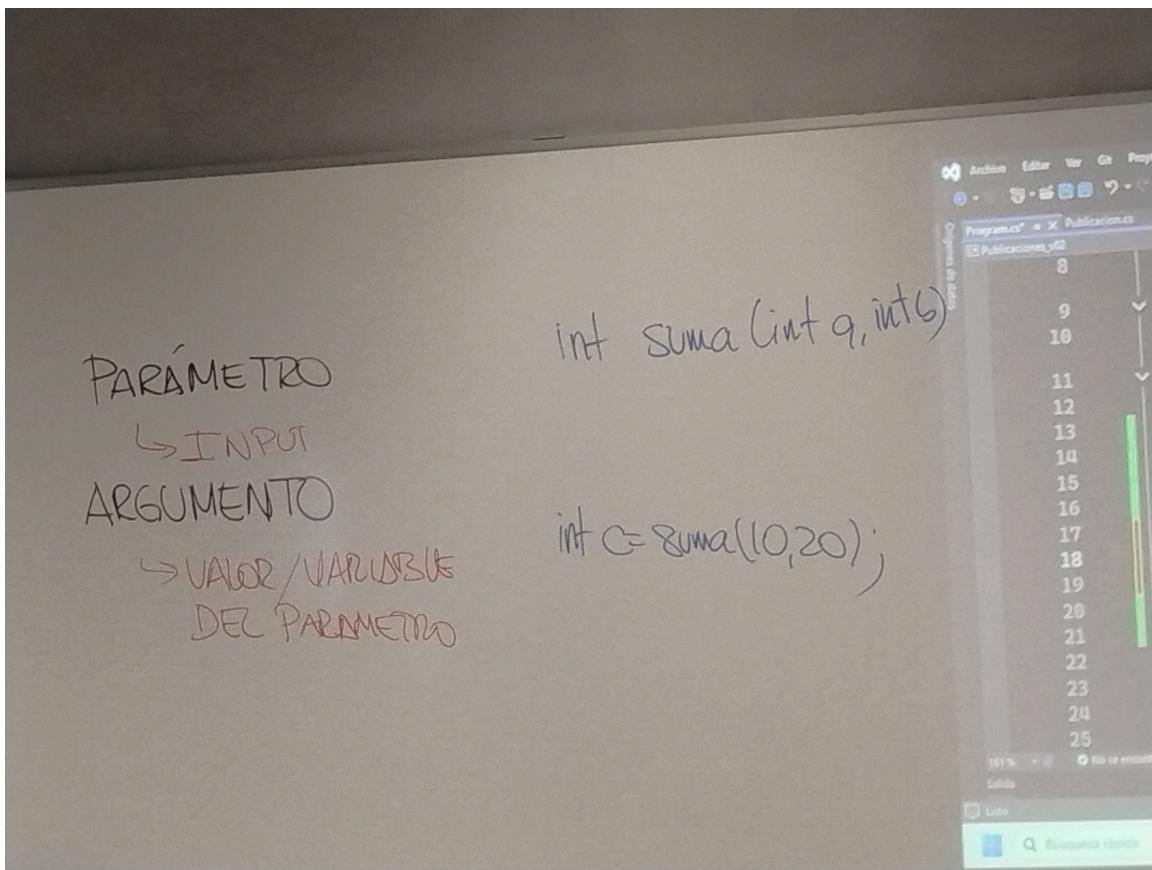
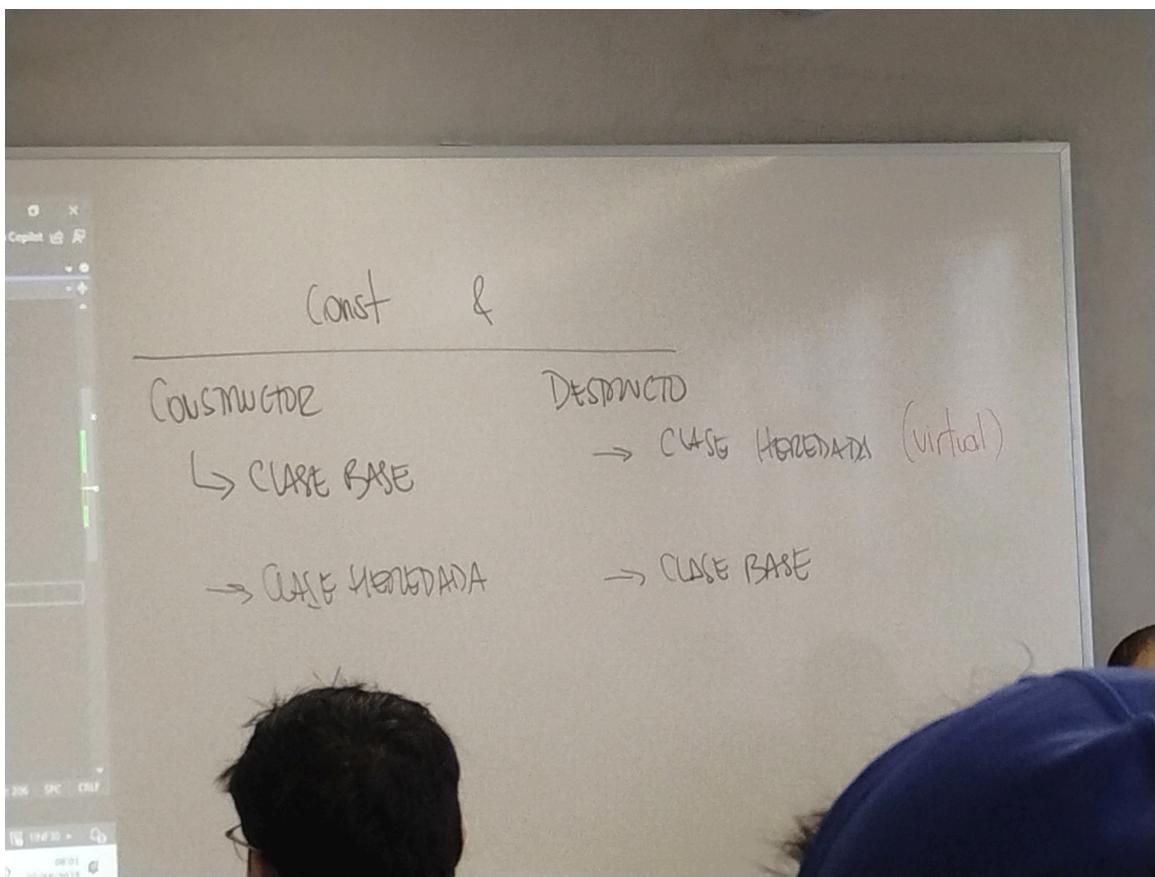


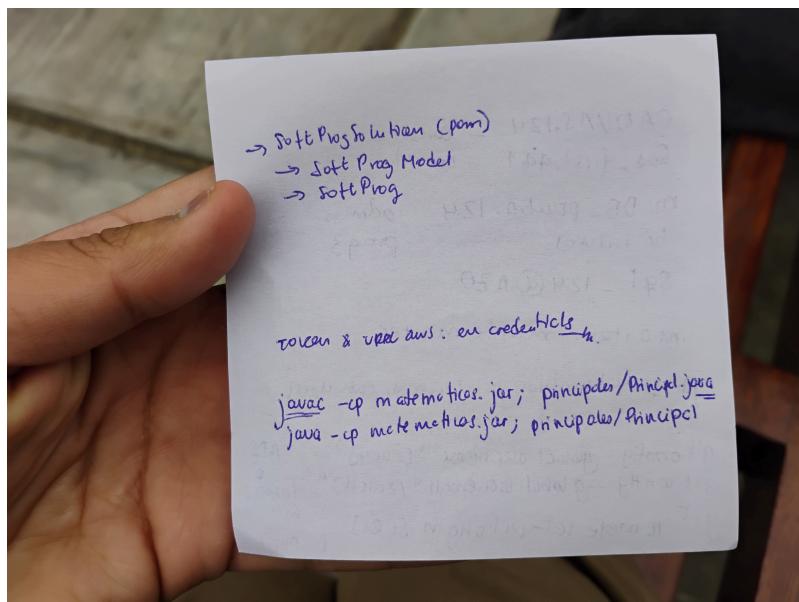
Programación 3

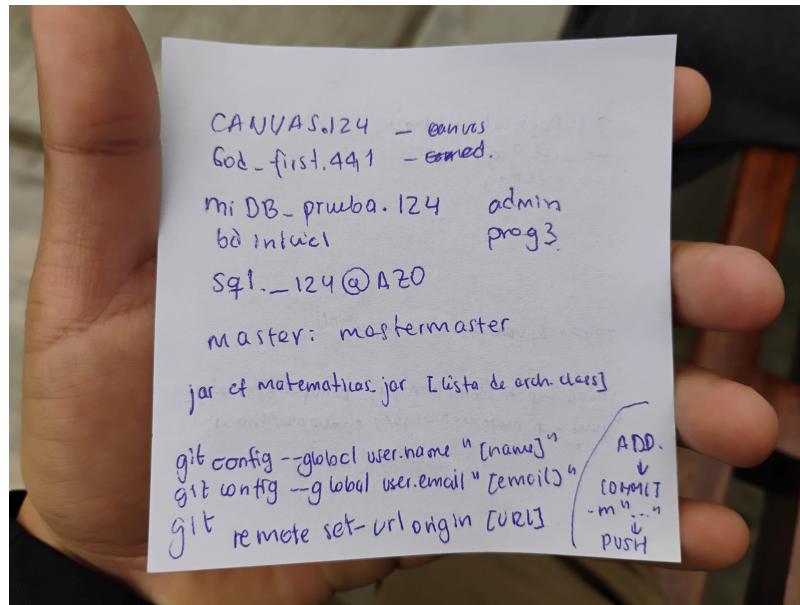




Clase > referencia

Jerarquía alta izquierda





Sesión 4

Sesion 5

Sesión 6

Sesion 7

Sesión 8

Sesión 11

SESION 1: INTRODUCCIÓN

Traducción: interpretación y compilación

Traducción

Traducir el lenguaje de un tipo a otro tipo.

La traducción se define como el proceso de convertir un programa escrito en un lenguaje de **alto nivel** a un **lenguaje de máquina** que pueda ser ejecutado por la

computadora. Este proceso puede realizarse mediante compiladores o intérpretes.



C#: Un compilador es un intérprete.



- Es el proceso de convertir un programa en lenguaje de alto nivel a lenguaje máquina.
- Se puede realizar por **compiladores** o **intérpretes**.

Compilador

La **compilación** es una traducción.

Un **compilador** es un programa que lee un programa escrito en un lenguaje de programación (**programa fuente**) y lo traduce en un programa equivalente escrito en otro lenguaje de programación (**programa destino**). En caso que el programa destino sea un programa ejecutable, este puede ser invocado por el usuario para procesar datos de entrada y así obtener los datos de salida que requiera computar.



C: Es un lenguaje compilado.

Compilación



- Traduce todo el código fuente a un programa ejecutable antes de ejecutarlo.
- Ejemplo: C, C++, Java (al compilar a bytecode).
- Ventaja: ejecución más rápida.
- Desventaja: tiempo de compilación, no detecta errores en tiempo de ejecución directamente.

Intérprete

El **intérprete** es otro tipo de traductor. En lugar de producir un programa destino en el proceso de traducción, el intérprete **directamente ejecuta las operaciones** especificadas en el programa fuente a través de datos de entradas proveídos por el usuario.



PSeint: Es un lenguaje interpretado.

Python: Es interpretado.

Interpretación



- Ejecuta el código línea por línea directamente, sin generar un archivo ejecutable.
- Ejemplo: Python, JavaScript.
- Ventaja: ideal para pruebas rápidas y debugging.
- Desventaja: ejecución más lenta.

Java Development Kit

Es un lenguaje **multiplataforma**. Funciona en Windows, Linux, MacOS. Está orientado a objetos (POO).



Java: Es un lenguaje compilado e interpretado.

JDK (Java Development Kit) `java` `javac` `jar`

JRE (Java Runtime Environment) `awt` `JDBC` `I/O` `Math`

JVM (Java Virtual Machine) `interpretación` `garbage collector`



Java Development Kit (JDK)

- Incluye:
 - Herramientas como `javac` (compilador), `java` (ejecución), `jar`, etc.
 - **JRE (Java Runtime Environment)**: lo necesario para ejecutar programas Java.
 - **JVM (Java Virtual Machine)**: interpreta el bytecode.

Hola Mundo en Java

```
public class HolaMundo {  
    public static void main(String[] arg) {  
        System.out.println("Hola Mundo!");  
    }  
}
```

Pasos (cmd):

1. Compilar: `javac HolaMundo.java`

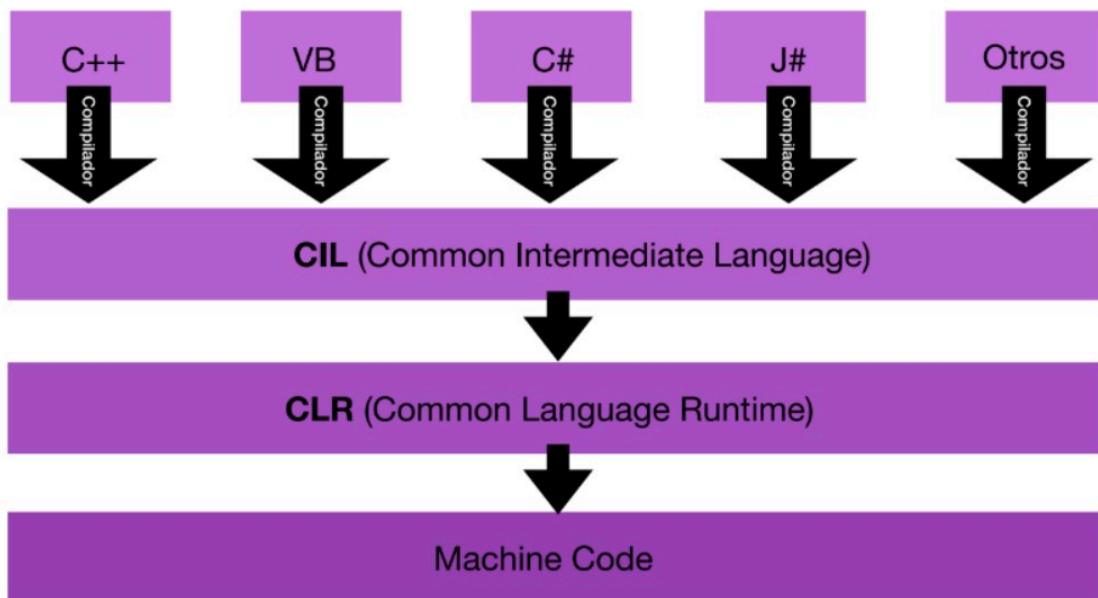
2. Ejecutar: `java HolaMundo`

Extra: Verificación de archivos generados: `dir`

Configuración Sublime Text para Java

```
{  
    "cmd": "javac ${file_name} && java ${file_base_name}",  
    "shell": true  
}
```

.Net Framework Developer



- Lenguajes como C#, VB.NET, etc., compilan a **CIL (Common Intermediate Language)**.
- Este es interpretado por el **CLR (Common Language Runtime)**.
- Equivalente a JVM en Java.

Hola Mundo en C#

```

using System;           using es una directiva que permite usar los tipos de
namespace HolaMundo   datos definidos en un espacio de nombres.

{
    class Program       class es una palabra reservada sirve para declarar una clase en C#
    {
        static void Main(string[] args)      Método Main
        {
            Console.WriteLine("Hola Mundo!");  Operación de salida
        }
    }
}

```

Pasos:

1. Compilar: `csc HolaMundo.cs`
2. Ejecutar: `HolaMundo.exe`

Configuración Sublime Text para C#

```
{
  "cmd": "csc ${file_name} && ${file_base_name}",
  "shell": true
}
```

- **Namespace:** contenedor lógico.
- **Static:** Hace que un atributo o método sea de la clase, así puedes utilizar ese método sin instanciar un objeto.



C#

C Sharp Compiler → CSC



JAVA: programa → **javac** → **.class** byte code → java (**intérprete**)
#C: programa → **CSC** → **.exe**



Clase: es la implementación de un tipo abstracto de datos que tiene un comportamiento y un conjunto de datos.

Objeto: un elemento/una instancia.

Instancia: referencia a un objeto.



Encapsulamiento: ocultar cómo se realizan las cosas internamente, abstraerlas.

Abstracción: ocultar.

Patrón DAO: AbstRAE, oculta.

Método complejo: cualquier método que no sea el constructor o selector.

El constructor define/implementa una obligación.

Resumen



Conceptos Clave: Traducción, Compilación e Interpretación

- **Traducción:** Proceso de convertir código fuente (alto nivel) a lenguaje máquina.
- **Dos tipos principales:**
 - **Compilador:** Traduce todo el programa a un ejecutable antes de correrlo.
 - **Intérprete:** Ejecuta directamente cada línea del código fuente en tiempo real, sin generar un archivo ejecutable.



Comparación entre Compilador e Intérprete

Característica	Compilador	Intérprete
Traducción	Completa antes de ejecución	Línea por línea durante la ejecución

Característica	Compilador	Intérprete
Salida	Código máquina ejecutable	No genera código destino
Velocidad	Más rápido en ejecución	Más lento (traduce en tiempo real)
Ejemplo	C, C++, Java (compilación previa)	Python, JavaScript

Java Development Kit (JDK)

- **JDK**: Herramientas para desarrollar aplicaciones en Java.
- **JRE**: Entorno para ejecutar programas Java.
- **JVM**: Máquina virtual que interpreta el bytecode de Java.
- Herramientas comunes: `javac` (compilador), `java` (ejecutor), `jar`.

Hola Mundo en Java:

```
public class HolaMundo {
    public static void main(String[] args) {
        System.out.println("Hola Mundo!");
    }
}
```

Pasos:

1. Compilar: `javac HolaMundo.java`
2. Ejecutar: `java HolaMundo`

Configuración en Sublime Text (Java):

```
{
    "cmd": "javac ${file_name} && java ${file_base_name}",
    "shell": true
}
```

.NET Framework y C#

- Lenguajes como C#, VB, J# se compilan a **CIL (Common Intermediate Language)**.
- **CLR (Common Language Runtime)** ejecuta el CIL en la máquina real.
- **Hola Mundo en C#:**

```
using System;

namespace HolaMundo {
    class Program {
        static void Main(string[] args) {
            Console.WriteLine("Hola Mundo!");
        }
    }
}
```

Pasos:

1. Compilar: `csc HolaMundo.cs`
2. Ejecutar: `HolaMundo.exe`

Configuración en Sublime Text (C#):

```
{
  "cmd": "csc ${file_name} && ${file_base_name}",
  "shell": true
}
```

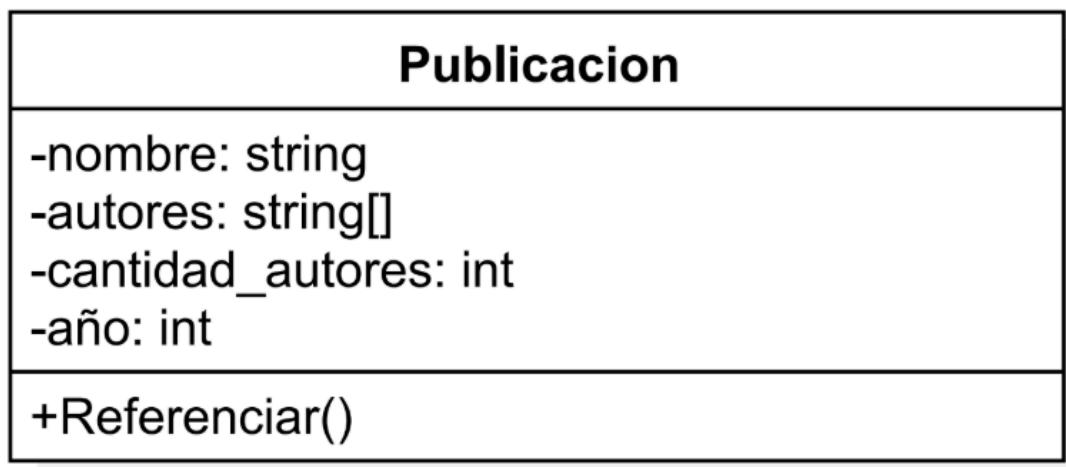
SESION 2: POO

Elementos Clave de la Programación Orientada a Objetos (POO)

◆ Versión 1 – Introducción a clases y objetos

- **Clase:** Molde o plantilla (ej. `Publicacion`)
- **Objeto / Instancia:** Ejemplar concreto de la clase.
- **Atributos:** Datos miembros (como `nombre`, `año`, etc.).
- **Modificadores de acceso:** `private`, `protected`, `public`.
- **Constructores:**
 - Por defecto
 - Con parámetros
 - Constructor copia
- **Encapsulamiento:** Ocultar implementación interna.
- **Métodos complejos y sobreescritura.**

UML ejemplo v1:

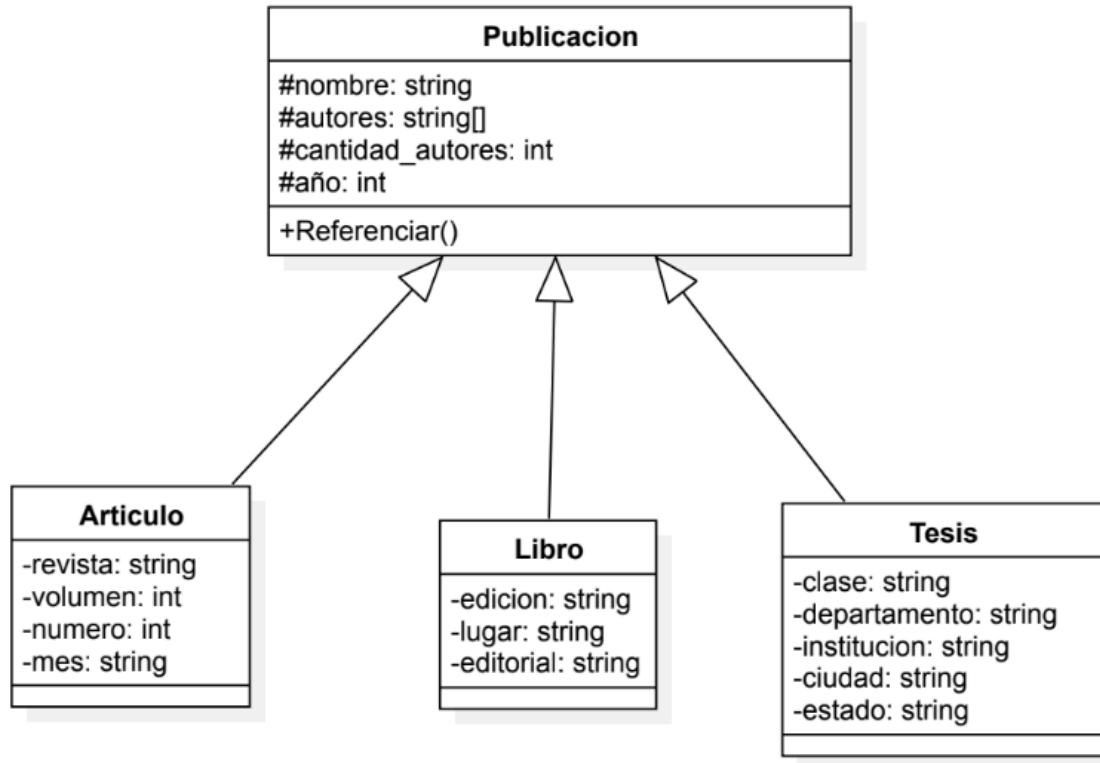


◆ Versión 2 – Herencia

- **Herencia:** Permite que una clase derive de otra.
- **Clases derivadas:** `Articulo`, `Libro`, `Tesis` heredan de `Publicacion`.
- **Invocación al constructor de la superclase.**

- **Atributos protegidos (#):** visibles en subclases.

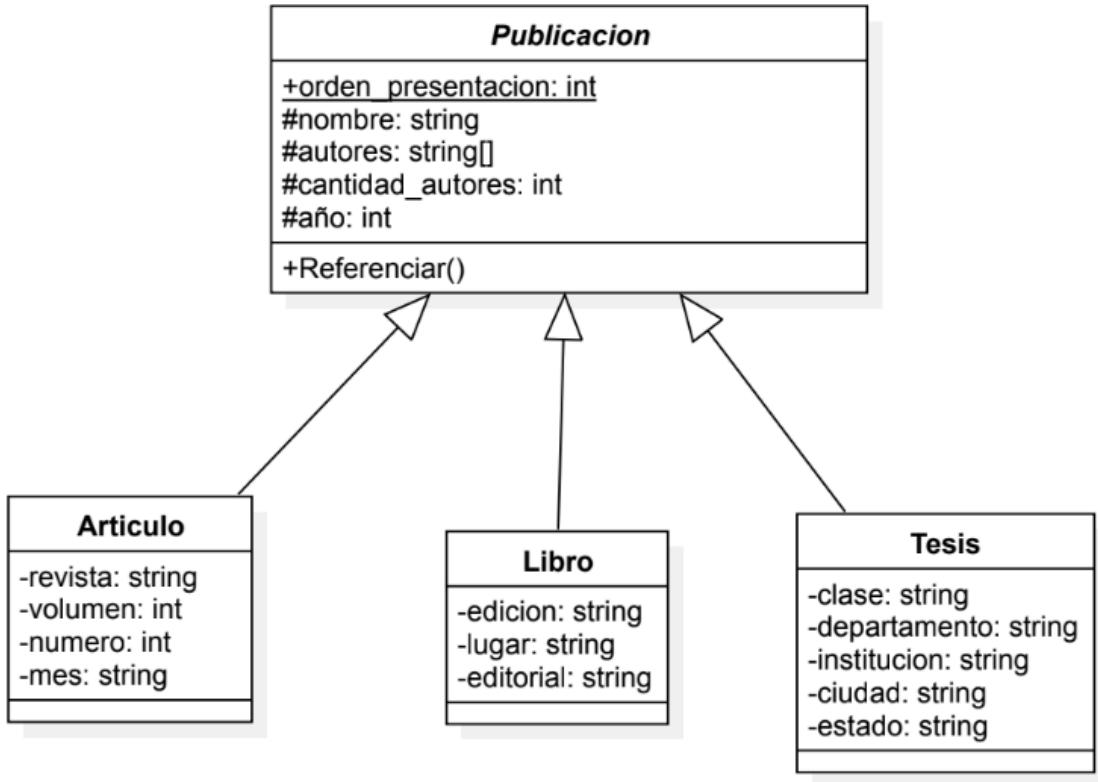
UML ejemplo v2:



◆ Versión 3 – Abstracción y polimorfismo

- **Clase abstracta:** No se puede instanciar directamente.
- **Método abstracto:** Definido sin implementación en la clase base.
- **Polimorfismo:** Capacidad de tratar objetos de distintas subclases como objetos de la superclase.
- **Arreglo de objetos:** Permite almacenar múltiples publicaciones.
- **Atributos estáticos:** Compartidos entre todas las instancias.

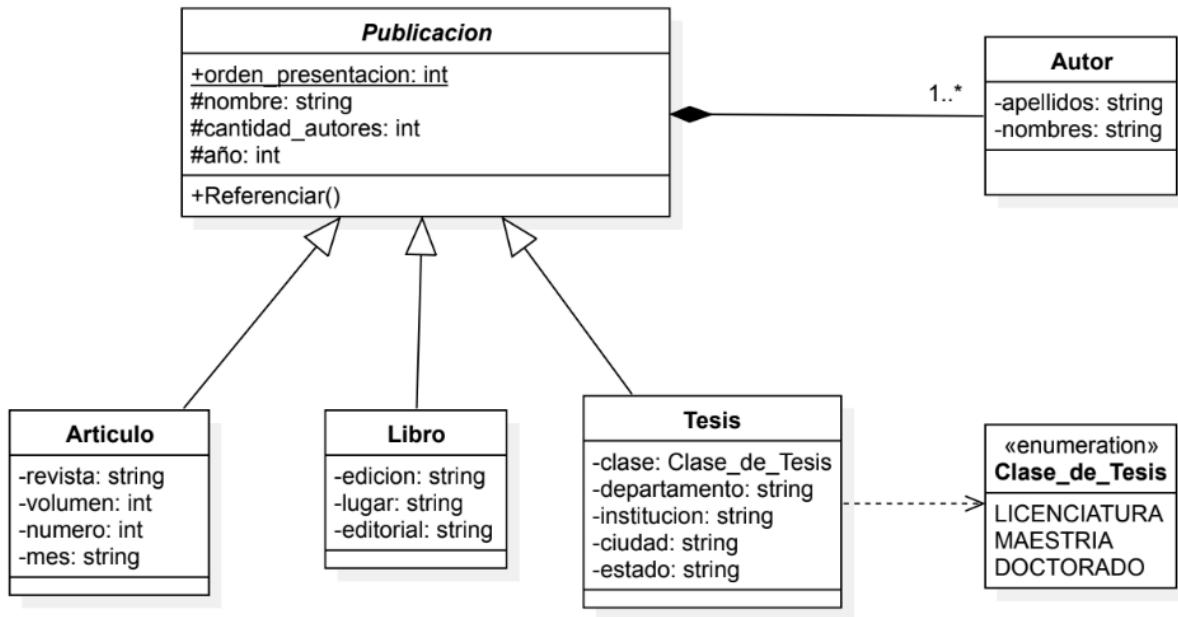
UML ejemplo v3:



◆ Versión 4 – Enums, clases agregadas y anidadas

- **Tipos enumerados (enum)**: Conjunto de constantes (ej. tipo de tesis).
- **Clases agregadas**: Una clase contiene instancias de otras (**Autor** dentro de **Publicacion**).
- **Clases anidadas**: Clases definidas dentro de otras.

UML ejemplo v4:



SESION 3: LIBRERIAS EN JAVA

Paquetes en Java

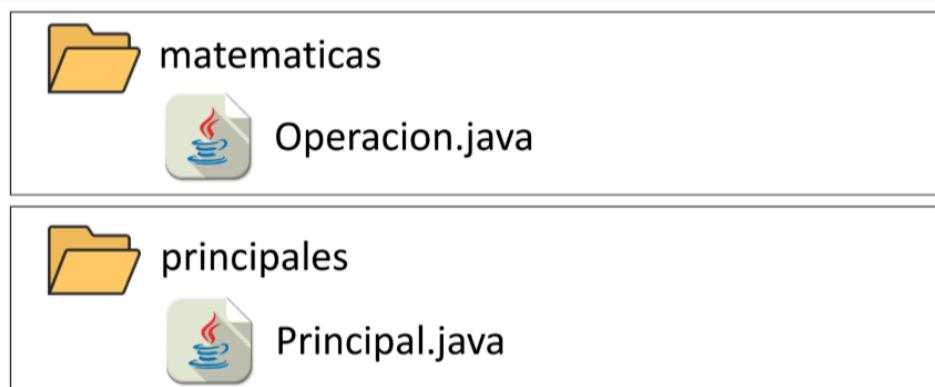
- Un paquete en Java es un **contenedor lógico y físico** que organiza las partes de un programa y que por lo general tiene una funcionalidad y elementos comunes.
- Los paquetes se organizan en **directorios o carpetas** de estructura jerárquica que agrupan distintos artefactos de software como clases, interfaces, enumerados y otros paquetes.
- Cada clase en Java debe **pertenecer a un paquete**, ya sea explícitamente especificado o, formando parte del paquete predeterminado.
- Ayuda a **organizar el código**, evitar conflictos de nombres y reutilizar componentes.

Convenciones para nombres

- Usar nombre en **minúsculas**.

- En empresas u organización, usar el dominio Web invertido:
 - Dominio: miempresa.com
 - Paquete: com.miempresa.proyecto.modulo
- Evitar nombres cortos.
- Se recomienda organizar los paquetes por módulo o elementos de arquitectura.

Ejemplo



Operacion.java

```

package matematicas;
public class Operacion {
    public int sumar(int a, int b) {
        return a + b;
    }
    public int restar(int a, int b) {
        return a - b;
    }
}
  
```

Principal.java

```

package principales;
import matematicas.*;

public class Principal {
    public static void main(String[] args) {
        Operacion op = new Operacion();
        int x = op.sumar(10, 90);
        int y = op.restar(10, 90);
        System.out.println(x);
        System.out.println(y);
    }
}

```

Se compila empezando por las clases menos dependientes (**cmd**).

1. `javac matematicas/*.java`
2. `javac principales/*.java`
3. `java principales/Principal`

Librerías

- Una librería es un programa cuyos elementos pueden ser utilizados por otros programas.
- La forma de crear y utilizar una librería dependen del lenguaje de programación o del sistema operativo.

Librerías en JAVA

- Están estrechamente relacionadas a los **directorios**.
- Una librería es un directorio/paquete con un conjunto de archivos **CLASS** (`.class`) que forman parte de la librería.
- Para encontrar una librería, tanto el compilador como el intérprete utilizan la variable de entorno **CLASSPATH**.
- Se pueden **empaquetar en archivos** `.jar` (Java Archive).

Archivos JAR

- Comprimen uno o más paquetes conservando la información acerca de los directorios (paquetes) donde se encuentran cada archivo `.class`.
- Pueden ser utilizados en CLASSPATH como lugares de búsqueda de archivos `.class`.
- Constituyen una mejor forma distribuir un conjunto de archivos `.class`, como los que forman una aplicación, dado que se tiene la opción de comprimir estos.

Creación

1. Compilar tus clases

Generar los archivos **CLASS** que conformarán la librería.

Primero compila el archivo `.java` para generar el archivo `.class`:

```
javac matematicas/Operacion.java
```

2. Crear el archivo JAR (la librería)

Empaquetar los archivos **CLASS** mediante el siguiente comando:

```
jar cvf "nombre del paquete" [clases]
```

Ahora, vas a crear la **librería** `paquete1.jar` que contiene esa clase.

```
jar cvf paquete1.jar matematicas/*.class
```



Qué significa:

- `jar` → herramienta para crear archivos `.jar`
- `c` → crear
- `v` → ver los detalles en consola
- `f` → nombre del archivo a generar
- `matematicas/*.class` → incluye todos los `.class` de esa carpeta

Resultado: tienes un archivo llamado `paquete1.jar`, que guarda la clase `Operacion`.

3. Usar la librería para compilar otro programa

Ahora, compilas el programa principal (`Principal.java`) que **usa esa librería**:

```
javac -cp paquete1.jar principales/*.java
```

`-cp` significa "classpath", le estás diciendo al compilador:

"Busca las clases en este archivo .jar también"

4. Ejecutar el programa que usa la librería

```
java -cp paquete1.jar; principales.Principal
```



En Windows se usa `;` para separar rutas. En Linux/Mac se usa `:`.

¿Y si quiero crear un JAR ejecutable?

Un **JAR ejecutable** es una caja mágica que solo necesitas **doble click o un solo comando para correrla**.

5. Crear el archivo `MANIFEST.MF`

Crea un archivo de texto llamado `MANIFEST.MF` con este contenido:

```
Manifest-Version: 1.0  
Created-By: 1.8.0_92 (Oracle Corporation)  
Class-Path: paquete1.jar  
Main-Class: principales.Principal
```

Le estás diciendo:

- **Class-Path** : dónde están las clases extra que vas a usar.
- **Main-Class** : cuál es la clase que contiene el método **main**.



Como es un txt, escribe el siguiente comando para **renombrar correctamente** el archivo:

```
rename MANIFEST.MF.txt MANIFEST.MF
```

6. Crear el JAR ejecutable

```
jar cvfm mipaqueteejecutable.jar MANIFEST.MF principales/*.class
```

Este comando:

- **c** → crea
- **v** → muestra detalles
- **f** → define el nombre del **.jar** de salida
- **m** → indica el archivo **MANIFEST.MF**

Esto crea el archivo **mipaqueteejecutable.jar**, que puedes ejecutar así:

```
java -jar mipaqueteejecutable.jar
```

RESUMEN

1. Paquetes en Java

- **Definición:** Un paquete es un contenedor de clases, interfaces y otros elementos que ayuda a organizar el código en forma modular.
- **Estructura:** Los paquetes se reflejan como carpetas/directorios. Cada clase en Java pertenece a un paquete.
- **Convenciones:**
 - Usar minúsculas.
 - Usar el dominio web invertido para grandes proyectos: `com.miempresa.modulo`.
 - Organizar por arquitectura o funcionalidad.

Ejemplo básico:

```
package matematicas;
public class Operacion {
    public int sumar(int a, int b) { return a + b; }
    public int restar(int a, int b) { return a - b; }
}
```

```
package principales;
import matematicas.*;

public class Principal {
    public static void main(String[] args) {
        Operacion op = new Operacion();
        System.out.println(op.sumar(10, 90));
        System.out.println(op.restar(10, 90));
    }
}
```

- **Compilación:** Compilar primero las clases menos dependientes.

2. Librerías en Java

- **Definición:** Conjunto de clases ya compiladas (`.class`) agrupadas en un directorio o archivo `.jar`.
- **Ubicación:** Se especifica mediante la variable de entorno `CLASSPATH`.

Creación de una Librería:

1. Compilar clases necesarias:

```
javac Foo.java Bar.java
```

2. Empaquetar:

```
jar cvf Paq1.jar Foo.class Bar.class
```

Uso de JAR:

- Comprimen varias clases.
- Mantienen estructura de directorios.
- Se pueden usar como fuente de clases desde el `CLASSPATH`.

JAR ejecutable:

- Se necesita un **archivo** `MANIFEST.MF` que indique la clase principal y las librerías utilizadas.

3. Organización Avanzada con POO

Basado en el sistema de **Gestión de Publicaciones**, se extiende el uso de paquetes para organizar:

- `publicaciones`
 - `publicaciones.estandardIEEE`
 - `publicaciones.estandardAPA`

Se emplea el patrón **Abstract Factory** para instanciar objetos como `Articulo`, `Libro` y `Tesis` usando interfaces.

```
interface AbstractFactoryPublicacion {  
    Articulo crearArticulo();  
    Libro crearLibro();  
    Tesis crearTesis();  
}
```

ADICIONAL: PATRONES CREACIONALES

Claro. El patrón **Builder** es un patrón de diseño creacional que te permite construir objetos complejos paso a paso. La idea principal es **separar el proceso de construcción de un objeto de su representación final**.

Piensa en cómo pides una pizza 🍕. No le dices al cocinero todos los ingredientes en una sola frase larguísima y confusa. Lo haces paso a paso:

1. Quiero una masa delgada.
2. Ponle salsa de tomate.
3. Añade queso mozzarella.
4. Añade pepperoni.
5. ¡Listo! Ya tienes tu pizza.

El patrón Builder funciona exactamente igual en el código.

El Problema que Resuelve: El "Constructor Telescópico"

Imagina que tienes una clase **Coche** con muchas opciones: modelo, motor, color, número de puertas, GPS, asientos de cuero, etc.

Si usas constructores normales, te enfrentarías a un problema llamado "**constructor telescópico**":

Java

```
// Un constructor para un coche básico  
public Coche(String modelo, String motor) { ... }  
  
// Otro para un coche con color  
public Coche(String modelo, String motor, String color) { ... }
```

```
// Otro para un coche con color y puertas
public Coche(String modelo, String motor, String color, int puertas) { ... }

// Otro para un coche con GPS pero sin color específico (¡ups!)
// Tendrías que pasar null, lo cual es feo y propenso a errores.
public Coche(String modelo, String motor, null, int puertas, boolean tieneGps) { ... }
```

Crear un coche se vuelve un lío:

```
Coche miCoche = new Coche("Sedan", "V6", "Rojo", 4, true, true, false);
```

¿Qué significa el último `true`? ¿Y el `false`? Es muy fácil cometer errores.

La Solución del Patrón Builder

El patrón Builder resuelve esto dividiendo la creación en dos partes:

1. **El Builder:** Un objeto que sabe cómo construir el coche paso a paso.
2. **El Producto:** El objeto final (`Coche`) que se obtiene del Builder.

¿Cómo funciona?

1. Creas una clase `Builder` (a menudo como una clase estática anidada dentro de la clase `Coche`).
2. El `Builder` tiene métodos para cada una de las partes del coche (`conMotor()`, `pintarDeColor()`, `conGPS()`, etc.).
3. Estos métodos devuelven el propio `Builder` (`return this;`), lo que permite encadenar las llamadas de una manera muy legible (esto se conoce como *interfaz fluida*).
4. Finalmente, el `Builder` tiene un método `build()` que ensambla todas las piezas y te entrega el objeto `Coche` final y configurado.

Ejemplo en Código

Veamos cómo se vería la clase `Coche` usando el patrón Builder.

Java

```
// 1. El Producto Final (El objeto que queremos crear)
public class Coche {
    private final String modelo; // Requerido
    private final String motor; // Requerido
    private final String color; // Opcional
    private final int puertas; // Opcional
```

```

private final boolean tieneGPS; // Opcional

// El constructor es privado, solo el Builder puede llamarlo
private Coche(CocheBuilder builder) {
    this.modelo = builder.modelo;
    this.motor = builder.motor;
    this.color = builder.color;
    this.puertas = builder.puertas;
    this.tieneGPS = builder.tieneGPS;
}

// 2. El Builder (La clase que construye el producto)
public static class CocheBuilder {
    private final String modelo; // Requerido
    private final String motor; // Requerido
    private String color = "Blanco"; // Valor por defecto
    private int puertas = 4; // Valor por defecto
    private boolean tieneGPS = false; // Valor por defecto

    // El constructor del Builder solo pide los campos obligatorios
    public CocheBuilder(String modelo, String motor) {
        this.modelo = modelo;
        this.motor = motor;
    }

    // Métodos para las partes opcionales. Devuelven 'this' para encadenar.
    public CocheBuilder pintarDeColor(String color) {
        this.color = color;
        return this;
    }

    public CocheBuilder conPuertas(int puertas) {
        this.puertas = puertas;
        return this;
    }

    public CocheBuilder agregarGPS(boolean tieneGPS) {
        this.tieneGPS = tieneGPS;
        return this;
    }

    // 3. El método final que crea y devuelve el objeto Coche
    public Coche build() {
        return new Coche(this);
    }
}

```

Ahora, para crear un coche, el código es mucho más limpio y fácil de entender:

Java

```
// Usando el Builder
Coche miDeportivo = new Coche.CocheBuilder("Deportivo", "V8")
    .pintarDeColor("Rojo")
    .conPuertas(2)
    .agregarGPS(true)
    .build();

Coche miSedanBasico = new Coche.CocheBuilder("Sedan", "V4")
    .build(); // Usa los valores por defecto para lo opcional
```

Como puedes ver, es súper legible. Sabes exactamente qué estás configurando en cada paso.

Ventajas Clave

- Código más legible: La creación del objeto se lee casi como una frase en lenguaje natural.
- Flexibilidad: Permite crear diferentes representaciones de un mismo objeto usando el mismo proceso de construcción.
- Evita constructores complejos: Elimina la necesidad de tener múltiples constructores con largas listas de parámetros.
- Objetos inmutables: El patrón Builder es ideal para crear objetos inmutables (cuyos estados no pueden cambiar después de la creación), ya que todos los valores se establecen antes de llamar al constructor final.

1. Factory Method (Método de Fábrica)

- **¿Qué hace?** Define una interfaz (o una clase abstracta) para crear un objeto, pero deja que sean las subclases quienes decidan qué clase concreta instanciar. En lugar de llamar directamente al constructor de un objeto (`new Producto()`), llamas a un "método de fábrica".
- **Analogía:** Piensa en una empresa de logística. La empresa principal (la "fábrica") tiene un método `crearTransporte()`. Si necesitas enviar algo por tierra, usas la `LogisticaTerrestre` y su método `crearTransporte()` te devolverá un `Camion`. Si lo necesitas por mar, la `LogisticaMaritima` te devolverá un `Barco`. El cliente solo pide un transporte, y la subclase especializada decide cuál vehículo crear.

```
// El cliente no sabe si se creará un Camion o un Barco.
Transporte miTransporte = miLogistica.crearTransporte();
```

2. Abstract Factory (Fábrica Abstracta)

- **¿Qué hace?** Es como una "fábrica de fábricas". Proporciona una interfaz para crear **familias de objetos relacionados o dependientes** sin especificar sus clases concretas. Es un nivel de abstracción por encima de Factory Method.
- **Analogía:** Imagina que estás amueblando una casa. Necesitas una familia de muebles que combine. La `FabricaMueblesModernos` te dará una `SillaModerna`, una `MesaModerna` y un `SofaModerno`. Por otro lado, la `FabricaMueblesVictorianos` te dará una `SillaVictoriana`, una `MesaVictoriana` y un `SofaVictoriano`. Tú solo le pides a la fábrica abstracta "dame una silla", y ella se encarga de darte la del estilo correcto que pertenece a esa familia.Java

```
// Creas todos los muebles de un mismo estilo sin mezclarlos.  
FabricaMuebles fabrica = new FabricaMueblesModernos();  
Silla silla = fabrica.crearSilla(); // Será una SillaModerna  
Mesa mesa = fabrica.crearMesa(); // Será una MesaModerna
```

3. Singleton (Instancia Única)

- **¿Qué hace?** Garantiza que una clase solo tenga **una única instancia** y proporciona un punto de acceso global a ella. Es útil para objetos que deben ser únicos en todo el sistema, como un gestor de configuración, un sistema de logging o una conexión a base de datos.
- **Analogía:** El presidente de un país. Solo puede haber un presidente en funciones a la vez. No importa cuántas veces intentes "elegir" o "invocar" al presidente, siempre te referirás a la misma y única persona. El método `getInstance()` sería como decir "necesito hablar con el presidente", y siempre te conectarás con la misma instancia.Java

```
// No importa cuántas veces lo llames, siempre obtienes el mismo objeto.  
Configuracion config1 = Configuracion.getInstance();  
Configuracion config2 = Configuracion.getInstance(); // config1 y config2 son el mismo objeto.
```

4. Prototype (Prototipo)

- **¿Qué hace?** Permite crear nuevos objetos **clonando una instancia existente**, en lugar de crearlos desde cero. Es útil cuando el costo de crear un objeto es muy alto (por ejemplo, requiere una consulta a la base de datos o un cálculo complejo).

- **Analogía:** La clonación de ovejas. En lugar de crear una nueva oveja desde cero (un proceso biológico largo y complejo), tomas una oveja existente (el prototipo) y creas una copia genéticamente idéntica. En el código, en lugar de `new ObjetoComplejo()`, haces `objetoExistente.clonar()`.Java

```
// Creas un objeto base y luego clonas para hacer variaciones.
Circulo circuloBase = new Circulo(10, 20, "rojo");
Circulo circuloClonado = circuloBase.clonar();
circuloClonado.setColor("azul"); // Ahora tienes dos círculos, uno rojo y otro azul.
```

Tabla Resumen

Patrón	Propósito Principal	Analogía
Factory Method	Dejar que las subclases decidan qué objeto concreto crear.	Una empresa de logística que crea camiones o barcos.
Abstract Factory	Crear familias de objetos relacionados que deben usarse juntos.	Una fábrica de muebles que produce solo estilo moderno o solo victoriano.
Builder	Construir un objeto complejo paso a paso para evitar constructores largos.	Pedir una pizza personalizando cada ingrediente uno por uno.
Singleton	Asegurar que solo existe una instancia de una clase.	El presidente de un país.
Prototype	Crear nuevos objetos copiando uno existente.	Clonar una oveja en lugar de crear una nueva desde cero.

SESIÓN 4: LIBRERIAS EN C#

1. Espacios de nombres (Namespaces)

- Un **namespace** es un contenedor lógico para organizar clases, interfaces, estructuras, métodos, etc.
- Ayuda a **evitar conflictos de nombres** entre bibliotecas o partes de una aplicación.
- **Convención:** PascalCase, por ejemplo:

```
namespace MiEmpresa.Proyecto.Modulo
```

- No requiere una estructura de carpetas estricta como en Java, pero se recomienda mantener una jerarquía lógica.

Ejemplo:

```
// Operacion.cs
namespace Matematicas {
    public class Operacion {
        public int Sumar(int a, int b) => a + b;
        public int Restar(int a, int b) => a - b;
    }
}
```

```
// Principal.cs
using Matematicas;
namespace Principales {
    public class Principal {
        public static void Main(string[] args) {
            Operacion op = new Operacion();
            Console.WriteLine(op.Sumar(10, 80));
            Console.WriteLine(op.Restar(10, 80));
        }
    }
}
```

2. Librerías en C# (DLLs - Dynamic Link Libraries)

- Una **DLL** es una biblioteca de enlace dinámico que contiene código compilado reutilizable.
- Permite que **múltiples programas compartan código sin recompilarlo**.

Comandos de compilación:

- Crear ejecutable:

```
csc Foo.cs Bar.cs
```

- Crear archivo `.dll`:

```
csc /target:library Foo.cs Bar.cs
```

- Crear DLL con nombre específico:

```
csc /out:miLibreria.dll /target:library Foo.cs Bar.cs
```

Problemas comunes:

- Si una DLL compartida es eliminada o movida, los programas que dependen de ella **dejarán de funcionar**.
- Se recomienda manejar las dependencias de DLLs cuidadosamente.

3. Aplicación práctica con POO

Se utiliza el sistema de **Gestión de Publicaciones** para aplicar namespaces y DLLs organizadamente:

- Namespace raíz: `Publicaciones`
- Subnamespaces: `Publicaciones.EstandarIEEE`, `Publicaciones.EstandarAPA`
- Se organiza en clases como: `Articulo`, `Libro`, `Tesis`, `Autor`, etc.
- Se aplica el patrón **Abstract Factory** para generar publicaciones según el estándar (APA o IEEE), usando interfaces y clases concretas.

Interfaz de fábrica:

```
public interface IFactoryPublicacion {  
    Articulo CrearArticulo();  
    Libro CrearLibro();  
    Tesis CrearTesis();  
}
```

Clases concretas:

```
public class ConcreteFactoryAPA : IFactoryPublicacion {  
    public Articulo CrearArticulo() => new Articulo();  
    public Libro CrearLibro() => new Libro();  
    public Tesis CrearTesis() => new Tesis();  
}
```

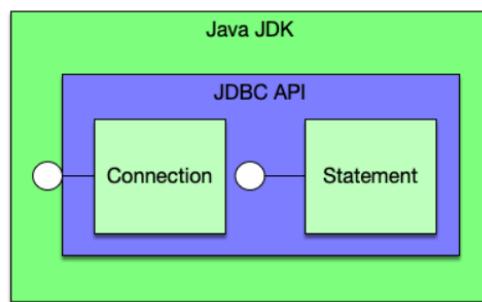
4. Recomendaciones para diseño y refactorización

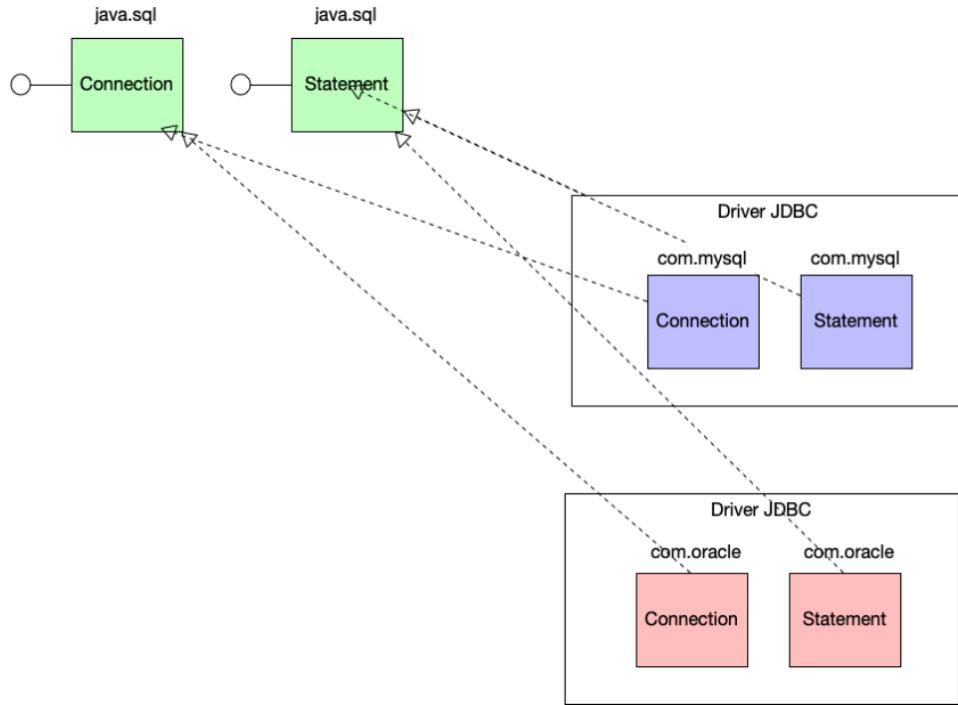
- Evitar **duplicación de código** en las clases `Articulo`, `Libro` y `Tesis` al implementar distintos estándares (IEEE, APA).
- Usar herencia o composición para abstraer lo común y reducir redundancia.

SESIÓN: BACKEND EN JAVA

JDBC Driver y MySQL

Muchos de los principiantes suelen apostar por esta segunda opción ya que se conectan a una base de datos y al ser esta MySQL pues no dudan . Lamentablemente es la incorrecta y eso es algo que cuesta entender ¿Porque tenemos que elegir `java.sql.Connection` y `java.sql.Statement`? Tanto `java.sql.Connection` como `java.sql.Statement` son interfaces del API de Java.





JDBC (Java Database Connectivity)

JDBC es una **API de Java** que permite conectar y operar con bases de datos **relacionales** desde aplicaciones Java.

Permite:

- Establecer una conexión.
- Ejecutar consultas SQL (SELECT, INSERT, UPDATE, DELETE).
- Recuperar resultados.
- Manipular datos desde Java.

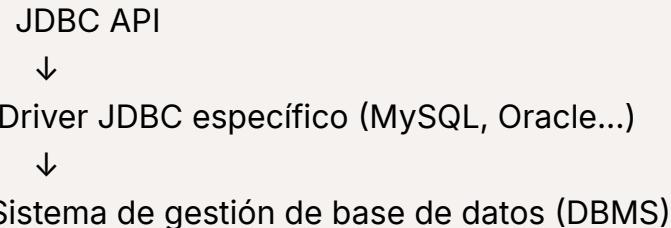
Puedes imaginar JDBC como un puente entre Java y la base de datos, sin importar si es MySQL, Oracle, SQL Server, etc.

Arquitectura y funcionamiento

La arquitectura general tiene estos componentes:

Tu programa Java





- La **JDBC API** es genérica (misma para todos).
- Cada base de datos requiere un **driver específico**.
- El **DriverManager** es quien gestiona los drivers registrados y entrega la conexión.

Connection

Un objeto **Connection** representa una conexión a una base de datos. La forma estándar de establecer una conexión con una base de datos es llamando al método **DriverManager.getConnection**. Este método toma como parámetro una cadena de caracteres que contiene una URL, usuario y password.

```
this.conexion = DriverManager.getConnection(this.getURL(), this.usuario, Cifrado.descifrarMD5(this.contraseña));
```

Conectarse a la base de datos en Java (algoritmo **getConnection**)

1. Obtener los datos de conexión

Normalmente se usan archivos de configuración (por ejemplo: `jdbc.properties`) para no escribir el usuario y contraseña en el código.

Datos que necesitas:

Clave	Significado
<code>driver</code>	Nombre completo del driver JDBC de MySQL que Java debe cargar.
<code>tipo_de_driver</code>	Protocolo JDBC + tipo de base de datos (aquí es MySQL). Se usa para formar la URL JDBC .

<code>base_de_datos</code>	Nombre de la base de datos a la que te vas a conectar.
<code>nombre_de_host</code>	Dirección del servidor remoto de base de datos (Ej: <code>localhost</code> , o sea, tu misma computadora o AWS RDS).
<code>puerto</code>	Puerto en el que escucha MySQL (por defecto es 3306).
<code>usuario</code>	Nombre de usuario con permisos sobre la base de datos.
<code>contrasenha</code>	Contraseña del usuario (encriptada).

```
# archivo jdbc.properties
driver=com.mysql.cj.jdbc.Driver
tipo_de_driver=jdbc:mysql
base_de_datos=soft_inv_test
nombre_de_host=localhost
puerto=3306
usuario=admin
contrasenha=GFvzT/oALwhgBEPIFFB2EA==
```

2. Cargar el driver JDBC

El **driver JDBC** es una librería específica que hace de puente entre Java y la base de datos.

```
Class.forName("com.mysql.cj.jdbc.Driver");
o
Class.forName(this.driver);
```

Esto le dice a Java: **"voy a usar un driver para conectarme a MySQL"**.

3. Generar la URL de conexión

La **URL JDBC** tiene la forma:

```
jdbc:mysql://host:puerto/nombreBaseDatos
```

Ejemplo:

```
private String getURL(){  
    String url = this.tipo_de_driver.concat("://");  
    url = url.concat(this.nombre_de_host);  
    url = url.concat(":");  
    url = url.concat(this.puerto);  
    url = url.concat("/");  
    url = url.concat(this.base_de_datos);  
    return url;  
}
```

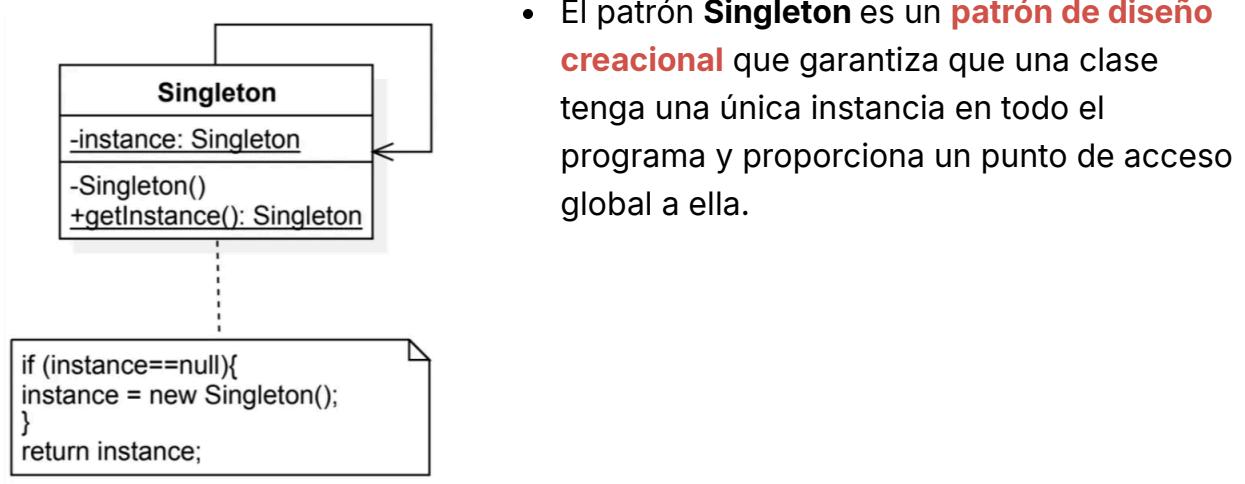
4. Obtener la conexión

Se hace con el método `DriverManager.getConnection`, pasando la URL, usuario y contraseña.

```
this.conexion = DriverManager.getConnection(this.getURL(), this.usuario, Cifrado.descifrarMD5(this.contraseña));
```

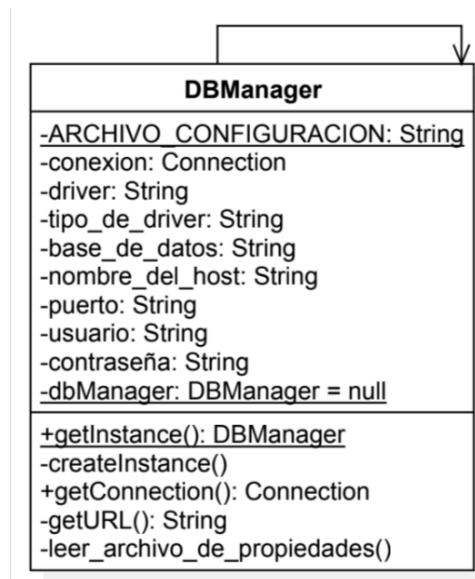
- `this.getURL()` → Genera la URL.
- `this.usuario` → Usuario de la BD.
- `Cifrado.descifrarMD5(this.contraseña)` → Toma la contraseña cifrada, la descifra y usa la original para conectar.

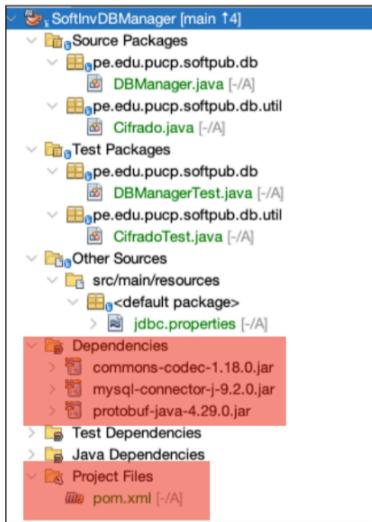
Patrón Singleton



- Características
 - Única instancia: Solo se crea una instancia de la clase.
 - Acceso global: Se accede a la instancia a través de un método estático.
 - Control sobre la instanciación: Se impide la creación de nuevas instancias desde fuera de la clase pues el constructor es privado.

Implementación del DBManager





Dependencias (pom.xml - Project Object Model)

```
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.18.0</version>
</dependency>
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>9.2.0</version>
</dependency>
```

32

<dependencies>

```
<!-- JDBC: Conector para MySQL -->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>9.2.0</version>
</dependency>

<!-- commons-codec: para codificación y decodificación (Base64) -->
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.18.0</version>
</dependency>

<!-- JUnit 5: para pruebas unitarias -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.10.2</version>
    <scope>test</scope>
```

```

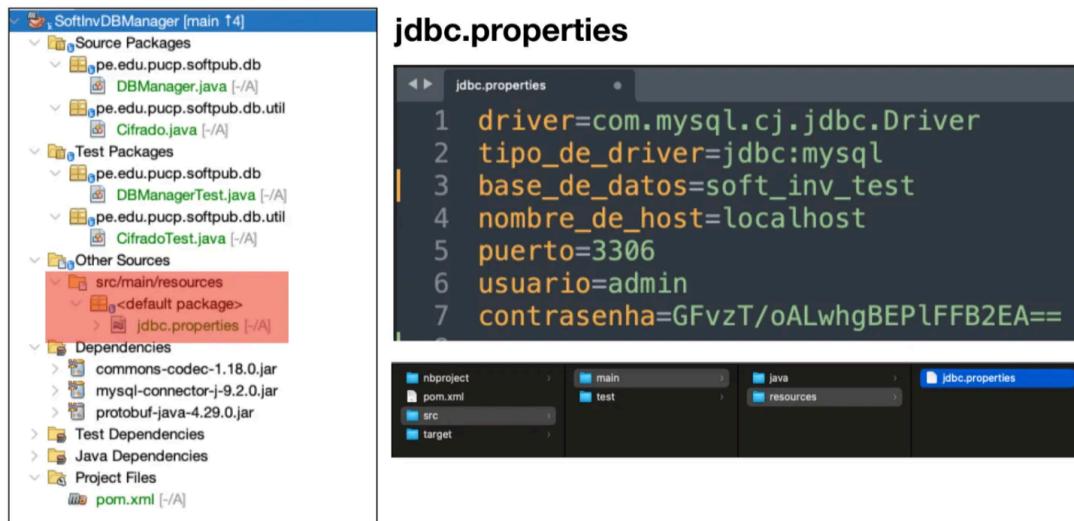
    </dependency>

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.10.2</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.10.2</version>
    <scope>test</scope>
</dependency>

</dependencies>

```



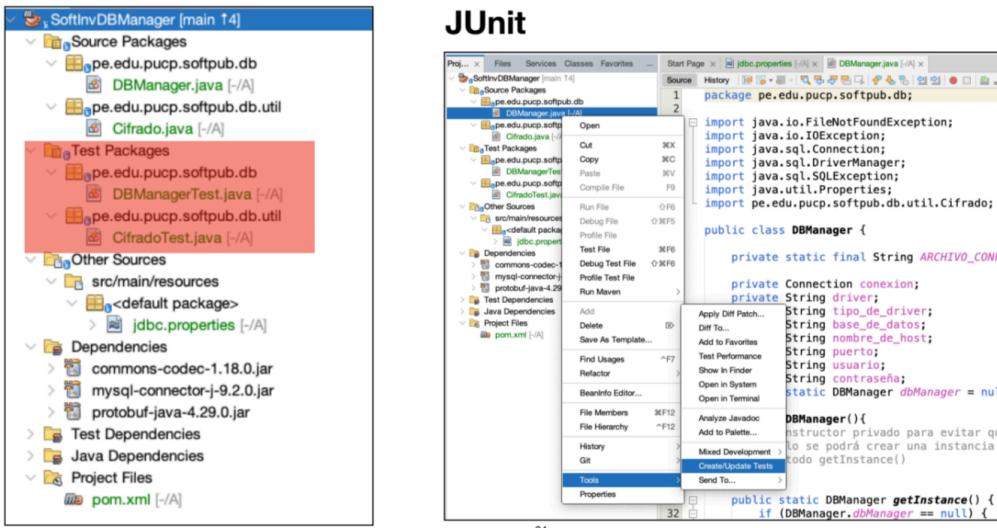
jdbc.properties

```

1 driver=com.mysql.cj.jdbc.Driver
2 tipo_de_driver=jdbc:mysql
3 base_de_datos=soft_inv_test
4 nombre_de_host=localhost
5 puerto=3306
6 usuario=admin
7 contrasenha=GFvzT/oALwhgBEPlFFB2EA==

```

33



Statement

Un objeto **Statement** se usa para enviar sentencias SQL a una base de datos. Una vez que se ha establecido una conexión con una base de datos particular, esa conexión puede ser usada para enviar sentencias SQL. Un objeto **Statement** se crea con el método `createStatement` de Connection como en el siguiente fragmento de código:

```
Statement stmt = con.createStatement();
```

- Se **crea a partir** de un objeto `Connection` (aquí, `con` es el objeto `Connection`).
- Después de conectarte a la base de datos, **usas `Statement` para ejecutar consultas.**

Tipos de ejecución

- `executeQuery(sql)` : para consultas tipo `SELECT`. Devuelve `ResultSet`
- `executeUpdate(sql)` : para operaciones `INSERT`, `UPDATE`, `DELETE`. Devuelve número de filas afectadas

SELECT

Cuando haces una consulta tipo `SELECT`, necesitas **leer los datos** devueltos fila por fila.

Para eso usamos `ResultSet`.

```
Statement sentencia = con.createStatement();
String query =
    "SELECT * FROM empleado";
ResultSet rs = sentencia.executeQuery(query);
while(rs.next()){
    String dni = rs.getString("dni");
    String nombres = rs.getString("nombres");
    String apellido_paterno = rs.getString("apellido_paterno");
    String apellido_materno = rs.getString("apellido_materno");
    System.out.println(dni + " " + nombres + " " + apellido_paterno + " " +
        apellido_materno);
}
con.close();
```



ResultSet

Un **ResultSet** contiene todos los registros (filas) que satisfacen las condiciones impuestas en una sentencia SQL y proporciona acceso a los datos en dichos registros a través de un conjunto de métodos **get** que permiten acceder a los diferentes campos o atributos (columnas) del registro actual. Un **ResultSet** mantiene un cursor que apunta al registro actual. El método **ResultSet.next()** se usa para moverse al siguiente registro del **ResultSet**, haciendo el siguiente registro el registro actual.

- Con `rs.next()` **avanzas** al siguiente registro.
- Con métodos como `getString(columna)` o `getInt(columna)`, **lees los valores** de las columnas.

INSERT

INSERTAR un empleado:

```
Statement stmt = con.createStatement();
String query = "INSERT INTO empleado (dni, nombres, apellido_paterno, apellido_materno) " +
    "VALUES ('12114689', 'Jorge', 'Mendoza', 'Lopez')";
int filas = stmt.executeUpdate(query);
con.close();
```

UPDATE

ACTUALIZAR un empleado:

```
Statement stmt = con.createStatement();
String query = "UPDATE empleado SET nombres = 'Karla Celeste' WHERE dni =
= '18276221'";
int filas = stmt.executeUpdate(query);
con.close();
```

DELETE

ELIMINAR un empleado:

```
Statement stmt = con.createStatement();
String query = "DELETE FROM empleado WHERE dni = '18276221'";
int filas = stmt.executeUpdate(query);
con.close();
```

SESIÓN: FRONTEND

SESIÓN:

soft test

soft-test

Resumen

Identificador de base de datos soft-test

Estado Disponible

CPU 3.45%

Clase db.t4g.micro

Role Instancia

Actividad actual 2 Conexiones

Motor MySQL Community

Región y AZ us-east-1b

Recomendaciones

Connectividad y seguridad Supervisión Registros y eventos Configuración Integraciones sin extracción, transformación y carga (ETL) Mantenimiento y copias de seguridad Migraciones de datos Etiquetas Recomendaciones

Connectividad y seguridad

Punto de enlace y puerto

Punto de enlace soft-test.cxcq0ck6m00u.us-east-1.rds.amazonaws.com

Puerto 3306

Redes

Zona de disponibilidad us-east-1b

VPC vpc-07deef3f304dd23eb2

Grupo de subredes default-vpc-07deef3f304dd23eb2

Subredes

subnet-09972a4b533be52022
subnet-09d4d80404d118d24b0
subnet-0d900952d5c4f948
subnet-05a773de6ad94b1839
subnet-0af14e517e8525f08

Tipo de red IPv4

Seguridad

Grupos de seguridad de la VPC default (sg-07b6f001742561ed5)

Activo

Accesible públicamente Sí

Entidad de certificación rds-cdb-rsa2048-g1

Información

Fecha de la entidad de certificación May 25, 2061, 18:34 (UTC-05:00)

Fecha de expiración del certificado de instancia de base de datos September 30, 2026, 17:22 (UTC-05:00)

Recursos de computación conectados (0) Información

Aquí aparecen las conexiones a los recursos de computación creados automáticamente por RDS. No aparecen las conexiones a recursos de computación creados manualmente.

soft-test.cxcq0ck6m00u.us-east-1.rds.amazonaws.com

db nombre: transitsoft / transitsoft-ssql

CONTRA: pacoflaco123

user: admin

puerto: 3306