Guía de Formación en JAVA para Trainees

Desde los Fundamentos hasta el Desarrollo Empresarial Moderno

Alejandro G. Vera

Guía Exhaustiva de Formación en Java para Trainees: Desde los Fundamentos hasta el Desarrollo Empresarial Moderno

Alejandro G Vera

Introducción: El Ecosistema Java y su Propósito

El lenguaje de programación Java, desde su concepción, ha demostrado una capacidad de adaptación y una resiliencia que lo han mantenido como una de las tecnologías más relevantes en el desarrollo de software durante décadas. Para un desarrollador en formación, comprender no solo la sintaxis, sino la filosofía y la arquitectura que sustentan este ecosistema, es fundamental para dominarlo verdaderamente. Este informe proporciona una hoja de ruta exhaustiva, desde los conceptos básicos de Core Java hasta las complejidades de los frameworks empresariales y las prácticas de desarrollo modernas.

Un Vistazo a la Historia y Filosofía de Java

Java nació en 1991 en Sun Microsystems bajo el nombre de "Oak", concebido por un equipo liderado por James Gosling. Su objetivo inicial era modesto pero visionario: crear un lenguaje de programación para la nueva generación de dispositivos electrónicos de consumo, como los decodificadores de televisión digital por cable.[1] Este mercado inicial no prosperó como se esperaba, pero el fracaso comercial del prototipo, un control remoto universal llamado Star Seven, no detuvo el avance del lenguaje subyacente.

El verdadero genio de Java residía en su filosofía de diseño, encapsulada en cinco objetivos principales [1]:

1. Debería usar el paradigma de la programación orientada a objetos.
2. Debería permitir la ejecución de un mismo programa en múltiples sistemas operativos.
3. Debería tener soporte de red informático integrado.
4. Debería permitir la ejecución segura de código desde fuentes remotas.
5. Debería ser fácil de aprender y usar.

El segundo objetivo, la capacidad de ejecutarse en múltiples plataformas, se convirtió en su mantra más famoso: "Write Once, Run Anywhere" (WORA), que se traduce como "Escríbelo una vez, ejecútalo en cualquier lugar". Esta promesa de portabilidad fue posible gracias a la creación de una máquina virtual propia, que abstraía las

particularidades de cada sistema operativo.[1]

La adaptabilidad inherente a la filosofía WORA fue la clave de su éxito monumental. Cuando el mercado de la televisión interactiva no se materializó, Java estaba perfectamente posicionado para la siguiente gran revolución tecnológica: la World Wide Web. Los applets de Java, pequeños programas que podían ejecutarse de forma segura dentro de un navegador web sin importar el sistema operativo del usuario, se convirtieron en la primera demostración masiva del poder de la portabilidad. A partir de ahí, la robustez y escalabilidad de Java lo llevaron a dominar el desarrollo de aplicaciones del lado del servidor, y más tarde, se convirtió en el lenguaje principal para el desarrollo de aplicaciones en Android, el sistema operativo móvil más utilizado del mundo. El nombre "Java", según cuenta la leyenda, fue elegido durante una pausa para el café, en honor a la isla de Indonesia famosa por su producción cafetera, lo que explica el icónico logotipo de la taza de café.[1]

Esta trayectoria histórica ofrece una lección crucial para el desarrollador en formación: la fortaleza de una tecnología no reside en su aplicación a un mercado específico, sino en la solidez y la visión de sus principios de diseño. La portabilidad, la seguridad y la simplicidad de Java le permitieron sobrevivir a su propósito original y prosperar en ecosistemas tecnológicos que sus creadores apenas podían imaginar.

La Maquinaria de Java: Desglose de JDK, JRE y JVM

Para entender cómo Java logra su promesa de portabilidad, es esencial descomponer su plataforma en tres componentes fundamentales: la Máquina Virtual de Java (JVM), el Entorno de Ejecución de Java (JRE) y el Kit de Desarrollo de Java (JDK). La confusión entre estos tres elementos es común entre los principiantes, pero comprender su jerarquía y sus roles específicos es el primer paso para dominar el lenguaje.

- **JVM (Java Virtual Machine - Máquina Virtual de Java):** La JVM es el corazón de la plataforma Java y la pieza clave que hace posible el principio WORA. No es un componente físico, sino una especificación para una máquina abstracta que puede ejecutar bytecode de Java. Cuando se compila un programa Java, el código fuente (.java) no se convierte directamente en código máquina nativo del sistema operativo, sino en un formato intermedio llamado bytecode (.class). La JVM, que debe estar instalada en el sistema de destino, interpreta este bytecode

y lo traduce a instrucciones nativas para el hardware y el sistema operativo específicos en tiempo de ejecución. Este proceso de dos pasos (compilación a bytecode y luego interpretación/compilación JIT por la JVM) es lo que permite que un mismo archivo .class se ejecute sin cambios en Windows, macOS o Linux, siempre que cada uno tenga su propia implementación de la JVM. Además de la ejecución de código, la JVM gestiona tareas críticas como la administración de memoria y la recolección de basura (Garbage Collection).

- **JRE (Java Runtime Environment - Entorno de Ejecución de Java):** El JRE es el paquete de software que se necesita para ejecutar una aplicación Java. Es la implementación concreta de la JVM para una plataforma específica. Contiene la JVM y las bibliotecas de clases estándar de Java (Java Class Library), que son un conjunto de código preescrito que los desarrolladores pueden utilizar (por ejemplo, para manejar colecciones, redes, etc.).[1] Un usuario final que solo quiere ejecutar un programa Java (como Minecraft) solo necesita instalar el JRE. No se "instala la JVM" de forma aislada; se instala una JRE que contiene una JVM.
- **JDK (Java Development Kit - Kit de Desarrollo de Java):** El JDK es el paquete completo necesario para desarrollar aplicaciones Java. Es un superconjunto del JRE, lo que significa que incluye todo lo que contiene el JRE (la JVM y las bibliotecas) y, además, las herramientas de desarrollo esenciales. Estas herramientas incluyen:
    - javac: El compilador, que convierte el código fuente .java en bytecode .class.
    - jdb: El depurador de Java.
    - javadoc: El generador de documentación a partir de comentarios en el código.
    - Otras utilidades para empaquetar y monitorizar aplicaciones.

En resumen, la relación es una jerarquía de roles: la JVM traduce, el JRE ejecuta y el JDK desarrolla. Un desarrollador escribe código utilizando el JDK. El compilador del JDK convierte el código en bytecode. Este bytecode puede ser distribuido y ejecutado en cualquier máquina que tenga instalado el JRE, porque la JVM dentro de ese JRE se encargará de la traducción final. Esta separación de responsabilidades es clave para entender conceptos posteriores como la compilación, el empaquetado y el despliegue de aplicaciones.

Ediciones de Java: Un Mundo de Posibilidades

El ecosistema Java se ha diversificado para satisfacer diferentes necesidades de

desarrollo, dando lugar a varias ediciones. Las dos más importantes para este informe son:

- **Java SE (Standard Edition):** Es la plataforma central y el fundamento de todo el ecosistema Java. Proporciona las APIs (Interfaces de Programación de Aplicaciones) y bibliotecas fundamentales para el desarrollo de aplicaciones de escritorio, de servidor y de consola. Los conceptos de Core Java que se explorarán en la Parte I de este informe pertenecen a Java SE.
- **Jakarta EE (Enterprise Edition):** Anteriormente conocida como Java Platform, Enterprise Edition (Java EE), esta es una extensión de Java SE. Jakarta EE define un conjunto de especificaciones y APIs diseñadas para construir aplicaciones empresariales a gran escala, distribuidas, seguras y transaccionales.[1] Proporciona una infraestructura para servicios como servlets web, persistencia de datos y mensajería, que se detallarán en la Parte II.

Con esta base conceptual, estamos listos para sumergirnos en el núcleo del lenguaje y construir, paso a paso, el conocimiento necesario para convertirnos en desarrolladores Java competentes.

Parte I: Dominio de Core Java

Esta primera parte del informe se centra en los pilares fundamentales de Java SE. Dominar estos conceptos es un requisito indispensable antes de aventurarse en el desarrollo de aplicaciones complejas con frameworks. Construiremos el conocimiento desde la sintaxis más básica hasta los conceptos avanzados de la programación moderna.

Sección 1: Los Cimientos del Lenguaje

Aquí exploraremos la gramática y las estructuras elementales que conforman cualquier programa en Java.

Sintaxis Fundamental: Escribiendo Código Legible

La sintaxis de un lenguaje de programación es su conjunto de reglas para formar sentencias válidas. Java, inspirado en C++, tiene una sintaxis que es a la vez potente y estructurada, pero que exige precisión.

- **Estructura de un Programa Java:** Todo en Java vive dentro de una clase. Un programa Java se compone de uno o más archivos con la extensión .java. Por convención y, en muchos casos, por requisito del compilador, el nombre del archivo debe coincidir exactamente con el nombre de la clase pública que contiene.[1] Las clases se agrupan lógicamente en paquetes (packages), que se corresponden con una estructura de directorios en el sistema de archivos. Esto ayuda a organizar proyectos grandes y a evitar conflictos de nombres.[1]
- **El Método main:** El punto de entrada de cualquier aplicación Java autónoma es el método main. Su firma es siempre public static void main(String args).[1]
    - public: Es un modificador de acceso que indica que el método puede ser llamado desde cualquier lugar, lo cual es necesario para que la JVM pueda iniciarlo.
    - static: Significa que el método pertenece a la clase en sí, no a una instancia específica (objeto) de la clase. Esto permite a la JVM llamar al método sin tener que crear primero un objeto.
    - void: Indica que el método no devuelve ningún valor.
    - main: Es el nombre que la JVM busca por convención para iniciar la ejecución.
    - String args: Es un parámetro que permite pasar argumentos de línea de comandos al programa como un array de cadenas de texto.

Ejemplo: Estructura Básica de un Programa

Java

```
// El paquete organiza las clases en un espacio de nombres.
// Corresponde a una estructura de directorios: com/ejemplo/
package com.ejemplo;

// El nombre del archivo DEBE ser HolaMundo.java
public class HolaMundo {
```

```
    // Este es el método main, el punto de entrada de la aplicación.
    public static void main(String args) {
        // Imprime un mensaje en la consola.
        System.out.println("¡Hola, Mundo!");
    }
}
```

- **Convenciones de Nomenclatura:** Para que el código sea legible y mantenible, es crucial seguir las convenciones de nomenclatura de Java [1]:
    - **Clases e Interfaces:** Usan UpperCamelCase (o PascalCase), donde la primera letra de cada palabra está en mayúscula (ej. MiClase, CalculadoraDeImpuestos).
    - **Métodos y Variables:** Usan lowerCamelCase, donde la primera letra de la primera palabra está en minúscula y las siguientes palabras comienzan con mayúscula (ej. miVariable, calcularImpuesto()).
    - **Constantes:** Usan SCREAMING_SNAKE_CASE, todo en mayúsculas y palabras separadas por guiones bajos (ej. TASA_DE_INTERES_MAXIMA).

Ejemplo: Convenciones de Nomenclatura

Java

```
// El nombre de la clase sigue la convención UpperCamelCase.
public class CalculadoraDeImpuestos {

    // El nombre de la constante sigue la convención SCREAMING_SNAKE_CASE.
    public static final double TASA_IVA_GENERAL = 0.21;

    // El nombre de la variable sigue la convención lowerCamelCase.
    private double baseImponible;

    // El nombre del método sigue la convención lowerCamelCase.
    public double calcularImpuestoFinal(double precioNeto) {
        // Lógica del método...
        return precioNeto * TASA_IVA_GENERAL;
    }
}
```

- **Bloques de Código y Puntos y Coma:** Java utiliza llaves {} para delimitar bloques de código, como el cuerpo de una clase o un método. Cada instrucción de código individual debe terminar con un punto y coma ;.[1] La omisión de un punto y coma es uno de los errores de compilación más comunes para los principiantes.
- **Comentarios:** Son esenciales para documentar el código. Java soporta tres tipos[1]:
  - Comentario de una sola línea: // Esto es un comentario.
  - Comentario de múltiples líneas: /* Esto es un comentario que puede ocupar varias líneas */.
  - Comentario de Javadoc: /** Esto es un comentario de documentación. Puede ser procesado por la herramienta javadoc para generar documentación HTML. */.

Variables y Tipos de Datos: El Almacén de la Información

Las variables son contenedores que almacenan datos. En Java, un lenguaje de tipado estático, cada variable debe ser declarada con un tipo de dato específico antes de poder ser utilizada.[1]

- **Declaración e Inicialización:** Se declara una variable especificando su tipo y nombre (int edad;). Se le puede asignar un valor en la misma línea (inicialización: int edad = 30;) o más tarde (edad = 31;).[1]

Ejemplo: Declaración e Inicialización de Variables

Java

```
public class DemoVariables {
    public static void main(String args) {
        // Declaración de una variable de tipo entero.
        int numeroDeEstudiantes;

        // Inicialización de la variable.
```

```
        numeroDeEstudiantes = 25;

        // Declaración e inicialización en una sola línea.
        double precioProducto = 19.99;

        // Declaración de una variable para almacenar texto.
        String nombreCliente = "Ana García";

        // Imprimir los valores de las variables.
        System.out.println("Número de estudiantes: " + numeroDeEstudiantes);
        System.out.println("Precio: " + precioProducto);
        System.out.println("Cliente: " + nombreCliente);
    }
}
```

- **Tipos de Datos Primitivos:** Son los tipos de datos más básicos y fundamentales del lenguaje. No son objetos y se almacenan directamente en la memoria (en el stack, para variables locales), lo que los hace muy eficientes.[1]

Tipo	Descripción	Tamaño en Memoria	Rango de Valores	Valor por Defecto
byte	Entero de 8 bits con signo	1 byte	-128 a 127	0
short	Entero de 16 bits con signo	2 bytes	-32,768 a 32,767	0
int	Entero de 32 bits con signo	4 bytes	-2,147,483,648 a 2,147,483,647	0
long	Entero de 64 bits con signo	8 bytes	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	0L
float	Punto flotante de	4 bytes	Aprox. ±3.40282347	0.0f

	precisión simple (32 bits)		E+38F		
double	Punto flotante de precisión doble (64 bits)	8 bytes	Aprox. ±1.797693134 86231570E+3 08	0.0d	
char	Carácter Unicode de 16 bits sin signo	2 bytes	0 a 65,535 (\u0000 a \uffff)	\u0000	
boolean	Valor lógico	~1 bit (depende de la JVM)	true o false	false	
Fuentes: [1]					

Ejemplo: Uso de Tipos Primitivos

Java

```
public class DemoPrimitivos {
    public static void main(String args) {
        int miEntero = 100;
        // La 'L' al final es necesaria para literales long.
        long poblacionMundial = 8000000000L;
        // La 'f' al final es necesaria para literales float.
        float piAproximado = 3.14f;
        double numeroEuler = 2.718281828459045;
        char inicial = 'A';
        boolean esJavaDivertido = true;

        System.out.println("Entero: " + miEntero);
        System.out.println("Long: " + poblacionMundial);
        System.out.println("Float: " + piAproximado);
        System.out.println("Double: " + numeroEuler);
```

```
        System.out.println("Char: " + inicial);
        System.out.println("Boolean: " + esJavaDivertido);
    }
}
```

- **Tipos de Datos por Referencia (No Primitivos):** A diferencia de los primitivos, estos tipos no almacenan el dato en sí, sino una referencia (una dirección de memoria) que apunta a un objeto almacenado en otra área de la memoria llamada heap.[1] Los tipos de referencia más comunes son:
    - **Clases:** Como la omnipresente clase String para secuencias de caracteres, o cualquier clase que el desarrollador cree.[1]
    - **Arrays:** Colecciones de tamaño fijo de elementos del mismo tipo, ya sean primitivos o de referencia.[1]
    - **Interfaces:** Contratos que definen un conjunto de métodos, que se explorarán más adelante.

La distinción entre tipos primitivos y de referencia no es meramente académica; tiene profundas consecuencias en el comportamiento del programa. No comprender esta diferencia es la causa raíz de muchos de los errores más comunes y frustrantes para los desarrolladores principiantes.

1. **Manejo de Memoria:** Las variables locales de tipo primitivo viven en una zona de memoria rápida llamada stack. Cuando el método termina, la variable desaparece. Los objetos (a los que apuntan las variables de referencia) viven en una zona de memoria más grande y dinámica llamada heap. La variable de referencia en el stack solo contiene la "dirección" del objeto en el heap.
2. **Paso de Parámetros:** Cuando se pasa una variable primitiva a un método, se pasa una copia de su valor (paso por valor). Cualquier cambio dentro del método no afecta a la variable original. Cuando se pasa una variable de referencia, también se pasa una copia, pero es una copia de la referencia (la dirección). Esto significa que tanto la variable original como el parámetro del método apuntan al mismo objeto en el heap. Por lo tanto, si el método modifica el estado de ese objeto, el cambio será visible fuera del método.

Ejemplo: Paso por Valor (Primitivos) vs. Paso por Referencia (Objetos)

Java

```
public class DemoPasoDeParametros {

    public static void modificarPrimitivo(int numero) {
        // Se modifica la copia local del valor.
        numero = 100;
        System.out.println("Dentro de modificarPrimitivo, numero es: " + numero);
    }

    public static void modificarReferencia(int array) {
        // Se modifica el objeto al que apunta la referencia.
        array = 100;
        System.out.println("Dentro de modificarReferencia, array es: " + array);
    }

    public static void main(String args) {
        int miNumero = 10;
        int miArray = {10, 20, 30};

        System.out.println("Antes de llamar, miNumero es: " + miNumero);
        modificarPrimitivo(miNumero);
        // El valor original del primitivo NO cambia.
        System.out.println("Después de llamar, miNumero es: " + miNumero);

        System.out.println("\nAntes de llamar, miArray es: " + miArray);
        modificarReferencia(miArray);
        // El contenido del objeto referenciado SÍ cambia.
        System.out.println("Después de llamar, miArray es: " + miArray);
    }
}
```

3. **Comparación de Igualdad:** El operador == se comporta de manera diferente. Para los primitivos, == compara si los valores son idénticos.[1] Para las referencias, == compara si las direcciones de memoria son idénticas, es decir, si ambas variables apuntan exactamente al mismo objeto. Para comparar el contenido de dos objetos (por ejemplo, dos Strings), se debe usar el método .equals().[1]

Ejemplo: Comparación con == vs. .equals()

Java

```
public class DemoComparacion {
    public static void main(String args) {
        String str1 = new String("Java");
        String str2 = new String("Java");
        String str3 = str1;

        // Compara las referencias (direcciones de memoria).
        // str1 y str2 son dos objetos diferentes, por lo que sus direcciones son distintas.
        System.out.println("str1 == str2: " + (str1 == str2)); // false

        // Compara el contenido de los objetos.
        System.out.println("str1.equals(str2): " + str1.equals(str2)); // true

        // str3 apunta al mismo objeto que str1.
        System.out.println("str1 == str3: " + (str1 == str3)); // true
    }
}
```

4. **Valor null:** Una variable de referencia puede tener un valor especial, null, que significa que no apunta a ningún objeto. Intentar usar un método o campo en una referencia null provoca el error en tiempo de ejecución más famoso de Java: NullPointerException.[1] Las variables primitivas no pueden ser null.
5. **Clases Envoltorio (Wrapper Classes):** Para cerrar esta brecha, Java proporciona clases envoltorio para cada tipo primitivo (ej. Integer para int, Double para double).[1] Estas clases permiten tratar a los valores primitivos como objetos, lo cual es esencial para usarlos en colecciones genéricas como List<Integer>, que no pueden almacenar tipos primitivos directamente.

Ejemplo: Uso de Clases Envoltorio en Colecciones

Java

```
import java.util.ArrayList;
```

```java
import java.util.List;

public class DemoWrapper {
    public static void main(String args) {
        // Una lista no puede almacenar tipos primitivos como 'int'.
        // Se debe usar su clase envoltorio 'Integer'.
        List<Integer> listaDeNumeros = new ArrayList<>();

        // Autoboxing: Java convierte automáticamente el primitivo 'int' a un objeto 'Integer'.
        listaDeNumeros.add(10);
        listaDeNumeros.add(20);
        listaDeNumeros.add(30);

        System.out.println("Lista de enteros: " + listaDeNumeros);

        // Unboxing: Java convierte automáticamente el objeto 'Integer' a un primitivo 'int'.
        int primerNumero = listaDeNumeros.get(0);
        System.out.println("Primer número (primitivo): " + primerNumero);
    }
}
```

Operadores en Java: Las Herramientas de Manipulación

Los operadores son símbolos especiales que realizan operaciones sobre variables y valores.[1] Java ofrece un rico conjunto de operadores:

- **Operadores Aritméticos:** Realizan operaciones matemáticas básicas: + (suma), - (resta), * (multiplicación), / (división) y % (módulo o resto de la división).[1] El operador
  + también se utiliza para concatenar (unir) cadenas de texto (String).[1]
- **Operadores de Asignación:** Asignan valores a las variables. El más simple es =, pero existen operadores compuestos como +=, -=, *=, que combinan una operación aritmética con la asignación (ej. $x += 5$ es equivalente a $x = x + 5$)).[1]
- **Operadores Unarios:** Operan sobre un único operando. Incluyen ++ (incremento), -- (decremento), ! (negación lógica), y los signos + y - para indicar valores positivos o negativos.[1]
- **Operadores Relacionales (de Comparación):** Comparan dos valores y

devuelven un resultado booleano (true o false). Son == (igual a), != (distinto de), > (mayor que), < (menor que), >= (mayor o igual que), y <= (menor o igual que).[1]

- **Operadores Lógicos:** Combinan expresiones booleanas. Son && (AND lógico), || (OR lógico) y ! (NOT lógico).[1] Los operadores
  && y || realizan una "evaluación de cortocircuito": si el resultado de la expresión se puede determinar evaluando solo el primer operando, el segundo no se evalúa. Por ejemplo, en false && algunaFuncion(), algunaFuncion() nunca se ejecutará.[1]
- **Operadores a nivel de bits (Bitwise):** Realizan operaciones directamente sobre la representación binaria de los números enteros: & (AND), | (OR), ^ (XOR), ~ (NOT), << (desplazamiento a la izquierda), >> (desplazamiento a la derecha con signo), >>> (desplazamiento a la derecha sin signo).[1] Son útiles en programación de bajo nivel y optimización.
- **Operador Ternario:** Es una forma compacta de la sentencia if-else. Su sintaxis es condicion? valor_si_true : valor_si_false.[1]

Es importante conocer la precedencia de operadores, que determina el orden en que se evalúan en una expresión compleja. Por ejemplo, la multiplicación y la división se realizan antes que la suma y la resta. En caso de duda, siempre es una buena práctica usar paréntesis () para forzar un orden de evaluación explícito y mejorar la legibilidad del código.[1]

Estructuras de Control de Flujo: Dirigiendo la Ejecución

Las estructuras de control permiten alterar el flujo secuencial de un programa, tomando decisiones y repitiendo acciones.[1]

- **Estructuras Condicionales (Toma de Decisiones):**
  - if, else if, else: Es la estructura de decisión más fundamental. El bloque if se ejecuta si una condición es true. Opcionalmente, se pueden añadir uno o más bloques else if para probar condiciones adicionales, y un bloque else final que se ejecuta si ninguna de las condiciones anteriores es true.[1]

Ejemplo: Estructura if-else if-else

Java

```java
public class DemoCondicional {
    public static void main(String args) {
        int nota = 75;

        if (nota >= 90) {
            System.out.println("Sobresaliente");
        } else if (nota >= 70) {
            System.out.println("Notable");
        } else if (nota >= 50) {
            System.out.println("Aprobado");
        } else {
            System.out.println("Suspenso");
        }
    }
}
```

* `switch`: Proporciona una alternativa más limpia a una cadena larga de `if-else if` cuando se compara una única variable contra múltiples valores constantes.[1] Cada `case` representa un valor a comparar. La sentencia `break` es crucial para salir del `switch` una vez que se encuentra una coincidencia; sin ella, la ejecución "caería" (fall-through) a los siguientes `case`. El bloque `default` es opcional y se ejecuta si ningún `case` coincide. Versiones más recientes de Java (desde la 12) han introducido las "expresiones switch" mejoradas, que son más concisas, no requieren `break` y pueden devolver un valor, simplificando aún más el código.[1]

Ejemplo: Estructura switch

Java

```java
public class DemoSwitch {
    public static void main(String args) {
        int diaDeLaSemana = 3; // 1=Lunes, 2=Martes,...
```

```
        String nombreDelDia;

        switch (diaDeLaSemana) {
            case 1:
                nombreDelDia = "Lunes";
                break; // El 'break' es crucial para evitar que la ejecución continúe.
            case 2:
                nombreDelDia = "Martes";
                break;
            case 3:
                nombreDelDia = "Miércoles";
                break;
            case 4:
                nombreDelDia = "Jueves";
                break;
            case 5:
                nombreDelDia = "Viernes";
                break;
            default:
                nombreDelDia = "Fin de semana";
                break;
        }
        System.out.println("Hoy es " + nombreDelDia);
    }
}
```

- **Estructuras Repetitivas (Bucles):**
  - for: Es ideal cuando se conoce de antemano el número de iteraciones. Su cabecera consta de tres partes: una inicialización (se ejecuta una vez al principio), una condición (se evalúa antes de cada iteración) y una actualización (se ejecuta después de cada iteración).[1]

Ejemplo: Bucle for

Java

```
public class DemoFor {
```

```
    public static void main(String args) {
        // Imprime los números del 1 al 5.
        // inicialización: int i = 1
        // condición: i <= 5
        // actualización: i++
        for (int i = 1; i <= 5; i++) {
            System.out.println("Iteración número: " + i);
        }
    }
}
```

* **Bucle `for-each` (Enhanced for loop):** Una sintaxis simplificada del bucle `for` para iterar sobre los elementos de un array o una colección sin necesidad de manejar un índice. Su formato es `for (Tipo elemento : coleccion) {... }`. Es menos propenso a errores y más legible.

Ejemplo: Bucle for-each

Java

```
public class DemoForEach {
    public static void main(String args) {
        String frutas = {"Manzana", "Naranja", "Plátano"};

        // Itera sobre cada elemento del array 'frutas'.
        for (String fruta : frutas) {
            System.out.println("Me gusta la " + fruta);
        }
    }
}
```

* `while`: Repite un bloque de código mientras una condición permanezca `true`. La condición se comprueba antes de cada iteración, por lo que es posible que el cuerpo del bucle no se ejecute nunca si la condición es inicialmente `false`.[1]

Ejemplo: Bucle while

Java

```
public class DemoWhile {
    public static void main(String args) {
        int contador = 5;

        // El bucle se ejecuta mientras 'contador' sea mayor que 0.
        while (contador > 0) {
            System.out.println("Cuenta atrás: " + contador);
            contador--; // Es crucial actualizar la variable de control para evitar un bucle infinito.
        }
    }
}
```

* `do-while`: Similar al `while`, pero la condición se comprueba después de cada iteración. Esto garantiza que el cuerpo del bucle se ejecute al menos una vez, independientemente del valor inicial de la condición.[1]

Ejemplo: Bucle do-while

Java

```java
public class DemoDoWhile {
    public static void main(String args) {
        int contador = 5;

        // El cuerpo del bucle se ejecuta al menos una vez.
        do {
            System.out.println("El valor del contador es: " + contador);
            contador++;
        } while (contador < 5); // La condición (5 < 5) es falsa, pero el bucle ya se ejecutó una vez.
    }
}
```

- **Sentencias de Salto (Branching Statements):** Permiten un control más fino sobre el flujo dentro de los bucles y métodos.
    - break: Termina inmediatamente la ejecución del bucle (for, while, do-while) o switch más interno en el que se encuentra.[1]
    - continue: Salta el resto del código en la iteración actual del bucle y procede con la siguiente iteración.[1]
    - return: Finaliza la ejecución de un método. Puede devolver un valor si el método no es void.[1]
    - **Etiquetas (Labels):** En situaciones complejas con bucles anidados, se pueden usar etiquetas para que break o continue se apliquen a un bucle externo específico, en lugar del más interno.[1]

Ejemplo: Uso de break y continue

Java

```java
public class DemoSalto {
    public static void main(String args) {
        for (int i = 1; i <= 10; i++) {
            // Si el número es 7, se salta esta iteración y se continúa con la siguiente.
            if (i == 7) {
                continue;
            }
            // Si el número es mayor que 8, se termina el bucle por completo.
            if (i > 8) {
```

```
            break;
        }
        System.out.println("Número: " + i);
    }
    System.out.println("Fin del bucle.");
  }
}
```

Sección 2: El Paradigma Orientado a Objetos (POO)

La Programación Orientada a Objetos (POO) no es solo un conjunto de características de un lenguaje, sino un paradigma, una forma de pensar y estructurar el software. En lugar de centrarse en procedimientos y funciones, la POO se centra en los "objetos", que modelan entidades del mundo real o conceptos abstractos, agrupando datos y comportamiento en una sola unidad.[1] Java es un lenguaje fuertemente orientado a objetos, y dominar sus principios es esencial.

Clases y Objetos: De los Planos a la Realidad

La distinción entre clase y objeto es la idea más fundamental de la POO.

- **Clase:** Una clase es una plantilla o un plano (blueprint) que define las características (atributos) y los comportamientos (métodos) que tendrán los objetos de un cierto tipo.[1] Por ejemplo, podríamos definir una clase Coche que especifique que todos los coches tendrán un color, una marca y una velocidad, y que podrán acelerar y frenar. La clase en sí no es un coche; es la definición de lo que significa ser un coche.
- **Objeto:** Un objeto es una instancia concreta de una clase.[1] Si Coche es la clase, un Ford Focus rojo específico o un Tesla Model S blanco son objetos de esa clase. Cada objeto tiene su propio estado (el valor de sus atributos: color = "rojo", marca = "Ford"), pero comparte el mismo comportamiento (los métodos definidos en la clase Coche).

La creación de un objeto a partir de una clase se llama instanciación. En Java, esto se

hace con la palabra clave new, que asigna memoria para el nuevo objeto y llama a un constructor para inicializarlo.[1] Una vez creado el objeto, se accede a sus atributos y métodos usando el operador punto (

.).[1]

Java

```
// Definición de la clase (el plano)
public class Coche {
    // Campos (estado)
    String color;
    String marca;

    // Métodos (comportamiento)
    void acelerar() {
        System.out.println("El coche está acelerando...");
    }
}

// Creación y uso de objetos (las instancias)
public class Garaje {
    public static void main(String args) {
        // Instanciación de dos objetos Coche
        Coche miCoche = new Coche();
        Coche cocheDeVecino = new Coche();

        // Asignación de estado a cada objeto
        miCoche.marca = "Ford";
        miCoche.color = "Rojo";
        cocheDeVecino.marca = "Tesla";
        cocheDeVecino.color = "Blanco";

        // Llamada a un método del objeto
        miCoche.acelerar(); // Imprime "El coche está acelerando..."
    }
}
```

Métodos y Constructores: Comportamiento e Inicialización

- **Métodos:** Son los bloques de código que definen el comportamiento de un objeto. Un método tiene una firma (su nombre y la lista de parámetros que acepta) y un tipo de retorno (el tipo de valor que devuelve, o void si no devuelve nada).[1]
- **Constructores:** Son un tipo especial de método que se utiliza para inicializar un objeto cuando se crea. Sus características distintivas son [1]:
  1. Tienen exactamente el mismo nombre que la clase.
  2. No tienen tipo de retorno, ni siquiera void.
  3. Se invocan implícitamente con la palabra clave new.

Una clase puede tener múltiples constructores, siempre que sus listas de parámetros sean diferentes (esto se llama sobrecarga de constructores o constructor overloading).[1] Si un desarrollador no proporciona ningún constructor, el compilador de Java crea uno por defecto sin argumentos (el constructor por defecto). Dentro de un constructor, la palabra clave

this es especialmente útil. Se puede usar para referirse al objeto actual, desambiguando entre un campo de la clase y un parámetro con el mismo nombre (ej. this.nombre = nombre;). También se puede usar this() para llamar a otro constructor de la misma clase, lo que ayuda a evitar la duplicación de código de inicialización.[1]

Ejemplo: Constructores, Sobrecarga y la palabra clave this

Java

```
public class Persona {
    String nombre;
    int edad;

    // Constructor sin argumentos.
    // Llama a otro constructor de la misma clase usando this() para reutilizar código.
    public Persona() {
```

```
        this("Desconocido", 0); // Llama al constructor que recibe String e int
    }

    // Constructor con dos argumentos.
    public Persona(String nombre, int edad) {
        // 'this.nombre' se refiere al campo de la clase.
        // 'nombre' se refiere al parámetro del método.
        this.nombre = nombre;
        this.edad = edad;
    }

    public void presentarse() {
        System.out.println("Hola, soy " + this.nombre + " y tengo " + this.edad + " años.");
    }

    public static void main(String args) {
        Persona p1 = new Persona(); // Usa el constructor sin argumentos
        p1.presentarse(); // Imprime: Hola, soy Desconocido y tengo 0 años.

        Persona p2 = new Persona("Carlos", 30); // Usa el constructor con argumentos
        p2.presentarse(); // Imprime: Hola, soy Carlos y tengo 30 años.
    }
}
```

Modificadores de Acceso: Definiendo los Límites

Los modificadores de acceso son palabras clave que establecen el nivel de visibilidad (o accesibilidad) de una clase, un campo o un método. Son una herramienta fundamental para implementar la encapsulación.[1]

Modificador	Misma Clase	Mismo Paquete	Subclase (Otro Paquete)	Global (Otro Paquete)
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No

default (sin modificador)	Sí	Sí	No	No	
private	Sí	No	No	No	
Fuentes: [1]					

La mejor práctica, y un principio clave del buen diseño de software, es utilizar siempre el modificador de acceso más restrictivo posible que tenga sentido para un miembro. Por norma general, los campos de una clase casi siempre deben ser private para proteger el estado interno del objeto.[1]

Los Cuatro Pilares de la POO: Construyendo Software Robusto

Los cuatro pilares de la POO -encapsulación, herencia, polimorfismo y abstracción- no son conceptos aislados, sino un conjunto de principios interrelacionados que guían el diseño de software robusto, mantenible y escalable.[1]

1. Encapsulación (Ocultación de Datos)

La encapsulación consiste en agrupar los datos (campos) y los métodos que operan sobre esos datos en una sola unidad (la clase), y ocultar los detalles de la implementación interna del mundo exterior.[1] En la práctica, esto se logra declarando los campos como

private y proporcionando métodos public (conocidos como getters y setters) para acceder y modificar esos campos de forma controlada.[1]

El beneficio principal de la encapsulación es la protección de la integridad del objeto. Al controlar cómo se accede y se modifica el estado, se puede evitar que el objeto entre en un estado inválido. Por ejemplo, un método setEdad() podría incluir una validación para no permitir edades negativas. Además, permite cambiar la implementación interna de la clase sin afectar al código que la utiliza, siempre que la interfaz pública (los métodos public) permanezca igual.

Ejemplo: Encapsulación para Proteger Datos

Java

```
public class CuentaBancaria {
    // El campo 'saldo' es privado. Solo se puede acceder a él a través de los métodos.
    private double saldo;

    public CuentaBancaria(double saldoInicial) {
        this.saldo = saldoInicial;
    }

    // Getter público para leer el saldo de forma segura.
    public double getSaldo() {
        return this.saldo;
    }

    // Setter público para modificar el saldo de forma controlada.
    public void depositar(double cantidad) {
        if (cantidad > 0) {
            this.saldo += cantidad;
        } else {
            System.out.println("La cantidad a depositar debe ser positiva.");
        }
    }
}
```

2. Herencia (Reutilización de Código)

La herencia es un mecanismo que permite a una clase (la subclase) adquirir los atributos y métodos de otra clase (la superclase).[1] Modela una relación "es-un" (

is-a). Por ejemplo, un Perro es un Animal. En Java, se implementa con la palabra clave

extends.

La herencia es una herramienta poderosa para la reutilización de código, pero debe usarse con precaución. Un uso excesivo de la herencia puede llevar a jerarquías de clases frágiles y complejas. Esto se debe a que la herencia crea un acoplamiento muy fuerte: un cambio en la superclase puede tener efectos imprevistos y romper la funcionalidad de todas sus subclases.

Esto nos lleva a un principio de diseño de software fundamental: "Favorecer la composición sobre la herencia". La herencia es apropiada cuando existe una verdadera relación "es-un" entre las clases. Sin embargo, si la relación es "tiene-un" (has-a), como en "un Coche tiene un Motor", es mucho más flexible y robusto usar la composición. En la composición, la clase Coche contendría una instancia de la clase Motor como uno de sus campos. Este enfoque conduce a un diseño más modular y menos acoplado, ya que los componentes pueden ser reemplazados o modificados de forma independiente.

Ejemplo: Herencia con extends y super

Java

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

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

    public void comer() {
        System.out.println(nombre + " está comiendo.");
    }
}

// Subclase que hereda de Animal
class Perro extends Animal {
    // La subclase tiene su propio constructor.
    public Perro(String nombre) {
```

```
        // 'super(nombre)' llama al constructor de la superclase (Animal).
        // Es necesario para inicializar los campos heredados.
        super(nombre);
    }

    // La subclase puede tener sus propios métodos.
    public void ladrar() {
        System.out.println(nombre + " está ladrando: ¡Guau!");
    }
}
```

3. Polimorfismo (Muchas Formas)

La palabra polimorfismo significa "muchas formas". En POO, es la capacidad de un objeto de ser tratado como una instancia de su propia clase, de su superclase, o de cualquiera de las interfaces que implementa.[1]

El polimorfismo se manifiesta principalmente a través de la sobrescritura de métodos (method overriding). Esto ocurre cuando una subclase proporciona una implementación específica para un método que ya está definido en su superclase. La anotación @Override se utiliza para indicar al compilador la intención de sobrescribir un método, lo que ayuda a detectar errores.[1]

El gran beneficio del polimorfismo es que permite escribir código genérico y flexible. Por ejemplo, se puede tener un método que acepte un List<Animal> y llame al método hacerSonido() en cada elemento. Si se le pasa una lista que contiene objetos Perro y Gato, el método correcto (ladrar() o maullar()) se invocará para cada objeto en tiempo de ejecución, sin que el código cliente necesite saber el tipo específico de cada animal.

Ejemplo: Polimorfismo con Sobrescritura de Métodos

```
Java
```

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

class Perro extends Animal {
    @Override // Anotación que indica la sobrescritura del método
    public void hacerSonido() {
        System.out.println("El perro ladra: ¡Guau!");
    }
}

class Gato extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El gato maúlla: ¡Miau!");
    }
}

public class DemoPolimorfismo {
    public static void main(String args) {
        // Una variable de tipo Animal puede referenciar a un objeto Perro.
        Animal miMascota = new Perro();
        miMascota.hacerSonido(); // Se ejecuta el método de Perro.

        // O puede referenciar a un objeto Gato.
        miMascota = new Gato();
        miMascota.hacerSonido(); // Se ejecuta el método de Gato.
    }
}
```

4. Abstracción (Ocultación de Complejidad)

La abstracción consiste en ocultar los detalles complejos de la implementación y exponer solo la funcionalidad esencial y relevante al usuario.[1] Se centra en el "qué"

hace un objeto, no en el "cómo" lo hace. En Java, la abstracción se logra a través de dos mecanismos principales:

- **Clases Abstractas:** Una clase declarada con la palabra clave abstract no puede ser instanciada. Puede contener tanto métodos abstractos (sin cuerpo, solo la firma) como métodos concretos (con implementación). Una subclase debe implementar todos los métodos abstractos de su superclase abstracta, o bien ser declarada también como abstracta.
- **Interfaces:** Una interfaz es una forma pura de abstracción. Es un contrato que define un conjunto de métodos abstractos (y, desde Java 8, también métodos default y static con implementación). Una clase que implements una interfaz se compromete a proporcionar una implementación para todos sus métodos abstractos.[1]

La abstracción y el polimorfismo están intrínsecamente ligados. Una interfaz (abstracción) define un contrato, como List. El polimorfismo es lo que nos permite usar ese contrato (List miLista =...) con diferentes implementaciones (... = new ArrayList() o ... = new LinkedList()) de forma intercambiable, sin que el código que usa miLista se vea afectado.

Ejemplo: Clases Abstractas e Interfaces

Java

```
// Una interfaz define un contrato de comportamiento (qué puede hacer).
interface Dibujable {
    void dibujar(); // Todos los métodos son públicos y abstractos por defecto.
}

// Una clase abstracta puede tener estado y comportamiento, tanto concreto como abstracto.
abstract class Figura {
    String color;

    // Método abstracto: las subclases DEBEN implementarlo.
    public abstract double calcularArea();

    // Método concreto: las subclases lo heredan tal cual.
    public void describir() {
```

```
        System.out.println("Soy una figura de color " + color);
    }
}

// Una clase concreta que hereda de una clase abstracta e implementa una interfaz.
class Circulo extends Figura implements Dibujable {
    double radio;

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

    @Override
    public void dibujar() {
        System.out.println("Dibujando un círculo.");
    }
}
```

Sección 3: APIs Esenciales de Java

Más allá de la sintaxis y los principios de la POO, la verdadera potencia de Java reside en su rica biblioteca de clases estándar, que proporciona APIs para realizar una vasta gama de tareas comunes.

Manejo de Excepciones: Gestionando lo Inesperado

Una excepción es un evento anómalo que ocurre durante la ejecución de un programa e interrumpe su flujo normal. El manejo de excepciones es el mecanismo que proporciona Java para tratar estos errores de manera controlada y robusta, evitando que la aplicación termine abruptamente.

- **Jerarquía de Excepciones:** En la cima de la jerarquía se encuentra la clase Throwable. De ella derivan dos ramas principales [1]:

    - **Error:** Representa problemas graves y, por lo general, irrecuperables, que ocurren en la JVM, como OutOfMemoryError. Una aplicación razonable no debería intentar capturarlos.
    - **Exception:** Representa condiciones que una aplicación puede y, a menudo, debe capturar y manejar.
- **Excepciones Verificadas (Checked) vs. No Verificadas (Unchecked):** Esta es una distinción crucial en Java.
    - **Checked Exceptions:** Son subclases de Exception que no son subclases de RuntimeException. El compilador de Java verifica en tiempo de compilación que estas excepciones sean manejadas. Ejemplos típicos son IOException (al trabajar con ficheros) o SQLException (al interactuar con bases de datos). El código que puede lanzar una de estas excepciones debe o bien manejarla con un bloque try-catch o declararla en la firma del método con la cláusula throws.[1]
    - **Unchecked Exceptions:** Son las clases que heredan de RuntimeException (como NullPointerException, ArrayIndexOutOfBoundsException, IllegalArgumentException). El compilador no obliga a manejarlas. Generalmente, indican errores de lógica en la programación que deberían ser corregidos en el código, en lugar de ser capturados en tiempo de ejecución.[1]
- **Bloques try-catch-finally:**
    - try: Este bloque encierra el código que podría lanzar una excepción.[1]
    - catch: Si se lanza una excepción en el bloque try, la ejecución salta al bloque catch correspondiente que maneja ese tipo de excepción. Se pueden encadenar múltiples bloques catch para manejar diferentes tipos de excepciones.[1]
    - finally: Este bloque contiene código que se ejecutará siempre, independientemente de si se lanzó una excepción o no. Es el lugar ideal para liberar recursos, como cerrar ficheros o conexiones de red, para evitar fugas de recursos.[1]
- **try-with-resources:** Introducida en Java 7, esta es la forma moderna y preferida de manejar recursos que implementan la interfaz AutoCloseable (como los streams de ficheros). El recurso se declara en la cabecera del try, y Java garantiza que su método close() será llamado automáticamente al final del bloque, eliminando la necesidad de un bloque finally explícito y haciendo el código más limpio y seguro.[1]

Ejemplo: Manejo de Excepciones con try-catch-finally

Java

```
import java.io.FileReader;
import java.io.IOException;

public class DemoTryCatchFinally {
    public static void leerFichero() {
        FileReader fr = null;
        try {
            fr = new FileReader("archivo_inexistente.txt");
            // Código para leer el fichero...
            System.out.println("Fichero abierto correctamente.");
        } catch (IOException e) {
            System.err.println("Error al abrir el fichero: " + e.getMessage());
        } finally {
            System.out.println("Este bloque 'finally' se ejecuta siempre.");
            try {
                if (fr!= null) {
                    fr.close(); // Es crucial cerrar el recurso para liberar memoria.
                    System.out.println("Fichero cerrado.");
                }
            } catch (IOException e) {
                System.err.println("Error al cerrar el fichero: " + e.getMessage());
            }
        }
    }
}
```

Ejemplo: Manejo de Recursos con try-with-resources (Enfoque Moderno y Preferido)

Java

```
import java.io.FileReader;
import java.io.IOException;
```

```
public class DemoTryWithResources {
    public static void leerFichero() {
        // El recurso 'FileReader' se declara dentro de los paréntesis del try.
        // Java se encargará de cerrarlo automáticamente al final del bloque.
        try (FileReader fr = new FileReader("archivo_inexistente.txt")) {
            // Código para leer el fichero...
            System.out.println("Fichero abierto correctamente.");
        } catch (IOException e) {
            // No es necesario un bloque 'finally' para cerrar el recurso.
            System.err.println("Error al abrir o leer el fichero: " + e.getMessage());
        }
    }
}
```

- **throw y throws:**
  - throw: Se utiliza para lanzar explícitamente una instancia de una excepción.[1]
  - throws: Se utiliza en la firma de un método para declarar que ese método puede lanzar una o más excepciones verificadas, delegando la responsabilidad de manejarlas al código que lo llama.[1]

El Framework de Colecciones (Collections Framework): Organizando Datos

El Framework de Colecciones de Java es un conjunto unificado de interfaces y clases para representar y manipular grupos de objetos de manera eficiente.[1] Se organiza en torno a un pequeño número de interfaces clave.

- **Interfaces Principales:**
  - **List:** Representa una colección ordenada (una secuencia) que permite elementos duplicados. Los elementos pueden ser accedidos por su índice numérico, comenzando desde 0.[1]
    - Implementaciones comunes: ArrayList (basado en un array interno, muy eficiente para el acceso por índice), LinkedList (basado en una lista doblemente enlazada, eficiente para inserciones y eliminaciones en medio de la lista).
  - **Set:** Representa una colección que no permite elementos duplicados. La igualdad de los elementos se determina generalmente con el método equals().[1]

        - Implementaciones comunes: HashSet (usa una tabla hash, ofrece el mejor rendimiento pero no garantiza ningún orden), LinkedHashSet (mantiene el orden de inserción), TreeSet (mantiene los elementos ordenados según su orden natural o un Comparator proporcionado).
    - **Map:** Técnicamente no extiende la interfaz Collection, pero es parte del framework. Almacena pares clave-valor. Las claves deben ser únicas, y cada clave se asocia a un único valor.[1]
        - Implementaciones comunes: HashMap (el más rápido, no garantiza el orden de las claves), LinkedHashMap (mantiene el orden de inserción de las claves), TreeMap (mantiene las claves ordenadas).

La elección de la estructura de datos correcta es una decisión de diseño crítica que afecta tanto a la corrección como al rendimiento de un programa.

Característica	List	Set	Map
Orden	Mantiene el orden de inserción.	Generalmente sin orden (HashSet), pero puede ser ordenado (TreeSet) o mantener orden de inserción (LinkedHashSet)	Generalmente sin orden (HashMap), pero puede ser ordenado por clave (TreeMap) o mantener orden de inserción (LinkedHashMap).
Permite Duplicados	Sí.	No.	No permite claves duplicadas. Permite valores duplicados.
Estructura de Datos	Colección de elementos individuales.	Colección de elementos únicos.	Colección de pares clave-valor.
Acceso a Elementos	Por índice numérico (ej. get(int index)).	No hay acceso directo por índice. Se itera sobre el conjunto.	Por clave (ej. get(Object key)).

Implementaciones Clave	ArrayList, LinkedList, Vector	HashSet, LinkedHashSet, TreeSet	HashMap, LinkedHashMap, TreeMap	
Fuentes: [1]				

Ejemplo: List - Colección ordenada que permite duplicados

Java

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

public class DemoList {
    public static void main(String args) {
        List<String> nombres = new ArrayList<>();
        nombres.add("Ana");
        nombres.add("Luis");
        nombres.add("Ana"); // Se permiten duplicados.
        System.out.println(nombres); // Salida: [Ana, Luis, Ana]
        System.out.println("Elemento en el índice 1: " + nombres.get(1)); // Salida: Luis
    }
}
```

Ejemplo: Set - Colección sin duplicados

Java

```
import java.util.HashSet;
import java.util.Set;

public class DemoSet {
    public static void main(String args) {
        Set<String> etiquetas = new HashSet<>();
```

```
        etiquetas.add("Java");
        etiquetas.add("Programación");
        boolean añadido = etiquetas.add("Java"); // Intenta añadir un duplicado.
        System.out.println(etiquetas); // Salida: [Programación, Java] (el orden no está garantizado)
        System.out.println("¿Se añadió el duplicado? " + añadido); // Salida: false
    }
}
```

Ejemplo: Map - Estructura de clave-valor

Java

```
import java.util.HashMap;
import java.util.Map;

public class DemoMap {
    public static void main(String args) {
        Map<String, Integer> notas = new HashMap<>();
        notas.put("Matemáticas", 9);
        notas.put("Lengua", 7);
        notas.put("Historia", 8);
        System.out.println(notas); // Salida: {Matemáticas=9, Lengua=7, Historia=8}
        System.out.println("Nota de Lengua: " + notas.get("Lengua")); // Salida: 7
    }
}
```

Entrada/Salida (I/O) de Ficheros: Interactuando con el Exterior

Las aplicaciones a menudo necesitan leer y escribir datos en ficheros. Java proporciona APIs robustas para estas operaciones.

- **APIs de I/O:** Existen dos modelos principales para el manejo de ficheros:
    - **java.io (Legacy I/O):** Es la API original, basada en el concepto de streams (flujos) de datos. Hay streams de bytes (InputStream/OutputStream) para

datos binarios, y streams de caracteres (Reader/Writer) para texto. Esta API es bloqueante, lo que significa que el hilo que realiza una operación de I/O se bloquea hasta que la operación se completa.[1]
    - **java.nio (New I/O):** Introducida en Java 1.4 y mejorada significativamente en Java 7, esta API ofrece un modelo más moderno y eficiente. Se basa en canales (Channels) y búferes (Buffers), y soporta operaciones no bloqueantes, lo que es crucial para aplicaciones de alta concurrencia. La clase java.nio.file.Files proporciona métodos de utilidad que simplifican enormemente las operaciones comunes con ficheros.[1]
- **Lectura de Ficheros (Enfoque Moderno):** La clase Files es la forma recomendada de trabajar hoy en día.
    - Para ficheros pequeños: Files.readAllLines(Path path) lee todas las líneas del fichero y las devuelve como una List<String>.[1]
    - Para ficheros grandes: Files.lines(Path path) devuelve un Stream<String>, lo que permite procesar el fichero línea por línea de manera perezosa y eficiente en memoria, ideal para combinar con la API de Streams.[1]
    - El enfoque clásico con BufferedReader envuelto en un FileReader sigue siendo válido y eficiente para leer texto línea por línea.[1]
- **Escritura de Ficheros (Enfoque Moderno):**
    - La clase Files también simplifica la escritura. Files.write(Path path, byte bytes) o Files.write(Path path, Iterable<? extends CharSequence> lines) permiten escribir datos en un fichero con una sola línea de código.[1]
    - El enfoque clásico con BufferedWriter envuelto en un FileWriter sigue siendo una opción robusta.[1]

Independientemente del enfoque, es imperativo asegurarse de que los recursos de I/O (streams, readers, writers) se cierren correctamente para liberar los recursos del sistema. La mejor práctica para esto es utilizar la sentencia try-with-resources, que garantiza el cierre automático de los recursos declarados en su cabecera.[1]

Sección 4: Java Moderno en la Práctica

Desde la versión 8, Java ha experimentado una profunda transformación, incorporando características de la programación funcional que han cambiado la forma en que los desarrolladores escriben código. Dominar estas características es indispensable para ser un programador Java moderno y eficaz.

Genéricos (Generics): Escribiendo Código Seguro y Reutilizable

Introducidos en Java 5, los genéricos resolvieron un problema fundamental de seguridad de tipos en el lenguaje.

- **El Problema que Resuelven:** Antes de los genéricos, las colecciones como ArrayList almacenaban objetos de tipo Object. Esto significaba que se podía añadir cualquier tipo de objeto a una misma lista. Al recuperar un elemento, era necesario realizar un cast (conversión de tipo) explícito y arriesgado. Si se intentaba hacer un cast a un tipo incorrecto, el programa fallaba en tiempo de ejecución con una ClassCastException.[1]
- **Concepto y Sintaxis:** Los genéricos permiten parametrizar clases, interfaces y métodos con un "parámetro de tipo", usualmente representado por una letra mayúscula entre corchetes angulares (ej. class Box<T>). Esto permite al compilador verificar la seguridad de tipos en tiempo de compilación, detectando errores antes de que el programa se ejecute.[1]

  El uso más común es con colecciones: List<String> miLista = new ArrayList<>();. Esta declaración le dice al compilador que miLista solo puede contener objetos String. Cualquier intento de añadir otro tipo, como un Integer, será un error de compilación. Además, al recuperar elementos, no se necesita ningún cast. La sintaxis <> en new ArrayList<>() se conoce como el "operador diamante" (diamond operator), una mejora de Java 7 que permite al compilador inferir el tipo.1
- **Comodines (Wildcards):** Los genéricos pueden ser muy restrictivos. Los comodines ofrecen una mayor flexibilidad.[1]
  - **Límite Superior (? extends Tipo):** Se refiere a un tipo desconocido que es Tipo o cualquier subtipo de Tipo. Es útil para métodos que leen de una estructura genérica. Por ejemplo, un método procesarAnimales(List<? extends Animal> lista) podría aceptar una List<Perro> o una List<Gato>.
  - **Límite Inferior (? super Tipo):** Se refiere a un tipo desconocido que es Tipo o cualquier supertipo de Tipo. Es útil para métodos que escriben en una estructura genérica.
  - **Sin Límites (?):** Se refiere a un tipo totalmente desconocido.
- **Borrado de Tipos (Type Erasure):** Es importante saber que, por razones de compatibilidad con versiones antiguas de Java, el compilador borra la información de los tipos genéricos después de la compilación. El bytecode

resultante no contiene <String> o <Integer>. Esto tiene algunas implicaciones, como la imposibilidad de usar tipos primitivos como parámetros de tipo (se deben usar sus clases envoltorio).[1]

Expresiones Lambda: La Puerta a la Programación Funcional

Las expresiones lambda, introducidas en Java 8, son una de las características más transformadoras del lenguaje. Proporcionan una sintaxis concisa para representar funciones anónimas.[1]

- **Interfaces Funcionales:** Una expresión lambda es, en esencia, una implementación de una interfaz funcional. Una interfaz funcional es cualquier interfaz que contiene un único método abstracto. La anotación @FunctionalInterface se puede usar para que el compilador verifique esta condición.[1] Ejemplos de interfaces funcionales en la API de Java son Runnable (su método run()) o Comparator (su método compare()).
- **Sintaxis:** La sintaxis básica de una lambda es (lista de parámetros) -> { cuerpo de la expresión }.[1]
  - Si no hay parámetros, se usan paréntesis vacíos: () -> System.out.println("Hola");.
  - Si hay un solo parámetro, los paréntesis son opcionales: parametro -> System.out.println(parametro);.
  - Si el cuerpo es una sola expresión, las llaves y la palabra clave return son opcionales: (a, b) -> a + b;.

Ejemplo: De Clase Anónima a Expresión Lambda

Java

```
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class DemoLambda {
```

```
    public static void main(String args) {
        List<String> nombres = Arrays.asList("Carlos", "Ana", "Beatriz");

        // ANTES de Java 8: Usando una clase anónima.
        Collections.sort(nombres, new Comparator<String>() {
            @Override
            public int compare(String a, String b) {
                return a.compareTo(b);
            }
        });
        System.out.println("Ordenado con clase anónima: " + nombres);

        // CON Java 8: Usando una expresión lambda.
        // Es mucho más conciso y legible.
        nombres = Arrays.asList("Carlos", "Ana", "Beatriz");
        Collections.sort(nombres, (a, b) -> a.compareTo(b));
        System.out.println("Ordenado con lambda: " + nombres);
    }
}
```

La API de Streams: Procesamiento de Datos Declarativo

La API de Streams, también de Java 8, utiliza las expresiones lambda para proporcionar una forma fluida y declarativa de procesar colecciones de datos. Este es un cambio de paradigma fundamental en la programación Java.

Antes de Java 8, el procesamiento de colecciones era imperativo: el desarrollador escribía explícitamente el "cómo" hacerlo, usando bucles for, variables de control, e ifs anidados. Este código es a menudo verboso y propenso a errores.[1]

La API de Streams permite un estilo declarativo: el desarrollador especifica "qué" quiere lograr, y la API se encarga de la iteración y la lógica subyacente. Por ejemplo, para filtrar una lista de productos por precio y obtener sus nombres, el código declarativo sería: listaProductos.stream().filter(p -> p.getPrecio() > 100).map(p -> p.getNombre()).collect(Collectors.toList());

Este código no solo es más corto y legible, sino que también oculta la complejidad de

la iteración y abre la puerta a optimizaciones automáticas, como el procesamiento en paralelo con un simple cambio a parallelStream().[1]

Ejemplo: Procesamiento Declarativo con la API de Streams

Java

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

// Una clase simple para representar un Producto.
class Producto {
    private String nombre;
    private double precio;
    public Producto(String nombre, double precio) { this.nombre = nombre; this.precio = precio;
}
    public String getNombre() { return nombre; }
    public double getPrecio() { return precio; }
}

public class DemoStreams {
    public static void main(String args) {
        List<Producto> listaProductos = Arrays.asList(
            new Producto("Portátil", 1200.00),
            new Producto("Ratón", 25.50),
            new Producto("Teclado", 75.00),
            new Producto("Monitor", 350.00)
        );

        // Usando la API de Streams para obtener los nombres de los productos caros.
        List<String> nombresProductosCaros = listaProductos.stream() // 1. Obtener el
stream
            .filter(p -> p.getPrecio() > 100.00) // 2. Operación intermedia: filtrar
            .map(p -> p.getNombre())           // 3. Operación intermedia: transformar
            .collect(Collectors.toList());     // 4. Operación terminal: recolectar
```

```
        System.out.println(nombresProductosCaros); // Salida: [Portátil, Monitor]
    }
}
```

- **Conceptos Clave de Streams:**
    - Un Stream no es una estructura de datos que almacena elementos; es una vista sobre una fuente de datos (como una Collection o un array) que transporta elementos a través de una cadena de operaciones.[1]
    - Las operaciones de Stream no modifican la fuente original; producen un nuevo stream o un resultado.[1]
    - Las operaciones son perezosas (lazy). Las operaciones intermedias no se ejecutan hasta que se invoca una operación terminal.[1]
- **Operaciones de Stream:**
    - **Creación:** Se obtiene un stream de una fuente, por ejemplo, miLista.stream().
    - **Operaciones Intermedias:** Devuelven un nuevo stream y se pueden encadenar. Las más comunes son:
        - filter(Predicate): Selecciona elementos que cumplen una condición.
        - map(Function): Transforma cada elemento en otro.
        - sorted(): Ordena los elementos.
        - distinct(): Elimina duplicados.
        - limit(n): Trunca el stream a n elementos.
          (Fuentes: 1)
    - **Operaciones Terminales:** Producen un resultado final o un efecto secundario y cierran el stream. Las más comunes son:
        - forEach(Consumer): Realiza una acción en cada elemento.
        - collect(Collector): Recopila los elementos en una List, Set o Map.
        - reduce(): Combina los elementos en un único resultado.
        - count(): Devuelve el número de elementos.
        - anyMatch(), allMatch(): Verifican si los elementos cumplen una condición.
          (Fuentes: 1)

Dominar las lambdas y los streams es un salto cualitativo para cualquier desarrollador Java, permitiendo escribir código más expresivo, conciso y mantenible.

Sección 5: Concurrencia y Multihilos

La concurrencia es la capacidad de un sistema para ejecutar múltiples tareas o partes

de un programa en períodos de tiempo superpuestos. Es una característica esencial de las aplicaciones modernas para lograr un alto rendimiento y una buena capacidad de respuesta.

Fundamentos de la Concurrencia

- **Multitasking vs. Multithreading:** Es importante distinguir entre procesos e hilos. Un proceso es una instancia de un programa en ejecución con su propio espacio de memoria aislado. El cambio entre procesos (multitasking basado en procesos o multiprocesamiento) es costoso. Un hilo (thread) es la unidad de ejecución más pequeña dentro de un proceso. Múltiples hilos dentro del mismo proceso comparten el mismo espacio de memoria, lo que hace que la comunicación entre ellos sea mucho más rápida y el cambio de contexto (multithreading) sea más ligero.[1]
- **Beneficios:** La concurrencia permite una mejor utilización de las CPUs con múltiples núcleos, mejora la capacidad de respuesta de las interfaces de usuario (evitando que se "congelen" mientras se realiza una tarea larga en segundo plano) y aumenta el rendimiento general de las aplicaciones que pueden paralelizar su trabajo.[1]
- **Ciclo de Vida de un Hilo:** Un hilo pasa por varios estados durante su existencia.[1]
    - NEW: El hilo ha sido creado pero aún no ha comenzado su ejecución.
    - RUNNABLE: El hilo está listo para ejecutarse y está esperando tiempo de CPU del planificador de hilos.
    - BLOCKED: El hilo está esperando para adquirir un bloqueo (lock) que está en posesión de otro hilo.
    - WAITING: El hilo está esperando indefinidamente a que otro hilo realice una acción específica (ej. a través de object.wait() o thread.join()).
    - TIMED_WAITING: Similar a WAITING, pero con un tiempo de espera máximo (ej. Thread.sleep()).
    - TERMINATED: El hilo ha completado su ejecución.

Creación y Gestión de Hilos en Java

Existen dos enfoques principales para crear un hilo en Java:

1. **Extendiendo la clase Thread:** Se crea una nueva clase que hereda de java.lang.Thread y se sobrescribe su método run(), que contiene el código que ejecutará el hilo.[1]
2. **Implementando la interfaz Runnable:** Se crea una clase que implementa la interfaz java.lang.Runnable y su único método run(). Luego, se crea una instancia de Thread y se le pasa la instancia de Runnable al constructor. Este es el enfoque preferido porque es más flexible. Dado que Java no admite herencia múltiple de clases, extender Thread impide que la clase herede de cualquier otra. Implementar Runnable no tiene esta limitación.[1]

Para iniciar la ejecución de un nuevo hilo, es crucial llamar al método start(). Llamar directamente al método run() no crea un nuevo hilo; simplemente ejecuta el código de run() en el hilo actual.[1]

Ejemplo: Creación de Hilos (Implementando Runnable - Preferido)

Java

```
// Se crea una clase que define la tarea a ejecutar en un hilo.
class TareaSimple implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Ejecutando en el hilo: " + Thread.currentThread().getName());
            try {
                Thread.sleep(500); // Pausa la ejecución por 500 milisegundos.
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class DemoCreacionHilos {
    public static void main(String args) {
        // Se crea una instancia de la tarea.
        TareaSimple tarea = new TareaSimple();
```

```
        // Se crea un objeto Thread, pasándole la tarea.
        Thread miHilo = new Thread(tarea);

        // Se inicia el nuevo hilo. Esto llama al método run() de la tarea.
        miHilo.start();

        System.out.println("Hilo principal continúa su ejecución...");
    }
}
```

Problemas de la Concurrencia y Sincronización

Cuando múltiples hilos acceden y modifican recursos compartidos (como variables de instancia o ficheros), pueden surgir problemas graves si no se gestionan adecuadamente.

- **Condiciones de Carrera (Race Conditions):** Ocurren cuando el resultado de una operación depende del orden y el tiempo de ejecución de múltiples hilos que acceden a un recurso compartido. Un ejemplo clásico es un contador simple incrementado por varios hilos simultáneamente; sin protección, el valor final será probablemente incorrecto debido a que la operación count++ no es atómica (implica leer, modificar y escribir el valor).[1]
- **Sincronización:** Para prevenir las condiciones de carrera, Java proporciona el mecanismo de sincronización. La palabra clave synchronized se puede aplicar a métodos o a bloques de código. Cuando un hilo entra en un bloque o método sincronizado, adquiere un "bloqueo" (lock) intrínseco. Ningún otro hilo puede entrar en otro bloque sincronizado sobre el mismo objeto hasta que el primer hilo libere el bloqueo. Esto crea una sección crítica, garantizando el acceso mutuamente exclusivo a los recursos compartidos.

Ejemplo: Sincronización para Prevenir Condiciones de Carrera

Java

```
class Contador {
```

```
    private int cuenta = 0;

    // El método está sincronizado. Solo un hilo puede ejecutarlo a la vez.
    public synchronized void incrementar() {
        cuenta++;
    }

    public int getCuenta() {
        return cuenta;
    }
}

public class DemoSincronizacion {
    public static void main(String args) throws InterruptedException {
        Contador contador = new Contador();

        Runnable tarea = () -> {
            for (int i = 0; i < 10000; i++) {
                contador.incrementar();
            }
        };

        Thread hilo1 = new Thread(tarea);
        Thread hilo2 = new Thread(tarea);

        hilo1.start();
        hilo2.start();

        // Espera a que ambos hilos terminen.
        hilo1.join();
        hilo2.join();

        // Sin 'synchronized', el resultado sería impredecible y casi siempre < 20000.
        // Con 'synchronized', el resultado es siempre 20000.
        System.out.println("Valor final del contador: " + contador.getCuenta());
    }
}
```

- **Deadlocks (Bloqueos Mutuos):** Es una situación peligrosa en la que dos o más

hilos se bloquean permanentemente, cada uno esperando que el otro libere un recurso que necesita. Por ejemplo, el Hilo A bloquea el Recurso 1 y espera por el Recurso 2, mientras que el Hilo B bloquea el Recurso 2 y espera por el Recurso 1. Para evitar deadlocks, una estrategia común es adquirir siempre los bloqueos en un orden global y consistente.[1]

El Paquete java.util.concurrent: Concurrencia Moderna

Manejar hilos directamente con synchronized, wait() y notify() es complejo y propenso a errores. Por ello, Java 5 introdujo el paquete java.util.concurrent, que proporciona un framework de alto nivel para la concurrencia.[1]

- **Executor Framework:** Es el corazón de este paquete. Desacopla la definición de una tarea de su ejecución.
  - **ExecutorService:** Es la interfaz principal. En lugar de crear hilos manualmente, se envían tareas (Runnable o Callable) a un ExecutorService. Este servicio gestiona un pool de hilos (thread pool), reutilizando hilos existentes para ejecutar las tareas, lo que mejora significativamente el rendimiento al evitar el alto coste de crear y destruir hilos constantemente.[1]
  - **Callable y Future:** Mientras que Runnable no puede devolver un valor, Callable sí puede. Cuando se envía un Callable a un ExecutorService, devuelve un objeto Future. Un Future representa el resultado de una computación asíncrona. Permite comprobar si la tarea ha finalizado, esperar su finalización y obtener su resultado (o la excepción que lanzó).[1]
- **Otras Utilidades Clave:** El paquete java.util.concurrent también incluye:
  - **Colecciones Concurrentes:** Implementaciones de colecciones seguras para hilos como ConcurrentHashMap, que ofrecen un rendimiento mucho mejor que sus contrapartes sincronizadas (Collections.synchronizedMap).
  - **Sincronizadores:** Clases como Semaphore (para limitar el número de hilos que pueden acceder a un recurso a la vez), CountDownLatch (para permitir que uno o más hilos esperen hasta que un conjunto de operaciones en otros hilos se complete) y CyclicBarrier (para que un grupo de hilos se esperen mutuamente en un punto de barrera).[1]
  - **Locks:** El framework Lock (ReentrantLock, ReadWriteLock) ofrece un mecanismo de bloqueo más flexible y sofisticado que la palabra clave synchronized.

El uso de las utilidades de java.util.concurrent es la práctica recomendada para el desarrollo de aplicaciones concurrentes en Java moderno.

Obras citadas

Obras citadas

1. www.netec.com, fecha de acceso: junio 27, 2025, https://www.netec.com/post/historia-y-curiosidades-de-java#:~:text=Java%20nace%20en%201991%20con,con%20una%20m%C3%A1quina%20virtual%20propia.
2. Historia de Java. Una historia completa del desarrollo de Java, de ..., fecha de acceso: junio 27, 2025, https://codegym.cc/es/groups/posts/es.594.historia-de-java-una-historia-completa-del-desarrollo-de-java-de-1991-a-2021
3. ¿Qué es Java y cuál es su historia? - YouTube, fecha de acceso: junio 27, 2025, https://www.youtube.com/watch?v=E8weQyNVWug
4. Java (lenguaje de programación) - Wikipedia, la enciclopedia libre, fecha de acceso: junio 27, 2025, https://es.wikipedia.org/wiki/Java_(lenguaje_de_programaci%C3%B3n)
5. ¿Qué es Java? - Explicación del lenguaje de programación Java ..., fecha de acceso: junio 27, 2025, https://aws.amazon.com/es/what-is/java/
6. Historia de Java: un lenguaje de programación con recorrido - Tokio School, fecha de acceso: junio 27, 2025, https://www.tokioschool.com/noticias/historia-java-el-origen-de-este-lenguaje-de-programacion/
7. JDK vs JRE vs JVM in Java: Key Differences Explained | DigitalOcean, fecha de acceso: junio 27, 2025, https://www.digitalocean.com/community/tutorials/difference-jdk-vs-jre-vs-jvm
8. Differences Between JDK, JRE and JVM - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/differences-jdk-jre-jvm/
9. JVM vs. JRE vs. JDK: What's the Difference? | IBM, fecha de acceso: junio 27, 2025, https://www.ibm.com/think/topics/jvm-vs-jre-vs-jdk
10. What is the difference between JDK and JRE? - java - Stack Overflow, fecha de acceso: junio 27, 2025, https://stackoverflow.com/questions/1906445/what-is-the-difference-between-jdk-and-jre
11. What is the difference between JVM, JDK, JRE & OpenJDK? - Stack Overflow, fecha de acceso: junio 27, 2025, https://stackoverflow.com/questions/11547458/what-is-the-difference-between-jvm-jdk-jre-openjdk
12. Java Documentation - Get Started - Oracle Help Center, fecha de acceso: junio 27, 2025, https://docs.oracle.com/en/java/
13. en.wikipedia.org, fecha de acceso: junio 27, 2025, https://en.wikipedia.org/wiki/Jakarta_EE

14. Overview :: Jakarta EE Tutorial :: Jakarta EE Documentation, fecha de acceso: junio 27, 2025, https://jakarta.ee/learn/docs/jakartaee-tutorial/current/intro/overview/overview.html
15. Sintaxis de Java: una breve introducción al lenguaje de programación, fecha de acceso: junio 27, 2025, https://codegym.cc/es/groups/posts/es.484.sintaxis-de-java-una-breve-introduccion-al-lenguaje-de-programacion
16. Introducción a la sintaxis básica en Java, fecha de acceso: junio 27, 2025, https://javamagician.com/sintaxis-java/
17. Tutorial de flujo de control en Java: Condicionales y bucles | Laboratorio práctico | LabEx, fecha de acceso: junio 27, 2025, https://labex.io/es/tutorials/java-java-control-flow-conditionals-and-loops-413751
18. Variables y Tipos de Datos en Java: Fundamentos Esenciales | Blog Coders Free, fecha de acceso: junio 27, 2025, https://codersfree.com/posts/variables-y-tipos-de-datos-en-java
19. Sintaxis Java: Fundamentos y estructura del lenguaje - CertiDevs, fecha de acceso: junio 27, 2025, https://certidevs.com/aprender-java-sintaxis-fundamentos
20. Tipos de datos Java - DataCamp, fecha de acceso: junio 27, 2025, https://www.datacamp.com/es/doc/java/java-data-types
21. Sintaxis Básica de Java - Compilando Conocimiento, fecha de acceso: junio 27, 2025, https://compilandoconocimiento.com/2017/01/02/sintaxis-basica-de-java/
22. Todo sobre los tipos de datos primitivos en Java - IONOS, fecha de acceso: junio 27, 2025, https://www.ionos.com/es-us/digitalguide/paginas-web/desarrollo-web/primitivos-de-java/
23. Data Types in Java – Primitive and Non-Primitive Data Types ..., fecha de acceso: junio 27, 2025, https://www.shiksha.com/online-courses/articles/data-types-in-java-primitive-and-non-primitive-data-types/
24. Data Types in Java : Primitive & Non-Primitive Data Types - ScholarHat, fecha de acceso: junio 27, 2025, https://www.scholarhat.com/tutorial/java/data-types-in-java
25. Primitive Data Types - Java Tutorials - Oracle Help Center, fecha de acceso: junio 27, 2025, https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
26. Tema 2: SINTAXIS BÁSICA DE JAVA - Espacio "Compartir", fecha de acceso: junio 27, 2025, https://espaciocompartir.inap.es/v3/pluginfile.php/3414/mod_resource/content/5/Tema2.pdf
27. Operadores Relacionales en Java: Comparando Valores en tus Programas - Coders Free, fecha de acceso: junio 27, 2025, https://codersfree.com/posts/operadores-relacionales-en-java
28. Tipos y métodos de Java básicos - IBM, fecha de acceso: junio 27, 2025, https://www.ibm.com/docs/es/iis/11.5.0?topic=jrules-basic-java-types-methods
29. Operadores Java - DataCamp, fecha de acceso: junio 27, 2025,

https://www.datacamp.com/es/doc/java/java-operators
30. Introducción a Java: Operadores - OpenWebinars, fecha de acceso: junio 27, 2025, https://openwebinars.net/blog/introduccion-a-java-operadores/
31. Variables y Operadores de Java - LabEx, fecha de acceso: junio 27, 2025, https://labex.io/es/tutorials/java-variables-and-operators-in-java-178553
32. Java operadores: tipos y ejemplos detallados - CertiDevs, fecha de acceso: junio 27, 2025, https://certidevs.com/tutorial-java-operadores
33. Estructuras de control en Java, fecha de acceso: junio 27, 2025, https://javamagician.com/java-estructuras-control/
34. Guía de Uso de Estructuras de Control en Java - ACADEMIA ..., fecha de acceso: junio 27, 2025, https://academiasanroque.com/guia-de-uso-de-estructuras-de-control-en-java/
35. Estructuras de decisión, control y colecciones. I. OBJETIVOS II. INTRODUCCIÓN Facultad: Ingeniería Escuela, fecha de acceso: junio 27, 2025, https://www.udb.edu.sv/udb_files/recursos_guias/informatica-ingenieria/java-avanzado/2019/i/guia-2.pdf
36. Estructuras de Control de Flujo en Java: Estructura If/If-Else, Switch, Break, For, fecha de acceso: junio 27, 2025, https://www.garciablazquez.es/estructuras-de-control-de-flujo-en-java-estructura-if-if-else-switch-break-for/
37. Java estructuras control: manejo y ejemplos - CertiDevs, fecha de acceso: junio 27, 2025, https://certidevs.com/tutorial-java-estructuras-control
38. Java OOP(Object Oriented Programming) Concepts - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/object-oriented-programming-oops-concept-in-java/
39. Understanding Java's Object-Oriented Programming (OOP) - Dummies.com, fecha de acceso: junio 27, 2025, https://www.dummies.com/article/technology/programming-web-design/java/understanding-javas-object-oriented-programming-oop-199089/
40. Lesson: Object-Oriented Programming Concepts (The Java ..., fecha de acceso: junio 27, 2025, https://docs.oracle.com/javase/tutorial/java/concepts/
41. Java Class and Objects (With Example) - Programiz, fecha de acceso: junio 27, 2025, https://www.programiz.com/java-programming/class-objects
42. Classes and Objects in Java - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/classes-objects-java/
43. Difference between the Constructors and Methods - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/difference-between-the-constructors-and-methods/
44. Methods vs Constructors in Java - Stack Overflow, fecha de acceso: junio 27, 2025, https://stackoverflow.com/questions/19061599/methods-vs-constructors-in-java
45. Java Constructor Tutorial: Learn Basics and Best Practices - DigitalOcean, fecha de acceso: junio 27, 2025,

https://www.digitalocean.com/community/tutorials/constructor-in-java
46. Java Constructors - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/constructors-in-java/
47. Java Constructors - DataCamp, fecha de acceso: junio 27, 2025, https://www.datacamp.com/doc/java/constructors
48. Full Java constructors tutorial | TheServerSide, fecha de acceso: junio 27, 2025, https://www.theserverside.com/video/Full-Java-constructors-tutorial
49. Java Classes & Objects - YouTube, fecha de acceso: junio 27, 2025, https://m.youtube.com/watch?v=IUqKuGNasdM&pp=ygULI3B1YmxpY2phdmE%3D
50. Access Modifiers in Java - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/access-modifiers-java/
51. Java Access Modifiers - Tutorialspoint, fecha de acceso: junio 27, 2025, https://www.tutorialspoint.com/java/java_access_modifiers.htm
52. Access Modifiers in Java: Everything You Need to Know - Simplilearn.com, fecha de acceso: junio 27, 2025, https://www.simplilearn.com/tutorials/java-tutorial/access-modifiers
53. Java Access Modifiers (With Examples) - Programiz, fecha de acceso: junio 27, 2025, https://www.programiz.com/java-programming/access-modifiers
54. Controlling Access to Members of a Class (The Java™ Tutorials > Learning the Java Language > Classes and Objects) - Oracle Help Center, fecha de acceso: junio 27, 2025, https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
55. OOP - 4 Pillars in Java - DEV Community, fecha de acceso: junio 27, 2025, https://dev.to/ellgalst/oop-4-pillars-in-java-2c1
56. What is Object-Oriented Programming (oop)? Explaining four major principles - SoftServe, fecha de acceso: junio 27, 2025, https://career.softserveinc.com/en-us/stories/what-is-object-oriented-programming-oop-explaining-four-major-principles
57. Java OOP : What are the Four Pillars? | by Nathan Chiche | NewTechTips - Medium, fecha de acceso: junio 27, 2025, https://medium.com/newtechtips/java-oop-what-are-the-four-pillars-55920313a9b4
58. Request: 4 pillars of OOP & meanings : r/Rightytighty - Reddit, fecha de acceso: junio 27, 2025, https://www.reddit.com/r/Rightytighty/comments/10rm22f/request_4_pillars_of_oop_meanings/
59. Four Main Object Oriented Programming Concepts of Java ..., fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/four-main-object-oriented-programming-concepts-of-java/
60. Beginner's Guide to Java Classes and Objects - Medium, fecha de acceso: junio 27, 2025, https://medium.com/@AlexanderObregon/beginners-guide-to-java-classes-and-objects-977000adcb21

61. Try Catch in Java - Exception handling (With Examples) | Simplilearn, fecha de acceso: junio 27, 2025, https://www.simplilearn.com/tutorials/java-tutorial/try-catch-in-java
62. Java Exception Handling - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/exceptions-in-java/
63. Try, Catch and Finally in Java | Scaler Topics, fecha de acceso: junio 27, 2025, https://www.scaler.com/topics/java/try-catch-and-finally-in-java/
64. Exploring All Possible Combinations of Try-Catch-Finally in Java: A Complete Guide with Examples | by Kunal Bhangale | Medium, fecha de acceso: junio 27, 2025, https://medium.com/@bhangalekunal2631996/exploring-all-possible-combinations-of-try-catch-finally-in-java-a-complete-guide-with-examples-fc36bf20d3de
65. Flow control in try catch finally in Java - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/flow-control-in-try-catch-finally-in-java/
66. Using try-catch java - Stack Overflow, fecha de acceso: junio 27, 2025, https://stackoverflow.com/questions/22644397/using-try-catch-java
67. Reading and Writing Files in Java: A Comprehensive Guide | by Piyu Jain - Medium, fecha de acceso: junio 27, 2025, https://medium.com/javarevisited/reading-and-writing-files-in-java-a-comprehensive-guide-3fab0bfaf4cc
68. A Complete Guide to Java Collection Framework : Java Tutorial ..., fecha de acceso: junio 27, 2025, https://www.codingshuttle.com/blogs/java-collection-framework-java-collections/
69. Collections in Java - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/collections-in-java-2/
70. Difference between List, Set, and Map in Java - Tutorialspoint, fecha de acceso: junio 27, 2025, https://www.tutorialspoint.com/difference-between-list-set-and-map-in-java
71. Difference between List, Set and Map in Java - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/difference-between-list-set-and-map-in-java/
72. Java Collections -- List Set Map, fecha de acceso: junio 27, 2025, https://web.stanford.edu/class/archive/cs/cs108/cs108.1092/handouts/02SCollections.pdf
73. Java Map Collection Tutorial and Examples - CodeJava.net, fecha de acceso: junio 27, 2025, https://www.codejava.net/java-core/collections/java-map-collection-tutorial-and-examples
74. Java File I/O - Tutorialspoint, fecha de acceso: junio 27, 2025, https://www.tutorialspoint.com/java/java_files_io.htm
75. Reading and writing files in Java (Input/Output) - Tutorial - Vogella, fecha de acceso: junio 27, 2025, https://www.vogella.com/tutorials/JavaIO/article.html

76. Reading, Writing, and Creating Files (The Java™ Tutorials > Essential Java Classes > Basic I/O) - Oracle Help Center, fecha de acceso: junio 27, 2025, https://docs.oracle.com/javase/tutorial/essential/io/file.html
77. Basic File I/O in Java - JDoodle Tutorials, fecha de acceso: junio 27, 2025, https://www.jdoodle.com/tutorials/java-module/java-beginner/basic-file-input-output-in-java
78. Java Generics Tutorial - Jenkov.com, fecha de acceso: junio 27, 2025, https://jenkov.com/tutorials/java-generics/index.html
79. Java Generics Explained: Benefits, Examples, and Best Practice ..., fecha de acceso: junio 27, 2025, https://www.digitalocean.com/community/tutorials/java-generics-example-method-class-interface
80. Generic Types - Java™ Tutorials, fecha de acceso: junio 27, 2025, https://docs.oracle.com/javase/tutorial/java/generics/types.html
81. Java Generics (With Examples) - Programiz, fecha de acceso: junio 27, 2025, https://www.programiz.com/java-programming/generics
82. The Basics of Java Generics | Baeldung, fecha de acceso: junio 27, 2025, https://www.baeldung.com/java-generics
83. Lambda Expressions in Java: A Concise Guide with Examples | by Abu Talha - Medium, fecha de acceso: junio 27, 2025, https://abu-talha.medium.com/lambda-expressions-in-java-a-concise-guide-with-examples-47c7ade952fb
84. Java Lambda Expressions - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/lambda-expressions-java-8/
85. Java Lambda Expressions - Tutorialspoint, fecha de acceso: junio 27, 2025, https://www.tutorialspoint.com/java/java-lambda-expressions.htm
86. Java Lambda Expressions (With Examples) - Programiz, fecha de acceso: junio 27, 2025, https://www.programiz.com/java-programming/lambda-expression
87. Lambda Expressions and Functional Interfaces: Tips and Best Practices - Baeldung, fecha de acceso: junio 27, 2025, https://www.baeldung.com/java-8-lambda-expressions-tips
88. Java 8 stream api tutorial - w3schools.blog, fecha de acceso: junio 27, 2025, https://www.w3schools.blog/java-8-stream-api-tutorial-with-examples
89. A Guide to Java Streams: In-Depth Tutorial With Examples - Stackify, fecha de acceso: junio 27, 2025, https://stackify.com/streams-guide-java-8/
90. The Java Stream API Tutorial | Baeldung, fecha de acceso: junio 27, 2025, https://www.baeldung.com/java-8-streams
91. Introduction to Java 8 Stream API - Jade Global, fecha de acceso: junio 27, 2025, https://www.jadeglobal.com/blog/introduction-java-eight-stream-api
92. Java 8 Stream Tutorial - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/java-8-stream-tutorial/
93. Java 8 Streams Tutorial for Beginners | by Pratik T - Medium, fecha de acceso: junio 27, 2025, https://medium.com/@pratik.941/java-8-streams-tutorial-for-beginners-f76e13df5777

94. Java Multithreading Tutorial - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/java-multithreading-tutorial/
95. Java Concurrency: Master the Art of Multithreading - BairesDev, fecha de acceso: junio 27, 2025, https://www.bairesdev.com/blog/java-concurrency/
96. Java Concurrency and Multithreading Tutorial - Jenkov.com, fecha de acceso: junio 27, 2025, https://jenkov.com/tutorials/java-concurrency/index.html
97. Java Concurrency Geeksforgeeks - cdcgroup.com, fecha de acceso: junio 27, 2025, https://cdcgroup.com/HomePages/browse/390700/JavaConcurrencyGeeksforgeeks.pdf
98. Java Multithreading - Tutorialspoint, fecha de acceso: junio 27, 2025, https://www.tutorialspoint.com/java/java_multithreading.htm
99. Java Concurrency & Multithreading Complete Course in 2 Hours | Zero to Hero - YouTube, fecha de acceso: junio 27, 2025, https://www.youtube.com/watch?v=WldMTtUWqTg
100. java.util.concurrent Package - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/java-util-concurrent-package/
101. Gradle Tutorial for Complete Beginners | Tom Gregory, fecha de acceso: junio 27, 2025, https://tomgregory.com/gradle/gradle-tutorial-for-complete-beginners/
102. Gradle tutorial : r/learnjava - Reddit, fecha de acceso: junio 27, 2025, https://www.reddit.com/r/learnjava/comments/f8s0ry/gradle_tutorial/
103. Learn Maven tutorial for beginners | TheServerSide, fecha de acceso: junio 27, 2025, https://www.theserverside.com/video/Learn-Maven-tutorial-for-beginners
104. Maven Tutorial - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/advance-java/maven-tutorial/
105. Maven Tutorial - Tutorialspoint, fecha de acceso: junio 27, 2025, https://www.tutorialspoint.com/maven/index.htm
106. Maven vs. Gradle | Medium, fecha de acceso: junio 27, 2025, https://medium.com/@ahmettemelkundupoglu/maven-vs-gradle-a-detailed-comparison-of-pros-and-cons-with-examples-594ba33cc57f
107. What is POM in Maven - BrowserStack, fecha de acceso: junio 27, 2025, https://www.browserstack.com/guide/what-is-pom-in-maven
108. Introduction to the POM – Maven, fecha de acceso: junio 27, 2025, https://maven.apache.org/guides/introduction/introduction-to-the-pom.html
109. Maven POM - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/advance-java/maven-pom/
110. POM Reference - Apache Maven, fecha de acceso: junio 27, 2025, https://maven.apache.org/pom.html
111. Understanding pom.xml in Maven: A Beginner-Friendly Guide | by Anju - Medium, fecha de acceso: junio 27, 2025, https://anju-chaurasiya2012.medium.com/understanding-pom-xml-in-maven-a-beginner-friendly-guide-7f2e23232d4a
112. A Complete Guide on Maven Lifecycle | BrowserStack, fecha de acceso: junio 27, 2025, https://www.browserstack.com/guide/maven-lifecycle
113. Maven Lifecycle: Phases, Goals With Best Practices - LambdaTest, fecha de

acceso: junio 27, 2025, https://www.lambdatest.com/learning-hub/maven-lifecycle
114. Introduction to the Build Lifecycle – Maven, fecha de acceso: junio 27, 2025, https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
115. Understanding the Maven Build Lifecycle: A Detailed Guide | by PV Prasanth | Medium, fecha de acceso: junio 27, 2025, https://medium.com/@pvprasanth474/understanding-the-maven-build-lifecycle-a-detailed-guide-97248506795d
116. Maven Build Lifecycle - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/advance-java/maven-build-lifecycle/
117. Maven Build Life Cycle - Tutorialspoint, fecha de acceso: junio 27, 2025, https://www.tutorialspoint.com/maven/maven_build_life_cycle.htm
118. Gradle Tutorial - Tutorialspoint, fecha de acceso: junio 27, 2025, https://www.tutorialspoint.com/gradle/index.htm
119. What is Gradle? Why Do We Use Gradle? [Updated] - Simplilearn.com, fecha de acceso: junio 27, 2025, https://www.simplilearn.com/tutorials/gradle-tutorial/what-is-gradle
120. Build File Basics - Gradle User Manual, fecha de acceso: junio 27, 2025, https://docs.gradle.org/current/userguide/build_file_basics.html
121. Understanding Gradle Lifecycles - Tala, fecha de acceso: junio 27, 2025, https://tala.co/blog/2023/03/01/understanding-gradle-lifecycles/
122. Build Lifecycle - Gradle User Manual, fecha de acceso: junio 27, 2025, https://docs.gradle.org/current/userguide/build_lifecycle.html
123. gradle-guide/guide/lifecycle-of-a-gradle-build.md at develop - GitHub, fecha de acceso: junio 27, 2025, https://github.com/palantir/gradle-guide/blob/develop/guide/lifecycle-of-a-gradle-build.md
124. 3. Build Hooks - Gradle Beyond the Basics [Book] - O'Reilly Media, fecha de acceso: junio 27, 2025, https://www.oreilly.com/library/view/gradle-beyond-the/9781449373801/ch03.html
125. Chapter 20. The Build Lifecycle - Propersoft, fecha de acceso: junio 27, 2025, https://propersoft-cn.github.io/pep-refs/projects/gradle/2.12/userguide/build_lifecycle.html
126. Core Concepts - Gradle User Manual, fecha de acceso: junio 27, 2025, https://docs.gradle.org/current/userguide/gradle_basics.html
127. How Gradle Works Part 1 - Startup, fecha de acceso: junio 27, 2025, https://blog.gradle.org/how-gradle-works-1
128. Gradle and Maven Comparison - Gradle, fecha de acceso: junio 27, 2025, https://gradle.org/maven-and-gradle/
129. Gradle vs. Maven: Performance, Compatibility, Speed, & Builds - Stackify, fecha de acceso: junio 27, 2025, https://stackify.com/gradle-vs-maven/
130. Spring Tutorial - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/spring/
131. Why Spring Matters to Jakarta EE - and Vice Versa - Eclipse News, fecha de

acceso: junio 27, 2025, https://newsroom.eclipse.org/eclipse-newsletter/2024/march/why-spring-matters-jakarta-ee-and-vice-versa
132. What is Dependency Injection and Inversion of Control in Spring Framework?, fecha de acceso: junio 27, 2025, https://stackoverflow.com/questions/9403155/what-is-dependency-injection-and-inversion-of-control-in-spring-framework
133. Dependency injection and inversion of control in Spring - TechTarget, fecha de acceso: junio 27, 2025, https://www.techtarget.com/searchapparchitecture/video/Dependency-injection-and-inversion-of-control-in-Spring
134. IoC vs Di : r/java - Reddit, fecha de acceso: junio 27, 2025, https://www.reddit.com/r/java/comments/1gljx3q/ioc_vs_di/
135. Inversion of Control in Spring Framework - DEV Community, fecha de acceso: junio 27, 2025, https://dev.to/be11amer/inversion-of-control-in-spring-framework-4mc0
136. Spring - Difference Between Inversion of Control and Dependency ..., fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/spring-difference-between-inversion-of-control-and-dependency-injection/
137. Inversion of control vs. dependency injection | TheServerSide, fecha de acceso: junio 27, 2025, https://www.theserverside.com/video/Inversion-of-control-vs-dependency-injection
138. Spring Boot Tutorial - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/advance-java/spring-boot/
139. Getting Started | Building an Application with Spring Boot, fecha de acceso: junio 27, 2025, https://spring.io/guides/gs/spring-boot/
140. Spring Boot for Beginners - Daily.dev, fecha de acceso: junio 27, 2025, https://daily.dev/blog/spring-boot-for-beginners
141. Spring Boot Tutorial - Tutorialspoint, fecha de acceso: junio 27, 2025, https://www.tutorialspoint.com/spring_boot/index.htm
142. SPRING MVC TUTORIAL - Cogent University, fecha de acceso: junio 27, 2025, https://www.cogentuniversity.com/post/spring-mvc-tutorial
143. Spring - MVC Framework - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/spring-mvc-framework/
144. Spring MVC Tutorial - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/spring-mvc/
145. Comprehensive Guide to Spring Data JPA with Example Codes | by ..., fecha de acceso: junio 27, 2025, https://medium.com/@vijayskr/comprehensive-guide-to-spring-data-jpa-with-example-codes-8db0c9683b0f
146. Spring Data JPA Tutorial - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/advance-java/spring-data-jpa-tutorial/
147. Introduction to Spring Data JPA - Baeldung, fecha de acceso: junio 27, 2025,

https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa
148. JPA with Spring Boot: A Comprehensive Guide with Examples - Medium, fecha de acceso: junio 27, 2025, https://medium.com/@bshiramagond/jpa-with-spring-boot-a-comprehensive-guide-with-examples-e07da6f3d385
149. The ULTIMATE Guide for Spring Data JPA & Hibernate | 5 Hours Tutorial - YouTube, fecha de acceso: junio 27, 2025, https://www.youtube.com/watch?v=mcl_nibV39s
150. Getting Started | Accessing Data with JPA - Spring, fecha de acceso: junio 27, 2025, https://spring.io/guides/gs/accessing-data-jpa/
151. Jakarta EE overview :: Open Liberty Docs, fecha de acceso: junio 27, 2025, https://openliberty.io/docs/latest/jakarta-ee.html
152. Jakarta Servlet :: Jakarta EE Tutorial :: Jakarta EE Documentation, fecha de acceso: junio 27, 2025, https://jakarta.ee/learn/docs/jakartaee-tutorial/current/web/servlets/servlets.html
153. Introduction to Java Servlets - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/java/introduction-java-servlets/
154. Getting Started with Web Applications :: Jakarta EE Tutorial, fecha de acceso: junio 27, 2025, https://jakarta.ee/learn/docs/jakartaee-tutorial/current/web/webapp/webapp.html
155. Overview :: Jakarta EE Tutorial, fecha de acceso: junio 27, 2025, https://jakarta.ee/learn/docs/jakartaee-tutorial/9.1/intro/overview/overview.html
156. Jakarta Servlet, Jakarta Faces and Jakarta Server Pages Explained ..., fecha de acceso: junio 27, 2025, https://jakarta.ee/learn/specification-guides/servlet-faces-and-server-pages-explained/
157. JSP Tutorial - Tutorialspoint, fecha de acceso: junio 27, 2025, https://www.tutorialspoint.com/jsp/index.htm
158. Introduction to Jakarta Persistence :: Jakarta EE Tutorial - Jakarta® EE, fecha de acceso: junio 27, 2025, https://jakarta.ee/learn/docs/jakartaee-tutorial/current/persist/persistence-intro/persistence-intro.html
159. Getting Started With JPA/Hibernate - DZone, fecha de acceso: junio 27, 2025, https://dzone.com/articles/getting-started-with-jpahibernate
160. How to Store and Retrieve Data Using Jakarta Persistence of Jakarta EE, fecha de acceso: junio 27, 2025, https://jakarta.ee/learn/starter-guides/how-to-store-and-retrieve-data-using-jakarta-persistence/
161. Chapter 13 Building RESTful Web Services with JAX-RS (The Java EE 6 Tutorial), fecha de acceso: junio 27, 2025, https://docs.oracle.com/cd/E19798-01/821-1841/6nmq2cp1v/index.html
162. Building RESTful Web Services with Jakarta REST :: Jakarta EE ..., fecha de acceso: junio 27, 2025, https://jakarta.ee/learn/docs/jakartaee-tutorial/current/websvcs/rest/rest.html
163. JAX-RS Tutorial - Create Jakarta RESTful Web Services - rieckpil, fecha de

acceso: junio 27, 2025, https://rieckpil.de/whatis-jakarta-restful-web-services-jax-rs/
164. Spring Boot vs Jakarta EE: Which One to Use in 2025? - Brilworks, fecha de acceso: junio 27, 2025, https://www.brilworks.com/blog/jakarta-ee-vs-spring-boot/
165. All-in-One Comparison Guide For Java EE vs. Spring Boot - Nintriva, fecha de acceso: junio 27, 2025, https://nintriva.com/blog/java-ee-spring-boot-comparison/
166. Spring Boot vs. Jakarta EE: Choosing the Right Framework for Your Java Application, fecha de acceso: junio 27, 2025, https://www.javacodegeeks.com/2024/12/spring-boot-vs-jakarta-ee-choosing-the-right-framework-for-your-java-application.html
167. Jakarta EE vs. Spring Boot: Choosing the Right Framework for Your ..., fecha de acceso: junio 27, 2025, https://blog.payara.fish/jakarta-ee-vs.-spring-boot-choosing-the-right-framework-for-your-project
168. Jakarta EE vs Spring: Which Java Framework is Right for Your Project? - spec india, fecha de acceso: junio 27, 2025, https://www.spec-india.com/blog/jakarta-ee-vs-spring
169. Software Testing Tutorial - GeeksforGeeks, fecha de acceso: junio 27, 2025, https://www.geeksforgeeks.org/software-testing/software-testing-tutorial/
170. JUnit Tutorial - Tutorialspoint, fecha de acceso: junio 27, 2025, https://www.tutorialspoint.com/junit/index.htm
171. Introduction to testing - Java Programming, fecha de acceso: junio 27, 2025, https://java-programming.mooc.fi/part-6/3-introduction-to-testing/
172. Getting Started | Spring on Kubernetes, fecha de acceso: junio 27, 2025, https://spring.io/guides/topicals/spring-on-kubernetes/
173. Cloud Native Java with Kubernetes, 2nd Edition[Video] - O'Reilly Media, fecha de acceso: junio 27, 2025, https://www.oreilly.com/library/view/cloud-native-java/9780137834051/
174. Spring Cloud Kubernetes, fecha de acceso: junio 27, 2025, https://spring.io/projects/spring-cloud-kubernetes/