INSTITUTO SUPERIOR TECNOLÓGICO

JAPÓN

Amor al conocimiento

GUÍA METOLÓGICA

LENGUAJE DE PROGRAMACIÓN II

DESARROLLO DE SOFTWARE

COMPILADOR: ING. YECENIA CEVALLOS

2019

INSTITUTO SUPUERIOR TECNOLÓGICO JAPÓN

GUIA DE APRENDIZAJE

1. IDENTIFICACIÓN DE

Nombre de la Asignatura: **LENGUAJE DE PROGRAMACIÓN II**	**Componentes del Aprendizaje**	

Resultado del Aprendizaje:

- Definir las estructuras de datos, incluyendo su forma y operaciones
- Construir estructuras de datos en aplicaciones prácticas.
- Implementa estructuras de datos en Java
- Utiliza las clases stream de caracteres para leer y escribir archivos
- Utiliza las clases Streams de bytes para leer y escribir archivos
- Aplica las conexiones JDBC para acceder a la información de una base de datos en la construcción de programas.
- Aplica programas con acceso a Base de datos como soluciones integrales para las organizaciones.
- Aplica expresiones lambda para búsqueda de datos en colecciones.
- Usa Enterprise Java Beans y Message Driven Beans.
- Hace uso de algunos patrones de diseño.

Docente de Implementación:

Ing. Yecenia Cevallos	**Duración:** 32 horas

Unidades	Competencia	Resultados de Aprendizaje	Actividades	Tiempo de Ejecución
Estructuras de datos en Java	Utiliza la interfaz Collection para el desarrollo de problemas computacionales en el lenguaje de programación Java	▪ Definir las estructuras de datos, incluyendo su forma y operaciones ▪ Construir estructuras de datos en aplicaciones prácticas. ▪ Implementa estructuras de datos en Java	▪ Clase expositiva. ▪ Práctica de Laboratorio	8

Java I/O	Utiliza la librería de java I/O para la lectura y escritura de archivos utilizando el lenguaje de programación Java	▪ Utiliza las clases stream de caracteres para leer y escribir archivos ▪ Utiliza las clases Streams de bytes para leer y escribir archivos	▪ Clase expositiva ▪ Práctica de Laboratorio ▪ Revisión de errores	4
Java I/O	Analiza la interface Java Database Connectivity (JDBC) para acceder a la información de una base de datos en la construcción de programas.	▪ Aplica las conexiones JDBC para acceder a la información de una base de datos en la construcción de programas. ▪ Aplica programas con acceso a Base de datos como soluciones integrales para las organizaciones.	▪ Clase expositiva ▪ Práctica de Laboratorio ▪ Revisión de errores	8
Expresiones Lambda	Analiza las expresiones lambda para el desarrollo de problemas computacionales en el lenguaje de programación Java	▪ Aplica expresiones lambda para búsqueda de datos en colecciones	▪ Clase expositiva ▪ Práctica de Laboratorio ▪ Revisión de errores	4

Enterprise Javabeans	Analiza la arquitectura Enterprise JavaBeans (EJB) para la creación de componentes de aplicaciones distribuidas y orientadas a transacciones utilizando el lenguaje de programación java.	▪ Usa Enterprise Java Beans y Message Driven Beans	▪ Clase expositiva ▪ Lluvia de ideas	4
Patrones de diseño	Analiza los patrones de diseño para el desarrollo de problemas computacionales en el lenguaje de programación Java	▪ Hace uso de algunos patrones de diseño	▪ Clase expositiva ▪ Lluvia de ideas	4

2. CONOCIMIENTOS PREVIOS Y RELACIONAD

Co-requisitos

3. UNIDADES TEÓRICAS

A. Base Teórica

UNIDAD 1: .ESTRUCTURAS DE DATOS EN JAVA

En la práctica, la mayor parte de información útil no aparece aislada en forma de datos simples, sino lo hace de forma organizada y estructurada. Los diccionarios, guías, enciclopedias, etc., son colecciones de datos que serían inútiles si no estuvieran organizados de acuerdo con unas determinadas reglas. Además tener estructurada la información supone ventajas adicionales, al facilitar el acceso y el manejo de los datos. Por ello parece razonable desarrollar la idea de la agrupación de datos, que tengan un tipo de estructura y organización interna.

Las estructuras de datos nos permiten resolver un problema de manera más sencilla gracias a que las reglas que las rigen nunca cambian, así que puedes asumir que ciertas cosas son siempre ciertas. Adicionalmente son dinámicas, si usas lenguajes de programación como Java, sabrás que necesitas definir el tamaño de los arrays antes de ser usados. Usando una estructura de datos, puedes hacer “un array” de tamaño indeterminado.

1.1.Colecciones

Las colecciones representan grupos de objetos, denominados elementos. Podemos encontrar diversos tipos de colecciones, según si sus elementos están ordenados, o si permitimos repetición de elementos o no.

Es el tipo más genérico en cuanto a que se refiere a cualquier tipo que contenga un grupo de elementos. Viene definido por la interfaz `Collection`, de la cual heredará cada subtipo específico.

En java las principales interfaces que disponemos para trabajar con colecciones son: (de los Santos, 2018)

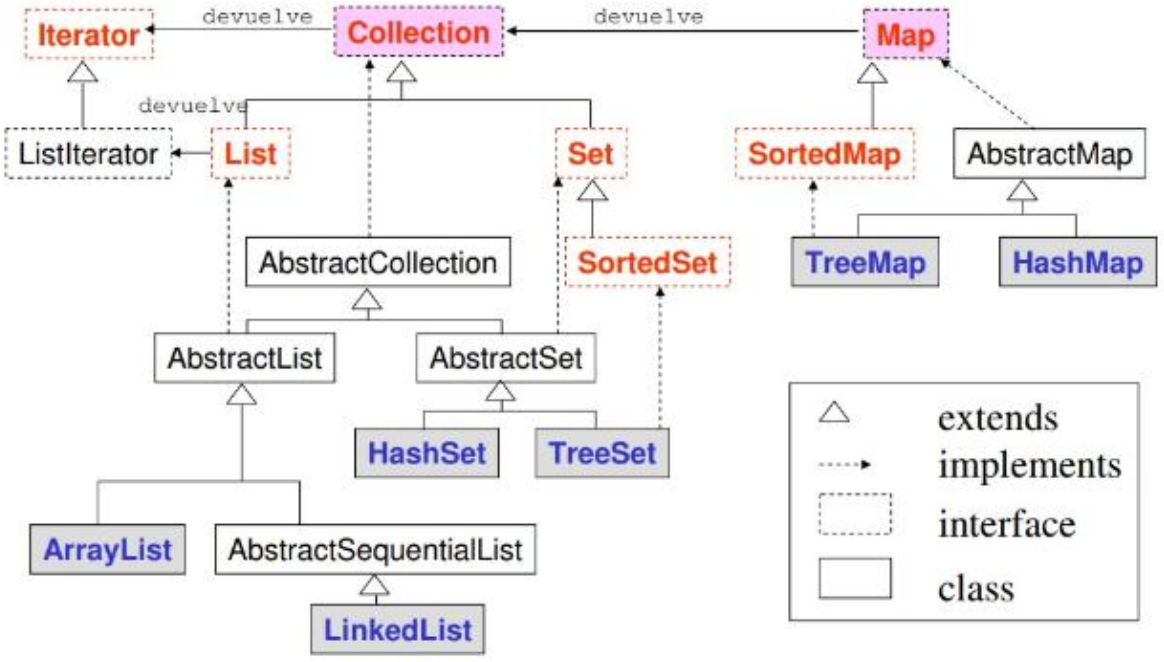

Ilustración 1: Jerarquía de interfaces

Fuente: Colecciones en java

Acedo Jose (2011), define lo siguiente:

Collection<E>: Un grupo de elementos individuales, frecuentemente con alguna regla aplicada a ellos.

List<E>: Elementos en una secuencia particular que mantienen un orden y permite duplicados. La lista puede ser recorrida en ambas direcciones con un ListIterator. Hay 3 tipos de constructores:

a. **ArrayList<E>**: Su ventaja es que el acceso a un elemento en particular es ínfimo. Su desventaja es que para eliminar un elemento, se ha de mover toda la lista para eliminar ese «hueco».

b. **Vector<E>**: Es igual que ArrayList, pero sincronizado. Es decir, que si usamos varios hilos, no tendremos de qué preocuparnos hasta cierto punto.

c. **LinkedList<E>**: En esta, los elementos están conectados con el anterior y el posterior. La ventaja es que es fácil mover/eliminar elementos de la lista, simplemente moviendo/eliminando sus referencias hacia otros elementos. La desventaja es que para usar el elemento N de la lista, debemos realizar N movimientos a través de la lista.

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // LinkedList
        List lista1 = new LinkedList();

        // Añadimos nodos y creamos un Iterator
        lista1.add("Madrid");
        lista1.add("Sevilla");
        lista1.add("Valencia");
        Iterator iterador = lista1.iterator();

        // Recorremos y mostramos la lista
        while (iterador.hasNext()) {
            String elemento = (String) iterador.next();
            System.out.print(elemento + " ");
        }
        System.out.println("--LinkedList--");

        // ArrayList
        List lista2 = new ArrayList();

        // Añadimos nodos y creamos un Iterator
        lista2.add("Madrid");
        lista2.add("Sevilla");
        lista2.add("Valencia");
        Iterator iterador2 = lista2.iterator();

        // Recorremos y mostramos la lista
        while (iterador2.hasNext()) {
            String elemento = (String) iterador2.next();
            System.out.print(elemento + " ");
        }
        System.out.println("--ArrayList--");
    }
}
```

Ilustración 2: Ejemplo List

Fuente: Colecciones y tipos genéricos en Java

Set<E>: No puede haber duplicados. Cada elemento debe ser único, por lo que sí existe uno duplicado, no se agrega. Por regla general, cuando se redefine *equals()*, se debe redefinir *hashCode()*. Es necesario redefinir *hashCode()* cuando la clase definida será colocada en un HashSet. Los métodos *add(o)* y *addAll(o)* devuelven false si o ya estaba en el conjunto.

Queue<E>: Colección ordenada con extracción por el principio e inserción por el principio (LIFO – Last Input, First Output) o por el final (FIFO – First Input, First Output). Se permiten elementos duplicados. No da excepciones cuando la cola está vacía/llena, hay métodos para interrogar, que devuelven *null*. Los métodos *put()*/*take()* se bloquean hasta que hay espacio en la cola/haya elementos.

Map<K,V>: Un grupo de pares objeto clave-valor, que no permite duplicados en sus claves. Es quizás el más sencillo, y no utiliza la interfaz Collection. Los principales métodos son: *put(), get(), remove()*.

a. **HashMap<K,V>**: Se basa en una tabla hash, pero no es sincronizado.
b. **HashTable<K,V>**: Es sincronizado, aunque que no permite *null* como clave.
c. **LinkedHashMap<K,V>**: Extiende de HashMap y utiliza una lista doblemente enlazada para recorrerla en el orden en que se añadieron. Es ligeramente más rápida a la hora de acceder a los elementos que su superclase, pero más lenta a la hora de añadirlos.

TreeMap<K,V>: Se basa en una implementación de árboles en el que ordena los valores según las claves. Es la clase más lenta.

```java
import java.util.*;

public class Ejemplo
{
    public static void main(String args[])
    {
        // Definir un HashMap
        HashMap global = new HashMap();

        // Insertar valores "key"-"value" al HashMap
        global.put("Laura", "667895789");
        global.put("Pepe", "645895756");
        global.put("Abelardo", "55895711");
        global.put("Daniel", "667111788");
        global.put("Arturo", "667598623");

        // Definir Iterator para extraer o imprimir valores
        for( Iterator it = global.keySet().iterator(); it.hasNext();) {
            String s = (String)it.next();
            String s1 = (String)global.get(s);
            System.out.println("Alumno: "+s + " - " + "Telefono: "+s1);
        }
    }
}
```

Ilustración 3: Ejemplo de Map

1.2.Listas enlazadas

Una lista enlazada es una colección o secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente elemento por un "enlace" o "referencia". La idea básica consiste en construir una lista cuyos elementos, llamados nodos, se componen de dos partes (campos): la primera parte contiene la información y es, por consiguiente, un valor de un tipo genérico (denominado Dato, TipoElemento, Info, etc.), y la segunda parte es una referencia (denominado enlace) que apunta (enlaza) al siguiente elemento de la lista. (Sistemas, n.d.)

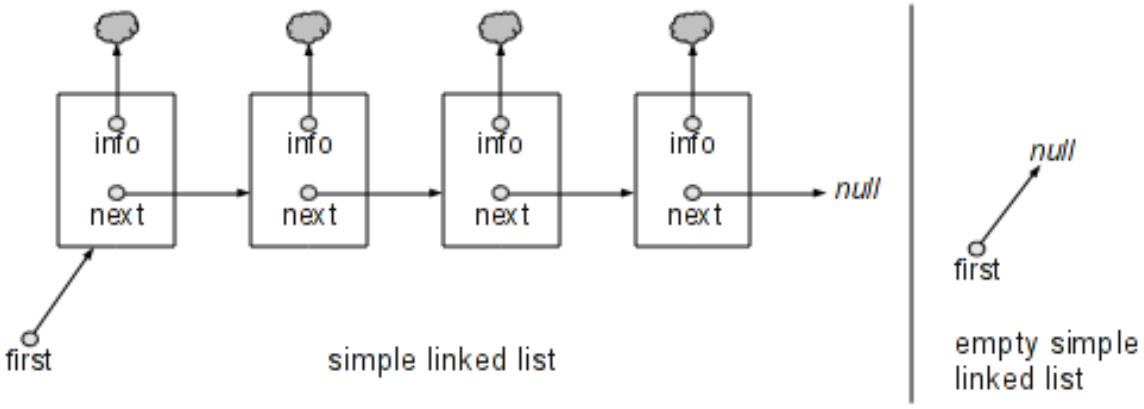

Ilustración 4: Listas enlazadas

Las listas se pueden dividir en cuatro categorías:

- **Listas simplemente enlazadas**. Cada nodo (elemento) contiene un único enlace que lo conecta al nodo siguiente o nodo sucesor. La lista es eficiente en recorridos directos (“adelante”).
- **Listas doblemente enlazadas**. Cada nodo contiene dos enlaces, uno a su nodo predecesor y otro a su nodo sucesor. La lista es eficiente tanto en recorrido directo (“adelante”) como en recorrido inverso (“atrás”).
- **Lista circular simplemente enlazada**. Una lista enlazada simplemente en la que el último elemento (cola) se enlaza al primer elemento (cabeza) de tal modo que la lista puede ser recorrida de modo circular (“en anillo”).
- **Lista circular doblemente enlazada**. Una lista doblemente enlazada en la que el último elemento se enlaza al primer elemento y viceversa. Esta lista se puede recorrer de modo circular (“en anillo”) tanto en dirección directa (“adelante”) como inversa (“atrás”).

Ejemplo:

ListasEnlazadas.java

Source | History

```
1  package Colecciones;
2  //LinkedList
3  //almacena un nodo
4
5  import java.util.*;
6
7  public class ListasEnlazadas {
8
9      public static void main(String[] args) {
10         LinkedList<String> persona = new LinkedList<String>();
11         persona.add("Pepe");
12         persona.add("Ana");
13         persona.add("Laura");
14         persona.add("Maria");
15
16         System.out.println("Numero de elementos"+persona.size());
17         for(String p: persona){
18             System.out.println(p);
19         }
20     }
21
22 }
23
```

Ilustración 5: Lista enlazada (1)

Output - Estructura_de_datos (run) | ListasEnlazadas.j

```
run:
Numero de elementos4
Pepe
Ana
Laura
Maria
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 6: Resultado lista enlazada

```java
package Colecciones;
//LinkedList
//almacena un nodo

import java.util.*;

public class ListasEnlazadas {

    public static void main(String[] args) {
        LinkedList<String> persona = new LinkedList<String>();
        persona.add("Pepe");
        persona.add("Ana");
        persona.add("Laura");
        persona.add("Maria");

        System.out.println("Numero de elementos" + persona.size());

        ListIterator<String> it = persona.listIterator();
        //se desplaza una posición hacia adelante
        it.next();
        it.add("Axel");


        for (String p : persona) {
            System.out.println(p);
        }
    }

}
```

Ilustración 7: Lista enlazada (2)

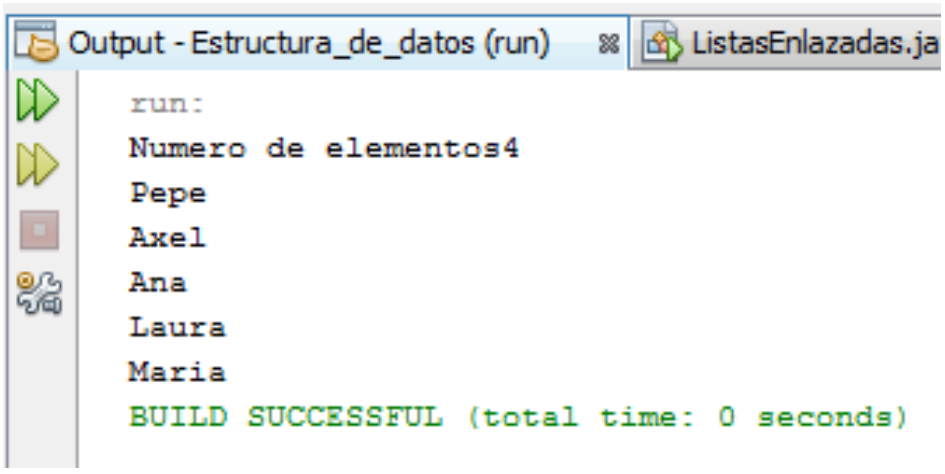

Ilustración 8: Resultado lista enlazada (2)

1.3.Colas (FIFO)

Torres (2013) define: En Programación, se le llama “Cola” al Tipo de Dato Abstracto que es una Lista en la que sus elementos se introducen (Encolan) únicamente por un extremo que le llamamos “Final de la Cola” y se remueven (Desencolan) únicamente por el extremo contrario al que le llamamos “Frente de la Cola” o “Principio de la Cola”.

FIFO -> Primero en entrar – primero en salir

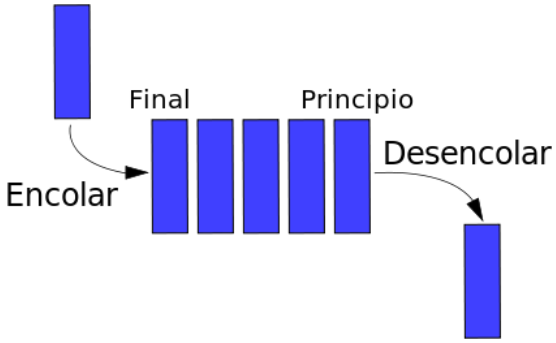

Ilustración 9: Cola

En Java podemos encontrar variadas formas de crear Colas en pocas líneas de código, un ejemplo es una de sus Interfaces que tiene como nombre “Queue”

```
public interface Queue<E>
extends Collection<E>
```

Métodos de Queue para manejo de Colas en Java, con la cual podemos crear Colas y que contiene los siguientes métodos para el uso de las mismas.

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Como podemos darnos cuenta, hay 2 métodos para cada operación, estos realizan de igual forma las operaciones, pero en cierto caso 1 de ellos retornará un valor especial "null" y el otro lanzará una excepción.

INSERTAR		
boolean	offer(E e)	Inserta el elemento especificado en esta cola si es posible hacerlo inmediatamente sin violar las restricciones de capacidad.
boolean	add(E e)	Inserta el elemento especificado en esta cola si es posible hacerlo inmediatamente sin violar las restricciones de capacidad, devuelve verdadero en caso de éxito y arroja una IllegalStateException si no hay espacio disponible actualmente.

Tabla 1: Insertar (Colas)

Eliminar		
E	Remove()	Recupera y elimina el encabezado de esta cola.
E	Poll()	Recupera y elimina el encabezado de esta cola, o devuelve nulo si esta cola está vacía.

Tabla 2: Eliminar (Colas)

Recuperar

E	peek()	Recupera, pero no elimina, el encabezado de esta cola, o devuelve nulo si esta cola está vacía.

Tabla 3: Recuperar (Colas)

Ejemplo:

```
package Colas_FIFO;

import java.util.LinkedList;
import java.util.Queue;

public class A_colas_Queue {

    public static void main(String[] args) {
        Queue<Integer> cola = new LinkedList<Integer>();
        cola.offer(3);
        cola.offer(5);
        cola.offer(7);
        cola.offer(9);
        cola.add(11);
        cola.offer(13);
        System.out.println("Cola llena" + cola);

        while (cola.peek() != null) {
            System.out.print("Elimino: \t");
            System.out.println(cola.poll());
            System.out.println("Siguiente frente= " + cola.peek());
        }
        System.out.println("Cola =" + cola.peek());
    }

}
```

Ilustración 10: Ejemplo Cola (1)

```
while (cola.remove()!=null) {
        System.out.println("Cola =" + cola.peek());

}
```

Ilustración 11: Recorrer una cola

```
run:
Cola llena[3, 5, 7, 9, 11, 13]
Cola =5
Exception in thread "main" java.util.NoSuchElementException
Cola =7
Cola =9
Cola =11
Cola =13
Cola =null
        at java.util.LinkedList.removeFirst(LinkedList.java:270)
        at java.util.LinkedList.remove(LinkedList.java:685)
        at Colas_FIFO.A_colas_Queue.main(A_colas_Queue.java:25)
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ilustración 12: Resultado

Conclusión:

Creando la cola con la interfaz “Queue” nos limitamos a tener elementos de 1 solo tipo de dato, en cambio, podemos crearla con la clase llamada “LinkedList”, en ella podemos utilizar los mismos métodos que creándola con la interfaz “Queue” y a parte podemos insertar elementos de diferentes tipos de datos en la misma Cola:

```java
package Colas_FIFO;

import java.util.LinkedList;

public class B_colas_linkedList {

    public static void main(String[] args) {
        LinkedList cola = new LinkedList();
        cola.offer(2);
        cola.add(4);
        cola.add("usuario 1");
        cola.offer(6);
        cola.add(8.6);
        cola.add("usuario 2");
        cola.offer(10);
        cola.add(10.5);
        cola.add("Elaborado por: YC");

        System.out.println("Cola llena: " + cola);

        while (cola.peek()!=null) {
            System.out.println("Elimino: \t"+cola.poll());
            System.out.println("Siguiente elemento: "+cola.peek());
        }
        System.out.println("Estado final de la cola: "+cola.peek());
    }
}
```

Ilustración 13: Ejemplo Cola (2)

1.4.Pilas

Acosta (2015), menciona lo siguiente acerca de pilas:

Una pila (stack en inglés) es una estructura de datos lineal que solo tienen un único punto de acceso fijo por el cual se añaden, eliminan o se consultan elementos. El modo de acceso a los elementos es de tipo LIFO (del inglés Last In First Out, último en entrar, primero en salir).

1.4.1. Características:

- La única forma de acceder a los elementos es desde el tope de la pila.

- Su administración es muy sencilla ya que tiene pocas operaciones.
- Si la pila está vacía no tiene sentido referirse a un tope ni a un fondo.
- En caso de querer acceder a un elemento que nos e encuentre en el tope de la pila se debe realizar un volcado de la pila a una pila auxiliar, una vez realizada la operación con el elemento se vuelve a volcar los elementos de la pila auxiliar a la original.

1.4.2. Operaciones básicas

apilar (valor): también conocido como push agrega el valor al tope de la pila.

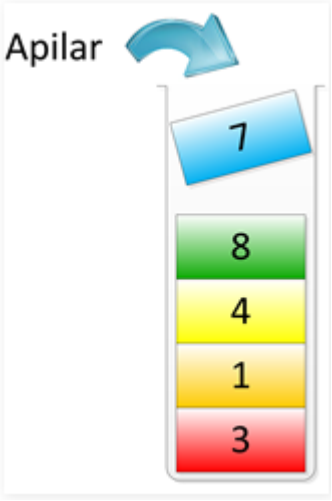

Ilustración 14: Pila - Apilar

retirar (): también conocido como pop retira el último elemento apilado.

Ilustración 15: Pila – Des apilar

cima (): devuelve el valor del elemento que está en la cima de la pila.

esVacia (): retorna true si la pila no ha sido inicializada.

Vaciar: vacía todo el contenido de la Pila B en la Pila, dejando a B vacía.

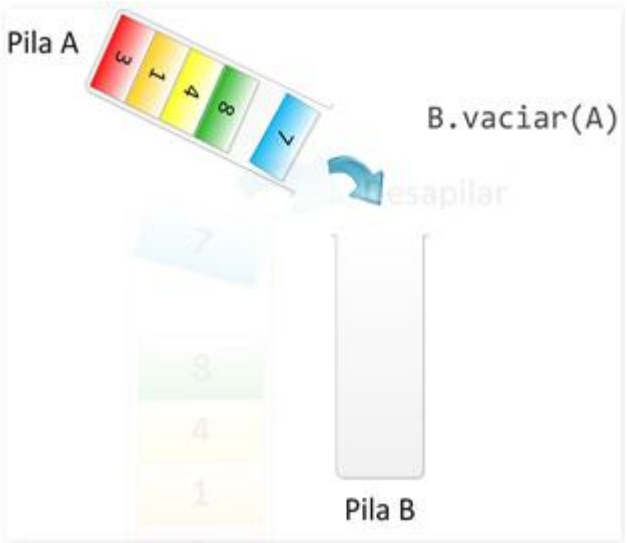

Ilustración 16: Pila - Vaciar

buscar (valor): retorna la true si el elemento a buscar existe en la pila.

eliminar(): elimina la pila

listar (): imprime en pantalla los elementos de la pila.

Ejemplos:

```
import java.util.Stack;

public class Main {

    public static void main(String[] args) {

        Stack<Libro> pila = new Stack<Libro>();

        Libro l1 = new Libro("Título 1", "Autor 1");
        Libro l2 = new Libro("Título 2", "Autor 2");
        Libro l3 = new Libro("Título 3", "Autor 3");

        pila.push(l1); // adiciona un libro a la pila
        pila.push(l2);
        pila.push(l3);

        System.out.println(pila.peek().getTitulo()); // el último elemento adicionado

        while (!pila.isEmpty()) { // mostrar pila completa
            System.out.println(pila.pop().getTitulo()); // extrae un elemento de la pila
        }

    }
}
```

Ilustración 17: Clase principal

```java
public class Libro {

    private String titulo;
    private String autor;

    public Libro() {
        this.titulo = "";
        this.autor = "";
    }

    public Libro(String titulo, String autor) {
        this.titulo = titulo;
        this.autor = autor;
    }

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public String getAutor() {
        return autor;
    }

    public void setAutor(String autor) {
        this.autor = autor;
    }
}
```

Ilustración 18: Clase Libro

Ejemplo 2:

```java
package estructurasDeDatos;

public class Pila {
 private int[] pila;
 private int top;
 private int tamanioPila;

 public Pila(){
  tamanioPila = 1;
  top = -1;
  pila = new int[tamanioPila];
 }

 //las operaciones basicas de la pila
 public void push(int dato){
  if (top == (tamanioPila -1))
   redimensionar();
  pila[++top] = dato;
 }

 public Integer pop(){
  if(top < 0)
   return null;
  return pila[top--];
 }

 private void redimensionar(){
  int[] temp = pila;
  tamanioPila *= 2;
  pila = new int[tamanioPila];

  for(int i = 0; i <= top; i++)
   pila[i] = temp[i];
 }
}
```

Ilustración 19: Ejemplo Pilas (2)

Fuente: Aprendiendo – Programación estructurada

```java
package estructurasDeDatos;

public class PilaPrueba {

 public static void main(String[] args) {
  Pila pila = new Pila();

  pila.push(10);
  pila.push(20);
  pila.push(30);

  System.out.println(pila.pop());
  System.out.println(pila.pop());
  System.out.println(pila.pop());
  System.out.println(pila.pop());
 }

}
```

Ilustración 20: Clase principal - Ejemplo pilas (2)

Fuente: Aprendiendo – Programación estructurada

1.5.Arboles

Torres (2016), nos dice:

Un árbol se define como una colección de nodos donde cada uno además de almacenar información, guarda las direcciones de sus sucesores.

Los árboles representan las estructuras **no-lineales** y **dinámicas** de datos más importantes en computación.

Dinámicas, puesto que la estructura árbol puede cambiar durante la ejecución de un programa.

No- lineal puesto que a cada elemento del árbol pueden seguirle varios elementos.

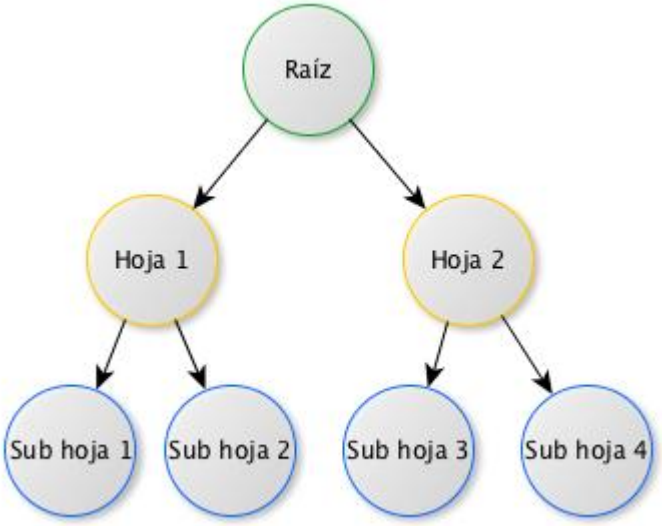

Ilustración 21: Árbol

1.5.1. Características de los árboles

- **Hijo**: Es aquel nodo que siempre va a tener un nodo antecesor o padre, son aquellos que se encuentran en el mismo nivel
- **Padre**: Es aquel que tiene hijos y también puede tener o no antecesores.
- **Hermano**: Dos nodos son hermanos si son apuntados por el mismo nodo, es decir si tienen el mismo padre.

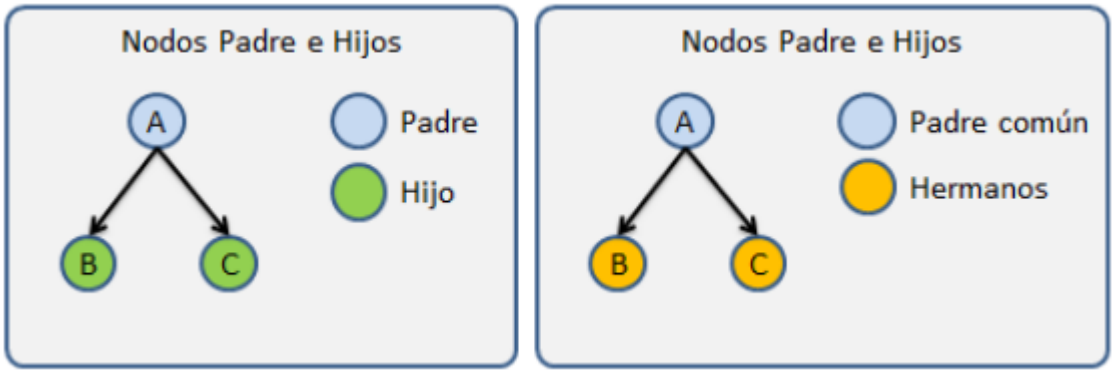

Ilustración 22: Nodos Padre, Hijo y Hermanos

- **Raíz**: Es el nodo principal de un árbol y no tiene antecesores.
- **Hoja** o terminal: Son aquellos nodos que no tienen hijos o también los nodos finales de un árbol.

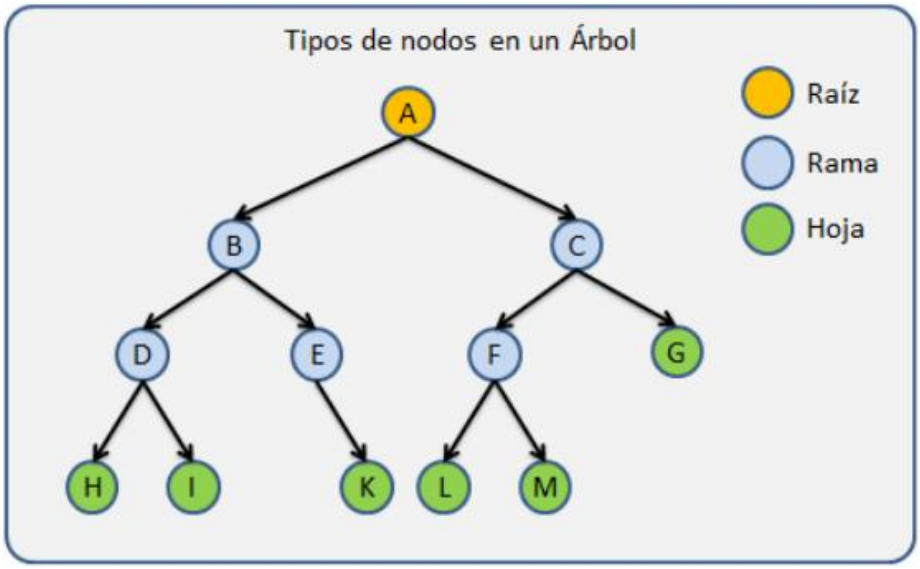

Ilustración 23: Tipos de nodos en un árbol

- **Interior**: Se dice que un nodo es interior si no es raíz ni hoja.
- **Nivel de un nodo**: Se dice que el nivel de un nodo es el número de arcos que deben ser recorridos, partiendo de la raíz para llegar hasta él.
- **Altura del árbol**: Se dice que la altura de un árbol es el máximo de los niveles considerando todos sus nodos.
- **Grado de un nodo**: se dice que el grado de un nodo es el número de hijos que tiene dicho nodo.

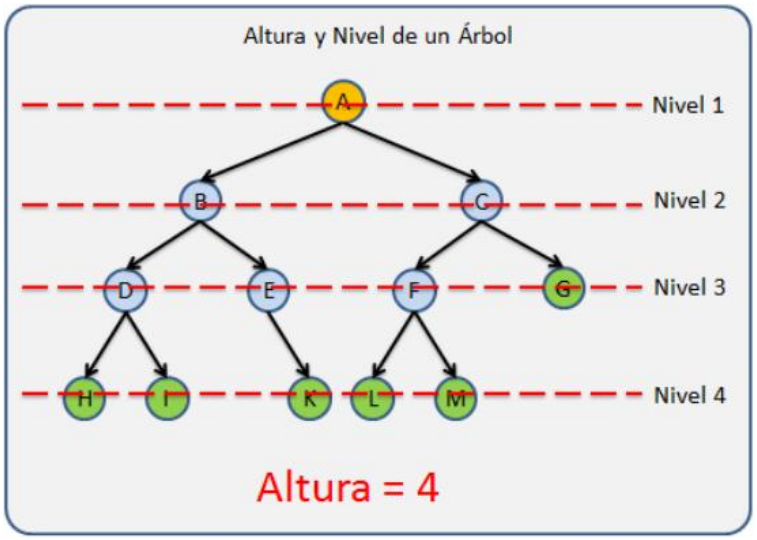

Ilustración 24: Altura y nivel - árbol

- Conocemos como **peso** al número de nodos que tiene un Árbol. Este factor es importante porque nos da una idea del tamaño del árbol y el tamaño en memoria que nos puede ocupar en tiempo de ejecución.

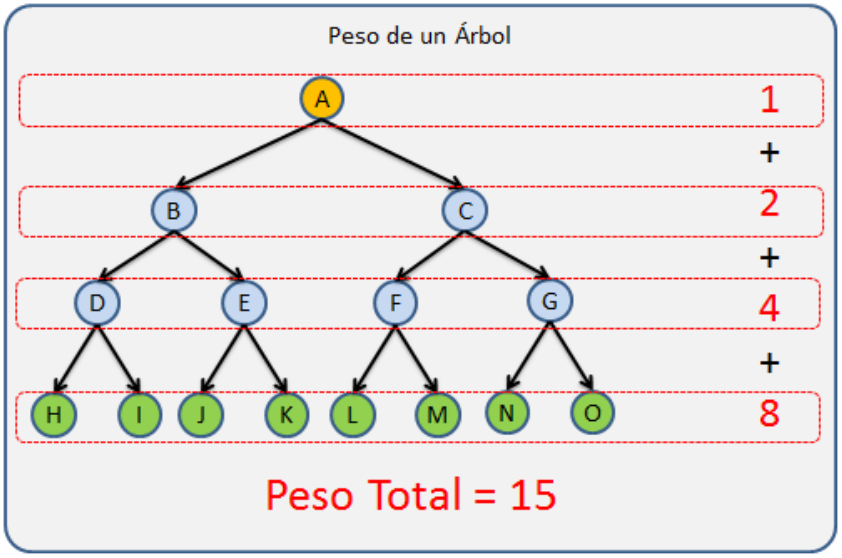

Ilustración 25: Peso de un árbol

1.5.2. Tipos de Árboles

- **Árboles Binarios:** Un árbol binario es un conjunto finito de elementos, el cual está vacío o dividido en tres subconjuntos separados: raíz del árbol, subárbol izquierdo y subárbol derecho
- **Árbol de búsqueda binario auto-balanceable:** Es el que intenta mantener su *altura*, o el número de niveles de nodos bajo la raíz, tan pequeños como sea posible en todo momento, automáticamente
- **Árboles AVL:** están siempre equilibrados de tal modo que para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa.
- **Árboles Rojo-Negro:** Un árbol rojo-negro es un árbol binario de búsqueda en el que cada nodo tiene un atributo de color cuyo valor es **rojo** o **negro**.
- **Árboles AA:** utilizado para almacenar y recuperar información ordenada de manera eficiente
- **Árbol de segmento:** es una estructura de datos en forma de árbol para guardar intervalos o segmentos. Permite consultar cuál de los segmentos guardados contiene un punto.
- **Árboles Multicamino:** es un árbol ordenado cuyos nodos deben tener un número específico de hijos.
- **Árboles B:** Es un árbol de búsqueda que puede estar vacío o aquel cuyos nodos pueden tener varios hijos, existiendo una relación de orden entre ellos.

1.5.3. Recorridos de Árboles

Pre-orden:

1. Visitar la **Raíz**
2. Recorrer el sub-árbol **izquierdo**
3. Recorrer el sub-árbol **derecho**

In-orden:

1. Recorrer el sub-árbol **izquierdo**
2. Visitar la **raíz**
3. Recorrer el sub-árbol **derecho**

Post-orden:

1. Recorrer el sub-árbol **izquierdo**
2. Recorrer el sub-árbol **derecho**
3. Visitar la **raíz**

Ejemplo de Código de Árboles en Java

```java
package pruebaarbol;
import pruebaarbol.model.Nodo;

/**
*
* @author Daniel
*/
public class Pruebaarbol {

  /**
  * @param args the command line arguments
  */
  public static void main(String[] args) {

    //Se asignan los valores a los nodos = 1,2,3,4,5 y 6 dentro del arbol
    //Nodo raiz = 1
    Nodo raiz = new Nodo(1);

    //Nodo2 Izquierdo = 2
    Nodo nodo2 = new Nodo(2);

    //Nodo3 Derecho = 3
    Nodo nodo3 = new Nodo(3);

    //Se asigna el valor 6 al nodo que sera hijo del nodo 3 a la derecha
    nodo3.setNodoDerecho(new Nodo(6));

    //Se asigna el valor 5 al nodo que sera hijo del nodo 3 a la izquierda
    nodo3.setNodoIzquierdo(new Nodo(5));

    //Se asigna el valor 4 al nodo que sera hijo del nodo 4 a la izquierda
    nodo2.setNodoIzquierdo(new Nodo(4));
```

```
32
33      //Se crean el Nodo 2 a la izquierda y el Nodo 3 a la derecha de la raiz
34      raiz.setNodoIzquierdo(nodo2);
35      raiz.setNodoDerecho(nodo3);
36
37      //Resultado en pantalla
38      System.out.println("Recorrido Preorden: ");
39      preOrden(raiz);
40      System.out.println("Recorrido Inorden: ");
41      inorden(raiz);
42      System.out.println("Recorrido PostOrden: ");
43      posOrden(raiz);
44    }
45
46    //Metodo Preorden
47    private static void preOrden(Nodo raiz) {
48      if (raiz != null) {
49        System.out.print(raiz.getDato() + " - ");
50        preOrden(raiz.getNodoIzquierdo());
51        preOrden(raiz.getNodoDerecho());
52      }
53    }
54
55    //Metodo Inorden
56    private static void inorden(Nodo raiz) {
57      if (raiz != null) {
58        inorden(raiz.getNodoIzquierdo());
59        System.out.print(raiz.getDato()+ " - ");
60        inorden(raiz.getNodoDerecho());
61      }
```

```
62
63    //Metodo PostOrden
64    private static void posOrden(Nodo raiz) {
65      if (raiz != null) {
66        posOrden(raiz.getNodoIzquierdo());
67        posOrden(raiz.getNodoDerecho());
68        System.out.print(raiz.getDato() + " - ");
69      }
70    }
71  }
72
73
74
```

Ilustración 26: Ejemplo árboles

Fuente: Arboles en Java (Micheller Torres)

```java
package pruebaarbol.model;

/**
*
* @author Daniel
*/
public class Nodo {
  //Variables
  private int dato;
  private Nodo izq;
  private Nodo der;

  //Constructor
  public Nodo(int dato){
    this.dato = dato;
  }
  //Para saber el nodo izquierdo
  public Nodo getNodoIzquierdo(){
    return izq;
  }
  //Para saber el nodo derecho
  public Nodo getNodoDerecho(){
    return der;
  }
  //Se crea nodo derecho
  public void setNodoDerecho(Nodo nodo){
    der = nodo;
  }
  //Se crea nodo Izquierdo
  public void setNodoIzquierdo(Nodo nodo){
    izq = nodo;
  }

  public int getDato(){
    return dato;
  }
}
```

Ilustración 27: Clase Nodo

UNIDAD 2: JAVA I/O

2.1. Manejo de archivos

"Un **archivo** no es más que una estructura que permite almacenar información en disco y poderla recuperar en el tiempo" (Largo, 2016a).

A diferencia de otras estructuras en Java como arreglos, matrices, pilas, colas, listas, que solo existen en memoria durante la duración del programa, **un archivo se almacena en disco** (disco duro, disco externo, usb etc) y permanece en el tiempo pudiendo luego ser modificado.

Las operaciones más básicas para manejar archivos como leer y escribir se lo hace a través de flujos, un flujo no es más que un **canal de comunicación** por el que se envía información (conjunto de bytes) al archivo.

Para poder establecer un flujo o canal mediante el cual podamos escribir datos en un archivo o leer datos de un archivo, se utiliza objetos stream, las clases que se utiliza para crear estos objetos específicamente heredan de las clases principales **InputStream** y **OutputStream**.

En Java existen varias clases que implementan métodos para la lectura y escritura de archivos, y que lo hacen a **niveles muy bajos que son bytes, y a niveles más altos como son caracteres, cadenas y hasta objetos.**

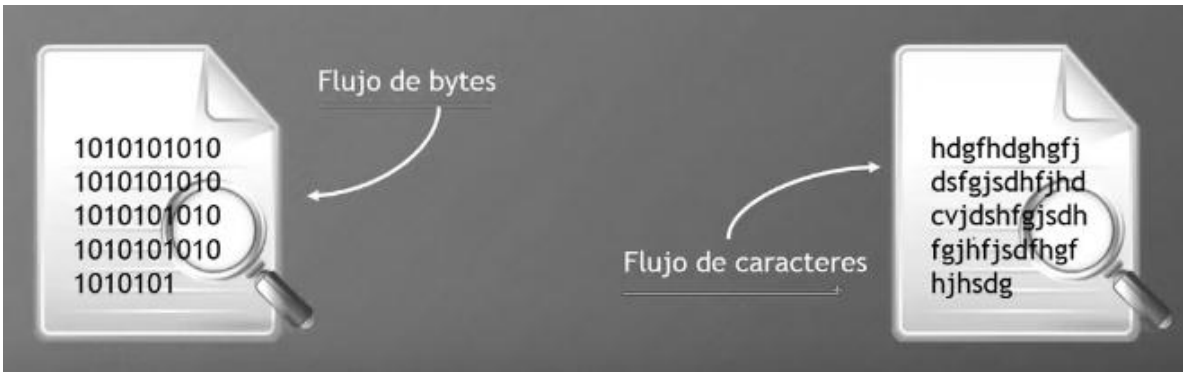

Ilustración 28: Flujos

Para traer la información, un programa abre un stream sobre una fuente de información (un fichero, memoria, un socket) y lee la información, de esta forma:

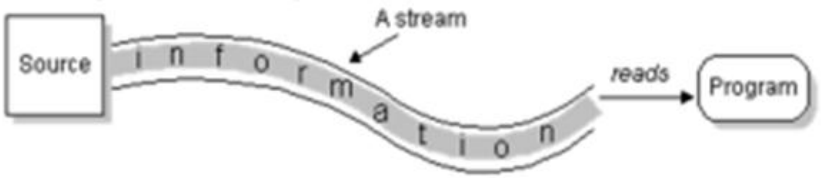

Ilustración 29: Traer información

De igual forma, un programa puede enviar información a un destino externo abriendo un stream sobre un destino y escribiendo la información en este, de esta forma:

Ilustración 30: Enviar información

Los algoritmos para leer y escribir:

abrir un stream

mientras haya información

leer o escribir información

cerrar el stream

2.2. LIBRERÍA I/O

El paquete java.io contiene una colección de clases stream que soportan estos algoritmos para leer y escribir. Estas clases están divididas en dos árboles basándose en los tipos de datos (**caracteres o bytes**) sobre los que opera.

2.2.1. Streams de caracteres

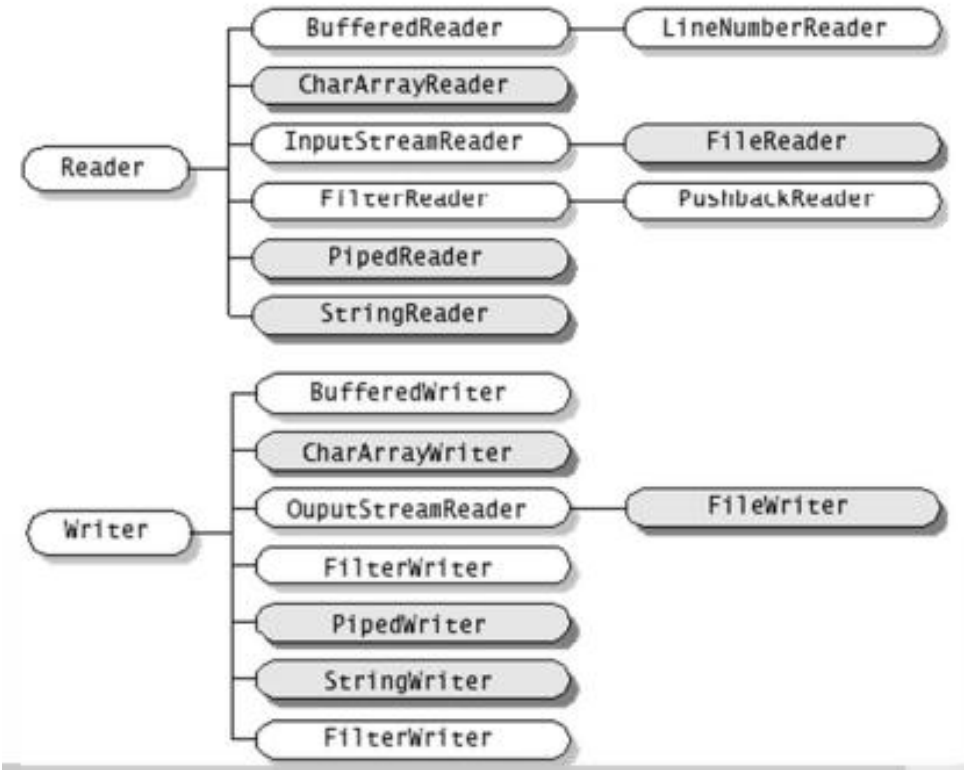

Ilustración 31: Streams de caracteres

Reader y Writer son las superclases abstractas para streams de caracteres en java.io. Reader proporciona el API y una implementación para readers (streams que leen caracteres de 16-bits) y

Writer proporciona el API y una implementación para writers (streams que escriben caracteres de 16-bits).

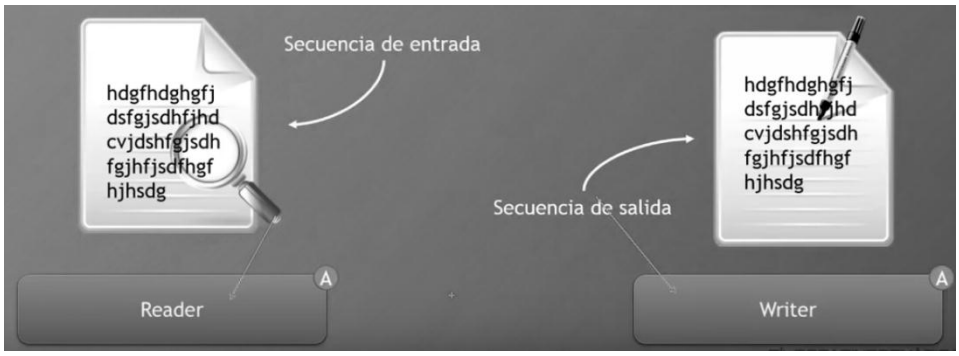

Ilustración 32: Reader y Writer

Ejemplo: Acceso a ficheros (file)

escribir.java

```
1  package file;
2  import java.io.FileWriter;
3  import java.io.IOException;
4  import java.util.logging.Level;
5  import java.util.logging.Logger;
6  /* * * @author YEC*/
7  public class escribir {
8      public static void main(String[] args) {
9          escribir_Fichero ef=new escribir_Fichero();
10         ef.escribir();
11     }
12 }
13 class escribir_Fichero {
14     public void escribir() {
15         String frase = "Yecenia Cevallos - POO - ISTJ";
16         try {
17             FileWriter escritura = new FileWriter("C:/ficheros_java/fichero.txt", true);
18             for (int i = 0; i < frase.length(); i++) {
19                 escritura.write(frase.charAt(i));
20
21             }
22             escritura.close();
23
24         } catch (IOException ex) {
25             Logger.getLogger(escribir_Fichero.class.getName()).log(Level.SEVERE, null, ex);
26         }
27     }
28
29 }
30
```

Ilustración 33: Ejemplo FileWriter

leer.java

Source | History

```
1  package file;
2  import java.io.*;
3
4  public class leer {
5      public static void main(String[] args) {
6          leer_fichero lf= new leer_fichero();
7          lf.leer();
8      }
9  }
10 class leer_fichero {
11         public void leer() {
12             try {
13                 FileReader lectura = new FileReader("C:/ficheros_java/fichero.txt");
14                 int c = 0;
15
16                 while (c != -1) {
17                     c = lectura.read();
18                     char letra=(char) c;
19                     System.out.print(letra);
20                 }
21
22                 lectura.close();
23
24             } catch (IOException ex) {
25                 System.out.println("No se encontro el archivo" + ex);
26             }
27         }
28     }
```

Ilustración 34: Ejemplo FileReader

2.2.2. Streams de bytes

Los programas deberían usar los streams de bytes, descendientes de InputStream y OutputStream, para leer y escribir bytes de 8-bits. Estos streams se usan normalmente para leer y escribir datos binarios como imágenes y sonidos.

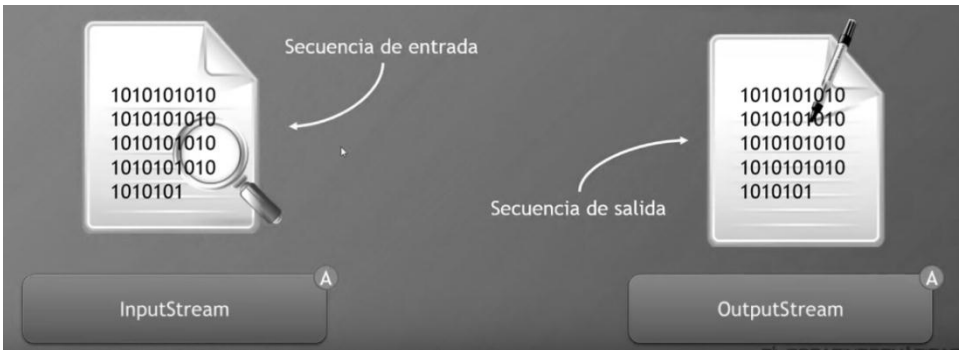

Ilustración 35: Streams de bytes

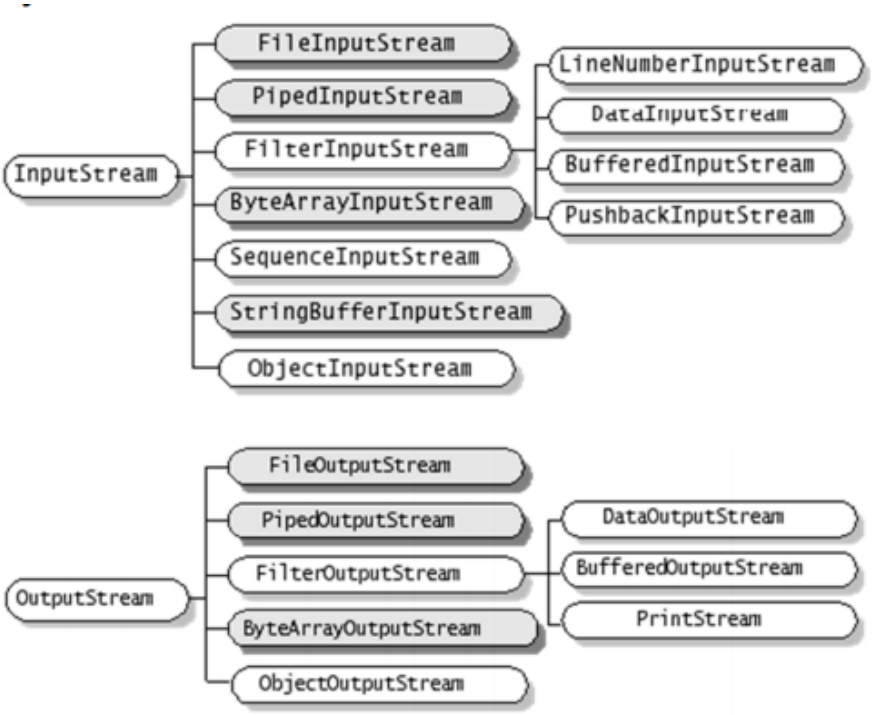

Ilustración 36: Clasificación de Streams de bytes

BufferedReader

Esta clase tiene el método readLine() para leer una línea de texto a la vez. Sin embargo, BufferedReader no puede usarse por sí mismo, debemos envolver un Reader dentro de un BufferedReader. Proporciona un buffer de almacenamiento temporal

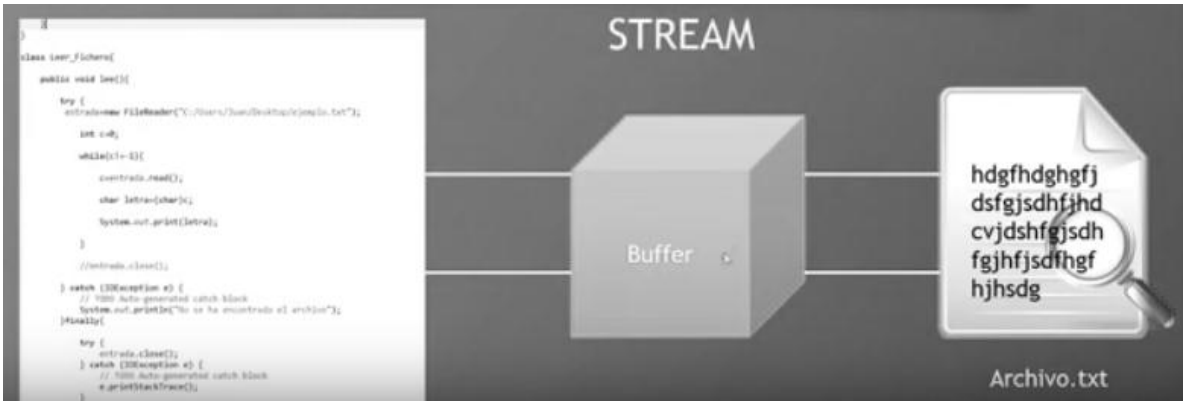

Ilustración 37: BufferedReader

BufferedWriter

Proporciona un buffer de almacenamiento temporal. Los datos se escriben en el buffer, y cuando éste se llena, los contenidos se escriben a un fichero, lo que reduce el número de veces que se tiene que acceder al disco o a la red.

Un objeto BufferedWriter envuelve un objeto FileWriter, entonces se llama al método write para escribir los datos en el fichero especificado. Es importante utilizar los métodos flush y close para vaciar el buffer y cerrar el fichero para liberar recursos.

Acceso a ficheros (buffered)

leer_archivo.java

Source | History

```
1  package bufferedReader;
2
3  import java.io.*;
4
5  public class leer_archivo {
6
7      public static void main(String[] args) {
8
9          try {
10             String cad="";
11             BufferedReader br = new BufferedReader(new FileReader("C:/ficheros_java/fichero1.txt"));
12             while ((cad = br.readLine()) != null) {
13                 System.out.println(cad);
14             }
15             br.close();
16         } catch (IOException ioe) {
17             System.out.println(ioe);
18         }
19
20     }
21
22 }
```

Ilustración 38: Ejemplo BufferedReader

escribir_archivo.java

Source | History

```
1  package bufferedReader;
2
3  import java.io.*;
4
5  public class escribir_archivo {
6
7      public static void main(String[] args) {
8          try {
9              String cad1 = "Cadena 1.\n";
10             String cad2 = "Cadena 2.\n";
11             String cad3 = "Campaña mamá.\n";
12             BufferedWriter bw = new BufferedWriter(new FileWriter("C:/ficheros_java/fichero1.txt",true));
13             bw.write(cad1);
14             bw.newLine();
15             bw.write(cad2);
16             bw.newLine();
17              bw.write(cad3);
18             bw.newLine();
19             bw.flush();
20             bw.close();
21
22         } catch (IOException ex) {
23             System.out.println("ERROR" + ex);
24         }
25     }
26
27 }
```

Ilustración 39: Ejemplo BufferedWriter

2.2.3. Guardar objetos en ficheros en java

Cuando ejecutamos una aplicación OO lo normal es crear múltiples instancias de las clases que tengamos definidas en el sistema. Cuando cerramos esta aplicación todos los objetos que tengamos en memoria se pierden.

Para solucionar este problema los lenguajes de POO nos proporcionan unos mecanismos especiales para poder guardar y recuperar el estado de un objeto y de esa manera poder utilizarlo como si no lo hubiéramos eliminado de la memoria. Este tipo de mecanismos se conoce como **persistencia de los objetos**.

En Java hay que implementar una interfaz y utilizar dos clases: – Interfaz Serializable (interfaz vacía, no hay que implementar ningún método)

- Streams: ObjectOutputStream y ObjectInputStream.

ObjectOutputStream y ObjectInputStream, permiten leer y escribir grafos de objetos, es decir, escribir y leer los bytes que representan al objeto. El proceso de transformación de un objeto en un stream de bytes se denomina serialización.

Los objetos ObjectOutputStream y ObjectInputStream deben ser almacenados en ficheros, para hacerlo utilizaremos los streams de bytes **FileOutputStream y FileInputStream**, ficheros de acceso secuencial.

Ejemplo:

Persona.java

Source | History

```
1   package objectStrem;
2
3   import java.io.*;
4
5   public class Persona implements Serializable {
6
7       private String cedula, nombre, apellido, lugar;
8       private int edad;
9
10      public Persona(String cedula, String nombre, String apellido, String lugar, int edad) {
11          this.cedula = cedula;
12          this.nombre = nombre;
13          this.apellido = apellido;
14          this.lugar = lugar;
15          this.edad = edad;
16      }
17
18      public String getCedula() {return cedula;}
19      public String getNombre() {return nombre;}
20      public String getApellido() {return apellido;}
21      public String getLugar() {return lugar;}
22      public int getEdad() {return edad;}
23
24      public String toString() {
25          return "Nombre: " + nombre + " Apellido: " + apellido
26                  + " Lugar:" + lugar + " Edad: " + edad + "\n";
27      }
28  }
```

Ilustración 40: Acceso a ficheros (objectStrem) 1

Escribir_objeto.java

Source | History

```
1  package objectStrem;
2
3  import java.io.*;
4  import java.util.HashMap;
5
6  public class Escribir_objeto {
7
8      public static void main(String[] args) {
9          Persona p1 = new Persona("123", "Yecenia", "Cevallos", "Quito", 30);
10         Persona p2 = new Persona("456", "Vanessa", "Calva", "Loja", 30);
11         Persona p3 = new Persona("789", "Axel", "Castillo", "Azuay", 30);
12         Persona p4 = new Persona("012", "Andrea", "Cabrera", "Loja", 30);
13
           HashMap<String, Persona> pers = new HashMap<String, Persona>();
15         pers.put(p1.getCedula(), p1);
16         pers.put(p2.getCedula(), p2);
17         pers.put(p3.getCedula(), p3);
18         pers.put(p4.getCedula(), p4);
19
20         try {
21             FileOutputStream ruta = new FileOutputStream("C:/ficheros_java/fichero_objetos.txt");
               ObjectOutputStream oos = new ObjectOutputStream(ruta);
23             oos.writeObject(pers);
24             oos.close();
25         } catch (Exception ex) {
26             System.out.println(ex.toString());
27         }
28     }
29 }
```

Ilustración 41: Acceso a ficheros (objectStrem) 2

leer_fichero.java

Source | History

```
1  package objectStrem;
2
3  import java.io.*;
4  import java.util.HashMap;
5
6  public class leer_fichero {
7
8      public static void main(String[] args) {
9          try {
10             HashMap<String, Persona> pers = new HashMap<String, Persona>();
11             FileInputStream ruta = new FileInputStream("C:/ficheros_java/fichero_objetos.txt");
12
13             ObjectInputStream leerPersona = new ObjectInputStream(ruta);
14             try {
15                 while (true) {
16                     pers =  (HashMap) leerPersona.readObject();
17                     System.out.println(pers);
18                     //leer.close();
19                 }
20             } catch (EOFException e) {
21                 System.out.println("Lectura finalizada");
22             }
23             leerPersona.close();
24         } catch (Exception ex) {
25             System.out.println(ex);
26         }
27     }
28 }
```

Ilustración 42: Acceso a ficheros (objectStrem) 3

2.3. Conexiones a base de datos.

Para realizar la conexión a base de datos desde java necesitamos hacer uso de Java Database Connectivity (JDBC). Esta es una interface de acceso a bases de datos estándar SQL que proporciona un acceso uniforme a una gran variedad de bases de datos relacionales. JDBC también proporciona una base común para la construcción de herramientas y utilidades de alto nivel.

Las bases de datos que deseemos conectar deberán proveernos el driver JDBC en un empaquetado JAR para añadirlo a las librerías del proyecto, por ejemplo en NetBeans se podría ir al directorio Libraries del Proyecto, hacer clic derecho sobre él y elegir Add Library y en la lista podría encontrase la que necesitemos, o si queremos agregarla manualmente Add JAR/Folder y seleccionar desde la dirección donde lo tenemos almacenado.

En primer lugar, para realizar este ejemplo debemos tener instalado tres cosas en nuestro sistema:

- La máquina virtual de Java (Para ejecutar Java, claro está)
- MySQL (en mi caso yo tengo instalado Xampp que viene con Apache y MySQL, entre otros servicios
- Netbeans (Con Java instalado)

El driver que será el encargado de gestionar la conexión con la base de datos y existe uno para cada base de datos.

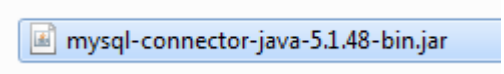

Ilustración 43: Conector Mysql

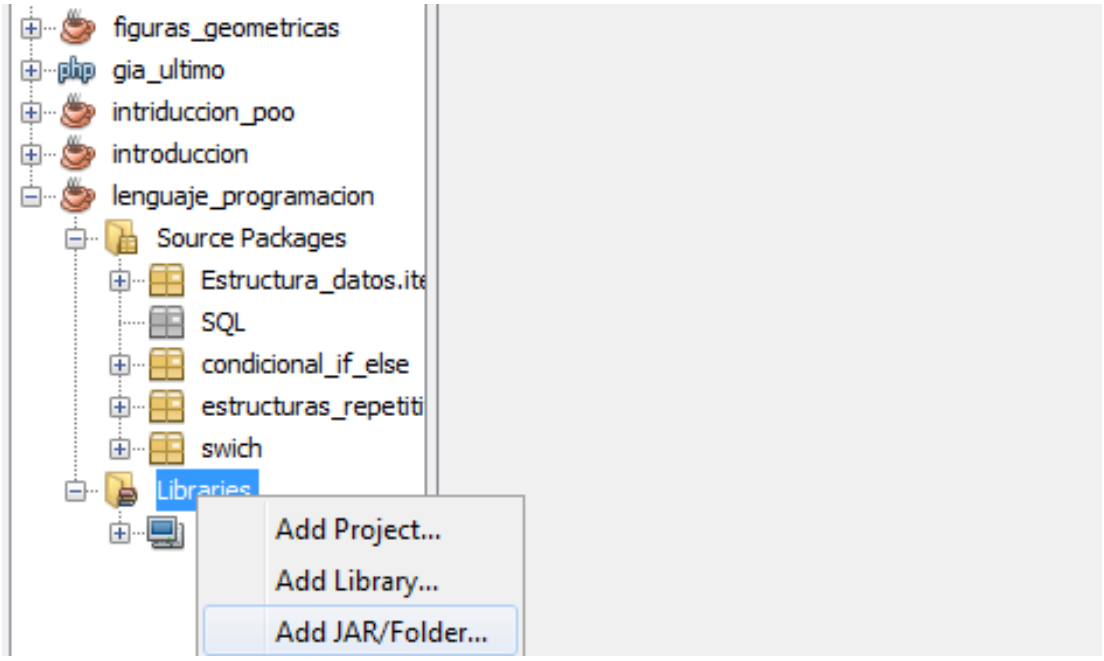

Ilustración 44:Añadir conector

Para implementar la conexión con la base de datos, vamos a crear una interfaz gráfica con un botón que permita verificar la conexión con la base de datos.

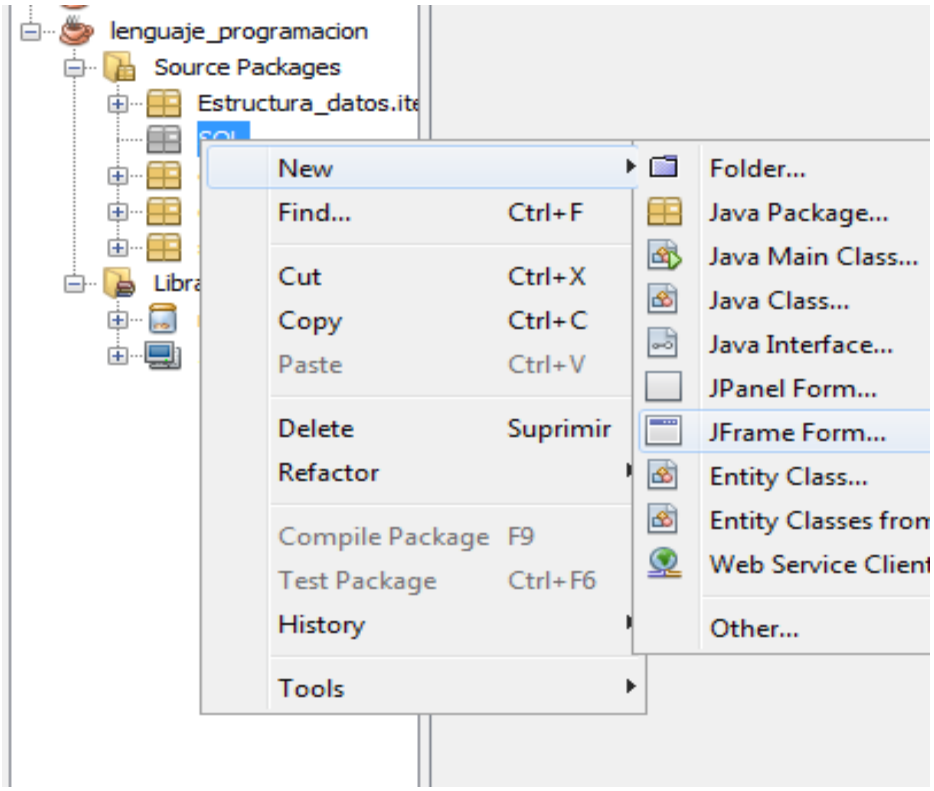

Ilustración 45: Crear el formulario

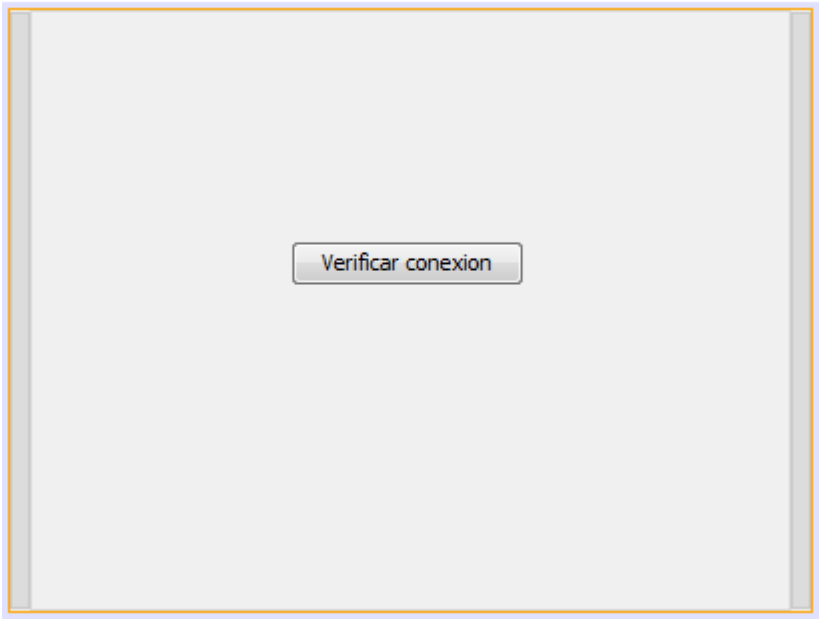

Ilustración 46: Formulario

Crearemos una base de datos utilizando SqlYog, la cual se llama **cliente, con los siguientes campos:**

	Field	Type	Comment
🔑	cedula	char(10) NOT NULL	
	apellidos	varchar(30) NOT NULL	
	nombres	varchar(30) NOT NULL	
	f_nacimiento	date NOT NULL	
	sexo	char(1) NOT NULL	
	correo	varchar(30) NOT NULL	

Ilustración 47: Campos base de datos

PARÁMETROS DE LA CADENA DE CONEXIÓN A LA BASE DE DATOS

Una cadena de conexión está compuesta de la siguiente manera:

jdbc:mysql://host:port/database?user=name_user&password=secret_password

En donde:

- **host**: es la IP donde está la base de datos MySQL, si está en nuestra misma máquina será localhost.
- **port**: es el puerto por donde escucha el servidor de base de datos, por defecto para MySQL es el 3306.
- **database**: es el nombre de la base de datos a la cual nos queremos conectar.
- **user**: es el usuario de la base de datos
- **password**: es la clave con que el usuario puede ingresar a la base de datos.

Con la url, el usuario y el password, creamos una conexión.

```
public class conexion extends javax.swing.JFrame {

    public static final String URL = "jdbc:mysql://localhost:3306/cliente";
    public static final String USUARIO = "root";
    public static final String CLAVE = "";
    Connection con;
```

Ilustración 48: Conexión a BDD

El método getconnection() de la clase drivermanager para la conexión

El método getConnection() es un método estático de la clase DriverManager por esa razón es que se accede sin crear un objeto, esta clase está disponible dentro del paquete java.sql, el cual debes importar para que no te de error.

Dado que el método getConnection() tiene sobrecarga, existen tres formas de conexión:

1. getConnection(String url)
2. Connection getConnection(String url, Properties info)
3. Connection getConnection(String url, String user, String password)

```
public Connection getConnection() {

    try {
        Class.forName("com.mysql.jdbc.Driver");
        con = (Connection) DriverManager.getConnection(URL, USUARIO, CLAVE);
        JOptionPane.showMessageDialog(null, "Conexion exitosa");
    } catch (Exception e) {
        System.out.println("ERROR:" + e);
    }
    return con;
}
```

Ilustración 49: Método getVConection

El resultado es un objeto Resulset que contiene los datos de nuestra consulta.

El botón del formulario escribimos el siguiente código:

```
private void btn_conexionActionPerformed(java.awt.event.ActionEvent evt) {
    try {

        con = getConnection();
        PreparedStatement ps;
        ResultSet rs;
        ps = con.prepareStatement("SELECT * FROM cliente");
        rs = ps.executeQuery();
        if (rs.next()) {
            JOptionPane.showMessageDialog(null, rs.getString("nombre"));

        } else {
            JOptionPane.showMessageDialog(null, "No existen datos");
        }
    } catch (Exception e) {
        System.out.println("ERROR"+e);
    }
}
```

Ilustración 50: Conexión a la base de datos

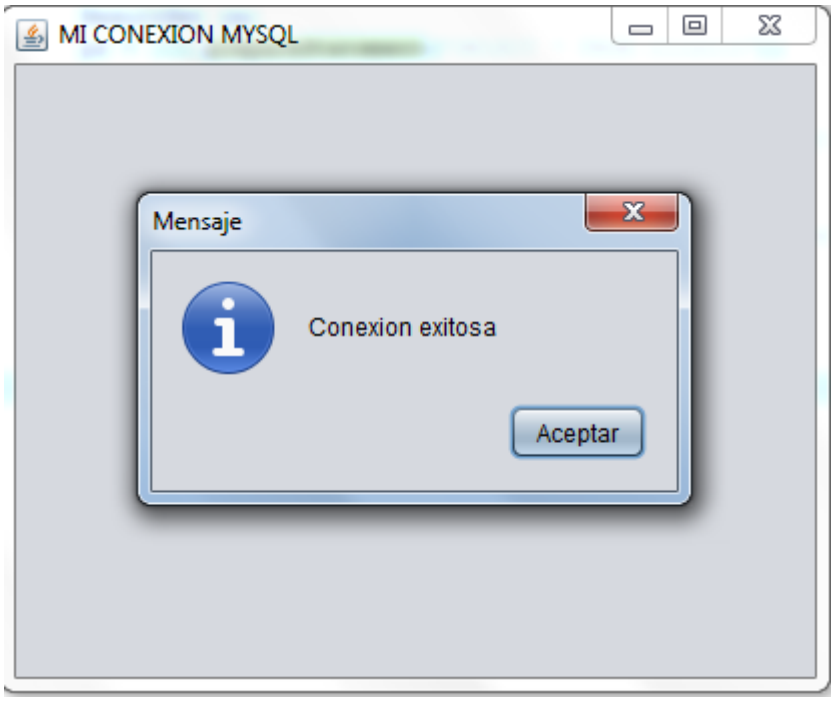

Ilustración 51: Conexión exitosa

Una vez verificada la conexión exitosa con la base de datos, realizaremos el CRUD (Crear, Leer, Actualizar, Eliminar) mediante un formulario:

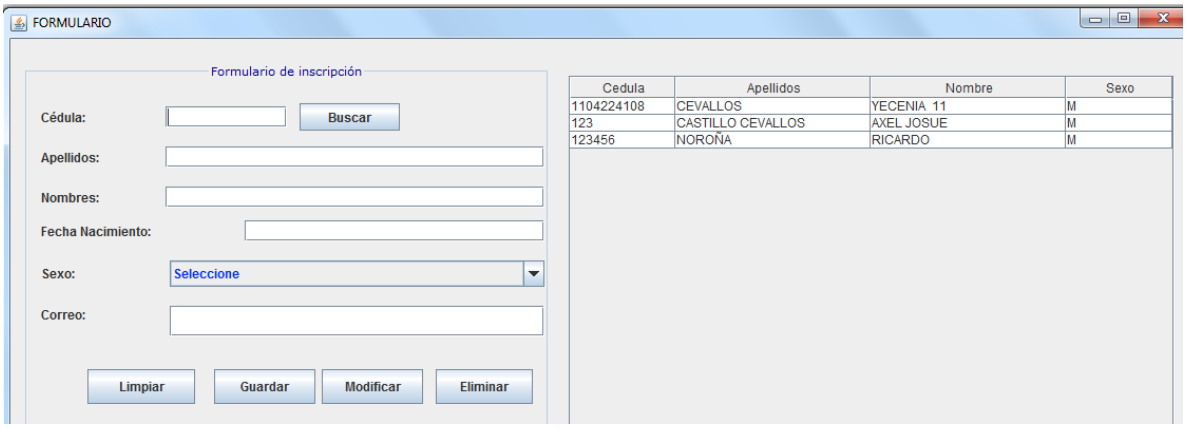

Ilustración 52: Formulario

Limpiar: Se crea ese método que permite limpiar los campos.

```java
public void limpiar() {
    txt_cedula.setText("");
    txt_apellidos.setText("");
    txt_nombres.setText("");
    txt_correo.setText("");
    txt_fechaN.setText("");
    cbx_sexo.setSelectedIndex(0);
}

private void btn_limpiarActionPerformed(java.awt.event.ActionEvent evt) {
    limpiar();
```

Ilustración 53: Limpiar

Guardar: Esta opción permite guardar en la base de datos

```java
private void btn_guardarActionPerformed(java.awt.event.ActionEvent evt) {
    con = null;
    try {
        con = getConnection();
        String buscar = "SELECT * FROM estudiantes WHERE cedula=?";
        ps = con.prepareStatement(buscar);
        ps.setString(1, txt_cedula.getText());
        rs = ps.executeQuery();
        if (rs.next()) {
            JOptionPane.showMessageDialog(null, "Registro existe");
        } else {
            String insertar = "INSERT INTO estudiantes(cedula,apellidos,nombres,f_nacimiento,sexo,correo) "
                    + "VALUES(?,?,?,?,?,?)";
            ps = con.prepareStatement(insertar);
            ps.setString(1, txt_cedula.getText());
            ps.setString(2, txt_apellidos.getText());
            ps.setString(3, txt_nombres.getText());
            ps.setDate(4, Date.valueOf(txt_fechaN.getText()));
            ps.setString(5, cbx_sexo.getSelectedItem().toString());
            ps.setString(6, txt_correo.getText());
            int resp = ps.executeUpdate();
            if (resp > 0) {
                JOptionPane.showMessageDialog(null, "Registro almacenado");
                limpiar();
                cargarTabla();
            } else {
                JOptionPane.showMessageDialog(null, "No se guardó");
            }
        }
        con.close();
    } catch (Exception e) {
        System.out.println("ERROR: " + e);
    }
}
```

Ilustración 54: Guardar

Opción modificar: Modifica los campos en la base de datos en base a un registro específico.

```
private void btn_modificarActionPerformed(java.awt.event.ActionEvent evt) {
    con = null;
    try {
        con = getConnection();
        String insertar = "UPDATE estudiantes SET apellidos=?,nombres=?,f_nacimiento=?,sexo=?,correo=? WHERE cedula=?";
        ps = con.prepareStatement(insertar);

        ps.setString(1, txt_apellidos.getText());
        ps.setString(2, txt_nombres.getText());
        ps.setDate(3, Date.valueOf(txt_fechaN.getText()));
        ps.setString(4, cbx_sexo.getSelectedItem().toString());
        ps.setString(5, txt_correo.getText());
        ps.setString(6, txt_cedula.getText());
        int resp = ps.executeUpdate();
        if (resp > 0) {
            JOptionPane.showMessageDialog(null, "Registro modificado");
            limpiar();
            cargarTabla();
        } else {
            JOptionPane.showMessageDialog(null, "No se modificó el registro");

        }
        con.close();
    } catch (Exception e) {
        System.out.println("ERROR AL ACUALIZAR" + e);
    }
}
```

Ilustración 55: Modificar

Buscar, mediante el número de cédula se busca en la base de datos.

Debes ser cuidadoso al recuperar en variables de Java los datos de tu consulta, usando los nombres de las columnas que has incluido en el query.

```
private void btn_buscarActionPerformed(java.awt.event.ActionEvent evt) {
    con = null;
    try {
        con = getConnection();
        String buscar = "SELECT * FROM estudiantes WHERE cedula=?";
        ps = con.prepareStatement(buscar);
        ps.setString(1, txt_cedula.getText());
        rs = ps.executeQuery();
        if (rs.next()) {
            txt_apellidos.setText(rs.getString("apellidos"));
            txt_nombres.setText(rs.getString("nombres"));
            txt_fechaN.setText(rs.getString("f_nacimiento"));
            txt_correo.setText(rs.getString("correo"));
            cbx_sexo.setSelectedItem(rs.getString("sexo"));
        } else {
            JOptionPane.showMessageDialog(null, "Registro no existe");
        }

    } catch (Exception e) {

    }
}
```

Ilustración 56: Modificar

Permite eliminar de la base de datos un registro específico.

```
private void btn_eliminarActionPerformed(java.awt.event.ActionEvent evt) {
    con = null;
    try {
        con = getConnection();
        String eliminar = "DELETE FROM estudiantes WHERE cedula=?";
        ps = con.prepareStatement(eliminar);
        ps.setString(1, txt_cedula.getText());

        int resp = ps.executeUpdate();
        if (resp > 0) {
            JOptionPane.showMessageDialog(null, "Registro eliminado");
            limpiar();
            cargarTabla();
        } else {
            JOptionPane.showMessageDialog(null, "No se eliminó");
        }
        con.close();
    } catch (Exception e) {
        System.out.println("ERROR AL ELIMINAR " + e);
    }
}
```

Ilustración 57: Eliminar

Con el método cargarTabla(), recuperamos la información de la base de datos y la mostramos en la tabla del formulario.

```
51      public void cargarTabla() {
52          DefaultTableModel modelo = new DefaultTableModel();
53          tbl_clientes.setModel(modelo);
54          con = null;
55          try {
56              con = getConnection();
57              String buscar = "SELECT cedula AS ced,apellidos AS ap,nombres AS nom,sexo AS sex FROM estudiantes";
58              ps = con.prepareStatement(buscar);
59              rs = ps.executeQuery();
60
61              ResultSetMetaData rsmd = rs.getMetaData();
62              int num_columnas = rsmd.getColumnCount();
63
64              modelo.addColumn("Cedula");
65              modelo.addColumn("Apellidos");
66              modelo.addColumn("Nombre");
67              modelo.addColumn("Sexo");
68
69              int[] ancho = {5, 100, 100, 5};
70              for (int j = 0; j < num_columnas; j++) {
71                  tbl_clientes.getColumnModel().getColumn(j).setPreferredWidth(ancho[j]);
72              }
73              while (rs.next()) {
74                  Object[] filas = new Object[num_columnas];
75                  for (int i = 0; i < num_columnas; i++) {
76                      filas[i] = rs.getObject(i + 1);
77                  }
78                  modelo.addRow(filas);
79              }
80
81              con.close();
82
83          } catch (Exception e) {
84              System.out.println("ERROR: " + e.toString());
85          }
86
87      }
```

Ilustración 58: Cargar Tabla

UNIDAD 3: EXPRESIONES LAMBDA

3.1. Definición

A partir de JDK 8, se agregó una característica a Java que mejoró profundamente el poder expresivo del lenguaje. Esta característica es la **expresión lambda**.

Las expresiones lambda son una forma de crear funciones anónimas y que se pueden utilizar en dónde el parámetro recibido sea una **interfaz funcional**

Walton (2018), define: La clave para entender la expresión lambda son dos conceptos. El primero es la **expresión lambda**, en sí misma. El segundo es la **interfaz funcional.** Comencemos con una definición simple de cada uno.

1. **Una expresión lambda es, esencialmente, un método anónimo** (es decir, sin nombre). Sin embargo, este método no se ejecuta solo. En cambio, se usa para implementar un método definido por una interfaz funcional. Por lo tanto, una expresión lambda da como resultado una forma de clase anónima. Las expresiones lambda también se conocen comúnmente como **cierres** (closures).
2. **Una interfaz funcional es una interfaz que contiene uno y solo un método abstracto**. Normalmente, este método especifica el propósito previsto de la interfaz. Por lo tanto, una interfaz funcional típicamente representa una sola acción. Por ejemplo, la interfaz estándar **Runnable** es una interfaz funcional porque define solo un método: **run()**. Entonces, *run()* define la acción de *Runnable*. Además, una interfaz funcional define el tipo de objetivo de una expresión lambda. Aquí hay un punto clave: una expresión lambda solo se puede usar

en un contexto en el que se especifica un tipo de objetivo. Otra cosa: una interfaz funcional a veces se conoce como tipo SAM, donde SAM significa **Single Abstract Method**.

Una interfaz funcional es una interfaz con uno y solo un método abstracto. La declaración es exactamente igual que las interfaces normales con dos características adicionales:

- Tiene un único método abstracto, como ya hemos dicho.
- De manera opcional puede estar anotada como @FunctionalInterface.

El motivo de que la interfaz tenga un único método abstracto es que será la expresión lambda la que proveerá de la implementación para dicho método.

La sintaxis básica se detalla a continuación:

(Parámetros) → {Cuerpo - lambda}

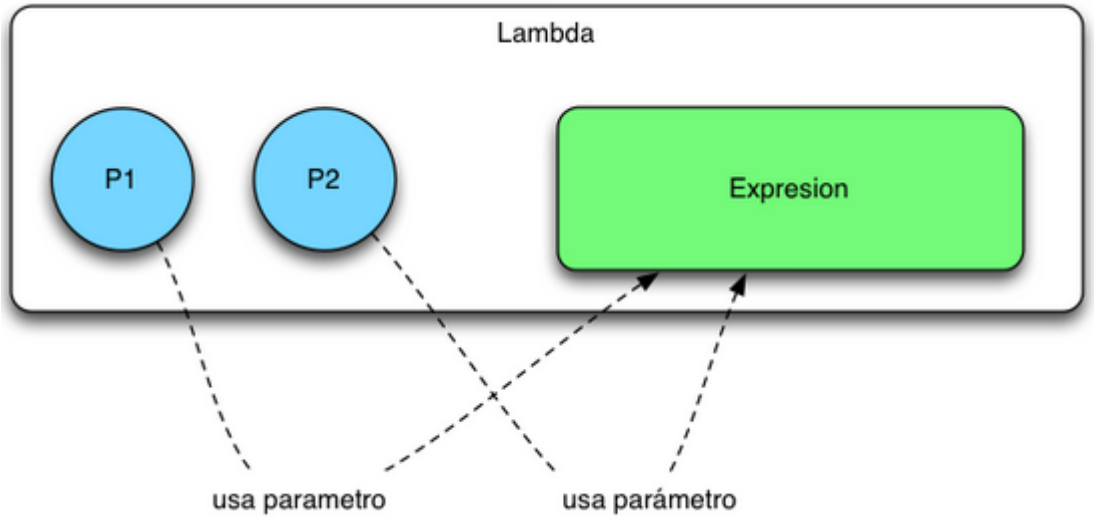

Ilustración 59: Sintaxis expresión Lambda

El operador lambda (→) separa la declaración de parámetros de la declaración del cuerpo de la función.

Parámetros:

- Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.
- Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.

Cuerpo de lambda:

- Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan especificar la cláusula **return** en el caso de que deban devolver valores.
- Cuando el cuerpo de la expresión lambda tiene más de una línea se hace necesario utilizar las llaves y es necesario incluir la cláusula **return** en el caso de que la función deba devolver un valor.

Algunos ejemplos de expresiones lambda pueden ser:

- z -> z + 2
- () -> System.out.println(» Mensaje 1 «)
- (int longitud, int altura) -> { return altura * longitud; }
- (String x) -> {

```
String retorno = x;
retorno = retorno.concat(» ***»);
return retorno;
}
```

```
() -> new ArrayList<>()
(int a, int b)-> a+b
(a) -> {
        System.out.println(a);
        return true;
       }
```

Ejemplos:

```
1        for (Integer numero : Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)) {
2            System.out.print(numero + " ");
3        }
```

Ilustración 60: Recorrer una lista de números en versiones anteriores de Java

```
1        // imprimir una lista utilizando expresiones lambda en Java 8
2         Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).forEach(
3                 n -> System.out.print(n + " "));
4
5         // otra forma utilizando expresiones Lambdas
6         Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).forEach(
7                 System.out::println);
```

Ilustración 61: Recorrer una lista de números utilizando expresiones Lambda en Java

Veamos un ejemplo sencillo, una interfaz funcional que declara un método para sumar dos números enteros y que retorna su valor, si te das cuenta el método sólo se declara.

```
1 @FunctionalInterface
2 public interface IFuncionLambda {
3     //método abstracto para sumar 2 números, que lo implementará el programador a partir de una expre
4     public void suma(int a, int b);
5 }
```

Ilustración 62: Ejemplo interfaz funcional

Ahora viene la implementación del método ***sumar***, en este caso los ***parámetros*** está declarados de forma implícita *(a, b)* de manera que el compilador empareje el tipo de dato con el que se encuentra en el método sumar.

Luego tenemos el ***cuerpo*** que es la implementación del método y que viene a ser

{ System.out.println(a + b); }.

```java
package com.ecodeup.com;

public class TestLambdas {

    public static void main(String[] args) {

        int x = 10;
        int y = 5;

        //se implementa el método de la interfaz con una expresión lambda
        IFuncionLambda iflambda = (a, b) -> {
            System.out.println(a + b);
        };
        //se utiliza el método con la implementación y se le envía los valores de x e y
        iflambda.suma(x, y);
    }
}
```

Ilustración 63: Ejemplo interfaz funcional - implementación de métodos

Ejemplo 2:

```java
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

```java
public interface MiInterfaz {
    default void saluda() {
        System.out.println("Un saludo!");
    }
    public abstract int calcula(int dato1, int dato2);
}
```

```java
@FunctionalInterface
public interface Comparator {
    // Se eluden los métodos default y estáticos
    int compare(T o1, T o2);
    // El método equals(Object obj) es implícitamente implementado por la clase objeto.
    boolean equals(Object obj);
}
```

Ilustración 64: Ejemplo interfaz funcional (2)

3.2. Clasificación de expresiones Lambda

Según Largo (2016b), Las expresiones Lambda se pueden dividir de la siguiente manera:

3.2.1. Predicados

Los predicados son expresiones que reciben un argumento y devuelven un valor lógico por ejemplo, se usa la interface **Predicate<T>:**

```
1            Predicate<String> predicate = (s) -> s.length() > 0;
2        //evalua si la cadena "predicado" es mayor a 0
3        System.out.println(predicate.test("predicado")); // true
4        //niega la valor de la evaulación
5        System.out.println(predicate.negate().test("predicado")); // false
```

Ilustración 65: Predicados (1)

En el siguiente ejemplo a partir de una lista de números enteros se imprime: los números pares, los números mayores a 5 y los impares.

```java
package com.ecodeup.lambda;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class Predicados {

    public static void main(String[] args) {
        List<Integer> listaNumeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7,8,9,10);

        System.out.println("Números pares:");
        evaluar(listaNumeros, (n)-> n%2 == 0 );

        System.out.println("Números impares:");
        evaluar(listaNumeros, (n)-> n%2 == 1 );

        System.out.println("Números mayores a 5:");
        evaluar(listaNumeros, (n)-> n > 5 );

    }
    public static void evaluar(List<Integer> listaNumeros, Predicate<Integer> predicado) {
        for(Integer n: listaNumeros)  {
            if(predicado.test(n)) {
                System.out.print(n + " ");
            }
        }
        System.out.println();
    }
}
```

Ilustración 66: Predicados (2)

3.2.2. Funciones

Las funciones reciben un argumento y devuelven un resultado, usan la interface **Function<T, R>**, revisemos un ejemplo sencillo.

```java
Function<Integer, Integer> suma = x -> x + 8;
System.out.println("La suma de 5 + 8: " + suma.apply(5));
```

Ilustración 67: Funciones (1)

Podemos también encontrar el tamaño de una cadena por ejemplo:

```java
Function<String, Integer> tamanioCadena = str -> str.length();
String cadena = "Lambdas tipo funciones";
System.out.println("Número de caracteres es : " + tamanioCadena.apply(cadena));
```

Ilustración 68: Funciones (2)

3.2.3. Proveedores

Las expresiones Lambda de este tipo no tienen ***parámetros*** de entrada, pero si devuelven un resultado, utilizan la interface ***Supplier<T>***.

Veamos un ejemplo sencillo, que básicamente obtiene la cadena enviada a la interface funcional, a través de una expresión Lambda tipo proveedor.

```
Supplier<String> cadena = () -> "Ejemplo de Proveedor";
System.out.println(cadena.get());
```

Ilustración 69: Funciones (3)

Un ejemplo un poco más detallado, primero creamos una clase Persona.

```
public class Persona {
    private String nombre;
    private String apellido;
    private String direccion;


    public Persona(String nombre, String apellido, String direccion) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.direccion = direccion;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellido() {
        return apellido;
    }
    public void setApellido(String apellido) {
        this.apellido = apellido;
    }
    public String getDireccion() {
        return direccion;
    }
    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }
}
```

Ilustración 70: Clase persona

Se implementa la expresión Lambda que es de tipo proveedor utilizando la clase Supplier.

```java
package com.ecodeup.lambda;

import java.util.function.Supplier;

public class TestLambda {

    public static void main(String[] args) {
                //se crea un proveedor de tipo Persona, el cual obtiene una persona
        Supplier<Persona> supplier = TestLambda::llenarPersona;
                //obtiene desde el proveedor la persona y la asigna a per
        Persona per = supplier.get();
                // imprime el nombre
        System.out.println(per.getNombre());
    }
    // asigna los nombres y dirección a la persona
    public static Persona llenarPersona(){
        return new Persona("Pablo", "Andrade", "Loja");
    }
}
```

Ilustración 71: Clase TestLambda

3.2.4. Consumidor

Utilizan la interfaz **Consumer<T>,** tienen un sólo argumento de entrada y no devuelven ningún valor, en este ejemplo se usa la misma clase Persona que se utilizó en el ejemplo de tipo proveedor.

```java
Consumer<Persona> persona = (p) -> System.out.println("Hola, " + p.getNombre());
persona.accept(new Persona("Jorge", "Valladares","Quito"));
```

Ilustración 72: Consumidor (1)

3.3. Referencia a métodos

Las referencias a los métodos nos permiten reutilizar un método como expresión lambda. Para hacer uso de las referencias a métodos basta con utilizar la siguiente sintáxis:

```
referenciaObjetivo::nombreDelMetodo
```

```java
File::canRead // en lugar de File f -> f.canRead();
```

Con las referencias a los métodos se ofrece una anotación más rápida para expresiones lambdas simples y existen 3 tipos diferentes:

- Métodos estáticos.
- Métodos de instancia de un tipo.

- Métodos de instancia de un objeto existente.

Nota: Para estos ejemplos se utilizó la clase Usuario que se encuentra en la parte de Resumen expresiones Lambdas que viene luego de este tema.

3.3.1. Método estático

```java
package com.ecodeup.lambdas;

import java.util.ArrayList;
import java.util.List;

public class TestReferenciaMetodos {

    public static void main(String[] args) {
        List names = new ArrayList();
          names.add("Andrea");
          names.add("Luisa");
          names.add("Diego");
          names.add("Paúl");
          names.add("Dario");
          names.forEach(System.out::println);
    }

}
```

Ilustración 73: Método estático

En este ejemplo se utiliza el método System.out::println, como referencia a métodos estáticos.

3.3.2. Métodos de instancia de un tipo

```
Referencia a métodos de instancia  Java
//recibe un objeto Usuario y devuelve la impresión de sus propiedades.
          Function<Usuario, String> ftoString= Usuario::toString;
          System.out.println(ftoString.apply(new Usuario("Santiago","Pardo",18,new Direccion("Nueva Dirección"))));
```

Ilustración 74: Método de instancia

En este ejemplo se utiliza el método toString() que fue redefinido en la clase Usuario.

3.3.3. Métodos de instancia de un objeto existente.

```java
Student student -> getMarks(student) // Expresión lambda sin referencias.
this::getMarks // Expresión lambda con referencia a método de un objeto existente.
```

Ilustración 75: Métodos de instancia de un objeto existente

Referencia a mensajes

```java
// referencia a mensajes
        LinkedList<Integer> lista = new LinkedList<Integer>(Arrays.asList(1, 2, 3));
        Supplier<Integer> funcion3 = lista::removeLast;
        System.out.println(funcion3.get()); // 3
        lista.forEach(System.out::println);
```

Ilustración 76: Referencia a mensajes

En este ejemplo se utiliza El método **removeLast** para eliminar el último elemento de la lista por último, se imprime la lista.

Referencia a constructores

```java
//referencia a constructores
        Supplier<Usuario> usu= Usuario::new;
        //Construye un objeto de tipo usuario que es devuelto por método get();
        Usuario usuario=usu.get();
```

Ilustración 77: Referencia a constructores

En este ejemplo se utiliza el operador **new** para crea una referencia a un objeto de tipo Usuario.

Resumen expresiones Lambdas

```java
package com.ecodeup.lambdas;

public class Usuario {

    private String nombre;
    private String apellido;
    private int edad;
    private Direccion dir;

    public Usuario(String nombre, String apellido, int edad, Direccion dir) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
        this.dir=dir;
    }

    public Usuario() {

    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getApellido() {
        return apellido;
    }

    public void setApellido(String apellido) {
        this.apellido = apellido;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }



    public Direccion getDir() {
        return dir;
    }
    public void setDir(Direccion dir) {
        this.dir = dir;
    }
    @Override
    public String toString() {
        return this.nombre+" "+this.apellido+" "+this.edad+" "+this.dir.getNombre();
    }
}
```

```java
package com.ecodeup.lambdas;

public class Direccion {
    private String nombre;


    public Direccion(String nombre) {
        this.nombre = nombre;
    }

    public Direccion() {

    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }



}
```

```java
package com.ecodeup.lambdas;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;

public class TestLambdas {

    public static void main(String[] args) {
        List< Usuario> names = new ArrayList<Usuario>();
          names.add(new Usuario("Elivar", "Oswaldo", 10, new Direccion("San Pedro")));
          names.add(new Usuario("Antonio", "Carrion", 15, new Direccion("bellavista")));
          names.add(new Usuario("Juan", "Andrade", 12,new Direccion("San Pedro1")));
          names.add(new Usuario("Luis", "Aguilar", 17,new Direccion("San Pedro2")));
          names.add(new Usuario("Fidel", "Narvaez", 8,new Direccion("San Pedro3")));
          names.add(new Usuario("Paul", "Guevara", 5,new Direccion("San Pedro4")));
          //Predicate<Tipo>
          //predicado: obtiene le número de usuarios con edad mayor >12 años
          System.out.println("Ejemplo de predicado:");
          System.out.println("Usuarios mayores a 12 años: "+names.stream().filter(x->x.getEdad()>12

          //Function<T,R>
          //Funcion: Obtiene la dirección que corresponde al usuario de la posición 0 de la lista
          System.out.println("\nEjemplo de función:");
          Function<Usuario,Direccion> funDireccion= v->v.getDir();
          System.out.println(funDireccion.apply(names.get(0)).getNombre());

         // Consumer<Tipo>
          //ejemplo consumidor: Actualiza el apellido del usuario de la posición 0 de la lista
          System.out.println("\nEjemplo de consumidor:");
          Consumer<Usuario> cambiaApellido = u->u.setApellido("Aguirre");
          cambiaApellido.accept(names.get(0));

          //imprime usuario actualizado
          names.forEach(System.out::println);
```

```
38
39              //Supplier<Tipo>
40              //proveedor: Crea un nuevo usuario y lo imprime con la función get
41              System.out.println("\nEjemplo de proveedor:");
42              Supplier<Usuario> u=()->new Usuario("Augusto", "Velez", 5,new Direccion("Cayambe"));
43              System.out.println(u.get());
44      }
45  }
```

UNIDAD 4: ENTERPRISE JAVABEANS

Enterprise JavaBeans (EJB) es una arquitectura que permite la creación de componentes de aplicaciones distribuidas y orientadas a transacciones. Las aplicaciones escritas utilizando EJB son escalables, transaccionales y multiusuarios.

Las características esenciales de EJB son:

- Contiene la lógica del negocio que opera con el *Enterprise Information System (EIS)*.
- Las instancias son creadas y manejadas por el container EJB.
- Puede ser configurado editando sus parámetros de entorno vía archivos XML.
- Las características de seguridad y transacciones se encuentran separadas de las clases EJB, lo que permite la operación de aplicaciones externas y middlewares.

4.1. Bean

Bean, significa vaina en inglés y es una clase destinada a almacenar una cantidad de datos para nuestro programa. Su fin es encapsular información, para reutilizar código fuente, estructurando el código fuente en unidades lo más sencillas posible.

Un JavaBean es un objeto Java al cual accedemos de forma directa desde nuestro programa

Figura 1: Acceso a un Java Bean

4.2. Java Bean

Largo, 2017 indica:

Java bean es una clase java plana, que tiene que cumplir con los siguientes requerimientos/convenciones:

- Debe tener un constructor vacío.
- Debe implementar la interfaz Serializable
- Las propiedades/atributos deben ser privados.
- Debe tener métodos getters o setters o ambos que permitan acceder a sus propiedades
- Un JavaBean es un objeto Java al cual accedemos de forma directa desde nuestro programa.

4.2.1. Propiedades de un JavaBean

Una propiedad en un JavaBean no es más que un atributo de la clase que puede ser de cualquier tipo de dato que implementa Java, también puede ser algún tipo de objeto que parte de una clase definida por el programador.

Cada propiedad deberá tener su respectivo método getter y setter, ya sea para acceder a su valor o para asignarle uno.

4.2.2. Por qué se usa la interfaz serializable?

Por lo general la información que se persiste debe viajar mediante la red a un servidor por lo que un objeto que se envía a guardar debe ser descompuesto en bytes, la interfaz serializable permite que un objeto sea descompuesto a bytes y que al otro lado pueda ser reconstruido.

4.2.3. Cuando se usa un JavaBean?

Un JavaBean se usa para crear un objeto y poderlo persistir en una base de datos, también permite, bien sea mostrar el objeto en la vista o capturar sus datos de la vista y posteriormente persistirlos en la base de datos.

Un JavaBean se asocia con una tabla en la base datos, de tal forma que las columnas de la tabla en la base de datos deben ser propiedades/atributos en un JavaBean, aunque pueden existir más propiedades, todo dependerá de las necesidades del programador.

Ejemplo:

```java
public class ClienteBean implements java.io.Serializable {
   private String nombre;
   private int edad;

   public ClienteBean() {
      // Constructor
   }

   public String getNombre() {
     return nombre;
   }

   public void setNombre(String nombre) {
     this.nombre = nombre;
   }

   public int getEdad() {
     return edad;
   }

   public void setEdad(int edad) {
     this.edad = edad;
   }
}
```

Figura 2: Java Bean

4.3. Enterprise java beans (EJB)

Nacieron con J2EE y se necesita un servidor de aplicaciones como el WebSphere o el Jboss para que puedan correr.

Los EJB no son solamente una clase Java, sino que tiene algunos artefactos asociados (xml al principio, anotations después), y además un ciclo de vida propio de ejecución dentro del **container EJB**.

"Un EJB (Enterprise Java Bean) es un componente que debe ejecutarse de un contenedor de EJBs y se diferencia bastante de un JavaBean normal, sin embargo un EJB es un componente al

cual no podemos acceder de una forma tan directa y siempre accedemos a través de algún tipo de intermediario.” (Álvarez, 2013).

Figura 3: Acceso a EJB

Este intermediario aporta una serie de servicios definidos por los standards en los cuales el EJB se puede apoyar. Los EJB de sesión disponen de dos proxies (intermediarios) a través de los cuales accedemos a ellos.

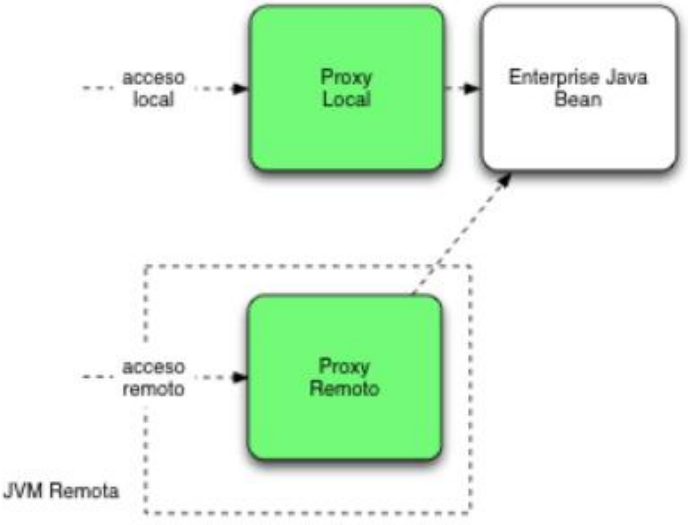

Figura 4: Proxis

- **Proxy Local:** Es el intermediario que nos permite un acceso al EJB desde la misma máquina virtual.
- **Proxy Remoto:** Es el intermediario que nos permite el acceso al EJB desde una máquina virtual remota.

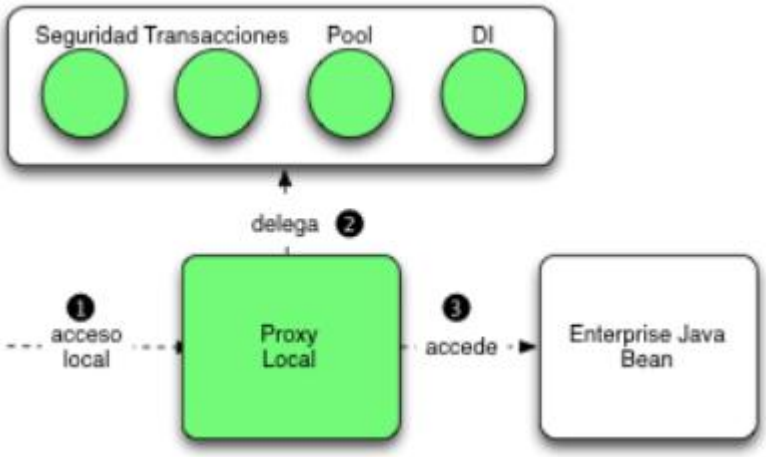

Figura 5: Acceso a EJB

Estos proxies son los encargados de dar acceso al EJB a todos los servicios adicionales que soporta el EJB Container como son Transaccionalidad, Seguridad etc. Para construir un EJB de Sesión deberemos definir los interfaces de acceso local y remoto en los cuales los proxies se apoyarán.

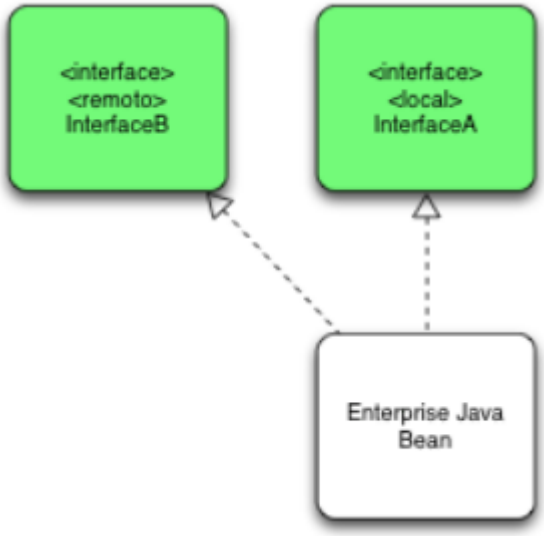

Figura 6: Tipos de interfaz de acceso

Cada uno de estos interfaces se encarga de definir los métodos que estarán a disposición de los clientes que los invocan.

4.4. Container EJB

"El contenedor EJB es un programa Java que corre en el servidor y que contiene todas las clases y objetos necesarios para el correcto funcionamiento de los enterprise beans" (Anónimo, n.d.).

El cliente nunca invoca directamente los métodos que poseen las instancias de los enterprise beans, si no que la invocación es recibida por el container EJB y luego delegada a la instancia. Al recibir las ejecuciones, el container EJB simplifica el trabajo de un desarrollador -también conocido como *Bean Provider*- al tener la posibilidad de ejecutar automáticamente tareas de middleware en forma implícita como:

- **Administración de transacciones**: Apertura y cierre de transacciones asociadas a las llamadas a los métodos del bean a través de una API de alto nivel conocida como *Java Transaction API (JTA)*
- **Seguridad:** Comprobación de permisos de acceso a los métodos del bean. Puede autenticar y autorizar cada usuario, a través de roles definidos en XML y/o utilizando JAAS.
- **Persistencia**: Automáticamente guarda cualquier objeto persistente al sistema de almacenamiento, de donde puede recuperarlos cuando sea necesario. (sincronización entre los datos del bean y tablas de una base de datos).
- **Accesibilidad remota**: El container EJB permite a los objetos acceso a servicios de red, sin tener que ser programado completamente por el desarrollador, sino que sólo definiendo interfaces de acceso remoto**. (comunicación entre el cliente y el bean en máquinas distintas).**
- **Acceso concurrente**: El container EJB automáticamente maneja invocaciones concurrentes desde los clientes, asegurando que un cliente ejecutará un bean a la vez, creando una cola

de espera para cada bean o instanciando múltiples objetos, evitando problemas de sincronización de threads.

- **Monitoreo**: El container EJB puede realizar un seguimiento de los métodos que son invocados y mostrar información de desempeño en tiempo real que apoye la administración del sistema.
- **Gestión de recursos**: gestión automática de múltiples recursos, como colas de mensajes, bases de datos o fuentes de datos en aplicaciones heredadas que no han sido traducidas a nuevos lenguajes/entornos y siguen usándose en la empresa.
- **Gestión de mensajes**: manejo de Java Message Service (JMS).
- **Escalabilidad**: posibilidad de constituir clusters de servidores de aplicaciones con múltiples hosts para poder dar respuesta a aumentos repentinos de carga de la aplicación con sólo añadir hosts adicionales.
- **Adaptación en tiempo de despliegue**: posibilidad de modificación de todas estas características en el momento del despliegue del bean.

La capa intermedia que proporciona el container EJB, es representada por el objeto EJBObject o por EJBLocalObject que debe extender toda interfaz que defina métodos a ser utilizados por el cliente. Un ejemplo de su uso se ilustra en la siguiente figura:

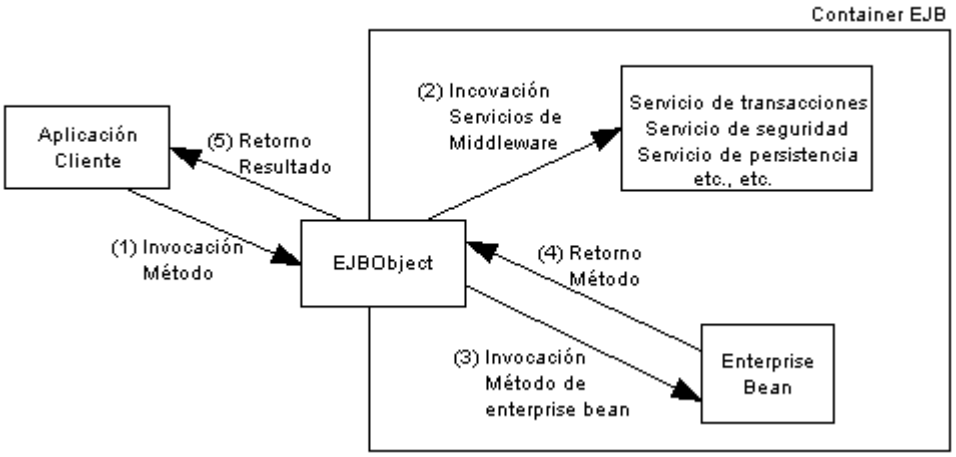

Figura 7: EJBObject

Fuente: Investigación de la Plataforma J2EE y su Aplicación Práctica

Además de estas tareas, el container EJB realiza la administración de las instancias existentes para los objetos, es decir, administra los diferentes ciclos de vida.

4.5. Funcionamiento de los componentes EJB

Según (Anónimo, n.d.), el funcionamiento de los componentes EJB se basa fundamentalmente en el trabajo del contenedor EJB. En la figura siguiente se aprecia una representación de muy alto nivel del funcionamiento básico de los enterprise beans.

En primer lugar, se muestra que el cliente que realiza peticiones al bean y el servidor que contiene el bean está ejecutándose en máquinas virtuales Java distintas, incluso pueden estar en distintos hosts.

Se resalta que el cliente *nunca* se comunica directamente con el enterprise bean, sino que el contenedor EJB proporciona un *EJBObject* que hace de interfaz. Cualquier petición del cliente (una llamada a un *método de negocio* del enterprise bean) se debe hacer a través del objeto EJB, el cual solicita al contenedor EJB una serie de servicios y se comunica con el enterprise bean.

Por último, el bean realiza las peticiones correspondientes a la base de datos.

El contenedor EJB se preocupa de cuestiones como:

- ¿Tiene el cliente permiso para llamar al método?
- Hay que abrir la transacción al comienzo de la llamada y cerrarla al terminar.
- ¿Es necesario refrescar el bean con los datos de la base de datos?

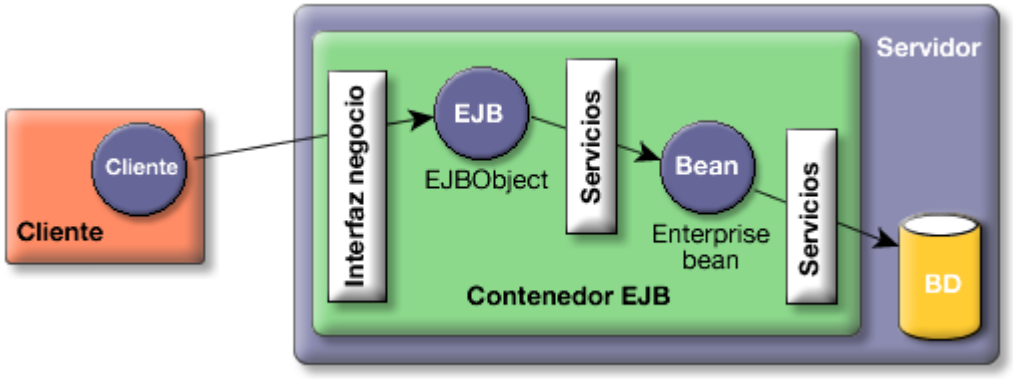

Figura 8: Representación de alto nivel del funcionamiento de los enterprise beans

Se muestra el siguiente ejemplo para una mejor comprensión del flujo de llamadas. Supongamos que tenemos una aplicación de bolsa y el bean proporciona una implementación de un Broker. La interfaz de negocio del Broker está compuesta de varios métodos, entre ellos, por ejemplo, los métodos **compra o vende**. Supongamos que desde el objeto cliente queremos llamar al método **compra**. Esto va a provocar la siguiente secuencia de llamadas:

1. ***Cliente***: "Necesito realizar una petición de compra al bean Broker."
2. ***EJBObject***: "Espera un momento, necesito comprobar tus permisos."
3. ***Contenedor EJB***: "Sí, el cliente tiene permisos suficientes para llamar al método compra."

4. ***Contenedor EJB***: "Necesito un bean Broker para realizar una operación de compra. Y no olvidéis comenzar la transacción en el momento de instanciaros."
5. ***Pool de beans***: "A ver... ¿a quién de nosotros le toca esta vez?".
6. ***Contenedor EJB***: "Ya tengo un bean Broker. Pásale la petición del cliente."

La idea de usar este tipo de diálogos para describir el funcionamiento de un proceso o una arquitectura de un sistema informático es de Kathy Sierra en sus libros *"Head First Java"* y "*Head First EJB"*.

4.6. Tipos de Enterprise Beans

Barrios (2003) en su **Investigación de la Plataforma J2EE y su Aplicación Práctica** indica que la arquitectura de EJB define tres tipos diferentes de objetos enterprise beans los cuales son:

4.6.1. Session beans.

Son los componentes que contienen la lógica de negocio que requieren los clientes de nuestra aplicación, modelan acciones como por ejemplo la lógica de calcular precios, transferir fondos entre cuentas, ejecutar una orden de compra, etc. Se ejecutan en representación de un único cliente. Son accedidos a través de un proxy (también llamado vista, término que utilizaré en adelante) tras realizar una solicitud al contenedor. Tras dicha solicitud, el cliente obtiene una vista del Session Bean, pero no el Session Bean real. Esto permite al contenedor realizar ciertas operaciones sobre el Session Bean real de forma transparente para el cliente (como gestionar su ciclo de vida, solicitar una instancia a otro contenedor trabajando en paralelo, etc).

Los componentes Session Bean pueden ser de tres tipos:

1. Stateless Session Beans

2. Stateful Session Beans
3. Singletons

Debido a su tiempo de vida reducido y a la no persistencia de sus datos, un session bean -ya sea *stateless* o *stateful*- no sobrevive a fallas en el container o en el servidor. En este caso el bean es eliminado de la memoria, siendo necesario que el cliente vuelva a iniciar una conexión para poder continuar con su uso.

4.6.1.1. Stateless EJB (Bean de sesión sin estado)

Están pensados para modelar los procesos de negocios que tienden naturalmente a una única interacción, por tanto no requieren de mantener un estado entre múltiples invocaciones. Después de la ejecución de cada método, el container puede decidir mantenerlo, destruirlo, limpiar toda la información resultante de ejecuciones previas, o reutilizarlo en otros clientes. La acción a tomar depende de la implementación del container.

Un stateless session bean solo debe contener información que no es específica a un cliente como una referencia a una fuente de recursos, que es guardada en una variable privada y que puede ser eliminada en cualquier instante por el container. Por tanto sólo puede definir un método sin parámetros para su creación, llamado *ejbCreate(),* ya que no es necesario que reciba un valor del cliente para ser inicializado.

Los métodos independientes y que están determinados sólo por los parámetros entregados por el cliente son candidatos a ser representados en este tipo de bean. Por ejemplo un método que realice un cálculo matemático con los valores entregados y retorne el resultado, o un método que verifique la validez de un número de tarjeta de crédito son posibles métodos implementables por stateless session beans.

Acedo (2012) indica que para incrementar el rendimiento, se podrían utilizar **stateless** session bean en alguna de las siguientes situaciones:

- El estado del bean no tiene datos para un cliente en específico.
- En un único método de invocación, el bean realiza una tarea genérica para todos los clientes. Por ejemplo, se podría utilizar un stateless session bean para mandar un email que confirme una compra online.
- El bean implementa un servicio web.

4.6.1.2. Ciclo de vida de stateless session beans

1. El container decide crear una nueva instancia, que depende de la implementación del container y puede variar desde instanciar al recibir la primera solicitud al bean o instanciar *N* beans al iniciar la aplicación.
2. El container ejecuta el método *setSessionContext* con el objetivo de entregar al bean una referencia al container y sus configuraciones
3. Ejecuta el método *ejbCreate* para ser inicializado, por ejemplo obtener una referencia a una base de datos. En este instante el bean se encuentra en estado ready en el cual recibe y ejecuta las invocaciones a sus métodos, donde en cada ejecución puede ser asignado a clientes diferentes según la implementación del container.
4. Una vez que el container decide que la instancia debe ser eliminada, se ejecuta el método ejbRemove para permitir que libere los recursos externos que utilice y termine en forma correcta su ejecución.

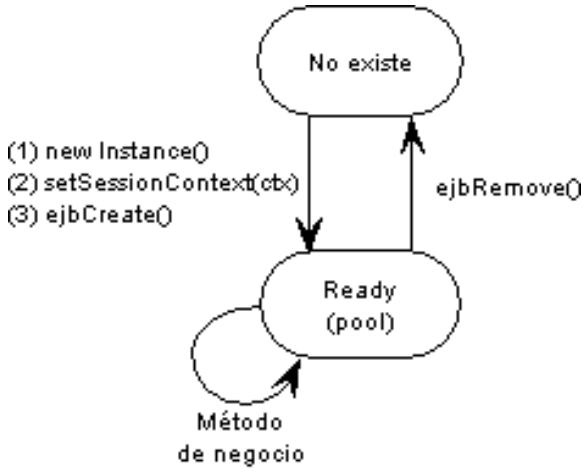

Figura 9: Ciclo de vida de stateless session beans

Fuente: Investigación de la Plataforma J2EE y su Aplicación Práctica

Debido a que no mantienen estado de interacciones, todas las instancias de la misma clase son equivalentes e indistinguibles para un cliente, sin importar quien utilizó un bean en el pasado. Por tanto pueden ser intercambiados entre un cliente y otro en cada invocación de un método, creando un *pool* de stateless session beans.

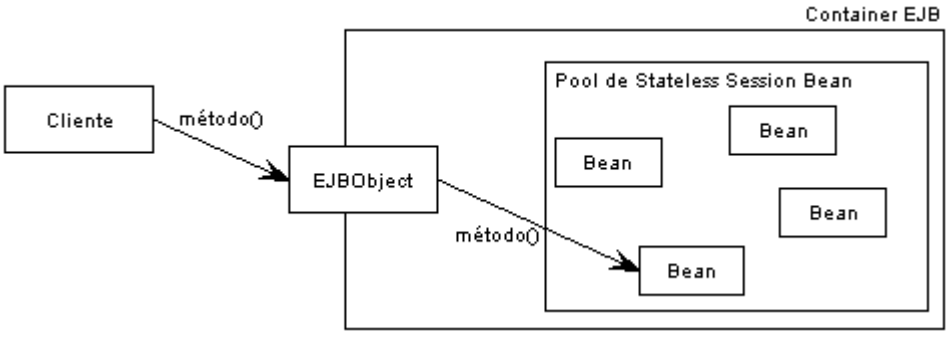

Figura 10: Pool de stateless session beans

Fuente: Investigación de la Plataforma J2EE y su Aplicación Práctica

Es necesario señalar que sólo un cliente puede invocar una instancia a la vez, esto es, el container no debe permitir la ejecución de múltiples *threads* sobre una misma instancia con el objeto de evitar posibles problemas derivados del procesamiento paralelo.

4.6.1.3. Stateful EJB (Bean de sesión con estado).

Al contrario del stateless session beans, este mantiene el estado entre distintas invocaciones realizadas por el mismo cliente (estado conversacional). Para lograr esto es necesario guardar el estado en que se encuentra el bean luego de cada ejecución del cliente, el cual se mantiene y actualiza para cada nueva invocación del mismo cliente.

Un ejemplo de stateful session bean es implementar un carro de compras donde el cliente selecciona variados ítemes que deben ser agregados a él (mantiene los objetos que hemos añadido mientras navegamos por las diferentes páginas).

La interacción del cliente con el bean se divide en un conjunto de pasos. En cada paso se añade nueva información al estado del bean. Cada paso de interacción suele denominarse con nombres como setNombre o setDireccion, siendo nombre y direccion dos variables de instancia del bean.

Debido a que el bean guarda el estado conversacional con un cliente determinado, no le es posible al contenedor crear un almacén de beans y compartirlos entre muchos clientes. Por ello, el manejo de beans de sesión con estado es más pesado que el de beans de sesión sin estado.

Acedo (2012), recomienda el uso de **stateful** cuando se cumpla una de las siguientes situaciones:

- El estado del bean representa la interacción entre el bean y un cliente específico.

- El bean necesita mantener información acerca del cliente a lo largo de varias invocaciones de métodos.
- El bean es mediador entre el cliente y los otros componentes de la aplicación, presentando una vista simplificada para el cliente.

4.6.1.4. Ciclo de vida de Stateful Session Beans

Barrios (2003), describe el ciclo de vida:

Los stateful session beans, al no poder ser intercambiados entre clientes deben mantenerse en espera del *request* del cliente al cual corresponden, sin poder ser reutilizados en este período. Para disminuir este derroche de recursos sin utilización se crean los conceptos de estado *activo* y estado *pasivo*.

Para limitar el número de instancias de stateful session beans en la memoria, el container tiene la capacidad de utilizar el protocolo de serialización de Java para convertirlo en un conjunto de datos almacenables en una unidad externa. Luego, para que el bean pueda eliminar recursos que luego puede volver a adquirir como referencias a bases de datos o sockets, el container invoca el método *ejbPassivate*. Una vez terminado este método, el bean entra al estado *pasivo* y la instancia puede ser reinicializada o eliminada.

Cuando un bean se encuentra *pasivo* y el container desea transformarlo en *activo*, debe leer la información guardada para restaurar su estado original y luego invocar el método *ejbActivate* para recuperar los recursos eliminados previamente.

El algoritmo para decidir cuándo un bean debe cambiar de estado es dependiente del container. Normalmente se utiliza una estrategia *Least Recently Used (LRU)* que simplemente elige el bean con más tiempo inactivo para cambiar su estado a *pasivo*. La única excepción a esta regla es

cuando el bean se encuentra realizando una transacción ya que no puede cambiar de estado mientras ésta no finalice. Después, el algoritmo utilizado normalmente para activar un bean pasivo es esperar que el cliente correspondiente invoque uno de sus métodos. El ciclo de vida de un stateful session bean se puede ver en la siguiente figura:

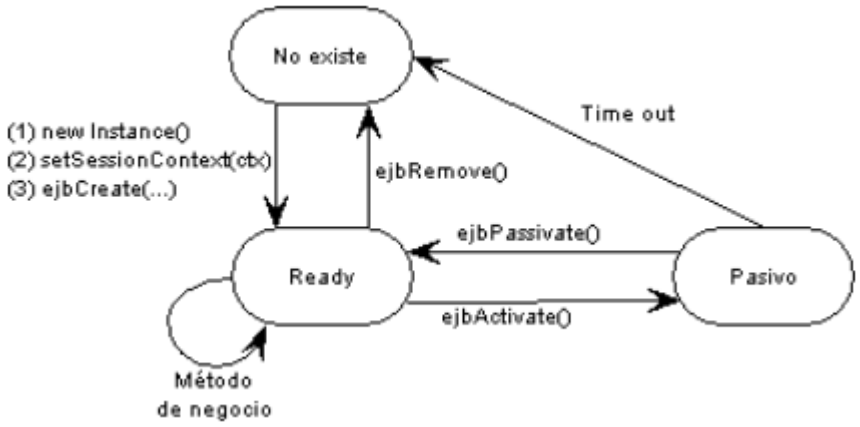

Figura 11: Ciclo de vida de stateful session beans

Fuente: Investigación de la Plataforma J2EE y su Aplicación Práctica

Al mantener un estado interno que representa a un cliente, un stateful session bean no puede ser intercambiado entre clientes, lo que imposibilita un *pool* de stateful session beans.

Los componentes ***Singleton*** son un nuevo tipo de Session Bean introducido en EJB 3.1. Un Singleton es un componente que puede ser compartido por muchos clientes, de manera que una y solo una instancia es creada. A nivel de eficiencia en uso de memoria y recursos son indiscutíblemente los mejores, aunque su uso está restringido a resolver ciertos problemas muy específicos.

Los singleton session bean son apropiados en las siguientes circunstancias:

- El estado necesita ser compartido a través de la aplicación.
- Un único enterprise bean necesita ser accedido por múltiples hebras concurrentemente.

4.6.2. Entity beans.

Contienen el modelo de datos del negocio y la lógica interna de los datos como por ejemplo un producto, una orden, un empleado, la lógica del cambio de nombre de un cliente, reducir la cantidad de dinero de una cuenta, etc. Su tiempo de vida es tan largo como los datos en el sistema de almacenamiento que representan.

Los componentes *Entity Beans* (Beans de Entidad, a partir de ahora EB) son representaciones de información (en forma de POJO's) que es almacenada en una base de datos. El encargado de gestionar los EB es `EntityManager`, un servicio que es suministrado por el contenedor y que está incluido en la especificación Java Persistence API (JPA - API de Persistencia en Java). JPA es parte de EJB desde la versión 3.0. (Álvarez, 2013)

Al contrario que los Session Beans y los Message-Driven Beans, los Entity Beans no son componentes del lado del servidor. En otras palabras, no trabajamos con una vista del componente, si no con el componente real.

4.6.2.1. Ciclo de vida de Entity Beans

Cuando el cliente desea crear un nuevo entity bean significa que desea crear un nuevo registro en la base de datos. Para esto invoca el método *create* que lo toma el container y lo delega al método *ejbCreate*. El método *ejbCreate* toma los parámetros recibidos, los utiliza para asignarlos a sus atributos y debe retornar al container el objeto *PK* que lo identifica en forma única. Con la llave primaria el container puede crear el entity bean que es el que retorna al cliente.

Cuando el cliente desea eliminar un registro debe invocar el método *remove* que es recibido por el container, el que delega la ejecución sobre *ejbRemove* que debe eliminar el registro de la base de datos y preparase para ser borrado por el recolector de basura. Sin embargo el container no debe necesariamente permitir eliminar la instancia existente, si no que puede agregarla al *pool* de instancias para ser reutilizada.

Para llevar a cabo esta reutilización es necesario crear o eliminar cualquier tipo de recursos externos que utilice la implementación del entity bean que no esté ligada a la base de datos como archivos o sockets. Por esto cuando el container decide utilizar una instancia del *pool* debe ejecutar del método *ejbActivate* para que el bean obtenga los recursos que necesite y luego el método *ejbLoad* para obtener los datos. Cuando el container deposita una instancia en el *pool* debe invocar el método *ejbStore* para guardar su estado y luego *ejbPassivate* para que liberar sus recursos.

Como los entity beans están identificados únicamente por su llave primaria, se pueden implementar búsquedas de datos llamadas finders que son análogas a ejecutar sentencias select en SQL. El ciclo de vida que es administrado por el container para entity beans se muestra en la siguiente figura:

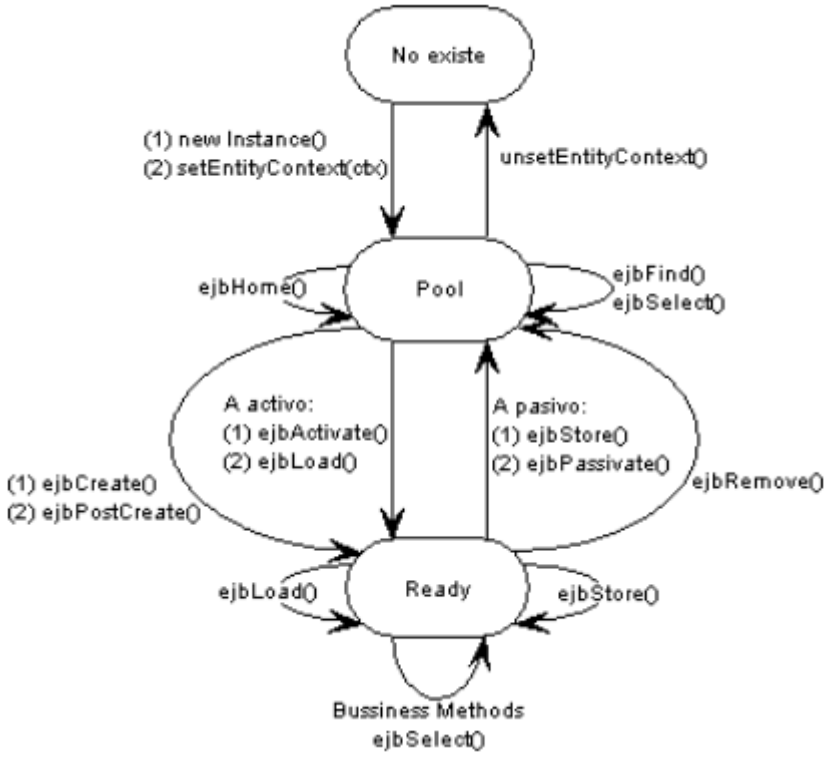

Figura 12: Ciclo de vida de un entity bean

Fuente: Investigación de la Plataforma J2EE y su Aplicación Práctica

4.6.3. Message-driven beans.

Modelan acciones, pero sólo se ejecutan luego de recibir un mensaje (Un message-driven bean es invocado por el container como resultado de la recepción de un mensaje enviado por un cliente utilizando Java Message Service (JMS)). Contienen la lógica de procesar un mensaje en forma asíncrona como puede ser recibir un mensaje con la necesidad de actualizar el stock de cierto producto e invocar el session bean que se encargan de solucionarlo.

Un cliente no ejecuta directamente un message-driven bean, sino que sólo debe utilizar la API de JMS para enviar mensajes. Por esto, un message-driven bean no tiene una clase *home* ni

interfaz local o remota ni retorna valores o excepciones al cliente. El cliente no espera que su mensaje sea respondido si no que continúan su ejecución una vez enviado.

Los message-driven beans sólo reciben mensajes JMS, sin conocer de antemano la información sobre contenido del mensaje recibido. Por esta razón sólo tienen un método con lógica de negocio llamado *onMessage()*, que recibe un *Message* JMS que puede representar todos los tipos de mensajes existentes en JMS como mensajes de bytes, de texto y de objetos serializables. Luego hay que discriminar el tipo de mensaje recibido utilizando el operador *instanceOf*.

Los message-driven beans son *stateless* ya que no mantienen estados de conversación entre cada procesamiento de mensajes recibidos, por lo cual las instancias de la misma clase son equivalentes entre sí y deben implementar solo un método *ejbCreate()* sin parámetros

Al igual que los Stateless Session Beans, los Message-Driven Beans no mantienen estado entre invocaciones.

4.6.3.1. Ciclo de vida de Message-driven beans

Para evitar errores de programación, un message-driven bean no puede ser ejecutado por más de un thread a la vez. Por esta razón y además por no guardar estados, el container puede crear un pool de este tipo de beans similar al existente para stateless session beans.

Debido a su naturaleza *stateless* el ciclo de vida de un message-driven es simple. Cuando el container desea agregar un nuevo message-driven bean al *pool* debe crear una nueva instancia, asignar el contexto para que obtenga parámetros de sistema y finalmente ejecutar el método *ejbCreate()*. Después el bean se encuentra en capacidad para recibir y procesar mensajes. Cuando el container decide reducir el número de instancias existentes invoca el método *ejbRemove()* terminando su ciclo de vida.

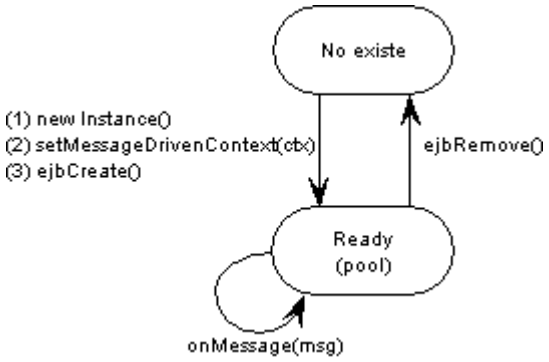

Figura 13: Ciclo de vida de un message-driven bean

Fuente: Investigación de la Plataforma J2EE y su Aplicación Práctica

Debido a su reducido tiempo de vida y a la no persistencia de sus datos, un message-driven bean no sobrevive a fallas en el container o en el servidor, debiendo ser re instanciado para continuar recibiendo mensajes luego de ésta.

Ejemplo:

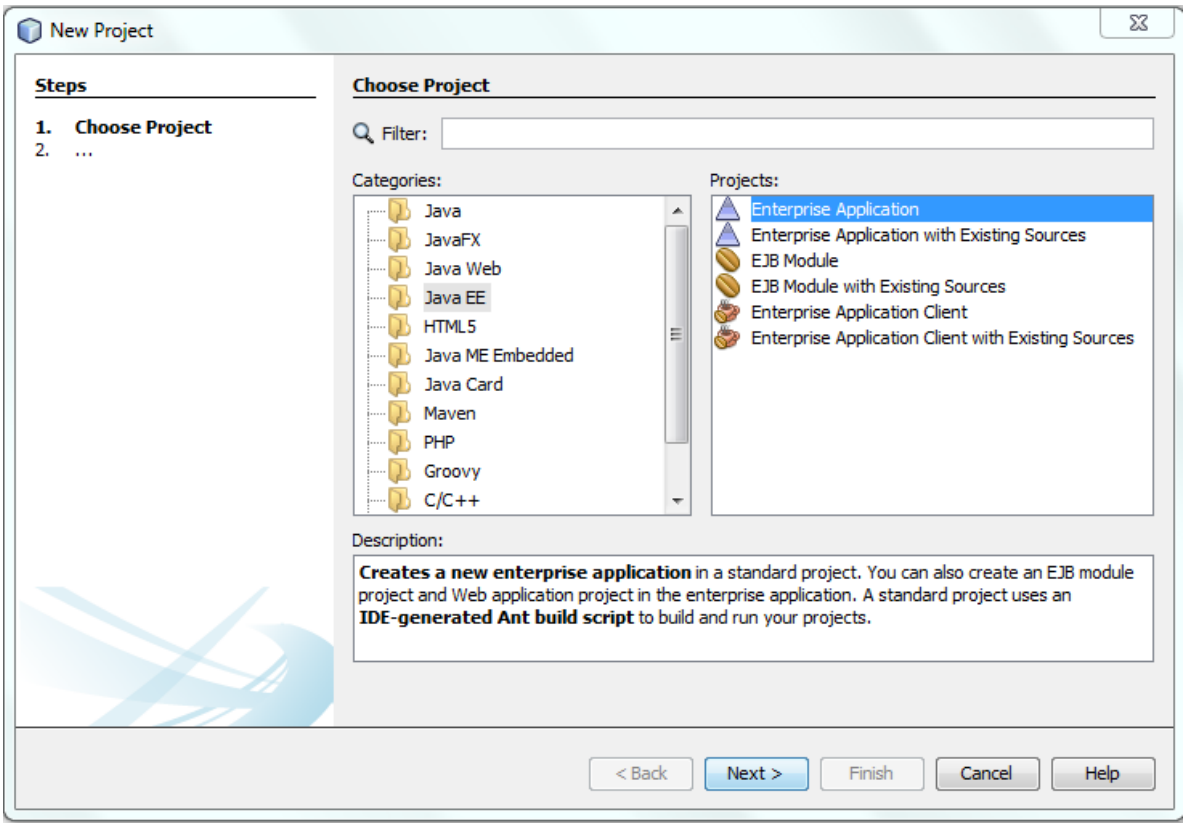

Ilustración 78: Crear el proyecto

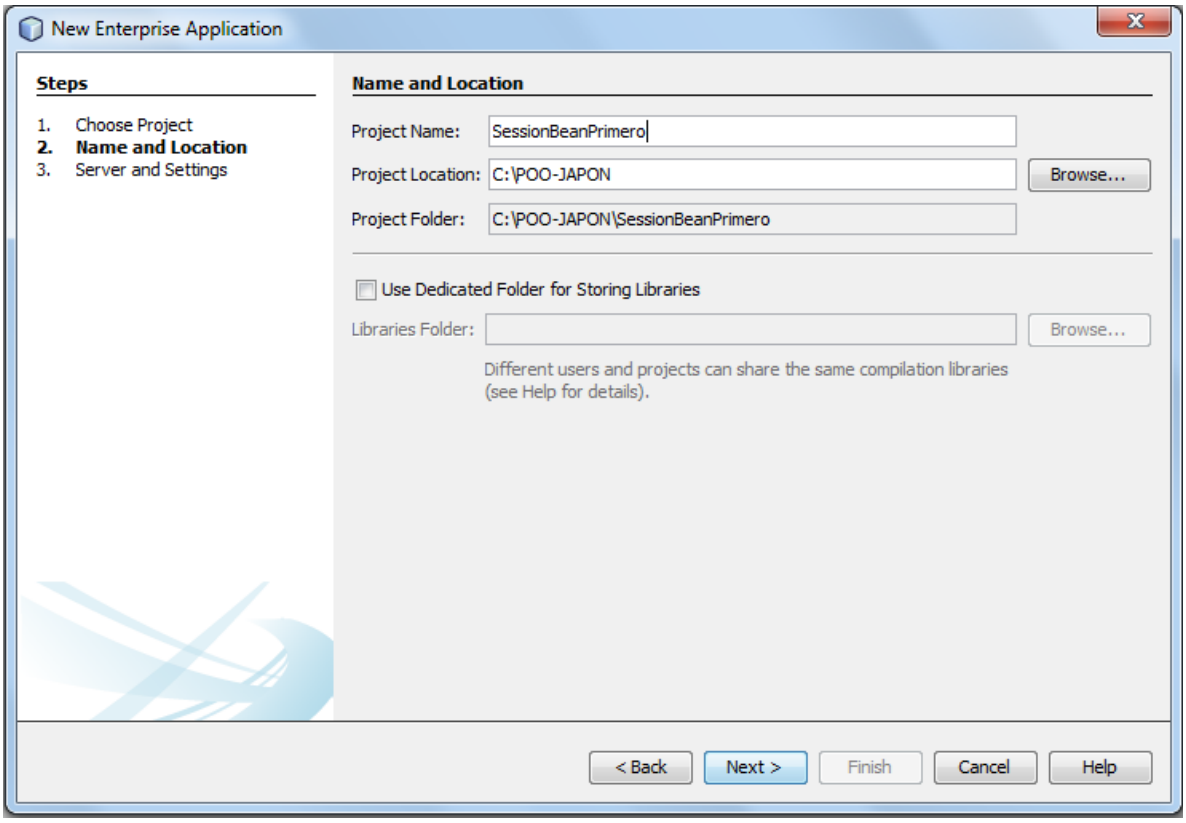

Ilustración 79: Asignar nombre al proyecto

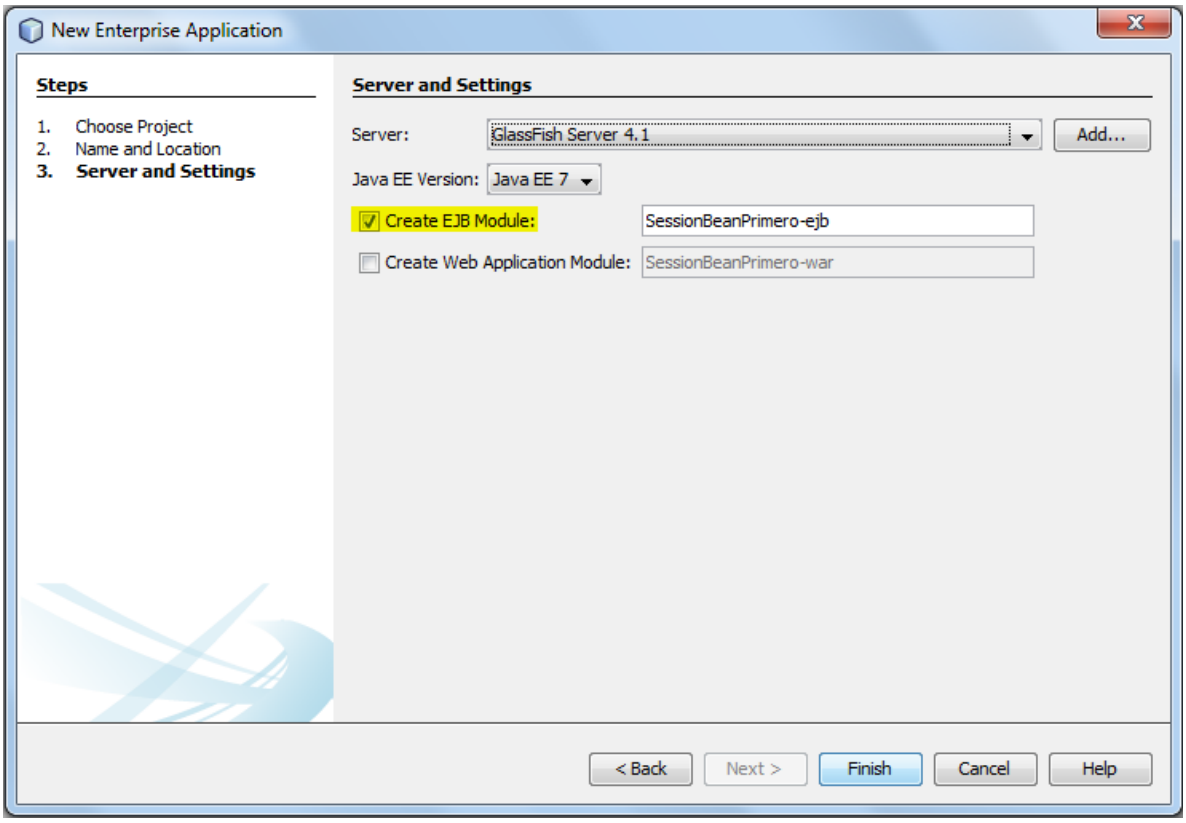

Ilustración 80: Seleccionar módulos

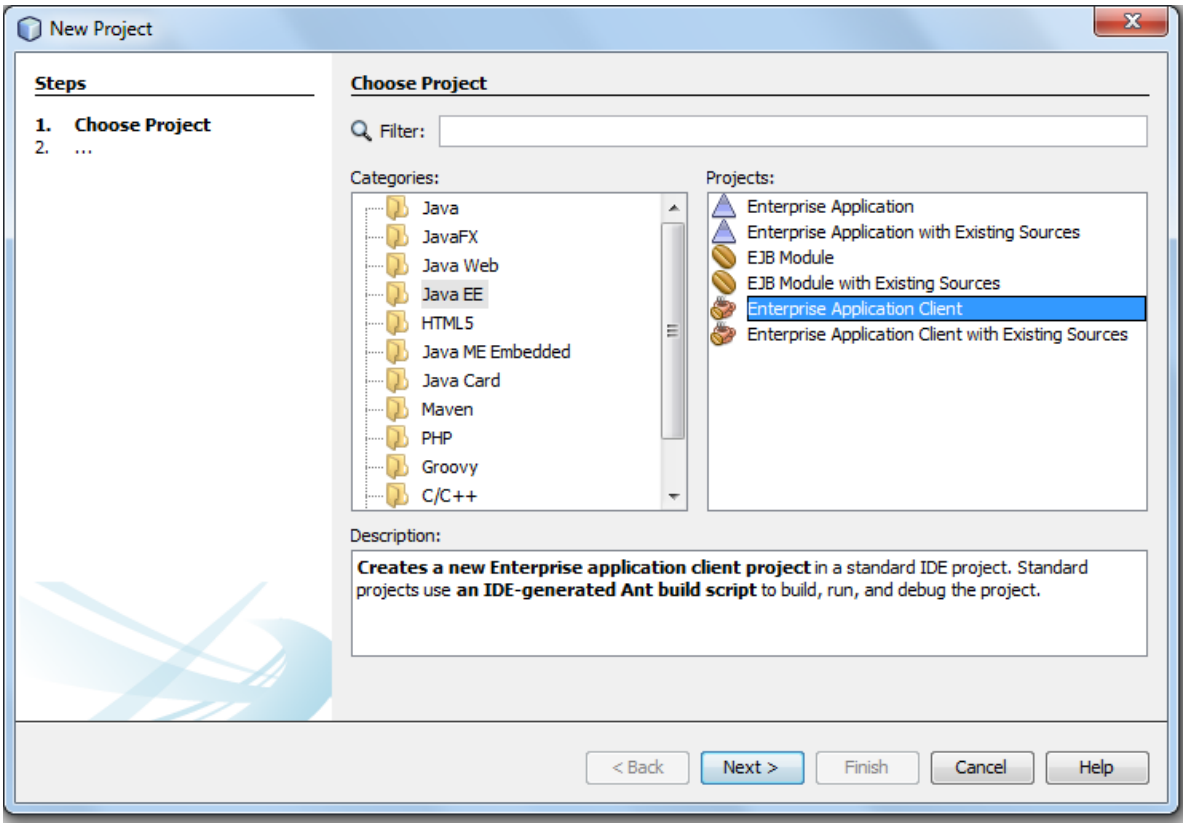

Ilustración 81: Crear aplicacion Cliente

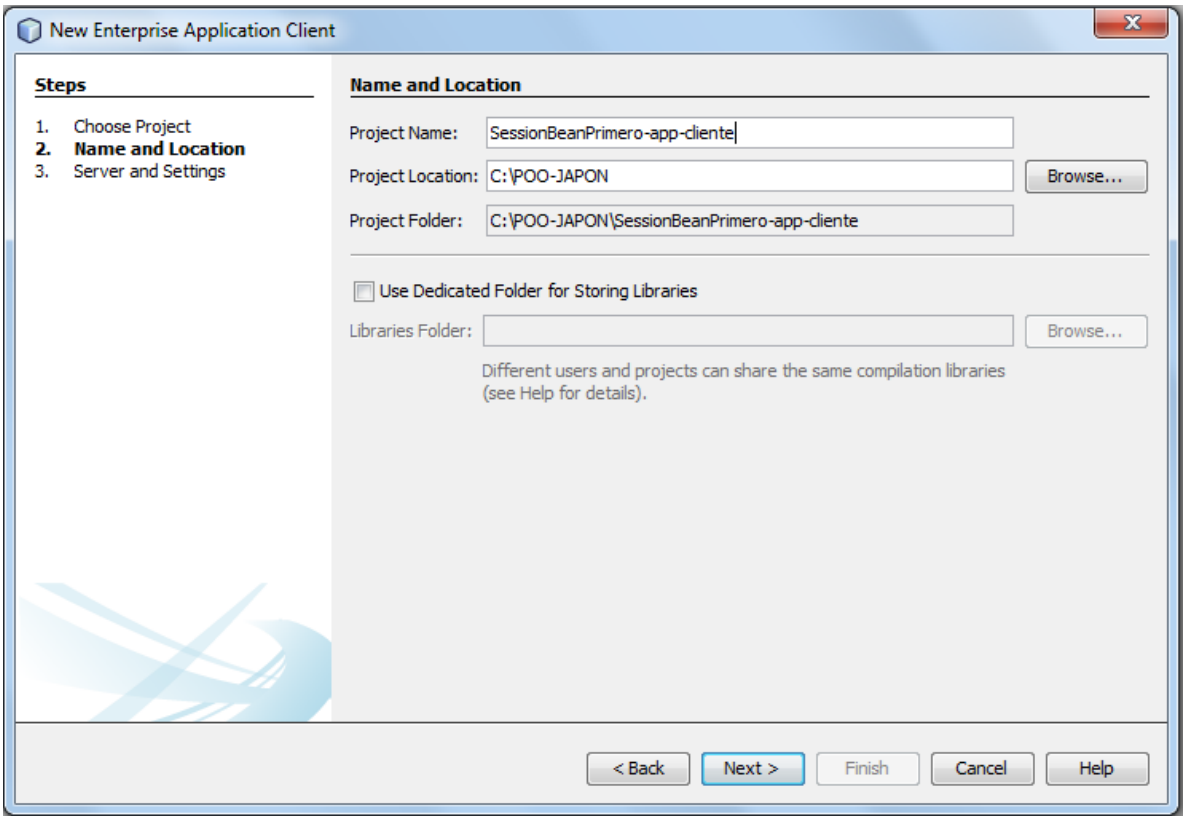

Ilustración 82: Nombre aplicación

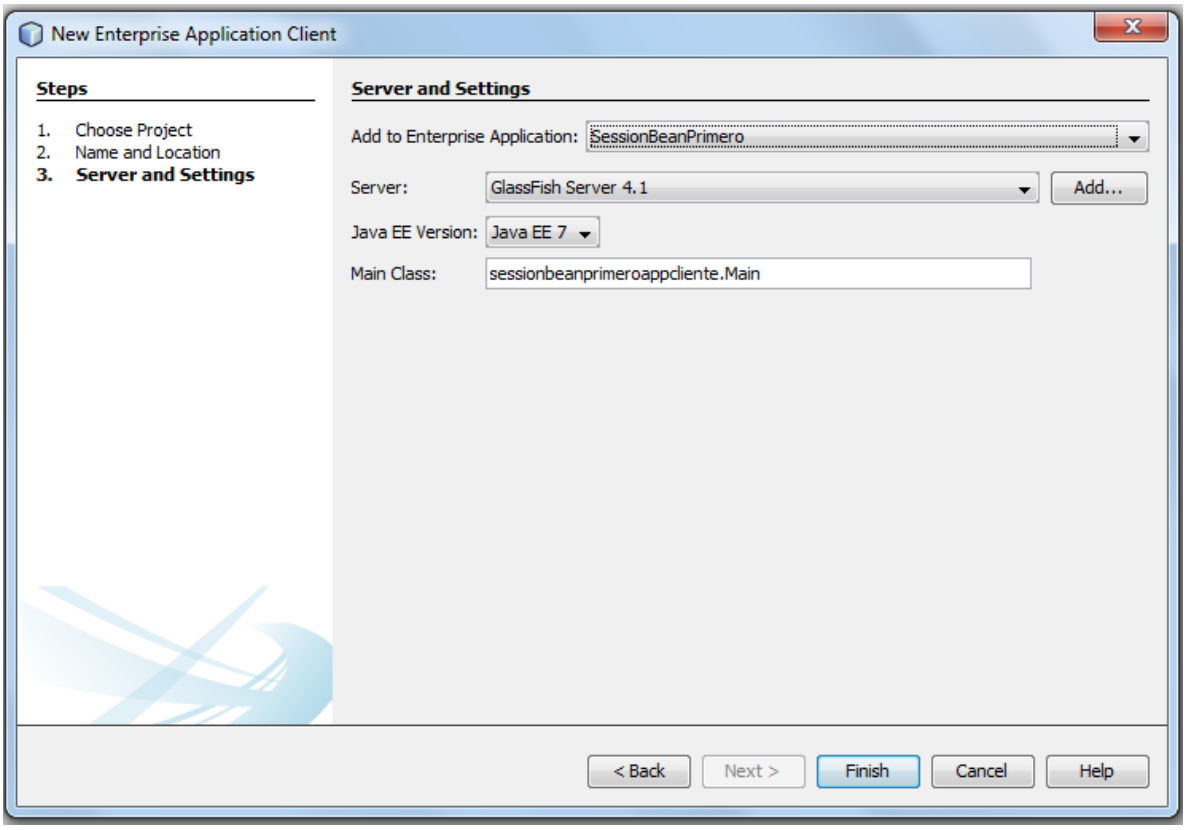

Ilustración 83: Añadir a una aplicación Enterpise

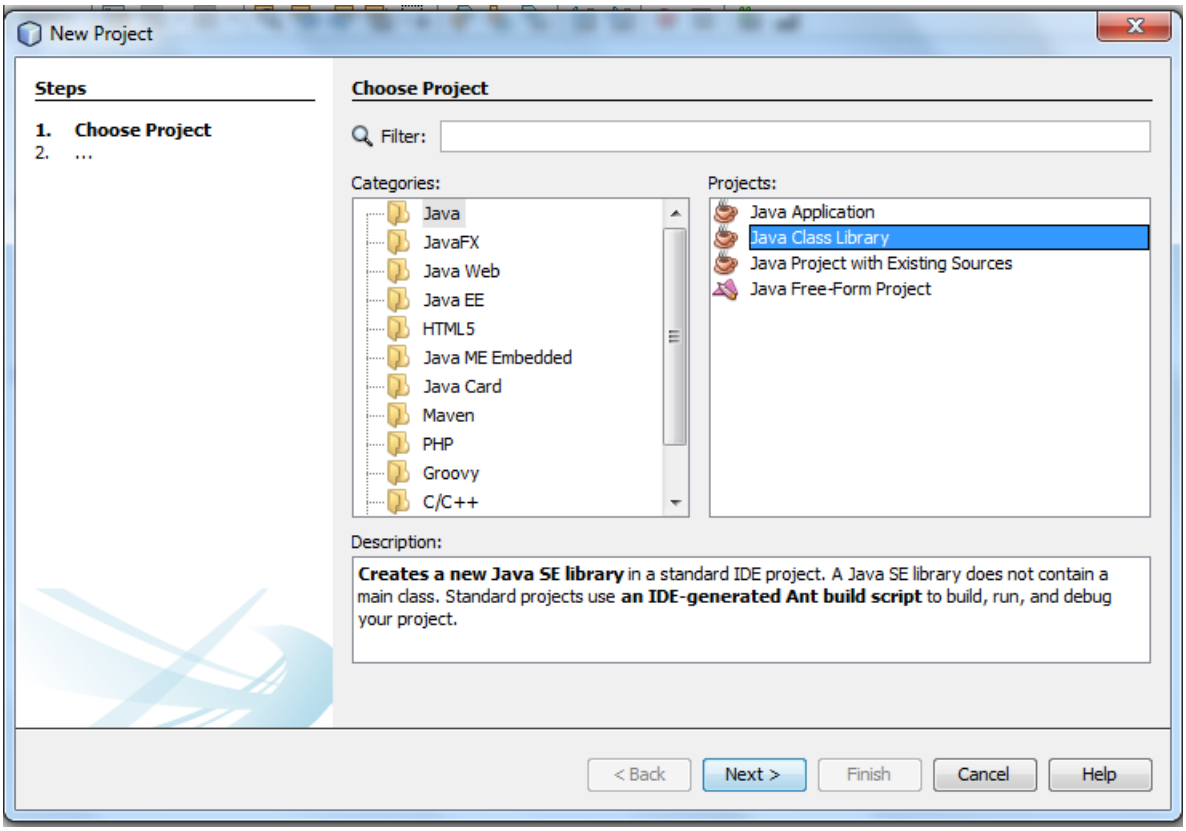

Ilustración 84: Crear Proyecto Java Class Library

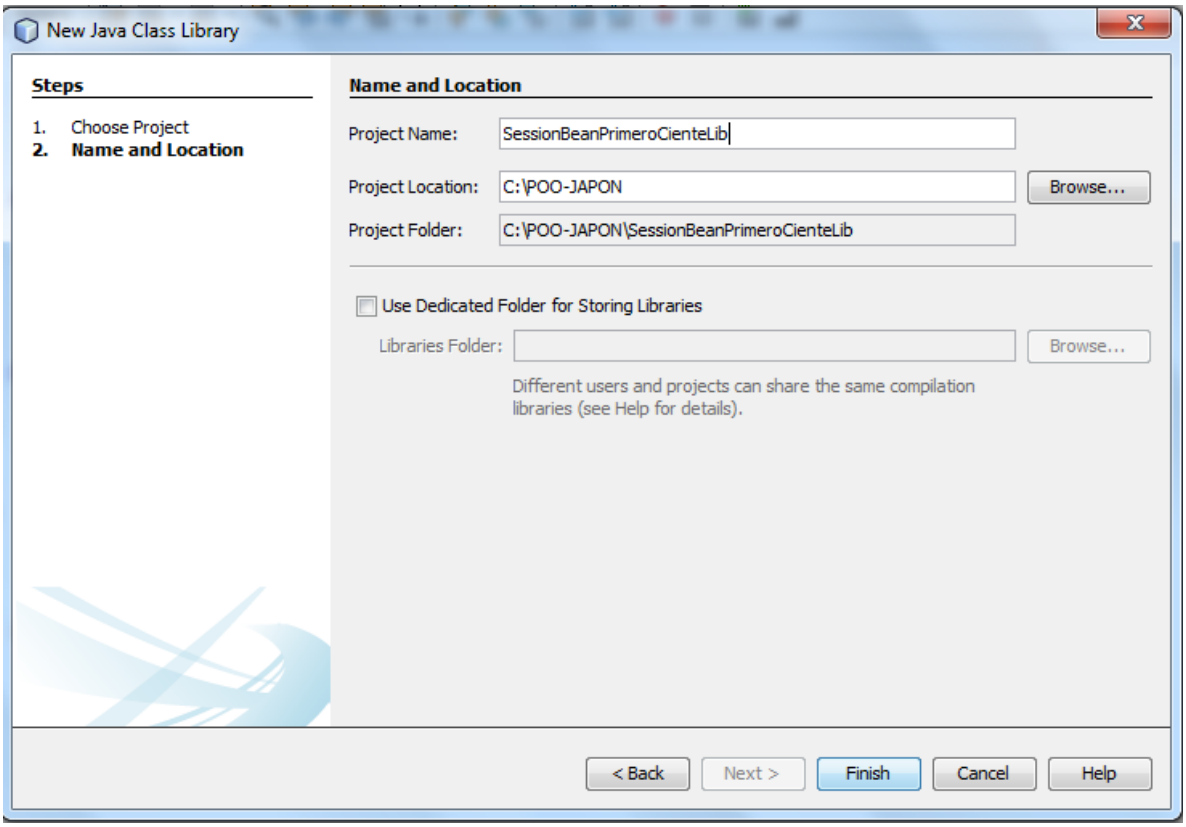

Ilustración 85: Nombre al proyecto

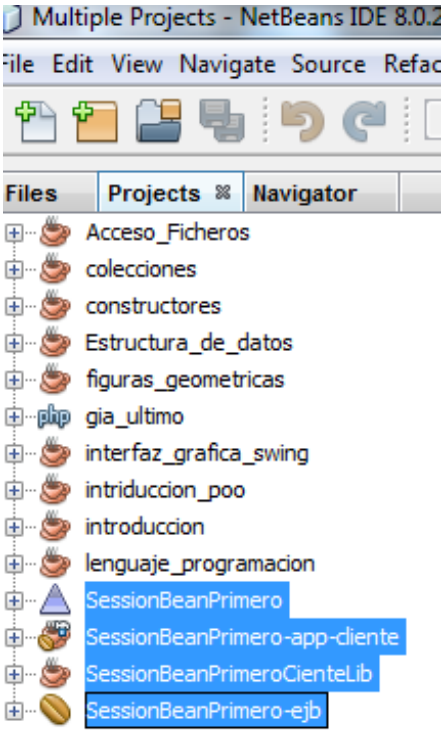

Ilustración 86: Proyecto creados

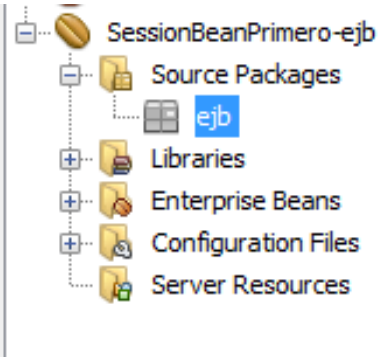

Ilustración 87: Crear Paquete

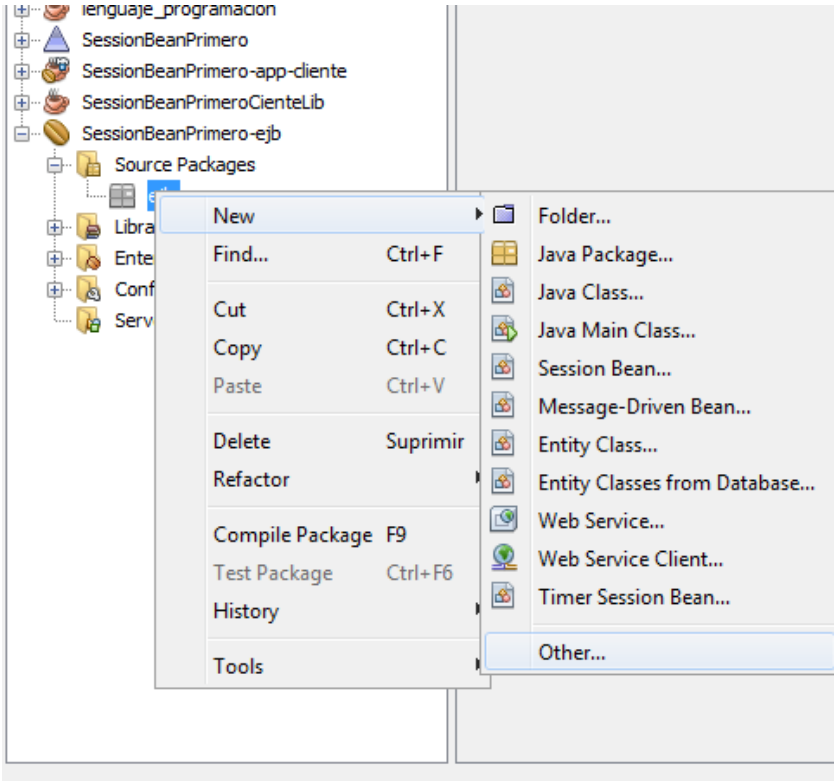

Ilustración 88: Crear una SessionBean

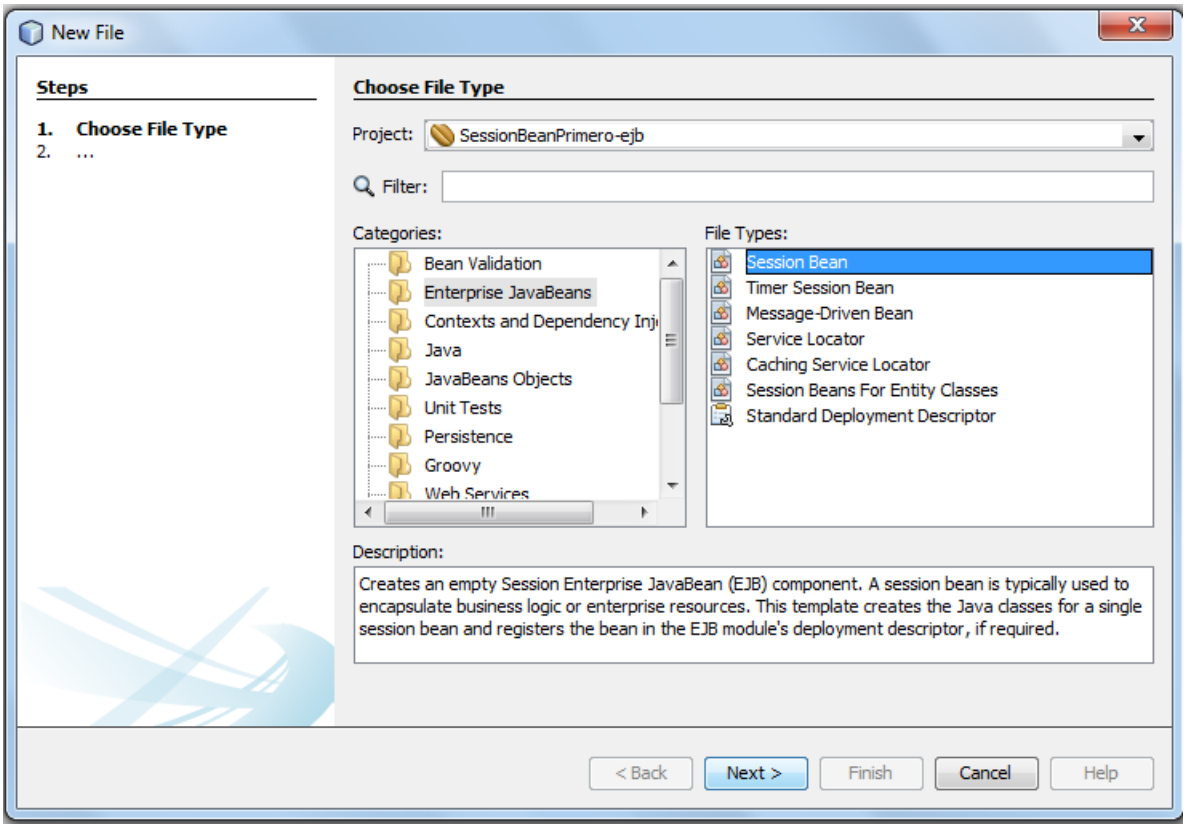

Ilustración 89: Seleccionar SessionBean

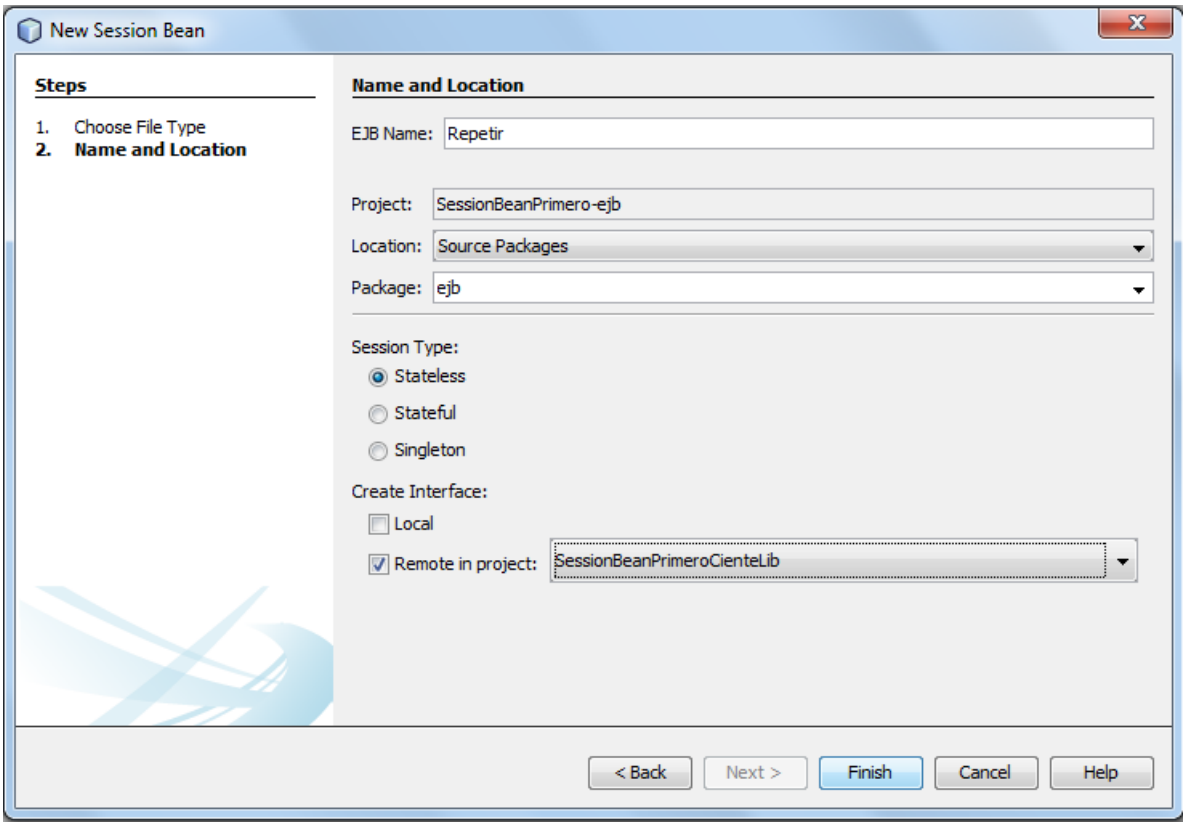

Ilustración 90: Nombre SessionBean

Repetir.java

Source | History

```
1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package ejb;
7
8  import javax.ejb.Stateless;
9
10 /**
11  *
12  * @author YEC
13  */
14 @Stateless
15 public class Repetir implements RepetirRemote {
16
17     // Add business logic below. (Right-click in editor and choose
18     // "Insert Code > Add Business Method")
19 }
20
```

Ilustración 91: Clase repetir

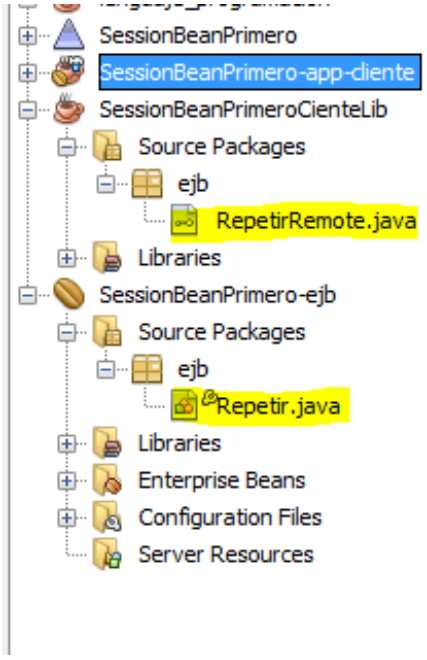

Ilustración 92: Clases que se crean

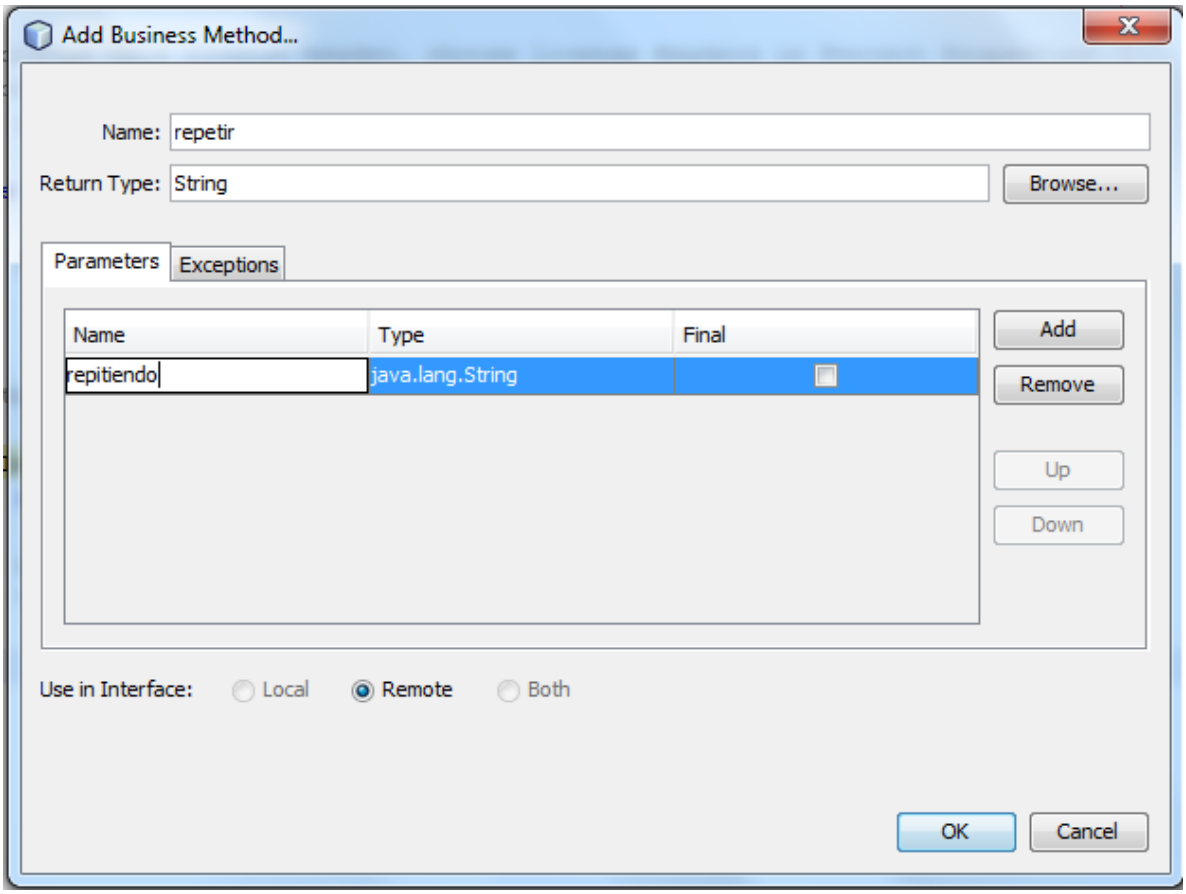

Ilustración 93: Crear un método

```
 * @author YEC
 */
@Stateless
public class Repetir implements RepetirRemote {

    @Override
    public String repetir(String repitiendo) {
        return null;
    }

    // Add business logic below. (Right-click in edit
    // "Insert Code > Add Business Method")
```

Ilustración 94: Método agregado

```
3   */
4   @Stateless
5   public class Repetir implements RepetirRemote {
6
7       @Override
8       public String repetir(String repitiendo) {
9           return "Repitiendo:"+repitiendo;
0       }
1
```

Ilustración 95: Cambiar retorno

```
*/
@Remote
public interface RepetirRemote {

    String repetir(String repitiendo);

}
```

Ilustración 96: Declaración del método en la interfaz

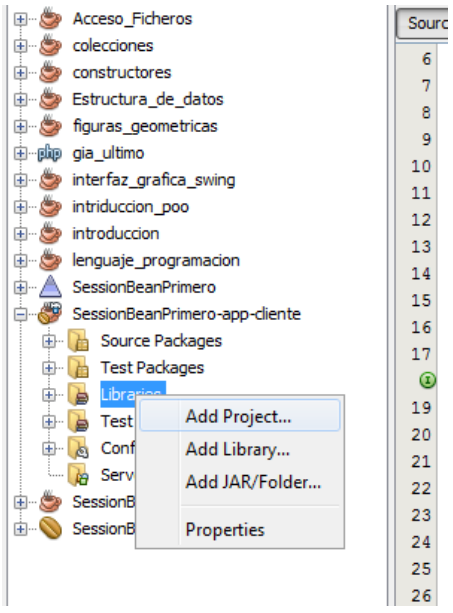

Ilustración 97: SessionBeanPrimero-app-cliente debe usar la librería SessionBeanPrimeroCienteLib

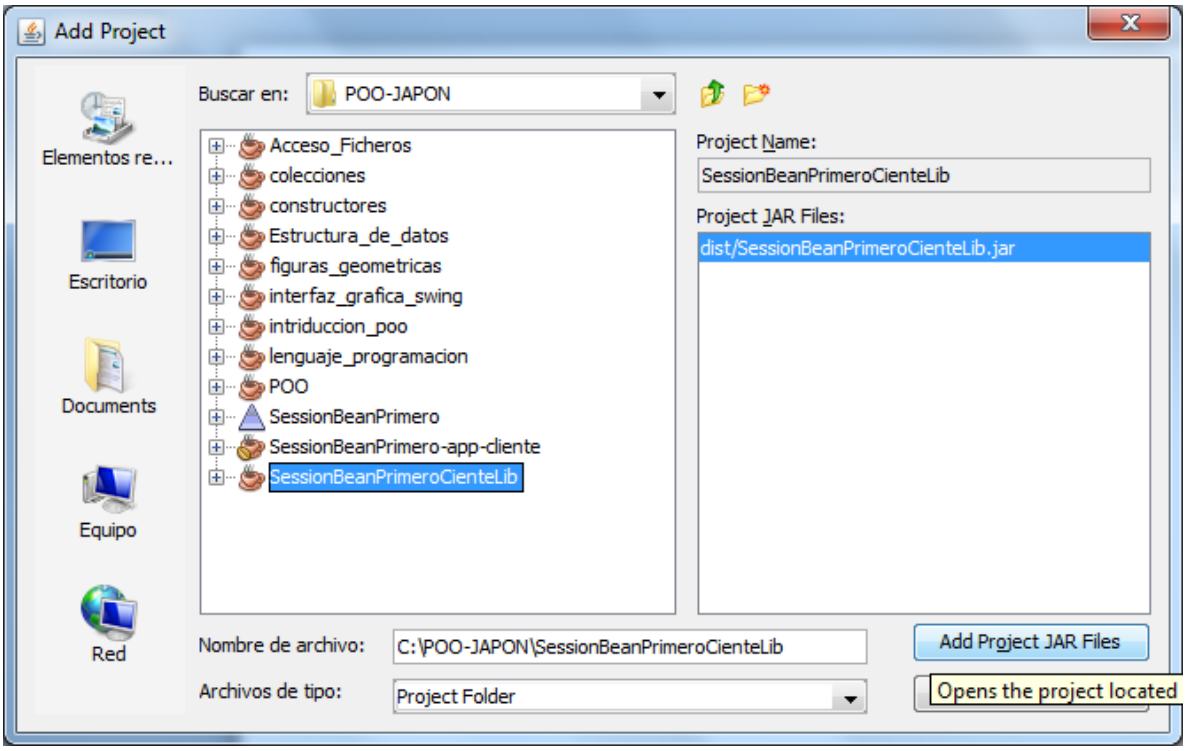

Ilustración 98: Buscar proyecto

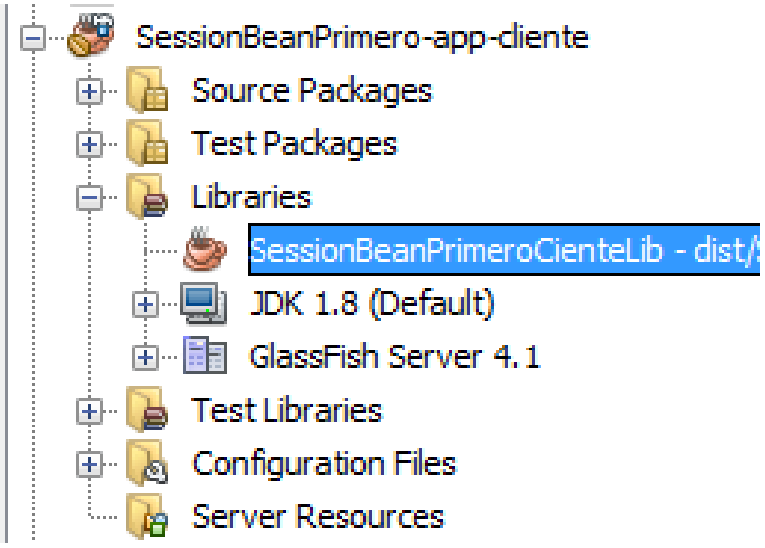

Ilustración 99: Librería agregada

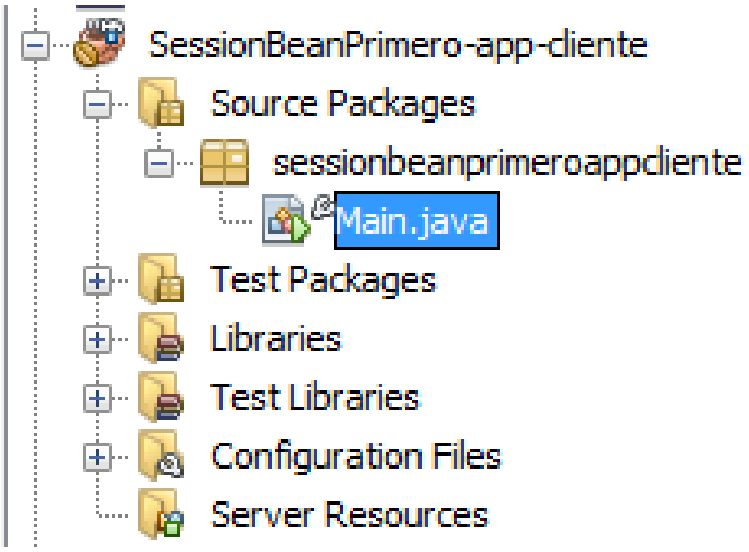

Ilustración 100: Editar el Main

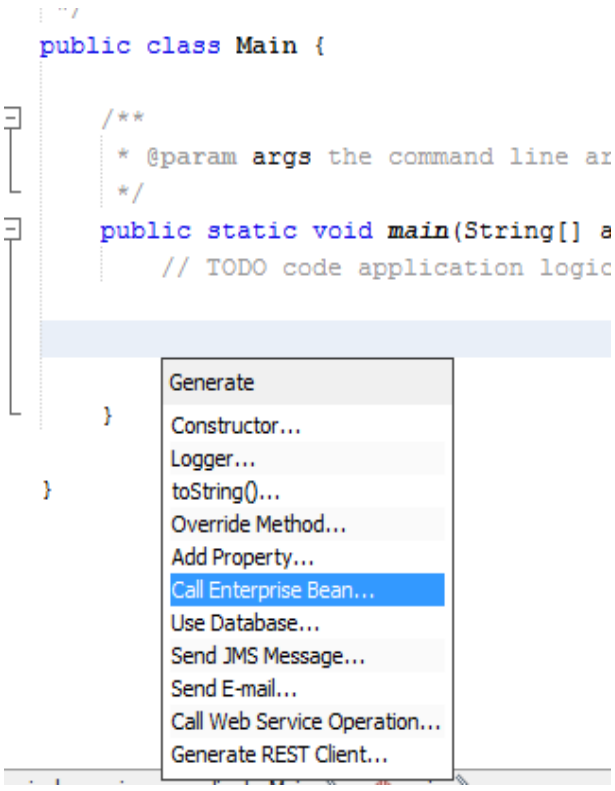

Ilustración 101: Insertar

```
 */
public class Main {
    @EJB
    private static RepetirRemote repetir;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here



    }

}
```

Ilustración 102: Anotación @EJB

```
import ejb.RepetirRemote;
import javax.ejb.EJB;
import javax.swing.JOptionPane;

/**
 *
 * @author YEC
 */
public class Main {
    @EJB
    private static RepetirRemote repetir;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        JOptionPane.showMessageDialog(null, repetir.repetir("Correcto!"));



    }

}
```

Ilustración 103:

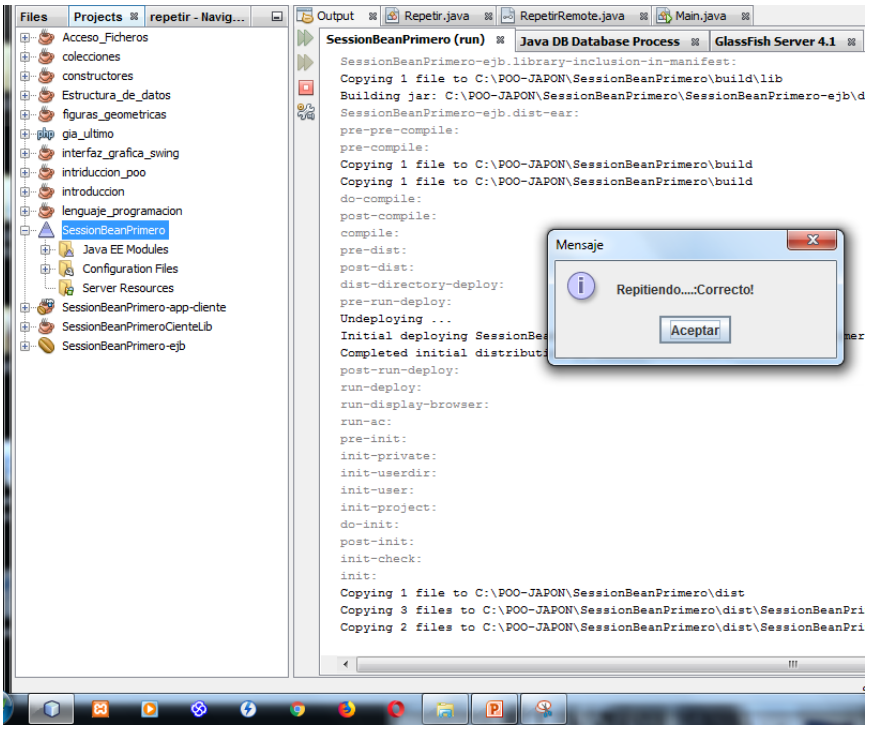

Ilustración 104: Resultado

UNIDAD 5: PATRONES DE DISEÑO

5.1. Introducción

Como programadores seguramente nos hemos dado cuenta de que muchas veces resolvemos un mismo problema de manera diferente. Al principio de nuestra carrera experimentamos diferentes formas de enfocar un problema, pero con el paso del tiempo nos gusta ir directamente al grano, aplicar la solución más escalable y más reutilizable.

En la programación se ha oído hablar de patrones de diseño e incluso se ha utilizado en el desarrollo de las aplicaciones permitiéndonos ahorrar tiempo, establecer un lenguaje común y validar el código.

5.2. Que es un patrón de diseño

“Los patrones de diseño son soluciones para problemas típicos y recurrentes que nos podemos encontrar a la hora de desarrollar una aplicación” (Charlascylon, 2014).

Según Sánchez (2017), un patrón de diseño son formas “estandarizadas” de resolver problemas comunes de diseño en el desarrollo de software y las ventajas de uso son:

- Conforman un amplio catálogo de problemas y soluciones
- Estandarizan la resolución de determinados problemas
- Condensan y simplifican el aprendizaje de las buenas prácticas
- Proporcionan un vocabulario común entre desarrolladores
- Evitan “reinventar la rueda”

5.3. Clasificación de los patrones de diseño

InformaticaPC.com (n.d.), menciona, que los patrones de diseño se clasifican en tres tipos diferentes dependiendo del tipo de problema que resuelven y estos pueden ser de creación, estructurales y de comportamiento.

	CREACIÓN	ESTRUCTURALES	COMPORTAMIENTO
CLASES	Factory Method		Interpreter Template Method
OBJETOS	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Ilustración 105: Tipos de patrones de diseño

5.3.1. Patrones de creación:

Utilizados para crear y configurar clases y objetos.

Los patrones creacionales están basados en dos conceptos:

a. Encapsular el conocimiento acerca de los tipos concretos que nuestro sistema utiliza. Estos patrones normalmente trabajarán con interfaces, por lo que la implementación concreta que utilicemos queda aislada.

b. Ocultar cómo estas implementaciones concretas necesitan ser creadas y cómo se combinan entre sí.

Los patrones creacionales más conocidos son:

- **Abstract Factory**: Este patrón resulta útil en casos en los que necesitemos crear familias de objetos relacionados o dependientes entre sí, sin especificar sus clases concretas.

  Observa su diagrama en UML:

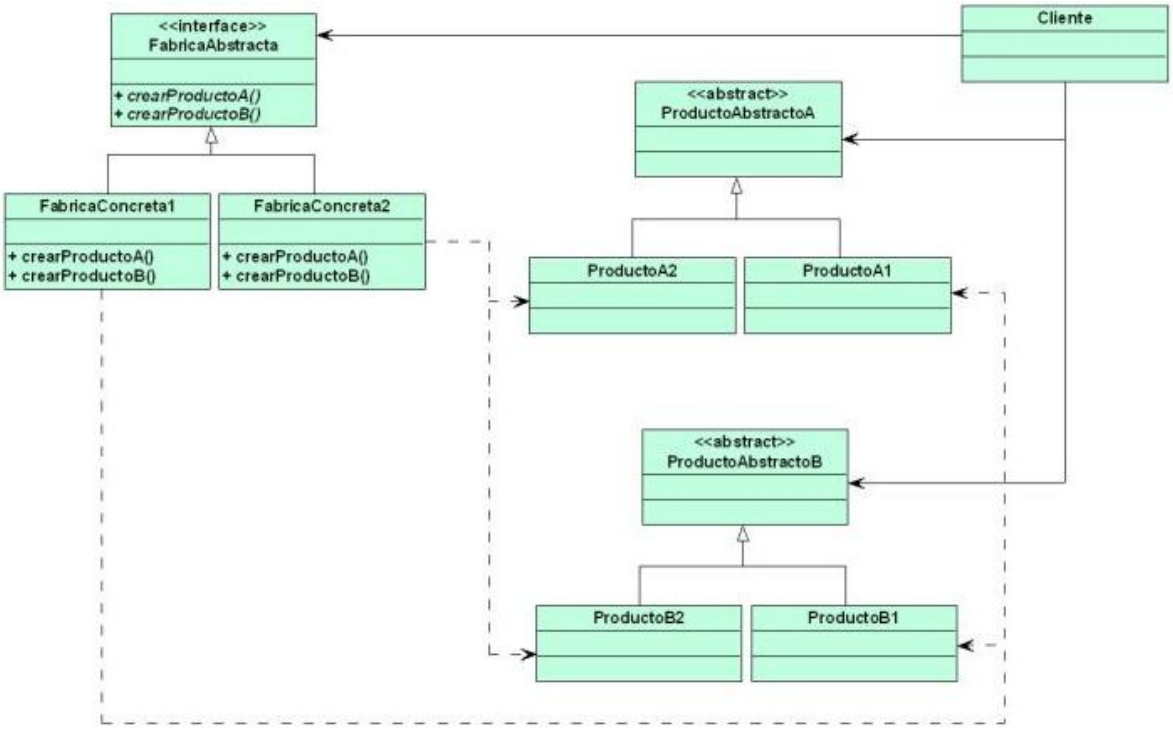

Ilustración 106: Abstract Factory

- **Factory Method:** Expone un método de creación, delegando en las subclases la implementación de este método.

Podemos utilizar este patrón cuando definamos una clase a partir de la que se crearán objetos pero sin saber de qué tipo son, siendo otras subclases las encargadas de decidirlo.

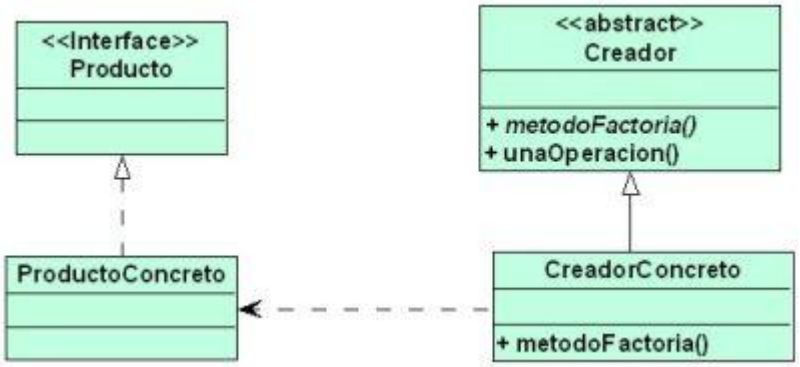

Ilustración 107: Factory Method

- **Builder**: Este patrón puede ser utilizado cuando necesitemos crear objetos complejos compuestos de varias partes independientes.

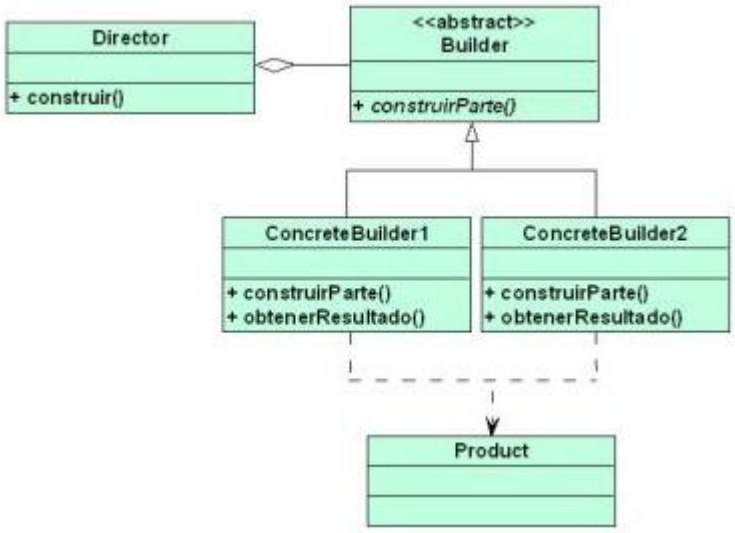

Ilustración 108: Builder

- **Singleton:** limita a uno el número de instancias posibles de una clase en nuestro programa, y proporciona un acceso global al mismo.

Dicha clase se creará de forma que tenga una propiedad estática y un constructor privado, así como un método público estático que será el encargado de crear la instancia (cuando no exista) y guardar una referencia a la misma en la propiedad estática (devolviendo ésta).

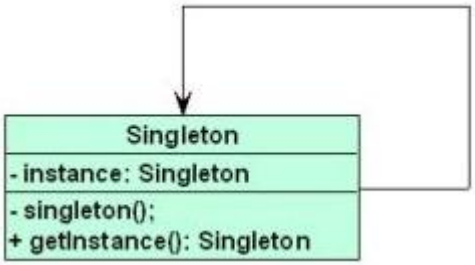

Ilustración 109: Singleton

- **Prototype:** Permite la creación de objetos basados en «plantillas». Un nuevo objeto se crea a partir de la clonación de otro objeto.

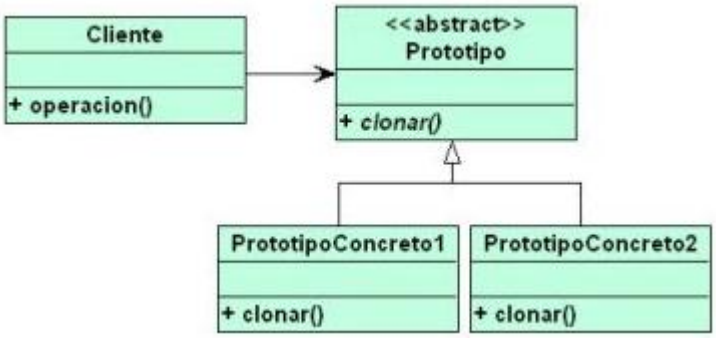

Ilustración 110: Prototype

Cuando nos disponemos a clonar un objeto es **importante** tener en cuenta si guarda referencias de otros o no. En este punto hay que distinguir las siguientes formas de replicarlos:

- **Copia superficial**: el objeto clonado tendrá los mismos valores que el original, guardando también **referencias a otros objetos** que contenga (por lo que si son modificados desde el objeto original o desde alguno de sus clones el cambio afectará a todos ellos).

- **Copia profunda**: el objeto clonado tendrá los mismos valores que el original así como **copias de los objetos** que contenga el original (por lo que si son modificados por cualquiera de ellos, el resto no se verán afectados).

5.3.2. Patrones estructurales:

Son patrones que nos facilitan la modelización de nuestro software especificando la forma en la que unas clases se relacionan con otras. Su objetivo es desacoplar las interfaces e implementar clases y objetos. Crean grupos de objetos.

Según Antonio Leiva (2016),Estos son los patrones estructurales que definió la Gang of Four:

- **Adapter**: Permite a dos clases con diferentes interfaces trabajar entre ellas, a través de un objeto intermedio con el que se comunican e interactúan. Este patrón permite que trabajen juntas clases con interfaces incompatibles.

Para ello, un objeto adaptador reenvía al otro objeto los datos que recibe (a través de los métodos que implementa, definidos en una clase abstracta o interface) tras manipularlos en caso necesario.

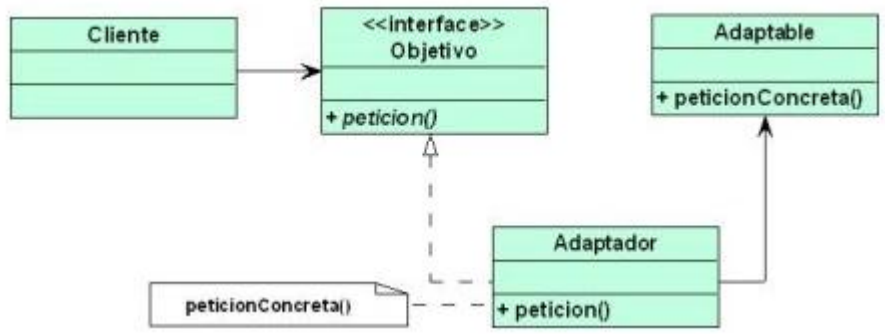

Ilustración 111: Adapter

- **Bridge**: Según el libro de **GoF** este patrón de diseño permite desacoplar una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.

Supongamos que tenemos una clase abstracta en la que se define un método que deberá implementar cada clase que herede de ella: ¿cómo haríamos si una clase hija necesitase implementarlo de forma que realizase acciones diferentes dependiendo de determinadas circunstancias?

En dichos casos nos resultaría útil el patrón **Bridge** (puente) ya que 'desacopla una abstracción' (un método abstracto) al permitir indicar (durante la ejecución del programa) a una clase qué 'implementación' del mismo debe utilizar (qué acciones ha de realizar).

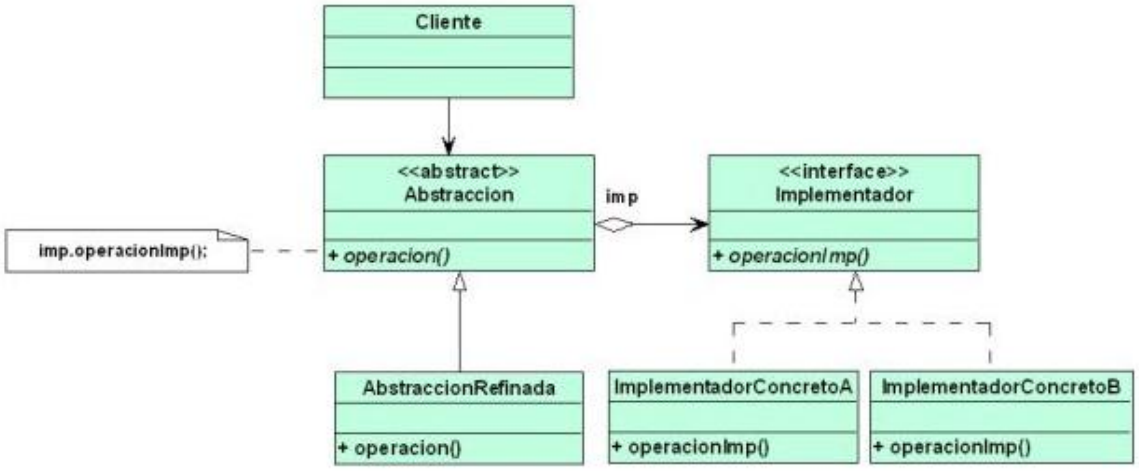

Ilustración 112: Bridge

- **Composite**: Facilita la creación de estructuras de objetos en árbol, donde todos los elementos emplean una misma interfaz. Cada uno de ellos puede a su vez contener un listado de esos objetos, o ser el último de esa rama.

Este útil patrón permite crear y manejar estructuras de objetos en forma de árbol, en las que un objeto puede contener a otro(s).

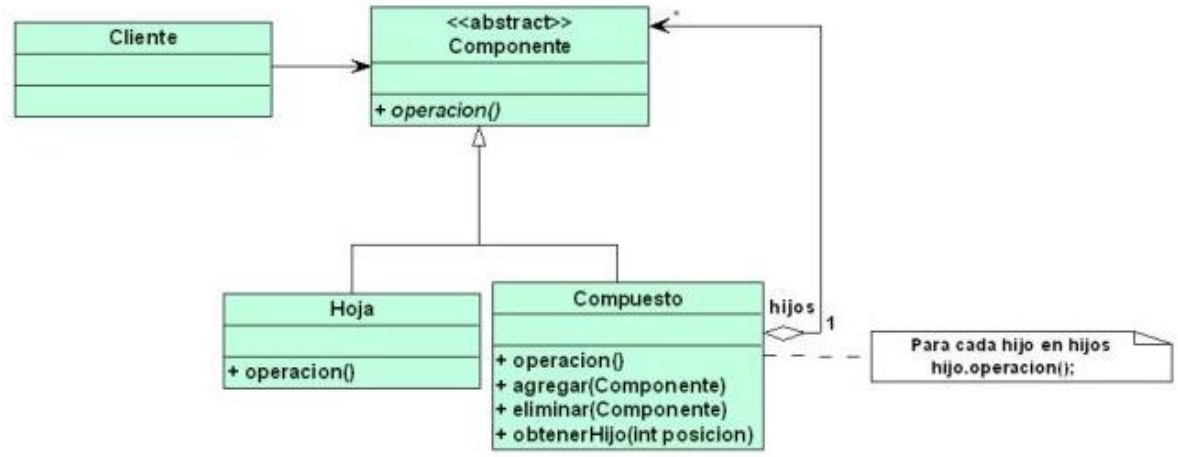

Ilustración 113: Composite

En este punto cabe aclarar que las estructuras de este tipo se componen de nodos (un objeto que a su vez contiene otros objetos) y Hojas (objetos que no contienen otros), y que ambos comparten una misma Interface que define métodos que deben implementar.

- **Decorator**: Permite añadir funcionalidad extra a un objeto (de forma dinámica o estática) sin modificar el comportamiento del resto de objetos del mismo tipo.

Sencillo e interesante patrón que permite añadir funcionalidades a un objeto en aquellos casos en los que no sea necesario o recomendable hacerlo mediante herencia

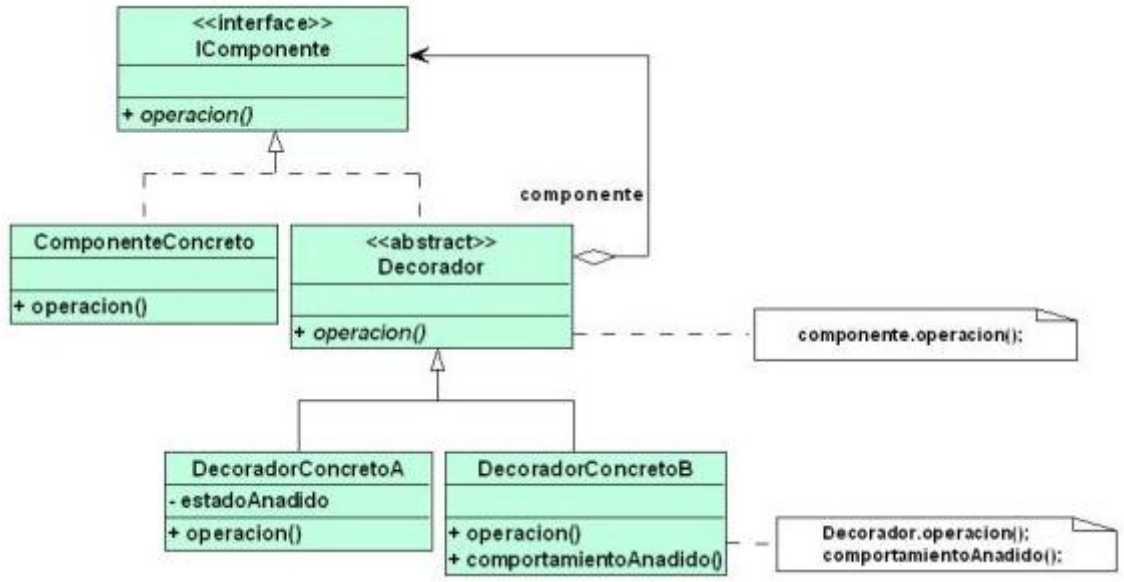

Ilustración 114: Decorator

- **Facade**: Una facade (o fachada) es un objeto que crea una interfaz simplificada para tratar con otra parte del código más compleja, de tal forma que simplifica y aísla su uso. Un ejemplo podría ser crear una fachada para tratar con una clase de una librería externa.

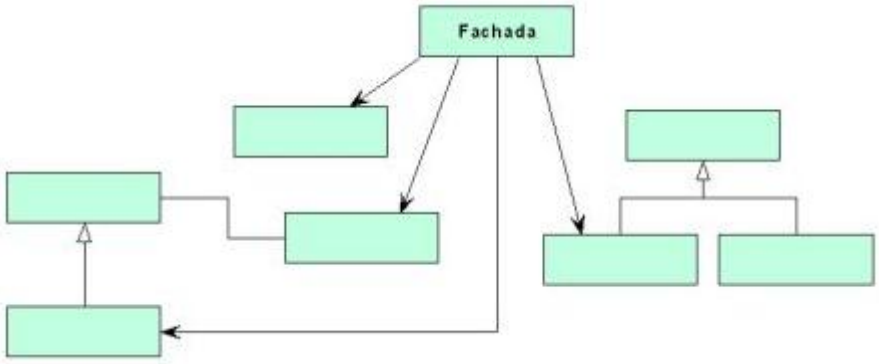

Ilustración 115: Facade

- **Flyweight**: Una gran cantidad de objetos comparte un mismo objeto con propiedades comunes con el fin de ahorrar memoria.

Este patrón resulta tremendamente útil para evitar crear un gran número de objetos similares, mejorando con ello el rendimiento de la aplicación.

No se trata de crear muchos objetos de forma dinámica, sino de crear sólo un objeto intermedio para cada entidad concreta.

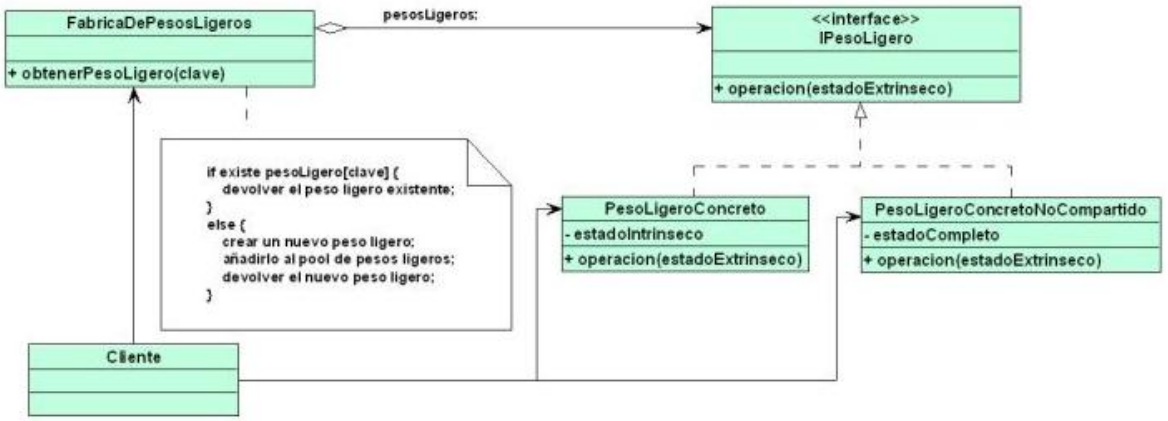

Ilustración 116: Flyweight

- **Proxy**: Es una clase que funciona como interfaz hacia cualquier otra cosa: una conexión a Internet, un archivo en disco o cualquier otro recurso que sea costoso o imposible de duplicar.

Este patrón se basa en proporcionar un objeto que haga de intermediario (proxy) de otro, para controlar el acceso a él.

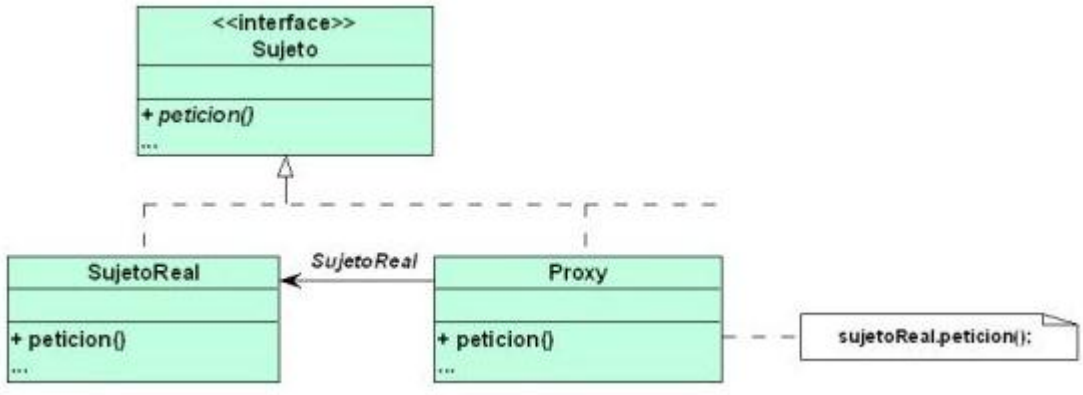

Ilustración 117: Proxy

5.3.3. Patrones de comportamiento

En este último grupo se encuentran la mayoría de los patrones, y se usan para gestionar algoritmos, relaciones y responsabilidades entre objetos.

- Los patrones de comportamiento son:

- **Command**: Son objetos que encapsulan una acción y los parámetros que necesitan para ejecutarse.

Este patrón resulta útil en escenarios en los que se han de enviar peticiones a otros objetos sin saber qué operación se ha de realizar, y ni tan siquiera quién es el receptor de dicha petición.

Dicho de otro modo: tenemos varios objetos que realizan acciones similares de forma diferente, y queremos que se procese la adecuada dependiendo del objeto solicitado.

El **Invocador** no sabe quién es el **Receptor** ni la acción que se realizará, tan sólo invoca un **Comando** que ejecuta la acción adecuada:

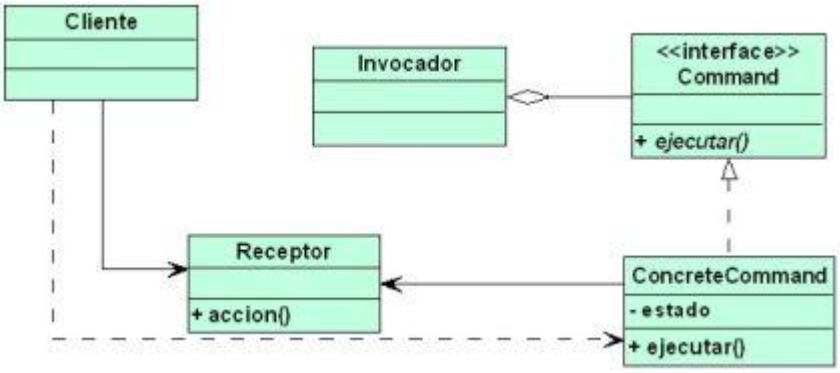

Ilustración 118: Command

- **Chain of responsibility**: se evita acoplar al emisor y receptor de una petición dando la posibilidad a varios receptores de consumirlo. Cada receptor tiene la opción de consumir esa petición o pasárselo al siguiente dentro de la cadena.

Este patrón puede resultarnos útil en casos en los que un objeto emisor de una petición desconozca qué objeto(s) podrá(n) atender a la misma.

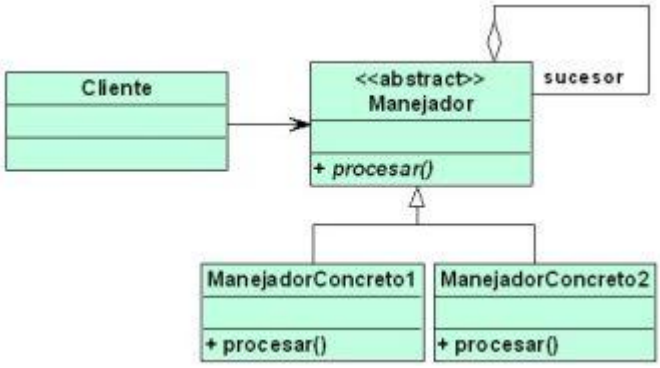

Ilustración 119: Chain of responsibility

- **Interpreter**: Define una representación para una gramática así como el mecanismo para evaluarla. El árbol de sintaxis del lenguaje se suele modelar mediante el patrón Composite.

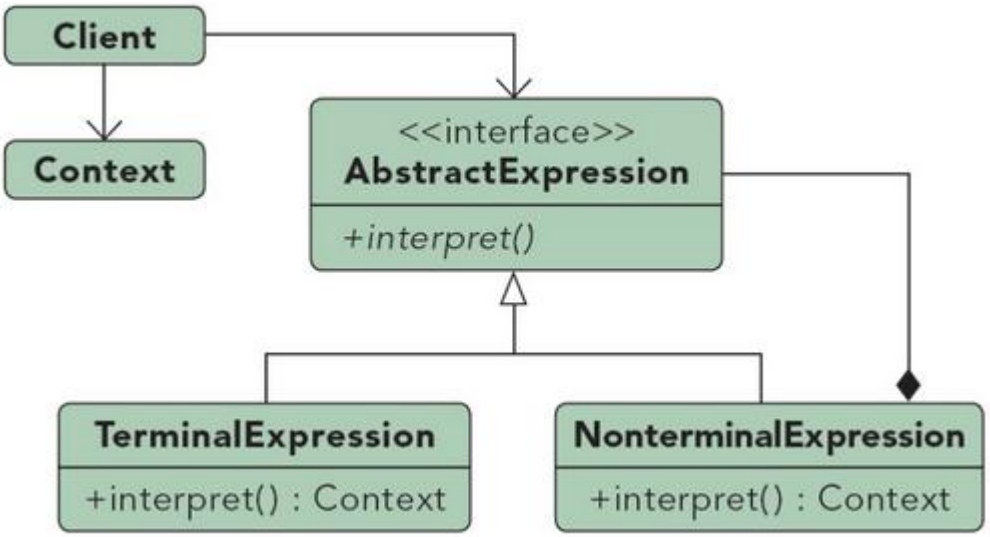

Ilustración 120: Interpreter

- **Iterator**: Se utiliza para poder movernos por los elementos de un conjunto de forma secuencial sin necesidad de exponer su implementación específica.

Así pues, este patrón de diseño nos resultará útil para acceder a los elementos de un array o colección de objetos contenida en otro objeto:

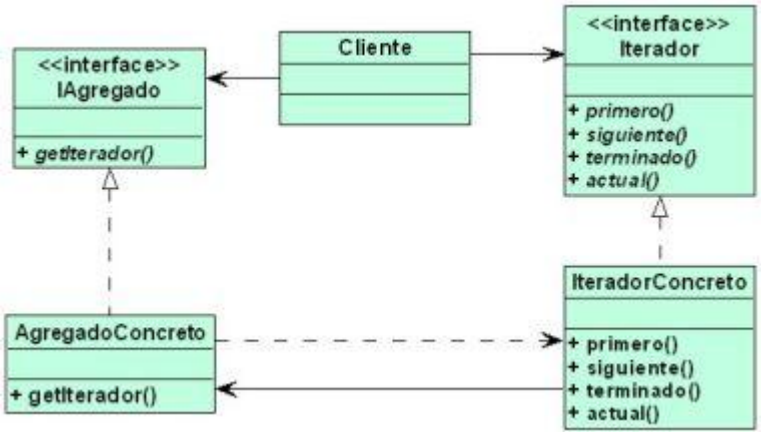

Ilustración 121: Iterator

- **Mediator**: Objeto que encapsula cómo otro conjunto de objetos interactúan y se comunican entre sí.

Una de las ventajas que ofrece la programación orientada a objetos (POO) es la posibilidad de reutilizar el código fuente, pero a medida que creamos objetos que se interrelacionan entre sí es menos probable que un objeto pueda funcionar sin la ayuda de otros.

Para evitar esto podemos utilizar el patrón **Mediator**, en el que se define una clase que hará de mediadora encapsulando la comunicación entre los objetos, evitándose con ello la necesidad de que lo hagan directamente entre sí.

- **Memento**: Este patrón de diseño es útil cuando manejamos un objeto que necesitaremos restaurar a estados anteriores (como por ejemplo cuando utilizamos la función de deshacer en un procesador de textos).

En la imagen se muestra su estructura según el libro de **GoF**.

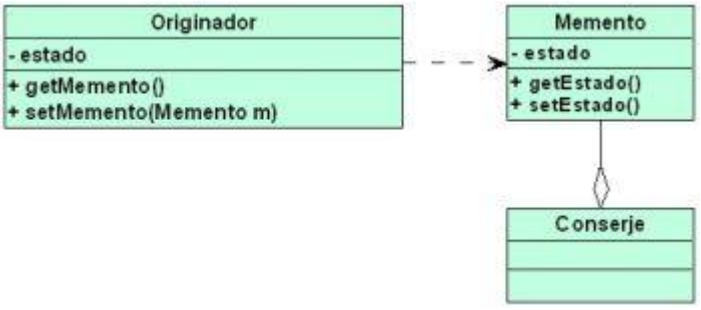

Ilustración 122: Memento

- **Observer**: Puede ser utilizado cuando hay objetos que dependen de otro, necesitando ser notificados en caso de que se produzca algún cambio en él.

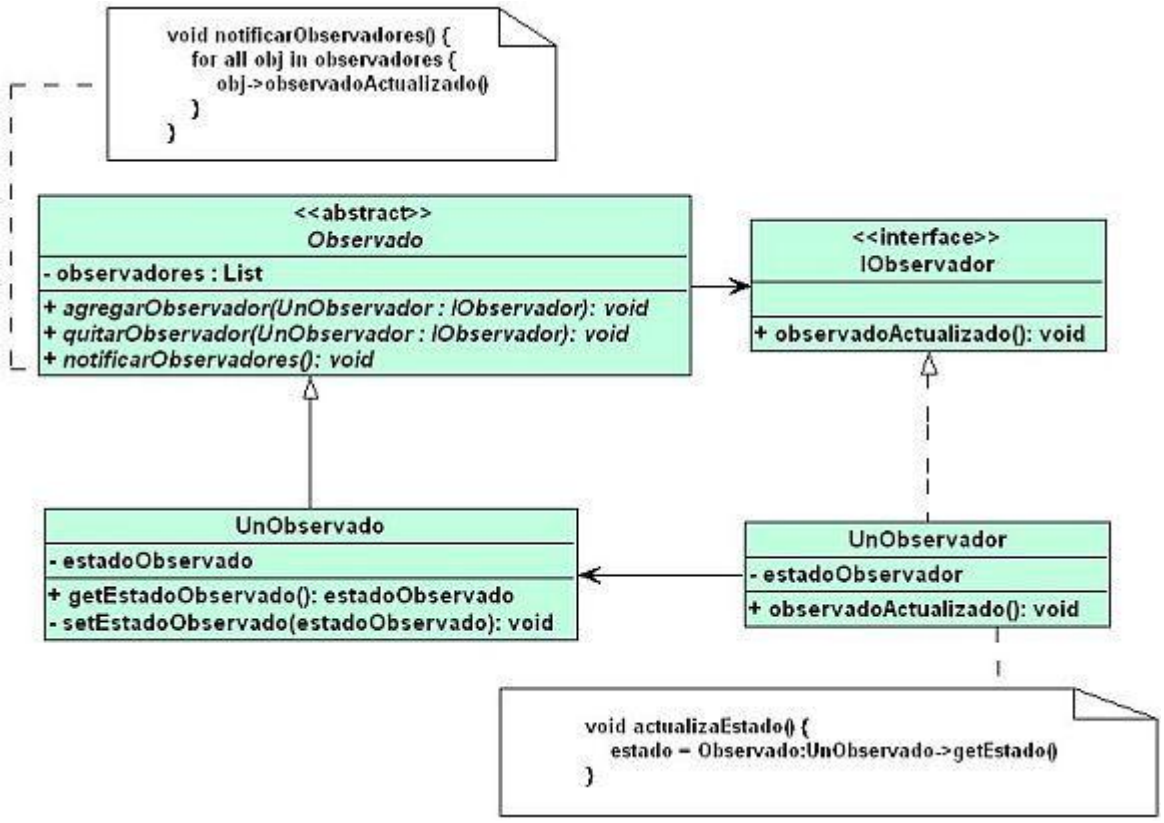

Ilustración 123: Observer

Se utilizarán los métodos agregarObservador() y quitarObservador() de la clase abstracta UnObservado para registrar en una lista qué objetos de tipo UnObservador deberán ser notificados o dejar de serlo cuando se produzca algún cambio en él (en tal caso recorrerá dicha lista para enviar una notificación a cada uno de ellos).

El mensaje será enviado a UnObservador (que implementa la **interface** IObservador) utilizando su método observadoActualizado().

- **State**: Permite modificar la forma en que un objeto se comporta en tiempo de ejecución, basándose en su estado interno.

Según el siguiente diagrama en UML podemos apreciar que el objeto cuyo estado es susceptible de cambiar (**Contexto**) contendrá una referencia a otro objeto que define los distintos tipos de estado en que se puede encontrar.

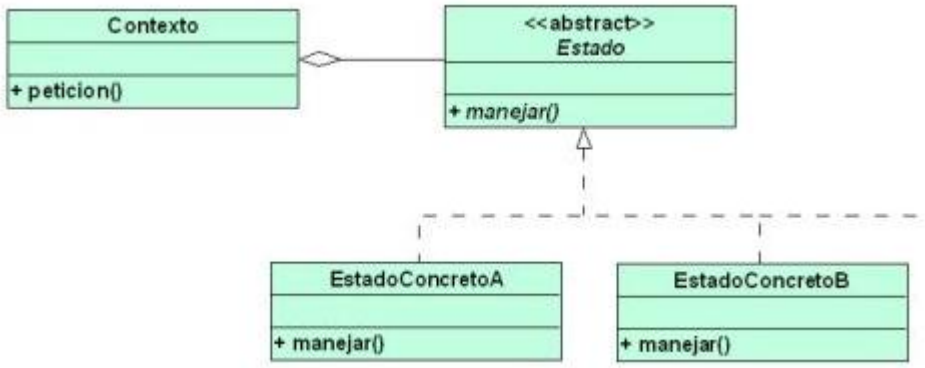

Ilustración 124: State

- **Strategy**: Permite la selección del algoritmo que ejecuta cierta acción en tiempo de ejecución.

Podemos hacer uso de este patrón para crear un objeto que pueda comportarse de formas diferentes (lo cual se definirá en el momento de su instanciación o creación).

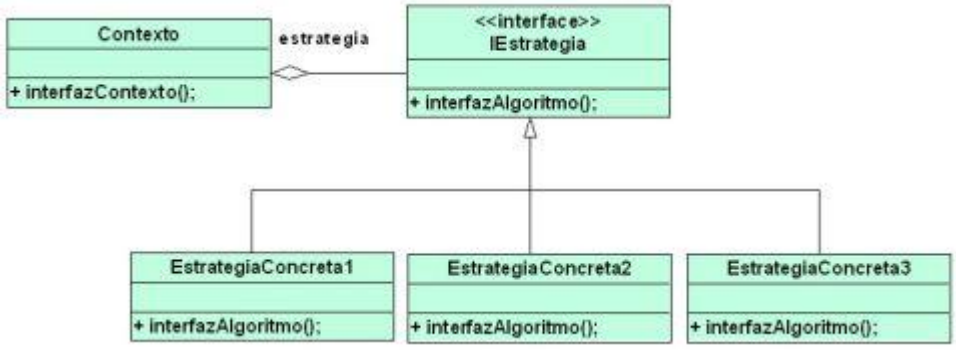

Ilustración 125: Strategy

- **Template Method**: Especifica el esqueleto de un algoritmo, permitiendo a las subclases definir cómo implementan el comportamiento real.

Este sencillo patrón resulta útil en casos en los que podamos implementar en una clase abstracta el código común que será usado por las clases que heredan de ella, permitiéndoles que implementen el comportamiento que varía mediante la reescritura (total o parcial) de determinados métodos.

La diferencia con la forma común herencia y sobreescritura de los métodos abstractos estriba en que la clase abstracta contiene un método denominado 'plantilla' que hace llamadas a los que han de ser implementados por las clases que hereden de ella.

A continuación veremos un ejemplo en el que utilizamos este patrón de diseño.

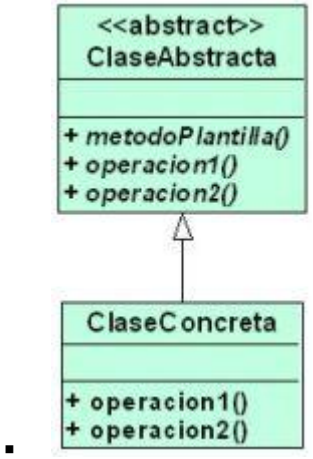

Ilustración 126: Template Method

- **Visitor**: Este patrón permite añadir funcionalidades a una clase sin tener que modificarla, siendo usado para manejar algoritmos, relaciones y responsabilidades entre objetos.

Así pues, nos resultará útil cuando necesitemos realizar operaciones distintas y no relacionadas sobre una estructura de objetos, aunque si lo utilizamos y luego tenemos que modificar alguna de las clases implicadas, hemos de tener en cuenta que se produce cierto nivel de acoplamiento entre ellas.

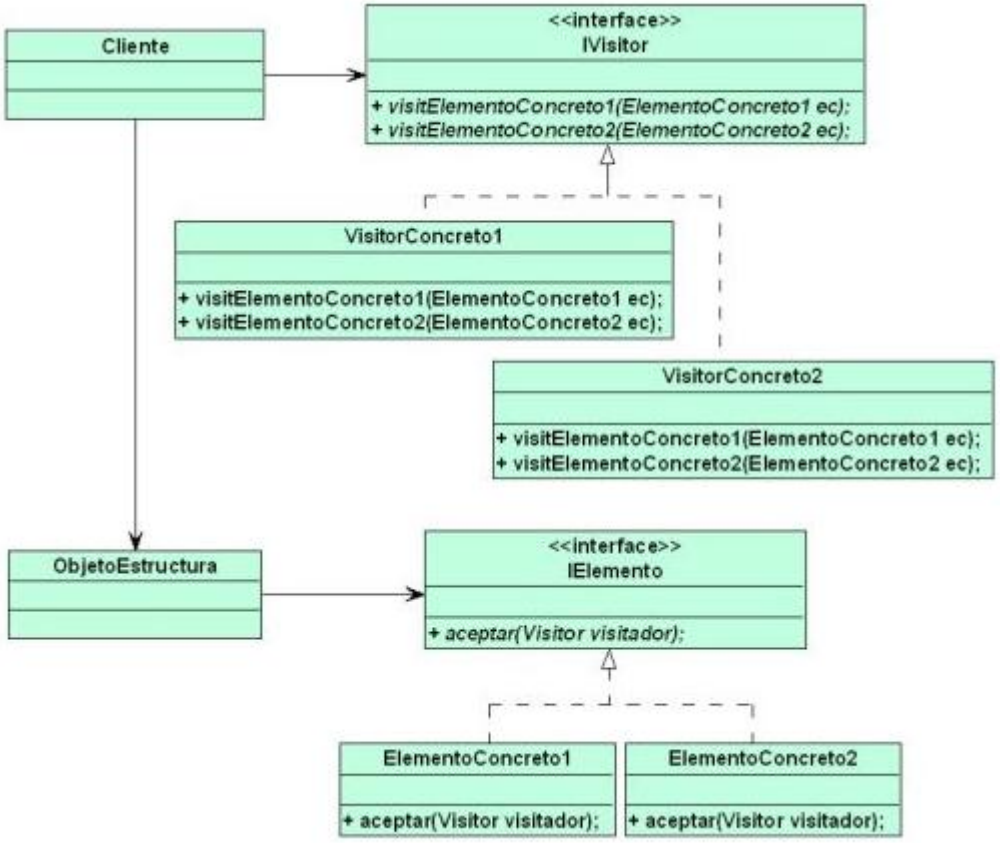

Ilustración 127: Visitor

B. Base de Consulta

TÍTULO	AUTOR	EDICIÓN	AÑO	IDIOMA	EDITORIAL
Colecciones en Java	Carolina de los Santos		2018	Español	http://lecasabe.com/colecciones-en-java/
Colecciones y tipos genéricos en Java	Jose I Acedo		2011	Español	http://programacion.jias.es/2011/10/colecciones-genericos-en-java/
Pilas en Java – Código libre	Bryan Acosta		2015	Español	http://codigolibre.weebly.com/blog/pilas-en-java
Aprendiendo – Programación estructurada	Carlos Pastrano		2016	Español	http://aprendamosprogramacionpastranoalmeida.blogspot.com/2016/11/pilas-en-java.html https://gist.github.com/alvareztech/41417c2edbca16df34f1157933e97ec0#file-main-java
Árboles en Java	Michelle Torres		2016	Español	https://blog.michelletorres.mx/arbol

					es-en-java/
Lectura y escritura de archivos de texto en java	Elivar Largo		2016	Español	https://www.ecodeup.com/como-escribir-y-leer-archivos-de-texto-plano-en-java/
Programación Orientada a Objetos: Persistencia	Universidad de Alcalá		2019	Español	https://docplayer.es/17202134-Programacion-orientada-a-objetos-tema-7-persistencia.html
Java Persistence API (JPA)	Oscar Blancarte		2019	Español	https://www.oscarblancarteblog.com/tutoriales/java-persistence-api-jpa/
Conectar Java con MySQL en Netbeans	Fernando Gaitán		2015	Español	https://fernando-gaitan.com.ar/conectar-java-con-mysql-en-netbeans/
TODO sobre Expresiones Lambda (FÁCIL) | Java desde Cero	Alex Walton		2018	Español	https://javadesdecero.es/avanzado/expresiones-lambda-java/
Entendiendo paso a paso las	Elivar Largo		2016	Español	https://www.ecodeup.com/entendie

expresiones Lambda en Java 8					ndo-paso-a-paso-las-expresiones-lambda-en-java/
JSP Servlet: Qué es un JavaBean?	Elivar Largo		2017	Español	https://www.ecodeup.com/jsp-servlet-que-es-un-java-bean/
Introducción a EJB 3.1 (I)	Cecilio Álvarez Caules		2013	Español	https://www.arquitecturajava.com/introduccion-a-ejb-3-1-i/
Introducción a la tecnología EJB	Anónimo			Español	http://www.jtech.ua.es/j2ee/2003-2004/abierto-j2ee-2003-2004/ejb/sesion01-apuntes.htm
¿Cuándo usar stateless, stateful o singleton en un bean de sesión?	Jose I Acedo		2012	Español	http://programacion.jias.es/2012/01/%C2%BFcuando-usar-stateless-stateful-singleton/
Investigación de la Plataforma J2EE y su aplicación práctica	Juan Manuel Barrios N.		2013	Español	https://users.dcc.uchile.cl/~jbarrios/J2EE/MemoriaJ2EE.pdf
Patrones de diseño: qué son y	Charlascylon Ruben		2014	Español	https://www.genbeta.com/desarroll

por qué debes usarlos					o/patrones-de-diseno-que-son-y-por-que-debes-usarlos
Patrones de Diseño Software tutorial.	InformaticaPC.com		2016	Español	https://informaticapc.com/patrones-de-diseno/factory-method.php
					https://codigofacilito.com/articulos/mvc-model-view-controller-explicado

C. Base práctica con ilustraciones

4. ESTRATEGIAS DE APRENDIZAJE

ESTRATEGIA DE APRENDIZAJE 1: Análisis y Planeación
Descripción: Discusión sobre las lecturas y videos.
Ambiente(s) requerido: Aula amplia con buena iluminación.
Material (es) requerido: Infocus.
Docente: Con conocimiento de la materia.

5. ACTIVIDADES

- Controles de lectura
- Taller en clase
- Evaluación final

Se presenta evidencia física y digital con el fin de evidenciar en el portafolio de cada aprendiz su resultado de aprendizaje. Este será evaluable y socializable

6. EVIDENCIAS Y EVALUACIÓN

Tipo de Evidencia	**Descripción (de la evidencia)**
De conocimiento:	Prueba escrita cerrada Rúbrica de evaluación

Elaborado por: **(Docente)**	**Revisado Por:** **(Coordinador)**	**Reportado Por:** **(Vicerrector)**

INSTITUTO SUPERIOR TECNOLÓGICO

JAPÓN

Amor al conocimiento

Guía Metodológica Lenguaje de programación II
Carrera Desarrollo de Software
Ing. Yecenia Cevallos
2019

Coordinación Editorial Dirección:
Lucía Begnini Dominguez.
Coordinación Editorial:
Milton Altamirano Pazmiño, Alexis Benavides.
Diagramación: Sebastián Gallardo.
Corrección de Estilo: Lucía Begnini.
Diseño: Sebastián Gallardo.
Instituto superior tecnológico Japón
AMOR AL CONOCIMIENTO