes de EJB 3.1, todos los **EJB** debían estar **empaquetados** en estos archivos.

Como buena parte de todas las **aplicaciones J2EE**, contienen un **front-end web** y un **back-end EJB**, lo que significa, que se debe crear un .ear que contenga una aplicación con **dos módulos**:

- **.war**
- y **ejb-jar**.

Esta es una buena práctica en el sentido de establecer una clara **separación estructural** entre la parte 'delantera' y la 'trasera.' No obstante, esta diferenciación será complicado delimitarla en aplicaciones simples.

Hay cuatro tipos de módulos J2EE para aplicaciones web EJB:

1. Módulos **EJB**, que contienen archivos con **clases** EJB y **descriptores** de despliegue EJB. El módulo EJB está empaquetado como un archivo **JAR** con extensión **.jar**.
2. El módulo **Web**, que contiene:
   - archivos con **Servlet**,
   - archivos **JSP**,
   - archivos de **soporte** de clases,
   - archivos **GIF** y **HTML**,
   - y **descriptores** de implementación de aplicaciones web.

   El módulo web está empaquetado como un archivo **JAR** con una extensión **.war** (**archivo web**).
3. El módulo de la aplicación **cliente**, que contiene archivos con **clases y descriptores** de la aplicación cliente.

   El módulo de la aplicación cliente está empaquetado como un archivo **JAR** con extensión **.jar**.
4. Módulo **adaptador** de recursos, que contiene:
   - todas las **interfaces**,
   - **clases**,
   - **bibliotecas** nativas,

- y otros **documentos** de Java,
- y sus **descriptores** de implementación del adaptador de recursos.

### Herramientas para desarrollo de componentes no visuales

Actualmente, los entornos de desarrollo de componentes más utilizados son:

- **Intellij**,
- **NetBeans**,
- y **Eclipse**.

- Eclipse se utiliza principalmente para el desarrollo de **componentes Java** (como EJB),
- y Microsoft.NET para el desarrollo de **ensamblajes** (**COM +, DCOM**).

Estos entornos proporcionan una alternativa a la **generación automática de código** y facilitan enormemente el desarrollo de aplicaciones que utilizan componentes.

## Ejemplo de uso de Transaction

Trabajaremos sobre el objeto EntityTransaction.

Haremos uso de dicho objeto para realizar la persistencia en la capa de acceso a datos:

Uso Entity Transaction

```
private static void agregarEntidad {
    EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("PERSISTENCIA");
    EntityManager entityManager = entityManagerFactory.createEntityManager();
    EntityTransaction entityTransaction = entityManager.getTransaction();
    entityTransaction.begin();
    Student estudiante = new Student("Juan", "Lopez", "micorreo@codigos.com");
    entityManager.persist(estudiante);
    entityTransaction commit();
    entityManager.close();
    entityManagerFactory.close();
}
```

Como podemos observar en el código expuesto, englobamos en un **método privado** el método que nos hará dichas **funciones**. En él:

1. en primer lugar, creamos una **variable** de tipo **EntityManagerFactory** donde damos nombre a nuestra factoría.
2. A continuación creamos un **entityManager** del cual extraemos la **transacción** con **getTransaction()** para poder empezar a **realizar operaciones**.
3. **Creamos** un estudiante 'normal', como ejemplo.
4. Lo **persistimos**,
5. y con nuestra transacción hacemos **commit**() para que sea efectiva.

De este modo usaremos nuestro objeto para **persistir** los diferentes **componentes** y objetos Java en general.

https://docs.oracle.com/javaee/5/tutorial