PROGRAMACIÓN – B

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

DESARROLLO DE APLICACIONES WEB

iLERNA.

Maquetado e impreso por Ilerna Online S. L. U.

Impreso en España - Printed in Spain

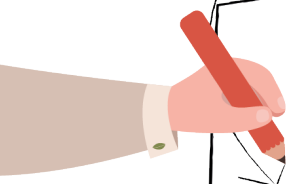
3.ª edición: marzo 2025

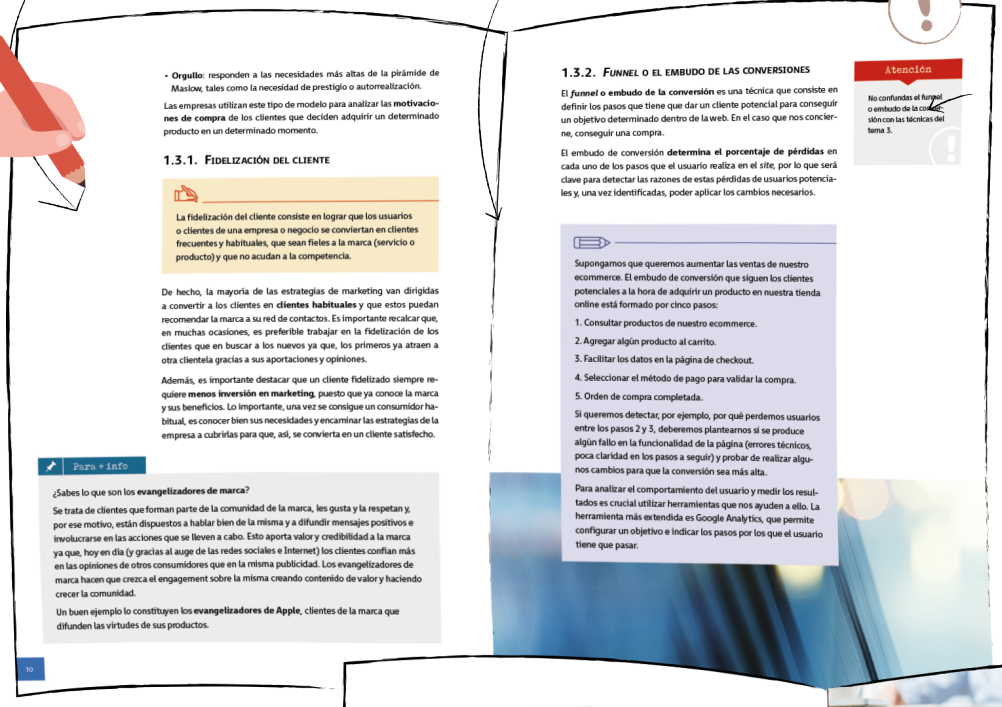
Guía de uso

¡Aprovecha al máximo este libro con todos sus recursos!

Concepto: definiciones fundamentales que encontrarás en cada tema.

Ejemplo: una muestra real sobre la teoría que estás aprendiendo.





- **Orgullo**: responden a las necesidades más altas de la pirámide de Maslow, tales como la necesidad de prestigio o autorrealización.

Las empresas utilizan este tipo de modelo para analizar las **motivaciones de compra** de los clientes que deciden adquirir un determinado producto en un determinado momento.

1.3.1. Fidelización del cliente

La fidelización del cliente consiste en lograr que los usuarios o clientes de una empresa o negocio se conviertan en clientes frecuentes y habituales, que sean fieles a la marca (servicio o producto) y que no acudan a la competencia.

De hecho, la mayoría de las estrategias de marketing van dirigidas a convertir a los clientes en **clientes habituales** y que estos puedan recomendar la marca a su red de contactos. Es importante recalcar que, en muchas ocasiones, es preferible trabajar en la fidelización de los clientes que en buscar a los nuevos ya que, los primeros ya atraen a otra clientela gracias a sus aportaciones y opiniones.

Además, es importante destacar que un cliente fidelizado siempre requiere **menos inversión en marketing**, puesto que ya conoce la marca y sus beneficios. Lo importante, una vez se consigue un consumidor habitual, es conocer bien sus necesidades y encaminar las estrategias de la empresa a cubrirlas para que, así, se convierta en un cliente satisfecho.

Para + info

¿Sabes lo que son los **evangelizadores de marca**?

Se trata de clientes que forman parte de la comunidad de la marca, les gusta y la respetan y, por ese motivo, están dispuestos a hablar bien de la misma y a difundir mensajes positivos e involucrarse en las acciones que se lleven a cabo. Esto aporta valor y credibilidad a la marca ya que, hoy en día (y gracias al auge de las redes sociales e Internet) los clientes confían más en las opiniones de otros consumidores que en la misma publicidad. Los evangelizadores de marca hacen que crezca el engagement sobre la misma creando contenido de valor y haciendo crecer la comunidad.

Un buen ejemplo lo constituyen los **evangelizadores de Apple**, clientes de la marca que difunden las virtudes de sus productos.

10

1.3.2. *Funnel* o el embudo de las conversiones

Atención

No confundas el funnel o embudo de la conversión con las técnicas del tema 3.

El ***funnel* o embudo de la conversión** es una técnica que consiste en definir los pasos que tiene que dar un cliente potencial para conseguir un objetivo determinado dentro de la web. En el caso que nos concierne, conseguir una compra.

El embudo de conversión **determina el porcentaje de pérdidas** en cada uno de los pasos que el usuario realiza en el site, por lo que será clave para detectar las razones de estas pérdidas de usuarios potenciales y, una vez identificadas, poder aplicar los cambios necesarios.

Supongamos que queremos aumentar las ventas de nuestro ecommerce. El embudo de conversión que siguen los clientes potenciales a la hora de adquirir un producto en nuestra tienda online está formado por cinco pasos:

1. Consultar productos de nuestro ecommerce.
2. Agregar algún producto al carrito.
3. Facilitar los datos en la página de checkout.
4. Seleccionar el método de pago para validar la compra.
5. Orden de compra completada.

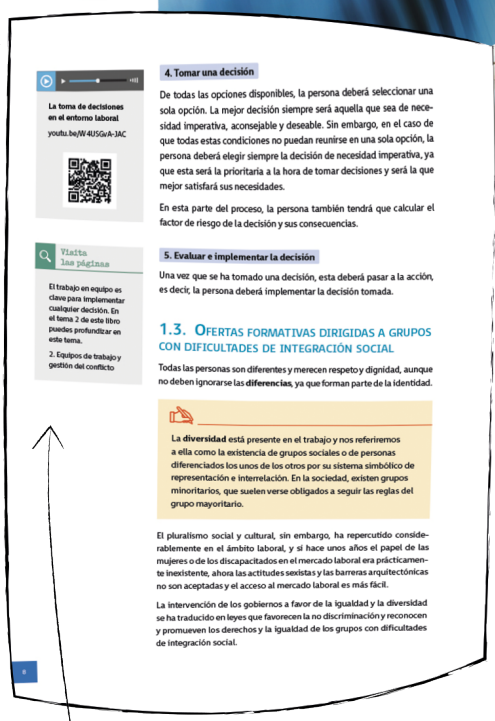
Si queremos detectar, por ejemplo, por qué perdemos usuarios entre los pasos 2 y 3, deberemos plantearnos si se produce algún fallo en la funcionalidad de la página (errores técnicos, poca claridad en los pasos a seguir) y probar de realizar algunos cambios para que la conversión sea más alta.

Para analizar el comportamiento del usuario y medir los resultados es crucial utilizar herramientas que nos ayuden a ello. La herramienta más extendida es Google Analytics, que permite configurar un objetivo e indicar los pasos por los que el usuario tiene que pasar.

Atención: ¡no olvides nunca esta información!

Ponte a prueba: a lo largo del libro, encontrarás preguntas con las que podrás evaluar si estás comprendiendo bien los contenidos. Al final del libro se encuentran las soluciones.

Para + info: contenidos adicionales con los que podrás ampliar tu conocimiento.



La toma de decisiones en el entorno laboral
youtu.be/W4U5GvA-JAC

Visita las páginas

El trabajo en equipo es clave para implementar cualquier decisión. En el tema 2 de este libro puedes profundizar en este tema.

2. Equipos de trabajo y gestión del conflicto

4. Tomar una decisión

De todas las opciones disponibles, la persona deberá seleccionar una sola opción. La mejor decisión siempre será aquella que sea de necesidad imperativa, aconsejable y deseable. Sin embargo, en el caso de que todas estas condiciones no puedan reunirse en una sola opción, la persona deberá elegir siempre la decisión de necesidad imperativa, ya que esta será la prioritaria a la hora de tomar decisiones y será la que mejor satisfará sus necesidades.

En esta parte del proceso, la persona también tendrá que calcular el factor de riesgo de la decisión y sus consecuencias.

5. Evaluar e implementar la decisión

Una vez que se ha tomado una decisión, esta deberá pasar a la acción, es decir, la persona deberá implementar la decisión tomada.

1.3. Ofertas formativas dirigidas a grupos con dificultades de integración social

Todas las personas son diferentes y merecen respeto y dignidad, aunque no deben ignorarse las **diferencias**, ya que forman parte de la identidad.

La **diversidad** está presente en el trabajo y nos referiremos a ella como la existencia de grupos sociales o de personas diferenciados los unos de los otros por su sistema simbólico de representación e interrelación. En la sociedad, existen grupos minoritarios, que suelen verse obligados a seguir las reglas del grupo mayoritario.

El pluralismo social y cultural, sin embargo, ha repercutido considerablemente en el ámbito laboral, y si hace unos años el papel de las mujeres o de los discapacitados en el mercado laboral era prácticamente inexistente, ahora las actitudes sexistas y las barreras arquitectónicas no son aceptadas y el acceso al mercado laboral es más fácil.

La intervención de los gobiernos a favor de la igualdad y la diversidad se ha traducido en leyes que favorecen la no discriminación y reconocen y promueven los derechos y la igualdad de los grupos con dificultades de integración social.

8

Las Administraciones públicas, en especial las locales, trabajan para facilitar la integración social y la inclusión en el mercado de trabajo de los individuos o grupos desfavorecidos y/o en riesgo de exclusión social. Así, en el ámbito de sus competencias, adaptarán las ofertas formativas a las necesidades específicas de los jóvenes con fracaso escolar, de los discapacitados, de las minorías étnicas, de los parados de larga duración y, en general, de todas aquellas personas con riesgo de exclusión social.

Estas ofertas formativas tratan de ayudar a la adquisición de diferentes capacidades en un proceso formativo que dura toda la vida y, además, tratan de incluir módulos asociados al Catálogo Nacional de Cualificaciones Profesionales, pudiendo incorporar módulos apropiados para la adaptación a las necesidades específicas del colectivo beneficiario.

A nivel normativo, es conveniente destacar el Real Decreto Legislativo 1/2013, de 29 de noviembre, por el que se aprueba el Texto Refundido de la Ley General de derechos de las personas con discapacidad y de su inclusión social. Esta ley garantiza el derecho a la igualdad de oportunidades y de trato, así como el ejercicio real y efectivo de derechos por parte de las personas con discapacidad en igualdad de condiciones respecto del resto de ciudadanos y ciudadanas.

La lucha contra la discriminación tiene su reconocimiento a nivel constitucional en los artículos 9.2, 10, 14 y 49 de la Constitución Española. A nivel internacional también se combate contra cualquier tipo de discriminación con instrumentos legales, por ejemplo, con la Convención Internacional sobre los Derechos de las Personas con Discapacidad.

Ponte a prueba

¿Es verdadero o falso que las Administraciones públicas, en especial las locales, trabajan para facilitar la integración social y la inclusión en el mercado de trabajo de los individuos o grupos desfavorecidos y/o en riesgo de exclusión social?

a) Verdadero
b) Falso

Cuando hablamos de la existencia de grupos sociales o de personas diferenciados los unos de los otros por su sistema simbólico de representación e interrelación, ¿de qué estamos hablando?

a) Las barreras sociales
b) La diversidad
c) Exclusivamente de los grupos minoritarios

9

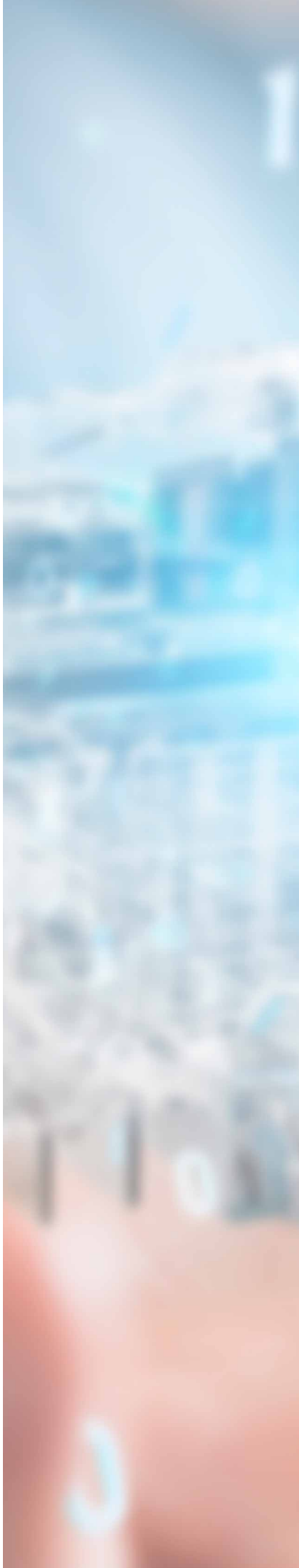
Vídeos: te ayudarán a repasar y comprender mejor los contenidos. Puedes acceder a ellos mediante el enlace o el código QR.

Visita las páginas: en otro apartado del libro encontrarás información relacionada con este punto o tema.

Legislación: referencias o fragmentos de leyes y normativas.

ÍNDICE

Programación - B



6 PROGRAMACIÓN ORIENTADA A OBJETOS II

La programación orientada a objetos (POO) es un paradigma fundamental en el desarrollo de software moderno, que permite organizar el código en torno a objetos que encapsulan tanto el estado como el comportamiento. Después de haber aprendido los conceptos básicos de la POO, este segundo módulo se centrará en tres de los conceptos más avanzados y poderosos de este paradigma: herencia, polimorfismo y *casting*.

Estos principios permiten crear sistemas más complejos y escalables, mejorando la reutilización del código, la flexibilidad y la capacidad de extensión de las aplicaciones.

Para + info

Herencia, polimorfismo y *casting* son conceptos avanzados de la POO que permiten construir aplicaciones más flexibles, escalables y reutilizables.

6.1. Métodos y variables estáticas

En Java, los **métodos** y **variables estáticas** pertenecen a la **clase**, no a una instancia específica de la clase. Esto significa que se pueden acceder y utilizar sin crear un objeto de esa clase. Una variable estática se inicializa una sola vez y su valor se comparte entre todas las instancias de la clase. De manera similar, los métodos estáticos pueden ser llamados directamente desde la clase, sin necesidad de crear un objeto.

Por ejemplo, si una variable es declarada como estática, todas las instancias de la clase comparten la misma variable, en lugar de tener una copia separada en cada objeto. Los métodos estáticos son ideales para operaciones que no dependen de los atributos específicos de un objeto, como las funciones utilitarias o los métodos que operan sobre datos generales.

Para + info

Los **métodos y variables estáticas** pertenecen a la clase, no a una instancia específica, y se pueden acceder sin necesidad de crear un objeto.

Cómo se utilizan en Java

En Java, para declarar una variable o método como estático, se utiliza la palabra clave static. Los miembros estáticos se acceden directamente mediante el nombre de la clase en lugar de a través de un objeto de esa clase.

Declaración y acceso a variables y métodos estáticos:

```
public static int contador;
```

Declaración de un método estático:

```
public static void metodoEstatico() {
    // Código
}
```

Acceso a una variable estática:

```
Clase.contador;
```

Llamada a un método estático:

```
Clase.metodoEstatico();
```


Ejemplos usando métodos y variables estáticas:

Ejemplo 1: contador de objetos con una variable estática

Enunciado: implementa una clase que use una variable estática para contar cuántos objetos se han creado de esa clase.

Código:

```
public class ContadorObjetos {
    // Variable estática para contar el número de objetos
    public static int contador = 0;

    // Constructor
    public ContadorObjetos() {
        contador++;
    }

    // Método estático para obtener el número de objetos creados
    public static int obtenerContador() {
        return contador;
    }
```

```
    public static void main(String[] args) {
        ContadorObjetos obj1 = new ContadorObjetos();
        ContadorObjetos obj2 = new ContadorObjetos();
        ContadorObjetos obj3 = new ContadorObjetos();

        // Imprimir el número total de objetos creados
        System.out.println("Número total de objetos creados: " +
                           ContadorObjetos.obtenerContador());
    }
}
```

Explicación: en este ejemplo, se utiliza una variable estática contador para llevar la cuenta de cuántos objetos de la clase ContadorObjetos se han creado. Cada vez que se crea un nuevo objeto, el contador se incrementa. El método estático obtenerContador() permite acceder al número total de objetos creados sin la necesidad de crear otro objeto.

Ejemplo 2: uso de un método estático en una clase utilitaria

Enunciado: crea una clase utilitaria que use un método estático para realizar operaciones matemáticas simples, como sumar dos números.

Código:

```
public class UtilidadMatematica {
    // Método estático para sumar dos números
    public static int sumar(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        int resultado = UtilidadMatematica.sumar(10, 20);
        System.out.println("La suma de 10 y 20 es: " + resultado);
    }
}
```

Explicación: este ejemplo muestra una clase UtilidadMatematica que contiene un método estático sumar() para realizar una suma de dos números enteros. El método estático se llama directamente a través del nombre de la clase, sin la necesidad de crear un objeto.

Ejemplo 3: uso de una constante estática en una clase

Enunciado: define una constante estática que represente el valor de PI y úsala para calcular el área de un círculo.

```
    // Método estático para calcular el área de un círculo
    public static double calcularArea(double radio) {
        return PI * radio * radio;
    }

    public static void main(String[] args) {
        double radio = 5.0;
        double area = Circulo.calcularArea(radio);
        System.out.println("El área del círculo con radio " + radio + "
                           es: " + area);
    }
}
```

Explicación: en este ejemplo, la clase Circulo define una constante estática PI, que representa el valor matemático de Pi. Luego, se utiliza en el método estático calcularArea() para calcular el área de un círculo. Las constantes estáticas se declaran con final y no pueden cambiar su valor.

Los métodos y variables estáticas en Java son fundamentales cuando es necesario compartir información o funcionalidad entre todas las instancias de una clase o cuando no es necesario crear una instancia para acceder a ciertas operaciones o datos. Son especialmente útiles en clases utilitarias y en la creación de constantes globales.

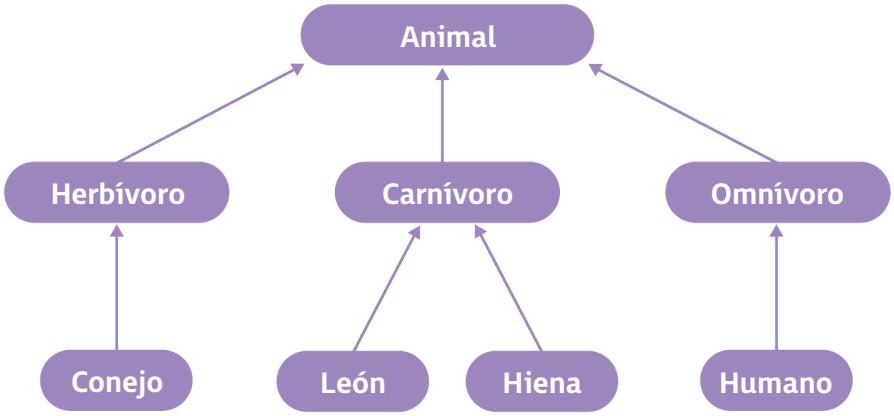
6.2. Herencia

Para + info

La **herencia** permite que una clase derive propiedades y comportamientos de otra clase, promoviendo la reutilización y especialización del código.

La **herencia** es uno de los pilares fundamentales de la Programación Orientada a Objetos (POO). Es un mecanismo que permite que una clase (denominada **subclase** o **clase derivada**) herede atributos y métodos de otra clase (llamada **superclase** o **clase base**). Esto permite la reutilización de código y facilita la extensión de funcionalidades sin necesidad de reescribir código ya existente.

La herencia permite modelar relaciones jerárquicas entre clases, donde las subclases pueden especializarse o modificar el comportamiento de la clase base (a través de la **sobreescritura** de métodos). Además, pueden agregar nuevas propiedades y métodos específicos para su funcionalidad.



Cómo se utiliza en Java

En Java, la herencia se implementa utilizando la palabra clave extends. Cuando una clase hereda de otra, la subclase hereda todos los atributos y métodos públicos y protegidos de la superclase. La subclase también puede sobrescribir métodos de la superclase para personalizar su comportamiento.

Sintaxis básica de la herencia en Java

```
class SuperClase {
    // Métodos y atributos de la superclase
}

class SubClase extends SuperClase {
    // Métodos y atributos adicionales o modificados
de la subclase
}
```

- **Clase base (superclase)**: la clase de la que se heredan los atributos y métodos.
- **Clase derivada (subclase)**: la clase que extiende la clase base, heredando sus propiedades.

Tres ejemplos usando herencia en Java:

Ejemplo 1: herencia básica

Enunciado: crea una clase base Animal con un método hacerSonido(), y luego implementa una clase Perro que herede de Animal y sobreescriba el método para dar un comportamiento específico.

Código:

```
// Clase base
class Animal {
    public void hacerSonido() {
        System.out.println("El animal hace un sonido.");
    }
}

// Clase derivada
class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El perro ladra.");
    }
}

public class EjemploHerencia1 {
    public static void main(String[] args) {
        // Crear instancia de la clase derivada
        Perro perro = new Perro();
        perro.hacerSonido(); // Imprime: "El perro ladra."

        // También podemos usar la clase base para referenciar la subclase
        Animal animal = new Perro();
        animal.hacerSonido(); // Imprime: "El perro ladra."
    }
}
```

Explicación: en este ejemplo, la clase Perro hereda de la clase Animal. Aunque Animal tiene un método hacerSonido(), la subclase Perro sobrescribe este método para proporcionar un comportamiento específico. Además, mostramos cómo se puede utilizar una referencia de la clase base (Animal) para invocar el comportamiento de la subclase.

Ejemplo 2: uso de la palabra clave super

Enunciado: usa la palabra clave super para acceder al método de la clase base dentro de la clase derivada.

Código:

```
// Clase base
class Vehiculo {
    public void describir() {
        System.out.println("Este es un vehículo.");
    }
}

// Clase derivada
class Coche extends Vehiculo {
    @Override
    public void describir() {
        // Llamar al método de la clase base
        super.describir();
        System.out.println("Este es un coche.");
    }
}

public class EjemploHerencia2 {
    public static void main(String[] args) {
        Coche coche = new Coche();
        coche.describir();
        // Imprime:
        // "Este es un vehículo."
        // "Este es un coche."
    }
}
```

Explicación: en este ejemplo, la clase Coche hereda de Vehiculo. Utilizamos la palabra clave super para llamar al método describir() de la clase base Vehiculo antes de agregar un comportamiento adicional en la subclase. Esto permite extender la funcionalidad del método heredado.

Ejemplo 3: herencia multinivel

Enunciado: implementa una herencia multinivel donde Animal es la clase base, Mamifero es una clase intermedia, y Gato es la subclase final.

Código:

```
// Clase base
class Animal {
    public void moverse() {
        System.out.println("El animal se está moviendo.");
    }
}

// Clase derivada (intermedia)
class Mamifero extends Animal {
    public void respirar() {
        System.out.println("El mamífero está respirando.");
    }
}

// Clase derivada (final)
class Gato extends Mamifero {
    @Override
    public void moverse() {
        System.out.println("El gato está caminando sigilosamente.");
    }
}

public class EjemploHerencia3 {
    public static void main(String[] args) {
        Gato gato = new Gato();
        gato.moverse();    // Imprime: "El gato está caminando sigilosa-
mente."
        gato.respirar();   // Imprime: "El mamífero está respirando."
    }
}
```


Explicación: este ejemplo muestra una herencia multinivel en la que la clase Gato hereda indirectamente de Animal a través de Mamifero. Aquí, Gato sobrescribe el método moverse() de la clase base Animal para proporcionar una implementación más específica, pero también tiene acceso al método respirar() de Mamifero.

La **herencia** es una característica clave en la programación orientada a objetos que permite reutilizar y extender código existente. Facilita la creación de jerarquías de clases que comparten atributos y comportamientos comunes, mientras que las subclases pueden especializarse al sobrescribir métodos o añadir nuevas funcionalidades.

6.3. Interfaces

En la programación orientada a objetos, una **interfaz** es una estructura que define un conjunto de métodos que una clase debe implementar, pero no proporciona la implementación de esos métodos. Es decir, una interfaz es un contrato que las clases deben cumplir, asegurando que aquellas que implementan la interfaz proporcionen su propia implementación de los métodos definidos.

A diferencia de la herencia tradicional, en la que una clase puede heredar de una sola clase padre, una clase en Java puede implementar múltiples interfaces, lo que permite un mayor grado de flexibilidad y promueve la reutilización del código. Las interfaces en Java no pueden tener atributos ni métodos concretos (hasta Java 8, donde se introdujeron métodos con implementación por defecto), y todos los métodos de una interfaz son públicos y abstractos por defecto.

Cómo se utilizan en Java

En Java, las interfaces se declaran con la palabra clave interface, y las clases que implementan una interfaz deben usar la palabra clave implements. Si una clase implementa una interfaz, debe proporcionar la implementación de todos los métodos abstractos definidos en la interfaz.

Sintaxis de una interfaz

```
interface NombreInterfaz {
    void metodo1();
    void metodo2();
}
```

Implementación de una interfaz en una clase

```
class MiClase implements NombreInterfaz {
    @Override
    public void metodo1() {
        // Implementación del método
    }

    @Override
    public void metodo2() {
        // Implementación del método
    }
}
```

Para + info

Una **interfaz** define un contrato que las clases deben cumplir, asegurando que implementen los métodos necesarios sin dictar cómo deben hacerlo.

Ejemplos usando interfaces en Java:

Ejemplo 1: definición e implementación de una interfaz

Enunciado: crea una interfaz Vehiculo que defina dos métodos: acelerar() y frenar(). Luego, implementa la interfaz en una clase Coche.

Código:

```
// Definición de la interfaz
interface Vehiculo {
    void acelerar();
    void frenar();
}

// Implementación de la interfaz en la clase Coche
class Coche implements Vehiculo {
    @Override
    public void acelerar() {
        System.out.println("El coche está acelerando.");
    }

    @Override
    public void frenar() {
        System.out.println("El coche está frenando.");
    }
}

public class EjemploInterfaz1 {
    public static void main(String[] args) {
        Coche miCoche = new Coche();
        miCoche.acelerar();  // Imprime: "El coche está acelerando."
        miCoche.frenar();    // Imprime: "El coche está frenando."
    }
}
```


Explicación: en este ejemplo, la interfaz Vehiculo define dos métodos abstractos, acelerar() y frenar(). La clase Coche implementa esta interfaz y proporciona la implementación de ambos métodos. Esto asegura que cualquier clase que implemente la interfaz Vehiculo debe proporcionar implementaciones para estos métodos.

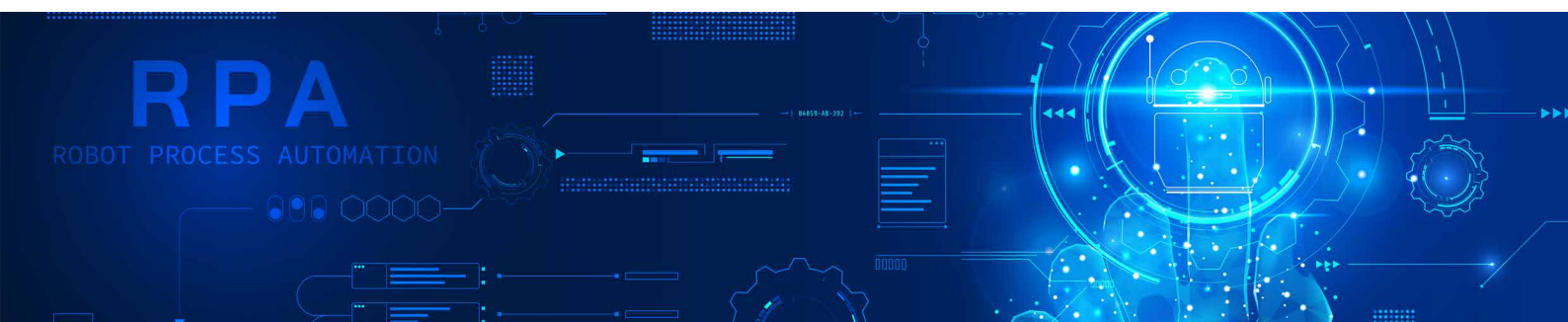
Ejemplo 2: una clase que implementa múltiples interfaces

Enunciado: crea dos interfaces: Navegable y Volador, cada una con un método, y luego implementa ambas en la clase Hidroavion.

Código:

```
// Definición de la interfaz Navegable
interface Navegable {
    void navegar();
}

// Definición de la interfaz Volador
interface Volador {
    void volar();
}

// Implementación de ambas interfaces en la clase Hidroavion
class Hidroavion implements Navegable, Volador {
    @Override
    public void navegar() {
        System.out.println("El hidroavión está navegando en el agua.");
    }

    @Override
    public void volar() {
        System.out.println("El hidroavión está volando en el aire.");
    }
}

public class EjemploInterfaz2 {
    public static void main(String[] args) {
        Hidroavion miHidroavion = new Hidroavion();
        miHidroavion.navegar();  // Imprime: "El hidroavión está navegando
en el agua."
        miHidroavion.volar();    // Imprime: "El hidroavión está volando
en el aire."
    }
}
```

Explicación: este ejemplo muestra cómo una clase (Hidroavion) puede implementar múltiples interfaces (Navegable y Volador). Cada interfaz define un comportamiento específico, y la clase Hidroavion implementa ambos comportamientos, lo que le permite tanto navegar como volar.



Ejemplo 3: uso de una interfaz para describir comportamientos en común

Enunciado: crea una interfaz Imprimible para objetos que puedan ser impresos, luego implementa la interfaz en las clases Documento e Imagen.

Código:

```
// Definición de la interfaz Imprimible
interface Imprimible {
    void imprimir();
}

// Clase Documento que implementa Imprimible
class Documento implements Imprimible {
    private String contenido;

    public Documento(String contenido) {
        this.contenido = contenido;
    }

    @Override
    public void imprimir() {
        System.out.println("Imprimiendo documento: " + contenido);
    }
}

// Clase Imagen que implementa Imprimible
class Imagen implements Imprimible {
    private String archivo;

    public Imagen(String archivo) {
        this.archivo = archivo;
    }

    @Override
    public void imprimir() {
        System.out.println("Imprimiendo imagen desde el archivo: " + archivo);
    }
}

public class EjemploInterfaz3 {
    public static void main(String[] args) {
        Imprimible doc = new Documento("Reporte Anual");
        Imprimible img = new Imagen("foto.jpg");

        doc.imprimir();  // Imprime: "Imprimiendo documento: Reporte Anual"
        img.imprimir();  // Imprime: "Imprimiendo imagen desde el archivo:
foto.jpg"
    }
}
```

Explicación: en este ejemplo, se crea una interfaz Imprimible que define el método imprimir(). Las clases Documento e Imagen implementan esta interfaz de diferentes maneras, pero ambas proporcionan una implementación concreta del método imprimir(). Esto demuestra cómo la interfaz puede ser utilizada para garantizar que diferentes clases tengan un comportamiento en común, pero cada una lo implemente a su manera.

Las **interfaces** en Java permiten definir contratos que las clases deben cumplir sin especificar cómo debe realizarse la implementación. Esto promueve la reutilización del código, facilita la modularidad y permite que las clases implementen múltiples comportamientos sin la limitación de la herencia única. Las interfaces son una herramienta poderosa en la programación orientada a objetos, especialmente cuando se diseñan sistemas extensibles y flexibles.

6.4. Métodos y variables abstractas

Un **método abstracto** es un método que no tiene implementación en la clase en la que está definido; simplemente se declara y su implementación real se deja a las clases derivadas o subclases. Los métodos abstractos sólo pueden existir dentro de **clases abstractas** o **interfaces**. Una clase que contiene uno o más métodos abstractos debe ser declarada como abstracta y no puede ser instanciada directamente. El propósito de los métodos abstractos es proporcionar una base común para un grupo de clases relacionadas que comparten un comportamiento, pero que pueden implementar ese comportamiento de manera diferente.

Las **variables abstractas** como tal no existen en Java. Sin embargo, en una clase abstracta se pueden declarar **constantes** o **variables** que se utilizarán en las subclases, pero en Java las variables no pueden ser abstractas; solo los métodos pueden serlo.

Los **métodos abstractos** definen el comportamiento que las subclases deben implementar, mientras que las clases abstractas proporcionan una base común para otras clases.

Cómo se utilizan en Java

En Java, los métodos abstractos se declaran dentro de clases abstractas o interfaces. Las clases que heredan de una clase abstracta deben proporcionar la implementación de todos los métodos abstractos, a menos que también sean declaradas como abstractas. Las clases abstractas permiten a los desarrolladores crear una estructura que otras clases deben seguir.

Declaración de un método abstracto

```
abstract class Animal {
    public abstract void hacerSonido(); // Método
abstracto
}
```

Clase que implementa un método abstracto

```
class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El perro ladra.");
    }
}
```

Ejemplos usando métodos abstractos en Java:

Ejemplo 1: clase abstracta con método abstracto

Enunciado: crea una clase abstracta Animal con un método abstracto hacerSonido(). Luego, implementa una subclase Perro que proporcione la implementación de este método.

Código:

```
// Clase abstracta
abstract class Animal {
    // Método abstracto
    public abstract void hacerSonido();
}

// Subclase que implementa el método abstracto
class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El perro ladra.");
    }
}

public class EjemploAbstracto1 {
    public static void main(String[] args) {
        // No se puede instanciar la clase abstracta
        // Animal animal = new Animal(); // Esto dará error

        // Se puede instanciar una subclase concreta
        Perro perro = new Perro();
        perro.hacerSonido(); // Imprime: "El perro ladra."
    }
}
```

Explicación: en este ejemplo, Animal es una clase abstracta con un método abstracto hacerSonido(). La clase Perro extiende Animal y proporciona una implementación para el método abstracto. Este ejemplo muestra cómo las subclases concretas deben implementar los métodos abstractos.

Ejemplo 2: clase abstracta con métodos comunes y abstractos

Enunciado: crea una clase abstracta Vehiculo con un método abstracto acelerar() y un método concreto detener(). Luego, implementa una subclase Coche que proporcione la implementación del método abstracto.

Código:

```
// Clase abstracta
abstract class Vehiculo {
    // Método abstracto
    public abstract void acelerar();

    // Método concreto
    public void detener() {
        System.out.println("El vehículo se detiene.");
    }
}

// Subclase que implementa el método abstracto
class Coche extends Vehiculo {
    @Override
    public void acelerar() {
        System.out.println("El coche acelera.");
    }
}

public class EjemploAbstracto2 {
    public static void main(String[] args) {
        // Instanciar la subclase Coche
        Coche coche = new Coche();
        coche.acelerar(); // Imprime: "El coche acelera."
        coche.detener();  // Imprime: "El vehículo se detiene."
    }
}
```


Explicación: en este ejemplo, Vehiculo es una clase abstracta que contiene un método abstracto acelerar() y un método concreto detener(). La clase Coche proporciona una implementación específica para el método abstracto acelerar(). Los métodos concretos en clases abstractas permiten compartir lógica común entre las subclases.

Ejemplo 3: uso de una interfaz con métodos abstractos

Enunciado: crea una interfaz Figura con un método abstracto calcularArea(). Luego, implementa la interfaz en una clase Circulo que calcule el área de un círculo.

Código:

```
// Interfaz con método abstracto
interface Figura {
    double calcularArea();
}

// Clase que implementa la interfaz
class Circulo implements Figura {
    private double radio;

    public Circulo(double radio) {
        this.radio = radio;
    }

    @Override
    public double calcularArea() {
        return Math.PI * radio * radio;
    }
}


public class EjemploAbstracto3 {
    public static void main(String[] args) {
        // Crear un objeto de la clase Circulo
        Circulo circulo = new Circulo(5.0);

        // Calcular el área del círculo
        double area = circulo.calcularArea();
        System.out.println("El área del círculo es: " + area);
    }
}
```


Explicación: en este ejemplo, se crea una interfaz Figura con un método abstracto calcularArea(). La clase Circulo implementa esta interfaz y proporciona una implementación del método para calcular el área de un círculo. Las interfaces en Java son una forma de garantizar que las clases que las implementan sigan un contrato específico.

Los **métodos abstractos** permiten a las clases definir comportamientos que deben ser implementados por sus subclases, proporcionando una base común para un grupo de clases relacionadas. Las clases abstractas y las interfaces en Java ofrecen un diseño flexible para la programación orientada a objetos, lo que permite definir comportamientos sin especificar los detalles de su implementación en las clases base.

6.5. Sobreescritura de métodos

La sobreescritura es fundamental para la creación de jerarquías de clases en las que las subclases pueden personalizar o modificar el comportamiento de sus superclases, promoviendo la reutilización de código y el polimorfismo.

La **sobreescritura de métodos** (o *method overriding* en inglés) es una característica de la programación orientada a objetos que permite a una subclase proporcionar una implementación específica de un método que ya está definido en su superclase.

Cuando un método es sobreescrito, el nuevo método en la subclase reemplaza la versión del método en la superclase, permitiendo que la subclase modifique o amplíe el comportamiento original del método.

Para + info

La **sobreescritura de métodos** permite a las subclases modificar el comportamiento de los métodos heredados de sus superclases.

Cómo se trabaja con ella en Java

En Java, para sobreescribir un método, la subclase debe definir un método con el mismo nombre, el mismo tipo de retorno y los mismos parámetros que el método en la superclase.

Es una buena práctica (y en algunos casos necesario) utilizar la anotación @Override antes del método sobreescrito para indicar que se está sobreescribiendo un método de la superclase. Esto también ayuda al compilador a verificar que la sobreescritura se está realizando correctamente.

Declaración de un método sobreescrito

```
@Override
public tipoRetorno nombreMetodo(TipoParametro parame-
tro) {
    // Nueva implementación del método
}
```

Ejemplos de cómo trabajar la sobreescritura en Java:

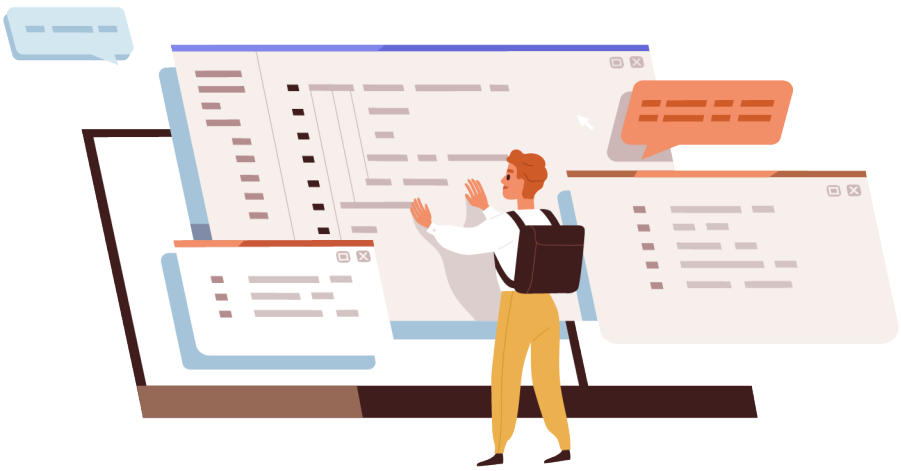
Ejemplo básico de sobreescritura de métodos

```
// Superclase
class Animal {
    public void hacerSonido() {
        System.out.println("El animal hace un sonido");
    }
}

// Subclase
class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El perro ladra");
    }
}
```

Uso de la sobreescritura en Perro:

```
public class Main {
    public static void main(String[] args) {
        Animal miAnimal = new Animal();
        miAnimal.hacerSonido();  // Llama al método de la superclase

        Perro miPerro = new Perro();
        miPerro.hacerSonido();  // Llama al método sobreescrito en la
                                   subclase
    }
}
```

En este ejemplo, la clase Perro sobrescribe el método hacerSonido de la clase Animal. Al llamar al método en un objeto de tipo Perro, se ejecuta la versión sobreescrita del método.



Ejemplo avanzado con sobreescritura y acceso a la superclase

```
// Superclase
class Vehiculo {
    public void arrancar() {
        System.out.println("El vehículo está arrancando");
    }
}

// Subclase
class Coche extends Vehiculo {
    @Override
    public void arrancar() {
        // Llama al método de la superclase
        super.arrancar();
        // Añade comportamiento específico
        System.out.println("El coche está listo para conducir");
    }
}
```

Uso de la sobreescritura en Coche:

```
public class Main {
    public static void main(String[] args) {
        Coche miCoche = new Coche();
        miCoche.arrancar();  // Llama al método sobreescrito en la
                             subclase
    }
}
```

En este ejemplo, la clase Coche sobrescribe el método arrancar de la clase Vehiculo, pero también utiliza super.arrancar() para llamar al método original de la superclase antes de agregar su propio comportamiento.

Ejemplo con sobreescritura de constructores

La sobreescritura de constructores como tal no es posible en Java, ya que los constructores no se heredan. Sin embargo, se puede lograr un comportamiento similar mediante la **sobrecarga** de constructores (que es diferente de la sobreescritura). A continuación, se muestra cómo se puede implementar:

```
// Superclase
class Persona {
    String nombre;

    // Constructor de la superclase
    public Persona(String nombre) {
        this.nombre = nombre;
        System.out.println("Constructor de Persona llamado");
```

```
    }
}

// Subclase
class Empleado extends Persona {
    String puesto;

    // Constructor de la subclase
    public Empleado(String nombre, String puesto) {
        super(nombre);  // Llama al constructor de la superclase
        this.puesto = puesto;
        System.out.println("Constructor de Empleado llamado");
    }
}
```

Uso de la sobrecarga en Empleado:

```
public class Main {
    public static void main(String[] args) {
        Empleado empleado1 = new Empleado("Carlos García", "Desarrolla-
dor");
        System.out.println("Nombre: " + empleado1.nombre);
        System.out.println("Puesto: " + empleado1.puesto);
    }
}
```


En este ejemplo, el constructor de la subclase Empleado llama al constructor de la superclase Persona utilizando super(nombre). Aunque los constructores no se sobreescriben, se puede ver cómo se pueden extender utilizando el mecanismo de herencia y sobrecarga.

6.6. Polimorfismo

El polimorfismo es uno de los conceptos fundamentales de la programación orientada a objetos (POO). Permite que un objeto de una clase derivada sea tratado como un objeto de su clase base o de una interfaz que implementa, permitiendo así que el mismo método tenga diferentes comportamientos según el objeto que lo invoque. En términos simples, el polimorfismo permite que las mismas operaciones se realicen de manera diferente en distintas clases.

Existen dos tipos principales de polimorfismo:

- **Polimorfismo en tiempo de compilación (sobrecarga de métodos):** cuando un método con el mismo nombre puede tener diferentes implementaciones según el tipo de parámetros o su número.

- **Polimorfismo en tiempo de ejecución (sobreescritura de métodos):** cuando un método de una clase base es redefinido por una subclase, permitiendo que un objeto pueda comportarse de manera diferente dependiendo de su tipo específico en tiempo de ejecución.

Cómo se utiliza en Java

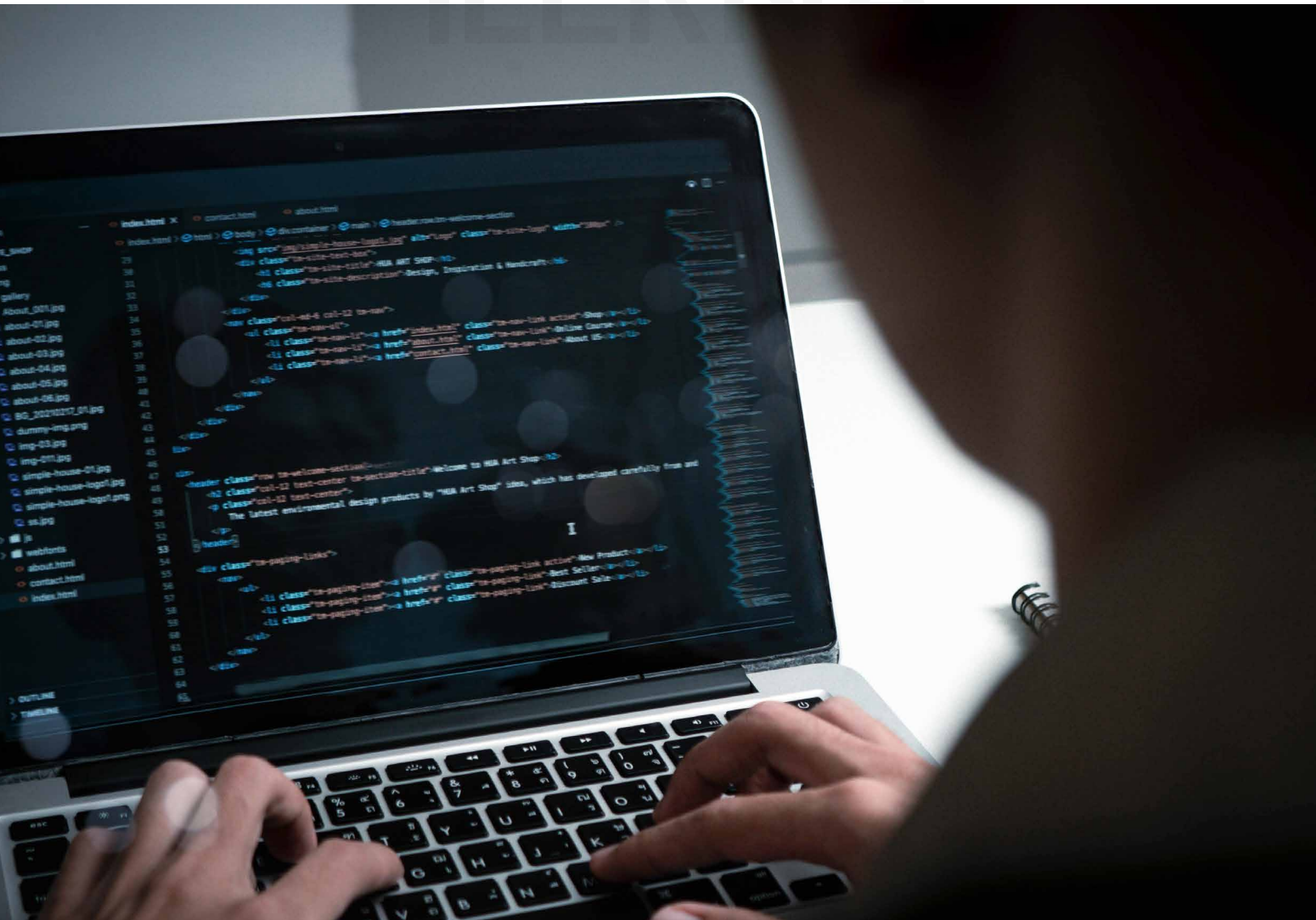
En Java, el polimorfismo se logra principalmente a través de la **herencia** y las **interfaces**. El polimorfismo en tiempo de ejecución se manifiesta cuando un método sobreescrito en una subclase es invocado a través de una referencia de la clase base. Java también admite **sobrecarga de métodos** como una forma de polimorfismo en tiempo de compilación.

Uso básico del polimorfismo en Java

- Un método puede ser sobrescrito en una subclase, lo que permite a la clase derivada modificar el comportamiento de la clase base.
- Los objetos de una subclase pueden ser referenciados por variables del tipo de la clase base, permitiendo el uso de una interfaz común para diferentes clases.

Para + info

El **polimorfismo** permite que un método tenga diferentes comportamientos según el objeto que lo invoque.



Ejemplos usando polimorfismo en Java:

Ejemplo 1: polimorfismo con sobreescritura de métodos

Enunciado: crea una clase base Animal con un método hacerSonido(). Implementa dos clases derivadas Perro y Gato, cada una con su propia implementación del método hacerSonido().

Código:

```
// Clase base
class Animal {
    public void hacerSonido() {
        System.out.println("El animal hace un sonido.");
    }
}

// Subclase Perro
class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El perro ladra.");
    }
}

// Subclase Gato
class Gato extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El gato maúlla.");
    }
}

public class EjemploPolimorfismo1 {
    public static void main(String[] args) {
        Animal miAnimal = new Animal();
        Animal miPerro = new Perro();
        Animal miGato = new Gato();

        miAnimal.hacerSonido();  // Imprime: "El animal hace un sonido."
        miPerro.hacerSonido();   // Imprime: "El perro ladra."
        miGato.hacerSonido();    // Imprime: "El gato maúlla."
    }
}
```


Explicación: en este ejemplo, la clase Animal tiene un método hacerSonido(), que es sobrescrito por las clases Perro y Gato. Dependiendo del tipo de objeto (Perro o Gato), se invoca la versión correspondiente de hacerSonido() en tiempo de ejecución, lo que demuestra el polimorfismo en acción.

Ejemplo 2: polimorfismo con interfaces

Enunciado: crea una interfaz Imprimible que tenga un método imprimir(), y luego implementa la interfaz en las clases Documento e Imagen.

Código:

```
// Definición de la interfaz
interface Imprimible {
    void imprimir();
}

// Clase Documento que implementa la interfaz
class Documento implements Imprimible {
    private String texto;

    public Documento(String texto) {
        this.texto = texto;
    }

    @Override
    public void imprimir() {
        System.out.println("Imprimiendo documento: " + texto);
    }
}

// Clase Imagen que implementa la interfaz
class Imagen implements Imprimible {
    private String archivo;

    public Imagen(String archivo) {
        this.archivo = archivo;
    }

    @Override
    public void imprimir() {
        System.out.println("Imprimiendo imagen desde el archivo: " + ar-
chivo);
    }
}

public class EjemploPolimorfismo2 {
    public static void main(String[] args) {
        Imprimible doc = new Documento("Reporte de Ventas");
        Imprimible img = new Imagen("foto.png");

        doc.imprimir();  // Imprime: "Imprimiendo documento: Reporte de
Ventas"
        img.imprimir();  // Imprime: "Imprimiendo imagen desde el archivo:
foto.png"
    }
}
```

Explicación: aquí, la interfaz Imprimible asegura que tanto la clase Documento como la clase Imagen implementen el método imprimir(). El polimorfismo permite tratar ambos objetos como Imprimible, pero la implementación de imprimir() es diferente según la clase.

Ejemplo 3: polimorfismo y sobrecarga de métodos

Enunciado: crea una clase Calculadora que tenga dos métodos sumar(), uno para sumar enteros y otro para sumar decimales.

Código:

```
class Calculadora {
    // Sobrecarga de método sumar para enteros
    public int sumar(int a, int b) {
        return a + b;
    }

    // Sobrecarga de método sumar para decimales
    public double sumar(double a, double b) {
        return a + b;
    }
}

public class EjemploPolimorfismo3 {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();

        // Usar el método sumar con enteros
        int sumaEnteros = calc.sumar(5, 10);
        System.out.println("Suma de enteros: " + sumaEnteros);
        // Imprime: "Suma de enteros: 15"

        // Usar el método sumar con decimales
        double sumaDecimales = calc.sumar(5.5, 3.3);
        System.out.println("Suma de decimales: " + sumaDecimales);
        // Imprime: "Suma de decimales: 8.8"
    }
}
```

Explicación: este ejemplo muestra el polimorfismo en tiempo de compilación, también conocido como **sobrecarga de métodos**. La clase Calculadora tiene dos métodos sumar() con diferentes tipos de parámetros (enteros y decimales). Dependiendo de los argumentos proporcionados, el compilador selecciona la versión correcta del método.

El **polimorfismo** es un concepto clave en la programación orientada a objetos que permite que una misma operación se realice de diferentes maneras dependiendo del objeto que la invoque.

En Java, el polimorfismo se logra principalmente mediante la sobreescritura de métodos y el uso de interfaces, permitiendo que las clases compartan una interfaz común mientras implementan sus propios comportamientos específicos.

También se manifiesta a través de la sobrecarga de métodos, lo que permite que una clase ofrezca múltiples versiones del mismo método para diferentes tipos de parámetros.

6.7. *Casting* entre tipos de referencia

El ***casting* entre tipos** en la programación orientada a objetos se refiere a la conversión de un tipo de referencia a otro dentro de una jerarquía de clases. Esto ocurre cuando se trata de convertir un objeto de un tipo a otro, ya sea de una clase base a una subclase (*casting* descendente o ***downcasting***) o de una subclase a una clase base (*casting* ascendente o ***upcasting***).

El ***upcasting*** es seguro y no requiere una conversión explícita, ya que una subclase siempre es un tipo de su clase base. El ***downcasting***, sin embargo, debe realizarse explícitamente y puede producir una excepción en tiempo de ejecución si el objeto no es realmente una instancia de la subclase a la que se está convirtiendo.

Para + info

El ***casting*** entre tipos permite convertir referencias entre clases relacionadas dentro de una jerarquía, ya sea de una subclase a una clase base (*upcasting*) o de una clase base a una subclase (*downcasting*).

Cómo se utiliza en Java

En Java, el ***upcasting*** se realiza automáticamente, mientras que el ***downcasting*** requiere el uso de un *casting* explícito. Para evitar excepciones como ClassCastException, se suele utilizar el operador instanceof antes de intentar un *downcasting*, para asegurarse de que la conversión es válida.

Sintaxis del *casting* en Java

***Upcasting* (implícito):**

```
ClaseBase obj = new SubClase();
```

***Downcasting* (explícito):**

```
SubClase obj2 = (SubClase) obj;
```

Ejemplos usando *casting* entre tipos en Java:

Ejemplo 1: *upcasting* (*casting* ascendente)

Enunciado: crea una clase Animal y una clase derivada Perro. Luego, realiza un *upcasting* automático de un objeto de la subclase Perro a su clase base Animal.

Código:

```
// Clase base
class Animal {
    public void hacerSonido() {
        System.out.println("El animal hace un sonido.");
    }
}

// Clase derivada
class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El perro ladra.");
    }
}

public class EjemploCasting1 {
    public static void main(String[] args) {
        // Crear un objeto de la clase derivada
        Perro miPerro = new Perro();

        // Upcasting implícito (de Perro a Animal)
        Animal miAnimal = miPerro;

        // Se invoca el método de la clase base, pero con el comportamien-
to de la subclase
        miAnimal.hacerSonido(); // Imprime: "El perro ladra."
    }
}
```

Explicación: en este ejemplo, se realiza un ***upcasting* implícito**. El objeto miPerro de la clase Perro se convierte automáticamente en un objeto de tipo Animal, lo que permite tratar al perro como un animal. Aunque miAnimal es de tipo Animal, el método sobreescrito en Perro se ejecuta gracias al polimorfismo.

Ejemplo 2: *downcasting* (*casting* descendente)

Enunciado: realiza un *casting* descendente para convertir una referencia de la clase base Animal a una referencia de la subclase Perro.

Código:

```
// Clase base
class Animal {
    public void hacerSonido() {
        System.out.println("El animal hace un sonido.");
    }
}

// Clase derivada
class Perro extends Animal {
    public void ladrar() {
        System.out.println("El perro ladra.");
    }
}

public class EjemploCasting2 {
    public static void main(String[] args) {
        // Upcasting implícito
        Animal miAnimal = new Perro();

        // Downcasting explícito (de Animal a Perro)
        if (miAnimal instanceof Perro) {
            Perro miPerro = (Perro) miAnimal;
            miPerro.ladrar(); // Imprime: "El perro ladra."
        } else {
            System.out.println("No se puede hacer casting.");
        }
    }
}
```

Explicación: en este ejemplo, el objeto miAnimal es de tipo Animal, pero en realidad apunta a un objeto de la clase Perro. Se realiza un ***downcasting* explícito** para convertir la referencia Animal a Perro, permitiendo el acceso a los métodos específicos de la subclase Perro. Antes de realizar el *casting*, se usa instanceof para verificar que el objeto es efectivamente de la subclase Perro, evitando una posible excepción.

Ejemplo 3: *casting* entre interfaces y clases

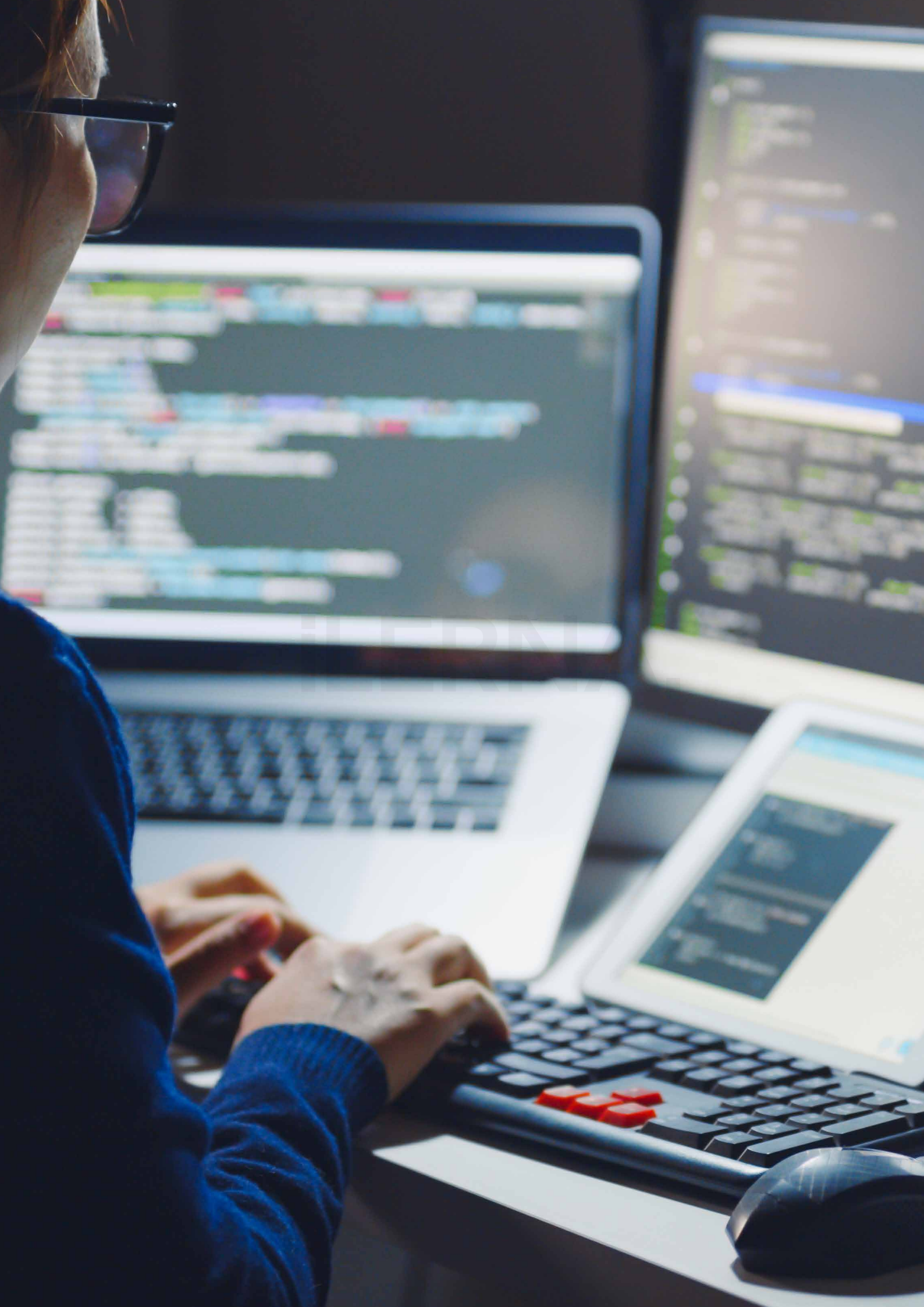
Enunciado: utiliza una interfaz Volador y una clase Avion que la implementa. Realiza un *casting* entre la interfaz y la clase concreta.

Código:

```
// Definición de la interfaz
interface Volador {
    void volar();
}

// Clase que implementa la interfaz
class Avion implements Volador {
    @Override
    public void volar() {
        System.out.println("El avión está volando.");
    }

    public void aterrizar() {
        System.out.println("El avión está aterrizando.");
    }
}

public class EjemploCasting3 {
    public static void main(String[] args) {
        // Upcasting implícito de Avion a Volador
        Volador miVolador = new Avion();
        miVolador.volar(); // Imprime: "El avión está volando."

        // Downcasting explícito de Volador a Avion
        if (miVolador instanceof Avion) {
            Avion miAvion = (Avion) miVolador;
            miAvion.aterrizar(); // Imprime: "El avión está aterrizando."
        }
    }
}
```

Explicación: en este ejemplo, se realiza un ***upcasting* implícito** de un objeto Avion a la interfaz Volador, lo que permite tratar al avión como un objeto que implementa la interfaz Volador. Luego, se realiza un ***downcasting* explícito** de vuelta a la clase concreta Avion para acceder a métodos específicos de la clase, como aterrizar(). Se usa instanceof para asegurarse de que el objeto puede ser convertido.

El ***casting* entre tipos de referencia** es una herramienta fundamental en la programación orientada a objetos para convertir objetos dentro de una jerarquía de clases o interfaces. El ***upcasting*** permite tratar objetos específicos como instancias de su clase base, mientras que el ***downcasting*** permite convertir referencias de clases base a subclases, lo que da acceso a métodos y comportamientos más específicos. Para evitar excepciones, es una buena práctica usar instanceof antes de realizar un *downcasting*.



7

ESTRUCTURAS DE DATOS

Las **estructuras de datos** son componentes fundamentales en la programación que permiten organizar, gestionar y almacenar datos de manera eficiente para que puedan ser utilizados de forma efectiva en un programa. Diferentes tipos de estructuras de datos están diseñados para manejar diferentes tipos de necesidades, desde listas y pilas hasta árboles y grafos, cada uno con sus propias ventajas y casos de uso. Comprender cómo funcionan las estructuras de datos y cuándo utilizarlas es crucial para diseñar algoritmos eficientes y escribir código optimizado.

Exploramos varias estructuras de datos, comenzando con arrays, que son una de las estructuras más básicas y utilizadas en la programación. Los arrays permiten almacenar múltiples elementos del mismo tipo en una estructura lineal, accesibles por su índice.

Las **estructuras de datos** organizan y gestionan la información en un programa, permitiendo un acceso y manipulación eficientes de los datos.

Enunciado del ejercicio:

Supongamos que tienes un array que almacena las calificaciones de 5 estudiantes en una asignatura. Las calificaciones son las siguientes: 8, 9, 7, 10, 6.

Tareas:

- **Acceder a elementos del array**: muestra la calificación del tercer estudiante.
- **Modificar un elemento**: actualiza la calificación del segundo estudiante a 10.
- **Calcular el promedio**: calcula el promedio de todas las calificaciones en el array.
- **Agregar una calificación**: considera cómo agregarías una nueva calificación al array (aunque los arrays en su forma básica tienen un tamaño fijo).

Resolución utilizando tablas:

Índice	Calificación original	Acción	Calificación final
0	8	-	8
1	9	Modificar a 10	10
2	7	-	7
3	10	-	10
4	6	-	6

Pasos para resolver:

1. **Acceso al tercer elemento:**
   - **Calificación del tercer estudiante:** 7 (corresponde al índice 2).
2. **Modificación del segundo elemento:**
   - **Antes**: calificación del segundo estudiante: 9.
   - **Después**: se actualiza a 10.
3. **Cálculo del promedio:**
   - **Suma de las calificaciones finales**: 8 + 10 + 7 + 10 + 6 = 41.
   - **Número de estudiantes**: 5.
   - **Promedio**: 41 / 5 = 8.2.
4. **Agregar una calificación:**
   - **Observación**: los arrays en su forma básica no permiten cambiar de tamaño. Para agregar una nueva calificación, tendrías que crear un nuevo array con más espacio o usar una estructura de datos dinámica como una lista.

Manipular datos almacenados en un array se puede realizar utilizando operaciones básicas como acceso, modificación y cálculo de promedios, y también introduce la limitación de tamaño fijo en los arrays.

7.1. Arrays

Un **array** es una estructura de datos que permite almacenar múltiples elementos del mismo tipo en una secuencia contigua de memoria. Cada elemento en un array se puede acceder utilizando un índice numérico, que representa su posición en la secuencia, comenzando desde 0. Los arrays son muy útiles para organizar y manejar conjuntos de datos que necesitan ser procesados en conjunto, como listas de números, cadenas de texto o cualquier otro tipo de datos. Una de las características clave de los arrays es que tienen un tamaño fijo, es decir, el número de elementos que pueden contener se define en el momento de su creación y no puede cambiar.

Un **array** es una estructura de datos que almacena múltiples elementos del mismo tipo en un espacio de memoria contiguo, accesibles mediante un índice.

Cómo se utilizan en Java

En Java, los arrays se utilizan para almacenar y acceder a colecciones de datos homogéneos de manera eficiente. Los arrays se definen especificando el tipo de los elementos que contendrán, seguido de corchetes ([]). Una vez definido, se puede inicializar el array especificando su tamaño o los valores de sus elementos.

Declaración e inicialización de un array en Java:

- **Declaración**:

```
int[] numeros;
```

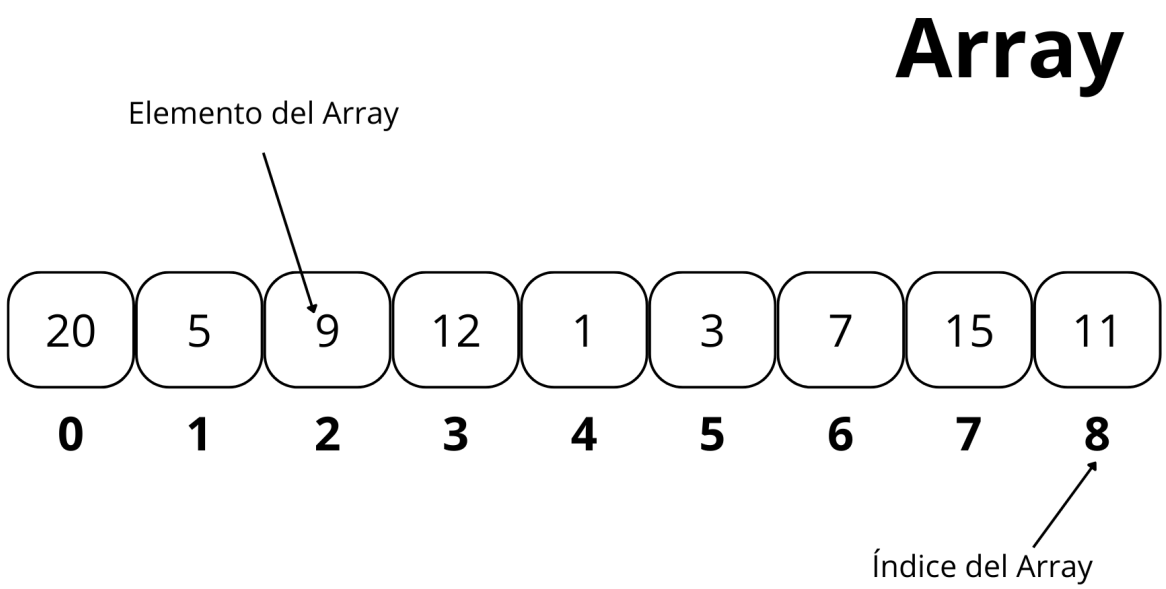
- **Inicialización con tamaño**:

```
numeros = new int[5];
```

- **Inicialización con valores**:

```
int[] numeros = {1, 2, 3, 4, 5};
```

Los elementos del array se acceden mediante su índice, como numeros[0] para acceder al primer elemento, y se pueden modificar asignando nuevos valores a posiciones específicas.



Ejemplos usando arrays:

Ejemplo 1: creación y acceso a elementos

Enunciado: crear un array que almacene los nombres de 4 ciudades y muestra el nombre de la segunda ciudad.

Código:

```
public class EjemploArray1 {
    public static void main(String[] args) {
        // Crear un array de cadenas para almacenar nombres de ciudades
        String[] ciudades = {"Madrid", "Barcelona", "Valencia", "Sevilla"};

        // Acceder al segundo elemento (índice 1) del array
        System.out.println("La segunda ciudad es: " + ciudades[1]);
    }
}
```

Explicación: en este ejemplo, se crea un array de cadenas ciudades que almacena los nombres de cuatro ciudades. Luego, se accede al segundo elemento del array usando ciudades[1], que corresponde a "Barcelona" y se imprime en la consola.

Ejemplo 2: modificación de elementos y cálculo de promedio

Enunciado: crea un array que almacene 5 calificaciones, modifica una de ellas y calcula el promedio.

Código:

```
public class EjemploArray2 {
    public static void main(String[] args) {
        // Crear un array de enteros para almacenar calificaciones
        int[] calificaciones = {85, 90, 78, 92, 88};

        // Modificar la calificación del tercer estudiante (índice 2)
        calificaciones[2] = 80;

        // Calcular el promedio
        int suma = 0;
        for (int i = 0; i < calificaciones.length; i++) {
            suma += calificaciones[i];
        }
        double promedio = suma / (double) calificaciones.length;

        // Mostrar el promedio
        System.out.println("El promedio de las calificaciones es: " + pro-
medio);
    }
}
```

Explicación: este ejemplo crea un array de enteros calificaciones que almacena cinco calificaciones. Luego, se modifica la tercera calificación y se calcula el promedio de todas las calificaciones, que se muestra en la consola.

Ejemplo 3: búsqueda de un elemento en el array

Enunciado: crea un array de números enteros y busca un número específico en el array, indicando si se encuentra o no.

Código:

```
public class EjemploArray3 {
    public static void main(String[] args) {
        // Crear un array de enteros
        int[] numeros = {3, 7, 12, 5, 9};

        // Número a buscar
        int numeroBuscado = 12;
        boolean encontrado = false;

        // Buscar el número en el array
        for (int i = 0; i < numeros.length; i++) {
            if (numeros[i] == numeroBuscado) {
                encontrado = true;
                break;
            }
        }

        // Mostrar el resultado de la búsqueda
        if (encontrado) {
            System.out.println("El número " + numeroBuscado + " se
                                encuentra en el array.");
        } else {
            System.out.println("El número " + numeroBuscado + " no se
                                encuentra en el array.");
        }
    }
}
```

Explicación: en este ejemplo, se crea un array de enteros numeros y se busca un número específico (en este caso, 12) en el array. Si se encuentra, se imprime un mensaje indicando su presencia; de lo contrario, se indica que no está en el array.

Los arrays se pueden utilizar en Java para almacenar, acceder, modificar y buscar datos, lo que demuestra su versatilidad como una estructura de datos fundamental en la programación.

7.2. Genericidad

La **genericidad** en programación es un mecanismo que permite a los desarrolladores crear clases, interfaces y métodos que operan con tipos de datos que se especifican en tiempo de ejecución, en lugar de estar definidos de manera rígida en el código fuente. En otras palabras, la genericidad permite escribir código más flexible y reutilizable, ya que se pueden crear estructuras de datos y algoritmos que funcionan con cualquier tipo de dato, sin comprometer la seguridad de tipos en el proceso.

En Java, la genericidad se implementa utilizando **generics**, que se definen mediante la sintaxis de tipo <T> en las clases, interfaces y métodos. Aquí, T es un marcador de tipo genérico que puede ser reemplazado por cualquier tipo de dato en tiempo de compilación.

La **genericidad** permite crear clases y métodos que pueden trabajar con cualquier tipo de dato, haciéndolos más flexibles y reutilizables.

Cómo se utiliza en Java

En Java, los **generics** se utilizan para crear clases y métodos que no están restringidos a un tipo de dato específico. Esto se logra declarando uno o más parámetros de tipo genérico en la definición de la clase o método. Estos parámetros de tipo son reemplazados por tipos reales cuando se instancia la clase o se llama al método.

Ejemplo básico de genericidad:

```
public class Caja<T> {
    private T contenido;

    public void guardar(T item) {
        this.contenido = item;
    }

    public T obtener() {
        return contenido;
    }
}
```

En este ejemplo, Caja<T> es una clase genérica donde T representa un tipo de dato que será especificado más tarde. Esta clase puede almacenar y devolver un elemento de cualquier tipo.

Tres ejemplos usando la genericidad con arrays:

Ejemplo 1: clase genérica con arrays

Enunciado: crea una clase genérica ContenedorArray que pueda almacenar un array de cualquier tipo de datos y permita acceder a los elementos del array.

Código:

```
public class ContenedorArray<T> {
    private T[] array;

    public ContenedorArray(T[] array) {
        this.array = array;
    }

    public T obtenerElemento(int indice) {
        if (indice >= 0 && indice < array.length) {
            return array[indice];
        } else {
            throw new IndexOutOfBoundsException("Índice fuera de rango");
        }
    }

    public int obtenerLongitud() {
        return array.length;
    }
}

public class EjemploGenericidad1 {
    public static void main(String[] args) {
        Integer[] numeros = {1, 2, 3, 4, 5};
        ContenedorArray<Integer> contenedor = new ContenedorArray<>(nume-
ros);

        System.out.println("Elemento en el índice 2: " +
                           contenedor.obtenerElemento(2));
        System.out.println("Longitud del array: " + contenedor.obtenerLon-
gitud());
    }
}
```

Explicación: en este ejemplo, la clase ContenedorArray<T> es una clase genérica que almacena un array de cualquier tipo T. Se puede acceder a los elementos del array y obtener su longitud utilizando métodos genéricos. Aquí se demuestra con un array de enteros Integer[].

Ejemplo 2: método genérico para buscar en arrays

Enunciado: crea un método genérico que busque un elemento en un array y devuelva el índice del primer elemento coincidente o -1 si no se encuentra.

Código:

```
public class UtilidadesArray {
    public static <T> int buscarElemento(T[] array, T elemento) {
        for (int i = 0; i < array.length; i++) {
            if (array[i].equals(elemento)) {
                return i;
            }
        }
        return -1;
    }
}

public class EjemploGenericidad2 {
    public static void main(String[] args) {
        String[] nombres = {"Ana", "Juan", "Pedro", "Lucía"};
        int indice = UtilidadesArray.buscarElemento(nombres, "Pedro");

        if (indice != -1) {
            System.out.println("El nombre 'Pedro' se encuentra en el índi-
ce: " +
                               indice);
        } else {
            System.out.println("El nombre 'Pedro' no se encuentra en el
array.");
        }
    }
}
```

Explicación: el método buscarElemento es un método genérico que puede trabajar con arrays de cualquier tipo T. El método recorre el array y busca el primer elemento que coincida con el valor proporcionado, devolviendo su índice o -1 si no se encuentra.

Ejemplo 3: intercambiar elementos en un array genérico

Enunciado: crea un método genérico que intercambie dos elementos en un array.

Código:

```
publ36+ic class UtilidadesArray {
    public static <T> void intercambiarElementos(T[] array, int indice1,
int
                                                  indice2) {
        if (indice1 >= 0 && indice1 < array.length && indice2 >= 0 && in-
dice2 <
            array.length) {
            T temp = array[indice1];
            array[indice1] = array[indice2];
            array[indice2] = temp;
        } else {
            throw new IndexOutOfBoundsException("Índice fuera de rango");
        }
    }
}


public class EjemploGenericidad3 {
    public static void main(String[] args) {
        String[] frutas = {"Manzana", "Naranja", "Plátano", "Pera"};

        System.out.println("Array antes del intercambio:");
        for (String fruta : frutas) {
            System.out.print(fruta + " ");
        }

        UtilidadesArray.intercambiarElementos(frutas, 1, 3);

        System.out.println("\nArray después del intercambio:");
        for (String fruta : frutas) {
            System.out.print(fruta + " ");
        }
    }
}
```

Explicación: este método genérico intercambiarElementos intercambia dos elementos en un array genérico T[]. Es flexible y puede trabajar con arrays de cualquier tipo. En este ejemplo, se utiliza un array de cadenas (String[]) para demostrar cómo se intercambian los elementos "Naranja" y "Pera".

La genericidad puede hacer que el código sea más flexible y reutilizable, permitiendo a los desarrolladores crear soluciones que funcionen con cualquier tipo de datos. La genericidad es una característica poderosa en Java que promueve la reutilización del código y reduce la duplicación de esfuerzos al manejar múltiples tipos de datos.

7.3. Listas enlazadas

Una **lista enlazada** es una estructura de datos lineal que consiste en una secuencia de elementos llamados nodos, donde cada nodo contiene dos partes: un **valor** (o dato) y una **referencia** (o enlace) al siguiente nodo en la secuencia. A diferencia de los arrays, las listas enlazadas no almacenan sus elementos en ubicaciones de memoria contiguas; en su lugar, cada elemento está conectado al siguiente, formando una cadena. Existen varios tipos de listas enlazadas, como las listas enlazadas simples, listas doblemente enlazadas y listas circulares.

En una **lista enlazada simple**, cada nodo tiene un enlace al siguiente nodo, mientras que el último nodo apunta a null, indicando el final de la lista. En una **lista doblemente enlazada**, cada nodo tiene un enlace al siguiente nodo y un enlace al nodo anterior, lo que permite la navegación en ambas direcciones.

Las **listas enlazadas** son estructuras de datos donde cada elemento se conecta al siguiente mediante un enlace, permitiendo una organización dinámica de datos.

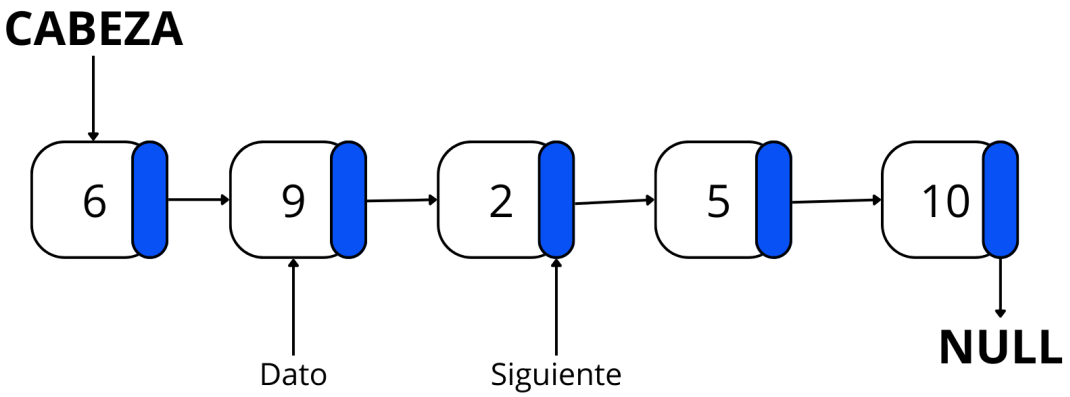
Cómo se utilizan en Java

En Java, las listas enlazadas se implementan a través de la clase LinkedList, que es parte del paquete java.util. Por su parte, LinkedList es una implementación de lista doblemente enlazada, lo que significa que cada nodo tiene referencias tanto al siguiente nodo como al anterior. Esta clase proporciona métodos para realizar operaciones comunes en listas, como añadir, eliminar y buscar elementos de manera eficiente.

Características clave de LinkedList en Java

- **Inserción y eliminación**: son más eficientes en comparación con los arrays para operaciones que implican inserciones y eliminaciones frecuentes, ya que no es necesario desplazar elementos.
- **Acceso secuencial**: permiten un acceso eficiente a los elementos de la lista en secuencia, aunque el acceso aleatorio a los elementos es menos eficiente que en un array.

Lista Enlazada



Tres ejemplos usando listas enlazadas con la librería de Java:

Ejemplo 1: creación y operaciones básicas con LinkedList

Enunciado: crea una LinkedList de cadenas para almacenar nombres, añade algunos nombres a la lista y muestra los nombres almacenados.

Código:

```
import java.util.LinkedList;

public class EjemploLinkedList1 {
    public static void main(String[] args) {
        // Crear una LinkedList para almacenar nombres
        LinkedList<String> nombres = new LinkedList<>();

        // Añadir elementos a la LinkedList
        nombres.add("Juan");
        nombres.add("Ana");
        nombres.add("Pedro");

        // Mostrar los elementos en la LinkedList
        System.out.println("Nombres en la lista:");
        for (String nombre : nombres) {
            System.out.println(nombre);
        }
    }
}
```

Explicación: en este ejemplo, se crea una LinkedList de cadenas para almacenar nombres. Se añaden algunos nombres a la lista usando el método add() y luego se recorren y muestran los nombres utilizando un bucle for-each.

Ejemplo 2: inserción y eliminación de elementos

Enunciado: crea una LinkedList de números enteros, inserta un número al principio y otro al final, y elimina el primer y último número de la lista.

Código:

```
import java.util.LinkedList;

public class EjemploLinkedList2 {
    public static void main(String[] args) {
        // Crear una LinkedList para almacenar números enteros
        LinkedList<Integer> numeros = new LinkedList<>();

        // Añadir elementos a la LinkedList
        numeros.add(10);
        numeros.add(20);
        numeros.add(30);

        // Insertar al principio y al final
        numeros.addFirst(5);
        numeros.addLast(35);

        // Eliminar el primer y último elemento
        numeros.removeFirst();
        numeros.removeLast();

        // Mostrar los elementos restantes
        System.out.println("Números en la lista después de inserciones y
                            eliminaciones:");
        for (int numero : numeros) {
            System.out.println(numero);
        }
    }
}
```

Explicación: este ejemplo muestra cómo insertar y eliminar elementos en una LinkedList. Se añaden números al principio y al final de la lista, y luego se eliminan los elementos del inicio y el final, mostrando el resultado final.

Ejemplo 3: uso de LinkedList como cola (Queue)

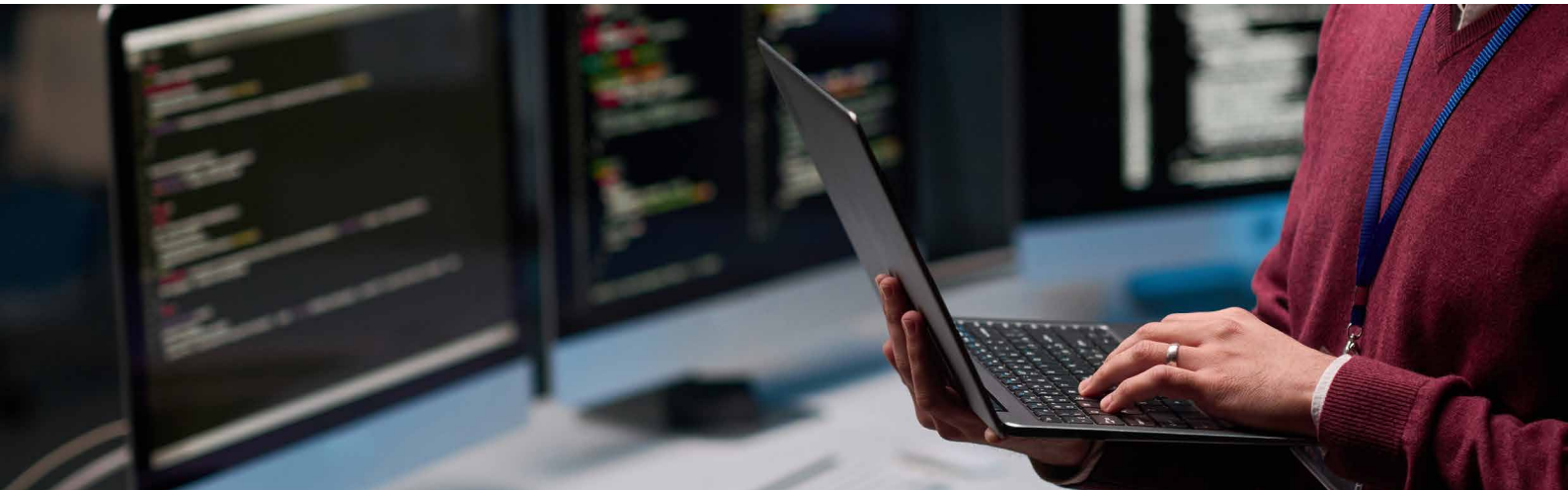
Enunciado: utiliza una LinkedList como una cola para gestionar tareas, donde se añaden tareas al final de la cola y se eliminan tareas del principio.

Código:

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

public class EjemploLinkedList3 {
    public static void main(String[] args) {
        // Crear una LinkedList para gestionar tareas
        Queue<String> tareas = new LinkedList<>();

        // Añadir tareas a la cola
        tareas.add("Tarea 1");
        tareas.add("Tarea 2");
        tareas.add("Tarea 3");

        // Procesar y eliminar tareas de la cola
        while (!tareas.isEmpty()) {
            System.out.println("Procesando: " + tareas.poll());
        }

        // Mostrar el estado de la cola
        System.out.println("Todas las tareas han sido procesadas.");
    }
}
```

Explicación: en este ejemplo, se utiliza una LinkedList como una cola (implementando la interfaz Queue) para gestionar una serie de tareas. Las tareas se añaden al final de la cola y se eliminan del principio a medida que se procesan. Esto ilustra cómo una LinkedList puede ser útil para implementar una cola simple.

Utilizar la LinkedList en Java permite gestionar datos de manera flexible y eficiente, aprovechando las ventajas de las listas enlazadas para inserciones, eliminaciones, y acceso secuencial.



7.4. Pilas

Una **pila** (o stack en inglés) es una estructura de datos lineal que sigue el principio **LIFO** (*Last In, First Out*), lo que significa que el último elemento que se añade a la pila es el primero que se retira. La pila tiene dos operaciones principales:

- **Push**: añade un elemento en la parte superior de la pila.
- **Pop**: retira el elemento que está en la parte superior de la pila.

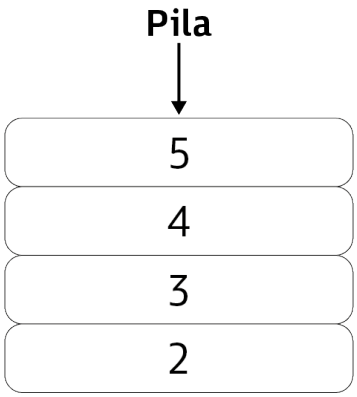
Además, hay otras operaciones comunes como **peek** (o *top*), que permite ver el elemento en la parte superior sin retirarlo, y **isEmpty**, que verifica si la pila está vacía.

Las pilas son muy útiles en escenarios donde es necesario mantener un historial de acciones, como en la navegación web (donde se retrocede a la página anterior), en la evaluación de expresiones aritméticas, y en la gestión de llamadas de funciones recursivas.

Una **pila** es una estructura de datos que sigue el principio LIFO, donde el último elemento en entrar es el primero en salir.

Cómo se utilizan en Java

En Java, las pilas se pueden implementar utilizando la clase Stack del paquete java.util. Esta clase proporciona todos los métodos necesarios para gestionar una pila, como push, pop, peek, isEmpty, entre otros. Aunque Stack es una clase antigua que hereda de Vector, sigue siendo útil para muchas aplicaciones.



Operaciones básicas con Stack en Java

- **Crear una pila:** Stack<T> stack = new Stack<>();
- **Añadir un elemento**: stack.push(elemento);
- **Retirar el elemento superior:** stack.pop();
- **Ver el elemento superior sin retirarlo**: stack.peek();
- **Verificar si la pila está vacía**: stack.isEmpty();

Tres ejemplos usando pilas con la librería de Java:

Ejemplo 1: verificación de paréntesis balanceados

Enunciado: utiliza una pila para verificar si los paréntesis en una expresión están balanceados.

Código:

```
import java.util.Stack;

public class EjemploPilas1 {
    public static boolean verificarParentesisBalanceados(String expresion)
{
        Stack<Character> pila = new Stack<>();

        for (char ch : expresion.toCharArray()) {
            if (ch == '(') {
                pila.push(ch);
            } else if (ch == ')') {
                if (pila.isEmpty()) {
                    return false;
                }
                pila.pop();
            }
        }
        return pila.isEmpty();
    }

    public static void main(String[] args) {
        String expresion1 = "(a + b) * (c - d)";
        String expresion2 = "(a + b * (c - d)";

        System.out.println("Expresión 1 balanceada: " +
                           verificarParentesisBalanceados(expresion1));
        System.out.println("Expresión 2 balanceada: " +
                           verificarParentesisBalanceados(expresion2));
    }
}
```

Explicación: este ejemplo utiliza una pila para verificar si los paréntesis en una expresión están balanceados. Se recorre la cadena de caracteres, añadiendo los paréntesis abiertos a la pila y retirándolos cuando se encuentra un paréntesis cerrado correspondiente. Al final, si la pila está vacía, los paréntesis están balanceados.

Ejemplo 2: conversión de decimal a binario

Enunciado: convierte un número decimal a binario utilizando una pila para almacenar los restos.

Código:

```
import java.util.Stack;

public class EjemploPilas2 {
    public static String convertirDecimalABinario(int numero) {
        Stack<Integer> pila = new Stack<>();
        while (numero > 0) {
            pila.push(numero % 2);
            numero /= 2;
        }

        StringBuilder binario = new StringBuilder();
        while (!pila.isEmpty()) {
            binario.append(pila.pop());
        }
        return binario.toString();
    }

    public static void main(String[] args) {
        int numero = 19;
        String binario = convertirDecimalABinario(numero);
        System.out.println("El número " + numero + " en binario es: " +
binario);
    }
}
```

Explicación: este ejemplo convierte un número decimal a binario utilizando una pila. Los restos obtenidos durante la conversión se almacenan en la pila, y luego se retiran en orden inverso para formar el número binario.

Ejemplo 3: deshacer operaciones

Enunciado: implementa una funcionalidad básica de deshacer utilizando una pila, donde se pueden deshacer las últimas operaciones realizadas.

Código:

```
import java.util.Stack;

public class EjemploPilas3 {
    public static void main(String[] args) {
        Stack<String> pilaOperaciones = new Stack<>();

        // Realizar algunas operaciones
        pilaOperaciones.push("Operación 1");
        pilaOperaciones.push("Operación 2");
        pilaOperaciones.push("Operación 3");

        System.out.println("Operaciones realizadas: " + pilaOperaciones);

        // Deshacer la última operación
        System.out.println("Deshaciendo: " + pilaOperaciones.pop());

        // Mostrar las operaciones restantes
        System.out.println("Operaciones restantes: " + pilaOperaciones);
    }
}
```

Explicación: en este ejemplo, se utiliza una pila para implementar la funcionalidad de deshacer operaciones. Las operaciones se añaden a la pila con push, y cuando se desea deshacer la última operación, se utiliza pop para retirarla de la pila. Luego, se muestra el estado de las operaciones restantes.

Se puede utilizar la clase Stack en Java para implementar diversas funcionalidades prácticas, desde la verificación de expresiones hasta la conversión de números y el manejo de operaciones. Las pilas son una herramienta poderosa y flexible en la programación, especialmente para problemas donde el orden de los elementos es crítico.

7.5. Colas

Una **cola** (o queue en inglés) es una estructura de datos lineal que sigue el principio **FIFO** (*First In, First Out*), lo que significa que el primer elemento que entra en la cola es el primero que sale. En otras palabras, las colas funcionan como una línea de espera en la que los elementos se añaden al final (encolan) y se retiran del principio (desencolan). Las colas son ampliamente utilizadas en situaciones donde se debe mantener el orden de los elementos, como en la gestión de tareas, la impresión de documentos, o el manejo de procesos en un sistema operativo.

Las operaciones principales de una cola son:

- **Offer/enqueue**: añadir un elemento al final de la cola.
- **Poll/dequeue**: retirar el elemento del principio de la cola.
- **Peek**: ver el elemento en el principio de la cola sin retirarlo.
- **isEmpty**: verificar si la cola está vacía.

Una **cola** es una estructura de datos que sigue el principio FIFO, donde el primer elemento en entrar es el primero en salir.

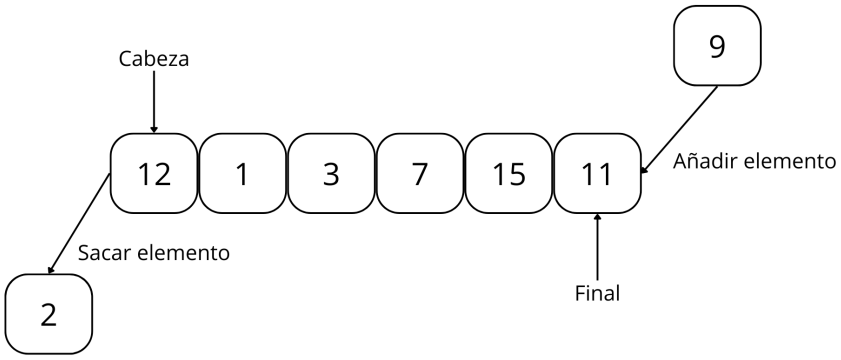
Cómo se utilizan en Java

En Java, las colas se pueden implementar utilizando la interfaz Queue del paquete java.util. Java proporciona varias implementaciones de colas, siendo LinkedList una de las más comunes, así como PriorityQueue, que mantiene los elementos ordenados de acuerdo con una prioridad.

Operaciones básicas con Queue en Java

- **Crear una cola:** Queue<T> queue = new LinkedList<>();
- **Añadir un elemento:** queue.offer(elemento);
- **Retirar el elemento del principio:** queue.poll();
- **Ver el elemento del principio sin retirarlo:** queue.peek();
- **Verificar si la cola está vacía:** queue.isEmpty();

Cola



Tres ejemplos usando colas con la librería de Java:

Ejemplo 1: gestión de tareas con una cola

Enunciado: utiliza una cola para gestionar una lista de tareas pendientes, donde las tareas se añaden al final de la cola y se procesan desde el principio.

Código:

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

public class EjemploColas1 {
    public static void main(String[] args) {
        // Crear una cola de tareas
        Queue<String> tareas = new LinkedList<>();

        // Añadir tareas a la cola
        tareas.offer("Revisar correos");
        tareas.offer("Escribir informe");
        tareas.offer("Preparar reunión");

        // Procesar las tareas en orden FIFO
        while (!tareas.isEmpty()) {
            System.out.println("Procesando tarea: " + tareas.poll());
        }

        System.out.println("Todas las tareas han sido procesadas.");
    }
}
```

Explicación: en este ejemplo, se utiliza una Queue de tipo LinkedList para gestionar tareas. Las tareas se añaden al final de la cola con offer, y se procesan desde el principio con poll. El ciclo continúa hasta que todas las tareas se han procesado.

Ejemplo 2: simulación de una cola de personas en un banco

Enunciado: simula una cola de personas en un banco, donde las personas se añaden a la cola a medida que llegan y se atienden en el orden en que llegaron.

Código:

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

public class EjemploColas2 {
    public static void main(String[] args) {
        // Crear una cola para representar la fila en el banco
        Queue<String> filaBanco = new LinkedList<>();

        // Añadir personas a la cola
        filaBanco.offer("Carlos");
        filaBanco.offer("Ana");
        filaBanco.offer("Luis");
        filaBanco.offer("Marta");

        // Atender a las personas en el orden en que llegaron
        while (!filaBanco.isEmpty()) {
            System.out.println("Atendiendo a: " + filaBanco.poll());
        }

        System.out.println("Todas las personas han sido atendidas.");
    }
}
```

Explicación: este ejemplo simula una fila de personas en un banco utilizando una Queue de LinkedList. Las personas se añaden a la cola cuando llegan y se atienden en el orden en que llegaron. Esto ilustra cómo una cola puede modelar escenarios del mundo real que requieren un orden FIFO.

Ejemplo 3: cola de prioridad para manejo de emergencias

Enunciado: utiliza una PriorityQueue para gestionar emergencias en un hospital, donde las emergencias con mayor prioridad se atienden primero.

Código:

```
import java.util.PriorityQueue;

public class EjemploColas3 {
    public static void main(String[] args) {
        // Crear una PriorityQueue para emergencias (prioridad inversa)
        PriorityQueue<Emergencia> emergencias = new PriorityQueue<>((e1, e2) ->
e2.getPrioridad() - e1.getPrioridad());
```

```
        // Añadir emergencias a la cola
        emergencias.offer(new Emergencia("Infarto", 5));
        emergencias.offer(new Emergencia("Corte menor", 1));
        emergencias.offer(new Emergencia("Fractura de brazo", 3));
        emergencias.offer(new Emergencia("Alergia severa", 4));

        // Atender emergencias según prioridad
        while (!emergencias.isEmpty()) {
            Emergencia emergencia = emergencias.poll();
            System.out.println("Atendiendo emergencia: " + emergencia.getNombre() +
                               " con prioridad " + emergencia.getPrioridad());
        }

        System.out.println("Todas las emergencias han sido atendidas.");
    }
}

class Emergencia {
    private String nombre;
    private int prioridad;

    public Emergencia(String nombre, int prioridad) {
        this.nombre = nombre;
        this.prioridad = prioridad;
    }

    public String getNombre() {
        return nombre;
    }

    public int getPrioridad() {
        return prioridad;
    }
}
```

Explicación: en este ejemplo, se utiliza una PriorityQueue para manejar emergencias en un hospital. Las emergencias se añaden con una prioridad, y la PriorityQueue asegura que las emergencias con mayor prioridad (un número más alto) se atiendan primero. Esto demuestra cómo una cola de prioridad puede ser útil para manejar situaciones donde no solo el orden de llegada, sino también la importancia, deben ser considerados.

Las colas se pueden utilizar en diferentes contextos para gestionar datos y tareas en un orden FIFO. En Java, las colas son una estructura de datos fundamental que se puede aplicar en una amplia variedad de escenarios, desde la gestión de tareas simples hasta sistemas más complejos como la administración de prioridades.

7.6. Grafos

Un **grafo** es una estructura de datos que representa un conjunto de **nodos** (también llamados **vértices**) y las **conexiones** entre ellos, conocidas como **aristas**. Los grafos se utilizan para modelar relaciones y conexiones complejas entre entidades, como redes sociales, rutas de transporte, sistemas de recomendación, y más. Los grafos pueden ser **dirigidos** o **no dirigidos**: en un grafo dirigido, las aristas tienen una dirección (es decir, van de un nodo a otro), mientras que, en un grafo no dirigido, las aristas simplemente conectan dos nodos sin una dirección específica.

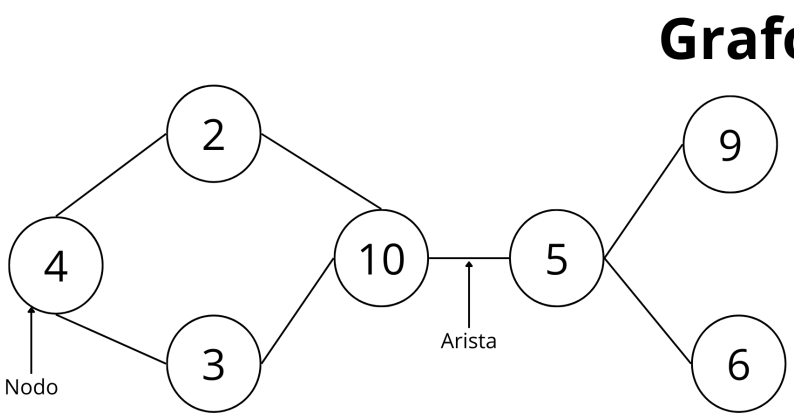
Los grafos pueden ser ponderados o no ponderados. En un grafo ponderado, cada arista tiene un peso o coste asociado, que puede representar, por ejemplo, la distancia entre dos puntos en un mapa o la fuerza de una conexión en una red social.

Un **grafo** es una estructura de datos que conecta nodos mediante aristas, utilizada para modelar relaciones complejas entre entidades.

Cómo se utilizan en Java

En Java, los grafos pueden ser implementados utilizando varias técnicas, incluyendo matrices de adyacencia, listas de adyacencia, o utilizando estructuras más avanzadas como la biblioteca de colecciones Graph en el paquete java.util o bibliotecas de terceros como **JGraphT**.

- **Lista de adyacencia**: una estructura común en la que cada nodo mantiene una lista de otros nodos a los que está conectado. Es eficiente en términos de espacio para grafos dispersos (con pocas aristas en comparación con el número de nodos).
- **Matriz de adyacencia**: una matriz en la que las filas y columnas representan nodos, y los valores de la matriz indican si existe una arista entre los nodos correspondientes. Es más simple pero menos eficiente en términos de espacio para grafos dispersos.



Tres ejemplos usando grafos con la librería de Java:

Ejemplo 1: implementación básica de un grafo no dirigido con lista de adyacencia

Enunciado: implementa un grafo no dirigido utilizando una lista de adyacencia y agrega métodos para añadir nodos y aristas, y para imprimir el grafo.

Código:

```
import java.util.*;

public class GrafoNoDirigido {
    private Map<String, List<String>> listaAdyacencia;

    public GrafoNoDirigido() {
        this.listaAdyacencia = new HashMap<>();
    }

    public void añadirNodo(String etiqueta) {
        listaAdyacencia.putIfAbsent(etiqueta, new ArrayList<>());
    }

    public void añadirArista(String origen, String destino) {
        listaAdyacencia.get(origen).add(destino);
        listaAdyacencia.get(destino).add(origen);
    }

    public void imprimirGrafo() {
        for (String nodo : listaAdyacencia.keySet()) {
            System.out.print(nodo + " -> ");
            for (String vecino : listaAdyacencia.get(nodo)) {
                System.out.print(vecino + " ");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        GrafoNoDirigido grafo = new GrafoNoDirigido();
```

```
// Añadir nodos
        grafo.añadirNodo("A");
        grafo.añadirNodo("B");
        grafo.añadirNodo("C");
        grafo.añadirNodo("D");

        // Añadir aristas
        grafo.añadirArista("A", "B");
        grafo.añadirArista("A", "C");
        grafo.añadirArista("B", "D");
        grafo.añadirArista("C", "D");

        // Imprimir el grafo
        grafo.imprimirGrafo();
    }
}
```

Explicación: este ejemplo implementa un grafo no dirigido utilizando una lista de adyacencia. Los nodos se representan como claves en un HashMap, y las aristas se almacenan como listas de adyacencia para cada nodo. El método imprimirGrafo muestra las conexiones entre los nodos.

Salida esperada:

```
A -> B C
B -> A D
C -> A D
D -> B C
```


Ejemplo 2: implementación de un grafo dirigido con lista de adyacencia

Enunciado: implementa un grafo dirigido utilizando una lista de adyacencia y agrega métodos para añadir nodos y aristas, y para realizar una búsqueda en profundidad (DFS).

Código:

```
import java.util.*;

public class GrafoDirigido {
    private Map<String, List<String>> listaAdyacencia;

    public GrafoDirigido() {
        this.listaAdyacencia = new HashMap<>();
    }

    public void añadirNodo(String etiqueta) {
        listaAdyacencia.putIfAbsent(etiqueta, new ArrayList<>());
    }
```

```
    public void añadirArista(String origen, String destino) {
        listaAdyacencia.get(origen).add(destino);
    }

    public void dfs(String inicio) {
        Set<String> visitados = new HashSet<>();
        Stack<String> pila = new Stack<>();
        pila.push(inicio);

        while (!pila.isEmpty()) {
            String nodo = pila.pop();
            if (!visitados.contains(nodo)) {
                System.out.print(nodo + " ");
                visitados.add(nodo);
                for (String vecino : listaAdyacencia.get(nodo)) {
                    if (!visitados.contains(vecino)) {
                        pila.push(vecino);
                    }
                }
            }
        }
        System.out.println();
    }

    public static void main(String[] args) {
        GrafoDirigido grafo = new GrafoDirigido();

        // Añadir nodos
        grafo.añadirNodo("A");
        grafo.añadirNodo("B");
        grafo.añadirNodo("C");
        grafo.añadirNodo("D");

        // Añadir aristas
        grafo.añadirArista("A", "B");
        grafo.añadirArista("A", "C");
        grafo.añadirArista("B", "D");
        grafo.añadirArista("C", "D");

        // Realizar DFS desde el nodo A
        System.out.println("Recorrido DFS desde el nodo A:");
        grafo.dfs("A");
    }
}
```

Explicación: este ejemplo implementa un grafo dirigido utilizando una lista de adyacencia. Además de las operaciones básicas de añadir nodos y aristas, se incluye un método para realizar un recorrido en profundidad (DFS) a partir de un nodo específico.

Salida esperada:

```
Recorrido DFS desde el nodo A:
A C D B
```

Ejemplo 3: implementación del algoritmo de Dijkstra para encontrar la ruta más corta en un grafo ponderado utilizando PriorityQueue

```
import java.util.*;

public class GrafoPonderado {
    private Map<String, List<Arista>> listaAdyacencia;

    public GrafoPonderado() {
        this.listaAdyacencia = new HashMap<>();
    }

    public void añadirNodo(String etiqueta) {
        listaAdyacencia.putIfAbsent(etiqueta, new ArrayList<>());
    }

    public void añadirArista(String origen, String destino, int peso) {
        listaAdyacencia.get(origen).add(new Arista(destino, peso));
    }

    public void dijkstra(String inicio) {
        Map<String, Integer> distancias = new HashMap<>();
        PriorityQueue<Arista> colaPrioridad = new PriorityQueue<>(Compara-
tor.comparingInt(arista -> arista.peso));
        Set<String> visitados = new HashSet<>();

        for (String nodo : listaAdyacencia.keySet()) {
            distancias.put(nodo, Integer.MAX_VALUE);
        }
        distancias.put(inicio, 0);
        colaPrioridad.add(new Arista(inicio, 0));

        while (!colaPrioridad.isEmpty()) {
            Arista aristaActual = colaPrioridad.poll();
            String nodoActual = aristaActual.destino;

            if (visitados.contains(nodoActual)) continue;
            visitados.add(nodoActual);

            for (Arista arista : listaAdyacencia.get(nodoActual)) {
                String nodoVecino = arista.destino;
                int nuevaDistancia = distancias.get(nodoActual) + arista.
peso;

                if (nuevaDistancia < distancias.get(nodoVecino)) {
                    distancias.put(nodoVecino, nuevaDistancia);
                    colaPrioridad.add(new Arista(nodoVecino, nuevaDistan-
cia));
                }
            }
        }
```

```
        // Imprimir las distancias más cortas desde el nodo inicial
        for (Map.Entry<String, Integer> entry : distancias.entrySet()) {
            System.out.println("Distancia desde " + inicio + " a " + en-
try.getKey() + " es " + entry.getValue());
        }
    }

    public static void main(String[] args) {
        GrafoPonderado grafo = new GrafoPonderado();

        // Añadir nodos al grafo
        grafo.añadirNodo("A");
        grafo.añadirNodo("B");
        grafo.añadirNodo("C");
        grafo.añadirNodo("D");
        grafo.añadirNodo("E");

        // Añadir aristas con pesos
        grafo.añadirArista("A", "B", 10);
        grafo.añadirArista("A", "C", 15);
        grafo.añadirArista("B", "D", 12);
        grafo.añadirArista("B", "E", 15);
        grafo.añadirArista("C", "E", 10);
        grafo.añadirArista("D", "E", 2);
        grafo.añadirArista("D", "C", 1);

        // Ejecutar Dijkstra desde el nodo A
        grafo.dijkstra("A");
    }
}

class Arista {
    String destino;
    int peso;

    public Arista(String destino, int peso) {
        this.destino = destino;
        this.peso = peso;
    }
}
```


Explicación del código:

1. **GrafoPonderado:**
   - **añadirNodo(String etiqueta):** añade un nodo al grafo.
   - **añadirArista(String origen, String destino, int peso):** añade una arista ponderada entre dos nodos.
   - **dijkstra(String inicio):** implementa el algoritmo de Dijkstra para encontrar las distancias más cortas desde un nodo de inicio a todos los demás nodos del grafo.

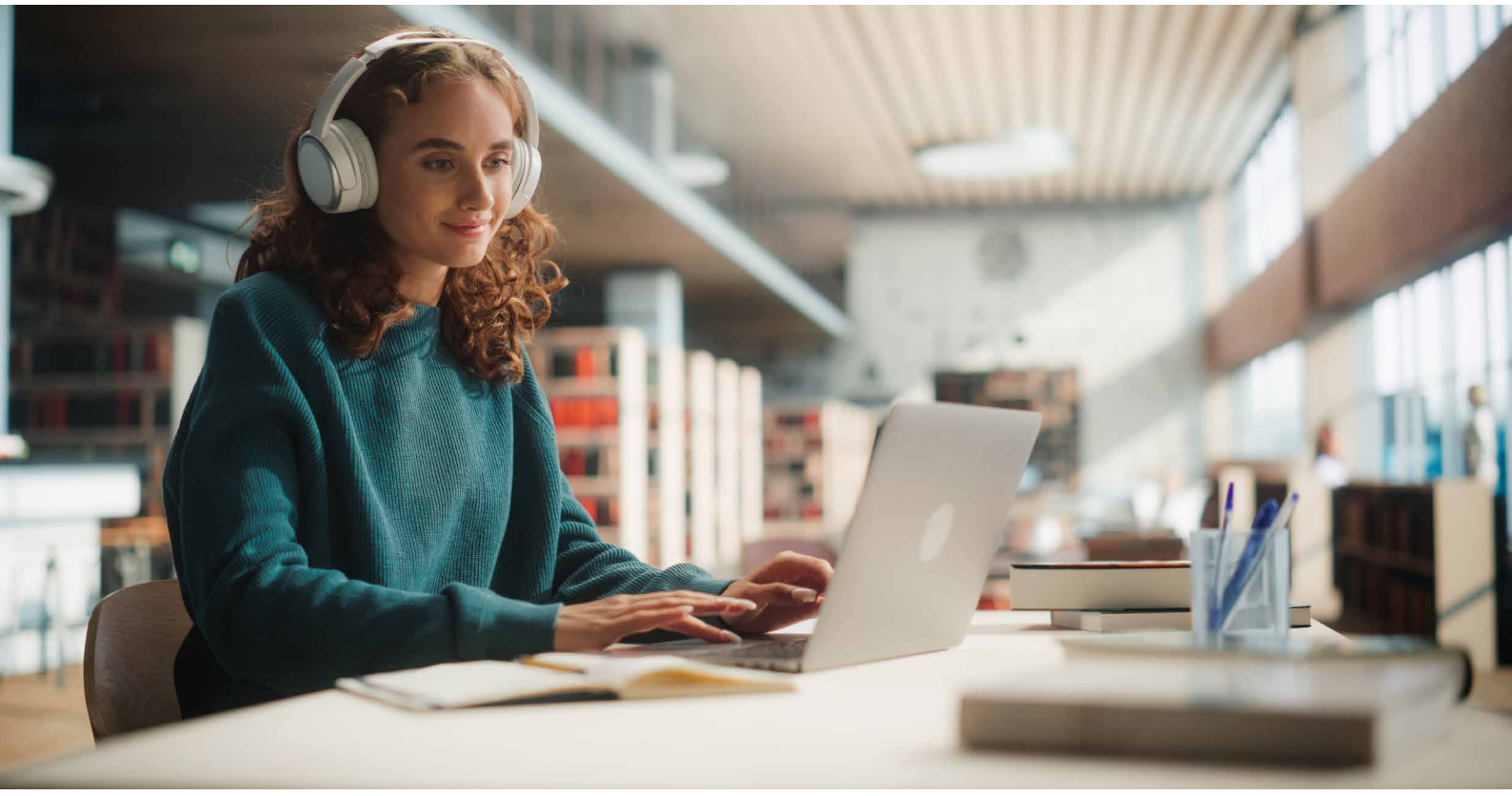
2. **Algoritmo de Dijkstra:**
   - Se inicializan todas las distancias a infinito (Integer.MAX_VALUE) excepto la del nodo inicial, que se establece en 0.
   - Se usa una PriorityQueue para seleccionar el nodo con la distancia mínima no procesado.
   - Para cada nodo, se actualizan las distancias de sus nodos vecinos si se encuentra una ruta más corta a través del nodo actual.
3. **Clase Arista:**
   - Representa una conexión (arista) entre dos nodos en el grafo con un peso asociado.

Ejecución del código:

Cuando ejecutas el código, calculará y mostrará las distancias más cortas desde el nodo "A" a todos los demás nodos en el grafo. La salida sería algo como esto:

```
Distancia desde A a A es 0
Distancia desde A a B es 10
Distancia desde A a C es 14
Distancia desde A a D es 22
Distancia desde A a E es 24
```

Así es como se puede implementar el algoritmo de Dijkstra utilizando una PriorityQueue en Java para resolver el problema de la ruta más corta en un grafo ponderado.



7.7. Otras estructuras de datos

Además de arrays, listas, pilas, colas y grafos, existen otras estructuras de datos que son fundamentales para resolver problemas específicos en programación. Entre ellas, los **diccionarios/HashMaps** y los **árboles** (incluyendo **árboles binarios**) son esenciales para organizar, buscar y manipular datos de manera eficiente.

A continuación, se exploran estas estructuras de datos, cómo funcionan, cómo se utilizan en Java, y se proporciona un ejemplo práctico para cada una.

Diccionario / HashMap

¿Cómo funcionan?

Un **diccionario** (o **HashMap** en Java) es una estructura de datos que almacena pares de clave-valor. Cada valor se asocia a una clave única, que se utiliza para recuperar el valor asociado de manera eficiente. Internamente, un HashMap utiliza una tabla hash para mapear las claves a sus valores, permitiendo operaciones de búsqueda, inserción y eliminación en tiempo promedio constante, O(1).

Las colisiones, cuando dos claves diferentes producen el mismo valor hash, se manejan utilizando técnicas como la encadenamiento (listas enlazadas) o la sondeo lineal.

¿Cómo se utilizan en Java?

En Java, la clase HashMap en el paquete java.util se utiliza para implementar diccionarios. Esta clase permite almacenar y manipular pares clave-valor, donde las claves deben ser únicas.

Operaciones básicas

- **Crear un HashMap:** HashMap<K, V> map = new HashMap<>();
- **Añadir un par clave-valor:** map.put(clave, valor);
- **Recuperar un valor por su clave:** V valor = map.get(clave);
- **Eliminar un par clave-valor:** map.remove(clave);
- **Verificar si una clave existe:** map.containsKey(clave);

Ejemplo usando HashMap:

Enunciado: crea un diccionario que almacene nombres de personas y sus edades, y luego busca la edad de una persona específica.

Código:

```
import java.util.HashMap;

public class EjemploHashMap {
    public static void main(String[] args) {
        // Crear un HashMap para almacenar nombres y edades
        HashMap<String, Integer> diccionario = new HashMap<>();

        // Añadir pares clave-valor al HashMap
        diccionario.put("Juan", 30);
        diccionario.put("Ana", 25);
        diccionario.put("Luis", 40);

        // Buscar la edad de "Ana"
        String nombre = "Ana";
        if (diccionario.containsKey(nombre)) {
            System.out.println("La edad de " + nombre + " es: " + diccio-
nario.get(nombre));
        } else {
            System.out.println(nombre + " no se encuentra en el diccionario.");
        }
    }
}
```

Explicación: este ejemplo muestra cómo usar HashMap para almacenar pares de nombres y edades. Luego, busca y muestra la edad de una persona específica utilizando la clave (nombre).

Árboles

Un **árbol** es una estructura de datos jerárquica que consiste en nodos conectados por aristas. El nodo superior se llama **raíz**, y cada nodo puede tener cero o más nodos hijos. Un árbol sin ciclos donde cada nodo tiene a lo sumo un padre se llama un árbol general. Los árboles se utilizan en múltiples aplicaciones como la representación de estructuras de jerarquía (como archivos en un sistema), manipulación de expresiones matemáticas, y búsquedas eficientes.

En un árbol, los nodos pueden ser visitados en diferentes órdenes, como **preorden**, **inorden**, y **postorden**.

¿Cómo se utilizan en Java?

En Java, los árboles se pueden implementar utilizando clases que definen los nodos y las relaciones entre ellos. Aunque no hay una implementación directa de árboles genéricos en la biblioteca estándar de Java, se pueden construir fácilmente utilizando nodos y enlaces manuales.

Ejemplo usando un árbol:

Enunciado: implementa un árbol simple donde cada nodo tiene un valor de tipo entero y dos hijos, y realiza un recorrido en preorden.

Código:

```
class Nodo {
    int valor;
    Nodo izquierdo, derecho;

    public Nodo(int valor) {
        this.valor = valor;
        izquierdo = derecho = null;
    }
}

class Arbol {
    Nodo raiz;

    public Arbol(int valor) {
        raiz = new Nodo(valor);
    }

    public void preorden(Nodo nodo) {
        if (nodo != null) {
            System.out.print(nodo.valor + " ");
            preorden(nodo.izquierdo);
            preorden(nodo.derecho);
        }
    }
}

public class EjemploArbol {
    public static void main(String[] args) {
        Arbol arbol = new Arbol(1);
        arbol.raiz.izquierdo = new Nodo(2);
        arbol.raiz.derecho = new Nodo(3);
        arbol.raiz.izquierdo.izquierdo = new Nodo(4);
        arbol.raiz.izquierdo.derecho = new Nodo(5);

        System.out.println("Recorrido en preorden del árbol:");
        arbol.preorden(arbol.raiz);
    }
}
```

Explicación: este ejemplo muestra cómo implementar un árbol simple con nodos enteros. Luego, se realiza un recorrido en preorden para visitar cada nodo, imprimiendo sus valores.

Árboles binarios

Un **árbol binario** es un tipo específico de árbol en el que cada nodo tiene como máximo dos hijos, denominados hijo izquierdo e hijo derecho. Los árboles binarios son fundamentales en la informática, ya que sirven de base para estructuras de datos más avanzadas como los **árboles binarios de búsqueda** y los **árboles AVL**.

- **Árbol binario de búsqueda (BST)**: es un árbol binario en el que para cada nodo, todos los valores en el subárbol izquierdo son menores que el valor del nodo, y todos los valores en el subárbol derecho son mayores.

¿Cómo se utilizan en Java?

En Java, los árboles binarios, incluidos los árboles binarios de búsqueda, se pueden implementar utilizando nodos que tienen referencias a sus hijos izquierdo y derecho. Java no proporciona una clase Tree en la biblioteca estándar, por lo que los árboles binarios se implementan manualmente.

Ejemplo usando un árbol binario de búsqueda (BST):

Enunciado: implementa un árbol binario de búsqueda que permita insertar nodos y buscar un valor específico.

Código:

```
class NodoBST {
    int valor;
    NodoBST izquierdo, derecho;

    public NodoBST(int valor) {
        this.valor = valor;
        izquierdo = derecho = null;
    }
}

class ArbolBinarioBusqueda {
    NodoBST raiz;

    public void insertar(int valor) {
        raiz = insertarRecursivo(raiz, valor);
    }

    private NodoBST insertarRecursivo(NodoBST raiz, int valor) {
        if (raiz == null) {
            raiz = new NodoBST(valor);
            return raiz;
        if (valor < raiz.valor) {
            raiz.izquierdo = insertarRecursivo(raiz.izquierdo, valor);
```

```
        } else if (valor > raiz.valor) {
            raiz.derecho = insertarRecursivo(raiz.derecho, valor);
        }

        return raiz;
    }

    public boolean buscar(int valor) {
        return buscarRecursivo(raiz, valor);
    }

    private boolean buscarRecursivo(NodoBST raiz, int valor) {
        if (raiz == null) {
            return false;
        }
        if (valor == raiz.valor) {
            return true;
        }
        return valor < raiz.valor ? buscarRecursivo(raiz.izquierdo,
                        valor) : buscarRecursivo(raiz.derecho, valor);
    }
}

public class EjemploArbolBinario {
    public static void main(String[] args) {
        ArbolBinarioBusqueda bst = new ArbolBinarioBusqueda();

        // Insertar valores en el BST
        bst.insertar(50);
        bst.insertar(30);
        bst.insertar(70);
        bst.insertar(20);
        bst.insertar(40);
        bst.insertar(60);
        bst.insertar(80);

        // Buscar un valor en el BST
        int valorBuscado = 40;
        if (bst.buscar(valorBuscado)) {
            System.out.println("El valor " + valorBuscado + " se
                                encuentra en el BST.");
        } else {
            System.out.println("El valor " + valorBuscado + " no se
                                encuentra en el BST.");
        }
    }
}
```

Explicación: este ejemplo implementa un árbol binario de búsqueda (BST) que permite insertar valores y buscar un valor específico. El árbol asegura que los valores menores se encuentren en el subárbol izquierdo y los mayores en el subárbol derecho.

Los **diccionarios/HashMaps**, **árboles** y **árboles binarios** son estructuras de datos poderosas que permiten manejar datos complejos y realizar operaciones eficientes en muchas aplicaciones de programación. La capacidad de organizar, buscar y manipular datos utilizando estas estructuras es fundamental para el desarrollo de software avanzado.

8 PROGRAMACIÓN Y BASES DE DATOS

Las **bases de datos** son un componente esencial en el desarrollo de aplicaciones modernas, ya que permiten almacenar, gestionar y recuperar grandes cantidades de información de manera eficiente. Integrar bases de datos en la programación permite a los desarrolladores construir aplicaciones más dinámicas y poderosas, que pueden gestionar datos en tiempo real, como aplicaciones web, sistemas empresariales y aplicaciones móviles. La capacidad de conectarse y trabajar con bases de datos es una habilidad fundamental para cualquier programador, especialmente en entornos de desarrollo que manejan grandes volúmenes de datos.

En este tema, aprenderemos cómo interactuar con bases de datos desde un programa, explorando el proceso de conexión, consultas, actualizaciones y el uso de datos persistentes.

Las **bases de datos** permiten almacenar y gestionar información de manera eficiente, y conectarlas con un programa es esencial para construir aplicaciones dinámicas y potentes.

Proceso de conexión a una base de datos y su importancia

El proceso de conectarse a una base de datos desde un programa es esencial para interactuar con los datos que esta contiene. A continuación, se describe de manera general cómo funciona este proceso:

1. **Seleccionar el tipo de base de datos:**

   Antes de conectarse a una base de datos, es necesario elegir el tipo de base de datos adecuado para el proyecto. Los tipos más comunes son bases de datos relacionales (como MySQL, PostgreSQL o SQLite) y bases de datos no relacionales (como MongoDB). Cada uno tiene sus propias características y se utiliza para diferentes tipos de aplicaciones.

2. **Cargar el controlador (driver) adecuado:**

   Las bases de datos relacionales, como MySQL o PostgreSQL, requieren un **driver JDBC** (Java Database Connectivity) para conectarse desde un programa Java. Este controlador actúa como un puente entre la aplicación y la base de datos, permitiendo que se comuniquen.

3. **Establecer la conexión:**

   El siguiente paso es **establecer la conexión** con la base de datos. Para ello, el programa debe proporcionar una URL de conexión (que incluye el tipo de base de datos, dirección del servidor y nombre de la base de datos), así como las credenciales (usuario y contraseña) necesarias para autenticar el acceso.

4. **Enviar consultas SQL:**

   Una vez establecida la conexión, el programa puede interactuar con la base de datos a través de **consultas SQL**. Estas consultas pueden ser de varios tipos: SELECT para leer datos, INSERT para agregar datos, UPDATE para modificar datos existentes y DELETE para eliminarlos.

5. **Procesar los resultados:**

   Las consultas SQL, especialmente las de lectura, generan resultados que el programa debe procesar. Estos resultados se manejan en un formato estructurado (generalmente como tablas) que puede ser recorrido y manipulado desde el código.

6. **Cerrar la conexión:**

   Una vez que se han realizado todas las operaciones necesarias, es crucial **cerrar la conexión** con la base de datos para liberar los recursos del sistema. Mantener conexiones abiertas innecesariamente puede afectar negativamente al rendimiento y seguridad del sistema.

Importancia del proceso

La conexión a una base de datos es un proceso fundamental que permite a las aplicaciones ser dinámicas, escalables y centradas en los datos. Las bases de datos ofrecen un entorno seguro para almacenar información crucial, como datos de usuarios, productos, transacciones y más. Sin esta capacidad de conectarse y manipular bases de datos, las aplicaciones no podrían gestionar ni procesar información de manera eficiente.

Además, establecer conexiones de manera segura y eficiente garantiza que la aplicación funcione de manera correcta, evite errores de datos y proteja la integridad y confidencialidad de la información, especialmente en aplicaciones que manejan datos sensibles o grandes volúmenes de transacciones.

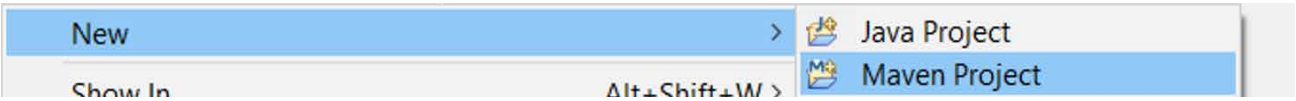
Conectar un programa a una base de datos es una habilidad esencial para los desarrolladores, ya que permite a las aplicaciones ser interactivas y centradas en los datos. Este proceso implica establecer una conexión

segura, enviar consultas SQL, procesar los resultados y cerrar adecuadamente la conexión, todo lo cual es crucial para el buen funcionamiento y la eficiencia de las aplicaciones modernas.

8.1. Conexión a SQLite

Crear proyecto maven en Eclipse

Vamos a crear un nuevo proyecto en Eclipse, en concreto un "Maven Project".

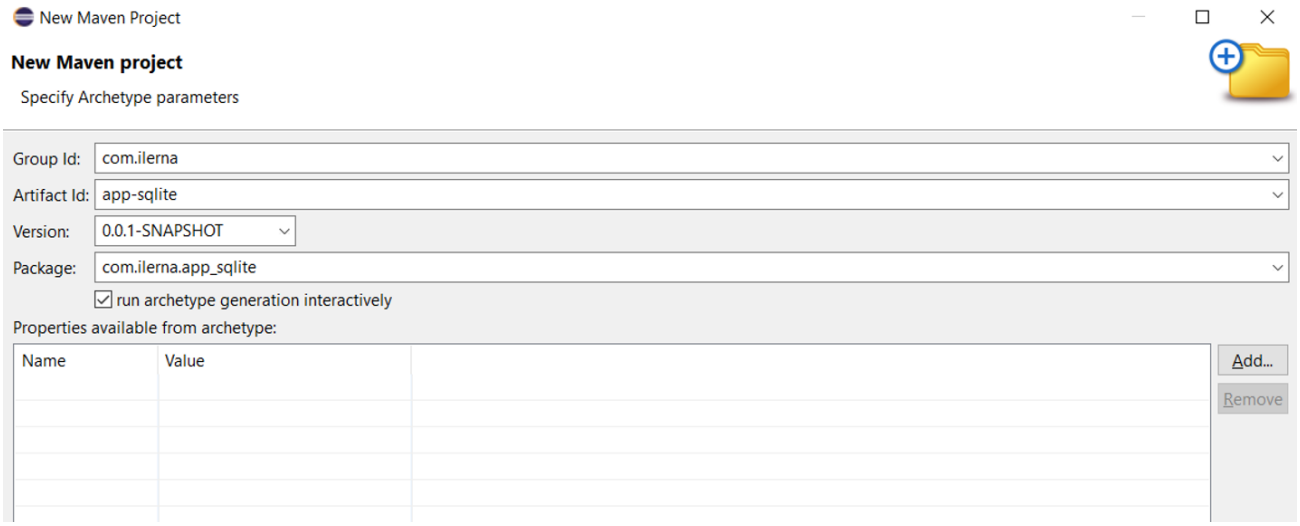


Maven es una herramienta de gestión y comprensión de proyectos que proporciona una forma estandarizada de construir, gestionar dependencias y empaquetar proyectos Java. Un proyecto Maven es un proyecto que sigue las convenciones y utiliza las características proporcionadas por Apache Maven.

De entre sus características destaca que Maven facilita la gestión de bibliotecas y dependencias externas. Al especificar las dependencias en el archivo pom.xml, Maven las descarga automáticamente del repositorio central de Maven o de otros repositorios especificados.

Este tipo de gestión de dependencias es muy importante ya que evita que los programadores estemos pendientes de descargar de forma manual las dependencias de nuestros proyectos.

Los arquetipos son plantillas para la creación de proyectos y, en este caso, hecemos clic en a Create *a simple project (skip archetype)*.



Proporciona los valores necesarios:

- Group ID: un identificador único para tu grupo o empresa (por ejemplo, com.example).
- Artifact ID: el identificador único para el proyecto (por ejemplo, my-app).
- Version: la versión del proyecto (por ejemplo, 1.0-SNAPSHOT).
- Package: el paquete base para el proyecto (opcional).

Qué es JDBC

Para que realmente podamos utilizar SQLite dentro de Java necesitamos utilizar el driver JDBC de SQLite. JDBC (Java Database Connectivity) es una API de Java que define cómo un cliente puede acceder a una base de datos. Proporciona métodos para consultar y actualizar datos en una base de datos de manera independiente del sistema de gestión de bases de datos (DBMS) que se esté utilizando. JDBC es parte de la plataforma Java Standard Edition, desde Java 2 (JDK 1.1).

Componentes clave de JDBC

1. **DriverManager:**
   - Gestiona una lista de controladores de bases de datos.
   - Se utiliza para establecer una conexión a la base de datos seleccionando el controlador adecuado.
2. **Connection:**
   - Representa una conexión a una base de datos específica.
   - Se utiliza para crear Statement, PreparedStatement y CallableStatement objetos, así como para gestionar transacciones.
3. **Statement:**
   - Utilizado para ejecutar consultas SQL estáticas.
   - Puede ejecutar consultas SQL y devolver los resultados a través de un ResultSet.
4. **PreparedStatement:**
   - Una subclase de Statement que permite ejecutar consultas SQL precompiladas.
   - Utiliza parámetros para aumentar la eficiencia y la seguridad contra ataques de inyección SQL.

5. **CallableStatement:**
   - Utilizado para ejecutar procedimientos almacenados en la base de datos.
   - Puede aceptar parámetros de entrada y devolver parámetros de salida.
6. **ResultSet:**
   - Representa un conjunto de resultados devuelto por una consulta SELECT.
   - Permite navegar a través de los datos fila por fila y acceder a los valores de las columnas por nombre o índice.

Funcionamiento de JDBC

1. **Carga del driver:**
   - El controlador JDBC específico para la base de datos debe cargarse antes de establecer una conexión.
   - Esto se hace generalmente con la clase Class.forName("driverClassName").
2. **Establecer la conexión:**
   - Utilizando DriverManager.getConnection(url, user, password), donde url es la URL de la base de datos.
3. **Crear una sentencia:**
   - Una vez que se establece la conexión, se puede crear un Statement o PreparedStatement para ejecutar consultas SQL.
4. **Ejecutar consultas:**
   - Utilizar métodos como executeQuery para consultas SELECT que devuelven un ResultSet, o executeUpdate para INSERT, UPDATE o DELETE que devuelven el número de filas afectadas.
5. **Procesar los resultados:**
   - Navegar a través del ResultSet para obtener los datos de la consulta.
6. **Cerrar recursos:**
   - Es importante cerrar el ResultSet, Statement y Connection después de su uso para liberar recursos.

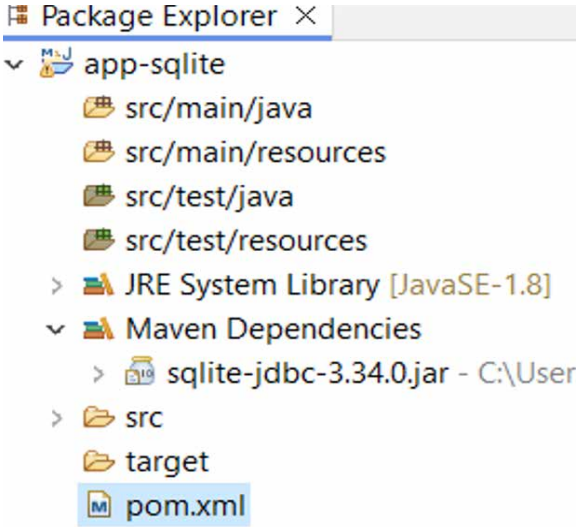
Ejemplo básico de JDBC:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCDemo {
    public static void main(String[] args) {
        Connection connection = null;
        Statement statement = null;
        ResultSet resultSet = null;

        try {
            // Cargar el driver JDBC
            Class.forName("org.sqlite.JDBC");

            // Establecer la conexión a la base de datos
            connection = DriverManager.getConnection("jdbc:sqlite:sample.
db");

            // Crear una declaración
            statement = connection.createStatement();

            // Ejecutar una consulta
            resultSet = statement.executeQuery("SELECT * FROM users");

            // Procesar el conjunto de resultados
            while (resultSet.next()) {
                System.out.println("ID: " + resultSet.getInt("id"));
                System.out.println("Name: " + resultSet.getString("na-
me"));
            }
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        } finally { // Cerrar recursos
            try {
                if (resultSet != null) resultSet.close();
                if (statement != null) statement.close();
                if (connection != null) connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Para poder utilizar el JDBC de SQLite lo tenemos que descargar desde las dependencias de Maven, para ello debemos escribir las líneas de dependencias en el archivo **pom.xml**.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ilerna</groupId>
  <artifactId>app-sqlite</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>AppSQLite</name>

  <dependencies>
        <!-- Dependencia JDBC de SQLite -->
        <dependency>
            <groupId>org.xerial</groupId>
            <artifactId>sqlite-jdbc</artifactId>
            <version>3.34.0</version>
        </dependency>
        <!-- Otras dependencias -->
    </dependencies>

</project>
```

Una vez escrito nos descarga el .JAR en nuestro proyecto y actualizando el POM tendremos la última versión de la librería.



Instalar SQLite

Para instalar SQLite en Windows, puedes seguir estos pasos detallados. SQLite no requiere una instalación complicada, ya que consiste en un único archivo ejecutable que puedes descargar y colocar en una ubicación accesible.

link.ilerna.es/ql5r

Para descargar SQLite abre tu navegador web y ve a la página oficial de descargas de SQLite: https://www.sqlite.org/download.html

Localiza el archivo ZIP descargado (sqlite-tools-win64-x86-XXXXXXX.zip). Haz clic derecho sobre el archivo y selecciona *Extraer todo...*

Selecciona una ubicación donde deseas extraer los archivos. Por ejemplo, puedes extraerlo en C:\sqlite.

Para poder ejecutar SQLite desde cualquier ubicación en la línea de comandos, necesitas agregar el directorio que contiene sqlite3.exe a la variable de entorno PATH.

1. **Abrir la configuración de variables de entorno:**
   - Haz clic derecho en *Este PC* o *Mi PC* en el escritorio o en el Explorador de archivos.
   - Selecciona *Propiedades*.
   - Haz clic en *Configuración avanzada del sistema*.
   - En la pestaña *Opciones avanzadas*, haz clic en *Variables de entorno*.
2. **Editar la variable de entorno PATH:**
   - En la sección *Variables del sistema*, localiza y selecciona la variable Path, luego haz clic en *Editar...*
   - Haz clic en *Nuevo* y añade la ruta al directorio donde extrajiste sqlite3.exe. Por ejemplo, C:\sqlite.
3. **Guardar los cambios:**
   - Haz clic en *Aceptar* en todas las ventanas para aplicar los cambios.

Crear BBDD en SQLite

Ahora que tienes SQLite instalado, puedes crear y gestionar bases de datos directamente desde la línea de comandos:

1. **Crear una nueva base de datos:**
   - En el símbolo del sistema, navega hasta el directorio donde deseas crear la base de datos.
   - Escribe sqlite3 mydatabase.db para crear una nueva base de datos llamada mydatabase.db.
2. **Ejecutar comandos SQL:**
   - Una vez que la base de datos esté abierta, puedes ejecutar comandos SQL directamente en la consola de SQLite.

```
sqlite> CREATE TABLE users (id INTEGER PRIMARY KEY,
name TEXT NOT NULL);
sqlite> INSERT INTO users (name) VALUES ('John Doe');
sqlite> SELECT * FROM users;
```

Añadir la BBDD al proyecto Maven

En un proyecto Maven en Eclipse, puedes añadir tu base de datos SQLite de manera que sea accesible durante la ejecución de tu aplicación.

Incluir la base de datos en el directorio de recursos

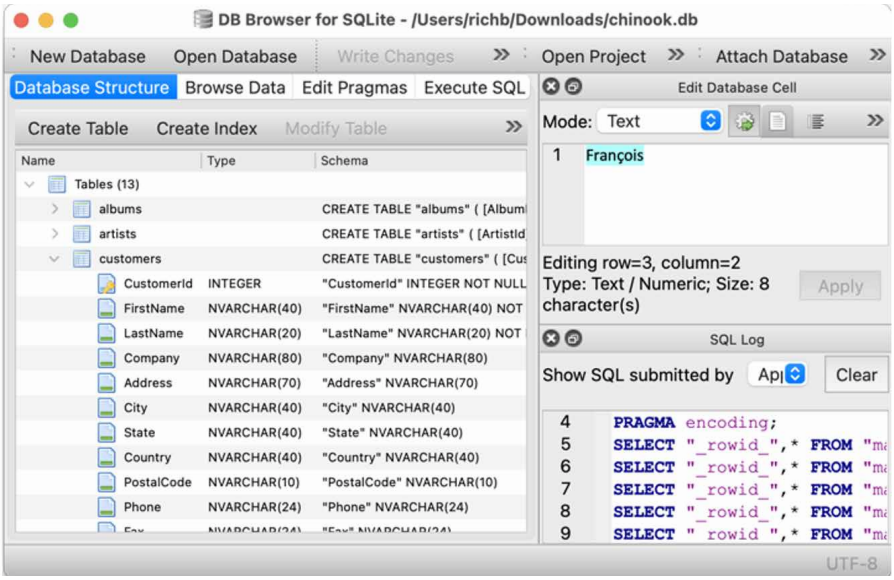
Esta opción es útil si quieres incluir la base de datos como parte del proyecto y asegurar que se empaqueta junto con tu aplicación.

1. **Crear una carpeta de recursos:**
   - En el Package Explorer, haz clic derecho en el directorio src/main/resources y selecciona *New -> Folder*.
   - Nombra la nueva carpeta como db (o cualquier otro nombre que prefieras).
2. **Añadir la base de datos SQLite:**
   - Copia tu archivo de base de datos SQLite (por ejemplo, mydatabase.db) en la carpeta src/main/resources/db.

link.ilerna.es/sl6l

DB Browser para gestionar SQLite

Una vez instalado SQLIte en nuestro equipo podemos acceder al contenido de nuestra base de datos con este gestor de base de datos para SQLite llamado DB Browser: https://sqlitebrowser.org/



Este gestor nos va a permitir acceder a la información de la base de datos de una manera visual y nos permitirá hacer las operaciones básicas más SQL.

Usar SQLite en Java

El último paso que nos queda es utilizar SQLite en Java para poder trabajar con la base de datos que creamos anteriormente. Para ello vamos a utilizar el siguiente código en una clase que crearemos en src/main llamada Principal.java.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.net.URL;

public class Principal {
    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        try {
            // Cargar el driver JDBC de SQLite
            Class.forName("org.sqlite.JDBC");
```

```
            // Obtener la ruta de la base de datos en
los recursos
            URL dbUrl = Principal.class.getResour-
ce("/db/mydatabase.db");
            String url = "jdbc:sqlite:" + dbUrl.get-
Path();

            // Establecer la conexión a la base de
datos
            conn = DriverManager.getConnection(url);

            // Crear una declaración
            stmt = conn.createStatement();
            String sql = "CREATE TABLE IF NOT EXISTS
users (id INTEGER PRIMARY
                          KEY, name TEXT NOT NULL)";
            stmt.executeUpdate(sql);
            System.out.println("Tabla 'users' creada
o ya existe.");

            // Insertar datos
            sql = "INSERT INTO users (name) VALUES
('Bloody Mary')";
            stmt.executeUpdate(sql);
            System.out.println("Datos insertados en
la tabla 'users'.");

            // Consultar datos
            sql = "SELECT * FROM users";
            ResultSet rs = stmt.executeQuery(sql);
            while (rs.next()) {
                System.out.println("ID: " + rs.getIn-
t("id") + ", Name: " +
                                        rs.getString("na-
me"));
            }
            rs.close();
        } catch (ClassNotFoundException | SQLExcep-
tion e) {
            e.printStackTrace();
        } finally {
            try {
                if (stmt != null) stmt.close();
                if (conn != null) conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

8.2. Conexión a MySQL

En esta segunda base de datos vamos a trabajar con MySQL, para ello, seguiremos los siguientes pasos para poder trabajar con esta base de datos.

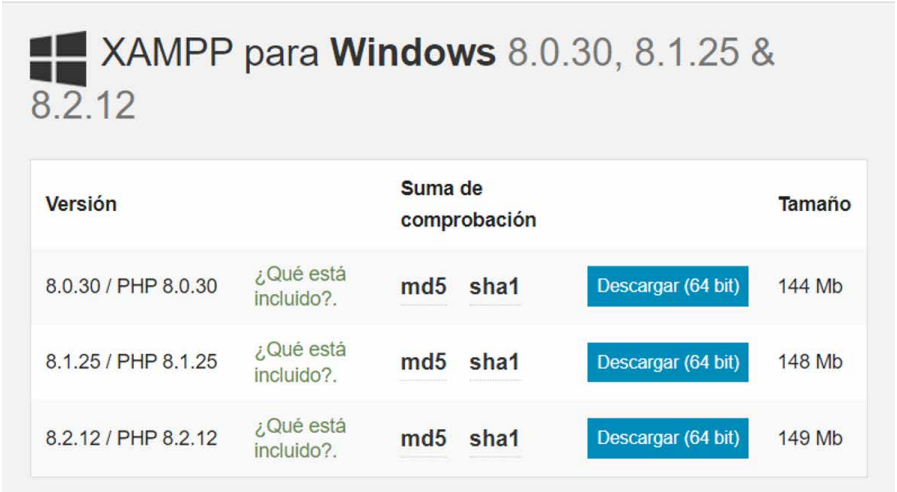
Instalar XAMPP

Para poder utilizar MySQL de una forma sencilla en nuestro equipo vamos a utilizar XAMPP, que incorpora una serie de servidores y servicios entre los que se encuentran un servidor Web y MySQL como base de datos.

link.ilerna.es/yzqj

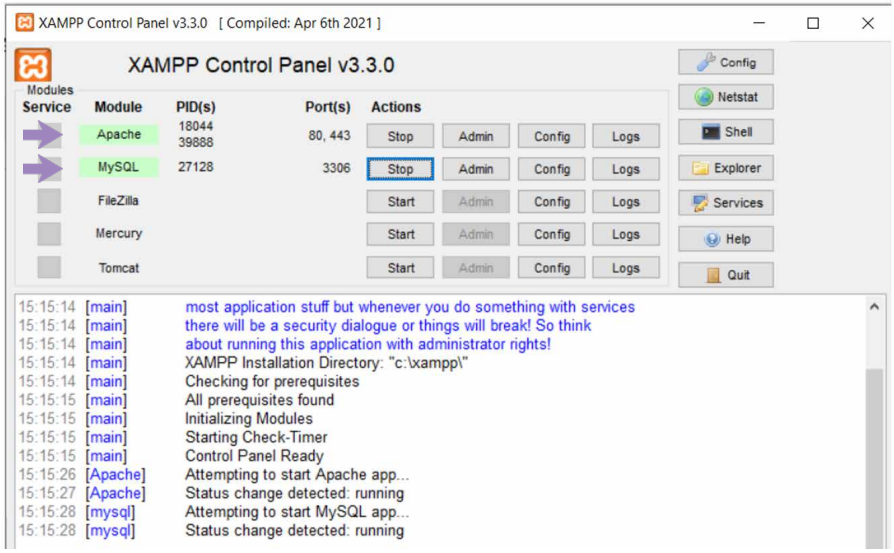
Para instalar XAMPP vamos a ir a la web: https://www.apachefriends.org/es/ y elegimos en el menú la opción Descargar.

Trataremos de instalar la última versión para nuestro SO para así poder utilizarlo.



Arrancar servidores

Una vez instalado XAMPP iniciaremos los servidores Apache y MySQL para tener activa nuestra base de datos.



Acceso al SGBD de MySQL

Realmente, Apache (servidor web) no es necesario para que funcione MySQL, pero nos permite acceder al SGBD (Sistema Gestor de Bases de Datos) de MySQL, que se llama phpMyAdmin. Para acceder a este gestor debemos entrar en la web: https://localhost

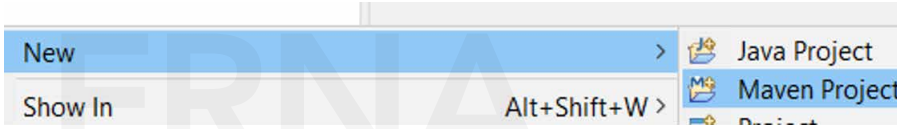
Localhost se refiere al propio equipo (ip 127.0.0.1) que nos permite acceder al servidor Apache instalado en XAMPP, una vez dentro de esta página elegimos la opción del menú phpMyAdmin.

Elegida esta opción ya accederemos al SGBD de MySQL que trae XAMPP.

Si nos fijamos, a la izquierda tenemos las diferentes bases de datos, y en esta página podremos llevar a cabo las diferentes opciones de trabajo de una base de datos.

Crear Proyecto Maven

Vamos a crear un nuevo proyecto en Eclipse, en concreto un "Maven Project".



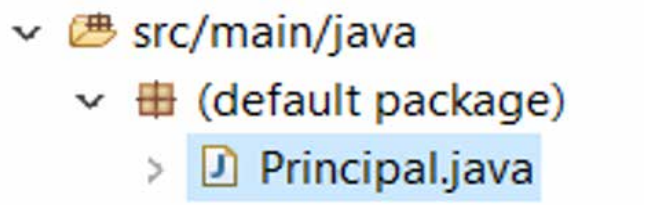
Maven es una herramienta de gestión y comprensión de proyectos que proporciona una forma estandarizada de construir, gestionar dependencias y empaquetar proyectos Java. Un proyecto Maven es un proyecto que sigue las convenciones y utiliza las características proporcionadas por Apache Maven.

Usar MySQL en Java

Después de crear el proyecto en Eclipse con Maven empezamos añadiendo la dependencia de MySQL para usar su driver JDBC en el archivo **pom.xml**.

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.28</version>
    </dependency>
</dependencies>
```

Luego creamos la clase Principal en src/main/java



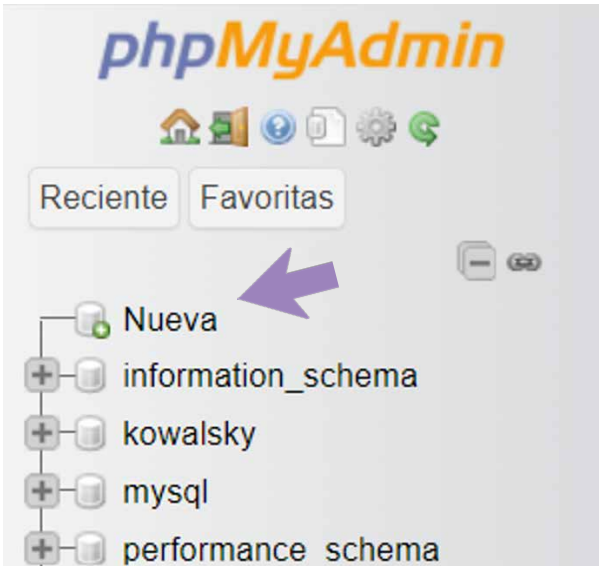
En la que escribiremos el siguiente código:

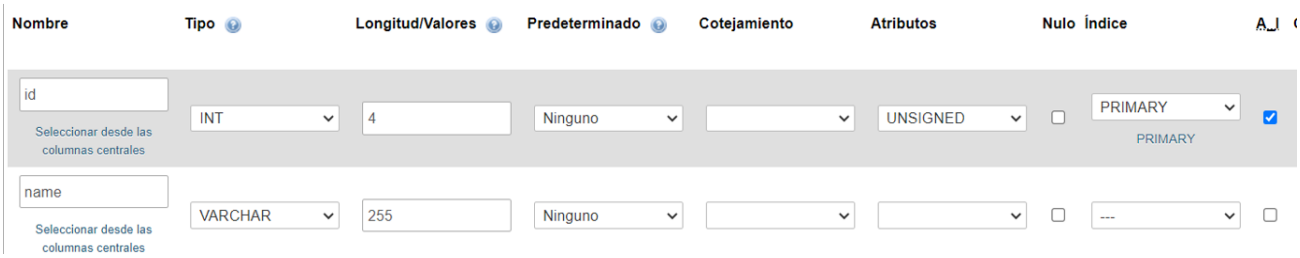
```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Principal {
    public static void main(String[] args) {
          Connection conn = null;
            // URL de conexión a tu base de datos
          String url = "jdbc:mysql://localhost:3306/
nombre_basedatos";          String user = "tu_usua-
rio";
          String password = "tu_contraseña";

          try {
                conn = DriverManager.getConnec-
tion(url, user, password);
                if (conn != null) {
                      System.out.println("Conexión
exitosa a la base de datos MySQL!");
                }
          } catch (SQLException e) {
                System.out.println("Error al conectar
a la base de datos MySQL:");
                e.printStackTrace();
          } finally {
                try {
                      if (conn != null) {
                            conn.close();
                      }
                } catch (SQLException ex) {
                      ex.printStackTrace();
                }
          }
    }
}
```

Vamos a fijarnos en estas líneas:

```
        Connection conn = null;
        // URL de conexión a tu base de datos
        String url = "jdbc:mysql://localhost:3306/
nombre_basedatos";            String user = "tu_usua-
rio";
      String password = "tu_contraseña";
```

Para trabajar con MySQL necesitamos conectarnos por URL a la base de datos por lo que debemos tener una base de datos creada. Para ello volvemos a phpMyAdmin y creamos la base de datos “users”.





Una vez creada la base de datos, la tabla y tras meter algún registro, vamos a cambiar los datos de acceso y la URL.

```
        Connection conn = null;
        // URL de conexión a tu base de datos
        String url = “jdbc:mysql://localhost:3306/
users”;
        String user = “root”;
      String password = “”;
```

El usuario *root* (raíz) de MySQL por defecto no tiene contraseña por lo que podemos dejar los accesos de esta forma.

Ahora vamos a modificar la clase principal para que funcione de igual forma que usamos en SQLite y así trabajar con la base de datos MySQL.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Principal {
    public static void main(String[] args) {
          Connection conn = null;
          Statement stmt = null;
          String url = "jdbc:mysql://localhost:3306/
users";
          String user = "root";
          String password = "";

          try {
                conn = DriverManager.getConnec-
tion(url, user, password);
                stmt = conn.createStatement();
                String sql = "CREATE TABLE IF NOT
EXISTS users (id INTEGER PRIMARY
                                  KEY, name TEXT NOT
NULL)";
                stmt.executeUpdate(sql);
                System.out.println("Tabla 'users'
creada o ya existe.");
                // Insertar datos
                sql = "INSERT INTO users (name) VA-
LUES ('Bloody Mary')";
                stmt.executeUpdate(sql);
                System.out.println("Datos insertados
en la tabla 'users'.");
                // Consultar datos
                sql = "SELECT * FROM users";
                ResultSet rs = stmt.executeQuery(s-
ql);
                while (rs.next()) {
                      System.out.println("ID: " + rs.
getInt("id") + ", Name: " +
                                              rs.getS-
tring("name"));
                }
                rs.close();
          } catch (SQLException e) {
                System.err.println("Error al conectar
```

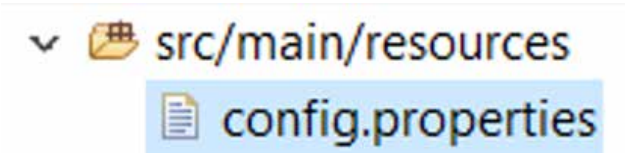
```
a la base de datos MySQL:");
                e.printStackTrace(System.err);
          } finally {
                try {
                      if (conn != null)
                            conn.close();
                      if (stmt != null)
                            stmt.close();
                } catch (SQLException ex) {
                      System.err.println("Error al
cerrar la conexión con MySQL");
                      ex.printStackTrace(System.err);
                }
          }
    }
}
```

Si vamos ahora a phpMyAdmin podremos ver los cambios que se produjeron en la BBDD de MySQL.

Tras esto, vamos a repetir el ejercicio de hacer un menú en modo consola pero esta vez trabajando con MySQL.

Archivo propiedades

Vamos a mejorar la forma en la que nos conectamos a MySQL y vamos a crear un archivo llamado **config.properties** para indicar propiedades en src/main/resources.



Indicamos los siguientes datos:

```
db.url=jdbc:mysql://localhost:3306/users
db.username=root
db.password=
```

Y ahora modificaremos el código para acceder a estas propiedades:

```
Properties props = new Properties();
        try (InputStream input = Principal.class.
getClassLoader().getResourceAsStream("config.proper-
ties")) {
            props.load(input);
        } catch (IOException e) {
            e.printStackTrace();
        }

        String url = props.getProperty("db.url");
        String user = props.getProperty("db.userna-
me");
        String password = props.getProperty("db.
password");
```

link.ilerna.es/MzwA

8.3. Docker

Para conectarnos a las bases de datos vamos a utilizar Docker Desktop. Para ello vamos al enlace: https://www.docker.com/products/docker-desktop/ y descargamos la versión para nuestro sistema operativo.

Docker es una plataforma de software que permite crear, probar y desplegar aplicaciones rápidamente. Utiliza contenedores, que son entornos ligeros, portátiles y autosuficientes que incluyen todo lo necesario para ejecutar una aplicación: código, runtime, herramientas del sistema, bibliotecas y configuraciones. Los contenedores aseguran que la aplicación se ejecute de manera consistente sin importar el entorno en el que se despliegue, ya sea en una máquina local, un servidor o en la nube.

Beneficios de Docker

- **Portabilidad:** los contenedores Docker se pueden ejecutar en cualquier sistema que tenga Docker instalado, independientemente de las diferencias en el sistema operativo subyacente.
- **Consistencia:** garantiza que el software funcione de la misma manera en diferentes entornos, evitando problemas de "funciona en mi máquina".
- **Eficiencia:** los contenedores son más ligeros que las máquinas virtuales tradicionales, permitiendo ejecutar más contenedores en el mismo hardware.
- **Aislamiento:** los contenedores proporcionan un entorno aislado para las aplicaciones, lo que mejora la seguridad y facilita la gestión de dependencias.

¿Qué es Docker Desktop?

Docker Desktop es una aplicación que proporciona una manera fácil de instalar y usar Docker en sistemas operativos Windows y macOS. Incluye Docker Engine, Docker CLI, Docker Compose y otras herramientas necesarias para trabajar con contenedores.

Funcionalidades de Docker Desktop

- **Docker Engine:** el motor que ejecuta y gestiona los contenedores Docker.
- **Docker CLI:** una interfaz de línea de comandos para interactuar con Docker.
- **Docker Compose:** una herramienta para definir y ejecutar aplicaciones multicontenedor. Utiliza un archivo YAML para configurar los servicios de la aplicación.
- **Kubernetes:** Docker Desktop incluye Kubernetes, permitiendo a los desarrolladores ejecutar y gestionar clústeres de Kubernetes localmente.
- **Interfaz gráfica:** proporciona una interfaz de usuario para gestionar contenedores, imágenes y volúmenes, facilitando la visualización y la administración de los recursos Docker.
- **Integración con el sistema operativo:** Docker Desktop se integra con el sistema operativo, permitiendo el uso de volúmenes compartidos y la red local.

Beneficios de Docker Desktop

- **Facilidad de instalación y configuración:** simplifica la instalación y configuración de Docker en entornos Windows y macOS.
- **Desarrollo local eficiente:** permite a los desarrolladores construir, probar y depurar aplicaciones en contenedores localmente antes de desplegarlas en producción.
- **Entornos de desarrollo consistentes:** asegura que todos los desarrolladores trabajen en un entorno de desarrollo idéntico, minimizando los problemas de configuración y dependencias.
- **Gestión de contenedores:** proporciona herramientas para gestionar contenedores y clústeres de Kubernetes directamente desde el escritorio.

Usos comunes de Docker y Docker Desktop

- **Desarrollo de software:** crear entornos de desarrollo replicables y consistentes.
- **Pruebas automatizadas:** ejecutar pruebas de integración y de extremo a extremo en contenedores aislados.
- **Despliegue de aplicaciones:** desplegar aplicaciones de manera rápida y segura en diferentes entornos (desarrollo, pruebas, producción).
- **Microservicios:** desarrollar y desplegar aplicaciones basadas en microservicios, donde cada microservicio se ejecuta en su propio contenedor.
- **Modernización de aplicaciones legadas:** contenerizar aplicaciones legadas para facilitar su despliegue y administración.

Docker y Docker Desktop nos proporcionan una plataforma robusta para el desarrollo, prueba y despliegue de aplicaciones en entornos consistentes y portátiles, mejorando la eficiencia y la productividad de los desarrolladores.

8.4. Conexión a PostgreSQL

La tercera base de datos que vamos a trabajar se llama PostgreSQL. Para trabajar con ella, seguiremos los siguientes pasos.

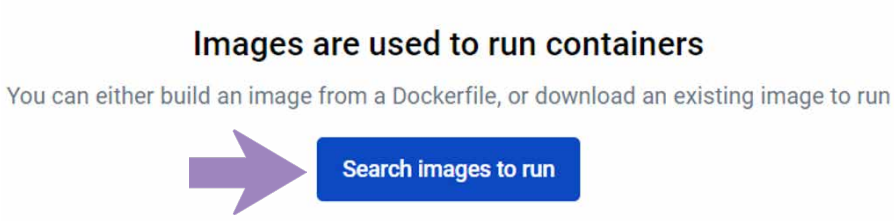
Crear Proyecto Maven en Eclipse

Vamos a crear un nuevo proyecto en Eclipse, en concreto un "Maven Project".

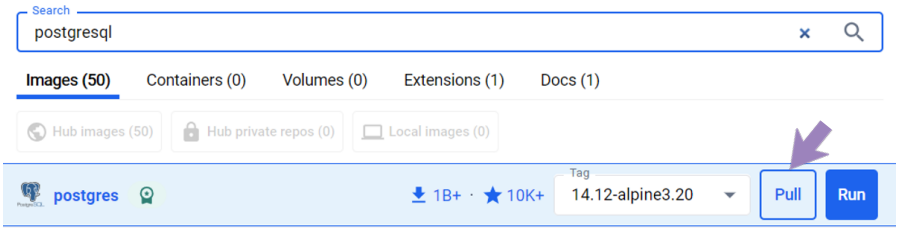
Maven es una herramienta de gestión y comprensión de proyectos que proporciona una forma estandarizada de construir, gestionar dependencias y empaquetar proyectos Java. Un proyecto Maven es un proyecto que sigue las convenciones y utiliza las características proporcionadas por Apache Maven.

Descargar y lanzar imagen en Docker

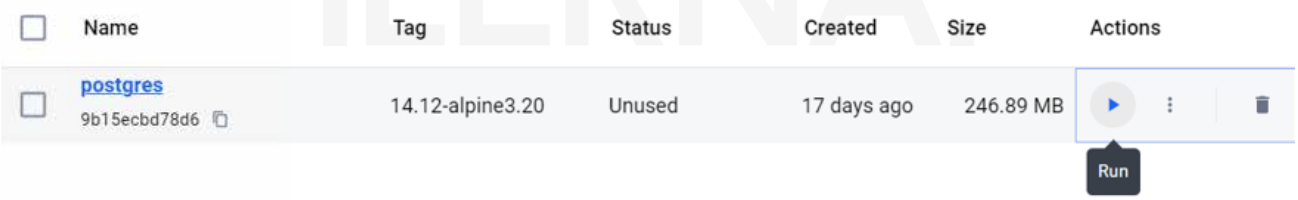
Lo primero es iniciar Docker Desktop. Después vamos a descargarnos una imagen en Docker, para ello hacemos clic en *Search images to run*.



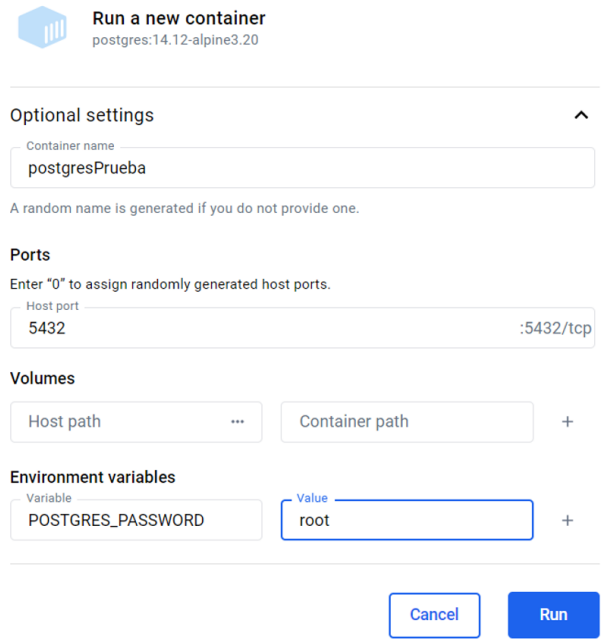
Y escogemos la imagen postgres que es la oficial de PostgreSQL y pulsamos Pull.



Ahora que está descargada vamos a hacer clic en *Run*.



Indicamos los siguientes datos para lanzar la imagen que se va a convertir en un contenedor.



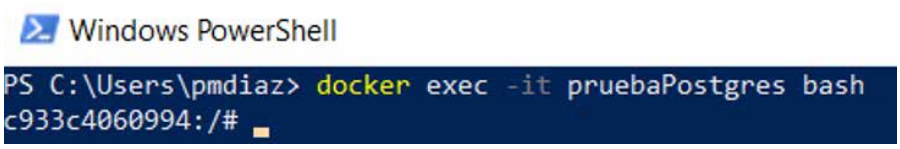
Es muy importante **mapear los puertos** que hacemos en la opción Ports, en este apartado estamos indicando que el puerto 5432 de nuestro equipo localhost apunta al puerto 5432 del contenedor de Docker.

Crear BBDD en PostgreSQL

Necesitamos crear una base de datos en PostgreSQL para poder conectarnos. Como utilizamos Docker, vamos a usar la línea de comandos, en este caso Windows Powershell (usaremos la que estimemos acorde al sistema operativo).

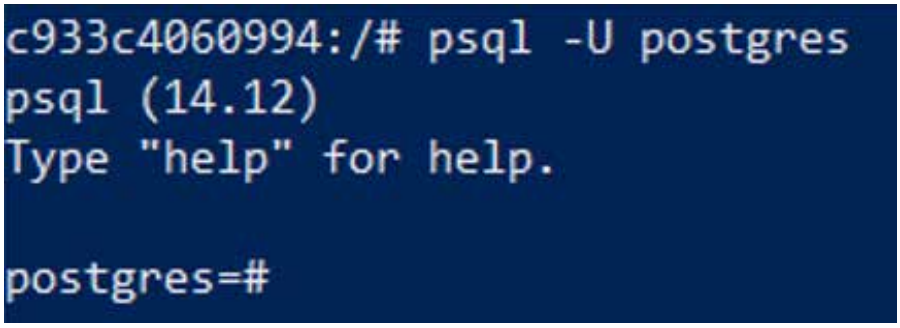
Vamos a utilizar el comando docker exec para abrir una sesión interactiva en el contenedor de PostgreSQL desde la terminal:

```
docker exec -it pruebaPostgres bash
```



Una vez dentro del contenedor, iniciaremos psql para conectarnos a PostgreSQL:

```
psql -U postgres
```



Dentro de psql, ejecutaremos el siguiente comando para crear una nueva base de datos:

```
CREATE DATABASE users;
```

Una vez que hayamos creado la base de datos, podemos salir de psql ejecutando: **\q**

Finalmente, saldremos de la sesión interactiva del contenedor PostgreSQL: **exit**

PostgreSQL es un sistema de gestión de bases de datos relacional (RDBMS) de código abierto y altamente avanzado. Es conocido por su robustez, escalabilidad y capacidad para manejar grandes volúmenes de datos de manera eficiente. PostgreSQL es ampliamente utilizado en aplicaciones que requieren almacenamiento seguro y fiable de datos, como sistemas de gestión empresarial, aplicaciones web y análisis de datos.

Características principales de PostgreSQL

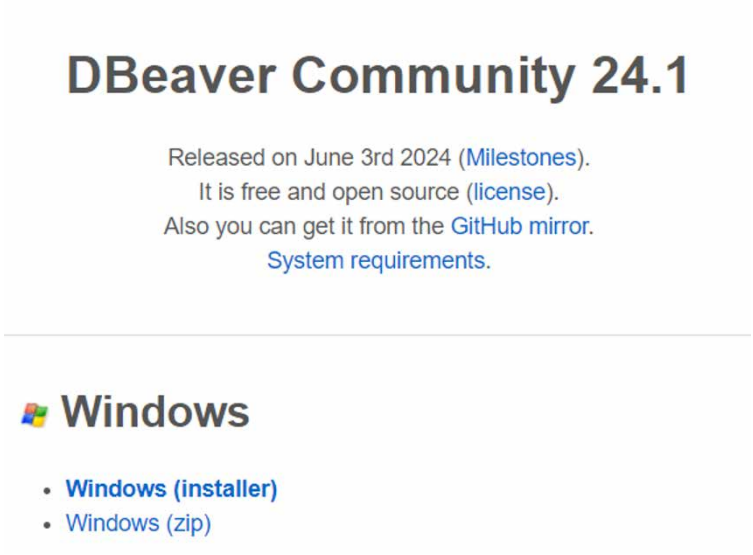
- **Soporte para SQL completo:** PostgreSQL cumple con los estándares SQL ANSI y soporta una amplia gama de características SQL avanzadas.
- **Extensibilidad:** permite definir nuevos tipos de datos, funciones y lenguajes de procedimientos almacenados, lo que lo hace altamente extensible y personalizable.
- **Transacciones ACID:** soporta transacciones ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad), asegurando la integridad de los datos incluso en condiciones adversas.
- **Integridad referencial:** ofrece soporte para claves foráneas y otras restricciones de integridad referencial para mantener la coherencia de los datos.
- **Replicación y alta disponibilidad:** dispone de herramientas integradas para replicación síncrona y asíncrona , así como para configuraciones de alta disponibilidad.
- **Soporte para indexación avanzada:** ofrece una variedad de tipos de índices como B-tree, hash, GIN, GiST, entre otros, para optimizar las consultas.
- **Soporte para JSON y otros tipos de datos no estructurados:** PostgreSQL permite almacenar y consultar datos JSON y otros tipos de datos semiestructurados.

Instalar SGBD para PostgreSQL

Vamos a instalar un SGBD para utilizar con PostgreSQL y, de esta forma, poder hacer operaciones desde el sistema gestor.

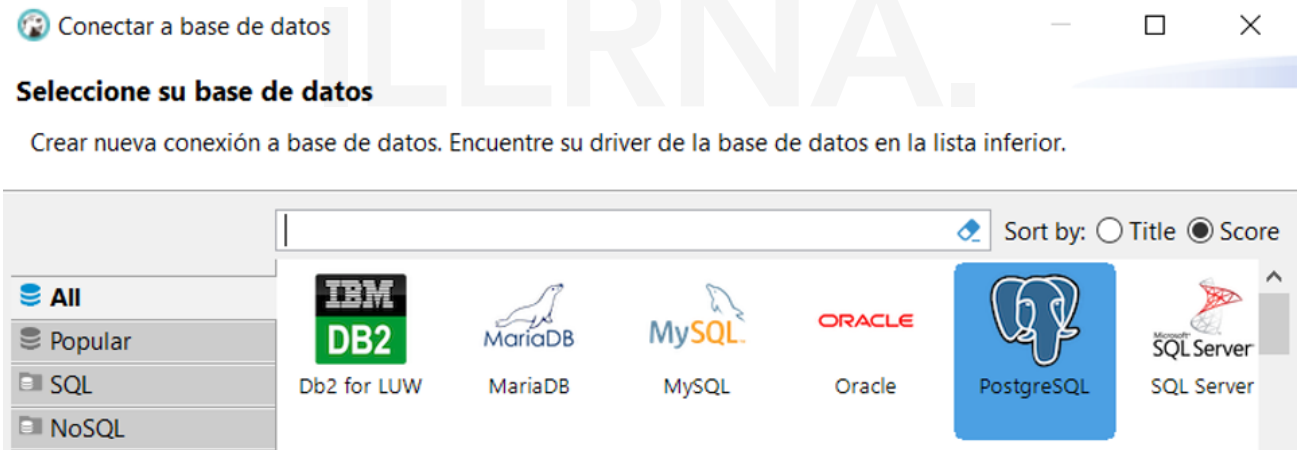
En este caso vamos a instalar DBeaver, en este enlace podemos descargar la versión para nuestro sistema operativo https://dbeaver.io/download/

link.ilerna.es/YzeF

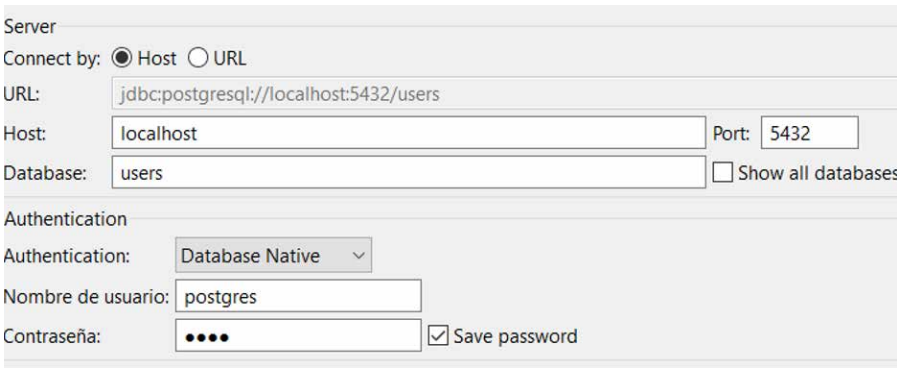


Lo más interesante de DBeaver es que permite gestionar diferentes bases de datos, por lo que podríamos usarlo como un SGBD universal si trabajamos con varios tipos de bases de datos.

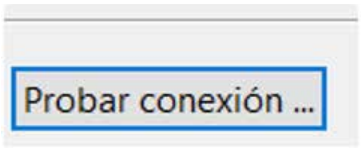
Al lanzar la aplicación, nos permite indicar qué base de datos es con la que vamos a trabajar.



Usamos estos datos:



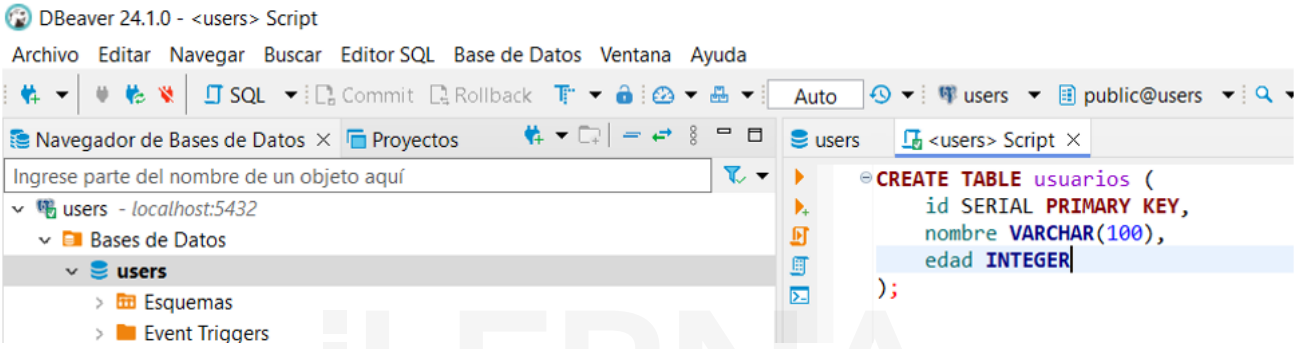
Verificamos la conexión en el botón *Probar conexión*.



Si todo va bien, debe indicar la información de la conexión.

Crear la tabla en DBeaver

Para crear la tabla users en PostgreSQL con DBeaver vamos a usar el editor SQL, que nos permite ejecutar las instrucciones que le indiquemos.



El código SQL es el siguiente:

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL
);
```

Usar PostgreSQL en Java

El último paso que nos queda es utilizar PostgreSQL en Java para poder trabajar con la base de datos que creamos anteriormente. Para ello, vamos a utilizar el siguiente código en una clase que crearemos en src/main, llamada Principal.java.

Para poder utilizar el JDBC de PostgreSQL debemos descargarlo desde las dependencias de Maven, para lo que tenemos que escribir las líneas de dependencias en el archivo pom.xml.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.
org/POM/4.0.0 https://maven.apache.org/xsd/ma-
ven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.ilerna</groupId>
 <artifactId>pruebaPostgres</artifactId>
 <version>0.0.1-SNAPSHOT</version>
 <name>pruebaPostgres</name>
 <dependencies>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.3.1</version>
    </dependency>
 </dependencies>
</project>
```

Creamos el archivo de propiedades:

```
db.url=jdbc:postgresql://localhost:5432/users
db.username=postgres
db.password=root
```

Y ya disponemos del código para trabajar con PostgreSQL y Java.

```
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

public class Principal {
```

```
    public static void main(String[] args) {
          Connection conn = null;
          Statement stmt = null;

          Properties props = new Properties();
        try (InputStream input = Principal.class.
getClassLoader().getResourceAsStream("config.proper-
ties")) {
            props.load(input);
        } catch (IOException e) {
          System.err.println("Error al leer las pro-
piedades *:(");
            e.printStackTrace();
        }

        String url = props.getProperty("db.url");
        String user = props.getProperty("db.userna-
me");
        String password = props.getProperty("db.
password");

          try {
                conn = DriverManager.getConnec-
tion(url, user, password);
                stmt = conn.createStatement();
                String sql = "CREATE TABLE IF NOT
EXISTS users (id SERIAL
                                    PRIMARY KEY, name
VARCHAR(100) NOT NULL)";
                stmt.executeUpdate(sql);
                System.out.println("Tabla 'users'
creada o ya existe.");
                // Insertar datos
                sql = "INSERT INTO users (nombre) VA-
LUES ('Bloody Mary')";
                stmt.executeUpdate(sql);
                System.out.println("Datos insertados
en la tabla 'users'.");
                // Consultar datos
                sql = "SELECT * FROM users";
                ResultSet rs = stmt.executeQuery(s-
ql);
                while (rs.next()) {
                      System.out.println("ID: " + rs.
getInt("id") + ", Name: " +
                                              rs.getS-
tring("name"));
                }
                rs.close();
          } catch (SQLException e) {
                System.err.println("Error al conectar
```

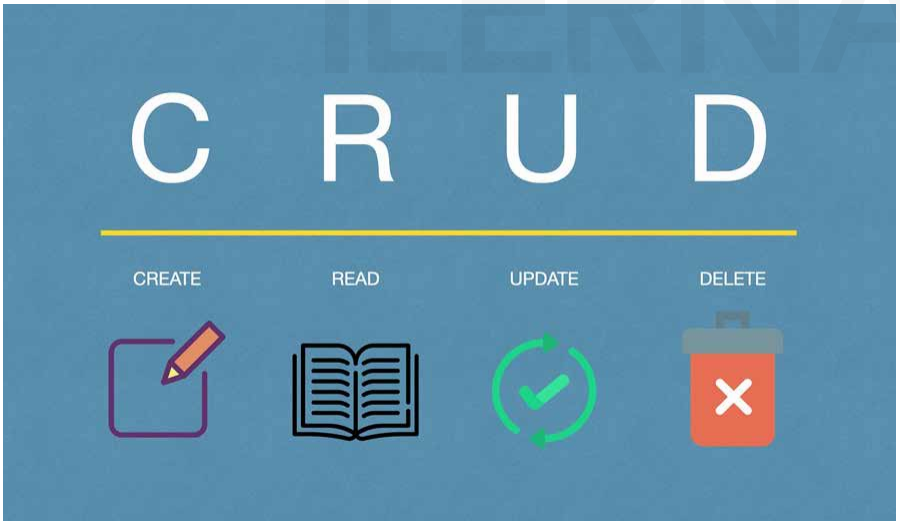
```
a la base de datos
                                          PostgreSQL:");
                e.printStackTrace(System.err);
          } finally {
                try {
                      if (conn != null)
                            conn.close();
                      if (stmt != null)
                            stmt.close();
                } catch (SQLException ex) {
                      System.err.println("Error al
cerrar la conexión con MySQL");
                      ex.printStackTrace(System.err);
                }
          }
    }
}
```

9 CRUD CON BASES DE DATOS

El término CRUD es un acrónimo que describe las cuatro operaciones fundamentales que se pueden realizar en una base de datos: **Crear (Create), Leer (Read), Actualizar (Update) y Eliminar (Delete)**. Estas operaciones son la base de la gestión de datos en cualquier aplicación que interactúe con una base de datos, ya que permiten realizar las tareas esenciales de almacenamiento, acceso y modificación de la información. Implementar estas operaciones en un sistema no solo es crucial para el manejo de datos, sino que también asegura que la información almacenada esté siempre actualizada y sea precisa.

Este tema se enfocará en cómo realizar cada una de estas operaciones CRUD desde un programa, utilizando las tres bases de datos vistas en el tema anterior y brindando a los desarrolladores las herramientas necesarias para interactuar con bases de datos de manera efectiva y eficiente.

CRUD es el conjunto de operaciones esenciales para crear, leer, actualizar y eliminar datos en una base de datos, permitiendo gestionar la información de manera completa y efectiva.



Operaciones CRUD: qué son y cómo se utilizan

- **Crear (Create):**
  - **Descripción:** implica **insertar nuevos datos** en la base de datos. Este proceso generalmente se realiza utilizando la sentencia SQL INSERT, que añade un nuevo registro en una tabla específica. Es esencial para introducir nueva información, como agregar un nuevo usuario, registrar una nueva transacción, o almacenar cualquier otro tipo de dato.

  - **Importancia:** permite el crecimiento de la base de datos al añadir nuevos datos y es el punto de entrada para toda la información que se manejará posteriormente.

- **Leer (Read):**
  - **Descripción:** se refiere a **consultar o recuperar datos** de la base de datos. Esto se hace mediante la sentencia SQL SELECT, que permite acceder a la información almacenada según criterios específicos. Es la operación más utilizada, ya que permite visualizar y analizar los datos.
  - **Importancia:** sin la capacidad de leer datos, los usuarios y aplicaciones no podrían acceder a la información almacenada, lo que haría la base de datos inútil.
- **Actualizar (Update):**
  - **Descripción:** consiste en **modificar los datos existentes** en la base de datos. Esto se logra mediante la sentencia SQL UPDATE, que permite cambiar valores específicos en los registros ya almacenados. Esta operación es fundamental para mantener la información actualizada y corregir cualquier error.
  - **Importancia:** garantiza que los datos en la base de datos reflejen el estado más actual de la información, evitando la obsolescencia y asegurando la precisión.
- **Eliminar (Delete):**
  - **Descripción:** implica **borrar datos** de la base de datos. Utilizando la sentencia SQL DELETE, se pueden eliminar registros que ya no son necesarios o que deben ser borrados por razones de seguridad o integridad. Esta operación debe realizarse con cuidado, ya que una vez eliminados los datos, generalmente no se pueden recuperar.
  - **Importancia:** mantiene la base de datos limpia y libre de datos innecesarios o erróneos, y también ayuda a gestionar el espacio y la eficiencia del sistema.

El concepto de **CRUD** es esencial para cualquier aplicación que interactúe con bases de datos, ya que define las operaciones básicas necesarias para gestionar la información almacenada. Comprender y aplicar estas operaciones es crucial para desarrollar sistemas robustos, dinámicos y confiables, capaces de manejar datos de manera efectiva.

9.1. CRUD con SQLite

Para desarrollar un **CRUD** (Create, Read, Update, Delete) usando SQLite con Java y la configuración Maven que vimos anteriormente, seguiremos los siguientes pasos.

Preparación del entorno: configuración del proyecto

Ya tenemos configurado el proyecto Maven con la dependencia de SQLite y hemos añadido nuestra base de datos en la carpeta src/main/resources/db. Ahora, nuestro objetivo es realizar las operaciones CRUD en esta base de datos. A continuación, implementaremos las funciones de:

- **Crear (Create):** insertar datos en la base de datos.
- **Leer (Read):** consultar datos almacenados.
- **Actualizar (Update):** modificar datos existentes.
- **Eliminar (Delete):** borrar datos de la base de datos.

Conectar a la base de datos

Lo primero es establecer la conexión con la base de datos tal y como ya hemos configurado en nuestra clase Principal.java. Esta conexión es fundamental, ya que cada operación CRUD necesitará acceder a la base de datos para ejecutar sus consultas.

```
Connection conn = null;
Statement stmt = null;

try {
    // Cargar el driver JDBC de SQLite
    Class.forName("org.sqlite.JDBC");

    // Obtener la ruta de la base de datos en los re-
cursos
    URL dbUrl = Principal.class.getResource("/db/my-
database.db");
    String url = "jdbc:sqlite:" + dbUrl.getPath();

    // Establecer la conexión a la base de datos
    conn = DriverManager.getConnection(url);

} catch (ClassNotFoundException | SQLException e) {
    e.printStackTrace();
}
```

Create (insertar datos en la base de datos)

Para la operación de **Crear** (Create), necesitamos insertar nuevos registros en una tabla. La tabla de usuarios (users) fue creada previamente en el ejemplo, por lo que ahora simplemente agregaremos más usuarios.

```
public void insertarUsuario(String nombre) {
    try {
        String sql = "INSERT INTO users (name) VALUES
('" + nombre + "')";
        stmt.executeUpdate(sql);
        System.out.println("Usuario '" + nombre + "'
insertado con éxito.");
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Pasos para Create:

1. Se utiliza la sentencia INSERT INTO para agregar un nuevo registro.
2. Se crea una instrucción SQL y se ejecuta utilizando el método executeUpdate() de Statement, que retorna el número de filas afectadas.

Read (consultar datos de la base de datos)

La operación de **Leer** (Read) nos permite consultar y recuperar los datos almacenados en la base de datos. Utilizamos la sentencia SQL SELECT para este propósito.

```
public void leerUsuarios() {
    try {
        String sql = "SELECT * FROM users";
        ResultSet rs = stmt.executeQuery(sql);

        while (rs.next()) {
            int id = rs.getInt("id");
            String nombre = rs.getString("name");
            System.out.println("ID: " + id + ", Nom-
bre: " + nombre);
        }
        rs.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Pasos para Read:

1. Utilizamos una consulta SELECT para obtener todos los usuarios.
2. El resultado se almacena en un ResultSet, que luego es recorrido fila por fila utilizando rs.next() para obtener los datos de cada usuario.

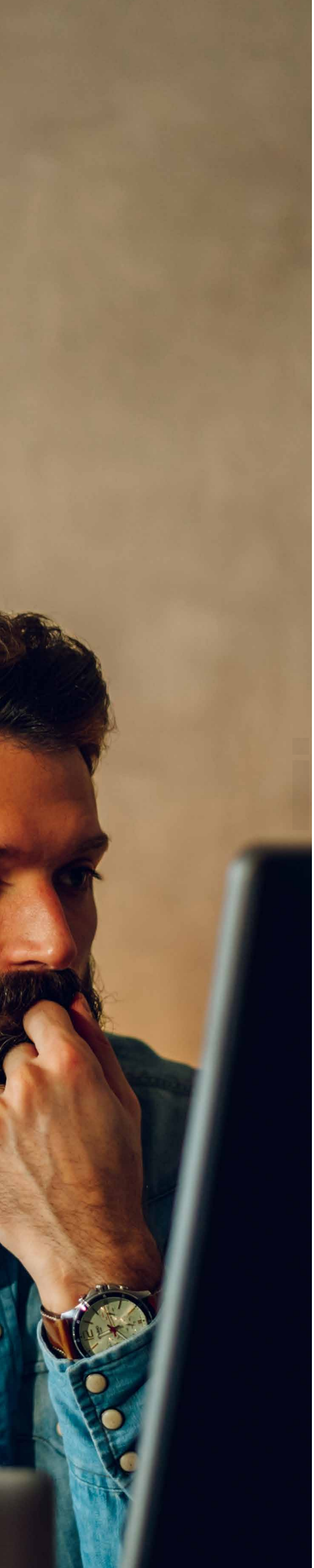
Update (actualizar datos en la base de datos)

La operación de **Actualizar** (Update) permite modificar los registros existentes en la base de datos. Utilizamos la sentencia SQL UPDATE para ello.

```
public void actualizarUsuario(int id, String nuevo-
Nombre) {
    try {
        String sql = "UPDATE users SET name = '" +
nuevoNombre + "' WHERE id = " +
                     id;
        stmt.executeUpdate(sql);
        System.out.println("Usuario con ID: " + id +
" ha sido actualizado a '" +
                           nuevoNombre + "'.");
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Pasos para Update:

1. Utilizamos la sentencia SQL UPDATE para cambiar los datos de un registro existente.
2. Se especifica el WHERE para identificar el registro a modificar, y se utiliza executeUpdate() para ejecutar la instrucción SQL.



Delete (eliminar datos de la base de datos)

Finalmente, la operación de **Eliminar** (Delete) nos permite borrar registros de la base de datos. Para ello, utilizamos la sentencia SQL DELETE.

```
public void eliminarUsuario(int id) {
    try {
        String sql = "DELETE FROM users WHERE id = "
+ id;
        stmt.executeUpdate(sql);
        System.out.println("Usuario con ID: " + id +
" ha sido eliminado.");
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Pasos para Delete:

1. Utilizamos la sentencia SQL DELETE para eliminar un registro.
2. Se especifica el WHERE para identificar el registro que se desea eliminar y se utiliza executeUpdate() para ejecutar la instrucción.

Cerrar la conexión

Recuerda que, tras realizar cualquier operación en la base de datos, siempre es una buena práctica **cerrar** los recursos para liberar memoria y evitar posibles bloqueos en la base de datos:

```
finally {
    try {
        if (stmt != null) stmt.close();
        if (conn != null) conn.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Hemos visto cómo desarrollar las cuatro operaciones del CRUD en Java usando SQLite. Cada una de estas operaciones es esencial para gestionar los datos en cualquier aplicación. Implementar correctamente estas funciones asegura que nuestra aplicación sea capaz de interactuar con la base de datos de manera eficiente, permitiendo crear, leer, actualizar y eliminar datos según sea necesario.

9.2. CRUD con MySQL

Para desarrollar un **CRUD** usando MySQL con Java y la configuración Maven que hemos visto anteriormente, vamos a realizar los siguientes pasos.

Conectar a la base de datos

Para poder realizar las operaciones CRUD, primero necesitamos establecer una conexión a la base de datos utilizando los datos de configuración del archivo config.properties. El siguiente código muestra cómo leer estas propiedades y conectarse a la base de datos:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;
import java.io.InputStream;
import java.io.IOException;

public class DBConnection {
    public static Connection getConnection() {
        Connection conn = null;
        Properties props = new Properties();

        try (InputStream input = DBConnection.class.
getClassLoader().getResourceAsStream("config.proper-
ties")) {
            props.load(input);
            String url = props.getProperty("db.url");
            String user = props.getProperty("db.user-
name");
            String password = props.getProperty("db.
password");

            conn = DriverManager.getConnection(url,
user, password);
            System.out.println("Conexión establecida
con la base de datos MySQL.");

        } catch (IOException | SQLException e) {
            e.printStackTrace();
        }

        return conn;
    }
}
```

Este método getConnection() obtiene la conexión a la base de datos MySQL usando las propiedades del archivo config.properties.

Create (insertar datos en la base de datos)

La operación **Crear** (Create) nos permite insertar nuevos registros en una tabla. Usamos la sentencia SQL INSERT INTO para esto. A continuación, creamos un método para insertar usuarios en la tabla users:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class UsuarioDAO {
    public void insertarUsuario(String nombre) {
        String sql = "INSERT INTO users (name) VALUES
(?)";

        try (Connection conn = DBConnection.getCon-
nection();
             PreparedStatement pstmt = conn.prepareS-
tatement(sql)) {

            pstmt.setString(1, nombre);
            pstmt.executeUpdate();
            System.out.println("Usuario '" + nombre +
"' insertado correctamente.");

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Pasos para Create:

1. **PreparedStatement:** utilizamos un PreparedStatement para evitar inyecciones SQL y manejar parámetros.
2. **Inserción de datos:** el método pstmt.executeUpdate() inserta el nuevo registro en la base de datos.

Read (consultar datos de la base de datos)

La operación **Leer** (Read) permite recuperar los datos de la tabla. Usamos la sentencia SQL SELECT para obtener información. A continuación, creamos un método para leer los usuarios:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class UsuarioDAO {
    public void leerUsuarios() {
        String sql = "SELECT * FROM users";

        try (Connection conn = DBConnection.getCon-
nection();
             PreparedStatement pstmt = conn.prepareS-
tatement(sql);
             ResultSet rs = pstmt.executeQuery()) {

            while (rs.next()) {
                int id = rs.getInt("id");
                String nombre = rs.getString("name");
                System.out.println("ID: " + id + ",
Nombre: " + nombre);
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Pasos para Read:

1. **PreparedStatement:** se utiliza para crear una consulta SQL.
2. **ResultSet:** almacena el resultado de la consulta, y lo recorremos con rs.next() para acceder a los datos.

Update (actualizar datos en la base de datos)

La operación **Actualizar** (Update) permite modificar los datos existentes en la base de datos. Usamos la sentencia SQL UPDATE para actualizar los registros:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
```

```
public class UsuarioDAO {
    public void actualizarUsuario(int id, String nue-
voNombre) {
        String sql = "UPDATE users SET name = ? WHERE
id = ?";

        try (Connection conn = DBConnection.getCon-
nection();
             PreparedStatement pstmt = conn.prepareS-
tatement(sql)) {

            pstmt.setString(1, nuevoNombre);
            pstmt.setInt(2, id);
            int filasActualizadas = pstmt.executeUpda-
te();

            if (filasActualizadas > 0) {
                System.out.println("Usuario con ID: "
+ id + " actualizado correctamente.");
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```


Pasos para Update:

1. **PreparedStatement:** usamos un PreparedStatement para evitar problemas de seguridad.
2. **Especificar el WHERE:** es necesario para asegurar que estamos actualizando el registro correcto.

Delete (eliminar datos de la base de datos)

Finalmente, la operación **Eliminar** (Delete) permite borrar registros de la base de datos. Usamos la sentencia SQL DELETE para eliminar un usuario específico:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class UsuarioDAO {
    public void eliminarUsuario(int id) {
        String sql = "DELETE FROM users WHERE id =
```

```
?";

        try (Connection conn = DBConnection.getCon-
nection();
             PreparedStatement pstmt = conn.prepareS-
tatement(sql)) {

            pstmt.setInt(1, id);
            int filasEliminadas = pstmt.executeUpda-
te();

            if (filasEliminadas > 0) {
                System.out.println("Usuario con ID: "
+ id + " eliminado correctamente.");
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Pasos para Delete:

1. **PreparedStatement:** se usa para evitar inyecciones SQL y manejar el parámetro.
2. **Especificar el WHERE:** esto asegura que eliminamos el registro correcto, evitando borrar todos los datos accidentalmente.

Cerrar conexiones

En cada uno de los métodos CRUD, es esencial cerrar los recursos (Connection, PreparedStatement, y ResultSet) para evitar problemas de rendimiento o conexiones innecesarias abiertas.

Cada operación abre una conexión, ejecuta una instrucción y luego cierra esa conexión inmediatamente, lo que asegura que los recursos de la base de datos se liberen rápidamente.

Desarrollar un CRUD en Java utilizando MySQL sigue una estructura clara: establecer una conexión con la base de datos, crear consultas SQL usando PreparedStatement para seguridad y eficiencia y procesar los resultados. Implementar estas operaciones nos permitirá gestionar de manera efectiva los datos almacenados en la base de datos, realizando inserciones, consultas, actualizaciones y eliminaciones de manera segura y eficiente.

9.3. CRUD con PostgreSQL

PostgreSQL es una bases de datos objeto-relacionales (BBDD OR) que combina características de las bases de datos relacionales (SQL) y de las bases de datos orientadas a objetos, permitiendo almacenar estructuras más complejas sin perder la eficiencia del modelo relacional.

Para desarrollar un **CRUD** usando PostgreSQL con Java y la configuración que hemos proporcionado (con Maven y config.properties para la conexión), vamos a seguir un conjunto de pasos para implementar las funcionalidades necesarias.

En este proceso, usaremos el controlador JDBC para PostgreSQL que ya está configurado en tu archivo pom.xml, así como el archivo config.properties para cargar las credenciales de conexión.

Conectar a la base de datos

Lo primero que necesitamos es establecer una conexión a la base de datos PostgreSQL. El siguiente método leerá los datos del archivo config.properties y proporcionará una conexión.


```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;
import java.io.InputStream;
import java.io.IOException;

public class DBConnection {
    public static Connection getConnection() {
        Connection conn = null;
        Properties props = new Properties();

        try (InputStream input = DBConnection.class.
getClassLoader().getResourceAsStream("config.proper-
ties")) {
            props.load(input);
            String url = props.getProperty("db.url");
            String user = props.getProperty("db.user-
name");
            String password = props.getProperty("db.
password");

            conn = DriverManager.getConnection(url,
user, password);
            System.out.println("Conexión establecida
con la base de datos PostgreSQL.");

        } catch (IOException | SQLException e) {
```

```
            e.printStackTrace();
        }

        return conn;
    }
}
```

Este método getConnection() obtiene la conexión a la base de datos PostgreSQL usando las propiedades de config.properties.

Create (insertar datos en la base de datos)

Para la operación **Crear** (Create), necesitamos insertar un nuevo registro en la tabla users. Utilizamos la sentencia SQL INSERT INTO para insertar datos en la tabla.

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class UsuarioDAO {
    public void insertarUsuario(String nombre) {
        String sql = "INSERT INTO users (nombre) VA-
LUES (?)";

        try (Connection conn = DBConnection.getCon-
nection();
             PreparedStatement pstmt = conn.prepareS-
tatement(sql)) {

            pstmt.setString(1, nombre);
            pstmt.executeUpdate();
            System.out.println("Usuario '" + nombre +
"' insertado correctamente.");

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Pasos para Create:

1. Utilizamos un PreparedStatement para evitar ataques de inyección SQL y manejar parámetros de manera segura.
2. El método pstmt.executeUpdate() inserta el registro en la base de datos.

Read (consultar datos de la base de datos)

La operación Leer (Read) permite consultar los datos almacenados en la tabla users. Usamos la sentencia SQL SELECT para obtener la información.

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class UsuarioDAO {
    public void leerUsuarios() {
        String sql = "SELECT * FROM users";

        try (Connection conn = DBConnection.getCon-
nection();
             PreparedStatement pstmt = conn.prepareS-
tatement(sql);
             ResultSet rs = pstmt.executeQuery()) {

            while (rs.next()) {
                int id = rs.getInt("id");
                String nombre = rs.getString("nom-
bre");
                System.out.println("ID: " + id + ",
Nombre: " + nombre);
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

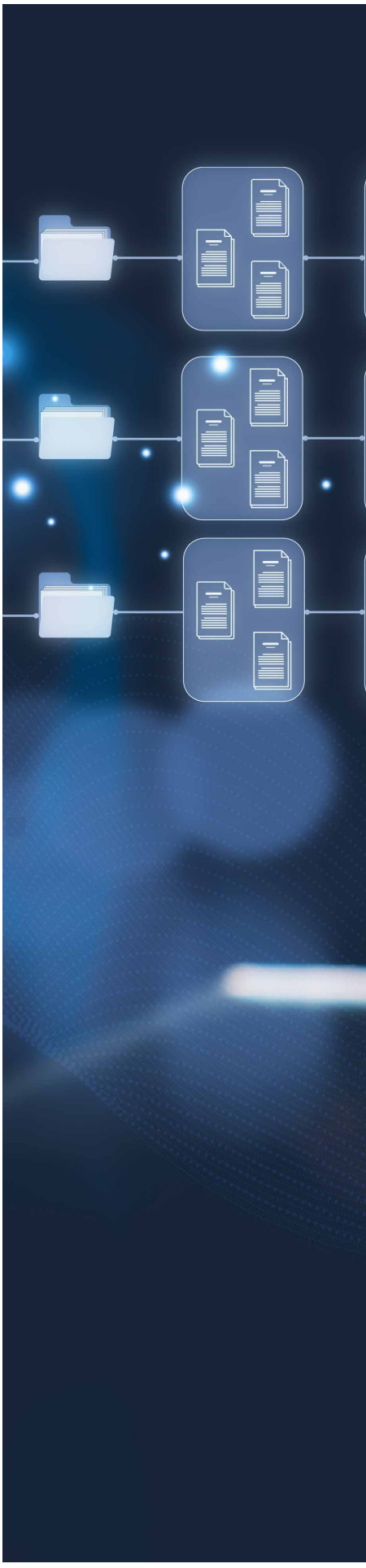
Pasos para Read:

1. Utilizamos un PreparedStatement para construir la consulta SQL.
2. El ResultSet se usa para almacenar y recorrer los resultados de la consulta.

Update (actualizar datos en la base de datos)

La operación **Actualizar** (Update) permite modificar los registros existentes en la base de datos. Utilizamos la sentencia SQL UPDATE para actualizar los datos.

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class UsuarioDAO {
    public void actualizarUsuario(int id, String nue-
voNombre) {
        String sql = "UPDATE users SET nombre = ?
WHERE id = ?";

        try (Connection conn = DBConnection.getCon-
nection();
             PreparedStatement pstmt = conn.prepareS-
tatement(sql)) {

            pstmt.setString(1, nuevoNombre);
            pstmt.setInt(2, id);
            int filasActualizadas = pstmt.executeUpda-
te();

            if (filasActualizadas > 0) {
                System.out.println("Usuario con ID: "
+ id + " actualizado correctamente.");
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```



Pasos para Update:

1. Usamos PreparedStatement para evitar inyecciones SQL y manejar los parámetros de forma segura.
2. Aseguramos que la sentencia SQL contiene el WHERE correcto para actualizar el registro deseado.

Delete (eliminar datos de la base de datos)

La operación Eliminar (Delete) permite borrar registros de la base de datos. Utilizamos la sentencia SQL DELETE para eliminar un usuario específico.

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class UsuarioDAO {
```

```
    public void eliminarUsuario(int id) {
        String sql = "DELETE FROM users WHERE id =
?";

        try (Connection conn = DBConnection.getCon-
nection();
             PreparedStatement pstmt = conn.prepareS-
tatement(sql)) {

            pstmt.setInt(1, id);
            int filasEliminadas = pstmt.executeUpda-
te();

            if (filasEliminadas > 0) {
                System.out.println("Usuario con ID: "
+ id + " eliminado correctamente.");
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```


Pasos para Delete:

1. Utilizamos PreparedStatement para manejar el parámetro de manera segura y prevenir problemas de inyección SQL.
2. Aseguramos que el WHERE se usa correctamente para identificar el registro que se va a eliminar.

Cerrar conexiones

Recuerda que, después de realizar cualquier operación en la base de datos, es importante cerrar los recursos (como Connection, PreparedStatement y ResultSet) para liberar recursos y evitar fugas de memoria.

```
finally {
    try {
        if (stmt != null) stmt.close();
        if (conn != null) conn.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Desarrollar un CRUD en Java con PostgreSQL sigue un enfoque claro: primero establecemos una conexión a la base de datos, y luego usamos sentencias SQL (INSERT, SELECT, UPDATE, DELETE) para realizar las operaciones CRUD.

Usar PreparedStatement es esencial para evitar inyecciones SQL y manejar los datos de manera eficiente. La implementación de estas operaciones permite gestionar los datos de la base de datos PostgreSQL de forma segura y eficiente en tu aplicación Java.

9.4. CRUD con db4o

db4o (*database for objects*) es una **base de datos orientada a objetos (BBDD OO)** desarrollada para integrarse fácilmente en aplicaciones Java y .NET. Su principal ventaja es que permite almacenar y recuperar objetos de manera nativa sin necesidad de mapearlos a tablas relacionales, lo que simplifica el desarrollo y mejora el rendimiento en comparación con bases de datos SQL tradicionales.

db4o es especialmente útil en aplicaciones embebidas y en sistemas con recursos limitados, ya que es ligera y no requiere un servidor centralizado. Los objetos se pueden persistir directamente en archivos sin necesidad de definir esquemas previos, lo que facilita el mantenimiento y la evolución de la estructura de datos.

Entre sus características destacadas están la consulta mediante Native Queries (expresiones Java o C# que reemplazan el SQL), la indexación automática, la capacidad de replicación y el soporte para transacciones y concurrencia. También ofrece mecanismos para la compresión y cifrado de datos.

A pesar de sus ventajas, db4o dejó de recibir soporte y desarrollo activo en 2014 tras la adquisición de su empresa por Versant. Aun así, sigue siendo una opción interesante para proyectos que requieran una solución simple y eficiente para el almacenamiento de objetos sin una base de datos relacional.

Aquí tienes una estructura para hacer un **CRUD en Java con db4o** siguiendo un enfoque similar al de PostgreSQL:

Configuración de db4o

Para trabajar con **db4o** en Java, primero necesitas agregar la dependencia en **Maven**:

```
<dependency>
    <groupId>com.db4o</groupId>
    <artifactId>db4o-full-java5</artifactId>
    <version>8.1</version>
</dependency>
```

link.ilerna.es/lxdW

Si no usas Maven, puedes descargar la librería desde db4o y añadirla manualmente al classpath.

Clase de conexión

En **db4o**, la conexión a la base de datos se maneja a través de un **ObjectContainer**. Crearemos una clase DBManager para gestionar la conexión.

```
import com.db4o.*;
public class DBManager {
    private static final String DB_FILE = "usuarios.
db";
    private static ObjectContainer db = null;

    // Obtener la conexión
    public static ObjectContainer getConnection() {
        if (db == null || db.ext().isClosed()) {
            db = Db4oEmbedded.openFile(Db4oEmbedded.
newConfiguration(), DB_FILE);
        }
        return db;
    }

    // Cerrar la conexión
    public static void closeConnection() {
        if (db != null) {
            db.close();
        }
    }
}
```

Crear la clase Usuario

```
public class Usuario {
    private String nombre;
    private int edad;

    public Usuario() { } // Constructor vacío para
db4o

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

    // Getters y Setters
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nom-
bre = nombre; }

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

```
}

    @Override
    public String toString() {
        return "Usuario{" + "nombre='" + nombre +
'\'' + ", edad=" + edad + '}';
    }
}
```

Implementación del CRUD en UsuarioDAO

Crear (Create)

```
import com.db4o.ObjectContainer;
public class UsuarioDAO {
    public void insertarUsuario(Usuario usuario) {
        ObjectContainer db = DBManager.getConnec-
tion();
        db.store(usuario);
        db.commit();
        System.out.println("Usuario insertado: " +
usuario);
    }
}
```

Leer (Read)

```
import com.db4o.ObjectSet;

public class UsuarioDAO {
    public void leerUsuarios() {
        ObjectContainer db = DBManager.getConnec-
tion();
        ObjectSet<Usuario> usuarios = db.query(Usua-
rio.class);

        if (usuarios.isEmpty()) {
            System.out.println("No hay usuarios en la
base de datos.");
        } else {
            for (Usuario u : usuarios) {
                System.out.println(u);
            }
        }
    }
}
```

Actualizar (Update)

```
public class UsuarioDAO {
    public void actualizarUsuario(String nombre,
String nuevoNombre, int nuevaEdad) {
        ObjectContainer db = DBManager.getConnec-
tion();
        ObjectSet<Usuario> resultado = db.queryB-
yExample(new Usuario(nombre, 0));

        if (!resultado.isEmpty()) {
            Usuario usuario = resultado.next();
            usuario.setNombre(nuevoNombre);
            usuario.setEdad(nuevaEdad);
            db.store(usuario);
            db.commit();
            System.out.println("Usuario actualizado:
" + usuario);
        } else {
            System.out.println("Usuario no encontra-
do.");
        }
    }
}
```


Eliminar (Delete)

```
public class UsuarioDAO {
    public void eliminarUsuario(String nombre) {
        ObjectContainer db = DBManager.getConnec-
tion();
        ObjectSet<Usuario> resultado = db.queryB-
yExample(new Usuario(nombre, 0));

        if (!resultado.isEmpty()) {
            Usuario usuario = resultado.next();
            db.delete(usuario);
            db.commit();
            System.out.println("Usuario eliminado: "
+ usuario);
        } else {
            System.out.println("Usuario no encontra-
do.");
        }
    }
}
```

Prueba del CRUD

```
public class Main {
    public static void main(String[] args) {
        UsuarioDAO dao = new UsuarioDAO();

        // Crear usuarios
        dao.insertarUsuario(new Usuario("Juan", 25));
        dao.insertarUsuario(new Usuario("María",
30));

        // Leer usuarios
        System.out.println("Usuarios en la base de
datos:");
        dao.leerUsuarios();

        // Actualizar usuario
        dao.actualizarUsuario("Juan", "Juan Pérez",
26);

        // Leer usuarios después de la actualización
        System.out.println("Usuarios después de la
actualización:");
        dao.leerUsuarios();

        // Eliminar usuario
        dao.eliminarUsuario("María");

        // Leer usuarios después de la eliminación
        System.out.println("Usuarios después de la
eliminación:");
        dao.leerUsuarios();

        // Cerrar conexión
        DBManager.closeConnection();
    }
}
```

Este CRUD en **Java con db4o** te permite:

- Insertar objetos de manera **directa** sin necesidad de convertirlos en tablas.
- Consultar todos los objetos almacenados.
- Actualizar un objeto buscándolo por un criterio.
- Eliminar objetos fácilmente.

10 PRÓXIMOS PASOS

El aprendizaje en programación y bases de datos proporciona las nociones para crear aplicaciones prácticas y resolver problemas complejos. Esta formación abarca conceptos esenciales, como la programación orientada a objetos, que permite modelar problemas del mundo real utilizando clases, herencia y polimorfismo. Además, se enseñan técnicas para depurar y manejar errores, facilitando la identificación y resolución de problemas en el código.

La interacción con bases de datos es un aspecto fundamental, ya que permite conectar aplicaciones a sistemas como SQLite o MySQL y realizar operaciones básicas como crear, leer, actualizar y eliminar datos (CRUD). Asimismo, el manejo de estructuras de datos como arrays y listas, junto con librerías estándar de Java, amplía las capacidades para desarrollar interfaces gráficas de usuario y manipular datos de forma eficiente.

Este conocimiento básico capacita para desarrollar proyectos como aplicaciones CRUD para gestionar inventarios o agendas, sistemas básicos de gestión con usuarios y permisos, o simulaciones y videojuegos sencillos. También se sientan las bases para enfrentar proyectos más ambiciosos, como los exigidos en el proyecto final del ciclo formativo, abordando al menos el 70% de los requisitos con las habilidades adquiridas.

El siguiente paso en este camino incluye conceptos avanzados, como patrones de diseño, programación concurrente y optimización de algoritmos, así como habilidades en desarrollo web y móvil. La incursión en Python complementa estos conocimientos, destacando su simplicidad, versatilidad y popularidad en áreas como *Machine Learning* e inteligencia artificial.

Lo aprendido no solo permite crear aplicaciones prácticas, sino también explorar nuevas tecnologías y lenguajes, sentando las bases para una carrera sólida en programación y desarrollo de software. La transición a áreas avanzadas y especializadas amplía las posibilidades y retos, consolidando al estudiantado como profesional del sector tecnológico.

10.1. Qué puedo hacer con lo aprendido

Tras lo visto en esta asignatura, habremos adquirido un conjunto de habilidades clave en programación y bases de datos que nos permitirán crear aplicaciones sencillas y medianamente complejas. En este curso, hemos aprendido sobre:

- **Programación orientada a objetos:** comprender conceptos como clases, objetos, herencia, polimorfismo, y encapsulación. Esto permite modelar problemas del mundo real en código y crear aplicaciones bien estructuradas.
- **Depuración y manejo de errores:** saber cómo identificar y corregir errores en el código, utilizando herramientas como los logs, el depurador y el manejo de excepciones.
- **Interacción con bases de datos:** conectarse a bases de datos como SQLite, MySQL y PostgreSQL desde Java, lo que permite manejar datos de manera eficiente y realizar operaciones CRUD (crear, leer, actualizar y eliminar) en las aplicaciones.
- **Uso de librerías y estructuras de datos:** trabajar con librerías estándar de Java para manejar archivos, datos complejos y GUIs, además de entender el uso de arrays, listas, pilas y colas para organizar datos.

Proyectos que podríamos realizar

- **Aplicaciones CRUD:** como una agenda de contactos o un sistema de inventario donde se pueda agregar, modificar, eliminar y consultar registros.
- **Aplicaciones con interfaces gráficas sencillas:** utilizando las bibliotecas aprendidas, podríamos crear interfaces simples como calculadoras, gestores de tareas o aplicaciones de escritorio que interactúan con bases de datos.
- **Sistemas de gestión básicos:** aplicaciones que manejen usuarios, roles y permisos para pequeñas empresas o grupos de trabajo, conectados a bases de datos.
- **Simulaciones o juegos básicos:** usando el paradigma orientado a objetos, podríamos modelar pequeños videojuegos o simulaciones basadas en reglas predefinidas.

Si analizamos lo que se pide en el proyecto final del ciclo, aprendiendo y desarrollando las habilidades vistas en esta asignatura, podríamos disponer ya al menos del 70% de los requerimientos para llevar a cabo este tipo de proyectos. Por ello, es muy importante tener claros los conceptos tratados y ser capaces de llevarlos a la práctica mediante los diferentes ejercicios vistos en clase.

10.2. Qué es lo siguiente

Ahora es el momento de pensar en los siguientes pasos que nos permitirán continuar creciendo en el oficio de la programación y expandir estos conocimientos para enfrentar desafíos más complejos.

Programación avanzada

El siguiente paso natural es profundizar en los conceptos avanzados de programación. Esto incluye:

- **Patrones de diseño:** aprender a utilizar patrones como Singleton, Factory, Observer, entre otros, para escribir código más eficiente y estructurado.
- **Programación concurrente y multihilo:** trabajar con múltiples hilos permitirá crear aplicaciones más rápidas y eficientes que ejecuten tareas simultáneamente.
- **Optimización y eficiencia:** mejorar el rendimiento de las aplicaciones, optimizando algoritmos y comprendiendo la complejidad temporal y espacial.

Desarrollo web

El desarrollo web es una habilidad clave para cualquier programador moderno. Aquí, aprenderá a:

- **Trabajar con lenguajes y tecnologías web** como HTML, CSS, y JavaScript para crear interfaces de usuario atractivas y funcionales.
- **Backend con Java:** usando frameworks como Spring o Java EE para desarrollar aplicaciones web robustas conectadas a bases de datos.
- **APIs y servicios web:** desarrollar y consumir APIs RESTful para la integración entre diferentes sistemas.

Desarrollo móvil

La siguiente área a explorar podría ser el desarrollo de aplicaciones móviles:

- **Android con Java y Kotlin:** dado que ya dominamos Java, será relativamente sencillo empezar a crear aplicaciones móviles para dispositivos Android. También podremos trabajar con el lenguaje Kotlin específico de aplicaciones Android.
- **Desarrollo multiplataforma:** con tecnologías como Flutter o React Native, podremos aprender a desarrollar aplicaciones que funcionen tanto en Android como en iOS.

Profundización en bases de datos

- **Bases de datos no relacionales (NoSQL):** conocer bases de datos como MongoDB o Cassandra para trabajar con datos no estructurados.
- **Optimización de consultas SQL:** aprender a optimizar consultas y manejar grandes volúmenes de datos.
- **Sistemas de alta disponibilidad y replicación:** implementar soluciones de bases de datos escalables y redundantes.

DevOps y gestión de proyectos

A medida que avancemos, también será importante conocer aspectos de **DevOps** y gestión de proyectos. **DevOps** es una combinación de **Development** (Desarrollo) y **Operations** (Operaciones). Es una cultura, conjunto de prácticas y herramientas que integran y automatizan los procesos entre los equipos de desarrollo de software y los equipos de operaciones de TI.

Su objetivo principal es **acelerar el ciclo de vida del desarrollo de software**, desde la creación y prueba hasta la implementación y mantenimiento, mejorando la colaboración entre estos equipos y entregando productos de alta calidad de forma más rápida y eficiente.

- **Control de versiones con Git:** dominar el uso de sistemas de control de versiones para trabajar en equipo y gestionar proyectos de manera eficiente.
- **CI/CD (integración y entrega continua):** aprender a automatizar el ciclo de vida de una aplicación con herramientas como Jenkins, Docker y Kubernetes.

Desarrollo de software seguro

- **Seguridad en el desarrollo:** comprender y aplicar principios de seguridad en el desarrollo de software para proteger las aplicaciones contra vulnerabilidades comunes como la inyección SQL, el manejo de datos sensibles y ataques de fuerza bruta.

Metodologías ágiles

- Finalmente, es importante la familiarización con **metodologías ágiles** como Scrum o Kanban, las cuales son ampliamente utilizadas en equipos de desarrollo para mejorar la colaboración, la productividad y la entrega de productos de calidad.

El siguiente paso formativo nos llevará a desarrollar habilidades avanzadas y especializadas que nos prepararán para trabajar en entornos profesionales.

El camino para convertirse en profesionales de la programación abarca áreas como el desarrollo web, móvil, optimización de bases de datos y una comprensión profunda de las mejores prácticas en ingeniería de software. Cada nuevo paso abrirá las puertas a proyectos más grandes, más complejos y desafiantes.

10.3. Probemos otro lenguaje: Python

Tras haber visto los conceptos de programación orientada a objetos en Java y trabajar con muchos de los aspectos que se pueden realizar en ese lenguaje, vamos a ir un poco más allá en este final del libro y vamos a trabajar con otro lenguaje de programación diferente a Java, en este caso usaremos Python.

Python es un lenguaje de programación de alto nivel, interpretado y de propósito general. Sus principales características incluyen:

- **Simplicidad y legibilidad:** Python tiene una sintaxis clara y sencilla que permite a los desarrolladores centrarse en resolver problemas en lugar de preocuparse por la complejidad del código.
- **Multiparadigma:** soporta programación orientada a objetos, funcional y estructurada.
- **Extensa biblioteca estándar:** Python viene con una biblioteca estándar muy rica que incluye módulos para manejar tareas comunes como trabajar con archivos, redes, expresiones regulares, matemáticas y más.

- **Gran comunidad y ecosistema:** tiene una gran comunidad de usuarios y desarrolladores, lo que resulta en una vasta cantidad de recursos, librerías de terceros y documentación.
- **Interoperabilidad:** Python puede integrarse con otros lenguajes de programación como C, C++, y Java.
- **Portabilidad:** es compatible con múltiples plataformas (Windows, MacOS, Linux).

Python ha ganado una enorme popularidad en el campo del ***Machine Learning*** (ML) y la **inteligencia artificial** (IA), y hay varias razones por las que es tan importante en esta área:

- **Amplia disponibilidad de bibliotecas especializadas:** cuenta con un ecosistema robusto de bibliotecas dedicadas al *Machine Learning* y la ciencia de datos:
  - **NumPy y SciPy:** para cálculos numéricos y científicos, permitiendo realizar operaciones eficientes con grandes conjuntos de datos.
  - **Pandas:** facilita la manipulación y análisis de datos estructurados, permitiendo cargar, limpiar y transformar datos de manera eficiente.
  - **Scikit-learn:** proporciona herramientas para construir modelos de *Machine Learning*, como regresiones, clasificación, clustering y reducción de dimensionalidad.
  - **TensorFlow y PyTorch:** son bibliotecas líderes para el aprendizaje profundo (***deep learning***), que permiten construir redes neuronales artificiales y ejecutar complejas tareas de inteligencia artificial.
- **Facilidad de uso y aprendizaje:** en el ámbito del *Machine Learning*, los científicos de datos suelen centrarse más en la lógica matemática y estadística detrás de los modelos que en el código complejo. La simplicidad de Python permite a los usuarios implementar fácilmente algoritmos sin preocuparse por la sintaxis complicada. Esto lo convierte en una excelente opción tanto para principiantes como para expertos en IA.
- **Extensibilidad y flexibilidad:** permite integrar y trabajar con otros lenguajes de programación, lo que facilita la colaboración entre equipos y la incorporación de otros recursos en los proyectos de *Machine Learning*. Además, Python puede ser utilizado en diferentes etapas del desarrollo de un modelo, desde el preprocesamiento de los datos hasta la implementación del modelo entrenado en producción.

- **Comunidad y soporte:** la comunidad de Python para *Machine Learning* es una de las más grandes y activas en el mundo. Esto significa que hay abundante documentación, tutoriales, paquetes y recursos disponibles para ayudar a los desarrolladores a avanzar rápidamente en sus proyectos de *Machine Learning*.
- **Compatibilidad con herramientas de visualización:** también destaca por sus potentes bibliotecas de visualización de datos como **Matplotlib**, **Seaborn** y **Plotly**, que permiten a los desarrolladores crear gráficos y visualizar patrones dentro de los datos. Esto es fundamental en *Machine Learning* para entender las relaciones en los datos, evaluar la performance de los modelos y comunicar los resultados de manera clara y visual.
- **Uso en investigación y desarrollo:** Python es ampliamente utilizado en la academia y en la investigación debido a su facilidad para escribir código experimental rápido. Muchos de los avances en *Machine Learning* e inteligencia artificial se implementan primero en Python, lo que permite a los investigadores compartir sus avances con la comunidad de manera eficiente.

Python es un lenguaje fundamental en el mundo de la programación debido a su simplicidad, versatilidad y la inmensa comunidad que lo respalda.

Su importancia en ***Machine Learning*** radica en su facilidad de uso, el potente ecosistema de bibliotecas y frameworks especializados, así como su capacidad para gestionar todas las etapas del desarrollo de un modelo, desde el preprocesamiento de los datos hasta la implementación y visualización de resultados.

Estas características han hecho de Python el lenguaje preferido para el desarrollo de proyectos de *Machine Learning*, tanto en investigación como en la industria.

Para comenzar a programar en Python y usar Jupyter, sigue estos pasos:

Instalación de Python

link.ilerna.es/zzr1

1. **Descargar Python:** vamos al sitio web oficial de Python (https://www.python.org/downloads/) y descargamos la última versión compatible con nuestro sistema operativo.
2. **Instalar Python:** durante la instalación, nos aseguramos de marcar la opción Add Python to PATH para poder ejecutar Python desde cualquier lugar en la terminal.

Para **verificar la instalación** debemos abrir una terminal y ejecutar:

```
python --version
```

(Esto mostrará la versión instalada de Python).

Instalación de Jupyter Notebooks

Instalar pip (el gestor de paquetes de Python): si no está instalado con Python, podemos instalarlo con el siguiente comando:

```
python -m ensurepip --upgrade
```

Instalar Jupyter: ejecutar el siguiente comando para instalar Jupyter usando pip:

```
pip install jupyter
```

Iniciar Jupyter: para iniciar Jupyter Notebooks, ejecutar:

```
jupyter notebook
```

(Esto abrirá una interfaz en tu navegador donde podremos crear y ejecutar cuadernos Jupyter para programar en Python).

Cómo funciona la sintaxis de Python

Python tiene una sintaxis muy intuitiva y clara. Aquí se muestran algunos ejemplos básicos:

- **Variables y tipos de datos:**

```
# Asignación de variables
x = 5       # Entero
y = 3.14    # Flotante
nombre = "Python"  # Cadena

# Imprimir variables
print(x, y, nombre)
```

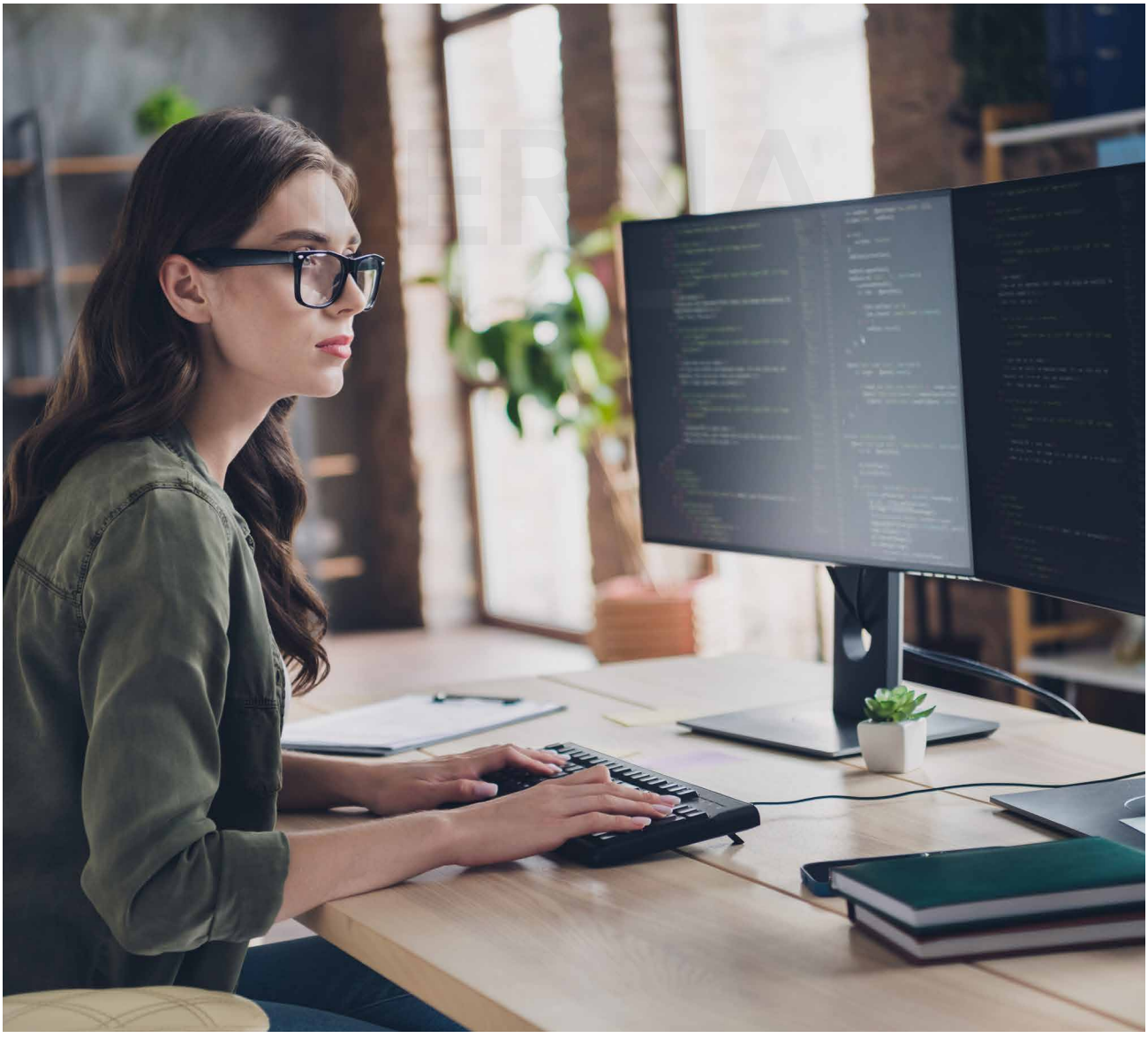
- **Condicionales y bucles:**

```
# Condicional
if x > 3:
    print("x es mayor que 3")
```

```
else:
    print(“x no es mayor que 3”)

# Bucle for
for i in range(5):
    print(i)
```

- **Funciones:**

```
# Definir una función
def suma(a, b):
    return a + b

# Llamar a la función
resultado = suma(5, 10)
print(resultado)
```



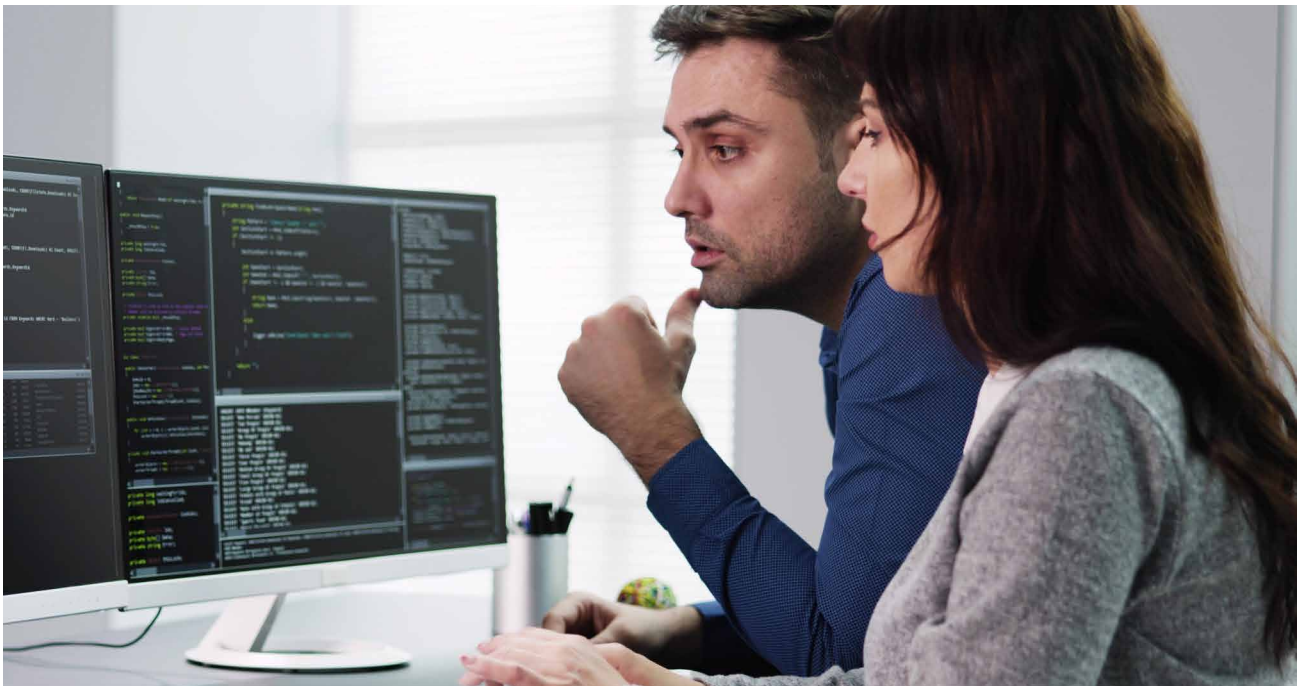
Ejercicios en Python

Ejercicio 1

Crear una calculadora con 3 operaciones sencillas.

Vamos a desarrollar una calculadora que permita sumar, restar y multiplicar dos números. Aquí vemos el código:

```
# Función para sumar
def sumar(a, b):
    return a + b

# Función para restar
def restar(a, b):
    return a - b

# Función para multiplicar
def multiplicar(a, b):
    return a * b

# Interfaz de usuario
def calculadora():
    print("Bienvenido a la calculadora")
    print("1: Sumar")
    print("2: Restar")
    print("3: Multiplicar")
    opcion = int(input("Selecciona una operación
(1/2/3): "))

    num1 = float(input("Ingresa el primer número:
"))
    num2 = float(input("Ingresa el segundo número:
"))

    if opcion == 1:
        print("Resultado:", sumar(num1, num2))
    elif opcion == 2:
        print("Resultado:", restar(num1, num2))
    elif opcion == 3:
        print("Resultado:", multiplicar(num1,
num2))
    else:
        print("Operación no válida")

# Ejecutar calculadora
calculadora()
```



Ejercicio 2

Trabajo con una librería gráfica para mostrar el resultado de una ecuación.

Para mostrar el gráfico de una función matemática, utilizaremos la librería **Matplotlib**. Podemos instalarla ejecutando pip install matplotlib.

Aquí tenemos el código para graficar la ecuación y= x3 + 2x2 – 1

```
import matplotlib.pyplot as plt
import numpy as np

# Crear el rango de valores de x
x = np.linspace(-10, 10, 100)

# Definir la ecuación
y = (x**3) + (2 * x**2) - 1

# Graficar
plt.plot(x, y, label='y = x^3 + 2x^2 - 1')
plt.title('Gráfico de la ecuación')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

Ejercicio 3

Crear la clase base Figura, clases hijas Rectángulo y Círculo, y mostrar las figuras por pantalla.

Aquí vamos a trabajar con la programación orientada a objetos (POO) en Python. Crearemos una clase base Figura, y luego las clases Rectangulo y Circulo, que heredarán de Figura. También usaremos la librería **Matplotlib** para mostrar las figuras gráficamente.

```
import matplotlib.pyplot as plt
import numpy as np

# Clase base Figura
class Figura:
    def __init__(self, nombre):
        self.nombre = nombre

    def area(self):
        pass

# Clase Rectangulo
class Rectangulo(Figura):
    def __init__(self, ancho, alto):
        super().__init__("Rectángulo")
        self.ancho = ancho
        self.alto = alto

    def area(self):
        return self.ancho * self.alto

    def dibujar(self):
        fig, ax = plt.subplots()
        rect = plt.Rectangle((0, 0), self.ancho, self.alto, fill=None, ed-
gecolor='r')
        ax.add_patch(rect)
        ax.set_xlim(-1, self.ancho + 1)
        ax.set_ylim(-1, self.alto + 1)
        plt.title(f"{self.nombre} - Área: {self.area()}")
        plt.show()

# Clase Círculo
class Circulo(Figura):
    def __init__(self, radio):
        super().__init__("Círculo")
        self.radio = radio

    def area(self):
        return np.pi * self.radio**2

    def dibujar(self):
        fig, ax = plt.subplots()
        circulo = plt.Circle((0, 0), self.radio, fill=None, edgecolor='b')
        ax.add_patch(circulo)
        ax.set_xlim(-self.radio-1, self.radio+1)
        ax.set_ylim(-self.radio-1, self.radio+1)
        plt.title(f"{self.nombre} - Área: {self.area():.2f}")
        plt.show()

# Crear y mostrar figuras
rect = Rectangulo(5, 3)
rect.dibujar()

circulo = Circulo(4)
circulo.dibujar()
```


To Do
In Progress
Testing
Work Item1
Work Item2
31%
28%
56%
65%
56%
75%

Bibliografía / webgrafía

Libros

Programación

- Abelson, H., & Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs* (2nd ed.). The MIT Press. ISBN: 978-0262510875
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). The MIT Press. ISBN: 978-0262033848
- Knuth, D. E. (1997). *The Art of Computer Programming*. Addison-Wesley. ISBN: 978-0201896831
- Martin, R. C. (2009). *Código Limpio: Manual de estilo para el desarrollo ágil de software*. Anaya Multimedia. ISBN: 978-8441532106

Programación en Java

- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. ISBN: 978-0134685991
- Schildt, H. (2019). *Java: The Complete Reference* (11th ed.). McGraw-Hill Education. ISBN: 978-1260440232
- Sierra, K., & Bates, B. (2005). *Head First Java* (2nd ed.). O'Reilly Media. ISBN: 978-0596009205

Programación en Python

- Matthes, E. (2019). *Python Crash Course: A Hands-On, Project-Based Introduction to Programming* (2nd ed.). No Starch Press. ISBN: 978-1593279288
- Ramalho, L. (2015). *Fluent Python: Clear, Concise, and Effective Programming*. O'Reilly Media. ISBN: 978-1491946008

Optimización del código

- Gorelick, M., & Ozsvald, I. (2020). *High Performance Python: Practical Performant Programming for Humans* (2nd ed.). O'Reilly Media. ISBN: 978-1492055020
- McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction* (2nd ed.). Microsoft Press. ISBN: 978-0735619678

Recursos online

- Codecademy. <https://www.codecademy.com>
- Coursera. <https://www.coursera.org>

- edX. <https://www.edx.org>
- freeCodeCamp. <https://www.freecodecamp.org>
- HackerRank. <https://www.hackerrank.com>
- Khan Academy. <https://www.khanacademy.org>
- LeetCode. <https://leetcode.com>
- Pluralsight. <https://www.pluralsight.com>
- Udemy. <https://www.udemy.com>
- W3Schools. <https://www.w3schools.com>

Referentes online

- Ariane Jurado de Bilbao. <https://twitter.com/Ari_Reinventada>
- Brais Moure. <https://mouredev.com/>
- Miguel Ángel Durán. <https://www.youtube.com/@midulive>
- Píldoras informáticas. <https://www.pildorasinformaticas.es/>