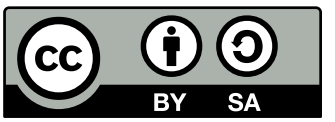
Programación en Java

Libro de texto para el módulo
"Programación (0485)"
de Formación Profesional

José J. Durán

Para acceder a los contenidos de este libro, y otros títulos, visita:
`https://github.com/cavefish-dev/libros`

Edición: 25 de agosto de 2025



Índice general

D Ejercicios

A

Introducción

1

Presentación del libro

Este libro está diseñado para introducir al lector en el mundo de la programación, proporcionando una base sólida y práctica. A lo largo de sus capítulos, se exploran conceptos fundamentales, ejemplos ilustrativos y ejercicios que fomentan el aprendizaje activo. El objetivo principal es acompañar al estudiante en su proceso de formación, facilitando la comprensión y el desarrollo de habilidades esenciales para programar en diversos lenguajes y entornos.

En este libro, se abordan temas como la sintaxis básica de los lenguajes de programación, estructuras de control, manejo de datos y conceptos avanzados como la programación orientada a objetos. Cada sección incluye explicaciones claras, ejemplos prácticos y ejercicios diseñados para reforzar el aprendizaje.

2

Cómo usar este libro

A lo largo de este libro encontrarás la información organizada en capítulos y secciones, cada una dedicada a un tema específico. Dentro de cada capítulo, se presentarán ejemplos de código usando el siguiente formato:

```java
public class Ejemplo {
  public static void main(String[] args) {
    System.out.println("¡Hola, mundo!");
  }
}
```

También se incluyen cajas de información destacada, como consejos, advertencias y ejercicios, para resaltar puntos importantes y facilitar la comprensión de los conceptos. Estas cajas están diseñadas para ser visualmente atractivas y fáciles de identificar.

Consejo

A lo largo del libro, encontrarás consejos útiles para mejorar tu comprensión y habilidades de programación. Estos consejos están diseñados para ayudarte a evitar errores comunes y optimizar tu proceso de aprendizaje.

Importante

Es fundamental prestar atención a las advertencias y notas importantes que se presentan en el libro. Estas secciones destacan aspectos críticos que pueden afectar tu comprensión o implementación de los conceptos, o que incluyan información adicional relevante para el tema tratado, pero que sean demasiado avanzados para el nivel del lector.

Ejercicio 2.1

Se incluirán ejercicios prácticos al final de cada capítulo para que puedas aplicar lo aprendido. Estos ejercicios están diseñados para ser desafiantes y fomentar la práctica activa. Los ejercicios se presentan en un formato claro

y conciso, permitiendo al lector enfocarse en la resolución de problemas específicos.

2.1. Estructura del libro

Este libro se divide en varias partes, cada una de las cuales aborda un aspecto diferente de la programación. Se recomienda seguir el orden de los capítulos para construir una comprensión progresiva de los conceptos.

La parte A (*Introducción*) proporciona una visión general de los contenidos de este libro, y como poder ejecutar los ejemplos y ejercicios propuestos.

La parte B (*Programación en Java*) se centra en el lenguaje Java, proporcionando una introducción a sus características y sintaxis. Se incluyen ejemplos prácticos y ejercicios que permiten al lector familiarizarse con el lenguaje y desarrollar habilidades de programación. Los puntos desarrollados en esta parte, buscan satisfacer los contenidos básicos del módulo *Programación* (código 0485) del ciclo formativo de grado superior *Desarrollo de Aplicaciones Multiplataforma* y *Desarrollo de Aplicaciones Web*.

La parte C (*Programación avanzada en Java*) se centra en conceptos más avanzados de programación, incluyendo patrones de diseño, programación funcional y técnicas de depuración. Esta sección está diseñada para aquellos que ya tienen una base en programación y desean profundizar en temas más complejos. Los contenidos de esta parte son extracurriculares y no forman parte del módulo *Programación* (0485), pero son altamente recomendables para el desarrollo profesional del alumnado.

La parte D (*Ejercicios*) contiene una serie de ejercicios prácticos que permiten al lector aplicar lo aprendido en los capítulos anteriores. Estos ejercicios están diseñados para reforzar la comprensión de los conceptos y fomentar la práctica activa.

3

Cómo programar los ejemplos y ejercicios

Para programar los ejemplos y ejercicios propuestos en este libro, se recomienda seguir los siguientes pasos:

1. **Configurar el entorno de desarrollo:** Asegúrate de tener instalado un entorno de desarrollo adecuado para el lenguaje de programación que estés utilizando. Para Java, se recomienda utilizar un IDE como IntelliJ IDEA o Eclipse.

2. **Leer atentamente el enunciado:** Antes de comenzar a programar, lee cuidadosamente el enunciado del ejercicio o ejemplo. Asegúrate de entender todos los requisitos y restricciones.

3. **Planificar la solución:** Antes de escribir código, es útil planificar la solución. Esto puede incluir la elaboración de diagramas de flujo, pseudocódigo o simplemente una lista de pasos a seguir.

4. **Implementar el código:** Comienza a escribir el código siguiendo el plan que has elaborado. No dudes en consultar la documentación oficial del lenguaje o las bibliotecas que estés utilizando.

5. **Probar y depurar:** Una vez que hayas implementado la solución, es fundamental probarla con diferentes casos de prueba. Si encuentras errores, utiliza herramientas de depuración para identificar y corregir los problemas.

6. **Refactorizar:** Después de que tu solución funcione correctamente, considera si hay oportunidades para mejorar el código. Esto puede incluir la eliminación de duplicaciones, la mejora de la legibilidad o la optimización del rendimiento.

Siguiendo estos pasos, podrás abordar de manera efectiva los ejemplos y ejercicios propuestos en este libro, desarrollando tus habilidades de programación y tu comprensión de los conceptos tratados.

3.1. Uso de JShell

JShell es una herramienta interactiva de Java que permite ejecutar fragmentos de código Java de manera rápida y sencilla. Es especialmente útil para probar pequeñas porciones de código, experimentar con nuevas ideas o aprender el lenguaje sin necesidad de crear un proyecto completo.

Para utilizar JShell, sigue estos pasos:

1. **Abrir JShell:** Si tienes Java instalado, puedes abrir JShell ejecutando el comando `jshell` en la terminal o consola de comandos.

2. **Escribir código:** Una vez abierto JShell, puedes escribir y ejecutar instrucciones Java directamente. Por ejemplo, prueba con `System.out.println("Hola, mundo");`.

3. **Explorar funcionalidades:** JShell permite definir variables, métodos y clases de forma interactiva. Puedes experimentar con fragmentos de código sin necesidad de compilar archivos completos.

4. **Salir de JShell:** Para salir, escribe `/exit` y presiona Enter.

JShell es ideal para aprender, probar ejemplos y depurar pequeñas partes de código Java de manera rápida.

3.2. Programar en JAVA sin un IDE

Si no dispones de un entorno de desarrollo integrado (IDE), puedes programar en Java utilizando únicamente la terminal y un editor de texto sencillo (como Notepad, VS Code, Vim, o Nano). Los pasos básicos son los siguientes:

1. **Escribir el código fuente:** Crea un archivo de texto con extensión `.java`. Por ejemplo, `HolaMundo.java`. Escribe el código Java en este archivo.

2. **Compilar el programa:** Abre la terminal y navega hasta la carpeta donde guardaste el archivo. Ejecuta el comando `javac HolaMundo.java` para compilar el código. Si no hay errores, se generará un archivo `HolaMundo.class`.

3. **Ejecutar el programa:** En la terminal, ejecuta el comando `java HolaMundo` para ejecutar el programa.

Este método es útil para comprender el proceso de compilación y ejecución de programas Java, y no requiere instalar software adicional más allá del JDK (Java Development Kit).

3.2.1. Compilar y ejecutar paquetes en Java

En Java, los paquetes permiten organizar el código en módulos y evitar conflictos de nombres. Para compilar y ejecutar programas que utilizan paquetes, sigue estos pasos:

1. **Estructura de directorios:** Crea una estructura de carpetas que refleje el nombre del paquete. Por ejemplo, si tu paquete se llama `com.ejemplo`, crea una carpeta `com/ejemplo/`.

2. **Escribir el código fuente:** Dentro de la carpeta correspondiente, crea el archivo Java y declara el paquete al inicio del archivo:

```java
package com.ejemplo;

public class HolaPaquete {
  public static void main(String[] args) {
    System.out.println("Hola desde un paquete");
  }
}
```

3. **Compilar recursivamente todos los archivos:** Desde la carpeta raíz del proyecto (donde está la carpeta `com`), ejecuta el comando:

```
javac com/**/*.java
```

   Esto compilará todos los archivos Java dentro de la estructura de paquetes de forma recursiva y generará los archivos `.class` correspondientes.

4. **Ejecutar el programa:** Para ejecutar la clase, usa el nombre completo del paquete:

```
java com.ejemplo.HolaPaquete
```

Este procedimiento es fundamental para proyectos Java más grandes y facilita la organización y reutilización del código.

B

Programación en Java

4

Identificación de los elementos de un programa informático

En este capítulo se presentan los elementos esenciales que componen un programa informático. Comprender estos conceptos es fundamental para desarrollar software de manera eficiente y estructurada. A continuación, se describen los principales componentes:

- **Estructura y bloques fundamentales:** Todo programa está organizado en bloques o secciones que definen su funcionamiento general, como la declaración de variables, funciones y el flujo principal de ejecución.
- **Variables:** Son espacios de memoria que permiten almacenar y manipular datos durante la ejecución del programa. Cada variable tiene un nombre identificador y puede cambiar su valor a lo largo del tiempo.
- **Tipos de datos:** Definen la naturaleza de los valores que pueden almacenar las variables, como números enteros, números reales, caracteres, cadenas de texto, entre otros.
- **Literales:** Son valores constantes escritos directamente en el código fuente, como `5`, `3.14` u `"Hola mundo"`.
- **Constantes:** Similares a las variables, pero su valor no puede modificarse una vez definido. Se utilizan para representar valores fijos a lo largo del programa.
- **Comentarios:** Son anotaciones incluidas en el código para explicar su funcionamiento o facilitar su comprensión. Los comentarios no afectan la ejecución del programa.

Estos conceptos constituyen la base sobre la que se construyen programas más complejos y permiten al programador expresar instrucciones de manera clara y precisa.

Para ilustrar estos conceptos, a lo largo de este capítulo, usaremos el siguiente fichero de ejemplo, el cual es un programa Java simple que imprime un mensaje en la consola:

```java
package programacion.ejemplos;

public class HelloWorld {

  private static final String DESPEDIDA = "Adiós mundo";

  /**
   * Método principal que se ejecuta al iniciar el programa.
   * Imprime un mensaje de saludo y una despedida.
   *
   * @param args Argumentos de la línea de comandos (no se utilizan en este ejemplo).
   */
  public static void main(String[] args) {
    String texto = "Hola mundo";
    System.out.println(texto);
    // Modificación del texto
    texto = "Hola mundo modificado";
    System.out.println(texto);
    System.out.println(DESPEDIDA);
    /*
     * Salida del programa:
     * Hola mundo
     * Hola mundo modificado
     * Adiós mundo
     */
  }
}
```

Ejemplo 4.1: Programa Hola Mundo

4.1. Estructura y bloques fundamentales

Los programas en Java se organizan en distintos ficheros, los cuales pueden representar clases, interfaces, o enumerados. Cada fichero debe tener un nombre que coincida con el nombre de la clase pública que contiene, y la extensión del fichero debe ser `.java`. Por ejemplo, si tenemos una clase llamada `HelloWorld`, el fichero debe llamarse `HelloWorld.java`. Además, cada fichero debe contener una única clase pública, aunque puede contener otras clases no públicas.

Estos ficheros se organizan mediante el uso de paquetes, que son una forma de agrupar clases relacionadas. Un paquete se define al inicio del fichero con la palabra clave `package` seguida del nombre del paquete. En el ejemplo, nuestra

clase `HelloWorld` forma parte de un paquete llamado `programacion.ejemplos`, y estará dentro de la carpeta `ejemplos`, que a su vez estará dentro de la carpeta `programacion`.

En nuestro ejemplo, definimos la clase `HelloWorld` usando la palabra clave `class` seguida del nombre de la clase, y el cuerpo de la clase se define entre llaves. Dentro de la clase, podemos declarar variables, métodos y otros elementos que componen su funcionalidad. Delante de la declaración de la clase podemos incluir modificadores de acceso como `public` o `private` para controlar la visibilidad de la clase desde otros ficheros.

Las clases nos permiten organizar el código de manera modular y reutilizable. Para una definición más detallada de clases, se recomienda consultar el capítulo 8.

Importante

Recuerda que en Java, cada fichero está formado por bloques delimitados por llaves `{}`, y que cada bloque puede contener a su vez otros bloques, o instrucciones, las cuales deben delimitarse con punto y coma `;`. Esta estructura jerárquica es fundamental para entender cómo se organiza el código en Java.

4.1.1. Modificadores de acceso

Los modificadores de acceso en Java determinan la visibilidad de clases, métodos y variables. Los principales son:

- `public`: El elemento es accesible desde cualquier otra clase.
- `private`: El elemento solo es accesible dentro de la propia clase.
- `protected`: El elemento es accesible dentro del mismo paquete y por las subclases.
- **(sin modificador):** También llamado *package-private*, el elemento es accesible solo dentro del mismo paquete.

4.1.2. Método main

El método `main` es el punto de entrada de un programa Java. Al ejecutar un programa, la máquina virtual de Java busca este método para iniciar la ejecución. Recibe como parámetro de entrada un array[1] de cadenas de texto (

[1]Para más información sobre el uso de arrays, consulta el capítulo 10

`String[] args`), que permite pasar argumentos al programa desde la línea de comandos.

El resultado de ejecutar este programa es la siguiente salida en la consola:

```
Hola mundo
Hola mundo modificado
Adiós mundo
```

4.2. Variables, tipos de datos y constantes

Las variables son espacios de memoria que permiten almacenar datos temporales durante la ejecución de un programa. Cada variable tiene un nombre identificador y un tipo de dato asociado, que define la naturaleza de los valores que puede almacenar. En Java, las variables pueden hacer referencia a tipos primitivos (capítulo 5), o a objetos (capítulo 6). la definición de una variable se realiza mediante la siguiente sintaxis:

```
tipo nombreVariable = valorInicial;
```

Donde `tipo` es el tipo de dato de la variable, `nombreVariable` es el identificador de la variable y `valorInicial` es el valor que se asigna a la variable al momento de su declaración. Es posible declarar una variable sin asignarle un valor inicial, pero en ese caso, no se podrá utilizar hasta que se le asigne un valor.

Es posible declarar varias variables del mismo tipo en una sola línea, separándolas por comas:

```
tipo nombreVariable1=valorInicial1, nombreVariable2=↪
  valorInicial2, ...;
```

Es importante mencionar que las variables en Java son sensibles a mayúsculas y minúsculas, por lo que `miVariable` y `mivariable` serían consideradas dos variables diferentes.

Para modificar el valor de una variable ya declarada, simplemente se asigna un nuevo valor utilizando la misma sintaxis de asignación:

```
nombreVariable = nuevoValor;
```

Por último, las variables pueden ser declaradas como `final`, lo que indica que su valor no puede ser modificado una vez asignado. Esto es útil para definir constantes, que son valores fijos a lo largo del programa. La sintaxis para declarar una constante es similar a la de una variable, pero se utiliza la palabra clave `final` antes del tipo de dato.

4.2.1. Literales

Los literales son valores constantes que se escriben directamente en el código fuente. En Java, existen varios tipos de literales, entre ellos:

- **Literales enteros:** Representan números enteros, como `5`, `-10` o `0`.
- **Literales de punto flotante:** Representan números reales, como `3.14`, `-2.5` o `0.0`.
- **Literales de caracteres:** Representan un único carácter, encerrado entre comillas simples, como `'a'`, `'1'` o `'\n'`.
- **Literales de cadena de texto:** Representan secuencias de caracteres, encerradas entre comillas dobles, como `"Hola mundo"` o `"123"`.
- **Literales booleanos:** Los valores `true` o `false`.
- **Literales nulos:** Representan la ausencia de un valor, utilizando la palabra clave `null`.

4.2.2. Comentarios

Los comentarios son anotaciones en el código que no afectan su ejecución, pero sirven para explicar su funcionamiento o facilitar su comprensión. En Java, existen dos tipos de comentarios:

- **Comentarios de una línea:** Se inician con dos barras inclinadas (`//`) y se extienden hasta el final de la línea.
- **Comentarios de varias líneas:** Se encierran entre los símbolos `/* */`, permitiendo incluir texto en varias líneas.
- **Comentarios de documentación:** Se utilizan para generar documentación del código y se escriben con `/** */`. Estos comentarios pueden incluir etiquetas especiales como `@param` o `@return` para documentar los parámetros y el valor de retorno de un método.

Los comentarios son una herramienta esencial para mejorar la legibilidad del código y facilitar su mantenimiento a largo plazo.

Importante

En este ejemplo, llamamos al método `System.out.println` para imprimir mensajes en la consola. Este método es parte de la clase `System`,

que proporciona acceso a funcionalidades del sistema, como la entrada y salida estándar. En este caso, `out` es un objeto de tipo `PrintStream` que permite enviar texto a la consola.
Más adelante, en el capítulo 8, se explicará en detalle cómo funcionan las clases y objetos en Java, incluyendo la clase `System` y sus métodos. Por ahora, considéralo una herramienta que te ayudará a escribir tus programas y a ver los resultados de tu código en la consola.

Resumen

En este capítulo se han presentado los elementos fundamentales que conforman un programa informático, utilizando Java como lenguaje de referencia. Se ha explicado la estructura básica de un programa, el uso de clases y paquetes, así como la importancia del método `main` como punto de entrada. Además, se han detallado los conceptos de variables, tipos de datos, literales y constantes, junto con la sintaxis para su declaración y uso. Finalmente, se ha destacado el papel de los comentarios para mejorar la legibilidad y el mantenimiento del código. Estos conocimientos son esenciales para comprender y desarrollar programas más complejos en capítulos posteriores.

Ejercicio 4.1

Crea un programa que tenga en cuenta los siguientes puntos:

- Declara una constante con tu nombre.
- Declara una variable con un saludo.
- Imprime en la consola el valor de ambas variables utilizando el método `System.out.println`.
- Modifica el valor de la variable, con un texto de despedida, y vuelve a imprimirla.
- Utiliza comentarios para explicar cada parte del código.

5

Utilización de tipos primitivos

En este capítulo, exploraremos los tipos primitivos en Java, que son los bloques de construcción fundamentales para almacenar datos. Estos tipos son esenciales para la manipulación de información y la realización de cálculos en un programa. En particular, veremos los siguientes aspectos:

- **Tipos de datos primitivos:** Analizaremos los diferentes tipos de datos primitivos que existen en Java, como `int`, `double`, `char`, `boolean`, entre otros. Explicaremos sus características, rangos de valores y cuándo es apropiado utilizar cada uno.
- **Operadores y expresiones:** Estudiaremos los operadores básicos (aritméticos, relacionales, lógicos) y cómo se utilizan para construir expresiones que manipulan los valores de los tipos primitivos. Veremos ejemplos prácticos de su uso en código.
- **Conversiones de tipo:** Aprenderemos cómo convertir valores entre diferentes tipos primitivos, tanto de manera implícita (conversión automática) como explícita (casting). Discutiremos las posibles pérdidas de información y buenas prácticas al realizar conversiones.

5.1. Tipos de datos primitivos

En Java, podemos dividir los tipos primitivos en dos categorías principales: tipos numéricos y tipos no numéricos. Los tipos numéricos se utilizan para representar valores numéricos, mientras que los tipos no numéricos se utilizan para representar valores lógicos o caracteres.

5.1.1. Tipos numéricos

Los tipos numéricos se dividen en enteros y de punto flotante. Los tipos enteros son aquellos que representan números enteros, mientras que los de punto flotante representan números con decimales.

Tipos enteros

Los tipos enteros en Java son:

- **byte**: Ocupa 1 byte (8 bits) y puede almacenar valores entre -128 y 127.
- **short**: Ocupa 2 bytes (16 bits) y puede almacenar valores entre -32,768 y 32,767.
- **int**: Ocupa 4 bytes (32 bits) y puede almacenar valores entre -2,147,483,648 y 2,147,483,647.
- **long**: Ocupa 8 bytes (64 bits) y puede almacenar valores entre 2^{63} y $2^{63}-1$. Para definir un valor de tipo `long`, se debe añadir una `L` al final del número.

```java
byte numeroByte = 100; // Tipo byte
short numeroShort = 30000; // Tipo short
int numeroEntero = 42; // Tipo int
long numeroLargo = 1234567890123L; // Tipo long
```

Consejo

Es posible separar los dígitos de un número entero con guiones bajos `_` para mejorar la legibilidad. Por ejemplo, `1_000_000` es equivalente a `1000000`.

Si necesitamos realizar operaciones con números enteros que superen el rango de los tipos primitivos, podemos utilizar la clase `BigInteger` de la biblioteca estándar de Java, que permite trabajar con enteros de tamaño arbitrario.

Debido a que estos números son representados en binario, es posible asignarles valores usando distintas bases numéricas, como la base decimal, octal, hexadecimal o binaria. Para ello, se utilizan los siguientes prefijos:

- **Decimal:** No requiere prefijo, por ejemplo, `42`.
- **Octal:** Se indica con un `0` al inicio, por ejemplo, `052` (equivalente a 42 en decimal).
- **Hexadecimal:** Se indica con un `0x` o `0X` al inicio, por ejemplo, `0x2A` (equivalente a 42 en decimal).
- **Binario:** Se indica con un `0b` o `0B` al inicio, por ejemplo, `0b101010` (equivalente a 42 en decimal).

Tipos de punto flotante

Los tipos de punto flotante en Java son:

- `float`: Ocupa 4 bytes (32 bits) y puede almacenar números con decimales. Para definir un valor de tipo `float`, se debe añadir una `F` al final del número.

- `double`: Ocupa 8 bytes (64 bits) y puede almacenar números con mayor precisión que `float`. Es el tipo de punto flotante más utilizado por defecto.

```
float numeroFloat = 3.14F; // Tipo float
double numeroDouble = 2.718281828459045; // Tipo double
```

Importante

Al representar números flotantes, Java utiliza la notación de coma flotante, lo que puede llevar a errores de precisión en algunos casos. Por ejemplo, el número 0,1 no se puede representar sin introducir errores de redondeo. Esto se debe a la forma en que los números de punto flotante se almacenan en binario, utilizando el estándar IEEE 754, el cual representa los números como fracciones con divisor un número potencia de 2.
Para evitar estos problemas, es recomendable utilizar `BigDecimal` para cálculos financieros o cuando se requiera alta precisión.

5.1.2. Tipos no numéricos

Los tipos no numéricos en Java son `char` y `boolean`.

Tipo carácter

El tipo `char` se utiliza para representar un único carácter Unicode. Ocupa 2 bytes (16 bits) y puede almacenar cualquier carácter, incluyendo letras, números y símbolos. Los caracteres se definen entre comillas simples.

```
char letra = 'A'; // Tipo char
char simbolo = '#'; // Tipo char
char digito = '5'; // Tipo char
char espacio = ' '; // Tipo char (espacio en blanco)
char saltoLinea = '\n'; // Tipo char (salto de línea)
```

Tipo booleano

El tipo `boolean` se utiliza para representar valores lógicos, es decir, verdadero (`true`) o falso (`false`). Este tipo es fundamental para el control de flujo en los programas, ya que se utiliza en estructuras de control como condicionales y bucles.

```
1 boolean esVerdadero = true; // Tipo boolean
2 boolean esFalso = false; // Tipo boolean
```

5.2. Operadores y expresiones

Los operadores son símbolos que permiten realizar operaciones sobre los valores de los tipos primitivos. En Java, existen varios tipos de operadores, entre ellos:

- **Operadores aritméticos:** Permiten realizar operaciones matemáticas básicas como suma (`+`), resta (`-`), multiplicación (`*`), división (`/`) y módulo (`%`).
- **Operadores de asignación:** Permiten asignar valores a variables. El operador de asignación más común es el igual (`=`), pero también existen operadores compuestos como `+=`, `-=`, `*=`, `/=` y `%=` que combinan una operación aritmética con la asignación.
- **Operadores de incremento y decremento:** Permiten aumentar o disminuir el valor de una variable en 1. Se utilizan los operadores `++`↪ (incremento) y `--` (decremento). Estos operadores pueden ser prefijos (`++x` o `--x`) o sufijos (`x++` o `x--`), lo que afecta el orden de evaluación en expresiones más complejas.
- **Operadores de comparación:** Permiten comparar dos valores y devuelven un valor booleano. Los operadores de comparación incluyen igual a (`==`), diferente de (`!=`), mayor que (`>`), menor que (`<`), mayor o igual que (`>=`) y menor o igual que (`<=`).
- **Operadores lógicos:** Permiten combinar expresiones booleanas. Los operadores lógicos incluyen AND lógico (`&&`), OR lógico (`||`), XOR lógico (`^`) y NOT lógico (`!`).
- **Operadores de bits:** Permiten realizar operaciones a nivel de bits sobre valores enteros. Los operadores de bits incluyen AND bit a bit (`&`), OR bit a bit (`|`), XOR bit a bit (`^`), desplazamiento a la izquierda (`<<`),

desplazamiento a la derecha (`>>`) y desplazamiento a la derecha sin signo (`>>>`).

Los operadores lógicos solo pueden aplicarse a valores de tipo `boolean`, mientras que el resto de operadores solo pueden aplicarse a los otros tipos primitivos. El resultado de cada operación será del mismo tipo que los operandos, excepto en el caso de los operadores de comparación, que siempre devuelven un valor booleano.

Todos estos operadores se pueden combinar para formar expresiones más complejas. Es importante tener en cuenta el orden de precedencia de los operadores, que determina el orden en que se evalúan las expresiones. Por ejemplo, las multiplicaciones y divisiones tienen mayor precedencia que las sumas y restas, por lo que se evalúan primero. Además, se pueden utilizar paréntesis para forzar un orden específico de evaluación en las expresiones.

```java
int a = 10;
int b = 5;
int suma = a + b; // Suma
int resta = a - b; // Resta
int multiplicacion = a * b; // Multiplicación
int division = a / b; // División
int modulo = a % b; // Módulo
int incremento = a++; // Incremento (postfijo). Devuelve el valor de a antes de incrementar, y luego incrementa a en 1.
int decremento = --b; // Decremento (prefijo). Decrementa el valor de b en 1 y luego devuelve el nuevo valor.
boolean esIgual = (a == b); // Comparación de igualdad
boolean esMayor = (a > b); // Comparación de mayor que
boolean resultadoLogico = (a > 0 && b < 10); // Operador lógico AND
int desplazamientoIzquierda = a << 5; // Desplazamiento a la izquierda. El valor de a se multiplica por 2, 5 veces.
int desplazamientoDerecha = a >> 3; // Desplazamiento a la derecha. El valor de a se divide por 2, 3 veces.
int andBit = a & b; // AND bit a bit
```

Ejercicio 5.1

Crea un programa que calcule el área de un rectángulo y el perímetro de un cuadrado. Utiliza variables de tipo `int` para almacenar las dimensiones y muestra los resultados en la consola.

5.3. Conversiones de tipo

En Java, es común que necesitemos convertir valores entre diferentes tipos primitivos. Existen dos tipos de conversiones: implícitas y explícitas.

5.3.1. Conversión implícita

La conversión implícita, también conocida como *promoción*, ocurre automáticamente cuando asignamos un valor de un tipo más pequeño a una variable de un tipo más grande. Por ejemplo, al asignar un valor de tipo `int` a una variable de tipo `long`, Java realiza la conversión automáticamente sin necesidad de especificarlo explícitamente.

```
1 int numeroEntero = 42;
2 long numeroLargo = numeroEntero; // Conversión implícita de int ↪
    a long
3 float numeroFloat = 3.14;
4 double numeroDecimal = numeroFloat; // Conversión implícita de ↪
    float a double
5 float numeroFloat = (float) numeroDecimal; // Conversión ↪
    implícita de double a float
6 float numeroFloat2 = numeroEntero; // Conversión implícita de ↪
    int a float
7 char letra = 'A';
8 int codigoAscii = letra; // Conversión implicita de char a int (↪
    código ASCII)
```

Importante

Al usar dos operandos de distintos tipos en una operación, Java promoverá automáticamente el tipo más pequeño al tipo más grande para evitar pérdida de información. Por ejemplo, si sumamos un `int` y un `double`, el `int` se convertirá a `double` antes de realizar la suma. Igualmente, si realizamos una operación entre un `float` y un `int`, el `int` se convertirá a `float`.

Podemos hacer uso de esta propiedad para realizar divisiones entre enteros y obtener un resultado de tipo `double` o `float`. Por ejemplo: `10 / (float)3` dará como resultado `3.3333333` en lugar de `3`, ya que el `int` se convierte a `float` antes de realizar la división.

5.3.2. Conversión explícita

La conversión explícita, también conocida como *casting*, se utiliza cuando queremos convertir un valor de un tipo más grande a un tipo más pequeño. En este

caso, debemos indicar explícitamente que queremos realizar la conversión utilizando paréntesis. Es importante tener en cuenta que al realizar una conversión explícita, puede haber pérdida de información si el valor original no cabe en el nuevo tipo.

```
long numeroLargo = 1234567890123L;
int numeroEntero = (int) numeroLargo; // Conversión explícita de long a int (puede perder información)
double numeroDecimal = 3.14;
float numeroFloat = (float) numeroDecimal; // Conversión explícita de double a float (puede perder precisión)
int numeroEntero2 = numeroDecimal; // Conversión explícita de double a int (pierde la parte decimal)
int codigoAscii = 20;
char letra = codigoAscii; // Conversión explícita de int a char
```

Consejo

Para realizar conversiones entre números de punto flotante y enteros, es recomendable utilizar la clase `Math`, en particular los métodos `Math.round()`, `Math.floor()` y `Math.ceil()` para redondear, truncar o redondear hacia arriba respectivamente.

Resumen

En este capítulo hemos aprendido los conceptos fundamentales sobre los tipos primitivos en Java. Los puntos clave son:

- Java ofrece varios tipos primitivos para representar datos simples: enteros (`byte`, `short`, `int`, `long`), punto flotante (`float`, `double`), caracteres (`char`) y valores lógicos (`boolean`).
- Cada tipo primitivo tiene un tamaño y un rango de valores específico, lo que influye en el uso de memoria y la precisión de los cálculos.
- Los operadores permiten manipular los valores de los tipos primitivos mediante operaciones aritméticas, lógicas, de comparación, de asignación y a nivel de bits.
- Las conversiones de tipo pueden ser implícitas (cuando Java realiza la conversión automáticamente) o explícitas (cuando el programador indica la conversión mediante *casting*), y es importante tener en cuenta la posible pérdida de información.

- Para cálculos que requieren alta precisión o valores fuera del rango de los tipos primitivos, existen clases como `BigInteger` y `BigDecimal`.

Comprender y utilizar correctamente los tipos primitivos es esencial para escribir programas eficientes y seguros en Java.

Ejercicio 5.2

Realiza las siguientes conversiones de tipo y muestra el resultado en la consola:

1. Convierte un valor de tipo `int` a `long` y muestra el resultado.
2. Convierte un valor de tipo `double` a `float` y muestra el resultado.
3. Convierte un valor de tipo `long` a `int` y muestra el resultado (ten en cuenta la posible pérdida de información).
4. Convierte un carácter a su código ASCII (tipo `int`) y muestra el resultado.

6

Utilización de objetos

En este capítulo se introducirá el concepto de objetos en Java, que son instancias de clases. Los objetos permiten encapsular datos y comportamientos relacionados, facilitando la organización y reutilización del código. Este capítulo, se divide de la siguiente manera:

- **Características de los objetos:** Los objetos tienen identidad, estado (propiedades o atributos) y comportamiento (métodos).
- **Instanciación de objetos:** como son creados los objetos.
- **Utilización de métodos, y el uso de parámetros:** Los métodos definen acciones que pueden realizar los objetos y pueden recibir parámetros para modificar su comportamiento.
- **Utilización de propiedades:** Las propiedades o atributos almacenan información sobre el estado del objeto y pueden ser accedidas o modificadas mediante métodos.
- **Utilización de métodos estáticos:** Los métodos estáticos pertenecen a la clase y no a los objetos; se invocan usando el nombre de la clase.
- **Constructores:** Son métodos especiales que se ejecutan al crear un objeto y permiten inicializar sus propiedades.
- **Destrucción de objetos y liberación de memoria:** En Java, la liberación de memoria se realiza automáticamente mediante el recolector de basura (*garbage collector*), que destruye los objetos que ya no se utilizan.

6.1. Características de los objetos

Un objeto en Java es una entidad que combina estado y comportamiento. El estado se representa mediante atributos (variables de instancia), y el comportamiento mediante métodos (funciones asociadas al objeto). La Estructura de un

objeto se define en una clase, que actúa como plantilla para crear instancias de objetos.

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

  void saludar() {
    System.out.println("Hola, mi nombre es " + nombre);
  }
}
```

Ejemplo 6.1: Ejemplo de clase y objeto

En el ejemplo anterior, `nombre` y `edad` son atributos, y `saludar()` es un método.

Importante

El concepto de clase se verá en detalle en el capítulo 8, donde se explicará cómo se definen y utilizan las clases en Java. Por ahora, es importante entender que una clase es una plantilla que define las propiedades y comportamientos de los objetos que se crearán a partir de ella.

6.2. Instanciación de objetos

Para crear un objeto, se utiliza el operador `new` seguido del constructor de la clase.

```
Persona persona1 = new Persona();
persona1.nombre = "Ana";
persona1.edad = 25;
persona1.saludar(); // Imprime: Hola, mi nombre es Ana
```

Ejemplo 6.2: Instanciación de un objeto

Una vez creada una clase, se pueden crear múltiples instancias de esa clase, cada una con su propio estado.

6.3. Utilización de métodos y el uso de parámetros

Los métodos pueden recibir parámetros para modificar su comportamiento. Por ejemplo:

```
class Calculadora {
  int sumar(int a, int b) {
    return a + b;
  }
}
```

Ejemplo 6.3: Método con parámetros

En este ejemplo, el método `sumar` recibe dos parámetros `a` y `b` y devuelve su suma. Para utilizar este método, se crea un objeto de la clase `Calculadora` y se llama al método:

```
Calculadora calc = new Calculadora();
int resultado = calc.sumar(3, 4); // resultado es 7
```

Ejemplo 6.4: Uso del método

6.4. Utilización de propiedades

Las propiedades de un objeto pueden ser accedidas y modificadas directamente si son públicas (ver ejemplo 6.2), aunque es recomendable usar métodos *getter* y *setter* para proteger el acceso.

```
class Persona {
  private String nombre;

  public String getNombre() {
    return nombre;
  }

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

Ejemplo 6.5: Uso de getters y setters

6.5. Utilización de métodos estáticos

Los métodos estáticos pertenecen a la clase y no a los objetos. Se invocan usando el nombre de la clase.

```
class Utilidades {
  static int cuadrado(int x) {
    return x * x;
  }
```

```
}
```

Ejemplo 6.6: Método estático

```
int resultado = Utilidades.cuadrado(5); // resultado es 25
```

Ejemplo 6.7: Uso de método estático

El uso de métodos estáticos es útil para funciones que no dependen del estado de un objeto específico, como utilidades matemáticas o funciones de ayuda.

Consejo

Son de utilidad los métodos estáticos de la clase `Math`, como `Math.sqrt()` para calcular raíces cuadradas, o `Math.random()` para generar números aleatorios. Estos métodos son ampliamente utilizados en programación y facilitan tareas comunes sin necesidad de crear instancias de objetos.
También son de gran utilidad los métodos de la clase `System`, como `System.out.println()` para imprimir en la consola, o `System.currentTimeMillis()` para obtener el tiempo actual en milisegundos. Estos métodos proporcionan funcionalidades esenciales para interactuar con el entorno de ejecución y son ampliamente utilizados en la mayoría de los programas Java.
Otros métodos estáticos que utilizarás en el futuro incluyen `String.valueOf()` para convertir otros tipos de datos a cadenas, o `Integer.parseInt()` para convertir cadenas a enteros. Estos métodos son fundamentales para la manipulación de datos y la conversión entre tipos de datos en Java.

6.6. Constructores

Un constructor es un método especial que se ejecuta al crear un objeto. Sirve para inicializar los atributos. Por defecto, Java proporciona un constructor sin parámetros si no se define ninguno. Sin embargo, es común definir constructores personalizados para establecer valores iniciales.

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

  Persona(String nombre, int edad) {
    this.nombre = nombre;
    this.edad = edad;
  }
```

```
}
```

Ejemplo 6.8: Definición de constructor

```
Persona persona2 = new Persona("Luis", 30);
```

Ejemplo 6.9: Uso del constructor

Es posible que una clase tenga múltiples constructores, cada uno con diferentes parámetros. Esto se conoce como *sobrecarga de constructores*. Además, si no se define un constructor, Java proporciona uno por defecto que no toma parámetros y no inicializa los atributos.

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

  // Constructor sin parámetros
  Persona() {
    this.nombre = "Desconocido";
    this.edad = 0;
  }

  // Constructor con un parámetro
  Persona(String nombre) {
    this.nombre = nombre;
    this.edad = 0;
  }

  // Constructor con dos parámetros
  Persona(String nombre, int edad) {
    this.nombre = nombre;
    this.edad = edad;
  }

  void mostrarInformacion() {
    System.out.println("Nombre: " + nombre + ", Edad: " + edad);
  }
}
```

Ejemplo 6.10: Clase con varios constructores

De esta forma, se pueden crear objetos de la clase `Persona` usando diferentes constructores:

```
Persona p1 = new Persona();
Persona p2 = new Persona("Ana");
Persona p3 = new Persona("Luis", 30);

p1.mostrarInformacion(); // Imprime: Nombre: Desconocido, Edad: ↪
  0
```

```
p2.mostrarInformacion(); // Imprime: Nombre: Ana, Edad: 0
p3.mostrarInformacion(); // Imprime: Nombre: Luis, Edad: 30
```

Ejemplo 6.11: Uso de varios constructores

6.7. Destrucción de objetos y liberación de memoria

En Java, la gestión de memoria es automática. El recolector de basura (*garbage collector*) elimina los objetos que ya no tienen referencias. No es necesario liberar memoria manualmente, aunque se puede sugerir la recolección con `System.gc()`, pero su ejecución no está garantizada.

Importante

Aunque Java maneja automáticamente la memoria, es importante evitar referencias circulares entre objetos, ya que pueden impedir que el recolector de basura libere memoria adecuadamente.
Además, es recomendable cerrar recursos como archivos o conexiones de red utilizando bloques `try-with-resources` o asegurándose de llamar a los métodos `close()` correspondientes para liberar recursos externos.

6.8. La clase `String`

La clase `String` en Java es una de las más utilizadas y tiene características particulares que la diferencian de otros tipos de objetos.

6.8.1. Inmutabilidad de los objetos `String`

Los objetos de tipo `String` son **inmutables**, es decir, una vez creados, su valor no puede cambiar. Si se realiza una operación que parece modificar una cadena, en realidad se crea un nuevo objeto `String` con el resultado.

```
String saludo = "Hola";
saludo = saludo + " mundo"; // Se crea un nuevo objeto String
```

Ejemplo 6.12: Inmutabilidad de String

En el ejemplo anterior, la variable `saludo` apunta primero a una cadena y luego a otra, pero el objeto original no se modifica.

6.8.2. Creación de cadenas

Las cadenas pueden crearse de varias formas:

```
String s1 = "texto"; // Literal
String s2 = new String("texto"); // Constructor (no recomendado)
```

Ejemplo 6.13: Formas de crear cadenas

La forma recomendada es usar literales, ya que es más eficiente y permite el uso del *pool* de cadenas.

6.8.3. Concatenación y métodos útiles

La concatenación de cadenas puede hacerse con el operador `+` o usando el método `concat()`:

```
String nombre = "Ana";
String saludo = "Hola, " + nombre;
```

Ejemplo 6.14: Concatenación de cadenas

Algunos métodos útiles de la clase `String` son:

- `length()`: Devuelve la longitud de la cadena.
- `toUpperCase()`, `toLowerCase()`: Convierte la cadena a mayúsculas o minúsculas.
- `charAt(int index)`: Devuelve el carácter en la posición indicada.
- `substring(int beginIndex, int endIndex)`: Extrae una subcadena.
- `equals(String other)`: Compara el contenido de dos cadenas.
- `trim()`: Elimina espacios en blanco al inicio y al final.

```
String texto = " Java ";
int longitud = texto.length(); // 7
String mayusculas = texto.toUpperCase(); // " JAVA "
String sinEspacios = texto.trim(); // "Java"
boolean igual = texto.trim().equals("Java"); // true
```

Ejemplo 6.15: Ejemplo de métodos de String

6.8.4. Comparación de cadenas

Para comparar el contenido de dos cadenas, se debe usar el método `equals()` y no el operador `==`, ya que este último compara referencias, no contenido.

```
String a = "hola";
String b = "hola";
boolean iguales = a.equals(b); // true
boolean mismasReferencias = (a == b); // true en este caso, pero no siempre
```

Ejemplo 6.16: Comparación de cadenas

Importante

Recuerda siempre usar `equals()` para comparar cadenas por su contenido. El operador `==` solo debe usarse para comparar si dos referencias apuntan al mismo objeto.

Resumen

En este capítulo se ha presentado el concepto de objeto en Java, su creación, uso de métodos y atributos, métodos estáticos, constructores y la gestión automática de memoria. Estos conceptos son fundamentales para la programación orientada a objetos y la construcción de aplicaciones robustas y reutilizables.

Ejercicio 6.1

Crea una clase `Coche` que tenga los siguientes atributos: `marca`, `modelo` y `año`. Implementa un constructor que inicialice estos atributos y un método `mostrarDetalles()` que imprima la información del coche en la consola. Luego, crea un objeto de la clase `Coche` y llama al método `mostrarDetalles()`.

El atributo `año` debe ser un número entero, mientras que `marca` y `modelo` deben ser cadenas de texto.

Al imprimir los detalles, la marca deberá aparecer en mayúsculas, el modelo en minúsculas y el año como un número entero. Por ejemplo, si el coche es un Ford Focus del año 2020, la salida debería ser `"FORD focus 2020"`.

7

Uso de estructuras de control

En la programación, las estructuras de control son fundamentales para definir el flujo de ejecución de un programa. Estas estructuras permiten tomar decisiones, repetir acciones y modificar el comportamiento del software según diferentes condiciones:

- **Estructuras de selección:** Permiten que el programa elija entre diferentes caminos de ejecución, dependiendo del cumplimiento de ciertas condiciones.
- **Estructuras de repetición:** Facilitan la ejecución de un bloque de código varias veces, ya sea un número determinado de veces o mientras se cumpla una condición.
- **Estructuras de salto:** Permiten alterar el flujo normal de ejecución, transfiriendo el control a otra parte del programa.
- **Control de excepciones:** Es un mecanismo que permite manejar situaciones inesperadas o errores durante la ejecución del programa, evitando que el software falle abruptamente.

7.1. Estructuras de selección

Las estructuras de selección permiten tomar decisiones en función de condiciones lógicas.

7.1.1. Sentencias condicionales: `if`, `else if`, y `else`

Las sentencias condicionales más comunes son `if`, `else if` y `else`. Estas permiten ejecutar diferentes bloques de código según se cumplan o no ciertas condiciones.

```java
int edad = 18;
if (edad >= 18) {
  System.out.println("Eres mayor de edad");
} else {
  System.out.println("Eres menor de edad");
}
```

Ejemplo 7.1: Ejemplo de sentencia if-else en Java

En este ejemplo, si la variable `edad` es mayor o igual a 18, se imprime `"Eres mayor de edad"`. De lo contrario, se imprime `"Eres menor de edad"`. Es posible incluir múltiples condiciones utilizando `else if` para manejar diferentes casos:

```java
int nota = 85;
if (nota >= 90) {
  System.out.println("Excelente");
} else if (nota >= 70) {
  System.out.println("Aprobado");
} else {
  System.out.println("Reprobado");
}
```

Ejemplo 7.2: Ejemplo de if-else if-else en Java

En este caso, se evalúa la variable `nota` y se imprime un mensaje diferente según el rango en el que se encuentre. Cada bloque de código se ejecuta solo si la condición correspondiente es verdadera, y si ninguna condición anterior se cumple.

Además, también es posible escribir condiciones sin bloque `else` para ejecutar una acción específica si la condición es verdadera, sin necesidad de un bloque alternativo:

```java
int temperatura = 30;
if (temperatura > 25)
  System.out.println("Hace calor");
```

Ejemplo 7.3: Ejemplo de if sin else en Java

Consejo

Recuerda que es importante utilizar llaves `()` para agrupar las condiciones y bloques de código, especialmente cuando se trabaja con múltiples líneas de código dentro de un bloque `if` o `else`. Esto mejora la legibilidad y evita errores en la ejecución del programa.

Es recomendable utilizar comentarios para explicar la lógica detrás de las condiciones, especialmente en casos más complejos.

Es posible crear condiciones compuestas utilizando operadores lógicos co-

mo `&&` (AND) y `||` (OR).

7.1.2. Sentencias de selección: `Switch`

Otra estructura de selección es **`switch`**, útil cuando se desea comparar una variable contra varios valores posibles:

```
int dia = 3;
switch (dia) {
  case 1:
    System.out.println("Lunes");
    break;
  case 2:
    System.out.println("Martes");
    break;
  case 3:
    System.out.println("Miércoles");
    break;
  case 6:
  case 7:
    System.out.println("Fin de semana");
    break;
  default:
    System.out.println("Otro día");
}
```

Ejemplo 7.4: Ejemplo de switch en Java

En este ejemplo, la variable `dia` se compara con diferentes casos. Si coincide con alguno, se ejecuta el bloque de código correspondiente. El uso de **`break`** es importante para evitar que el programa continúe ejecutando los siguientes casos después de encontrar una coincidencia. Si no se encuentra ninguna coincidencia, se ejecuta el bloque **`default`**.

Consejo

El uso de **`switch`** es especialmente útil cuando se tienen múltiples condiciones basadas en el mismo valor, ya que mejora la legibilidad del código en comparación con múltiples sentencias **`if-else if`**.
Si varios casos comparten el mismo bloque de código, se pueden agrupar sin necesidad de repetir el código, como se muestra en el ejemplo con los casos 6 y 7.

Además, es posible usar la sentencia **`switch`** con cadenas de texto a partir de Java 7, lo que permite comparar una variable de tipo `String` con diferentes valores:

```java
String dia = "Lunes";
switch (dia) {
  case "Lunes":
    System.out.println("Hoy es lunes");
    break;
  case "Martes":
    System.out.println("Hoy es martes");
    break;
  case "Miércoles":
    System.out.println("Hoy es miércoles");
    break;
  default:
    System.out.println("Otro día");
}
```

Ejemplo 7.5: Ejemplo de switch con String en Java

Otra de las peculiaridades de la sentencia `switch` es que permite devolver un valor.

```java
public String obtenerDiaSemana(int dia) {
  return switch (dia) {
    case 1 -> "Lunes";
    case 2 -> "Martes";
    case 3 -> "Miércoles";
    case 4 -> "Jueves";
    case 5 -> "Viernes";
    case 6, 7 -> "Fin de semana";
    default -> "Día inválido";
  };
}
```

Ejemplo 7.6: Switch que devuelve un valor en Java

En este ejemplo, el método `obtenerDiaSemana` utiliza la sentencia `switch` para devolver directamente un valor según el caso, usando la sintaxis de expresión introducida en Java 14.

Importante

La sentencia `switch` también puede emplearse para convertir objetos a diferentes clases.

Por ejemplo, si tienes una variable de tipo `Object` y necesitas ejecutar acciones distintas según su tipo específico, puedes utilizar `switch` para realizar el casteo de manera segura:

```java
  public void procesar(Object obj) {
    switch (obj) {
      case String s -> System.out.println("Cadena: " + s.toUpperCase());
```

```
4       case Integer i -> System.out.println("Entero: " + (i
    * 2));
5       case Double d -> System.out.println("Decimal: " + (d
    / 2));
6       default -> System.out.println("Tipo desconocido");
7     }
8   }
9
```

Ejemplo 7.7: Ejemplo de switch para convertir objetos en Java

En este caso, el método `procesar` utiliza `switch` para determinar el tipo del objeto recibido y ejecutar una acción específica para cada tipo, realizando la conversión automáticamente en cada caso.

Ejercicio 7.1

Escribe un programa que imprima por pantalla si un año es bisiesto o no. Un año es bisiesto si es divisible por 4, excepto los años que son divisibles por 100, a menos que también sean divisibles por 400.
Utiliza una estructura de selección adecuada para resolver el problema.

7.2. Estructuras de repetición

Permiten ejecutar un bloque de código varias veces.

7.2.1. Bucle `for`

El bucle `for` es ideal cuando se conoce de antemano el número de iteraciones que se deben realizar. Su sintaxis es la siguiente:

```
1 for (int i = 0; i < 5; i++) {
2   System.out.println("Iteración " + i);
3 }
```

Ejemplo 7.8: Ejemplo de bucle for en Java

En este ejemplo, el bucle se ejecuta 5 veces, imprimiendo el número de iteración en cada paso. La variable `i` se inicializa en 0, y se incrementa en 1 en cada iteración hasta que alcanza 5.

Este tipo de bucle es el más apropiado cuando se necesita iterar sobre un rango de números o elementos de una colección, ya que permite controlar fácilmente el inicio, la condición de continuación y el incremento.

```
1 String texto = "Hola";
```

```java
for (int i = 0; i < texto.length(); i++) {
  char caracter = texto.charAt(i);
  System.out.println("Carácter en posición " + i + ": " + caracter);
}
```

Ejemplo 7.9: Iterar sobre los caracteres de un String en Java

En este ejemplo, el bucle recorre la cadena `texto` desde la posición 0 hasta la longitud de la cadena menos uno, imprimiendo cada carácter junto con su posición.

Importante

Todos los elementos del bucle `for` son opcionales, lo que significa que puedes omitir la inicialización, la condición y el incremento. Sin embargo, es importante tener cuidado al hacerlo, ya que puede llevar a bucles infinitos si no se controla adecuadamente.

7.2.2. Bucle `while`, y `do-while`

El bucle `while` se utiliza cuando no se conoce de antemano el número de iteraciones, y se ejecuta mientras una condición sea verdadera. Su sintaxis es la siguiente:

```java
int contador = 0;
while (contador < 5) {
  System.out.println("Contador: " + contador);
  contador++;
}
```

Ejemplo 7.10: Ejemplo de bucle while en Java

Este bucle continuará ejecutándose mientras la variable `contador` sea menor que 5. En cada iteración, se imprime el valor del contador y luego se incrementa en 1.

También existe el bucle `do-while`, que garantiza que el bloque de código se ejecute al menos una vez antes de evaluar la condición:

```java
int contador = 0;
do {
  System.out.println("Contador: " + contador);
  contador++;
} while (contador < 5);
```

Ejemplo 7.11: Ejemplo de bucle do-while en Java

Usaremos el bucle `while` cuando la condición de continuación no dependa de un contador fijo, sino de una condición que puede cambiar durante la ejecución

del programa. Por ejemplo, cuando iteremos sobre una colección usando un iterador (capítulo 10).

Importante

Es posible crear bucles infinitos si la condición nunca se vuelve falsa. Por ejemplo, si olvidamos incrementar el contador en el bucle `while` anterior, el programa se quedará atascado en un bucle infinito. Siempre asegúrate de que la condición eventualmente se vuelva falsa para evitar este problema.

Ejercicio 7.2

Crea un programa que imprime por pantalla los primeros 10 números de la serie de Fibonacci.
La serie de Fibonacci comienza con 0 y 1, y cada número siguiente es la suma de los dos anteriores: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

7.3. Estructuras de salto

Las estructuras de salto permiten modificar el flujo de ejecución de manera abrupta.

7.3.1. Sentencia `return`

La sentencia `return` se utiliza para salir de un método y, opcionalmente, devolver un valor. Cuando se ejecuta `return`, el control del programa regresa al punto donde se llamó al método.

```
1 public int sumar(int a, int b) {
2   return a + b;
3 }
```

Ejemplo 7.12: Ejemplo de return en Java

Es posible utilizar `return` para salir de un método antes de que se alcance el final del mismo, lo que puede ser útil para evitar la ejecución de código innecesario o para manejar condiciones especiales.

```
1 public boolean contiene(String texto, char caracter) {
2   if (texto == null || texto.isEmpty()) {
3     return false; // Si el texto es nulo o vacío, retorna false
4   }
5   for (int i = 0; i < texto.length(); i++) {
6     if (texto.charAt(i) == caracter) {
7       return true; // Sale del método en cuanto encuentra un ↪
  cero
```

```
    }
  }
  return false; // Si no encuentra ningún cero, retorna false
}
```

Ejemplo 7.13: Uso de return dentro de un bucle for en Java

En este ejemplo, el método `contiene` busca un carácter específico en una cadena de texto. Si encuentra el carácter, utiliza `return` para salir del método inmediatamente, devolviendo `true`. Si no lo encuentra, al final del bucle se devuelve `false`.

Ejercicio 7.3

Recupera el ejercicio 7.1 y modifica el programa para que retorne `true` si el año es bisiesto y `false` en caso contrario. Utiliza una estructura de control adecuada para resolver el problema.
Crea un método llamado `esBisiesto` que reciba un año como parámetro y retorne un valor booleano indicando si el año es bisiesto o no.
Después, imprime la lista de los siguientes 20 años bisiestos a partir del año actual, utilizando el método `esBisiesto` para determinar si cada año es bisiesto o no.

7.3.2. Sentencia `break` y `continue`

La sentencia `break` se utiliza para salir de un bucle o de una estructura de selección `switch` de manera inmediata. Por ejemplo, en un bucle `for` o `while`, al ejecutar `break`, el control del programa salta al final del bucle.

```
for (int i = 0; i < 10; i++) {
  if (i == 5) {
    break; // Sale del bucle cuando i es igual a 5
  }
  System.out.println("Número: " + i);
}
```

Ejemplo 7.14: Ejemplo de break en Java

En este ejemplo, el bucle se detiene cuando `i` alcanza el valor 5, y no se imprimen los números posteriores.

La sentencia `continue` se utiliza para saltar la iteración actual de un bucle y continuar con la siguiente iteración. Esto es útil cuando se desea omitir ciertas condiciones sin salir completamente del bucle.

```
for (int i = 0; i < 10; i++) {
  if (i % 2 == 0) {
    continue; // Salta la iteración si i es par
```

```
    }
    System.out.println("Número impar: " + i);
}
```

Ejemplo 7.15: Ejemplo de continue en Java

En este ejemplo, el bucle imprime solo los números impares del 0 al 9, ya que la sentencia `continue` salta las iteraciones donde `i` es par.

7.4. Control de excepciones

El control de excepciones permite manejar errores que pueden ocurrir durante la ejecución del programa, evitando que este termine inesperadamente.

```
try {
    int resultado = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("No se puede dividir por cero");
} finally {
    System.out.println("Fin del manejo de excepciones");
}
```

Ejemplo 7.16: Manejo de excepciones en Java

En este ejemplo, el bloque `try` contiene código que podría generar una excepción (en este caso, una división por cero). Si ocurre una excepción, el control pasa al bloque `catch`, donde se maneja el error. El bloque `finally` se ejecuta siempre, independientemente de si se produjo una excepción o no.

Si invocamos un método que declara que puede lanzar una excepción, debemos manejarla adecuadamente utilizando un bloque `try-catch` o declarando la excepción en la firma del método.

```
public class DivisionEjemplo {
    public static void main(String[] args) {
        DivisionEjemplo ejemplo = new DivisionEjemplo();
        try {
            ejemplo.dividir(10, 2);
            ejemplo.dividir(10, 0);
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

    public void dividir(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("No se puede dividir por ↪
    cero");
        }
        System.out.println("Resultado: " + (a / b));
```

```
17    }
18 }
```

Ejemplo 7.17: Declaración de excepciones en un método Java

Importante

El uso de excepciones es crucial para crear programas robustos y confiables. Permite anticiparse a posibles errores y manejar situaciones inesperadas de manera controlada, mejorando la experiencia del usuario y la estabilidad del software.
Esto es muy importante cuando se trabaja con entradas del usuario (como veremos en el capítulo 11), operaciones de red o acceso a bases de datos, donde los errores pueden ser comunes y deben ser gestionados adecuadamente para evitar fallos en la aplicación.

7.5. Recursividad

Ahora que conoces las estructuras de control básicas, es importante mencionar la recursividad, que es una técnica donde un método se llama a sí mismo para resolver un problema. Aunque no es una estructura de control en sí misma, es una forma de controlar el flujo de ejecución.

Este es un concepto avanzado, pero es útil para resolver problemas que pueden dividirse en subproblemas más pequeños. Por ejemplo, el cálculo del factorial de un número:

```
1 public int factorial(int n) {
2   if (n == 0) {
3     return 1; // Caso base
4   } else {
5     return n * factorial(n - 1); // Llamada recursiva
6   }
7 }
```

Ejemplo 7.18: Ejemplo de recursividad en Java

En este ejemplo, el método `factorial` se llama a sí mismo con un valor decreciente de `n` hasta llegar al caso base (`n == 0`), donde retorna 1. La recursividad es una herramienta poderosa, pero debe usarse con cuidado para evitar desbordamientos de pila (*stack overflow*) si no se define un caso base adecuado. En el ejemplo anterior, el caso base es cuando `n` es igual a 0, pero si se llama al método con un valor negativo, se generará un error de pila.

Resumen

En este capítulo hemos aprendido a utilizar las principales estructuras de control en programación: selección (`if`, `else if`, `else`, `switch`), repetición (`for`, `while`), salto (`return`, `break`, `continue`) y control de excepciones (`try-catch-finally`). Estas herramientas permiten modificar el flujo de ejecución de los programas, tomar decisiones, repetir acciones y manejar errores de forma controlada. Dominar su uso es fundamental para desarrollar programas robustos, legibles y eficientes.

Importante

Es posible combinar diferentes estructuras de control para crear programas más complejos y adaptados a las necesidades específicas de cada situación. Por ejemplo, se pueden utilizar bucles dentro de estructuras de selección, o viceversa, para lograr un control más preciso del flujo de ejecución.
Sin embargo, es importante evitar la complejidad excesiva en el código, ya que esto puede dificultar su comprensión y mantenimiento. Utiliza comentarios y nombres descriptivos para las variables y métodos, lo que ayudará a otros desarrolladores (y a ti mismo en el futuro) a entender mejor la lógica del programa.
También es recomendable seguir las buenas prácticas de programación, como la indentación adecuada y el uso de espacios en blanco, para mejorar la legibilidad del código. Esto facilitará la colaboración en equipo y el mantenimiento a largo plazo del software.
Una regla general es no anidar demasiadas estructuras de control, ya que esto puede hacer que el código sea difícil de seguir. En su lugar, considera dividir el código en métodos más pequeños y manejables que realicen tareas específicas, como habrás podido comprobar en el ejercicio de los años bisiestos.

8

Desarrollo de clases

En este capítulo se abordan los conceptos fundamentales relacionados con el desarrollo de clases en programación orientada a objetos:

1. **Concepto de clase:** Una clase es una plantilla o modelo que define las características (atributos) y comportamientos (métodos) que tendrán los objetos creados a partir de ella.

2. **Estructura y miembros de una clase. Visibilidad:** Una clase está compuesta por miembros, que pueden ser atributos (variables) o métodos (funciones).

3. **Creación de propiedades:** Las propiedades son atributos que almacenan información sobre el estado de un objeto. Se definen dentro de la clase y pueden tener distintos niveles de acceso.

4. **Creación de métodos:** Los métodos son funciones que definen el comportamiento de los objetos. Se implementan dentro de la clase y pueden manipular los atributos o realizar operaciones específicas.

5. **Creación de constructores:** El constructor es un método especial que se ejecuta al crear un objeto. Se utiliza para inicializar los atributos de la clase.

6. **Utilización de clases y objetos:** Para utilizar una clase, se crean objetos (instancias) a partir de ella. Los objetos permiten acceder a los atributos y métodos definidos en la clase.

7. **Utilización de clases heredadas:** La herencia permite crear nuevas clases basadas en clases existentes, reutilizando y extendiendo su funcionalidad.

8.1. Concepto de clase

En Java, una **clase** es una plantilla para crear objetos. Define tanto los atributos (propiedades que describen el estado del objeto) como los métodos (acciones o comportamientos que puede realizar el objeto). Las clases permiten organizar y estructurar el código, facilitando la reutilización y el mantenimiento. Cada objeto creado a partir de una clase se denomina *instancia* y posee sus propios valores para los atributos definidos en la clase.

Por ejemplo, una clase `Coche` puede representar las características y comportamientos comunes a todos los coches, como el color, la marca y la velocidad, así como las acciones de acelerar o frenar. A partir de esta clase, se pueden crear múltiples objetos, cada uno con sus propios valores para los atributos.

```
1  public class Coche {
2    String color;
3    String marca;
4    int velocidad;
5
6    public void acelerar(int incremento) {
7      velocidad += incremento;
8    }
9
10   public void frenar(int decremento) {
11     velocidad -= decremento;
12   }
13 }
```

Ejemplo 8.1: Ejemplo de clase en Java

8.2. Estructura y miembros de una clase. Visibilidad

En Java, los miembros de una clase (atributos y métodos) pueden tener distintos niveles de visibilidad, lo que determina desde dónde pueden ser accedidos. Los modificadores de acceso más comunes son:

- `public`: El miembro es accesible desde cualquier otra clase.
- `private`: El miembro solo es accesible dentro de la propia clase.
- `protected`: El miembro es accesible dentro de la propia clase, en las clases del mismo paquete y en las subclases (incluso si están en otros paquetes).
- *Sin modificador* (también llamado *package-private*): El miembro es accesible solo dentro de las clases del mismo paquete.

El uso adecuado de la visibilidad permite proteger los datos internos de la clase y controlar cómo se accede y modifica su estado. Por ejemplo, es una buena práctica declarar los atributos como `private` y proporcionar métodos públicos (`getters` y `setters`) para acceder o modificar su valor, siguiendo el principio de encapsulamiento.

```
1  public class Persona {
2    public String nombre;    // Público: accesible desde cualquier clase
3    protected int edad;     // Protegido: accesible desde subclases y el mismo paquete
4    String direccion;      // Sin modificador: accesible solo en el mismo paquete
5    private String dni;     // Privado: accesible solo dentro de la clase
6
7    public Persona(String nombre, String dni) {
8      this.nombre = nombre;
9      this.dni = dni;
10   }
11
12   // Getter y setter para el atributo privado dni
13   public String getDni() {
14     return dni;
15   }
16
17   public void setDni(String dni) {
18     this.dni = dni;
19   }
20 }
```

Ejemplo 8.2: Visibilidad de miembros en Java

En el ejemplo anterior, se observa cómo cada atributo tiene un nivel de visibilidad diferente. El atributo `dni` es privado y solo puede ser accedido mediante los métodos públicos `getDni()` y `setDni()`. Esto ayuda a mantener la integridad de los datos y a cumplir con el principio de encapsulamiento.

8.3. Creación de propiedades

En Java, las propiedades de una clase suelen implementarse como atributos privados, siguiendo el principio de encapsulamiento. Para acceder y modificar estos atributos desde fuera de la clase, se utilizan métodos públicos denominados `getters` y `setters`. Esta práctica permite controlar cómo se accede y modifica el estado interno del objeto, validando los valores si es necesario y evitando modificaciones no deseadas.

Ejemplo: Supongamos una clase `Circulo` con una propiedad `radio`. El atributo `radio` es privado y solo puede ser accedido o modificado mediante los métodos `getRadio()` y `setRadio(double radio)`. El método `setRadio` incluye una validación para asegurar que el radio sea positivo.

```
public class Circulo {
  private double radio;

  public Circulo(double radio) {
    setRadio(radio); // Se usa el setter para validar el valor inicial
  }

  // Getter: devuelve el valor del radio
  public double getRadio() {
    return radio;
  }

  // Setter: permite modificar el valor del radio con validación
  public void setRadio(double radio) {
    if (radio > 0) {
      this.radio = radio;
    }
  }
}
```

Ejemplo 8.3: Uso de getters y setters

Ventajas de usar getters y setters:

- Permiten validar los valores antes de asignarlos a los atributos.
- Facilitan el mantenimiento y la evolución del código, ya que se puede modificar la lógica interna sin afectar a las clases que usan la propiedad.
- Mejoran la seguridad y el control sobre el acceso a los datos internos de la clase.

Uso de propiedades:

```
Circulo c = new Circulo(5.0);
System.out.println(c.getRadio()); // Imprime 5.0
c.setRadio(10.0);          // Cambia el radio a 10.0
System.out.println(c.getRadio()); // Imprime 10.0
c.setRadio(-3.0);          // No cambia el radio, ya que el valor es inválido
System.out.println(c.getRadio()); // Imprime 10.0
```

Ejemplo 8.4: Acceso a propiedades mediante getters y setters

Consejo

Aunque en Java no existe un concepto de *propiedades* como en otros lenguajes (por ejemplo, C#), el uso de getters y setters es una práctica común para simular este comportamiento y mantener el encapsulamiento.

Ejercicio 8.1

Crea una clase llamada `Persona` con los siguientes atributos privados: `nombre` (`String`), `edad` (`int`) y `email` (`String`). Implementa métodos que expongan las propiedades de la clase, siendo todas de lectura, excepto el email, que deberá ser de lectura y escritura.

8.4. Creación de métodos

En Java, los métodos son funciones que definen el comportamiento de los objetos. Se pueden clasificar en dos tipos principales:

- **Métodos de instancia:** Operan sobre los atributos de un objeto específico. Para invocarlos, es necesario crear una instancia de la clase.
- **Métodos estáticos:** Se definen con la palabra clave `static` y pertenecen a la clase en sí, no a un objeto concreto. Se pueden llamar sin crear una instancia de la clase.

Los métodos pueden recibir parámetros y devolver valores. Además, pueden tener distintos niveles de visibilidad (`public`, `private`, etc.), lo que permite controlar desde dónde pueden ser accedidos.

Ejemplo:

```
public class Calculadora {
  // Método de instancia: requiere un objeto para ser llamado
  public int suma(int a, int b) {
    return a + b;
  }

  // Método estático: se puede llamar sin crear un objeto
  public static int resta(int a, int b) {
    return a - b;
  }
}
```

Ejemplo 8.5: Métodos de instancia y estáticos

Uso de métodos:

```java
Calculadora calc = new Calculadora();
int resultadoSuma = calc.suma(5, 3);       // Llamada a método de instancia
int resultadoResta = Calculadora.resta(5, 3); // Llamada a método estático
```

Ejemplo 8.6: Invocación de métodos

Ventajas de los métodos:

- Permiten reutilizar código y organizar la lógica en bloques funcionales.
- Facilitan el mantenimiento y la legibilidad del código.
- Permiten encapsular el comportamiento y proteger los datos internos de la clase.

Ejercicio 8.2

Crea una clase llamada `Rectangulo` que tenga dos atributos privados: `base` y `altura` (de tipo `double`). Implementa los siguientes métodos:

- Un constructor que reciba la base y la altura.
- Métodos `getBase()` y `getAltura()` para acceder a los atributos.
- Un método de instancia `area()` que devuelva el área del rectángulo.
- Un método estático `esCuadrado(Rectangulo r)` que devuelva `true` si la base y la altura son iguales, y `false` en caso contrario.

Prueba la clase creando un objeto y mostrando el área y si es cuadrado.

8.5. Creación de constructores

En Java, un **constructor** es un método especial que se utiliza para inicializar los objetos de una clase. El constructor tiene el mismo nombre que la clase y no tiene tipo de retorno (ni siquiera `void`). Se invoca automáticamente cuando se crea un objeto usando la palabra clave `new`.

Características principales de los constructores:

- Pueden recibir parámetros para inicializar los atributos del objeto.
- Si no se define ningún constructor, Java proporciona uno por defecto (sin parámetros).

- Se pueden definir varios constructores en una clase (sobrecarga de constructores), siempre que tengan diferentes listas de parámetros.

- Los constructores pueden llamar a otros constructores de la misma clase usando `this(...)` o al constructor de la superclase usando `super(...)`.

Ejemplo básico de constructor:

```java
public class Animal {
  private String especie;
  private int edad;

  // Constructor que inicializa los atributos
  public Animal(String especie, int edad) {
    this.especie = especie;
    this.edad = edad;
  }
}
```

Ejemplo 8.7: Constructor en una clase Java

Ejemplo de sobrecarga de constructores:

```java
public class Animal {
  private String especie;
  private int edad;

  // Constructor con parámetros
  public Animal(String especie, int edad) {
    this.especie = especie;
    this.edad = edad;
  }

  // Constructor sin parámetros (por defecto)
  public Animal() {
    this.especie = "Desconocida";
    this.edad = 0;
  }
}
```

Ejemplo 8.8: Sobrecarga de constructores

Uso de constructores:

```java
Animal a1 = new Animal("Perro", 3);
Animal a2 = new Animal();
```

Ejemplo 8.9: Creación de objetos usando constructores

Ejercicio 8.3

Crea una clase llamada `Libro` con los siguientes atributos privados: `titulo` (String), `autor` (String) y `paginas` (int). Implementa:

- Un constructor que reciba los tres atributos como parámetros.
- Un constructor sin parámetros que asigne valores por defecto.
- Métodos `getTitulo()`, `getAutor()` y `getPaginas()`.

Crea dos objetos `Libro`, uno usando cada constructor, y muestra sus datos por pantalla.

8.6. Utilización de clases y objetos

Para usar una clase, se crean objetos (instancias) con la palabra clave `new`. Una vez creado el objeto, se puede acceder a sus atributos y métodos utilizando el operador punto (`.`). Cada objeto tiene su propio estado, es decir, sus propios valores para los atributos definidos en la clase.

Ejemplo básico:

```
Coche miCoche = new Coche();    // Se crea un objeto de la clase Coche
miCoche.color = "rojo";       // Se asigna el color
miCoche.marca = "Toyota";      // Se asigna la marca
miCoche.acelerar(20);        // Se llama al método acelerar
System.out.println(miCoche.velocidad); // Se muestra la velocidad actual
```

Ejemplo 8.10: Creación y uso de objetos en Java

Acceso a métodos y atributos:

Cuando los atributos son privados, se accede a ellos mediante métodos públicos (`getters` y `setters`), siguiendo el principio de encapsulamiento:

```
Circulo c = new Circulo(5.0);
System.out.println("Radio: " + c.getRadio());
c.setRadio(10.0);
System.out.println("Nuevo radio: " + c.getRadio());
```

Ejemplo 8.11: Acceso a atributos privados mediante getters y setters

Creación de múltiples objetos:

Cada objeto creado a partir de una clase es independiente de los demás:

```
Coche coche1 = new Coche();
Coche coche2 = new Coche();
```

```
3
4 coche1.color = "azul";
5 coche2.color = "verde";
6
7 System.out.println(coche1.color); // azul
8 System.out.println(coche2.color); // verde
```

Ejemplo 8.12: Instanciación de varios objetos

Resumen de pasos para utilizar una clase:

1. Declarar una variable de tipo de la clase.
2. Crear el objeto con `new` y el constructor correspondiente.
3. Acceder a los atributos y métodos mediante el operador punto.

Ejercicio 8.4

Crea una clase llamada `CuentaBancaria` con los atributos privados `titular` (String) y `saldo` (double). Implementa:

- Un constructor que reciba el titular y el saldo inicial.
- Métodos `getTitular()` y `getSaldo()`.
- Un método `depositar(double cantidad)` que aumente el saldo.
- Un método `retirar(double cantidad)` que disminuya el saldo solo si hay suficiente dinero.

Crea un objeto `CuentaBancaria`, realiza un depósito y un retiro, y muestra el saldo final.

8.7. Utilización de clases heredadas

La herencia es un mecanismo fundamental de la programación orientada a objetos que permite crear nuevas clases basadas en clases existentes. La clase que hereda se denomina **subclase** o *clase derivada*, y la clase de la que se hereda se llama **superclase** o *clase base*. La subclase hereda los atributos y métodos de la superclase, y puede añadir nuevos miembros o redefinir (sobrescribir) los métodos heredados para modificar su comportamiento.

En Java, la herencia se implementa utilizando la palabra clave `extends`. Una subclase puede acceder a los miembros `public` y `protected` de la superclase, pero no a los miembros `private`. Además, mediante la palabra clave `super`, la subclase puede invocar el constructor o métodos de la superclase.

Ventajas de la herencia:

- Permite reutilizar código, evitando duplicidad.
- Facilita la extensión y el mantenimiento del software.
- Permite la creación de jerarquías de clases y la especialización de comportamientos.

Ejemplo básico de herencia:

```java
public class Vehiculo {
  protected String marca;

  public Vehiculo(String marca) {
    this.marca = marca;
  }

  public void mostrarMarca() {
    System.out.println("Marca: " + marca);
  }
}

public class Moto extends Vehiculo {
  private String tipo;

  public Moto(String marca, String tipo) {
    super(marca); // Llama al constructor de la superclase
    this.tipo = tipo;
  }

  public void mostrarTipo() {
    System.out.println("Tipo de moto: " + tipo);
  }

  // Sobrescritura de método
  @Override
  public void mostrarMarca() {
    System.out.println("Marca de la moto: " + marca);
  }
}
```

Ejemplo 8.13: Ejemplo de herencia en Java

En este ejemplo, la clase `Moto` hereda de `Vehiculo`. Puede acceder al atributo protegido `marca` y al método `mostrarMarca()`, además de definir su propio atributo `tipo` y el método `mostrarTipo()`. También sobrescribe el método `mostrarMarca()` para personalizar su comportamiento.

Uso de clases heredadas:

```
Moto miMoto = new Moto("Yamaha", "Deportiva");
miMoto.mostrarMarca(); // Llama al método sobrescrito en Moto
miMoto.mostrarTipo(); // Llama al método propio de Moto
```

Ejemplo 8.14: Uso de herencia

Importante

En Java, la herencia es simple, es decir, una clase solo puede heredar de una única superclase directa. Sin embargo, una clase puede implementar múltiples interfaces para lograr un comportamiento similar a la herencia múltiple. Veremos más sobre herencia en el capítulo 12.

Ejercicio 8.5

Crea una clase llamada `Empleado` con los atributos privados `nombre` (`String`) y `salario` (lstinlinedouble), y los métodos `getNombre()` , `getSalario()` y `mostrarInformacion()` que muestre los datos del empleado. Luego, crea una subclase llamada `Gerente` que herede de `Empleado` y añada el atributo privado `departamento` (`String`) y el método `getDepartamento()`. Sobrescribe el método `mostrarInformacion()` para que también muestre el departamento. Crea un objeto de tipo `Gerente` y muestra su información.

8.7.1. La clase `Object`

En Java, todas las clases heredan, directa o indirectamente, de la clase `Object` , que se encuentra en el paquete `java.lang`. Esto significa que cualquier clase que declares, aunque no lo especifiques explícitamente, es una subclase de `Object`. La clase `Object` proporciona métodos fundamentales que están disponibles para todos los objetos en Java.

Método	Descripción
`toString()`	Devuelve una representación en forma de cadena del objeto. Es común sobrescribir este método para mostrar información relevante del objeto.
`equals(Object obj)`	Compara si dos objetos son iguales. Por defecto, compara referencias, pero se puede sobrescribir para comparar el contenido.
`hashCode()`	Devuelve un valor hash para el objeto, útil en estructuras como `HashMap` o `HashSet`.

Método	Descripción
`clone()`	Permite crear una copia del objeto (requiere implementar la interfaz `Cloneable`).
`getClass()`	Devuelve el objeto `Class` que representa la clase del objeto.

Cuadro 8.1: Principales métodos de la clase `Object`

Ejemplo de sobrescritura de `toString()` y `equals()`:

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

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

  @Override
  public String toString() {
    return "Persona[nombre=" + nombre + ", edad=" + edad + "]";
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass()) return false;
    Persona otra = (Persona) obj;
    return edad == otra.edad && nombre.equals(otra.nombre);
  }
}
```

Ejemplo 8.15: Sobrescritura de métodos de Object

Uso:

```java
Persona p1 = new Persona("Ana", 30);
Persona p2 = new Persona("Ana", 30);
System.out.println(p1); // Llama a toString()
System.out.println(p1.equals(p2)); // true, porque tienen los mismos datos
```

Ejercicio 8.6

Crea una clase llamada `Producto` con los atributos privados `codigo` (String) y `precio` (double). Implementa los métodos `getCodigo()`

y `getPrecio()`. Sobrescribe los métodos `toString()` y `equals(Object obj)` para que:

- `toString()` devuelva una cadena con el formato `Producto[codigo=..., precio=...]`.
- `equals(Object obj)` devuelva `true` si el código del producto es igual.

Crea dos objetos `Producto` con el mismo código y verifica si son iguales usando `equals()`.

Resumen

En este capítulo se han presentado los conceptos fundamentales de la programación orientada a objetos relacionados con las clases. Se explicó qué es una clase y cómo define atributos y métodos para crear objetos. Se abordaron los distintos niveles de visibilidad (public, private, protected) y la importancia del encapsulamiento mediante getters y setters. Se mostró cómo crear métodos de instancia y estáticos, así como el uso y sobrecarga de constructores para inicializar objetos. Además, se explicó cómo utilizar clases y objetos en Java, y se introdujo el concepto de herencia para reutilizar y extender funcionalidades. Estos conocimientos son la base para diseñar programas robustos y estructurados en Java.

9

Depuración y pruebas unitarias

La depuración y las pruebas unitarias son etapas fundamentales en el desarrollo de software, ya que permiten identificar y corregir errores, así como garantizar que el código funcione según lo esperado. En este capítulo se abordarán conceptos y herramientas clave para mejorar la calidad y confiabilidad de las aplicaciones.

1. **Aserciones:** Son instrucciones que permiten verificar que ciertas condiciones se cumplan durante la ejecución del programa. Ayudan a detectar errores lógicos y facilitan el mantenimiento del código.

2. **Prueba, depuración y documentación de la aplicación:** Se explorarán técnicas para probar el funcionamiento del software, métodos para depurar errores y buenas prácticas para documentar el proceso, asegurando así un desarrollo más robusto y eficiente.

9.1. Aserciones en Java

Las aserciones en Java permiten comprobar supuestos en el código durante la ejecución. Si una aserción falla, el programa lanza un error `AssertionError`.

```java
public class EjemploAsercion {
  public static void main(String[] args) {
    int edad = -5;
    assert edad >= 0 : "La edad no puede ser negativa";
    System.out.println("Edad: " + edad);
  }
}
```

Ejemplo 9.1: Ejemplo de aserción en Java

En este ejemplo, si `edad` es negativo, la aserción fallará y mostrará el mensaje correspondiente.

Importante

Las aserciones se incluyen en Java desde la versión 1.4, lo cual significa que código anterior a esa versión podía usar la palabra clave `assert` sin problemas, pero no se ejecutaría como aserciones a menos que se habiliten explícitamente.
Para habilitar las aserciones, se debe ejecutar el programa con la opción `-ea` (enable assertions) en la línea de comandos: `java -ea EjemploAsercion`
Si se usa JShell, se puede habilitar con: `jshell -R -ea`
Si se ejecuta sin esta opción, las aserciones no se evaluarán y el programa funcionará normalmente.

Aparte de la palabra clave `assert`, es posible usar librerías que ofrezcan funcionalidades similares, como `AssertJ`, que permiten realizar aserciones más complejas y legibles. Librerías de tests como JUnit también utilizan aserciones para validar el comportamiento del código durante las pruebas unitarias, e incluyen sus propios métodos de aserción.

Ejercicio 9.1

Crea un programa que calcule el factorial de un número entero positivo. Utiliza aserciones para verificar que el número es no negativo antes de calcular el factorial.

9.2. Pruebas unitarias con JUnit

JUnit es un marco de pruebas muy utilizado en Java para escribir y ejecutar pruebas unitarias. Permite verificar que los métodos de una clase funcionen correctamente.

```java
public class Calculadora {
  public int sumar(int a, int b) {
    return a + b;
  }
}
```

Ejemplo 9.2: Clase Calculadora

```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculadoraTest {
  @Test
  public void testSuma() {
```

```
7       Calculadora calc = new Calculadora();
8       int resultado = calc.sumar(2, 3);
9       assertEquals(5, resultado);
10    }
11  }
```

Ejemplo 9.3: Ejemplo básico de prueba unitaria con JUnit

La clase `CalculadoraTest` contiene un método de prueba `testSuma` que verifica que el método `sumar` de la clase `Calculadora` devuelve el resultado esperado. El método `assertEquals` se utiliza para comparar el valor esperado con el valor real. Este método está anotado con `@Test`, lo que indica a JUnit que es un método de prueba. Una anotación permite añadir metadatos a las clases y métodos, y en este caso, indica que el método debe ser ejecutado como una prueba al ejecutar los tests de JUnit.

Consejo

Para ejecutar las pruebas, se recomienda usar un entorno de desarrollo (IDE), tal como *Eclipse* o *IntelliJ IDEA*, o usar la línea de comandos con *Maven* o *Gradle*. También es posible automatizar la ejecución de pruebas unitarias con herramientas de integración continua como Jenkins o GitHub Actions.

Ejercicio 9.2

Crea una clase `Calculadora` con métodos para sumar, restar, multiplicar y dividir. Escribe pruebas unitarias para cada uno de estos métodos utilizando JUnit.

Método	Descripción
`assertEquals(expected, actual)`	Verifica que dos valores sean iguales.
`assertNotEquals(expected, actual)`	Verifica que dos valores sean diferentes.
`assertTrue(condition)`	Verifica que una condición sea verdadera.
`assertFalse(condition)`	Verifica que una condición sea falsa.
`assertNull(object)`	Verifica que un objeto sea `null`.
`assertNotNull(object)`	Verifica que un objeto no sea `null`.
`assertSame(expected, actual)`	Verifica que dos referencias apunten al mismo objeto.
`assertNotSame(expected, actual)`	Verifica que dos referencias no apunten al mismo objeto.

Método	Descripción
`fail()`	Falla la prueba de manera explícita.

Cuadro 9.1: Principales métodos de aserción en JUnit

9.3. Depuración en Java

La depuración permite ejecutar el programa paso a paso, inspeccionar variables y encontrar errores. Los entornos de desarrollo integrados (IDE) como *Eclipse* o *IntelliJ IDEA* ofrecen herramientas visuales para depurar.

Es común que cada entorno de desarrollo ofrezca características específicas para la depuración, como:

- **Puntos de interrupción (breakpoints):** Permiten detener la ejecución en una línea específica.
- **Inspección de variables:** Se pueden examinar los valores de las variables en tiempo real.
- **Ejecución paso a paso:** Permite avanzar instrucción por instrucción para analizar el flujo del programa.

Ejercicio 9.3

Utiliza el depurador de tu IDE para analizar el siguiente código y comprender cómo funciona la ejecución paso a paso. Identifica el valor de las variables en cada paso:

```
1  public class PrimeraLetraPalabra {
2    public static void main(String[] args) {
3      String texto = "Depuración y pruebas unitarias";
4      int numeroPalabras = 0;
5      for (int i = 0; i < texto.length(); i++) {
6        if (i == 0 || texto.charAt(i - 1) == ' ') {
7          System.out.println("Letra inicial " + texto.charAt(↪
     i) + " en la posición " + i);
8          numeroPalabras++;
9        }
10     }
11     System.out.println("Número total de palabras: " + ↪
     numeroPalabras);
12   }
13 }
```

Ejemplo 9.4: Bucle de ejemplo para depuración

9.4. Buenas prácticas de documentación

Documentar el proceso de depuración y pruebas es fundamental para el mantenimiento del software. En Java, se recomienda usar comentarios y la herramienta *Javadoc* para generar documentación automática.

```java
/**
 * Clase que realiza operaciones matemáticas básicas.
 */
public class Calculadora {
  /**
   * Suma dos números enteros.
   * @param a Primer sumando
   * @param b Segundo sumando
   * @return La suma de a y b
   */
  public int sumar(int a, int b) {
    return a + b;
  }
}
```

Ejemplo 9.5: Ejemplo de documentación con Javadoc

Importante

Es recomendable seguir un estilo de codificación consistente y documentar adecuadamente las clases, métodos y variables. Esto facilita la comprensión del código y su mantenimiento a largo plazo. Además, se pueden utilizar herramientas como *Checkstyle* o *SonarQube* para analizar la calidad del código y asegurar que se sigan las mejores prácticas.

Ejercicio 9.4

Añade comentarios y documentación *Javadoc* a la clase `Calculadora` que creaste en el ejercicio 9.2. Asegúrate de documentar cada método y sus parámetros.

Resumen

En este capítulo se abordaron conceptos esenciales para mejorar la calidad del software: el uso de aserciones para verificar supuestos en el código, la implementación de pruebas unitarias con JUnit para asegurar el correcto funcionamiento de los métodos, las herramientas de depuración disponibles en los entornos de desarrollo y la importancia de documentar adecuadamente el proceso. Estas

prácticas permiten detectar y corregir errores de manera eficiente, facilitando el mantenimiento y la confiabilidad de las aplicaciones.

10

Aplicación de las estructuras de almacenamiento

En este capítulo se abordarán los conceptos fundamentales relacionados con las estructuras de almacenamiento en programación. Estas estructuras permiten organizar y manipular datos de manera eficiente, facilitando la resolución de problemas complejos y el desarrollo de aplicaciones robustas. A continuación, se presentan los principales temas que se tratarán:

1. **Estructuras estáticas y dinámicas:** Se explicará la diferencia entre estructuras de almacenamiento cuyo tamaño es fijo en tiempo de compilación (estáticas) y aquellas que pueden crecer o reducirse durante la ejecución del programa (dinámicas).

2. **Creación de arreglos (arrays):** Se mostrará cómo declarar, inicializar y utilizar arreglos para almacenar colecciones de datos homogéneos.

3. **Matrices multidimensionales (arrays multidimensionales):** Se abordará el uso de arreglos con más de una dimensión, útiles para representar tablas, matrices matemáticas y otras estructuras complejas.

4. **Genericidad:** Se introducirá el concepto de estructuras genéricas, que permiten definir colecciones de datos independientes del tipo de los elementos que almacenan.

5. **Cadenas de caracteres. Expresiones regulares:** Se explicará cómo manipular texto mediante cadenas de caracteres y cómo utilizar expresiones regulares para buscar y procesar patrones en cadenas.

6. **Colecciones: Listas, Conjuntos y Diccionarios:** Se presentarán las principales colecciones dinámicas, sus características y casos de uso: listas (secuencias ordenadas), conjuntos (colecciones sin elementos repetidos) y diccionarios (pares clave-valor).

7. **Operaciones agregadas: filtrado, reducción y recolección:** Se describirán operaciones comunes sobre colecciones, como filtrar elementos, reducirlos a un solo valor o recolectar resultados en nuevas estructuras.

Cada uno de estos puntos será explicado en detalle, mostrando ejemplos prácticos y aplicaciones.

10.1. Estructuras estáticas y dinámicas

En Java, las estructuras de almacenamiento pueden ser estáticas (como los *arrays*) o dinámicas (como las colecciones de la biblioteca estándar). Las estructuras estáticas tienen un tamaño fijo, mientras que las dinámicas pueden crecer o reducirse durante la ejecución.

10.1.1. Estructuras estáticas

Las estructuras estáticas, como los arreglos, reservan una cantidad fija de memoria al momento de su creación. Esto significa que no es posible cambiar su tamaño durante la ejecución del programa. Son útiles cuando se conoce de antemano la cantidad de elementos que se van a almacenar y se requiere un acceso rápido a cada elemento mediante un índice. Sin embargo, su principal limitación es la falta de flexibilidad para adaptarse a cambios en la cantidad de datos.

Más adelante, veremos cómo se crean y utilizan los arreglos en Java, ya sean de una dimensión o multidimensionales.

10.1.2. Estructuras dinámicas

Las estructuras dinámicas, a diferencia de las estáticas, permiten modificar su tamaño durante la ejecución del programa. Esto se logra mediante el uso de clases como `ArrayList`, `LinkedList`, `HashSet` y `HashMap`, que forman parte de la biblioteca estándar de Java. Estas estructuras son ideales cuando no se conoce de antemano la cantidad de elementos a almacenar o cuando se requiere agregar y eliminar elementos de manera flexible. Aunque suelen consumir más memoria y pueden ser ligeramente más lentas que los arreglos, ofrecen una gran versatilidad y facilitan la gestión de datos en aplicaciones dinámicas.

```
1 import java.util.ArrayList;
2
3 ArrayList<String> nombres = new ArrayList<>(); // La lista está ↪
    vacía inicialmente
4 nombres.add("Ana"); // Agrega un elemento
```

```
nombres.add("Juan"); // Agrega otro elemento
nombres.add("Luis"); // Agrega otro elemento
System.out.println(nombres.get(1)); // Imprime "Juan"
nombres.remove(0); // Elimina el primer elemento ("Ana")
System.out.println(nombres.size()); // Imprime 2, ya que quedan dos elementos
```

Veremos más adelante en este capítulo cómo funcionan estas estructuras y cómo se pueden utilizar para resolver problemas comunes en programación.

10.2. Creación de arreglos (*arrays*)

Los arreglos son estructuras de datos que permiten almacenar múltiples elementos del mismo tipo en una sola variable. En Java, los arreglos son objetos y se pueden crear de forma estática o dinámica.

```
int[] numeros = new int[5]; // Array de 5 enteros
numeros[0] = 10;
numeros[1] = 20;
System.out.println(numeros[0]); // Imprime 10
```

En Java, los *arrays* se crean usando la palabra clave `new` seguida del tipo de datos y el tamaño del arreglo entre corchetes. Los elementos se acceden mediante índices, comenzando desde 0. No es posible cambiar el tamaño de un *array* una vez creado, lo que puede ser una limitación en algunos casos. Igualmente, si se intenta acceder a un índice fuera de los límites del arreglo, se lanzará una excepción `ArrayIndexOutOfBoundsException`. Por defecto, el *array* se inicializa con valores predeterminados: `0` para enteros, `false` para booleanos y `null` para objetos.

Importante

En Java, y la mayoría de lenguajes de programación, los arreglos comienzan en el número 0, debido a que representan posiciones relativas en la memoria. Por ejemplo, el primer elemento de un arreglo se encuentra en la posición 0, el segundo en la posición 1, y así sucesivamente. Esto significa que si un arreglo tiene 5 elementos, sus índices van del 0 al 4. Si intentas acceder al índice 5, obtendrás un error de índice fuera de rango.
En el caso del lenguaje de programación C, el acceso a los elementos de un arreglo se realiza mediante punteros, lo que permite una mayor flexibilidad, pero también puede llevar a errores si no se maneja correctamente. El primer elemento de un arreglo se encuentra en la dirección de memoria base del arreglo, y los siguientes elementos se encuentran en direcciones consecutivas. Por ejemplo, si tienes un arreglo de enteros de 5 elementos,

el primer elemento estará en la dirección base, el segundo en la dirección base más el tamaño de un entero, y así sucesivamente.

Es posible crear arreglos ya inicializados con valores específicos, lo cual simplifica la declaración y asignación de valores en una sola línea:

```
String[] diasSemana = {"Lunes", "Martes", "Miércoles", "Jueves",↪
    "Viernes", "Sábado", "Domingo"}; // Array de String con los ↪
   días de la semana
System.out.println(diasSemana[0]); // Imprime "Lunes"
System.out.println(diasSemana.length); // Imprime 7, el tamaño ↪
   del array
```

Ejercicio 10.1

Crea un arreglo de enteros de tamaño 10, inicialízalo con valores del 1 al 10 y muestra por pantalla el contenido del arreglo.
Pista: Puedes usar un bucle `for` *para recorrer el arreglo y mostrar sus elementos.*

Consejo

El método `main`, que es el punto de entrada de un programa Java, también puede recibir un arreglo de cadenas como argumento. Esto permite pasar parámetros al ejecutar el programa desde la línea de comandos.
Por ejemplo, si ejecutas el programa con `java MiPrograma arg1 arg2`, el arreglo `args` contendrá `["arg1", "arg2"]`.

Ejercicio 10.2

Crea un programa que reciba como argumentos una serie de números enteros, y que imprima por pantalla la suma de todos ellos.
Pista: Utiliza `Integer.parseInt(texto)`*, para convertir cada valor a un entero.*

10.3. Matrices multidimensionales (*arrays* multidimensionales)

Como los arreglos son objetos en Java, es posible crear arreglos de arreglos, lo que permite crear estructuras multidimensionales. Los arreglos multidimensionales son útiles para representar datos en forma de tablas, matrices o cualquier

otra estructura que requiera más de una dimensión.

```
int[][] matriz = {
  {1, 2, 3},
  {4, 5, 6}
};
System.out.println(matriz[1][2]); // Imprime 6
```

En el ejemplo podemos ver que el tipo del arreglo es `int[][]`, lo que indica que es un arreglo de arreglos de enteros. Cada subarreglo representa una fila de la matriz. Para acceder a un elemento específico, se utilizan dos índices: el primero para la fila y el segundo para la columna.

También es posible crear arreglos multidimensionales de forma dinámica, especificando el tamaño de cada dimensión:

```
int altura = 5;
int[][] trianguloDePascal = new int[altura][];
for (int i = 0; i < altura; i++) {
  trianguloDePascal[i] = new int[i + 1];
  trianguloDePascal[i][0] = 1;
  for (int j = 1; j < i; j++) {
    trianguloDePascal[i][j] = trianguloDePascal[i - 1][j - 1] + ↪
  trianguloDePascal[i - 1][j];
  }
  trianguloDePascal[i][i] = 1;
}
```

Ejercicio 10.3

Completa el ejemplo anterior, añadiendo un bucle que imprima el contenido de la matriz `trianguloDePascal` por pantalla, mostrando cada fila en una línea separada.
Pista: Utiliza un bucle anidado para recorrer las filas y columnas de la matriz. Pista: Puedes usar ***`System.out.print(valor)`*** *para un valor sin generar un salto de línea.*

Consejo

Para imprimir el contenido de un arreglo multidimensional de forma más legible, puedes utilizar la clase `Arrays` de Java, que proporciona un método `toString()` para arreglos unidimensionales y `deepToString()` para arreglos multidimensionales.

```
import java.util.Arrays;

int[] arreglo = {1, 2, 3, 4, 5};
System.out.println(Arrays.toString(arreglo)); // Imprime ↪
```

```
  [1, 2, 3, 4, 5]

int[][] matriz = {
  {1, 2, 3},
  {4, 5, 6}
};
System.out.println(Arrays.deepToString(matriz)); // Imprime
    [[1, 2, 3], [4, 5, 6]]
```

10.4. Genericidad

Java permite definir clases y métodos genéricos para trabajar con diferentes tipos de datos sin duplicar código. Esto permite que una misma clase o método pueda operar con distintos tipos de datos, aumentando la reutilización del código y la flexibilidad.

```
public class Caja<T> {
  private T contenido;
  public void guardar(T valor) { contenido = valor; }
  public T obtener() { return contenido; }
}

// Uso:
Caja<Integer> cajaEntero = new Caja<Integer>();
// Caja que almacena enteros.
// El tipo T se sustituye por Integer.
// Se puede obviar el tipo al instanciar, ya que Java lo infiere
    automáticamente. P.ej:
// Caja<String> cajaString = new Caja<>();
cajaEntero.guardar(100);
System.out.println(cajaEntero.obtener()); // Imprime 100
```

Esta flexibilidad permite que reutilicemos el mismo código para diferentes tipos de datos, sin necesidad de crear múltiples versiones de la misma clase o método. La *genericidad* es una característica poderosa que mejora la legibilidad y mantenibilidad del código.

Ejercicio 10.4

Crea una clase genérica llamada `Cons` que almacene dos valores de cualquier tipo. Implementa métodos para establecer y obtener los valores, y muestra un ejemplo de uso con diferentes tipos de datos.
Pista: Utiliza dos parámetros genéricos para los dos valores, p.ej. `Cons`

<T, U>.

10.5. Cadenas de caracteres. Expresiones regulares

A lo largo de este libro hemos visto como usar cadenas de texto, o *string*, para representar y manipular texto. Sin embargo, Java ofrece una amplia variedad de métodos y clases para trabajar con cadenas de caracteres de manera más avanzada. En este apartado, exploraremos algunas de las características más útiles relacionadas con las cadenas de caracteres y las expresiones regulares.

10.5.1. Manipulación de cadenas

Las cadenas de caracteres en Java son objetos de la clase `String`, que proporciona una gran cantidad de métodos para manipular texto. Algunos de los métodos más comunes incluyen:

- `length()`: Devuelve la longitud de la cadena.
- `charAt(int index)`: Devuelve el carácter en la posición especificada.
- `substring(int beginIndex, int endIndex)`: Devuelve una subcadena entre los índices especificados.
- `toLowerCase()` y `toUpperCase()`: Convierte la cadena a minúsculas o mayúsculas.
- `trim()`: Elimina los espacios en blanco al principio y al final de la cadena.
- `replace(CharSequence target, CharSequence replacement)`: Reemplaza todas las ocurrencias de una subcadena por otra.
- `split(String regex)`: Divide la cadena en un arreglo de subcadenas utilizando una expresión regular como delimitador.

Como la clase `String` es inmutable, cada vez que se realiza una operación que modifica la cadena, se crea un nuevo objeto `String`. Esto significa que las operaciones de concatenación y modificación pueden ser costosas en términos de rendimiento si se realizan repetidamente. Para operaciones más eficientes, se recomienda utilizar la clase `StringBuilder`.

Ejercicio 10.5

Crea un programa que reciba una matrícula de coche, con el formato `1234-ABC`, y realice las siguientes operaciones:

1. Comprueba si la matrícula es válida (4 dígitos seguidos de un guion y 3 letras).
2. Convierte la matrícula a mayúsculas.
3. Reemplaza el guion por un espacio.
4. Imprime la matrícula resultante.

10.5.2. Expresiones regulares

Las expresiones regulares son patrones que permiten buscar, validar y manipular cadenas de texto de manera flexible y potente. En Java, se utilizan principalmente a través de las clases `Pattern` y `Matcher` del paquete `java.util.regex`.

Por ejemplo, para comprobar si una cadena cumple con un formato específico, como una matrícula de coche, se puede usar una expresión regular:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

String matricula = "1234-ABC";
Pattern patron = Pattern.compile("\\d{4}-[A-Z]{3}");
Matcher matcher = patron.matcher(matricula);

if (matcher.matches()) {
  System.out.println("La matrícula es válida");
} else {
  System.out.println("La matrícula no es válida");
}
```

En este ejemplo, el patrón `"\\d{4}-[A-Z]{3}"` significa: cuatro dígitos, un guion y tres letras mayúsculas. El método `matches()` comprueba si toda la cadena cumple el patrón.

Las expresiones regulares también se pueden usar para buscar y reemplazar partes de una cadena:

```
String texto = "El número es 1234-ABC";
String resultado = texto.replaceAll("\\d{4}-[A-Z]{3}", "MATRÍCULA");
System.out.println(resultado); // Imprime: El número es MATRÍCULA
```

Las expresiones regulares son herramientas muy potentes para validar datos de entrada, extraer información y transformar texto en aplicaciones Java.

Consejo

Un objeto de la clase `Pattern` representa una máquina de estados finitos que reconoce cadenas de texto que coinciden con un patrón específico. Por otro lado, un objeto de la clase `Matcher` es responsable de realizar la búsqueda y coincidencia de patrones en una cadena de texto utilizando el patrón definido por el objeto `Pattern`.
Debido a esto, es recomendable compilar el patrón una sola vez, usando un objeto `static final`. Esto mejora el rendimiento, especialmente si el patrón se va a utilizar repetidamente, ya que evita recompilar la expresión regular cada vez que se necesite.
Por ejemplo:

```java
public class ValidadorMatricula {
  private static final Pattern PATRON_MATRICULA = Pattern.compile("\\d{4}-[A-Z]{3}");

  public static boolean esValida(String matricula) {
    Matcher matcher = PATRON_MATRICULA.matcher(matricula);
    return matcher.matches();
  }
}

// Uso:
// if (ValidadorMatricula.esValida(matricula)) {
//   System.out.println("La matrícula es válida");
// }

```

Ejercicio 10.6

Visita la web `https://regexr.com/` y prueba diferentes expresiones regulares para validar formatos de texto. Crea una expresión regular que valide direcciones de correo electrónico y otra que valide números de teléfono en formato español (`6XXXXXXXX` o `9XXXXXXXX`).

10.5.3. La clase `StringBuilder`

La clase `StringBuilder` permite construir y modificar cadenas de texto de manera eficiente, especialmente cuando se realizan muchas operaciones de con-

catenación. A diferencia de la clase `String`, que es inmutable, `StringBuilder` modifica la cadena original sin crear nuevos objetos en cada operación.

```
StringBuilder sb = new StringBuilder();
sb.append("Hola");
sb.append(" ");
sb.append("mundo");
System.out.println(sb.toString()); // Imprime "Hola mundo"
```

`StringBuilder` proporciona métodos como `append()`, `insert()`, `delete()`, `reverse()` y otros, que permiten manipular el contenido de la cadena de forma eficiente. Al finalizar, se puede obtener el resultado como un objeto `String` usando el método `toString()`.

Consejo

Si necesitas modificar cadenas de texto dentro de un bucle, utiliza `StringBuilder` para mejorar el rendimiento y evitar la creación innecesaria de objetos.

Ejercicio 10.7

Crea un programa que construya una frase a partir de palabras pasadas por argumento al método `main`. Utiliza `StringBuilder` para concatenar las palabras y muestra la frase resultante.
Si la entrada es `"Hola", "mundo", "Java"`, el programa debería imprimir `"Hola mundo Java."`. Ten en cuenta que debes añadir un espacio entre cada palabra y un punto al final de la frase.

10.6. Colecciones: Listas, Conjuntos y Diccionarios

Las colecciones son estructuras de datos que permiten almacenar y manipular grupos de objetos de manera eficiente. En Java, las colecciones se organizan en diferentes tipos, cada uno con características específicas. Los tipos más comunes son:

- **Listas:** Son colecciones ordenadas que permiten almacenar elementos duplicados. Se pueden acceder a los elementos por su índice. Ejemplos: `ArrayList`, `LinkedList`.
- **Conjuntos:** Son colecciones no ordenadas que no permiten elementos duplicados. Se utilizan para almacenar elementos únicos. Ejemplos: `HashSet`, `TreeSet`.

- **Diccionarios (Mapas):** Son colecciones de pares clave-valor, donde cada clave es única y se utiliza para acceder a su valor asociado. Ejemplos: `HashMap`, `TreeMap`.

- **Colas:** Son colecciones que siguen el principio FIFO (First In, First Out), donde los elementos se añaden al final y se eliminan del principio. Ejemplos: `LinkedList` (como cola) y `PriorityQueue`.

Consejo

Las colecciones en Java son parte del paquete `java.util`, por lo que es necesario importarlas antes de usarlas. Por ejemplo, para usar `ArrayList`, debes importar `java.util.ArrayList`.
Además, la mayoría de las colecciones, excepto `Map<K,V>`, implementan la interfaz `Collection<E>`, que proporciona métodos comunes para manipular grupos de elementos. Podrás ver una referencia a estas interfaces, al final del capítulo.

10.6.1. Listas (`List`)

Las listas permiten almacenar elementos en un orden específico y acceder a ellos mediante un índice. El tipo más común es `ArrayList<T>`.

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

List<String> frutas = new ArrayList<>();
frutas.add("Manzana");
frutas.add("Banana");
frutas.add("Naranja");
System.out.println(frutas.size()); // Imprime 3, el tamaño de la lista
System.out.println(frutas.get(1)); // Imprime "Banana"
frutas.remove("Manzana");
System.out.println(frutas); // Imprime [Banana, Naranja]
```

Esta colección permite añadir, eliminar y acceder a elementos de manera eficiente. Además, las listas pueden contener elementos duplicados y permiten el acceso aleatorio a los elementos mediante su índice.

Ejercicio 10.8

Crea una clase llamada `ListaDeCompras` que implemente una lista de productos. Debe permitir añadir productos, eliminar productos por nombre y mostrar todos los productos en la lista.

10.6.2. Conjuntos (`Set<T>`)

Los conjuntos almacenan elementos únicos, sin un orden específico. El tipo más común es `HashSet`.

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

Set<Integer> numeros = new HashSet<>();
numeros.add(1);
numeros.add(2);
numeros.add(2); // No se añade porque ya existe
System.out.println(numeros); // Imprime [1, 2]
System.out.println(numeros.contains(1)); // Imprime true
```

Este tipo de colección es útil cuando se necesita garantizar que no haya elementos duplicados y no se requiere un orden específico. Los conjuntos ofrecen operaciones eficientes para añadir, eliminar y comprobar la existencia de elementos.

Ejercicio 10.9

Crea un programa que imprima todos los números primos del 1 al 100 utilizando un conjunto. Crea el método `isPrime(int number)` para comprobar si un número es primo.
Pista: Cada vez que encuentres un número primo, añádelo al conjunto. Al final, imprime el conjunto.
Un número primo es aquel que es positivo y solo es divisible por 1 y por sí mismo. El número 1 no es primo.

10.6.3. Diccionarios (`Map`)

Los diccionarios, o mapas, almacenan pares clave-valor. El tipo más común es `HashMap`.

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

Map<String, Integer> edades = new HashMap<>();
edades.put("Ana", 25);
edades.put("Juan", 30);
System.out.println(edades.get("Ana")); // Imprime 25
edades.remove("Juan");
System.out.println(edades); // Imprime {Ana=25}
```

Los diccionarios permiten acceder a los valores mediante sus claves, lo que facilita la búsqueda y manipulación de datos. Son ideales para almacenar datos

relacionados, como configuraciones o registros, donde cada clave es única. Son eficientes para operaciones de inserción, eliminación y búsqueda.

Ejercicio 10.10

Crea una clase llamada `Agenda` que implemente un diccionario para almacenar nombres y números de teléfono. Debe permitir añadir contactos, eliminar contactos por nombre y buscar un número de teléfono por nombre.

10.6.4. Colas (`Queue`)

Las colas permiten procesar elementos en orden *FIFO* (primero en entrar, primero en salir).

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

Queue<String> tareas = new LinkedList<>();
tareas.add("Tarea 1");
tareas.add("Tarea 2");
System.out.println(tareas.poll()); // Imprime "Tarea 1" y la ↪
  elimina
System.out.println(tareas); // Imprime [Tarea 2]
```

Su uso es ideal para situaciones en las que se necesita guardar elementos en un orden específico y procesarlos en ese mismo orden, como en sistemas de gestión de tareas o procesamiento de eventos.

También existen colas de prioridad (`PriorityQueue`), que permiten ordenar los elementos según su prioridad al ser procesados. Los elementos quedan ordenados según su orden natural o un comparador proporcionado al crear la cola.

```
import java.util.PriorityQueue;

PriorityQueue<Integer> colaPrioridad = new PriorityQueue<>();
colaPrioridad.add(2);
colaPrioridad.add(1);
colaPrioridad.add(3);
System.out.println(colaPrioridad); // Imprime [1, 2, 3]
// El elemento con mayor prioridad (menor valor) es el primero
System.out.println(colaPrioridad.poll()); // Imprime 1 y la ↪
  elimina
System.out.println(colaPrioridad); // Imprime [2, 3]
```

También existen las pilas, las cuales implementan el principio *LIFO* (último dentro, primero fuera), donde el último elemento añadido es el primero en ser eliminado. En Java, se pueden implementar usando `Stack<T>` o `Deque<T>`.

```java
import java.util.Stack;
Stack<String> pila = new Stack<>();
pila.push("Elemento 1");
pila.push("Elemento 2");
System.out.println(pila.pop()); // Imprime "Elemento 2" y lo elimina
System.out.println(pila.peek()); // Imprime "Elemento 1" sin eliminarlo
System.out.println(pila); // Imprime [Elemento 1]
```

Ejercicio 10.11

Crea un programa que simule una cola de atención al cliente. Debe permitir añadir clientes a la cola, atender al primer cliente y mostrar el estado actual de la cola.

10.6.5. Iteradores

Los iteradores permiten recorrer los elementos de una colección de manera secuencial, sin exponer su estructura interna. En Java, la interfaz `Iterator<T>` proporciona los métodos `hasNext()`, `next()` y `remove()` para iterar y modificar colecciones de forma segura.

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

List<String> nombres = new ArrayList<>();
nombres.add("Ana");
nombres.add("Juan");
nombres.add("Luis");

Iterator<String> it = nombres.iterator();
while (it.hasNext()) {
  String nombre = it.next();
  System.out.println(nombre);
  if (nombre.equals("Juan")) {
    it.remove(); // Elimina "Juan" de la lista
  }
}
System.out.println(nombres); // Imprime [Ana, Luis]
```

El uso de iteradores es especialmente útil cuando se necesita eliminar elementos durante la iteración, ya que hacerlo directamente sobre la colección puede causar errores. Además, muchas colecciones en Java ofrecen el bucle `for-each` para recorrer sus elementos de forma más sencilla:

```java
for (String nombre : nombres) {
```

```
    System.out.println(nombre);
}
```

Sin embargo, en el bucle `for-each` no se puede eliminar elementos directamente. Para modificaciones durante la iteración, es preferible usar un `Iterator`.

Ejercicio 10.12

Crea un programa que recorra una lista de números enteros y elimine todos los números impares utilizando un iterador. Muestra la lista resultante.

10.7. Operaciones agregadas: filtrado, reducción y recolección

Desde Java 8, se introdujo el concepto de *streams*, que permite realizar operaciones funcionales sobre colecciones de manera más declarativa y concisa. Los *streams* permiten procesar secuencias de elementos de forma eficiente, aplicando operaciones como filtrado, mapeo, reducción y recolección. Veremos más en detalle el uso de programación funcional en Java en el capítulo 17.

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

List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);

// Filtrar números pares
numeros.stream()
  .filter(n -> n % 2 == 0) // Filtra los números pares
  .forEach(System.out::println); // Imprime 2 y 4

// Reducir (sumar todos los elementos)
int suma = numeros.stream()
  .reduce(0, (a, b) -> a + b); // Suma todos los números
System.out.println(suma); // Imprime 15

// Recolectar en una nueva lista los números mayores que 2
List<Integer> mayores = numeros.stream()
  .filter(n -> n > 2)
  .toList();
System.out.println(mayores); // Imprime [3, 4, 5]
```

Cada uno de estos métodos recibe una función lambda que define la operación a realizar. Una función lambda es una expresión que define un bloque de código que se puede pasar como argumento a un método. En el ejemplo anterior, `n-> n % 2 == 0` es una función lambda que verifica si un número es par.

Los *streams* son perezosos, lo que significa que las operaciones no se ejecutan hasta que se invoca una operación terminal, como `forEach()`, `collect()` o `reduce()`.

Es común combinar múltiples operaciones en una sola cadena de llamadas, lo que permite construir *pipelines* de procesamiento de datos de manera clara y concisa. Además, los *streams* pueden ser paralelizados fácilmente para aprovechar múltiples núcleos de procesamiento, mejorando el rendimiento en operaciones sobre grandes volúmenes de datos. Sin embargo, es importante tener en cuenta que no todas las colecciones son adecuadas para ser procesadas como *streams*, y algunas operaciones pueden no ser eficientes en ciertos tipos de colecciones. Por ejemplo, las operaciones de ordenación son más eficientes en listas que en conjuntos.

Ejercicio 10.13

Crea un programa que procese una lista de nombres y realice las siguientes operaciones:

1. Filtra los nombres que comienzan con la letra 'A'.
2. Convierte todos los nombres a mayúsculas.
3. Recolecta los nombres resultantes en una nueva lista y los imprime.

Resumen

En este capítulo hemos explorado las principales estructuras de almacenamiento en programación, desde los arreglos estáticos y dinámicos hasta las colecciones más avanzadas como listas, conjuntos, diccionarios, colas y pilas. Aprendiste a crear y manipular arreglos unidimensionales y multidimensionales, a utilizar la genericidad para escribir código reutilizable, y a trabajar con cadenas de caracteres y expresiones regulares para procesar texto de manera eficiente.

También conociste las colecciones de la biblioteca estándar de Java, sus características y casos de uso, así como el uso de iteradores para recorrer y modificar colecciones de forma segura. Finalmente, se introdujeron las operaciones agregadas y los *streams*, que permiten procesar colecciones de manera funcional y declarativa.

El dominio de estas estructuras y técnicas es fundamental para desarrollar programas robustos, eficientes y fáciles de mantener.

Cuadro 10.1: Principales métodos de la interfaz `Collection<E>`

Método	Descripción
`boolean add(E e)`	Añade el elemento especificado a la colección. Devuelve `true` si la colección cambió como resultado.
`boolean addAll(Collection<? extends E> c)`	Añade todos los elementos de la colección especificada a esta colección.
`void clear()`	Elimina todos los elementos de la colección.
`boolean contains(Object o)`	Devuelve `true` si la colección contiene el elemento especificado.
`boolean containsAll(Collection<?> c)`	Devuelve `true` si la colección contiene todos los elementos de la colección especificada.
`boolean isEmpty()`	Devuelve `true` si la colección no contiene elementos.
`Iterator<E> iterator()`	Devuelve un iterador sobre los elementos de la colección.
`boolean remove(Object o)`	Elimina una sola instancia del elemento especificado, si está presente.
`boolean removeAll(Collection<?> c)`	Elimina de la colección todos los elementos que están también en la colección especificada.
`boolean retainAll(Collection<?> c)`	Conserva solo los elementos que están también en la colección especificada.
`int size()`	Devuelve el número de elementos en la colección.
`Object[] toArray()`	Devuelve un arreglo que contiene todos los elementos de la colección.
`<T> T[] toArray(T[] a)`	Devuelve un arreglo que contiene todos los elementos de la colección; el tipo del arreglo es el especificado.

Cuadro 10.2: Principales métodos de la interfaz `Map<K, V>`

Método	Descripción
`V put(K key, V value)`	Asocia el valor especificado con la clave especificada. Si la clave ya existía, reemplaza el valor anterior y devuelve el valor antiguo.
`V get(Object key)`	Devuelve el valor asociado a la clave especificada, o `null` si no existe.
`boolean containsKey(Object key)`	Devuelve `true` si el mapa contiene la clave especificada.
`boolean containsValue(Object value)`	Devuelve `true` si el mapa contiene al menos una clave asociada al valor especificado.
`V remove(Object key)`	Elimina la clave (y su valor asociado) del mapa. Devuelve el valor eliminado o `null` si no existía.
`void clear()`	Elimina todas las asociaciones del mapa.
`int size()`	Devuelve el número de pares clave-valor en el mapa.
`boolean isEmpty()`	Devuelve `true` si el mapa no contiene asociaciones.
`Set<K> keySet()`	Devuelve un conjunto con todas las claves del mapa.
`Collection<V> values()`	Devuelve una colección con todos los valores del mapa.
`Set<Map.Entry<K, V>> entrySet()`	Devuelve un conjunto con todas las asociaciones clave-valor del mapa.
`V putIfAbsent(K key, V value)`	Si la clave no está asociada a ningún valor, la asocia al valor especificado y lo devuelve; si ya existe, no modifica el valor y devuelve el valor actual.
`void putAll(Map<? extends K, ? extends V> m)`	Copia todas las asociaciones del mapa especificado a este mapa.

11

Lectura y escritura de información

En el desarrollo de aplicaciones informáticas, la gestión de la entrada y salida de información es fundamental. Este capítulo aborda los conceptos y técnicas esenciales para la lectura y escritura de datos, tanto desde dispositivos externos como desde el propio sistema de archivos. Se estudiarán los diferentes tipos de flujos, el manejo de ficheros y directorios, así como las operaciones básicas de entrada desde teclado y salida a pantalla. El objetivo es proporcionar las bases necesarias para manipular información de manera eficiente y segura en programas de ordenador.

Los temas que se tratarán en este capítulo son:

1. **Salida a pantalla y entrada desde teclado:** Se estudiarán las técnicas para capturar información introducida por el usuario a través del teclado y para mostrar resultados en pantalla, incluyendo el uso de diferentes formatos de visualización.

2. **Flujos de datos:** Se explicarán los conceptos de flujo de entrada y salida, diferenciando entre flujos de bytes y de caracteres. Se presentarán las clases y estructuras más comunes para su manejo en distintos lenguajes de programación.

3. **Ficheros de datos y registros:** Se abordará la organización de la información en ficheros, el concepto de registro y las ventajas de estructurar los datos de esta manera.

4. **Utilización de los sistemas de ficheros:** Se explicará cómo interactuar con el sistema de archivos del sistema operativo, incluyendo la navegación por directorios y la obtención de información sobre los mismos. Además, se presentarán las operaciones básicas de creación, eliminación y modificación de ficheros y directorios.

11.1. Salida a pantalla y entrada desde teclado

La salida a pantalla y la entrada desde teclado son operaciones fundamentales en cualquier lenguaje de programación. En Java, se utilizan principalmente las clases `System.out` para la salida estándar (pantalla) y `Console` para la entrada estándar (teclado).

11.1.1. Salida a pantalla

Para mostrar información por pantalla, se utiliza el método `System.out.println()` para imprimir una línea de texto, o `System.out.print()` para imprimir sin salto de línea.

```
public class SalidaPantalla {
  public static void main(String[] args) {
    System.out.println("Hola, mundo!");
    System.out.print("Introduce tu nombre: ");
  }
}
```

Ejemplo 11.1: Ejemplo de salida a pantalla

Formateo de salida

Para mostrar información de manera más legible o con un formato específico, Java proporciona el método `System.out.printf()`, que permite imprimir texto utilizando especificadores de formato similares a los del lenguaje C. Por ejemplo, se pueden mostrar números con un número fijo de decimales, alinear texto o incluir variables dentro de una cadena.

```
public class FormateoSalida {
  public static void main(String[] args) {
    String nombre = "Ana";
    int edad = 22;
    double nota = 8.75;
    System.out.printf("Nombre: %s, Edad: %d, Nota: %.2f\n", ↪
  nombre, edad, nota);
    System.out.printf("%-10s %5d %7.2f\n", nombre, edad, nota); ↪
  // alineación
  }
}
```

Ejemplo 11.2: Formateo de salida con printf

En este ejemplo, los especificadores `%s`, `%d` y `%.2f` se utilizan para imprimir una cadena, un entero y un número decimal con dos cifras decimales, respectivamente. Además, se puede controlar la anchura y alineación de los campos usando valores como `%-10s` (alineado a la izquierda en 10 espacios).

El método `String.format()` funciona de manera similar, pero devuelve la cadena formateada en lugar de imprimirla directamente.

11.1.2. Entrada desde teclado

Una forma de leer datos desde el teclado en Java es utilizando la clase `Console`, que proporciona métodos para leer texto y contraseñas de manera sencilla y segura. La clase `Console` es especialmente útil cuando se desea ocultar la entrada del usuario, como en el caso de contraseñas.

A continuación se muestra un ejemplo de uso de `Console` para leer una línea de texto y una contraseña:

```
public class EntradaConsola {
  public static void main(String[] args) {
    java.io.Console consola = System.console();
    if (consola != null) {
      String nombre = consola.readLine("Introduce tu nombre: ");
      char[] contrasena = consola.readPassword("Introduce tu ↪
  contraseña: ");
      System.out.println("Hola, " + nombre + ". Tu contraseña ↪
  tiene " + contrasena.length + " caracteres.");
    } else {
      System.out.println("No se puede obtener la consola. ↪
  Ejecuta el programa desde una terminal.");
    }
  }
}
```

Ejemplo 11.3: Lectura desde teclado con Console

`Console` solo está disponible cuando el programa se ejecuta desde una terminal o consola real, no desde algunos entornos de desarrollo integrados (IDE). Si `System.console()` devuelve `null`, significa que la consola no está disponible.

Lectura de texto desde teclado, usando la clase `Scanner`

Para leer texto introducido por el usuario desde el teclado, se puede utilizar la clase `Scanner`:

```
import java.util.Scanner;

public class LeerTeclado {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Introduce una línea de texto: ");
    String texto = sc.nextLine();
    System.out.println("Has escrito: " + texto);
    sc.close();
```

```
10    }
11 }
```

Ejemplo 11.4: Lectura de texto desde teclado

Esta clase permite leer fácilmente cadenas, números y otros tipos de datos desde la entrada estándar. Al final de este capítulo, se incluye una tabla con los principales métodos de la clase `Scanner`.

Ejercicio 11.1

Crea un programa que lea por consola una serie de números enteros introducidos por el usuario, hasta que lea una línea vacía. El programa debe calcular y mostrar la suma de todos los números introducidos, así como la media aritmética.

Consejo

También puedes utilizar la clase `Scanner` para leer datos desde un archivo de texto. Para ello, simplemente crea un objeto `Scanner` pasando un objeto `File` como argumento. Por ejemplo:

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.util.Scanner;
4
5  public class LeerArchivoConScanner {
6    public static void main(String[] args) {
7      try (Scanner sc = new Scanner(new File("alumnos.txt")))↪
     {
8        while (sc.hasNextLine()) {
9          String linea = sc.nextLine();
10         System.out.println(linea);
11       }
12     } catch (FileNotFoundException e) {
13       e.printStackTrace();
14     }
15   }
16 }
```

Ejemplo 11.5: Lectura de un archivo de texto con Scanner

En este caso, el objeto `Scanner` leerá el contenido del archivo línea a línea, permitiendo procesar los datos de manera similar a como se haría con `BufferedReader`.

11.2. Flujos de datos

En Java, los flujos de datos (*streams*)[1] permiten la entrada y salida de información. Existen dos tipos principales: flujos de bytes y flujos de caracteres.

- **Flujos de bytes:** Usan clases como `InputStream` y `OutputStream`. Se emplean para datos binarios. Pueden representar cualquier tipo de dato, como imágenes o archivos de audio. Además, permite acceder a flujos de información de red o dispositivos externos.
- **Flujos de caracteres:** Usan clases como `Reader` y `Writer`. Se emplean para texto. Estos flujos simplifican la lectura y escritura de caracteres, gestionando automáticamente la codificación de caracteres (como UTF-8 o ISO-8859-1).

11.2.1. Lectura y escritura de bytes

Para leer y escribir bytes, se utilizan las clases `FileInputStream` y `FileOutputStream`. Estas clases permiten manipular archivos binarios directamente. Es importante usar estas clases junto a clases de buffer como `BufferedInputStream` y `BufferedOutputStream` para mejorar el rendimiento al leer y escribir grandes cantidades de datos.

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopiarArchivoBinario {
  public static void main(String[] args) {
    try (FileInputStream fis = new FileInputStream("imagen.jpg");
  ;
         FileOutputStream fos = new FileOutputStream("copia_imagen
  .jpg")) {
      byte[] buffer = new byte[1024];
      int bytesLeidos;
      while ((bytesLeidos = fis.read(buffer)) != -1) {
        fos.write(buffer, 0, bytesLeidos);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Ejemplo 11.6: Lectura y escritura de archivos binarios

[1] No confundir con la clase `Stream`

En este ejemplo, se copian los datos de un archivo binario a otro utilizando un búfer de bytes. El uso de `FileInputStream` y `FileOutputStream` permite trabajar con cualquier tipo de archivo, no solo texto.

Importante

Al trabajar con flujos de bytes, es fundamental cerrar los flujos después de su uso para liberar recursos del sistema. En Java, esto se puede hacer automáticamente utilizando el bloque `try-with-resources`, que asegura que los recursos se cierren correctamente al finalizar el bloque.

El uso de `BufferedInputStream` y `BufferedOutputStream` permite leer y escribir datos de archivos de manera más eficiente, ya que utilizan un búfer interno para reducir el número de accesos al disco.

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopiarArchivoConBuffer {
  public static void main(String[] args) {
    try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream("imagen.jpg"));
         BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("copia_imagen.jpg"))) {
      byte[] buffer = new byte[1024];
      int bytesLeidos;
      while ((bytesLeidos = bis.read(buffer)) != -1) {
        bos.write(buffer, 0, bytesLeidos);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Ejemplo 11.7: Copia eficiente de archivos binarios con buffer

En este ejemplo, `BufferedInputStream` y `BufferedOutputStream` mejoran el rendimiento al leer y escribir bloques de datos más grandes de una sola vez.

Lectura de archivos de red

Para leer archivos desde una red, Java proporciona clases como `InputStream` y `BufferedInputStream` junto con la clase `Socket` para conectarse a un servidor remoto. Por ejemplo, para descargar datos de un servidor web:

```java
import java.io.BufferedInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;

public class DescargarArchivo {
  public static void main(String[] args) {
    String urlArchivo = "https://ejemplo.com/archivo.pdf";
    try (InputStream in = new BufferedInputStream(new URL(urlArchivo).openStream());
         FileOutputStream fileOut = new FileOutputStream("archivo_descargado.pdf")) {
      byte[] buffer = new byte[1024];
      int bytesLeidos;
      while ((bytesLeidos = in.read(buffer, 0, 1024)) != -1) {
        fileOut.write(buffer, 0, bytesLeidos);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Ejemplo 11.8: Lectura de datos desde una URL

En este ejemplo, se abre un flujo de entrada desde una URL y se escribe el contenido descargado en un archivo local. El uso de `BufferedInputStream` mejora la eficiencia de la lectura de datos desde la red.

Ejercicio 11.2

Crea un programa que descargue una imagen desde una URL y la guarde en el sistema de archivos local. Utiliza las clases `URL`, `InputStream` y `FileOutputStream` para realizar la descarga.
El nombre del fichero a descargar se pasará como argumento al programa, así como el nombre del fichero donde se guardará la imagen. Por ejemplo, si el programa se llama `DescargarImagen`, se ejecutaría así:

```
java DescargarImagen https://ejemplo.com/imagen.jpg imagen_descargada.jpg
```

Una vez descargado el fichero, deberá mostrarse un mensaje indicando que la descarga se ha completado correctamente, y el tamaño del fichero descargado en bytes.

11.2.2. Lectura y escritura de caracteres

Para trabajar con texto, Java proporciona las clases `FileReader` y `FileWriter`, así como sus versiones con buffer: `BufferedReader` y `BufferedWriter`. Estas clases permiten leer y escribir archivos de texto de manera eficiente y sencilla.

Lectura de archivos de texto

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

public class LeerArchivoTexto {
  public static void main(String[] args) {
    try (BufferedReader br = new BufferedReader(new FileReader("↪
  alumnos.txt"))) {
      String linea;
      while ((linea = br.readLine()) != null) {
        System.out.println(linea);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Ejemplo 11.9: Lectura de un archivo de texto línea a línea

En este ejemplo, se lee el archivo `alumnos.txt` línea a línea y se muestra su contenido por pantalla. El uso de `BufferedReader` permite leer texto de manera eficiente.

Escritura de archivos de texto

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class EscribirArchivoTexto {
  public static void main(String[] args) {
    try (BufferedWriter bw = new BufferedWriter(new FileWriter("↪
  salida.txt"))) {
      bw.write("Primera línea de texto\n");
      bw.write("Segunda línea de texto\n");
    } catch (IOException e) {
      e.printStackTrace();
    }
```

```
    }
}
```

Ejemplo 11.10: Escritura de texto en un archivo

En este caso, se escribe texto en el archivo `salida.txt`. Si el archivo no existe, se crea automáticamente. Si existe, se sobrescribe su contenido.

Importante

Al igual que con los flujos de bytes, es fundamental cerrar los flujos de caracteres para liberar recursos. El uso de `try-with-resources` facilita esta tarea.

Ejercicio 11.3

Crea un programa que lea un fichero tabulado (CSV) y muestre su contenido por pantalla. El fichero contendrá varias líneas, cada una con varios campos separados por comas. Utiliza las clases `BufferedReader` y `String.split()` para procesar cada línea. El programa deberá mostrar cada campo en una línea separada, indicando el número de línea y el número de campo. Por ejemplo, si el fichero contiene:

```
nombre,edad,carrera
Juan,20,Ingeniería
Ana,22,Matemáticas
```

El programa deberá mostrar:

```
Línea 1, Campo 1: nombre
Línea 1, Campo 2: edad
Línea 1, Campo 3: carrera
Línea 2, Campo 1: Juan
Línea 2, Campo 2: 20
Línea 2, Campo 3: Ingeniería
Línea 3, Campo 1: Ana
Línea 3, Campo 2: 22
Línea 3, Campo 3: Matemáticas
```

Modos de acceso

En Java, es posible abrir un fichero en diferentes modos, como lectura, escritura o adición. Estos modos se especifican al crear el objeto correspondiente. Por ejemplo, al usar `FileWriter`, se puede indicar si se desea sobrescribir el fichero existente o añadir contenido al final del mismo:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
```

```
import java.io.IOException;

public class AnadirArchivo {
  public static void main(String[] args) {
    try (BufferedWriter bw = new BufferedWriter(new FileWriter("alumnos.txt", true))) {
      bw.write("Luis,21,Física\n");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Ejemplo 11.11: Apertura de un fichero en modo adición

En este caso, el segundo parámetro de `FileWriter` es `true`, lo que indica que se debe añadir el contenido al final del fichero en lugar de sobrescribirlo. Si se omite este parámetro, el fichero se sobrescribirá por defecto. Esto es útil para mantener un registro de datos sin perder la información existente.

11.2.3. Lectura y escritura de ficheros de acceso aleatorio

Los ficheros de acceso aleatorio permiten leer y escribir datos en cualquier posición del archivo, sin necesidad de procesar todo el contenido secuencialmente. En Java, esto se logra mediante la clase `RandomAccessFile`, que proporciona métodos para posicionarse en cualquier parte del fichero y realizar operaciones de lectura o escritura.

Uso de `RandomAccessFile`

La clase `RandomAccessFile` permite abrir un archivo en modo lectura (`"r"`) o lectura/escritura (`"rw"`). Se pueden leer y escribir datos primitivos (enteros, cadenas, etc.) y moverse a cualquier posición del archivo usando el método `seek()`.

```
import java.io.RandomAccessFile;
import java.io.IOException;

public class AccesoAleatorio {
  public static void main(String[] args) {
    try (RandomAccessFile raf = new RandomAccessFile("datos.bin", "rw")) {
      // Escribir datos en posiciones específicas
      raf.writeInt(100); // posición 0
      raf.writeDouble(3.14); // posición 4
      raf.writeUTF("Hola"); // posición 12

```

```
12        // Leer datos desde una posición concreta
13        raf.seek(0); // volver al inicio
14        int numero = raf.readInt();
15        double decimal = raf.readDouble();
16        String texto = raf.readUTF();
17
18        System.out.println("Número: " + numero);
19        System.out.println("Decimal: " + decimal);
20        System.out.println("Texto: " + texto);
21      } catch (IOException e) {
22        e.printStackTrace();
23      }
24    }
25  }
```

Ejemplo 11.12: Ejemplo básico de acceso aleatorio

En este ejemplo, se escriben y leen datos en posiciones específicas del archivo. El método `seek(long pos)` permite moverse a cualquier byte del fichero, lo que resulta útil para modificar registros concretos sin tener que leer todo el archivo.

Importante

El acceso aleatorio es especialmente útil para ficheros de registros de tamaño fijo, como bases de datos simples o índices, donde se necesita modificar o consultar rápidamente un registro concreto.
Otra ventaja es que el fichero no se carga completamente en memoria, lo que permite trabajar con archivos grandes sin consumir demasiada memoria RAM. Sin embargo, es importante tener en cuenta que el uso de `RandomAccessFile` puede ser más complejo que el de flujos secuenciales, ya que se debe gestionar manualmente la posición del cursor y el formato de los datos. Esto es especialmente relevante cuando se trabaja con ficheros que ocupen varios megabytes, o incluso gigabytes (por ejemplo, el mapa de un videojuego masivo en línea).

Consejo

Al trabajar con `RandomAccessFile`, es importante tener en cuenta el tamaño de los datos que se están leyendo o escribiendo. Por ejemplo, al usar `writeInt()` se escriben 4 bytes, y al usar `writeDouble()` se escriben 8 bytes. Asegúrate de posicionarte correctamente con `seek()` antes de realizar operaciones de lectura o escritura para evitar errores de formato o datos corruptos.

Ejercicio 11.4

Crea un programa que almacene una lista de registros de alumnos en un fichero binario usando `RandomAccessFile`. Cada registro debe tener un nombre (cadena de longitud fija, por ejemplo 20 caracteres), una edad (entero) y una nota (double). El programa debe permitir añadir nuevos registros, consultar un registro por su posición y modificar la nota de un alumno dado su número de registro.

El programa deberá tener otro método que lea todos un registro específico del fichero y lo muestre por pantalla. Utiliza `seek()` para posicionarte en el registro correspondiente al realizar las operaciones de consulta y modificación.

Para realizar este programa, deberás definir una estructura de datos que represente un alumno, y utilizar los métodos de `RandomAccessFile` para leer y escribir los datos en el fichero. Asegúrate de manejar correctamente las excepciones y cerrar el fichero al finalizar las operaciones.

Además, deberá crear un menú interactivo que permita al usuario elegir entre las diferentes operaciones (añadir, consultar, modificar) y realizar las acciones correspondientes. El menú deberá repetirse hasta que el usuario decida salir del programa.

11.3. Ficheros de datos y registros

Existen múltiples formas de organizar la información en ficheros. Una de las más comunes es el uso de registros, que permiten almacenar datos estructurados en un formato específico. En esta sección veremos como trabajar con los principales tipos de fichero usados para almacenar datos, y como representar la información de cada uno de ellos.

11.3.1. Ficheros tabulados (CSV)

En el ejercicio 11.3 se ha visto un ejemplo de fichero tabulado, que utiliza comas para separar los campos. Este formato es ampliamente utilizado para almacenar datos en forma de tablas, y es fácil de leer y escribir tanto por humanos como por programas. En él, la primera línea suele contener los nombres de los campos, y las siguientes líneas contienen los datos correspondientes.

La ventaja de este tipo de fichero es su simplicidad y compatibilidad con muchas aplicaciones, como hojas de cálculo y bases de datos. Además, es fácil de manipular con herramientas de programación, como habrás podido comprobar en el ejercicio.

Ejercicio 11.5

Modifica el programa del ejercicio 11.3 para que, en lugar de mostrar los campos por pantalla, los almacene en una lista de objetos. Cada objeto representará una fila del fichero, y tendrá atributos correspondientes a cada campo. Utiliza la clase `ArrayList` para almacenar los objetos. Cada línea deberá representarse como un objeto de la clase `Map<String, String>`, en el cual cada clave será el nombre del campo y el valor será el dato correspondiente. Al final, muestra por pantalla el contenido de la lista de objetos.

Una vez implementado el procesamiento del fichero, crea un método que convierta la lista de objetos de tipo `Map<String, String>`, en una lista de objetos de una clase personalizada, por ejemplo, `Alumno`, que tenga los atributos `nombre`, `edad` y `carrera`. El método deberá recorrer la lista de mapas y crear un objeto `Alumno` por cada línea del fichero, añadiéndolo a una nueva lista. Finalmente, muestra por pantalla el contenido de la lista de objetos `Alumno`.

En este ejercicio, se pretende que practiques la lectura de ficheros tabulados, el uso de colecciones y la creación de objetos personalizados.

Pista: Puedes implementar el método `toString()` de la clase `Alumno` para que muestre los datos del alumno de forma legible. Por ejemplo, si el objeto `Alumno` tiene los atributos `nombre`, `edad` y `carrera`, el método `toString()` podría devolver una cadena como: `"Nombre: Juan, Edad: 20, Carrera: Ingeniería"`. Esto facilitará la visualización de los datos al imprimir los objetos en la consola.

11.3.2. Ficheros de marcas (XML)

Los ficheros XML (*eXtensible Markup Language*) son un formato de texto que permite almacenar datos estructurados de manera jerárquica. Se utilizan ampliamente para intercambiar información entre aplicaciones y sistemas, ya que son legibles tanto por humanos como por máquinas.

Un fichero XML está compuesto por etiquetas que definen la estructura de los datos. Cada etiqueta puede contener atributos y valores, lo que permite representar información compleja de forma organizada.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<alumnos xmlns="http://www.example.com/alumnos">
  <alumno carrera="Ingeniería" edad="20" nombre="Juan"/>
  <alumno carrera="Matemáticas" edad="22" nombre="Ana"/>
  <alumno carrera="Física" edad="21" nombre="Luis"/>
```

```
</alumnos>
```

Ejemplo 11.13: Ejemplo de fichero XML

Para trabajar con ficheros XML en Java, se pueden utilizar bibliotecas como `JAXB` (Java Architecture for XML Binding) o `DOM` (Document Object Model). Estas bibliotecas permiten leer y escribir ficheros XML de manera sencilla, convirtiendo los datos en objetos Java y viceversa.

```
package programacion.ejemplos.xml;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;
import java.io.File;

public class LeerXML {
  public static void main(String[] args) {
    try {
      JAXBContext context = JAXBContext.newInstance(Alumnos.class);
      Unmarshaller unmarshaller = context.createUnmarshaller();
      Alumnos alumnos = (Alumnos) unmarshaller.unmarshal(new File("alumnos.xml"));
      for (Alumno a : alumnos.getAlumno()) {
        System.out.println(a);
      }
    } catch (JAXBException e) {
      e.printStackTrace();
    }
  }
}
```

Ejemplo 11.14: Lectura de un fichero XML con JAXB

La clase `LeerXML` utiliza `JAXB` para leer un fichero XML y convertirlo en una lista de objetos `Alumno`. Esto se logra ya que JAXB permite mapear las etiquetas XML a clases Java, facilitando la manipulación de los datos.

```
package programacion.ejemplos.xml;

import java.util.List;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "alumnos", namespace = "http://www.example.com/alumnos")
class Alumnos {
  private List<Alumno> alumno;

```

```
11    @XmlElement(name = "alumno", namespace = "http://www.example.↪
      com/alumnos")
12    public List<Alumno> getAlumno() {
13      return alumno;
14    }
15
16    public void setAlumno(List<Alumno> alumno) {
17      this.alumno = alumno;
18    }
19  }
```

Ejemplo 11.15: Clase que representa la lista de alumnos

La clase `Alumnos` es una colección de objetos `Alumno`. Utiliza anotaciones de JAXB para indicar cómo se deben serializar y deserializar los datos XML. La anotación `@XmlRootElement` indica que esta clase es la raíz del documento XML, y la anotación `@XmlElement` indica que cada objeto `Alumno` se corresponde con un elemento XML.

```
1  package programacion.ejemplos.xml;
2
3  import javax.xml.bind.annotation.XmlAttribute;
4  import javax.xml.bind.annotation.XmlRootElement;
5
6  @XmlRootElement(name = "alumno", namespace = "http://www.example↪
     .com/alumnos")
7  public class Alumno {
8    private String nombre;
9    private int edad;
10   private String carrera;
11
12   public Alumno() {
13     // Constructor por defecto requerido por JAXB
14   }
15
16   public Alumno(String nombre, int edad, String carrera) {
17     this.nombre = nombre;
18     this.edad = edad;
19     this.carrera = carrera;
20   }
21
22   @XmlAttribute(name = "nombre")
23   public String getNombre() {
24     return nombre;
25   }
26
27   public void setNombre(String nombre) {
28     this.nombre = nombre;
29   }
30
```

```
31    @XmlAttribute(name = "edad")
32    public int getEdad() {
33      return edad;
34    }
35
36    public void setEdad(int edad) {
37      this.edad = edad;
38    }
39
40    @XmlAttribute(name = "carrera")
41    public String getCarrera() {
42      return carrera;
43    }
44
45    public void setCarrera(String carrera) {
46      this.carrera = carrera;
47    }
48
49    @Override
50    public String toString() {
51      return "Nombre: " + nombre + ", Edad: " + edad + ", Carrera:↪
      " + carrera;
52    }
53  }
```

Ejemplo 11.16: Clase que representa a un alumno

En esta clase, se definen los atributos de un alumno, como `nombre`, `edad` y `carrera`. Las anotaciones `@XmlAttribute` indican que estos atributos se corresponden con atributos XML. Además, se incluye un método `toString()` para facilitar la visualización de los datos del alumno.

Para escribir datos en un fichero XML, se puede utilizar el siguiente ejemplo:

```
1  package programacion.ejemplos.xml;
2
3  import javax.xml.bind.JAXBContext;
4  import javax.xml.bind.JAXBException;
5  import javax.xml.bind.Marshaller;
6  import java.io.File;
7  import java.util.List;
8
9  public class EscribirXML {
10   public static void main(String[] args) {
11     try {
12       JAXBContext context = JAXBContext.newInstance(Alumnos.↪
     class);
13       Marshaller marshaller = context.createMarshaller();
14       marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, ↪
     Boolean.TRUE);
15
```

```
16        Alumnos alumnos = new Alumnos();
17        alumnos.setAlumno(List.of(
18          new Alumno("Juan", 20, "Ingeniería"),
19          new Alumno("Ana", 22, "Matemáticas"),
20          new Alumno("Luis", 21, "Física")
21        ));
22
23        marshaller.marshal(alumnos, new File("alumnos.xml"));
24      } catch (JAXBException e) {
25        e.printStackTrace();
26      }
27    }
28  }
```

Ejemplo 11.17: Escritura de un fichero XML con JAXB

Importante

Para poder trabajar con ficheros XML en Java, es necesario incluir la biblioteca `JAXB` en el proyecto. A partir de Java 9, esta biblioteca no está incluida por defecto, por lo que se debe añadir como dependencia en el archivo `pom.xml` si se utiliza Maven, o descargarla manualmente si se trabaja sin un gestor de dependencias.

En Maven, se puede añadir las siguientes dependencias al archivo `pom.xml`:

```
<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
  <version>2.3.1</version>
</dependency>
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
  <version>2.3.1</version>
</dependency>
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-core</artifactId>
  <version>2.3.0.1</version>
</dependency>
<dependency>
  <groupId>javax.activation</groupId>
  <artifactId>activation</artifactId>
  <version>1.1.1</version>
</dependency>
```

Asegúrate de que tu entorno de desarrollo esté configurado correctamente para utilizar estas bibliotecas, ya que son necesarias para trabajar con

ficheros XML en Java.

Ejercicio 11.6

Crea un programa con dos métodos estáticos, en uno creará un fichero XML, y en otro leerá el contenido de ese fichero y lo mostrará por pantalla. El fichero XML tendrá que contener un nodo raiz `calendario`, y dentro de este, varios nodos `evento` que representen eventos del calendario. Cada nodo `evento` tendrá los atributos `fecha`, `hora` y `descripcion`. Por ejemplo:

```
<calendario xmlns="http://www.example.com/calendario">
  <evento fecha="2023-10-01" hora="10:00" descripcion="↪
  Reunión con el equipo"/>
  <evento fecha="2023-10-02" hora="15:00" descripcion="↪
  Presentación del proyecto"/>
  <evento fecha="2023-10-03" hora="09:00" descripcion="↪
  Desayuno con el cliente"/>
</calendario>
```

11.4. Utilización de los sistemas de ficheros

Java proporciona la clase `File` y, desde Java 7, la API `java.nio.file` para interactuar con el sistema de archivos. También se pueden utilizar las clases `Path` y `Files` para realizar operaciones más avanzadas.

11.4.1. Navegación por directorios

Para listar el contenido de un directorio y navegar por la estructura de carpetas, se puede utilizar la clase `File` o la API moderna `java.nio.file`. A continuación se muestran ejemplos con ambas opciones.

Listar archivos y subdirectorios con `File`

```
import java.io.File;

public class ListarDirectorio {
  public static void main(String[] args) {
    File directorio = new File(".");
    File[] archivos = directorio.listFiles();
    if (archivos != null) {
      for (File archivo : archivos) {
```

```
        if (archivo.isDirectory()) {
          System.out.println("[DIR] " + archivo.getName());
        } else {
          System.out.println("[FILE] " + archivo.getName());
        }
      }
    }
  }
}
```

Ejemplo 11.18: Listar archivos y directorios con File

En este ejemplo, se lista el contenido del directorio actual. El método `isDirectory()` permite distinguir entre archivos y carpetas.

Navegación avanzada con `java.nio.file`

Desde Java 7, la API `java.nio.file` proporciona clases como `Path` y `Files` para trabajar con rutas y directorios de forma más flexible.

```
import java.nio.file.*;

public class ListarDirectorioNIO {
  public static void main(String[] args) throws Exception {
    Path ruta = Paths.get(".");
    try (DirectoryStream<Path> stream = Files.newDirectoryStream(ruta)) {
      for (Path entry : stream) {
        if (Files.isDirectory(entry)) {
          System.out.println("[DIR] " + entry.getFileName());
        } else {
          System.out.println("[FILE] " + entry.getFileName());
        }
      }
    }
  }
}
```

Ejemplo 11.19: Listar archivos usando java.nio.file

Esta API permite realizar operaciones más avanzadas, como filtrar archivos por extensión, recorrer subdirectorios recursivamente o acceder a atributos de los archivos.

Ejercicio 11.7

Crea un programa que recorra recursivamente un directorio y muestre por pantalla la estructura completa de carpetas y archivos, indicando el nivel de profundidad mediante sangrados o símbolos.

11.4.2. Creación y eliminación de ficheros y directorios

Para crear y eliminar archivos y directorios en Java, se pueden utilizar tanto la clase `File` como la API moderna `java.nio.file`. A continuación se muestran ejemplos de ambas formas.

Creación y eliminación con `File`

```
import java.io.File;
import java.io.IOException;

public class CrearEliminarFile {
  public static void main(String[] args) {
    File archivo = new File("nuevo_archivo.txt");
    File directorio = new File("nueva_carpeta");

    try {
      if (archivo.createNewFile()) {
        System.out.println("Archivo creado: " + archivo.getName↪
  ());
      }
      if (directorio.mkdir()) {
        System.out.println("Directorio creado: " + directorio.↪
  getName());
      }
    } catch (IOException e) {
      e.printStackTrace();
    }

    // Eliminar archivo y directorio
    if (archivo.delete()) {
      System.out.println("Archivo eliminado.");
    }
    if (directorio.delete()) {
      System.out.println("Directorio eliminado.");
    }
  }
}
```

Ejemplo 11.20: Crear y eliminar archivos y directorios con File

El método `createNewFile()` crea un archivo vacío si no existe, y `mkdir()` crea un nuevo directorio. Para eliminar, se usa `delete()`. Ten en cuenta que un directorio solo se puede eliminar si está vacío.

Creación y eliminación con `java.nio.file`

```java
import java.nio.file.*;

public class CrearEliminarNIO {
  public static void main(String[] args) throws Exception {
    Path archivo = Paths.get("archivo_nio.txt");
    Path directorio = Paths.get("carpeta_nio");

    // Crear archivo y directorio
    Files.createFile(archivo);
    Files.createDirectory(directorio);

    // Eliminar archivo y directorio
    Files.deleteIfExists(archivo);
    Files.deleteIfExists(directorio);
  }
}
```

Ejemplo 11.21: Crear y eliminar archivos y directorios con java.nio.file

La API `Files` proporciona métodos como `createFile()`, `createDirectory()`, `delete()` y `deleteIfExists()` para gestionar archivos y carpetas de forma sencilla y robusta.

Importante

Antes de eliminar un directorio, asegúrate de que esté vacío. Si necesitas eliminar un directorio con contenido, puedes usar el método `Files.walk()` para recorrer y eliminar todos los archivos y subdirectorios.

Ejercicio 11.8

Crea un programa que imprima por pantalla todos los ficheros que se encuentran en un directorio específico, y recursivamente en sus subdirectorios hasta un nivel máximo de profundidad (en este caso, 3). El programa deberá recibir como argumento el directorio a explorar. Utiliza la clase `Files` y sus métodos para listar los ficheros y directorios. El programa deberá mostrar el nombre de cada fichero, su tamaño en bytes y la ruta completa.

Resumen

En este capítulo se han presentado los conceptos y técnicas fundamentales para la gestión de entrada y salida de información en Java. Se ha explicado cómo realizar operaciones básicas de lectura desde teclado y escritura en pantalla, así como el uso de flujos de datos para trabajar con archivos binarios y de texto.

Se han mostrado ejemplos prácticos de lectura y escritura secuencial, acceso aleatorio a ficheros, y manipulación de archivos estructurados como CSV y XML. Además, se ha abordado la interacción con el sistema de archivos, incluyendo la creación, eliminación y navegación por directorios. Finalmente, se han propuesto ejercicios para afianzar los conocimientos adquiridos.

Cuadro 11.1: Principales métodos de la clase `Scanner`

Método	**Descripción**
`String nextLine()`	Lee una línea completa de texto
`String next()`	Lee la siguiente palabra (token)
`int nextInt()`	Lee el siguiente valor entero
`double nextDouble()`	Lee el siguiente valor de tipo `double`
`float nextFloat()`	Lee el siguiente valor de tipo `float`
`boolean nextBoolean()`	Lee el siguiente valor booleano
`boolean hasNext()`	Indica si hay otro token disponible
`boolean hasNextInt()`	Indica si el siguiente token es un entero
`boolean hasNextDouble()`	Indica si el siguiente token es un double
`void close()`	Cierra el objeto `Scanner`

12

Utilización avanzada de clases

En este capítulo profundizaremos en el uso avanzado de las clases en la programación orientada a objetos. Analizaremos conceptos fundamentales que permiten diseñar sistemas más robustos, reutilizables y mantenibles. Abordaremos temas como la composición de clases, la herencia y el polimorfismo, así como la organización de jerarquías mediante superclases y subclases. También exploraremos el uso de clases y métodos abstractos y finales, el papel de las interfaces, la sobreescritura de métodos y el funcionamiento de los constructores en contextos de herencia. Estos conceptos son esenciales para aprovechar al máximo el paradigma orientado a objetos y desarrollar aplicaciones complejas de manera eficiente.

En este capítulo, se abordarán los siguientes temas:

1. **Sobrecarga de métodos:** Veremos cómo definir múltiples métodos con el mismo nombre, pero diferentes parámetros, lo que permite una mayor flexibilidad en la implementación de funcionalidades.

2. **Composición de clases:** Veremos cómo crear clases complejas a partir de otras, facilitando la reutilización de código y la representación de relaciones de tipo "tiene-un" entre objetos.

3. **Herencia y polimorfismo:** Estudiaremos cómo la herencia permite derivar nuevas clases a partir de existentes y cómo el polimorfismo posibilita que distintos objetos respondan de manera específica a los mismos métodos.

4. **Jerarquía de clases: Superclases y subclases:** Analizaremos la organización jerárquica de las clases, donde las subclases heredan características de las superclases.

5. **Constructores y herencia:** Examinaremos el funcionamiento de los constructores en jerarquías de clases y cómo asegurar la correcta inicialización de los objetos.

6. **Clases y métodos abstractos y finales:** Explicaremos el uso de clases y métodos abstractos para definir comportamientos que deben ser implementados por las subclases, y el papel de los elementos finales para restringir la herencia o la sobrescritura.

7. **Interfaces:** Abordaremos cómo las interfaces permiten definir contratos de métodos que las clases deben implementar, promoviendo la reutilización y la flexibilidad sin herencia múltiple.

8. **Enumerados:** Veremos cómo definir tipos enumerados para representar un conjunto fijo de constantes, mejorando la legibilidad y la seguridad del código.

9. **El tipo `record`:** Introduciremos el tipo `record` en Java, que permite definir clases inmutables de manera concisa, facilitando la creación de objetos que representan datos sin necesidad de escribir código boilerplate.

12.1. Sobrecarga de métodos

La sobrecarga de métodos permite definir varios métodos con el mismo nombre pero diferentes listas de parámetros. Esto facilita la implementación de funcionalidades similares que requieren distintos tipos o cantidades de argumentos.

```
1  public class Calculadora {
2    public int sumar(int a, int b) {
3      return a + b;
4    }
5    public double sumar(double a, double b) {
6      return a + b;
7    }
8    public int sumar(int a, int b, int c) {
9      return a + b + c;
10   }
11 }
```

Ejemplo 12.1: Ejemplo de sobrecarga de métodos en Java

Métodos con número variables de argumentos

En Java, es posible definir métodos que aceptan un número variable de argumentos utilizando el operador ... (varargs). Esto permite que un método reciba cero o más argumentos del mismo tipo.

```
1  public class Calculadora {
2    public int sumar(int... numeros) {
```

```
    int suma = 0;
    for (int n : numeros) {
      suma += n;
    }
    return suma;
  }
}

// Uso:
Calculadora calc = new Calculadora();
System.out.println(calc.sumar(1, 2));      // Imprime 3
System.out.println(calc.sumar(1, 2, 3, 4));  // Imprime 10
```

Ejemplo 12.2: Método con número variable de argumentos

Esto resulta útil cuando no se conoce de antemano cuántos argumentos se pasarán al método.

Consejo

Es posible definir un número mínimo de argumentos en un método con varargs, combinando parámetros fijos y variables.
Por ejemplo, el siguiente método requiere al menos un argumento entero y luego acepta cualquier cantidad adicional:

```
public int multiplicar(int primero, int... otros) {
  int producto = primero;
  for (int n : otros) {
    producto *= n;
  }
  return producto;
}

// Uso:
multiplicar(2);      // Devuelve 2
multiplicar(2, 3, 4);  // Devuelve 24 (2*3*4)

```

Ejemplo 12.3: Método con argumentos fijos y variables

Además, es posible usar la lista de argumentos variables como un objeto, y pasarla a otros métodos que acepten un número variable de argumentos.

```
public void imprimirSuma(int... numeros) {
  System.out.println("La suma es: " + sumar(numeros));
}

public int sumar(int... numeros) {
  int suma = 0;
  for (int n : numeros) {
```

```
      suma += n;
    }
    return suma;
  }

  // Uso:
  imprimirSuma(1, 2, 3); // Imprime "La suma es: 6"

```

Ejemplo 12.4: Reutilización de varargs

Ejercicio 12.1

Crea las clases necesarias para representar un las siguientes figuras geométricas: círculo, cuadrado y triángulo. Crea una clase `CalculadoraFiguras` que tenga métodos sobrecargados para calcular el área de cada figura. Utiliza la sobrecarga de métodos para definir un método `calcularArea` que acepte diferentes tipos de figuras como parámetros.

12.2. Composición de clases

La composición consiste en construir clases complejas a partir de otras, estableciendo relaciones de tipo "tiene-un". Es preferible a la herencia cuando la relación no es estrictamente jerárquica.

```
public class Motor {
  public void encender() {
    System.out.println("Motor encendido");
  }
}

public class Auto {
  private Motor motor;
  public Auto() {
    motor = new Motor();
  }
  public void arrancar() {
    motor.encender();
    System.out.println("Auto en marcha");
  }
}
```

Ejemplo 12.5: Ejemplo de composición de clases en Java

El uso de la composición nos permite crear objetos más flexibles y reutilizables, ya que podemos combinar diferentes componentes sin necesidad de heredar de una clase base. Al usar herencia, podremos cambiar el comportamiento de la clase `Auto` sin necesidad de modificar la clase `Motor`, lo que facilita el mantenimiento y la evolución del código.

Ejercicio 12.2

Crea una clase `Libro` que contenga atributos como título, autor y número de páginas. Luego, crea una clase `Biblioteca` que tenga una lista de libros. Implementa métodos para añadir, eliminar y listar los libros en la biblioteca. Utiliza la composición para relacionar las clases.
Crea un método `imprimirLibros` en la clase `Biblioteca` que imprima los títulos de todos los libros almacenados. Utiliza un bucle para recorrer la lista de libros y mostrar sus títulos. El formato de cada línea debe ser: `Título del libro - Autor del libro - Número de páginas`, y deberá ser generado dentro de la clase `Libro`.

12.3. Herencia y polimorfismo

La herencia permite crear nuevas clases basadas en otras existentes. El polimorfismo posibilita que objetos de diferentes clases respondan de manera específica a los mismos métodos. Esto permite diseñar sistemas más flexibles y reutilizables, donde las subclases pueden extender o modificar el comportamiento de las superclases.

```
1  public class Animal {
2    public void hacerSonido() {
3      System.out.println("Sonido de animal");
4    }
5  }
6
7  public class Perro extends Animal {
8    @Override
9    public void hacerSonido() {
10     System.out.println("Guau");
11   }
12 }
13
14 public class Gato extends Animal {
15   @Override
16   public void hacerSonido() {
17     System.out.println("Miau");
18   }
19 }
```

```
20
21 // Uso del polimorfismo
22 Animal miAnimal = new Perro();
23 miAnimal.hacerSonido(); // Imprime "Guau"
```

Ejemplo 12.6: Ejemplo de herencia y polimorfismo en Java

Como puedes ver en el ejemplo, la clase `Animal` define un método `hacerSonido`, que es sobreescrito por las subclases `Perro` y `Gato`. Al crear una referencia de tipo `Animal` que apunta a un objeto de tipo `Perro`, podemos llamar al método `hacerSonido` y obtener el comportamiento específico del perro.

Para declarar una clase como subclase de otra, se utiliza la palabra clave **`extends`**. La subclase hereda todos los métodos y atributos de la superclase, y puede añadir nuevos o modificar los existentes. Además, las subclases pueden acceder a los métodos y atributos de la superclase utilizando la palabra clave **`super`**, en especial, podrán acceder a los miembros **`protected`**.

```
1  public class Vehiculo {
2    protected void encenderMotor() {
3      System.out.println("Motor encendido");
4    }
5  }
6
7  public class Coche extends Vehiculo {
8    public void arrancar() {
9      encenderMotor(); // Acceso permitido porque es protegido
10     System.out.println("Coche en marcha");
11   }
12 }
13
14 // Uso:
15 Coche miCoche = new Coche();
16 miCoche.arrancar(); // Imprime "Motor encendido" y "Coche en ↪
   marcha"
```

Ejemplo 12.7: Acceso a métodos protegidos de la superclase

12.3.1. Sobreescritura de métodos

La sobreescritura permite redefinir métodos heredados en subclases para adaptar o ampliar su funcionalidad.

```
1  public class Persona {
2    public void saludar() {
3      System.out.println("Hola");
4    }
5  }
6
```

```java
public class Estudiante extends Persona {
  @Override
  public void saludar() {
    System.out.println("Hola, soy estudiante");
  }
}
```

Ejemplo 12.8: Ejemplo de sobreescritura de métodos en Java

Esto se logra utilizando la anotación `@Override` para indicar que el método está siendo sobreescrito. Al llamar al método `saludar` en un objeto de tipo `Estudiante`, se ejecutará la implementación específica de esa clase. Si queremos llamar al método de la superclase, podemos hacerlo utilizando `super.saludar()`. Esto evitará la necesidad de duplicar código y permitirá que las subclases personalicen el comportamiento de los métodos heredados.

Ejercicio 12.3

Crea una jerarquía de clases que represente diferentes tipos de vehículos: `Vehiculo`, `Coche`, `Moto` y `Camion`. Implementa un método `mostrarDetalles` en cada clase que imprima información específica del vehículo. Utiliza el polimorfismo para crear una lista de vehículos y llamar al método `mostrarDetalles` para cada uno.

Pista: Puedes utilizar una lista de tipo `List<Vehiculo>` para almacenar los diferentes tipos de vehículos. Luego, recorre la lista y llama al método `mostrarDetalles` en cada vehículo.

12.4. Jerarquía de clases: Superclases y subclases

Las superclases definen comportamientos y atributos comunes, mientras que las subclases los heredan y pueden añadir o modificar funcionalidades.

```java
public class Garaje {
  protected List<Vehiculo> vehiculos;

  public void agregarVehiculo(Vehiculo vehiculo) {
    vehiculos.add(vehiculo);
  }

  public void eliminarVehiculo(Vehiculo vehiculo) {
    vehiculos.remove(vehiculo);
  }

  public void mostrarVehiculos() {
    for (Vehiculo v : vehiculos) {
      v.mostrarDetalles();
```

```
    }
  }
}

public class GarajeDeCoches extends Garaje {
  @Override
  public void agregarVehiculo(Vehiculo vehiculo) {
    if (vehiculo instanceof Coche) {
      super.agregarVehiculo(vehiculo);
    } else {
      System.out.println("Solo se pueden agregar coches al garaje de coches");
    }
  }
}
```

Ejemplo 12.9: Ejemplo de jerarquía de clases en Java

Consejo

Utiliza la palabra clave `instanceof` para verificar el tipo de un objeto antes de realizar una conversión de tipo (casting). Esto ayuda a evitar errores en tiempo de ejecución.

Es posible utilizar `instanceof` para comprobar si un objeto es una instancia de una clase específica o de una de sus subclases, dentro de una sentencia `switch`:

```
switch (vehiculo) {
  case Coche c -> System.out.println("Es un coche");
  case Moto m -> System.out.println("Es una moto");
  case Camion cam -> System.out.println("Es un camión");
  default -> System.out.println("Tipo de vehículo desconocido");
}

```

Ejemplo 12.10: Uso de instanceof y switch con patrones

Ejercicio 12.4

Recupera el ejercicio 12.2 y crea una jerarquía de clases que represente diferentes tipos de libros: `Libro`, `LibroDigital` y `LibroFisico`. Implementa un método `mostrarDetalles` en cada clase que imprima información específica del libro. Utiliza el polimorfismo para crear una lista de libros y llamar al método `mostrarDetalles` para cada uno. Cada clase deberá heredar de una superclase común `Libro`.

12.5. Constructores y herencia

Los constructores de las superclases pueden ser invocados desde las subclases usando `super()`, asegurando la correcta inicialización de los objetos. Esto permite que las subclases hereden el estado inicial de la superclase y añadan su propia lógica de inicialización, así como sus propios campos y métodos.

```
public class Empleado {
  private String nombre;
  public Empleado(String nombre) {
    this.nombre = nombre;
  }
}

public class Gerente extends Empleado {
  private String departamento;
  public Gerente(String nombre, String departamento) {
    super(nombre);
    this.departamento = departamento;
  }
}
```

Ejemplo 12.11: Ejemplo de constructores y herencia en Java

La llamada a `super(nombre)` en el constructor de `Gerente` asegura que el campo `nombre` de la superclase `Empleado` sea inicializado correctamente antes de que se ejecute el código específico del constructor de la subclase. Esta llamada debe ser la primera instrucción en el constructor de la subclase.

Importante

Si una superclase no tiene un constructor sin parámetros, es obligatorio llamar a uno de sus constructores desde la subclase utilizando `super()`. De lo contrario, el compilador generará un error.
Además, si la superclase tiene un constructor con parámetros, la subclase debe llamar a ese constructor con los argumentos adecuados. Si no se especifica una llamada a `super()`, el compilador intentará llamar al constructor sin parámetros de la superclase, lo que puede resultar en un error si no existe.

Ejercicio 12.5

Continúa con el ejercicio 12.2 y modifica los constructores de las clases `LibroDigital` y `LibroFisico` para que hereden de la superclase `Libro`. Asegúrate de que cada subclase inicialice correctamente los atributos heredados y los propios. Implementa un método `mostrarDetalles` en cada

subclase que imprima información específica del libro, incluyendo el tipo de libro (digital o físico).
En este caso, la clase `LibroFisico` será la única con el parámetro páginas, mientras que la clase `LibroDigital` tendrá un parámetro adicional para el formato del libro (por ejemplo, PDF, EPUB, etc.).

12.6. Clases y métodos abstractos y finales

Las clases y métodos abstractos definen comportamientos que deben ser implementados por las subclases. Los elementos finales (final) no pueden ser modificados ni heredados. A diferencia de las clases normales, las clases abstractas no pueden ser instanciadas directamente. En su lugar, se utilizan como base para crear subclases que implementen los métodos abstractos definidos en la clase abstracta.

```
1  public abstract class Figura {
2    public abstract double area();
3    public final void mostrar() {
4      System.out.println("Esta figura tiene un área de: " + area()↪
     );
5    }
6  }
7
8  public class Circulo extends Figura {
9    private double radio;
10   public Circulo(double radio) {
11     this.radio = radio;
12   }
13   @Override
14   public double area() {
15     return Math.PI * radio * radio;
16   }
17 }
```

Ejemplo 12.12: Ejemplo de clase y método abstracto y final en Java

Como puedes ver en el ejemplo, la clase `Figura` es abstracta y define un método abstracto `area()`. La subclase `Circulo` implementa este método y proporciona su propia lógica para calcular el área. Además, el método `mostrar()` es final, lo que significa que no puede ser sobrescrito por las subclases. Sin embargo, este método delega parte de su lógica al método abstracto `area()`, que sí debe ser implementado por las subclases. Esto facilita la reutilización de código y garantiza que todas las subclases proporcionen una implementación del método `area()`.

12.6.1. Clases `sealed`

Las clases `sealed` (selladas), introducidas en Java 17, permiten restringir qué clases pueden heredar de una clase o implementar una interfaz. Esto proporciona mayor control sobre la jerarquía de clases y mejora la seguridad y mantenibilidad del código.

Para declarar una clase como sellada, se utiliza la palabra clave `sealed` junto con la cláusula `permits`, que indica explícitamente qué subclases pueden extenderla. Las subclases deben ser `final`, `sealed` o `non-sealed`.

```
public sealed class Figura permits Circulo, Rectangulo { }

public final class Circulo extends Figura { }

public final class Rectangulo extends Figura { }
```

Ejemplo 12.13: Ejemplo de clase sealed en Java

En este ejemplo, la clase `Figura` es sellada y solo permite que las clases `Circulo` y `Rectangulo` la extiendan. Esto significa que no se pueden crear nuevas subclases de `Figura` a menos que se declare explícitamente en la cláusula `permits`. Si se intenta crear una subclase de `Figura` que no esté en la lista de permitidas, el compilador generará un error. Esto permite controlar la jerarquía de clases y evitar extensiones no deseadas.

También es posible declarar una subclase como `non-sealed` para permitir que otras clases la extiendan libremente:

```
public sealed class Figura permits Circulo, Rectangulo, Poligono
    { }

public final class Circulo extends Figura { }

public final class Rectangulo extends Figura { }

public non-sealed class Poligono extends Figura { }
```

Ejemplo 12.14: Subclase non-sealed

En este caso, la clase `Poligono` es `non-sealed`, lo que significa que cualquier clase puede extenderla, mientras que las clases `Circulo` y `Rectangulo` siguen siendo `final` y no pueden ser extendidas. Esto permite que en el futuro se puedan añadir nuevas subclases, pero con un control más estricto sobre la jerarquía de clases.

12.7. Interfaces

Las interfaces permiten definir contratos de métodos que las clases deben implementar. A diferencia de con las clases abstractas, una clase puede implementar múltiples interfaces. Las interfaces son útiles para definir comportamientos comunes que pueden ser implementados por diferentes clases, independientemente de su jerarquía de herencia.

```
interface Imprimible {
  void imprimir();
}

interface Guardable {
  void guardar();
}

public class Documento implements Imprimible, Guardable {
  @Override
  public void imprimir() {
    System.out.println("Imprimiendo documento...");
  }

  @Override
  public void guardar() {
    System.out.println("Guardando documento...");
  }
}

// Uso:
Documento doc = new Documento();
doc.imprimir(); // Imprime "Imprimiendo documento..."
doc.guardar(); // Imprime "Guardando documento..."
```

Ejemplo 12.15: Ejemplo de clase que implementa múltiples interfaces

En este ejemplo, la clase `Documento` implementa las interfaces `Imprimible` y `Guardable`, por lo que debe proporcionar una implementación para los métodos definidos en ambas interfaces. A diferencia de con la herencia, se usa la palabra clave **`implements`** para indicar que una clase implementa una o más interfaces. Es posible implementar una o varias interfaces y extender de una superclase al mismo tiempo, lo que permite combinar comportamientos de diferentes fuentes.

```
class Animal {
  public void comer() {
    System.out.println("El animal come");
  }
}

interface Volador {
```

```
8   void volar();
9 }
10
11 interface Nadador {
12   void nadar();
13 }
14
15 public class Pato extends Animal implements Volador, Nadador {
16   @Override
17   public void volar() {
18     System.out.println("El pato vuela");
19   }
20
21   @Override
22   public void nadar() {
23     System.out.println("El pato nada");
24   }
25 }
26
27 // Uso:
28 Pato pato = new Pato();
29 pato.comer(); // Imprime "El animal come"
30 pato.volar(); // Imprime "El pato vuela"
31 pato.nadar(); // Imprime "El pato nada"
```

Ejemplo 12.16: Ejemplo de extends e implements a la vez

En este ejemplo, la clase `Pato` hereda de `Animal` y también implementa las interfaces `Volador` y `Nadador`, combinando herencia e implementación de interfaces.

12.7.1. La interfaz `Comparable`

La interfaz `Comparable<T>` es una interfaz genérica de Java que permite definir un criterio de comparación natural entre objetos de una clase. Al implementar esta interfaz, una clase debe proporcionar la implementación del método `compareTo`, que se utiliza para comparar el objeto actual con otro objeto del mismo tipo.

```
1 public class Persona implements Comparable<Persona> {
2   private String nombre;
3   private int edad;
4
5   public Persona(String nombre, int edad) {
6     this.nombre = nombre;
7     this.edad = edad;
8   }
9
10  @Override
```

```
  public int compareTo(Persona otra) {
    return Integer.compare(this.edad, otra.edad);
  }
}
```

Ejemplo 12.17: Implementación de la interfaz Comparable

En este ejemplo, la clase `Persona` implementa `Comparable<Persona>` y define el método `compareTo` para comparar personas por su edad. Si el resultado es negativo, el objeto actual es menor; si es cero, son iguales; si es positivo, es mayor.

Esto permite ordenar colecciones de objetos usando métodos como `Collections.sort()`:

```
List<Persona> personas = new ArrayList<>();
personas.add(new Persona("Ana", 30));
personas.add(new Persona("Luis", 25));
personas.add(new Persona("Marta", 35));

Collections.sort(personas);
// Ahora la lista está ordenada por edad ascendente
```

Ejemplo 12.18: Ordenar una lista de objetos con Comparable

Implementar `Comparable` es útil cuando se necesita un orden natural para los objetos de una clase, como al almacenar elementos en estructuras ordenadas (`TreeSet`, `TreeMap`) o al ordenar listas.

12.7.2. Métodos por defecto en interfaces

A partir de Java 8, las interfaces pueden incluir métodos por defecto (default methods), que proporcionan una implementación predeterminada. Esto permite añadir nuevas funcionalidades a las interfaces sin romper el código existente que las implementa.

Para definir un método por defecto, se utiliza la palabra clave `default`:

```
interface Saludo {
  default void saludar() {
    System.out.println("¡Hola!");
  }
}

class Persona implements Saludo {
  // Puede sobrescribir saludar() si lo desea
}

// Uso:
Persona p = new Persona();
```

```
p.saludar(); // Imprime "¡Hola!"
```

Ejemplo 12.19: Método por defecto en una interfaz

Las clases que implementan la interfaz pueden sobrescribir el método por defecto si necesitan un comportamiento específico. Los métodos por defecto son útiles para evolucionar interfaces y evitar la duplicación de código.

Ejercicio 12.6

Continúa con el ejercicio 12.2 y crea una interfaz `ItemBiblioteca` que defina métodos como `mostrarDetalles` (sin implementación), e `imprimirDetalles` (que imprima por pantalla el resultado de `mostrarDetalles()`). Implementa esta interfaz en las clases `Libro`, `LibroDigital` y `LibroFisico`. Asegúrate de que cada clase proporcione su propia implementación de los métodos definidos en la interfaz.
Pista: Comprueba que ocurre si solo declaras la clase abstracta `Libro` como extensión de la interfaz `ItemBiblioteca` y no implementas la interfaz en ella.

12.8. Enumerados

Los enumerados permiten definir un conjunto fijo de constantes, mejorando la legibilidad y seguridad del código.

```
public enum DiaSemana {
  LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
}

// Uso del enumerado
DiaSemana hoy = DiaSemana.LUNES;
System.out.println(hoy); // Imprime "LUNES"
```

Ejemplo 12.20: Ejemplo de enumerado en Java

Usaremos enumerados para representar valores fijos y conocidos, y cuando queramos que no se puedan crear instancias arbitrarias de una clase. Los enumerados pueden tener métodos, constructores y atributos, lo que los hace muy versátiles.

12.8.1. Enumerados con métodos y atributos

Los enumerados pueden tener constructores, atributos y métodos, igual que una clase normal, pero sus instancias son fijas y predefinidas. Esto permite asociar información adicional y comportamientos a cada constante del enumerado.

```java
public enum NivelPrioridad {
  BAJA(1), MEDIA(2), ALTA(3);

  private final int valor;

  NivelPrioridad(int valor) {
    this.valor = valor;
  }

  public int getValor() {
    return valor;
  }

  public void mostrarMensaje() {
    switch (this) {
      case BAJA -> System.out.println("Prioridad baja");
      case MEDIA -> System.out.println("Prioridad media");
      case ALTA -> System.out.println("¡Prioridad alta!");
    }
  }
}

// Uso:
NivelPrioridad prioridad = NivelPrioridad.ALTA;
System.out.println(prioridad.getValor()); // Imprime 3
prioridad.mostrarMensaje(); // Imprime "¡Prioridad alta!"
```

Ejemplo 12.21: Enumerado con atributos y métodos

Además, los enumerados pueden implementar interfaces, lo que permite definir comportamientos comunes para todas las constantes del enumerado.

```java
interface Operacion {
  int aplicar(int a, int b);
}

public enum TipoOperacion implements Operacion {
  SUMA {
    @Override
    public int aplicar(int a, int b) {
      return a + b;
    }
  },
  RESTA {
    @Override
    public int aplicar(int a, int b) {
      return a - b;
    }
  },
  MULTIPLICACION {
    @Override
```

```
    public int aplicar(int a, int b) {
      return a * b;
    }
  }
}

// Uso:
TipoOperacion op = TipoOperacion.SUMA;
System.out.println(op.aplicar(3, 4)); // Imprime 7
```

Ejemplo 12.22: Enumerado que implementa una interfaz

12.8.2. Iteración sobre enumerados

Para iterar sobre todas las constantes de un enumerado, se puede utilizar el método estático `values()`, que devuelve un arreglo con todas las instancias del enumerado en el orden en que fueron declaradas.

```
for (DiaSemana dia : DiaSemana.values()) {
  System.out.println(dia);
}
```

Ejemplo 12.23: Iterar sobre los valores de un enumerado

Esto imprimirá todos los días de la semana uno por línea. Este patrón es útil para mostrar menús, validar entradas o realizar operaciones sobre todos los valores posibles de un enumerado.

12.8.3. Conversión de `String` a un valor de enumerado

Para convertir una cadena de texto (`String`) en un valor de un enumerado, se puede utilizar el método estático `valueOf` que proporciona Java. Este método lanza una excepción si la cadena no coincide exactamente con el nombre de una constante del enumerado. Es común combinarlo con un bucle para validar la entrada del usuario:

```
String entrada = "VIERNES";
try {
  DiaSemana dia = DiaSemana.valueOf(entrada);
  System.out.println("Día seleccionado: " + dia);
} catch (IllegalArgumentException e) {
  System.out.println("Valor no válido para el día de la semana."↪
  );
}
```

Ejemplo 12.24: Conversión de String a enumerado

Si se desea hacer la conversión de forma insensible a mayúsculas/minúsculas, se puede iterar sobre los valores del enumerado:

```
String entrada = "viernes";
DiaSemana diaSeleccionado = null;
for (DiaSemana dia : DiaSemana.values()) {
  if (dia.name().equalsIgnoreCase(entrada)) {
    diaSeleccionado = dia;
    break;
  }
}
if (diaSeleccionado != null) {
  System.out.println("Día seleccionado: " + diaSeleccionado);
} else {
  System.out.println("Valor no válido para el día de la semana."
  );
}
```

Ejemplo 12.25: Conversión insensible a mayúsculas/minúsculas

Este patrón es útil para validar y convertir entradas de usuario a valores de enumerados de forma segura.

12.8.4. Enumerados y `switch`

Los enumerados pueden utilizarse en sentencias **`switch`**, lo que permite ejecutar diferentes bloques de código según el valor del enumerado. A partir de Java 12, se puede usar la sintaxis mejorada de **`switch`** con flechas (`->`) y bloques de código.

```
DiaSemana dia = DiaSemana.LUNES;

switch (dia) {
  case LUNES, MARTES, MIERCOLES, JUEVES, VIERNES ->
    System.out.println("Es un día laborable");
  case SABADO, DOMINGO ->
    System.out.println("Es fin de semana");
}
```

Ejemplo 12.26: Uso de switch con enumerados

Esto facilita la escritura de código claro y seguro, ya que el compilador verifica que se contemplen todos los valores posibles del enumerado. Si se omite algún valor, el compilador puede advertirlo, ayudando a evitar errores.

Además, se pueden usar los enumerados en expresiones **`switch`** para devolver valores:

```
String mensaje = switch (dia) {
  case LUNES -> "Comienza la semana";
  case VIERNES -> "¡Ya es viernes!";
  case SABADO, DOMINGO -> "Disfruta el fin de semana";
  default -> "Día normal";
```

```
};
System.out.println(mensaje);
```

Ejemplo 12.27: Switch como expresión con enumerados

El uso de enumerados con `switch` mejora la legibilidad y robustez del código, especialmente cuando se manejan conjuntos fijos de valores.

12.8.5. Enumerados como `Singleton`

Los enumerados en Java pueden utilizarse para implementar el patrón *Singleton* de forma sencilla y segura. Un *singleton* es una clase de la que solo puede existir una única instancia en toda la aplicación. Utilizar un enumerado para este propósito garantiza que la instancia sea única incluso en presencia de serialización o reflexión.

```
public enum GestorConfiguracion {
  INSTANCIA;

  private String configuracion;

  public void cargar(String valor) {
    this.configuracion = valor;
  }

  public String obtener() {
    return configuracion;
  }
}

// Uso:
GestorConfiguracion.INSTANCIA.cargar("modo=produccion");
System.out.println(GestorConfiguracion.INSTANCIA.obtener()); // ↪
  Imprime "modo=produccion"
```

Ejemplo 12.28: Singleton usando un enumerado

Este enfoque es recomendado por los expertos de Java, ya que es simple, seguro frente a ataques de serialización y garantiza que solo exista una instancia del *singleton*.

Ejercicio 12.7

Continúa el ejercicio 12.2 y crea un enumerado `GeneroLibro` que represente diferentes géneros literarios por ejemplo: `FICCION`, `NO_FICCION`, `CIENCIA_FICCION`, etc. Modifica las clases `Libro`, `LibroDigital` y `LibroFisico` para incluir un atributo de tipo `GeneroLibro`. Modifica el método `mostrarDetalles` en cada clase para incluir el género del libro. Utiliza un

`switch` para imprimir un mensaje diferente según el género del libro.

12.9. El tipo `record`

A partir de Java 16, se introduce el tipo especial `record`, que permite definir clases inmutables de manera concisa. Los *records* son útiles para representar datos simples, como estructuras o tuplas, sin necesidad de escribir código repetitivo para constructores, métodos `equals`, `hashCode` y `toString`.

```java
public record Punto(int x, int y) { }

// Uso del record
Punto p = new Punto(3, 5);
System.out.println(p.x()); // Imprime 3
System.out.println(p.y()); // Imprime 5
System.out.println(p);   // Imprime "Punto[x=3, y=5]"
```

Ejemplo 12.29: Ejemplo de record en Java

Los *records* son inmutables: sus campos no pueden cambiarse después de la creación del objeto. Además, los *records* pueden implementar interfaces y contener métodos adicionales si es necesario.

```java
public record Persona(String nombre, int edad) {
  public boolean esMayorDeEdad() {
    return edad >= 18;
  }
}

// Uso:
Persona persona = new Persona("Ana", 20);
System.out.println(persona.esMayorDeEdad()); // Imprime true
```

Ejemplo 12.30: Record con método adicional

El uso de *records* mejora la legibilidad y reduce errores al trabajar con datos inmutables.

Consejo

Utiliza *records* cuando necesites representar datos inmutables de forma sencilla y clara. Son ideales para DTOs (*Data Transfer Objects*) y otras estructuras de datos simples.
Puedes usar *records* para definir clases que representen entidades con atributos inmutables, como coordenadas, fechas, o cualquier otro tipo de dato que no necesite ser modificado una vez creado. Además, son útiles para

agrupar parámetros de un método o función, evitando la necesidad de crear clases adicionales para representar esos datos.

Ejercicio 12.8

Crea un `record` llamado `Coordenadas` que represente un punto en un plano 2D con atributos `x` e `y` de tipo `int`, e implementa un método que calcule la distancia entre dos puntos. Dentro de este *record*, crea una constante que representa el punto de origen (0, 0).
Después, crea un método *angulo()* que calcule el ángulo entre dos puntos respecto al eje X. Utiliza la fórmula del ángulo en coordenadas polares: `angulo = Math.atan2(y, x)`. Asegúrate de que el método retorne el ángulo en grados. Una vez creado ese método, haz que los puntos sean ordenables, de forma que puedas ordenarlos por su ángulo respecto al origen.

Resumen

En este capítulo hemos explorado conceptos avanzados de la programación orientada a objetos en Java, como la sobrecarga y sobreescritura de métodos, la composición de clases, la herencia y el polimorfismo. Hemos visto cómo organizar jerarquías de clases mediante superclases y subclases, el uso de constructores en contextos de herencia y la importancia de las clases y métodos abstractos y finales. También se abordaron las interfaces, los enumerados y el tipo especial `record`, que permiten diseñar sistemas más robustos, flexibles y mantenibles. Estos conceptos son fundamentales para desarrollar aplicaciones complejas y aprovechar al máximo las capacidades del lenguaje Java.

13

Gestión de bases de datos

Las bases de datos son herramientas fundamentales en el desarrollo de aplicaciones modernas, ya que permiten almacenar, organizar y gestionar grandes volúmenes de información de manera eficiente y segura. La correcta gestión de bases de datos facilita el acceso concurrente, la integridad de los datos y la escalabilidad de los sistemas. En este capítulo se abordarán los conceptos esenciales relacionados con el acceso, manipulación y administración de bases de datos, así como los estándares y características principales que rigen su funcionamiento. Los temas a tratar incluyen:

1. **Acceso a bases de datos**: Se explicarán los conceptos fundamentales sobre cómo las aplicaciones interactúan con las bases de datos, los diferentes tipos de bases de datos (relacionales y no relacionales), así como los estándares más utilizados (por ejemplo, SQL, NoSQL, ODBC, JDBC) y las características que deben cumplir los sistemas de gestión de bases de datos (SGBD), como la integridad, seguridad, concurrencia y escalabilidad.

2. **Establecimiento de conexiones**: Se detallará el proceso para conectar una aplicación con una base de datos, incluyendo la configuración de parámetros de conexión, autenticación de usuarios, manejo de errores y buenas prácticas para mantener conexiones seguras y eficientes.

3. **Almacenamiento, recuperación, actualización y eliminación de información en bases de datos**: Se abordarán las operaciones básicas de manipulación de datos (CRUD: Create, Read, Update, Delete), el uso de sentencias SQL para realizar estas operaciones, la gestión de transacciones para asegurar la consistencia de los datos y ejemplos prácticos de cómo implementar estas acciones desde distintos lenguajes de programación.

13.1. Acceso a bases de datos

El acceso a bases de datos es un proceso fundamental en el desarrollo de aplicaciones. Existen diferentes tipos de bases de datos, siendo las más comunes

las bases de datos relacionales (como MySQL, PostgreSQL, Oracle) y las no relacionales o NoSQL (como MongoDB, Redis). Para interactuar con ellas, se utilizan estándares y tecnologías como SQL para bases de datos relacionales y APIs específicas para NoSQL.

En el caso de Java, el estándar más utilizado para acceder a bases de datos es JDBC (Java Database Connectivity), que proporciona una API para conectar y ejecutar consultas en bases de datos desde aplicaciones Java.

Cuadro 13.1: Comparativa de sistemas de gestión de bases de datos populares

Base de datos	**Tipo**	**Lenguaje principal**	**Características principales**
MySQL	SQL	SQL	Código abierto, ampliamente utilizada, soporte ACID, replicación, comunidad grande.
PostgreSQL	SQL	SQL	Código abierto, extensible, soporte avanzado de transacciones, tipos de datos personalizados.
Oracle Database	SQL	SQL	Comercial, alto rendimiento, escalabilidad, características empresariales avanzadas.
Microsoft SQL Server	SQL	SQL	Integración con productos Microsoft, herramientas de administración avanzadas, soporte empresarial.
SQLite	SQL	SQL	Ligera, embebida, sin servidor, ideal para aplicaciones móviles y de escritorio.
MongoDB	NoSQL	BSON / JSON	Orientada a documentos, escalabilidad horizontal, flexible, consultas ad-hoc.
Redis	NoSQL	Clave-valor	Almacenamiento en memoria, muy rápido, soporte para estructuras de datos avanzadas.
Cassandra	NoSQL	CQL	Distribuida, alta disponibilidad, escalabilidad horizontal, tolerancia a fallos.
Neo4j	NoSQL	Cypher	Orientada a grafos, ideal para relaciones complejas, consultas eficientes de grafos.

Base de datos	Tipo	Lenguaje principal	Características principales
Elasticsearch	NoSQL	JSON	Orientada a búsquedas, indexación de texto completo, análisis en tiempo real.

En este capítulo, nos centraremos en el acceso a bases de datos relacionales utilizando JDBC, que es la forma estándar de conectar aplicaciones Java con bases de datos. JDBC proporciona una interfaz para ejecutar consultas SQL, gestionar conexiones y manejar resultados de manera eficiente.

13.2. Establecimiento de conexiones

Para que una aplicación pueda interactuar con una base de datos, primero debe establecer una conexión. En Java, esto se realiza a través de JDBC, utilizando un *driver* específico para cada tipo de base de datos. A continuación se muestra un ejemplo básico de cómo establecer una conexión a una base de datos MySQL:

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.SQLException;
4
5  public class ConexionBD {
6    public static void main(String[] args) {
7      String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
8      String usuario = "root";
9      String contrasena = "contrasena";
10
11     try (Connection conexion = DriverManager.getConnection(url, ↪
     usuario, contrasena)) {
12       System.out.println("Conexión exitosa a la base de datos.")↪
     ;
13     } catch (SQLException e) {
14       System.out.println("Error al conectar: " + e.getMessage())↪
     ;
15     }
16   }
17 }
```

Ejemplo 13.1: Conexión a una base de datos MySQL con JDBC

Es importante manejar correctamente las excepciones y cerrar las conexiones para evitar fugas de recursos.

Consejo

Para la realización de los ejercicios de este capítulo, puedes utilizar una base de datos local como MySQL o SQLite. Si dispones de *Docker*, puedes levantar un contenedor con una base de datos MySQL o PostgreSQL para practicar. Para ello, ejecuta el siguiente comando:

```
docker run --name mysql -e MYSQL_ROOT_PASSWORD=contrasena -e MYSQL_DATABASE=mi_base_de_datos -p 3306:3306 mysql:latest

```

Recuerda disponer del *driver* JDBC correspondiente en tu proyecto. Para MySQL, puedes añadir la dependencia en tu archivo `pom.xml` si usas Maven:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.33</version>
</dependency>

```

13.2.1. Parámetros de conexión

Los parámetros de conexión son los datos necesarios para que una aplicación pueda comunicarse correctamente con una base de datos. Los parámetros más comunes incluyen:

- **URL de conexión**: Especifica la ubicación de la base de datos y el tipo de SGBD. Por ejemplo, para MySQL: `jdbc:mysql://localhost:3306/mi_base_de_datos`.
- **Usuario**: Nombre de usuario con permisos para acceder a la base de datos.
- **Contraseña**: Clave asociada al usuario.
- **Opciones adicionales**: Algunos SGBD permiten añadir parámetros extra en la URL, como el uso de SSL, el juego de caracteres, el tiempo de espera, etc.

Un ejemplo de URL de conexión para diferentes bases de datos:

- MySQL: `jdbc:mysql://localhost:3306/mi_base_de_datos`
- PostgreSQL: `jdbc:postgresql://localhost:5432/mi_base_de_datos`

- SQLite: `jdbc:sqlite:mi_base_de_datos.db`

Importante

Es recomendable no almacenar las credenciales directamente en el código fuente. Utiliza archivos de configuración o variables de entorno para gestionar estos datos de forma segura.
Para leer las credenciales, puedes almacenarlas en variables de entorno. Podrás acceder a ellas en tu código Java de la siguiente manera:

```
String usuario = System.getenv("DB_USER");
String contrasena = System.getenv("DB_PASSWORD");

```

13.2.2. Ejecución de consultas SQL

Para ejecutar consultas SQL, como la creación de tablas, puedes utilizar el método ‘executeUpdate‘ de un objeto ‘Statement‘ o ‘PreparedStatement‘. A continuación se muestra un ejemplo de cómo crear una tabla llamada ‘usuarios‘ desde Java usando JDBC:

```
String sql = "CREATE TABLE IF NOT EXISTS usuarios (" +
        "id INT AUTO_INCREMENT PRIMARY KEY," +
        "nombre VARCHAR(100) NOT NULL," +
        "email VARCHAR(100) NOT NULL UNIQUE" +
        ")";
try (Statement stmt = conexion.createStatement()) {
  stmt.executeUpdate(sql);
  System.out.println("Tabla 'usuarios' creada correctamente.");
} catch (SQLException e) {
  System.out.println("Error al crear la tabla: " + e.getMessage↪
  ());
}
```

Ejemplo 13.2: Crear una tabla en la base de datos

13.3. Almacenamiento, recuperación, actualización y eliminación de información en bases de datos

Las operaciones básicas sobre una base de datos se conocen como operaciones CRUD (Crear, Leer, Actualizar, Eliminar). A continuación se presentan ejemplos de cómo realizar estas operaciones en Java usando JDBC:

13.3.1. Insertar datos

Una de las operaciones más comunes es insertar datos en una tabla. Para ello, se utiliza la sentencia 'INSERT INTO'. A continuación se muestra un ejemplo de cómo insertar un registro en la tabla 'usuarios':

```
String sql = "INSERT INTO usuarios (nombre, email) VALUES (?, ?)";
try (PreparedStatement stmt = conexion.prepareStatement(sql)) {
  stmt.setString(1, "Juan");
  stmt.setString(2, "juan@email.com");
  stmt.executeUpdate();
}
```

Ejemplo 13.3: Insertar un registro en la base de datos

En este ejemplo, se utiliza un 'PreparedStatement' para evitar inyección SQL y mejorar la seguridad. Los signos de interrogación ('¿) son marcadores de posición que se reemplazarán por los valores reales al ejecutar la consulta.

13.3.2. Consultar datos

Para recuperar datos de una base de datos, se utiliza la sentencia 'SELECT'. A continuación se muestra un ejemplo de cómo consultar todos los registros de la tabla 'usuarios':

```
String sql = "SELECT * FROM usuarios";
try (Statement stmt = conexion.createStatement();
  ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    System.out.println("ID: " + rs.getInt("id"));
    System.out.println("Nombre: " + rs.getString("nombre"));
    System.out.println("Email: " + rs.getString("email"));
  }
}
```

Ejemplo 13.4: Consultar registros de la base de datos

El resultado de la consulta se almacena en un objeto 'ResultSet', que permite iterar sobre los registros devueltos. Cada llamada a 'rs.next()' avanza al siguiente registro. Los métodos 'getInt', 'getString', etc., se utilizan para obtener los valores de las columnas del registro actual. De esta manera, se pueden procesar los datos recuperados de la base de datos.

13.3.3. Actualizar datos

Para modificar datos existentes en una tabla, se utiliza la sentencia 'UPDATE'. Es importante especificar una condición en la cláusula 'WHERE' para evitar ac-

tualizar todos los registros accidentalmente. El siguiente ejemplo muestra cómo actualizar el correo electrónico de un usuario identificado por su nombre:

```
String sql = "UPDATE usuarios SET email = ? WHERE nombre = ?";
try (PreparedStatement stmt = conexion.prepareStatement(sql)) {
  stmt.setString(1, "nuevo@email.com");
  stmt.setString(2, "Juan");
  stmt.executeUpdate();
}
```

Ejemplo 13.5: Actualizar un registro en la base de datos

13.3.4. Eliminar datos

Para eliminar datos de una tabla, se utiliza la sentencia 'DELETE'. Es fundamental incluir una condición en la cláusula 'WHERE' para evitar eliminar todos los registros accidentalmente. El siguiente ejemplo muestra cómo eliminar un usuario identificado por su nombre:

```
String sql = "DELETE FROM usuarios WHERE nombre = ?";
try (PreparedStatement stmt = conexion.prepareStatement(sql)) {
  stmt.setString(1, "Juan");
  stmt.executeUpdate();
}
```

Ejemplo 13.6: Eliminar un registro de la base de datos

Ejercicio 13.1

Crea un programa en Java, que gestione un listín telefónico. Debe permitir realizar las siguientes operaciones:

- Insertar un nuevo contacto con nombre, teléfono y correo electrónico.
- Consultar todos los contactos.
- Actualizar el teléfono de un contacto dado su nombre.
- Eliminar un contacto dado su nombre.
- Buscar un contacto por nombre y mostrar sus datos.
- Buscar un contacto por teléfono y mostrar sus datos.

Pista: para simplificar el código, crea un método que cree la tabla, si el programa recibe por parámetro la palabra `init`*. Puedes hacer que este programa sea interactivo, pidiendo al usuario qué operación desea realizar y mostrando un menú por consola.*

13.4. Gestión de transacciones

Las transacciones permiten agrupar varias operaciones en una sola unidad de trabajo, asegurando que todas se realicen correctamente o ninguna se aplique en caso de error. En JDBC, se pueden gestionar transacciones de la siguiente manera:

```
try {
  conexion.setAutoCommit(false);

  // Operaciones SQL aquí

  conexion.commit(); // Confirma la transacción
} catch (SQLException e) {
  conexion.rollback(); // Revierte la transacción en caso de error
}
```

Ejemplo 13.7: Gestión de transacciones en JDBC

El método `setAutoCommit(false)` desactiva el modo de autocommit, lo que significa que las operaciones no se guardarán automáticamente en la base de datos hasta que se llame a `commit()`. Si ocurre un error, se puede llamar a `rollback()` para revertir todas las operaciones realizadas desde el último `commit`.

13.4.1. Gestión de transacciones en PostgreSQL

En PostgreSQL, las transacciones se gestionan de manera similar a otros SGBD. Sin embargo, es importante destacar que PostgreSQL ofrece características avanzadas como el aislamiento de transacciones y la posibilidad de utilizar puntos de guardado (*savepoints*) para revertir parcialmente una transacción.

```
BEGIN; -- Inicia una transacción
INSERT INTO usuarios (nombre, email) VALUES ('Ana', 'ana@example.com');
COMMIT; -- Confirma la transacción
```

Ejemplo 13.8: Uso de transacciones en PostgreSQL

A diferencia de otros SGBD, PostgreSQL permite crear puntos de guardado dentro de una transacción, lo que permite revertir a un estado anterior sin deshacer toda la transacción. Esto es útil para manejar errores en operaciones complejas.

```
BEGIN; -- Inicia una transacción
SAVEPOINT sp1; -- Crea un punto de guardado
INSERT INTO usuarios (nombre, email) VALUES ('Ana', 'ana@example.com');
```

```
4 ROLLBACK TO sp1; -- Revierte al punto de guardado
5 COMMIT; -- Confirma la transacción
```

Ejemplo 13.9: Uso de savepoints en PostgreSQL

Este mecanismo permite una mayor flexibilidad en la gestión de transacciones, permitiendo deshacer solo ciertas partes de una transacción sin perder todo el trabajo realizado hasta ese momento. Además, PostgreSQL ejecutará automáticamente un *rollback* si la conexión se cierra inesperadamente, garantizando la integridad de los datos, así como ejecutar cualquier trigger definido en la base de datos, y asegurar que las restricciones de integridad se mantengan.

Consejo

Utiliza transacciones para garantizar la integridad de los datos, especialmente en operaciones que afectan a múltiples tablas o registros. Asegúrate de manejar correctamente las excepciones y realizar un *rollback* en caso de error.

Resumen

En este capítulo hemos aprendido los conceptos fundamentales sobre la gestión de bases de datos en el desarrollo de aplicaciones. Se han presentado los diferentes tipos de bases de datos, los estándares y tecnologías más utilizados, y cómo establecer conexiones seguras desde Java utilizando JDBC. Además, se han mostrado ejemplos prácticos de las operaciones básicas CRUD (crear, leer, actualizar y eliminar datos), así como la importancia de las transacciones para mantener la integridad de la información. Estos conocimientos son esenciales para desarrollar aplicaciones robustas y seguras que requieran almacenar y manipular datos de manera eficiente.

Cuadro 13.2: Principales métodos de la clase `ResultSet` en JDBC

Método	Descripción
`boolean next()`	Avanza el cursor al siguiente registro del conjunto de resultados. Devuelve `true` si hay un registro disponible.
`int getInt(String columnLabel)`	Obtiene el valor de la columna especificada como un entero.
`String getString(String columnLabel)`	Obtiene el valor de la columna especificada como una cadena de texto.
`double getDouble(String columnLabel)`	Obtiene el valor de la columna especificada como un número de punto flotante.
`boolean getBoolean(String columnLabel)`	Obtiene el valor de la columna especificada como un valor booleano.
`Date getDate(String columnLabel)`	Obtiene el valor de la columna especificada como un objeto `java.sql.Date`.
`Object getObject(String columnLabel)`	Obtiene el valor de la columna especificada como un objeto genérico.
`void close()`	Cierra el `ResultSet` y libera los recursos asociados.
`boolean wasNull()`	Indica si la última columna leída tenía valor SQL `NULL`.
`int findColumn(String columnLabel)`	Devuelve el índice de la columna especificada por nombre.
`ResultSetMetaData getMetaData()`	Devuelve información sobre las columnas del conjunto de resultados.

14

Mantenimiento de la persistencia de los objetos

La persistencia de los objetos es un aspecto fundamental en el desarrollo de aplicaciones modernas, ya que permite almacenar y recuperar información de manera eficiente y segura. En este capítulo se exploran los conceptos y mecanismos relacionados con la persistencia, haciendo especial énfasis en las bases de datos orientadas a objetos. Se analizarán sus características principales, los pasos para su instalación y configuración, así como las técnicas para la creación, consulta y manipulación de datos. Además, se abordarán los distintos tipos de datos y colecciones que pueden gestionarse en estos sistemas, proporcionando una visión integral sobre el mantenimiento de la persistencia en aplicaciones orientadas a objetos. Los temas a tratar incluyen:

1. **Bases de datos orientadas a objetos:** En este apartado se explicará qué son las bases de datos orientadas a objetos, sus ventajas frente a otros modelos y su relevancia en aplicaciones modernas.

2. **Características de las bases de datos orientadas a objetos:** Se detallarán las principales propiedades de este tipo de bases de datos, como la encapsulación, herencia y polimorfismo.

3. **Instalación del gestor de bases de datos:** Se describirán los pasos necesarios para instalar y configurar un gestor de bases de datos orientado a objetos, incluyendo requisitos y recomendaciones.

4. **Creación de bases de datos:** Se explicará el proceso para crear una base de datos orientada a objetos, definiendo las estructuras y objetos necesarios.

5. **Tipos de datos objeto; atributos y métodos:** Se analizarán los distintos tipos de datos objeto, así como la definición de atributos y métodos asociados.

6. **Tipos de datos colección:** Se describirán los tipos de datos colección disponibles en bases de datos orientadas a objetos y su utilidad para gestionar conjuntos de objetos.

7. **Mecanismos de consulta:** Se abordarán las diferentes formas de consultar información en bases de datos orientadas a objetos, destacando las ventajas de cada método.

8. **El lenguaje de consultas: sintaxis, expresiones, operadores:** Se presentará el lenguaje utilizado para realizar consultas, explicando su sintaxis, expresiones y operadores disponibles.

9. **Recuperación, modificación y borrado de información:** Se explicarán las técnicas para recuperar, modificar y eliminar datos almacenados en la base de datos.

14.1. Bases de datos orientadas a objetos

Las bases de datos orientadas a objetos permiten almacenar información en forma de objetos, manteniendo las características de la programación orientada a objetos como encapsulación, herencia y polimorfismo. Aunque existen gestores específicos, en la práctica se utilizan *frameworks* como Hibernate para mapear objetos Java a tablas en bases de datos relacionales como MySQL.

14.2. Características de las bases de datos orientadas a objetos

Las bases de datos orientadas a objetos (*OODB*) se diseñan para almacenar información en forma de objetos, tal como se representan en lenguajes de programación orientados a objetos. A diferencia de las bases de datos relacionales, donde los datos se organizan en tablas y filas, en una *OODB* los datos se almacenan como instancias de clases, permitiendo una correspondencia directa entre el modelo de datos y el modelo de programación.

Entre las ventajas de las bases de datos orientadas a objetos destacan:

- **Persistencia transparente:** Los objetos pueden ser almacenados y recuperados sin necesidad de convertirlos a otro formato, lo que simplifica el desarrollo y mantenimiento del software.

- **Soporte para herencia y polimorfismo:** Las *OODB* permiten que las relaciones de herencia entre clases se reflejen directamente en la base de datos, facilitando la reutilización y extensión de modelos de datos.

- **Encapsulamiento:** Los datos y los métodos asociados a los objetos se almacenan juntos, manteniendo la integridad y coherencia del modelo de datos.

- **Gestión de relaciones complejas:** Las referencias entre objetos pueden representarse de forma natural, permitiendo la modelización de estructuras de datos complejas como grafos, árboles y colecciones anidadas.

Algunos ejemplos de sistemas de gestión de bases de datos orientadas a objetos son db4o, ObjectDB y Versant. Sin embargo, en la práctica, muchas aplicaciones utilizan *frameworks* de mapeo objeto-relacional (*ORM*) como *Hibernate*, que permiten trabajar con objetos en el código y almacenarlos en bases de datos relacionales tradicionales. Estos *frameworks* proporcionan una capa de abstracción que traduce las operaciones sobre objetos en consultas SQL, facilitando la persistencia sin perder las ventajas de la programación orientada a objetos.

La elección entre una base de datos orientada a objetos y una relacional depende de los requisitos del proyecto, la complejidad del modelo de datos y la integración con otras tecnologías. En aplicaciones donde el modelo de datos es altamente jerárquico o requiere relaciones complejas entre objetos, una *OODB* puede ofrecer ventajas significativas en términos de rendimiento y facilidad de desarrollo.

14.3. Instalación del gestor de bases de datos

En esta sección se abordará el proceso de instalación y configuración de los sistemas necesarios para gestionar la persistencia de objetos en aplicaciones Java. Aunque existen gestores de bases de datos orientadas a objetos como *db4o* y *ObjectDB*, en este capítulo se opta por utilizar *Hibernate* junto con *MySQL*. Esta elección se debe a que *Hibernate* es un *framework* ampliamente utilizado en la industria para el mapeo objeto-relacional (*ORM*), lo que permite trabajar con objetos Java y almacenarlos en bases de datos relacionales de manera eficiente y transparente. *MySQL*, por su parte, es uno de los sistemas de gestión de bases de datos más populares y robustos, con gran soporte y documentación.

El uso combinado de *Hibernate* y *MySQL* facilita el aprendizaje de técnicas modernas de persistencia, proporciona mayor flexibilidad y asegura compatibilidad con proyectos reales. A continuación se describen los pasos generales para instalar y configurar estos sistemas, permitiendo implementar la persistencia de objetos en aplicaciones Java de forma práctica y profesional.

14.3.1. Instalación y configuración de *MySQL* e *Hibernate*

Para trabajar con persistencia en Java usando *Hibernate* y *MySQL*, primero se debe instalar *MySQL* y configurar el acceso desde Java.

1. **Instalar MySQL:** Descargue e instale MySQL desde `https://dev.mysql.com/downloads/installer/`.

2. **Crear una base de datos:**

```
CREATE DATABASE ejemplo_hibernate;
```

3. **Agregar dependencias de Hibernate y MySQL Connector en su proyecto Java (pom.xml):**

```
<dependency>
  <groupId>org.hibernate.orm</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>6.4.4.Final</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.33</version>
</dependency>

```

Al igual que en el capítulo anterior, es posible ejecutar el gestor de bases de datos en un contenedor Docker. Esto permite una instalación rápida y sencilla, sin necesidad de configurar el entorno localmente. Podrás levantar la base de datos MySQL en un contenedor Docker con el siguiente comando:

```
docker run --name mysql -e MYSQL_ROOT_PASSWORD=tu_contrasena -e ↪
  MYSQL_DATABASE=ejemplo_hibernate -p 3306:3306 -d mysql:latest
```

14.4. Creación de bases de datos

Para crear una base de datos en MySQL, puede utilizar la consola de comandos o una herramienta gráfica como MySQL Workbench. El siguiente comando crea una base de datos llamada `ejemplo_hibernate`:

```
CREATE DATABASE ejemplo_hibernate;
```

Una vez creada la base de datos, Hibernate se encargará de generar las tablas necesarias a partir de las entidades Java definidas en el proyecto. Esto se logra mediante la configuración de la propiedad `hibernate.hbm2ddl.auto` en el archivo `persistence.xml`, que puede tomar valores como `update`, `create` o `validate`.

Por ejemplo, con la siguiente configuración:

```
<property name="hibernate.hbm2ddl.auto">update</property>
```

Hibernate actualizará automáticamente la estructura de la base de datos para reflejar los cambios en las entidades Java cada vez que la aplicación se inicie.

14.5. Tipos de datos objeto, atributos y métodos

En las bases de datos orientadas a objetos y en frameworks ORM como Hibernate, los tipos de datos objeto corresponden a las clases definidas en el lenguaje de programación. Cada clase representa una entidad y sus atributos se mapean a columnas de la base de datos. Los métodos permiten encapsular la lógica de negocio asociada a la entidad.

Por ejemplo, considere la siguiente entidad Java que representa un producto:

```
@Entity
@Table(name = "producto")
public class Producto {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String nombre;
  private double precio;

  // Método para aplicar descuento
  public void aplicarDescuento(double porcentaje) {
    this.precio = this.precio * (1 - porcentaje / 100);
  }

  // Getters y setters
}
```

En este ejemplo, `Producto` es un tipo de dato objeto, con atributos (`id`, `nombre`, `precio`) y métodos (`aplicarDescuento`). *Hibernate* se encarga de mapear estos atributos a columnas en la tabla `producto` de la base de datos, y los métodos permiten manipular los datos de la entidad de forma encapsulada.

14.6. Tipos de datos colección

En las bases de datos orientadas a objetos y en frameworks ORM como Hibernate, los tipos de datos colección permiten almacenar conjuntos de objetos relacionados, como listas, conjuntos o mapas. Estos tipos de colección son útiles para modelar relaciones uno-a-muchos o muchos-a-muchos entre entidades.

Hibernate soporta varias colecciones estándar de Java, como `List`, `Set` y `Map`. Para mapear una colección en una entidad, se utilizan anotaciones como `@OneToMany`, `@ManyToMany` y `@ElementCollection`.

Por ejemplo, considere una entidad `Categoria` que contiene una colección de productos:

```
@Entity
@Table(name = "categoria")
public class Categoria {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String nombre;

  @OneToMany(mappedBy = "categoria", cascade = CascadeType.ALL)
  private List<Producto> productos = new ArrayList<>();

  // Getters y setters
}
```

En este ejemplo, la entidad `Categoria` tiene una colección de objetos `Producto`. La anotación `@OneToMany` indica que una categoría puede tener muchos productos asociados. *Hibernate* se encarga de gestionar la relación y la persistencia de los objetos en la base de datos.

Las colecciones permiten modelar relaciones complejas y facilitan la gestión de conjuntos de datos relacionados dentro de las aplicaciones orientadas a objetos.

14.7. Mecanismos de consulta

En los sistemas de persistencia orientados a objetos y *frameworks ORM* como *Hibernate*, existen diversos mecanismos para consultar información almacenada en la base de datos. Los más comunes son:

- **Consultas mediante el API de Hibernate:** Permite recuperar objetos utilizando métodos como `find`, `get`, o `createQuery`.
- **Lenguaje de consultas orientado a objetos (HQL):** Hibernate Query Language es similar a SQL pero opera sobre entidades y sus atributos, no

sobre tablas.

- **Consultas nativas SQL:** Es posible ejecutar directamente sentencias SQL sobre la base de datos.
- **Criteria API:** Permite construir consultas de manera programática y segura, útil para consultas dinámicas.

A continuación se presentan ejemplos de cada mecanismo.

14.7.1. Consultas mediante el API de Hibernate

El API de Hibernate permite recuperar objetos directamente usando métodos como `find` y `get` del `EntityManager` o `Session`. Por ejemplo:

```
Producto producto = entityManager.find(Producto.class, 1L);
// O usando Session de Hibernate
Producto producto2 = session.get(Producto.class, 2L);
```

Estos métodos devuelven la instancia del objeto correspondiente al identificador proporcionado, facilitando la recuperación de datos sin necesidad de escribir consultas SQL explícitas.

14.7.2. Lenguaje de consultas orientado a objetos (HQL)

Hibernate Query Language (HQL) es un lenguaje similar a SQL, pero opera sobre entidades y sus atributos. Por ejemplo, para obtener todos los productos con precio mayor a 100:

```
List<Producto> productos = entityManager.createQuery(
  "FROM Producto p WHERE p.precio > 100", Producto.class
).getResultList();
```

HQL permite realizar consultas complejas utilizando la estructura de objetos definida en el modelo.

Por ejemplo, para obtener todas las categorías que tienen al menos tres productos con precio mayor a 100, se puede utilizar HQL con una consulta agregada:

```
List<Categoria> categorias = entityManager.createQuery(
  "SELECT c FROM Categoria c JOIN c.productos p " +
  "WHERE p.precio > 100 GROUP BY c.id HAVING COUNT(p) >= 3",
  Categoria.class
).getResultList();
```

Esta consulta recupera las instancias de `Categoria` que cumplen con el criterio especificado, demostrando cómo HQL permite expresar consultas complejas sobre relaciones y atributos de entidades.

También podemos realizar consultas en las que es necesario pasar parámetros específicos. Por ejemplo, para obtener los productos cuyo nombre contiene una palabra específica utilizando parámetros en HQL:

```
String palabraClave = "laptop";
List<Producto> productos = entityManager.createQuery(
  "FROM Producto p WHERE p.nombre LIKE :clave", Producto.class
)
.setParameter("clave", "%" + palabraClave + "%")
.getResultList();
```

En este ejemplo, el parámetro `:clave` permite construir consultas dinámicas y seguras, evitando problemas de inyección de SQL.

14.7.3. Consultas nativas SQL

Hibernate también permite ejecutar consultas SQL nativas directamente sobre la base de datos. Por ejemplo:

```
List<Producto> productos = entityManager.createNativeQuery(
  "SELECT * FROM producto WHERE precio > 100", Producto.class
).getResultList();
```

Las consultas nativas son útiles cuando se requiere aprovechar funciones específicas del gestor de base de datos.

También es posible realizar consultas nativas con parámetros, y entre varias tablas. Por ejemplo, para obtener los productos de una categoría específica:

```
Long categoriaId = 1L;
List<Producto> productos = entityManager.createNativeQuery(
  "SELECT * FROM producto WHERE categoria_id = :categoriaId", ↪
  Producto.class
)
.setParameter("categoriaId", categoriaId)
.getResultList();
```

14.7.4. Criteria API

La Criteria API permite construir consultas de manera programática y segura, facilitando la creación de consultas dinámicas. Por ejemplo:

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Producto> cq = cb.createQuery(Producto.class);
Root<Producto> root = cq.from(Producto.class);
cq.select(root).where(cb.gt(root.get("precio"), 100));
List<Producto> productos = entityManager.createQuery(cq).↪
  getResultList();
```

Este enfoque es especialmente útil cuando los criterios de consulta se definen en tiempo de ejecución.

14.8. El lenguaje de consultas: sintaxis, expresiones, operadores

El lenguaje de consultas en Hibernate, conocido como HQL (Hibernate Query Language), es similar a SQL pero opera sobre entidades y sus atributos. La sintaxis de HQL permite realizar consultas complejas utilizando expresiones y operadores específicos.

A continuación se presentan ejemplos prácticos de sintaxis, expresiones y operadores en HQL:

Selección de entidades

Para obtener todos los productos:

```
List<Producto> productos = entityManager.createQuery(
  "FROM Producto", Producto.class
).getResultList();
```

Filtrado con operadores de comparación

Para obtener productos con precio mayor a 500:

```
List<Producto> productos = entityManager.createQuery(
  "FROM Producto p WHERE p.precio > 500", Producto.class
).getResultList();
```

Uso de operadores lógicos

Para obtener productos cuyo precio esté entre 100 y 1000:

```
List<Producto> productos = entityManager.createQuery(
  "FROM Producto p WHERE p.precio >= 100 AND p.precio <= 1000", ↪
  Producto.class
).getResultList();
```

Expresiones de texto

Para buscar productos cuyo nombre contenga la palabra “tablet”:

```
List<Producto> productos = entityManager.createQuery(
  "FROM Producto p WHERE p.nombre LIKE :clave", Producto.class
)
.setParameter("clave", "%tablet%")
.getResultList();
```

Ordenamiento de resultados

Para obtener productos ordenados por precio descendente:

```
List<Producto> productos = entityManager.createQuery(
  "FROM Producto p ORDER BY p.precio DESC", Producto.class
).getResultList();
```

Consultas agregadas

Para obtener el número total de productos:

```
Long total = entityManager.createQuery(
  "SELECT COUNT(p) FROM Producto p", Long.class
).getSingleResult();
```

Para obtener el precio promedio de los productos:

```
Double promedio = entityManager.createQuery(
  "SELECT AVG(p.precio) FROM Producto p", Double.class
).getSingleResult();
```

Consultas con relaciones

Para obtener los productos de una categoría específica:

```
List<Producto> productos = entityManager.createQuery(
  "FROM Producto p WHERE p.categoria.id = :categoriaId", ↪
  Producto.class
)
.setParameter("categoriaId", 2L)
.getResultList();
```

Uso de parámetros

Para obtener productos con precio menor a un valor dado:

```
Double limite = 300.0;
List<Producto> productos = entityManager.createQuery(
  "FROM Producto p WHERE p.precio < :limite", Producto.class
)
.setParameter("limite", limite)
.getResultList();
```

Estos ejemplos muestran cómo utilizar la sintaxis de HQL, expresiones y operadores para realizar consultas eficientes y seguras sobre entidades y sus relaciones en Hibernate.

14.9. Recuperación, modificación y borrado de información

En los sistemas de persistencia orientados a objetos y frameworks ORM como Hibernate, las operaciones básicas sobre los datos incluyen la recuperación (lectura), modificación (actualización) y borrado (eliminación) de información. A continuación se presentan ejemplos prácticos de cada operación utilizando el EntityManager de JPA/Hibernate.

Recuperación de información

Para recuperar una entidad por su identificador:

```
Producto producto = entityManager.find(Producto.class, 1L);
```

Para recuperar una lista de entidades que cumplen un criterio:

```
List<Producto> productos = entityManager.createQuery(
  "FROM Producto p WHERE p.precio > 100", Producto.class
).getResultList();
```

Modificación de información

Para modificar los atributos de una entidad, primero se recupera el objeto, se actualizan sus valores y se confirma la transacción:

```
entityManager.getTransaction().begin();
Producto producto = entityManager.find(Producto.class, 1L);
producto.setPrecio(250.0);
entityManager.getTransaction().commit();
```

También se pueden realizar actualizaciones masivas mediante HQL:

```
entityManager.getTransaction().begin();
int actualizados = entityManager.createQuery(
  "UPDATE Producto p SET p.precio = p.precio * 0.9 WHERE p.↪
  precio > 500"
).executeUpdate();
entityManager.getTransaction().commit();
```

Borrado de información

Para eliminar una entidad específica:

```
entityManager.getTransaction().begin();
Producto producto = entityManager.find(Producto.class, 2L);
entityManager.remove(producto);
entityManager.getTransaction().commit();
```

Para eliminar varias entidades que cumplen un criterio:

```
entityManager.getTransaction().begin();
int eliminados = entityManager.createQuery(
  "DELETE FROM Producto p WHERE p.precio < 50"
).executeUpdate();
entityManager.getTransaction().commit();
```

Estas operaciones permiten gestionar de manera eficiente el ciclo de vida de los objetos persistentes en la base de datos, asegurando la integridad y coherencia de la información almacenada.

14.9.1. El concepto de repositorio

El patrón repositorio es una abstracción que encapsula la lógica de acceso a datos, proporcionando una interfaz clara para realizar operaciones de persistencia sobre entidades. En aplicaciones Java con *Hibernate*, los repositorios suelen implementarse como clases que utilizan el `EntityManager` para interactuar con la base de datos.

A continuación se muestra un ejemplo básico de un repositorio para la entidad `Producto`:

```
public class ProductoRepository {

  private EntityManager entityManager;

  public ProductoRepository(EntityManager entityManager) {
    this.entityManager = entityManager;
  }

  public Producto findById(Long id) {
    return entityManager.find(Producto.class, id);
  }

  public List<Producto> findAll() {
    return entityManager.createQuery("FROM Producto", Producto.↪
  class)
                .getResultList();
  }

  public void save(Producto producto) {
    entityManager.getTransaction().begin();
    entityManager.persist(producto);
    entityManager.getTransaction().commit();
  }

  public void update(Producto producto) {
    entityManager.getTransaction().begin();
```

```
26      entityManager.merge(producto);
27      entityManager.getTransaction().commit();
28    }
29
30    public void delete(Long id) {
31      entityManager.getTransaction().begin();
32      Producto producto = entityManager.find(Producto.class, id);
33      if (producto != null) {
34        entityManager.remove(producto);
35      }
36      entityManager.getTransaction().commit();
37    }
38  }
```

Este repositorio proporciona métodos para recuperar, guardar, actualizar y eliminar productos. El uso de repositorios facilita la organización del código, promueve la reutilización y permite separar la lógica de acceso a datos de la lógica de negocio.

En proyectos más grandes, es común definir interfaces para los repositorios y utilizar *frameworks* como *Spring Data JPA*, que generan automáticamente la implementación de los métodos básicos de persistencia. Por ejemplo:

```
1  public interface ProductoRepository extends JpaRepository<↪
     Producto, Long> {
2    List<Producto> findByPrecioGreaterThan(Double precio);
3  }
```

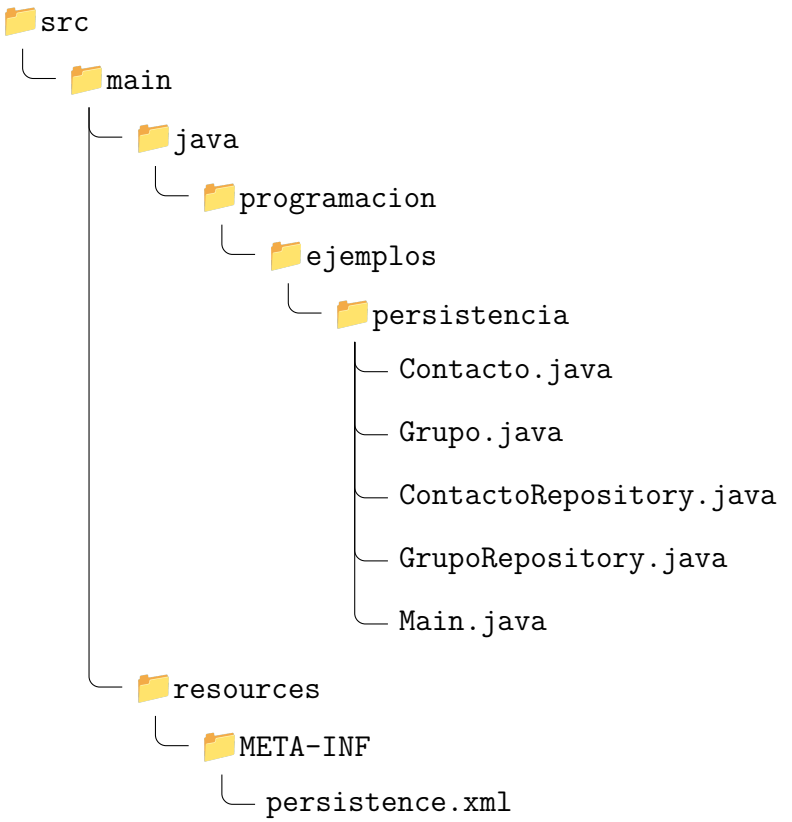
Con esta interfaz, Spring Data JPA proporciona automáticamente métodos para buscar productos por precio, sin necesidad de escribir la implementación manualmente. Además, permite personalizar consultas utilizando la convención de nombres, lo que simplifica aún más el acceso a datos.

El patrón repositorio es fundamental para mantener un diseño limpio y escalable en aplicaciones que utilizan Hibernate y JPA para la persistencia de objetos.

14.10. Ejemplo práctico: gestión de contactos

En esta sección se desarrollará una aplicación Java sencilla para la gestión de contactos, utilizando Hibernate y MySQL como sistema de persistencia. El ejemplo incluirá dos entidades relacionadas: `Contacto` y `Grupo`. Cada contacto pertenece a un grupo, y cada grupo puede tener varios contactos. Este ejemplo muestra cómo definir entidades relacionadas, configurar Hibernate, implementar repositorios y realizar operaciones básicas de persistencia en una aplicación Java para la gestión de contactos.

Estructura de archivos



Definición de entidades

```java
package programacion.ejemplos.persistencia;

import jakarta.persistence.*;

@Entity
@Table(name = "contacto")
public class Contacto {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String nombre;
  private String email;
  private String telefono;

  @ManyToOne
  @JoinColumn(name = "grupo_id")
```

```
  private Grupo grupo;

  // Getters y setters
}
```

```
package programacion.ejemplos.persistencia;

import jakarta.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "grupo")
public class Grupo {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String nombre;

  @OneToMany(mappedBy = "grupo", cascade = CascadeType.ALL)
  private List<Contacto> contactos = new ArrayList<>();

  // Getters y setters
}
```

Configuración de Hibernate

Cree el archivo `persistence.xml` con la configuración de conexión y mapeo de entidades:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
       version="2.1">
  <persistence-unit name="ejemplo_hibernate" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>programacion.ejemplos.persistencia.Contacto</class>
    <class>programacion.ejemplos.persistencia.Grupo</class>
    <properties>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/ejemplo_hibernate"/>
```

```
      <property name="javax.persistence.jdbc.user" value="root"/>
    >
      <property name="javax.persistence.jdbc.password" value="
    tu_contrasena"/>
      <property name="hibernate.dialect" value="org.hibernate.
    dialect.MySQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Repositorios para acceso a datos

```
package programacion.ejemplos.persistencia;

import jakarta.persistence.*;
import java.util.List;

public class ContactoRepository {
  private EntityManager entityManager;

  public ContactoRepository(EntityManager em) {
    this.entityManager = em;
  }

  public void save(Contacto contacto) {
    entityManager.getTransaction().begin();
    entityManager.persist(contacto);
    entityManager.getTransaction().commit();
  }

  public Contacto findById(Long id) {
    return entityManager.find(Contacto.class, id);
  }

  public List<Contacto> findAll() {
    return entityManager.createQuery("FROM Contacto", Contacto.
    class).getResultList();
  }

  public List<Contacto> findByGrupo(Long grupoId) {
    return entityManager.createQuery(
      "FROM Contacto c WHERE c.grupo.id = :grupoId", Contacto.
    class)
      .setParameter("grupoId", grupoId)
      .getResultList();
  }
```

```
}
```

```
package programacion.ejemplos.persistencia;

import jakarta.persistence.*;
import java.util.List;

public class GrupoRepository {
  private EntityManager entityManager;

  public GrupoRepository(EntityManager em) {
    this.entityManager = em;
  }

  public void save(Grupo grupo) {
    entityManager.getTransaction().begin();
    entityManager.persist(grupo);
    entityManager.getTransaction().commit();
  }

  public Grupo findById(Long id) {
    return entityManager.find(Grupo.class, id);
  }

  public List<Grupo> findAll() {
    return entityManager.createQuery("FROM Grupo", Grupo.class).getResultList();
  }
}
```

Clase principal para pruebas

```
package programacion.ejemplos.persistencia;

import jakarta.persistence.*;
import java.util.List;

public class Main {
  public static void main(String[] args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("ejemplo_hibernate");
    EntityManager em = emf.createEntityManager();

    GrupoRepository grupoRepo = new GrupoRepository(em);
    ContactoRepository contactoRepo = new ContactoRepository(em);

    // Crear grupo
    Grupo amigos = new Grupo();
```

```
16      amigos.setNombre("Amigos");
17      grupoRepo.save(amigos);
18
19      // Crear contactos
20      Contacto c1 = new Contacto();
21      c1.setNombre("Ana Pérez");
22      c1.setEmail("ana@email.com");
23      c1.setTelefono("6********");
24      c1.setGrupo(amigos);
25
26      Contacto c2 = new Contacto();
27      c2.setNombre("Luis Gómez");
28      c2.setEmail("luis@email.com");
29      c2.setTelefono("9********");
30      c2.setGrupo(amigos);
31
32      contactoRepo.save(c1);
33      contactoRepo.save(c2);
34
35      // Consultar contactos por grupo
36      List<Contacto> contactosAmigos = contactoRepo.findByGrupo(↪
    amigos.getId());
37      for (Contacto c : contactosAmigos) {
38        System.out.println(c.getNombre() + " - " + c.getEmail());
39      }
40
41      em.close();
42      emf.close();
43    }
44  }
```

Ejercicio 14.1

Cree una aplicación Java que implemente un sistema de gestión de tareas utilizando Hibernate y MySQL. El sistema debe permitir crear, actualizar, eliminar y listar tareas, cada una con un título, descripción, estado (pendiente, en progreso, completada) y fecha de creación. Además, implementa una funcionalidad para filtrar tareas por estado. El sistema debe incluir las siguientes entidades:

- `Tarea`: Representa una tarea con atributos como `titulo`, `descripcion`, `estado` y `fechaCreacion`.
- `Usuario`: Representa un usuario que puede tener varias tareas asignadas.

14.11. Resumen

En este capítulo se ha explorado el concepto de persistencia de objetos en aplicaciones Java, destacando la importancia de almacenar y recuperar información de manera eficiente. Se han presentado las bases de datos orientadas a objetos y sus principales características, así como el uso de *frameworks ORM* como *Hibernate* junto con *MySQL* para implementar la persistencia. Se han detallado los pasos de instalación, la creación de bases de datos, el mapeo de entidades y colecciones, y los mecanismos de consulta mediante *HQL*, SQL nativo y *Criteria API*. Además, se han explicado las operaciones básicas de recuperación, modificación y borrado de datos, y el patrón repositorio para organizar el acceso a datos. Finalmente, se ha desarrollado un ejemplo práctico de gestión de contactos y se ha propuesto un ejercicio para implementar un sistema de gestión de tareas, consolidando los conocimientos adquiridos sobre persistencia en aplicaciones orientadas a objetos.

Cuadro 14.1: Elementos del lenguaje HQL

Elemento HQL	Descripción y Ejemplo
`FROM`	Selecciona entidades. Ejemplo: `FROM Producto`
`WHERE`	Filtra resultados. Ejemplo: `FROM Producto p WHERE p.precio >100`
`ORDER BY`	Ordena resultados. Ejemplo: `FROM Producto p ORDER BY p.precio DESC`
`SELECT`	Selecciona atributos específicos. Ejemplo: `SELECT p.nombre FROM Producto p`
`JOIN`	Une entidades relacionadas. Ejemplo: `FROM Categoria c JOIN c.productos p`
`GROUP BY`	Agrupa resultados. Ejemplo: `SELECT c FROM Categoria c JOIN c.productos p GROUP BY c.id`
`HAVING`	Filtra grupos. Ejemplo: `HAVING COUNT(p) >3`
`COUNT, AVG, SUM, MIN, MAX`	Funciones agregadas. Ejemplo: `SELECT COUNT(p) FROM Producto p`
`UPDATE`	Actualiza entidades. Ejemplo: `UPDATE Producto p SET p.precio = p.precio * 0.9 WHERE p.precio >500`
`DELETE`	Elimina entidades. Ejemplo: `DELETE FROM Producto p WHERE p.precio <50`
`LIKE`	Búsqueda por patrón. Ejemplo: `FROM Producto p WHERE p.nombre LIKE :clave`
`IN`	Filtra por lista de valores. Ejemplo: `FROM Producto p WHERE p.estado IN ('pendiente', 'completada')`
`BETWEEN`	Filtra por rango. Ejemplo: `FROM Producto p WHERE p.precio BETWEEN 100 AND 500`
`IS NULL, IS NOT NULL`	Filtra valores nulos. Ejemplo: `FROM Producto p WHERE p.descripcion IS NULL`
`DISTINCT`	Elimina duplicados. Ejemplo: `SELECT DISTINCT p.estado FROM Producto p`
`SET PARAMETER`	Parámetros en consultas. Ejemplo: `.setParameter("clave", "%tablet%")`

15

Interfaces gráficas

En este capítulo exploraremos el desarrollo de interfaces gráficas de usuario (GUI) en Java. Aprenderás cómo crear aplicaciones visuales que permiten la interacción mediante ventanas, botones y otros componentes gráficos. Veremos los conceptos fundamentales de eventos y controladores, y construiremos ejemplos prácticos utilizando la biblioteca Swing. Al finalizar, tendrás las bases para diseñar programas interactivos y atractivos. Los temas a tratar incluyen:

1. **Interfaces gráficas:** Introducción a las interfaces gráficas de usuario (GUI), su importancia y los principales componentes visuales que permiten la interacción con el usuario. Se explicarán las bibliotecas disponibles en Java para crear GUIs, como AWT, Swing y JavaFX.

2. **Concepto de evento:** Explicación de qué son los eventos en una interfaz gráfica, cómo se generan y cómo permiten que la aplicación responda a las acciones del usuario, como clics de botones, movimientos del ratón y entradas de teclado.

3. **Creación de controladores de eventos:** Detalle sobre cómo implementar y asociar controladores (listeners) a los componentes gráficos para gestionar eventos. Se mostrarán ejemplos prácticos de cómo detectar y responder a diferentes tipos de eventos en Java.

15.1. Interfaces gráficas

Una interfaz gráfica de usuario (GUI, por sus siglas en inglés) permite a los usuarios interactuar con una aplicación mediante elementos visuales como ventanas, botones, cuadros de texto y menús. En Java, las interfaces gráficas se pueden crear utilizando bibliotecas como `AWT`, `Swing` y `JavaFX`.

Importante

Swing es la biblioteca más utilizada para crear interfaces gráficas en Java estándar. JavaFX es más moderna y recomendada para aplicaciones nuevas, pero Swing sigue siendo ampliamente usada en educación y proyectos existentes.

Además de Swing, existen otras bibliotecas para crear interfaces gráficas en Java:

- **AWT (Abstract Window Toolkit):** Fue la primera biblioteca gráfica de Java. Proporciona componentes básicos y depende del sistema operativo para renderizar los elementos, lo que puede causar diferencias visuales entre plataformas. Actualmente se usa principalmente para aplicaciones sencillas o compatibilidad con código antiguo.
- **JavaFX:** Es una biblioteca moderna para crear interfaces gráficas avanzadas, con soporte para gráficos 2D y 3D, animaciones, efectos visuales y estilos CSS. Permite desarrollar aplicaciones más atractivas y flexibles, y es la opción recomendada para proyectos nuevos.
- **SWT (Standard Widget Toolkit):** Utilizada principalmente en el entorno Eclipse, SWT ofrece acceso directo a los componentes nativos del sistema operativo, logrando una apariencia más integrada. Es menos común fuera de aplicaciones específicas.

Cada biblioteca tiene sus ventajas y desventajas. Swing es ampliamente usada y fácil de aprender, JavaFX ofrece más funcionalidades modernas, y AWT/SWT pueden ser útiles en casos particulares.

15.1.1. Ejemplo básico de interfaz gráfica con Swing

El siguiente ejemplo muestra cómo crear una ventana simple con un botón usando Swing:

```
1  import javax.swing.*;
2
3  public class VentanaBasica {
4    public static void main(String[] args) {
5      JFrame ventana = new JFrame("Mi primera ventana");
6      JButton boton = new JButton("Haz clic");
7      ventana.add(boton);
8      ventana.setSize(300, 200);
9      ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10     ventana.setVisible(true);
```

```
  }
}
```

Ejemplo 15.1: Ventana básica con Swing

Este programa crea una ventana con un botón. Al ejecutar el código, aparecerá una ventana titulada “Mi primera ventana”.

15.2. Concepto de evento

Un evento es una acción que ocurre en la interfaz gráfica, como hacer clic en un botón, mover el ratón o escribir en un cuadro de texto. Los eventos permiten que la aplicación responda a las acciones del usuario.

Consejo

En Java, los eventos se gestionan mediante *listeners* (escuchadores), que son objetos que detectan y responden a eventos específicos.

15.3. Creación de controladores de eventos

Para responder a eventos, se deben crear controladores (*listeners*) y asociarlos a los componentes gráficos. El siguiente ejemplo muestra cómo detectar un clic en un botón:

```
import javax.swing.*;
import java.awt.event.*;

public class EventoBoton {
  public static void main(String[] args) {
    JFrame ventana = new JFrame("Evento de botón");
    JButton boton = new JButton("Haz clic");
    JLabel etiqueta = new JLabel("Esperando clic...");

    boton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        etiqueta.setText("¡Botón pulsado!");
      }
    });

    ventana.setLayout(new java.awt.FlowLayout());
    ventana.add(boton);
    ventana.add(etiqueta);
    ventana.setSize(300, 100);
    ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    ventana.setVisible(true);
```

```
22    }
23 }
```

Ejemplo 15.2: Controlador de eventos para un botón

En este ejemplo, al pulsar el botón, la etiqueta cambia su texto.

Cuadro 15.1: Componentes gráficos comunes en Swing

Componente	Descripción
`JButton`	Botón. Permite al usuario realizar una acción al hacer clic.
`JLabel`	Etiqueta de texto. Muestra información estática o dinámica.
`JTextField`	Campo de texto. Permite la entrada de texto en una sola línea.
`JTextArea`	Área de texto. Permite la entrada y visualización de texto en varias líneas.
`JCheckBox`	Casilla de verificación. Permite seleccionar o deseleccionar una opción.
`JRadioButton`	Botón de opción. Permite seleccionar una opción dentro de un grupo.
`JComboBox`	Lista desplegable. Permite seleccionar una opción de una lista.
`JPanel`	Panel contenedor. Agrupa y organiza otros componentes.
`JFrame`	Ventana principal. Contenedor de nivel superior para la interfaz.

15.3.1. Ejemplo práctico: Calculadora simple

A continuación se presenta un ejemplo de una calculadora simple implementada con Swing. Esta aplicación permite al usuario ingresar dos números en campos de texto, pulsar un botón para sumarlos y ver el resultado en una etiqueta. El programa también gestiona errores si los valores ingresados no son números válidos.

```
1 import javax.swing.*;
2 import java.awt.event.*;
3
4 public class Calculadora {
5   public static void main(String[] args) {
6     JFrame ventana = new JFrame("Calculadora");
```

```
7       JTextField campo1 = new JTextField(5);
8       JTextField campo2 = new JTextField(5);
9       JButton boton = new JButton("Sumar");
10      JLabel resultado = new JLabel("Resultado: ");
11
12      boton.addActionListener(new ActionListener() {
13        public void actionPerformed(ActionEvent e) {
14          try {
15            int a = Integer.parseInt(campo1.getText());
16            int b = Integer.parseInt(campo2.getText());
17            resultado.setText("Resultado: " + (a + b));
18          } catch (NumberFormatException ex) {
19            resultado.setText("Error: ingresa números válidos");
20          }
21        }
22      });
23
24      ventana.setLayout(new java.awt.FlowLayout());
25      ventana.add(campo1);
26      ventana.add(campo2);
27      ventana.add(boton);
28      ventana.add(resultado);
29      ventana.setSize(350, 120);
30      ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31      ventana.setVisible(true);
32    }
33  }
```

Ejemplo 15.3: Calculadora simple en Swing

A continuación se explica paso a paso cómo funciona el ejemplo de la calculadora:

1. Se crea una ventana principal (`JFrame`) con el título “Calculadora”.

2. Se añaden dos campos de texto (`JTextField`) donde el usuario puede ingresar los números a sumar.

3. Se agrega un botón (`JButton`) con el texto “Sumar”.

4. Se coloca una etiqueta (`JLabel`) para mostrar el resultado de la suma o un mensaje de error.

5. Se asocia un controlador de eventos (`ActionListener`) al botón. Cuando el usuario hace clic en el botón:

   - El programa intenta leer los valores de los campos de texto y convertirlos a números enteros.

- Si ambos valores son válidos, se realiza la suma y se muestra el resultado en la etiqueta.
- Si alguno de los valores no es un número válido, se muestra un mensaje de error en la etiqueta.

6. Finalmente, se organiza la ventana con un diseño de flujo (`FlowLayout`), se añaden todos los componentes y se muestra la ventana al usuario.

Ejercicio 15.1

Modifica el ejemplo de la calculadora para que también permita restar, multiplicar y dividir los dos números. Añade botones para cada operación y muestra el resultado correspondiente en la etiqueta.

Resumen

En este capítulo aprendiste los conceptos básicos de las interfaces gráficas en Java, el manejo de eventos y la creación de controladores. Usando Swing, puedes construir aplicaciones interactivas y visuales. Practica creando tus propias ventanas y componentes para dominar la programación gráfica en Java.

C

Programación avanzada en Java

16

¿Cómo escribir código limpio?

En el desarrollo de software, escribir *Clean Code* (código limpio) es fundamental para garantizar la mantenibilidad, legibilidad y escalabilidad de los proyectos. El código limpio es fácil de entender, modificar y extender, lo que reduce la probabilidad de errores y facilita la colaboración entre desarrolladores.

16.1. Principios de Clean Code

El concepto de *Clean Code* va más allá de simplemente escribir código que funcione. Se trata de crear soluciones que sean fáciles de leer, entender y modificar por cualquier miembro del equipo, incluso mucho tiempo después de haber sido escritas. El código limpio minimiza la complejidad innecesaria, utiliza estructuras claras y evita ambigüedades. Además, fomenta la reutilización y la modularidad, lo que facilita la incorporación de nuevas funcionalidades y la corrección de errores.

Un código limpio suele tener las siguientes características:

- **Simplicidad:** Elimina todo lo innecesario y se enfoca en resolver el problema de la manera más directa posible.
- **Legibilidad:** Está escrito pensando en que otros desarrolladores lo leerán y lo mantendrán.
- **Consistencia:** Sigue convenciones y estilos definidos para todo el proyecto.
- **Testabilidad:** Facilita la escritura de pruebas automatizadas, lo que ayuda a garantizar su correcto funcionamiento.
- **Evolutividad:** Permite realizar cambios y mejoras sin afectar negativamente otras partes del sistema.

Adoptar una mentalidad de código limpio implica revisar y refactorizar constantemente, buscando siempre la mejor forma de expresar las ideas y soluciones en el código.

Algunos principios generales de Clean Code que puedes empezar a aplicar son:

- **Nombres significativos:** Utiliza nombres descriptivos para variables, funciones y clases, que reflejen claramente su propósito.
- **Funciones cortas y enfocadas:** Las funciones deben realizar una sola tarea y ser lo más breves posible.
- **Evita la duplicación:** El código duplicado dificulta el mantenimiento y aumenta el riesgo de errores.
- **Comentarios útiles:** Los comentarios deben explicar el *porqué* del código, no el *qué*, y evitar redundancias.
- **Formato consistente:** Mantén una estructura y estilo uniforme en todo el proyecto.
- **Manejo adecuado de errores:** Implementa mecanismos claros y robustos para el tratamiento de errores y excepciones.

16.2. Principios SOLID

Los principios SOLID son un conjunto de buenas prácticas orientadas a la programación orientada a objetos, que ayudan a crear sistemas flexibles y fáciles de mantener. A continuación se detallan cada uno de los principios.

16.2.1. Single Responsibility Principle (SRP)

Cada clase debe tener una única responsabilidad, es decir, debe estar encargada de una sola parte de la funcionalidad del programa. Esto facilita la comprensión y el mantenimiento del código, ya que los cambios en una responsabilidad no afectan otras partes del sistema.

Ejemplo: Una clase que gestiona la lógica de negocio no debería encargarse también de la persistencia de datos.

16.2.2. Open/Closed Principle (OCP)

Las entidades de software (clases, módulos, funciones) deben estar abiertas para extensión, pero cerradas para modificación. Esto significa que se debe poder agregar nueva funcionalidad sin alterar el código existente, generalmente mediante herencia o composición.

Ejemplo: Utilizar interfaces y clases abstractas para permitir la extensión sin modificar el código base.

16.2.3. Liskov Substitution Principle (LSP)

Las clases derivadas deben poder sustituir a sus clases base sin alterar el comportamiento esperado del programa. En otras palabras, los objetos de una subclase deben poder usarse en lugar de los de la clase base sin que el cliente note la diferencia.

Ejemplo: Si una función espera un objeto de tipo `Animal`, debería funcionar correctamente si se le pasa un objeto de tipo `Perro`, que hereda de `Animal`.

16.2.4. Interface Segregation Principle (ISP)

Los clientes no deben verse obligados a depender de interfaces que no utilizan. Es mejor tener varias interfaces específicas en lugar de una interfaz general con muchos métodos innecesarios.

Ejemplo: Separar una interfaz grande en varias interfaces pequeñas y especializadas.

16.2.5. Dependency Inversion Principle (DIP)

Los módulos de alto nivel no deben depender de módulos de bajo nivel, sino de abstracciones. Las abstracciones no deben depender de los detalles, sino los detalles de las abstracciones. Esto se logra mediante la inversión de dependencias, utilizando interfaces o clases abstractas.

Ejemplo: Inyectar dependencias a través de interfaces en lugar de instanciar clases concretas directamente.

16.3. *Code Calisthenics*

Los *Code Calisthenics* son ejercicios prácticos diseñados para mejorar la disciplina y la calidad del código que escribimos. Consisten en seguir reglas estrictas durante el desarrollo, con el objetivo de fomentar buenas prácticas y hábitos saludables en la programación. Algunas reglas comunes incluyen:

- Una sola instrucción por línea.
- Una sola variable declarada por línea.
- Métodos de máximo siete líneas.
- No utilizar más de dos niveles de indentación.
- No usar la palabra clave `else`.
- No utilizar números mágicos ni cadenas mágicas.

Estos ejercicios ayudan a identificar malas prácticas y a desarrollar un estilo de programación más limpio y estructurado. Aunque pueden parecer restrictivos, su objetivo es fomentar la reflexión sobre las decisiones de diseño y mejorar la calidad del código a largo plazo.

Ejercicio 16.1

Aplica estos principios en el siguiente código:

```java
public class MessClass {
  public static double convertirFarenheith(double x) {
    return (x - 32) * 5 / 9;
  }

  public static double convertirCelsius(double x) {
    return (x * 9 / 5) + 32;
  }

  public static int maxEnMatriz3(int[][][] matriz) {
    int max = Integer.MIN_VALUE;
    for (int[][] submatriz : matriz) {
      for (int[] fila : submatriz) {
        for (int elemento : fila) {
          if (elemento > max) {
            max = elemento;
          }
        }
      }
    }
    return max;
  }

  public static int mediana(int[] numeros) {
    Arrays.sort(numeros);
    if (numeros.length == 0) {
      throw new IllegalArgumentException("El array está vacío");
```

```
28      } else if (numeros.length % 2 == 1) {
29        return numeros[numeros.length / 2];
30      } else {
31        return (numeros[numeros.length / 2 - 1] + numeros[↪
    numeros.length / 2]) / 2;
32      }
33    }
34
35    public static double[] estadisticas(int[] numeros) {
36      if (numeros.length == 0) {
37        throw new IllegalArgumentException("El array está ↪
    vacío");
38      }
39      return new double[] {
40        Arrays.stream(numeros).average().orElse(0),
41        mediana(numeros),
42        maxEnMatriz3(new int[][][] { numeros })
43      };
44    }
45  }
46
```

Resumen

Aplicar los principios de *Clean Code* y SOLID permite desarrollar sistemas robustos, escalables y fáciles de mantener. Adoptar estas buenas prácticas mejora la calidad del software y facilita el trabajo en equipo.

Para profundizar en el tema de *Clean Code* y buenas prácticas de desarrollo, se recomiendan los siguientes libros:

- **Clean Code: A Handbook of Agile Software Craftsmanship** – Robert C. Martin. Un libro fundamental que explora los principios y técnicas para escribir código limpio y mantenible.
- **The Pragmatic Programmer: Your Journey to Mastery** – Andrew Hunt y David Thomas. Ofrece consejos prácticos y estrategias para mejorar la calidad y profesionalismo en el desarrollo de software.

17

Principios de programación funcional en programación orientada a objetos

La programación funcional es un paradigma que se centra en el uso de funciones como elementos fundamentales para la construcción de programas. A diferencia de otros enfoques, como la programación orientada a objetos, la programación funcional promueve la transparencia referencial y la inmutabilidad, lo que facilita la creación de código más predecible y fácil de mantener.

En este capítulo se explorarán los conceptos clave de la programación funcional y cómo pueden aplicarse dentro de la programación orientada a objetos. Se abordarán temas como el principio de transparencia referencial, la importancia de la inmutabilidad, el uso de métodos como elementos de primer orden, la definición y utilización de expresiones lambda e interfaces funcionales, así como el empleo de la clase `Optional` para manejar valores que pueden estar ausentes.

Estos conceptos permiten escribir código más expresivo, seguro y flexible, aprovechando las ventajas de ambos paradigmas para desarrollar aplicaciones robustas y modernas.

17.1. Transparencia referencial

La transparencia referencial es una característica de las funciones que garantiza que, para los mismos argumentos de entrada, siempre se obtiene el mismo resultado y no se producen efectos secundarios. Esto facilita la comprensión, el testeo y la depuración del código.

```
1 int suma(int a, int b) {
2   return a + b;
3 }
```

Ejemplo 17.1: Función con transparencia referencial

En el ejemplo anterior, la función `suma` es transparente referencial porque su resultado depende únicamente de los parámetros `a` y `b`, sin modificar ningún estado externo.

Por el contrario, una función que modifica variables externas o depende de ellas pierde esta propiedad:

```
private static int contador = 0;

int mediana(List<Integer> valores) {
  contador++;
  Collections.sort(valores);
  int n = valores.size();
  if (n % 2 == 0) {
    return (valores.get(n/2 - 1) + valores.get(n/2)) / 2;
  } else {
    return valores.get(n/2);
  }
}
```

Ejemplo 17.2: Función sin transparencia referencial

En este caso, la función `mediana` modifica el estado global (`contador`) y también altera la lista recibida como argumento, lo que afecta la transparencia referencial. El resultado puede variar si la lista cambia fuera de la función o si el estado global se utiliza en otros lugares.

17.2. Inmutabilidad

La inmutabilidad implica que los objetos no pueden cambiar su estado después de ser creados. Esto reduce errores y facilita la concurrencia. Para facilitar esto, podemos hacer uso de clases inmutables, que no permiten modificar sus atributos una vez instanciados, en particular, usando `record`.

```
public record Punto(int x, int y) { }

Punto p1 = new Punto(3, 5);
// No se pueden modificar los valores de x e y después de crear ↪
   el objeto
```

Ejemplo 17.3: Ejemplo de clase inmutable usando record

Si queremos modificar una objeto de esa clase, lo que hacemos es crear un nuevo objeto con los valores modificados:

```
Punto p2 = new Punto(p1.x() + 1, p1.y());
// p2 es un nuevo objeto, p1 permanece sin cambios
```

Ejemplo 17.4: Creación de un nuevo objeto inmutable

En este ejemplo, podemos simplificar el código, si creamos un método en la clase `Punto` que nos permita crear un nuevo objeto con un valor modificado:

```
public record Punto(int x, int y) {
  public Punto mover(int dx, int dy) {
    return new Punto(x + dx, y + dy);
  }
}

Punto p1 = new Punto(3, 5);
Punto p2 = p1.mover(1, 0);
// p2 es un nuevo objeto, p1 permanece sin cambios
```

Ejemplo 17.5: Método para crear un nuevo objeto inmutable

Importante

En Java es fácil romper el principio de inmutabilidad, especialmente cuando se utilizan colecciones mutables, o se pasan arrays como argumentos. Por ejemplo, si una clase inmutable contiene una lista mutable, es posible modificar el contenido de la lista aunque el objeto en sí no cambie. Para evitar esto, se recomienda utilizar colecciones inmutables (como las de la clase `List.of(...)` en Java) o realizar copias defensivas de los datos en el constructor y los métodos de acceso.

17.3. Métodos como elementos de primer orden

En Java, los métodos pueden ser tratados como elementos de primer orden usando referencias a métodos y expresiones lambda.

```
List<String> nombres = Arrays.asList("Ana", "Luis", "Pedro");
nombres.forEach(System.out::println);
```

Ejemplo 17.6: Referencia a método

También podemos utilizar referencias a métodos de instancia (de objeto) y de clase (estáticos). Esto permite pasar métodos existentes como argumentos a funciones que esperan una interfaz funcional.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4);
numeros.forEach(System.out::println); // System.out es un objeto
  , println es su método
```

Ejemplo 17.7: Referencia a método de clase

En este ejemplo, `System.out::println` es una referencia a un método de objeto.

Para métodos de clase (estáticos), la sintaxis es similar:

```java
class Utilidades {
  public static void mostrar(String texto) {
    System.out.println(texto);
  }
}

List<String> mensajes = List.of("Hola", "Adiós");
mensajes.forEach(Utilidades::mostrar); // Utilidades::mostrar es referencia a método de clase
```

Ejemplo 17.8: Referencia a método estático de clase

También podemos referenciar métodos de instancia de un objeto específico:

```java
class Persona {
  private final String nombre;
  public Persona(String nombre) { this.nombre = nombre; }
  public void saludar() { System.out.println("Hola, soy " + nombre); }
}

List<Persona> personas = List.of(new Persona("Ana"), new Persona("Luis"));
personas.forEach(Persona::saludar); // Llama saludar() en cada objeto Persona
```

Ejemplo 17.9: Referencia a método de instancia de objeto

Las referencias a métodos permiten escribir código más conciso y reutilizable, facilitando el uso de funciones como argumentos y mejorando la expresividad del lenguaje.

Consejo

Podemos pasar referencias a métodos de clases mutables si queremos acumular un estado interno. Por ejemplo, podríamos tener un acumulador que sume valores a medida que se procesan:

```java
import java.util.concurrent.atomic.AtomicInteger;
import java.util.stream.Stream;

AtomicInteger acumulador = new AtomicInteger(0);
Stream.of(1, 2, 3, 4, 5).forEach(acumulador::addAndGet);
// acumulador ahora contiene la suma de los valores del stream
System.out.println("Suma total: " + acumulador.get());

```

Ejemplo 17.10: Uso de AtomicInteger y Stream para acumular valores

17.4. Expresiones lambda e interfaces funcionales

Las expresiones lambda permiten definir funciones anónimas de manera concisa. Las interfaces funcionales son interfaces con un solo método abstracto. El uso de la anotación `@FunctionalInterface` ayuda a identificar estas interfaces, e impone una restricción de que solo habrá un único método no implementado.

```
@FunctionalInterface
interface Operacion {
  int aplicar(int a, int b);
}

Operacion suma = (a, b) -> a + b;
System.out.println(suma.aplicar(3, 4)); // Imprime 7
```

Ejemplo 17.11: Uso de lambda e interfaz funcional

Al crear estas interfaces, podremos ajustarnos a nuestras propias necesidades. Sin embargo, Java incluye las siguientes interfaces funcionales, las cuales son suficiente para la mayoría de problemas.

Importante

Las interfaces anteriores no declaran el lanzamiento de excepciones, por eso, el compilador fallará si se intenta lanzar una excepción dentro de su implementación. Para ello es necesario crear interfaces funcionales que declaren las excepciones, o adaptar el código de manera que el control de errores se realice de otra manera, por ejemplo usando la clase `Optional`.

17.5. La clase `Optional`

La clase `Optional` ayuda a manejar valores que pueden estar ausentes, evitando errores de referencia nula. Es especialmente útil en librerías que gestionan colecciones, o peticiones a servidores externos, ya que puede encapsular un posible valor no presente dentro de la clase `Optional`.

```
Optional<String> nombre = Optional.ofNullable(obtenerNombre());
nombre.ifPresent(n -> System.out.println("Nombre: " + n));
```

Ejemplo 17.12: Uso de Optional

Al usar esta clase, es posible reducir el número de comprobaciones de nulos y hacer el código más legible. Por ejemplo, en lugar de verificar si un objeto es nulo antes de usarlo, podemos usar métodos como `ifPresent`, `orElse` o `map` para manejar el valor de manera más fluida.

```java
// Este valor puede venir de una llamada a un método que devuelve Optional<String>
Optional<String> texto = Optional.of(" Hola mundo ");

String resultado = texto
  .map(String::trim)        // 1. Elimina espacios
  .filter(t -> t.startsWith("Hola"))// 2. Filtra si empieza por "Hola"
  .map(String::toUpperCase)     // 3. Convierte a mayúsculas
  .map(s -> s + "!")        // 4. Añade un signo de exclamación
  .orElse("Valor alternativo");   // 5. Devuelve valor alternativo si está ausente

System.out.println(resultado); // Imprime "HOLA MUNDO!"
```

Ejemplo 17.13: Ejemplo de llamadas encadenadas con Optional

Ejercicio 17.1

Toma uno de los ejercicios que has visto a lo largo de este libro, y reescríbelo haciendo uso de los principios de programación funcional. Usa la clase `Optional` para eliminar cualquier comprobación de nulos.

17.6. La clase `Stream`

La clase `Stream` es una herramienta fundamental en la programación funcional de Java, ya que permite procesar colecciones de datos de manera declarativa y eficiente. Un `Stream` representa una secuencia de elementos sobre la que se pueden realizar operaciones como filtrado, transformación, agrupamiento y reducción, sin modificar la colección original.

Las operaciones sobre `Stream` suelen ser encadenadas y pueden ser de dos tipos: intermedias (como `filter`, `map`, `sorted`) y terminales (como `collect`, `forEach`, `reduce`). Esto facilita la escritura de código conciso y expresivo.

```java
List<String> nombres = List.of("Ana", "Luis", "Pedro", "Lucía");
List<String> resultado = nombres.stream()
  .filter(n -> n.length() > 4)
  .map(String::toUpperCase)
  .sorted()
  .toList();

System.out.println(resultado); // Imprime [LUIS, LUCÍA, PEDRO]
```

Ejemplo 17.14: Uso básico de Stream

El uso de `Stream` junto con `Optional` es especialmente útil para manejar datos que pueden estar ausentes o para evitar errores de referencia nula. Por

ejemplo, al buscar un elemento en una colección, podemos obtener un `Optional` que indique si el elemento está presente:

```
List<String> nombres = List.of("Ana", "Luis", "Pedro", "Lucía");
Optional<String> encontrado = nombres.stream()
  .filter(n -> n.startsWith("L"))
  .findFirst();

encontrado.ifPresent(n -> System.out.println("Encontrado: " + n)↪
  );
```

Ejemplo 17.15: Stream y Optional para buscar elementos

En este ejemplo, `findFirst` devuelve un `Optional<String>` que estará vacío si no se encuentra ningún elemento que cumpla la condición. Así, evitamos comprobaciones explícitas de nulos y hacemos el código más seguro y legible.

Además, podemos combinar operaciones de `Stream` y `Optional` para transformar, filtrar y procesar datos de manera funcional:

```
List<String> nombres = List.of("Ana", "Luis", "Pedro", "Lucía");
String resultado = nombres.stream()
  .filter(n -> n.length() > 5)
  .findFirst()
  .map(String::toLowerCase)
  .orElse("No encontrado");

System.out.println(resultado); // Imprime "lucía" o "No ↪
  encontrado"
```

Ejemplo 17.16: Encadenamiento de Stream y Optional

El uso conjunto de `Stream` y `Optional` permite escribir código más robusto, evitando errores comunes y facilitando la programación funcional en Java.

Resumen

En este capítulo se han presentado los principios fundamentales de la programación funcional y su integración en la programación orientada a objetos. Se ha explicado la importancia de la transparencia referencial y la inmutabilidad para escribir código más seguro y predecible. Además, se ha mostrado cómo tratar métodos como elementos de primer orden mediante referencias y expresiones lambda, y el uso de interfaces funcionales para definir comportamientos flexibles. Finalmente, se ha destacado el papel de la clase `Optional` para gestionar valores ausentes de forma segura y expresiva. Estos conceptos permiten combinar lo mejor de ambos paradigmas y desarrollar aplicaciones modernas, robustas y fáciles de mantener.

Cuadro 17.1: Principales interfaces funcionales de Java y su uso

Interfaz funcional	**Descripción y uso principal**
`Predicate<T>`	Representa una función que recibe un argumento de tipo `T` y devuelve un valor booleano. Se usa para pruebas, filtros y condiciones.
`Consumer<T>`	Recibe un argumento de tipo `T` y no devuelve nada. Se utiliza para realizar acciones sobre objetos, como imprimir o modificar estructuras externas.
`Supplier<T>`	No recibe argumentos y devuelve un valor de tipo `T`. Se usa para generar o proveer valores, como fábricas o generadores.
`Function<T, R>`	Recibe un argumento de tipo `T` y devuelve un resultado de tipo `R`. Se emplea para transformar o mapear valores.
`UnaryOperator<T>`	Recibe y devuelve un valor del mismo tipo `T`. Es una especialización de `Function` para operaciones unarias.
`BinaryOperator<T>`	Recibe dos argumentos del mismo tipo `T` y devuelve un resultado del mismo tipo. Se usa para operaciones binarias, como suma o concatenación.
`BiFunction<T, U, R>`	Recibe dos argumentos de tipos `T` y `U`, y devuelve un resultado de tipo `R`. Se utiliza para combinar o transformar dos valores.
`BiConsumer<T, U>`	Recibe dos argumentos de tipos `T` y `U`, y no devuelve nada. Se emplea para realizar acciones sobre pares de valores.
`BiPredicate<T, U>`	Recibe dos argumentos de tipos `T` y `U`, y devuelve un valor booleano. Se usa para pruebas o condiciones sobre pares de valores.
`Runnable`	No recibe argumentos ni devuelve valores. Se utiliza para ejecutar código, especialmente en hilos o tareas asíncronas.

Cuadro 17.2: Principales métodos de la clase `Optional<T>`

Método	Descripción
`Optional.empty()`	Devuelve una instancia de `Optional` vacía, sin valor presente.
`Optional.of(T value)`	Devuelve un `Optional` que contiene el valor especificado, lanza excepción si el valor es nulo.
`Optional.ofNullable(T value)`	Devuelve un `Optional` que puede contener el valor especificado o estar vacío si es nulo.
`isPresent()`	Devuelve `true` si el valor está presente, `false` en caso contrario.
`ifPresent(Consumer<? super T> action)`	Ejecuta la acción dada si el valor está presente.
`get()`	Devuelve el valor si está presente, lanza excepción si está vacío.
`orElse(T other)`	Devuelve el valor si está presente, o el valor alternativo si está vacío.
`orElseGet(Supplier<? extends T> supplier)`	Devuelve el valor si está presente, o el resultado del proveedor si está vacío.
`orElseThrow()`	Devuelve el valor si está presente, o lanza `NoSuchElementException` si está vacío.
`orElseThrow(Supplier<? extends X> exceptionSupplier)`	Devuelve el valor si está presente, o lanza la excepción proporcionada si está vacío.
`map(Function<? super T,? extends U> mapper)`	Si hay valor, aplica la función y devuelve un `Optional` con el resultado; si está vacío, devuelve vacío.
`flatMap(Function<? super T, Optional<U>> mapper)`	Similar a `map`, pero la función debe devolver un `Optional`.

Método	Descripción
`filter(Predicate<? super T> ↪ predicate)`	Si hay valor y cumple el predicado, devuelve el mismo `Optional`; si no, devuelve vacío.

Cuadro 17.3: Principales métodos de la clase `Stream<T>`

Método	Descripción
`filter(Predicate<? super T> predicate)`	Devuelve un nuevo `Stream` con los elementos que cumplen el predicado.
`map(Function<? super T,? extends R> mapper)`	Transforma cada elemento usando la función dada y devuelve un `Stream` de los resultados.
`flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`	Aplica la función a cada elemento y aplana los resultados en un solo `Stream`.
`sorted()`	Devuelve un `Stream` con los elementos ordenados según el orden natural.
`sorted(Comparator<? super T> comparator)`	Devuelve un `Stream` con los elementos ordenados usando el comparador dado.
`distinct()`	Devuelve un `Stream` con elementos únicos, eliminando duplicados.
`limit(long maxSize)`	Devuelve un `Stream` con un máximo de `maxSize` elementos.
`skip(long n)`	Omite los primeros `n` elementos y devuelve el resto en un `Stream`.
`forEach(Consumer<? super T> action)`	Ejecuta la acción dada para cada elemento del `Stream`.
`collect(Collector<? super T, A, R> collector)`	Reduce el `Stream` a una colección o resultado usando el colector especificado.
`toList()`	Devuelve una lista con los elementos del `Stream`.
`reduce(BinaryOperator<T> accumulator)`	Reduce los elementos del `Stream` a un solo valor usando el acumulador.

Método	Descripción
`findFirst()`	Devuelve un `Optional` con el primer elemento del `Stream`, si existe.
`findAny()`	Devuelve un `Optional` con algún elemento del `Stream`, útil en procesamiento paralelo.
`anyMatch(Predicate<? super T> predicate)`	Devuelve `true` si algún elemento cumple el predicado.
`allMatch(Predicate<? super T> predicate)`	Devuelve `true` si todos los elementos cumplen el predicado.
`noneMatch(Predicate<? super T> predicate)`	Devuelve `true` si ningún elemento cumple el predicado.
`count()`	Devuelve el número de elementos en el `Stream`.

18

Patrones de diseño software

En este capítulo se estudian los patrones de diseño de software, que son soluciones reutilizables a problemas comunes en el desarrollo de aplicaciones. Los patrones de diseño ayudan a estructurar el código de manera eficiente, facilitando la mantenibilidad, la escalabilidad y la reutilización. Se dividen en tres grandes categorías: patrones creacionales, estructurales y de comportamiento. A lo largo del capítulo se explican los principales patrones de cada categoría, su propósito y ejemplos prácticos de implementación.

18.1. Patrones creacionales

Los patrones creacionales son aquellos que se centran en la manera de crear objetos en el software. Su objetivo principal es abstraer el proceso de instanciación, permitiendo que el sistema sea más flexible y desacoplado. Estos patrones ayudan a controlar cómo y cuándo se crean los objetos, facilitando la reutilización y la escalabilidad del código.

Entre los patrones creacionales más conocidos se encuentran:

- **Singleton**: Garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a ella.
- **Factory Method**: Define una interfaz para crear un objeto, pero permite que las subclases decidan qué clase instanciar.
- **Abstract Factory**: Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.
- **Builder**: Separa la construcción de un objeto complejo de su representación, permitiendo crear diferentes representaciones con el mismo proceso de construcción.
- **Prototype**: Permite copiar objetos existentes para crear nuevos objetos, evitando la creación desde cero.

Estos patrones son fundamentales para diseñar sistemas robustos y mantenibles, especialmente cuando se requiere flexibilidad en la creación de objetos.

18.1.1. El patrón *Abstract Factory*

El patrón *Abstract Factory* es un patrón creacional que permite crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. Este patrón proporciona una interfaz para crear objetos de diferentes tipos, pero deja en manos de las subclases la decisión de qué clase concreta instanciar. Es especialmente útil cuando el sistema debe ser independiente de cómo se crean, componen y representan los objetos.

Por ejemplo, supongamos que estamos desarrollando una interfaz gráfica que puede tener diferentes estilos visuales (por ejemplo, estilo clásico y estilo moderno). Cada estilo requiere un conjunto de componentes (botones, ventanas, menús) que deben ser compatibles entre sí. El patrón *Abstract Factory* permite definir una interfaz para crear estos componentes y luego implementar fábricas concretas para cada estilo.

A continuación se muestra un ejemplo de uso del patrón *Abstract Factory* en Java:

```
interface Button {
  void render();
}

interface Window {
  void open();
}

// Productos concretos
class ClassicButton implements Button {
  public void render() {
    System.out.println("Renderizando botón clásico");
  }
}

class ModernButton implements Button {
  public void render() {
    System.out.println("Renderizando botón moderno");
  }
}

class ClassicWindow implements Window {
  public void open() {
    System.out.println("Abriendo ventana clásica");
  }
}
```

```
27
28 class ModernWindow implements Window {
29   public void open() {
30     System.out.println("Abriendo ventana moderna");
31   }
32 }
33
34 // Abstract Factory
35 interface GUIFactory {
36   Button createButton();
37   Window createWindow();
38 }
39
40 // Fábricas concretas
41 class ClassicFactory implements GUIFactory {
42   public Button createButton() {
43     return new ClassicButton();
44   }
45   public Window createWindow() {
46     return new ClassicWindow();
47   }
48 }
49
50 class ModernFactory implements GUIFactory {
51   public Button createButton() {
52     return new ModernButton();
53   }
54   public Window createWindow() {
55     return new ModernWindow();
56   }
57 }
58
59 // Uso del patrón
60 public class Main {
61   public static void createGUI(GUIFactory factory) {
62     Button button = factory.createButton();
63     Window window = factory.createWindow();
64     button.render();
65     window.open();
66   }
67
68   public static void main(String[] args) {
69     GUIFactory factory = new ClassicFactory();
70     createGUI(factory);
71
72     factory = new ModernFactory();
73     createGUI(factory);
74   }
```

```
75 }
```

Ejemplo 18.1: Ejemplo de Abstract Factory en Java

En este ejemplo, el cliente puede cambiar el estilo de la interfaz simplemente cambiando la fábrica utilizada, sin modificar el código que utiliza los componentes.

Ejercicio 18.1

Implementa el patrón *Abstract Factory* para una aplicación que debe crear componentes de interfaz gráfica para dos sistemas operativos diferentes: Windows y Linux. Define las interfaces necesarias y las fábricas concretas para cada sistema operativo. Escribe un ejemplo de uso donde se creen botones y ventanas para ambos sistemas operativos y se muestre cómo el cliente puede cambiar el sistema operativo sin modificar el código que utiliza los componentes.

18.2. Patrones estructurales

Los patrones estructurales se enfocan en la manera en que las clases y los objetos se organizan y se relacionan entre sí para formar estructuras más grandes y flexibles. Su objetivo es facilitar la composición y reutilización de clases y objetos, permitiendo que el sistema evolucione sin grandes modificaciones.

Algunos de los patrones estructurales más utilizados son:

- **Adapter**: Permite que la interfaz de una clase sea compatible con la que espera el cliente, facilitando la integración de clases con interfaces incompatibles.
- **Bridge**: Separa la abstracción de su implementación, permitiendo que ambas evolucionen independientemente.
- **Composite**: Permite tratar objetos individuales y composiciones de objetos de manera uniforme.
- **Decorator**: Añade funcionalidades adicionales a un objeto de manera dinámica, sin modificar su estructura original.
- **Facade**: Proporciona una interfaz simplificada para un conjunto de interfaces en un subsistema, facilitando su uso.
- **Flyweight**: Reduce el consumo de memoria compartiendo la mayor cantidad posible de datos entre objetos similares.

- **Proxy**: Proporciona un objeto sustituto que controla el acceso a otro objeto, permitiendo realizar acciones adicionales antes o después del acceso.

Estos patrones ayudan a construir sistemas más modulares y escalables, facilitando la gestión de la complejidad en el diseño de software.

18.2.1. El patrón *Adapter*

El patrón *Adapter* es un patrón estructural que permite que dos interfaces incompatibles trabajen juntas. Su objetivo es convertir la interfaz de una clase en otra interfaz que el cliente espera, facilitando la reutilización de código existente sin modificarlo. El adaptador actúa como un intermediario entre el cliente y el servicio, traduciendo las llamadas y datos entre ambos.

Este patrón es útil cuando se necesita integrar clases con interfaces diferentes, por ejemplo, al utilizar bibliotecas de terceros o al migrar sistemas antiguos.

A continuación se muestra un ejemplo de uso del patrón *Adapter* en Java:

```java
interface MediaPlayer {
  void play(String filename);
}

// Clase existente con interfaz incompatible
class AdvancedMediaPlayer {
  public void playMp3(String filename) {
    System.out.println("Reproduciendo MP3: " + filename);
  }
  public void playMp4(String filename) {
    System.out.println("Reproduciendo MP4: " + filename);
  }
}

// Adaptador que permite usar AdvancedMediaPlayer como MediaPlayer
class MediaAdapter implements MediaPlayer {
  private AdvancedMediaPlayer advancedPlayer;

  public MediaAdapter() {
    advancedPlayer = new AdvancedMediaPlayer();
  }

  public void play(String filename) {
    if (filename.endsWith(".mp3")) {
      advancedPlayer.playMp3(filename);
    } else if (filename.endsWith(".mp4")) {
      advancedPlayer.playMp4(filename);
    } else {
      System.out.println("Formato no soportado: " + filename);
```

```
    }
  }
}

// Uso del patrón Adapter
public class Main {
  public static void main(String[] args) {
    MediaPlayer player = new MediaAdapter();
    player.play("cancion.mp3");
    player.play("video.mp4");
    player.play("documento.pdf");
  }
}
```

Ejemplo 18.2: Ejemplo de Adapter en Java

En este ejemplo, `MediaAdapter` permite que el cliente utilice la interfaz `MediaPlayer` para reproducir archivos MP3 y MP4, adaptando las llamadas a los métodos de `AdvancedMediaPlayer` según el formato del archivo.

Ejercicio 18.2

Implementa el patrón *Adapter* para integrar dos sistemas de pago con interfaces diferentes: uno que utiliza el método `pagarConTarjeta(String numeroTarjeta, double monto)` y otro que utiliza el método `realizarPago(String cuenta, double cantidad)`. Define una interfaz común `Pago` con el método `pagar(String identificador, double monto)` y crea los adaptadores necesarios para que ambos sistemas puedan ser utilizados de manera uniforme por el cliente. Escribe un ejemplo de uso donde se realicen pagos con ambos sistemas a través de la interfaz común.

18.3. Patrones de comportamiento

Los patrones de comportamiento se centran en la manera en que los objetos interactúan y se comunican entre sí. Su propósito principal es definir cómo se distribuyen las responsabilidades y cómo se gestionan las interacciones entre los distintos componentes del sistema, promoviendo la flexibilidad y la reutilización del código.

Algunos de los patrones de comportamiento más conocidos son:

- **Chain of Responsibility**: Permite que varios objetos tengan la oportunidad de manejar una solicitud, evitando el acoplamiento entre el emisor y el receptor.

- **Command**: Encapsula una petición como un objeto, permitiendo parametrizar clientes con diferentes solicitudes y soportar operaciones como deshacer.
- **Interpreter**: Define una representación para un lenguaje y un intérprete para analizar sentencias de ese lenguaje.
- **Iterator**: Proporciona una manera de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.
- **Mediator**: Define un objeto que encapsula la manera en que interactúan un conjunto de objetos, promoviendo el desacoplamiento.
- **Memento**: Permite capturar y restaurar el estado interno de un objeto sin violar la encapsulación.
- **Observer**: Define una dependencia uno-a-muchos entre objetos, de modo que cuando uno cambia su estado, todos sus dependientes son notificados automáticamente.
- **State**: Permite que un objeto altere su comportamiento cuando su estado interno cambia.
- **Strategy**: Define una familia de algoritmos, encapsula cada uno y los hace intercambiables.
- **Template Method**: Define el esqueleto de un algoritmo en una operación, dejando algunos pasos a las subclases.
- **Visitor**: Permite definir nuevas operaciones sobre una estructura de objetos sin cambiar las clases de los objetos sobre los que opera.

Estos patrones son esenciales para organizar la lógica de interacción y el flujo de control en aplicaciones complejas, facilitando la extensión y el mantenimiento del software.

18.3.1. El patrón *Visitor*

El patrón *Visitor* es un patrón de comportamiento que permite definir nuevas operaciones sobre una estructura de objetos sin modificar las clases de los objetos sobre los que opera. Este patrón es útil cuando se necesita realizar distintas operaciones sobre una colección de objetos con diferentes tipos, y se desea evitar la proliferación de métodos en las clases de dichos objetos.

El patrón *Visitor* separa la lógica de las operaciones de la estructura de los objetos, facilitando la extensión de funcionalidades sin alterar el código existente. Para ello, se define una interfaz `Visitor` con métodos para cada tipo de objeto, y cada clase de objeto acepta un visitante mediante el método `accept`.

A continuación se muestra un ejemplo de uso del patrón *Visitor* en Java:

```
interface Visitor {
  void visit(Book book);
  void visit(Magazine magazine);
}

interface Element {
  void accept(Visitor visitor);
}

class Book implements Element {
  String title;
  public Book(String title) { this.title = title; }
  public void accept(Visitor visitor) { visitor.visit(this); }
}

class Magazine implements Element {
  String name;
  public Magazine(String name) { this.name = name; }
  public void accept(Visitor visitor) { visitor.visit(this); }
}

// Visitor concreto
class PrintVisitor implements Visitor {
  public void visit(Book book) {
    System.out.println("Libro: " + book.title);
  }
  public void visit(Magazine magazine) {
    System.out.println("Revista: " + magazine.name);
  }
}

// Uso del patrón Visitor
public class Main {
  public static void main(String[] args) {
    Element[] items = { new Book("Patrones de Diseño"), new Magazine("Java Monthly") };
    Visitor visitor = new PrintVisitor();
    for (Element item : items) {
      item.accept(visitor);
    }
  }
```

```
41 }
```

Ejemplo 18.3: Ejemplo de Visitor en Java

En este ejemplo, el visitante `PrintVisitor` implementa la lógica para imprimir información de libros y revistas, y puede añadirse fácilmente nuevos visitantes para realizar otras operaciones sin modificar las clases `Book` y `Magazine`.

Ejercicio 18.3

Implementa el patrón *Visitor* para una aplicación que gestiona diferentes tipos de figuras geométricas: círculo, cuadrado y triángulo. Define una interfaz `Visitor` con métodos para cada tipo de figura y una interfaz `Figura` con el método `accept`. Crea un visitante concreto que calcule el área de cada figura y muestra un ejemplo de uso donde se calcula el área de varias figuras utilizando el visitante. Crea otro visitante que calcule el perímetro de cada figura y muestra un ejemplo de uso similar.

Resumen

En este capítulo se han presentado los principales patrones de diseño de software, agrupados en tres categorías: creacionales, estructurales y de comportamiento. Estos patrones ofrecen soluciones probadas a problemas recurrentes en el desarrollo de aplicaciones, facilitando la creación de sistemas más flexibles, escalables y mantenibles. La correcta aplicación de los patrones de diseño permite reducir el acoplamiento, mejorar la reutilización del código y simplificar la gestión de la complejidad en proyectos de software. Comprender y utilizar estos patrones es fundamental para cualquier desarrollador que aspire a escribir código profesional y robusto. Para profundizar en el estudio de los patrones de diseño, se recomienda consultar la referencia: `https://refactoring.guru/design-patterns`.

19

Programación paralela

En este capítulo se exploran los conceptos fundamentales de la programación paralela y concurrente, centrándose en el lenguaje Java. Aprenderás cómo crear y gestionar hilos, sincronizar el acceso a recursos compartidos, y utilizar herramientas como mutex, semáforos y clases atómicas para garantizar la seguridad y eficiencia en aplicaciones que requieren ejecutar múltiples tareas simultáneamente. Además, se presentan ejemplos prácticos y recomendaciones para evitar errores comunes en entornos concurrentes.

19.1. Introducción a la programación paralela

La programación paralela permite que un programa ejecute varias tareas al mismo tiempo, aprovechando los procesadores multinúcleo y mejorando el rendimiento en operaciones intensivas. En Java, la programación paralela se puede realizar mediante hilos (threads), el framework `Executor`, y las APIs de concurrencia y paralelismo.

Importante

La programación paralela requiere especial atención a la sincronización y la gestión de recursos compartidos para evitar errores como condiciones de carrera y bloqueos.

19.2. Las clases `Thread`, y `Runnable`

Un hilo (*thread*) es una unidad de ejecución dentro de un proceso. Los programas pueden crear múltiples hilos para realizar tareas simultáneamente, como cálculos, acceso a archivos o comunicación en red.

- **Extender la clase `Thread`**: Consiste en crear una nueva clase que herede de `Thread` y sobrescriba el método `run()`, donde se define el código que

ejecutará el hilo. Luego, se crea una instancia de la clase y se llama al método `start()` para iniciar el hilo. Esta forma es sencilla, pero limita la herencia, ya que Java no permite heredar de más de una clase.

- **Implementar la interfaz `Runnable`**[1]: Se crea una clase que implementa la interfaz `Runnable` y define el método `run()`. Después, se crea un objeto `Thread` pasando una instancia de la clase `Runnable` como argumento y se invoca `start()`. Esta forma es más flexible, ya que permite que la clase implemente otras interfaces o extienda otras clases.

```
class MiHilo extends Thread {
  public void run() {
    System.out.println("Hola desde el hilo " + getName());
  }
}

MiHilo hilo1 = new MiHilo();
MiHilo hilo2 = new MiHilo();
hilo1.start();
hilo2.start();
hilo1.join();
hilo2.join();
```

Ejemplo 19.1: Hilo mediante extensión de Thread

Al crear un hilo mediante la extensión de `Thread`, se sobrescribe el método `run()` para definir la tarea que ejecutará el hilo. Luego, se crea una instancia de la clase y se llama al método `start()` para iniciar el hilo. Este hilo completará su ejecución hasta que termine el método `run()`. El hilo principal esperará a que acabe cada hilo, si se llama al método `join()`.

```
class Tarea implements Runnable {
  public void run() {
    System.out.println("Ejecutando tarea en hilo " + Thread.↪
  currentThread().getName());
  }
}

Thread hilo = new Thread(new Tarea());
hilo.start();
```

Ejemplo 19.2: Hilo mediante Runnable

Al usar la interfaz `Runnable`, se define el método `run()` en una clase que implementa la interfaz. Al pasar un objeto de esa clase al constructor de la clase

[1]Esta es una interfaz funcional, así que es posible usar una expresión lambda en vez de crear una clase específica

`Thread`, se crea un hilo que ejecutará el código definido en `run()`, lo cual indica que el hilo delega el código a la clase `Runnable`.

```
Thread hiloLambda = new Thread(() -> {
  System.out.println("Ejecutando hilo con lambda: " + Thread.↪
  currentThread().getName());
});
hiloLambda.start();
```

Ejemplo 19.3: Hilo mediante expresión lambda

Una manera más compacta es usar una expresión lambda. De esta manera podemos reducir la cantidad de código necesario para crear un hilo. Sin embargo, al usar expresiones lambda, se pierde la capacidad de reutilizar el código en diferentes contextos, ya que no se puede instanciar la clase `Runnable` de nuevo.

En todos estos casos, el código de cada hilo se ejecutará independientemente del hilo principal, sin coordinación alguna. Por ello es importante introducir mecanismos de concurrencia.

19.3. Concurrencia

La concurrencia permite que varios hilos se ejecuten y compartan recursos. Es fundamental controlar el acceso a los recursos compartidos para evitar inconsistencias y errores. Existen múltiples estrategias al respecto, pero aquí hablaremos de las más comunes.

19.3.1. Las clases `Mutex`, y `Semaphore`

En Java, los mutex y semáforos se utilizan para controlar el acceso a recursos compartidos.

- **Mutex (ReentrantLock):**
  - Permite bloquear y desbloquear explícitamente, lo que da mayor control sobre la sección crítica.
  - Soporta bloqueos reentrantes, es decir, el mismo hilo puede adquirir el lock varias veces.
  - Ofrece métodos avanzados como `tryLock()` y soporte para condiciones (`Condition`).
- **Semaphore:**
  - Permite controlar el número de hilos que acceden simultáneamente a un recurso.

- Es útil para limitar el acceso concurrente, por ejemplo, a un grupo de conexiones o recursos.
- Puede implementar sincronización de múltiples productores/consumidores.

```java
public class EjemploConcurrencia {
  private static ReentrantLock lock = new ReentrantLock();
  private static Semaphore sem = new Semaphore(2);

  public static void main(String[] args) {
    Thread[] hilos = new Thread[5];
    // Crear los hilos
    for (int i = 0; i < 5; i++) {
      final int id = i+1;
      hilos[i] = new Thread(() -> metodoParalelo(id));
    }
    // Ejecutar cada hilo
    for (Thread hilo : hilos) {
      hilo.start();
    }
    // Esperar a que terminen todos los hilos
    for (Thread hilo : hilos) {
      try {
        hilo.join();
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
    }
  }

  private static void metodoParalelo(int id){
    System.out.println("Hilo " + id + " intentando acceder a la sección crítica");
    seccionCritica();
    System.out.println("Hilo " + id + " ha salido de la sección crítica");
    System.out.println("Hilo " + id + " intentando acceder al recurso con semáforo");
    usarSemaforo();
    System.out.println("Hilo " + id + " ha salido del recurso con semáforo");
  }

  public static void seccionCritica() {
    lock.lock();
    try {
      System.out.println("Accediendo a la sección crítica");
      Thread.sleep(1000);
```

```
40      } catch (InterruptedException e) {
41        Thread.currentThread().interrupt();
42      } finally {
43        lock.unlock();
44      }
45    }
46
47    public static void usarSemaforo() {
48      try {
49        sem.acquire();
50        System.out.println("Accediendo al recurso con semáforo");
51        Thread.sleep(1000);
52      } catch (InterruptedException e) {
53        Thread.currentThread().interrupt();
54      } finally {
55        sem.release();
56      }
57    }
58  }
```

19.4. Atomicidad y consistencia

La atomicidad garantiza que una operación se realiza completamente o no se realiza en absoluto. La consistencia asegura que los datos permanezcan válidos tras las operaciones concurrentes. Como cada hilo puede ejecutarse en distinto orden a otros, o durante las instrucciones de otro hilo, es importante controlar que el acceso a los datos, y su modificación, ocurra de manera atómica.

19.4.1. La palabra clave `synchronized`

La palabra clave `synchronized` permite sincronizar métodos o bloques de código para evitar que varios hilos accedan simultáneamente a una sección crítica.

```
1  class Contador {
2    private int valor = 0;
3    public synchronized void incrementar() {
4      valor++;
5    }
6    public int getValor() { return valor; }
7  }
```

Ejemplo 19.4: Método sincronizado

La palabra clave `synchronized` es una herramienta sencilla y poderosa para controlar el acceso concurrente a recursos compartidos en Java. Permite que sólo

un hilo acceda a una sección crítica a la vez, evitando condiciones de carrera y garantizando la consistencia de los datos.

Ventajas:

- **Facilidad de uso:** Es fácil de aplicar tanto en métodos como en bloques de código.
- **Legibilidad:** El código sincronizado es claro y explícito, lo que facilita su comprensión.
- **Integración:** Funciona bien con los mecanismos internos de Java, como los monitores de objetos.

Desventajas:

- **Rendimiento:** Puede causar bloqueos y reducir la concurrencia si se usa en exceso o en secciones de código extensas.
- **Deadlocks:** Un uso incorrecto puede provocar bloqueos mutuos entre hilos (deadlocks).
- **Falta de flexibilidad:** No permite intentar adquirir el bloqueo sin esperar, ni ofrece mecanismos avanzados como condiciones.
- **Dificultad de depuración:** Los errores de sincronización pueden ser difíciles de detectar y depurar.

Por ello, aunque `synchronized` es adecuado para casos simples, en aplicaciones complejas suele ser preferible usar clases de concurrencia avanzadas como `ReentrantLock` o herramientas específicas de la API de concurrencia de Java.

19.4.2. Las clases atómicas

Las clases atómicas de Java (como `AtomicInteger`, o `AtomicReference`) permiten realizar operaciones seguras sin necesidad de sincronización explícita.

```
1 import java.util.concurrent.atomic.AtomicInteger;
2
3 AtomicInteger contador = new AtomicInteger(0);
4 contador.incrementAndGet();
5 System.out.println(contador.get());
```

Ejemplo 19.5: Uso de AtomicInteger

El uso de clases atómicas es especialmente útil en situaciones donde se requiere un alto rendimiento y baja latencia, ya que minimizan la sobrecarga de la sincronización. Además, permite que código que se ejecuta en diferentes hilos modifique el mismo objeto de manera segura, sin necesidad de bloquearlo.

```java
public class UsuariosRepository {
  private final AtomicInteger contador = new AtomicInteger(0);
  private final UsuariosApi usuariosApi;

  // Constructor

  public int agregarUsuario(String nombre) {
    int id = contador.incrementAndGet();
    usuariosApi.agregarUsuario(id, nombre);
    return id;
  }

  public int getTotalUsuarios() {
    return contador.get();
  }

  public String getUsuario(int id) {
    return usuariosApi.getUsuario(id);
  }
}
```

En el código anterior, la clase `UsuariosRepository` utiliza un `AtomicInteger` para llevar un conteo de usuarios. Cada vez que se agrega un usuario, se incrementa el contador de manera atómica, garantizando que no haya problemas de concurrencia al acceder al mismo recurso desde múltiples hilos. Esto asegura que cada usuario tiene un identificador distinto, único y secuencial.

Ejercicio 19.1

Crea un programa que sume los números del 1 al 100 de forma paralela utilizando `AtomicInteger`. Cada hilo debe sumar un rango de números y actualizar el contador atómico. Al final, imprime el resultado total.

19.4.3. Colecciones concurrentes

Java proporciona colecciones concurrentes como `ConcurrentHashMap` y `CopyOnWrite` para trabajar de forma segura con múltiples hilos. Al usar este tipo de colecciones en nuestro código, permitiremos que en el futuro se puedan realizar operaciones concurrentes sin necesidad de bloqueos explícitos.

```java
import java.util.concurrent.ConcurrentHashMap;

ConcurrentHashMap<String, Integer> mapa = new ConcurrentHashMap<>();
Thread hilo = new Thread(() -> {
  mapa.put("c", 3);
});
```

```
7  mapa.put("a", 1);
8  hilo.start();
9  mapa.put("b", 2);
10 hilo.join();
11 System.out.println(mapa.get("c"));
```

Ejemplo 19.6: Uso de ConcurrentHashMap

Resumen

La programación paralela en Java permite ejecutar múltiples tareas simultáneamente, optimizando el uso de los procesadores y mejorando el rendimiento de las aplicaciones. Para lograrlo, se emplean hilos, mecanismos de sincronización como `synchronized`, locks y semáforos, así como clases atómicas y colecciones concurrentes que garantizan la seguridad y consistencia de los datos compartidos. El uso adecuado de estas herramientas es fundamental para evitar errores como condiciones de carrera y bloqueos, y para desarrollar aplicaciones robustas y eficientes en entornos concurrentes.

Cuadro 19.1: Principales métodos de la clase `AtomicInteger`

Método	**Descripción**
`get()`	Obtiene el valor actual de manera atómica.
`set(int newValue)`	Establece el valor de manera atómica.
`getAndSet(int newValue)`	Establece el valor y devuelve el valor anterior, de forma atómica.
`incrementAndGet()`	Incrementa el valor en uno y devuelve el resultado, de forma atómica.
`getAndIncrement()`	Devuelve el valor actual y luego lo incrementa en uno, de forma atómica.
`decrementAndGet()`	Decrementa el valor en uno y devuelve el resultado, de forma atómica.
`getAndDecrement()`	Devuelve el valor actual y luego lo decrementa en uno, de forma atómica.
`addAndGet(int delta)`	Suma el valor indicado y devuelve el resultado, de forma atómica.
`getAndAdd(int delta)`	Devuelve el valor actual y luego suma el valor indicado, de forma atómica.
`compareAndSet(int expect, int update)`	Actualiza el valor si coincide con el esperado, de forma atómica. Devuelve `true` si la actualización fue exitosa.

20

Clases de utilidad

Estas clases forman parte del paquete estándar de Java y están diseñadas para resolver problemas frecuentes en el desarrollo de aplicaciones. Utilizarlas correctamente permite escribir código más limpio, seguro y eficiente. A lo largo de este capítulo, se explicarán las funcionalidades principales de cada clase, con ejemplos prácticos que ilustran su uso en situaciones reales. El dominio de estas utilidades es fundamental para cualquier programador Java, ya que facilitan tareas como cálculos matemáticos, manipulación de colecciones, manejo de fechas y horas, generación de identificadores únicos, procesamiento de cadenas y entrada/salida de datos.

20.1. La clase `Math`

La clase `Math` proporciona métodos estáticos para realizar operaciones matemáticas avanzadas, como cálculos aritméticos, trigonométricos, exponenciales y logarítmicos. No es necesario crear una instancia de `Math`, ya que todos sus métodos son accesibles directamente desde la clase. Además de los métodos básicos, `Math` incluye constantes útiles como `Math.PI` (el valor de π) y `Math.E` (el número de Euler).

Por ejemplo, para calcular el área de un círculo y el logaritmo natural de un número:

```
1 double radio = 5.0;
2 double area = Math.PI * Math.pow(radio, 2); // Área del círculo
3 double logaritmo = Math.log(10); // Logaritmo natural de 10
```

La clase también ofrece métodos para generar números aleatorios (`Math.random()`), redondear decimales (`Math.ceil`, `Math.floor`), y convertir entre grados y radianes (`Math.toRadians`, `Math.toDegrees`), facilitando cálculos precisos y seguros en cualquier aplicación Java.

Algunos métodos destacados de la clase `Math` son:

- `Math.sqrt(x)`: Devuelve la raíz cuadrada de `x`.

- `Math.pow(a, b)`: Calcula `a` elevado a la potencia `b`.
- `Math.abs(x)`: Retorna el valor absoluto de `x`.
- `Math.round(x)`: Redondea `x` al entero más cercano.
- `Math.max(a, b)` y `Math.min(a, b)`: Obtienen el máximo o mínimo entre dos valores.
- Funciones trigonométricas como `Math.sin(x)`, `Math.cos(x)`, `Math.tan(x)`.

Ejercicio 20.1

Crea un método en Java, que devuelva el valor de la siguiente función matemática:

$$f(x) = \sum_{i=1}^{x} max(e^{-i}, \sin(i))$$

20.2. La clase `Arrays`

La clase `Arrays` proporciona métodos estáticos para trabajar eficientemente con arreglos en Java. Permite ordenarlos, buscarlos, copiarlos, rellenarlos, comparar su contenido y convertirlos en cadenas legibles, entre otras operaciones. Utilizar `Arrays` facilita tareas comunes y reduce errores al manipular datos en forma de array.

Algunos métodos destacados son:

- `Arrays.sort(array)`: Ordena el array de menor a mayor.
- `Arrays.binarySearch(array, valor)`: Busca un valor en un array ordenado y devuelve su posición.
- `Arrays.copyOf(array, nuevaLongitud)`: Crea una copia del array con la longitud indicada.
- `Arrays.fill(array, valor)`: Rellena todo el array con el valor especificado.
- `Arrays.equals(array1, array2)`: Compara si dos arrays son iguales.
- `Arrays.toString(array)`: Devuelve una representación en texto del array.

```java
int[] datos = {5, 2, 8, 1};
Arrays.sort(datos); // Ordena el array
int pos = Arrays.binarySearch(datos, 8); // Busca el valor 8
```

Ejemplo 20.1: Ejemplo de uso de la clase Arrays

Ejercicio 20.2

Crea una función generar bingo, que devuelve 20 números aleatorios entre 1 y 100. Los números deben ser únicos y estar en un array ordenado.

20.3. La clase `Collections`

La clase `Collections` es fundamental para trabajar con las colecciones del paquete `java.util`, como listas, conjuntos y mapas. Proporciona métodos estáticos que permiten realizar operaciones comunes de manera eficiente y segura, como ordenar, buscar, invertir, mezclar, sincronizar y crear colecciones inmutables.

Algunas utilidades destacadas de `Collections` son:

- `Collections.sort(lista)`: Ordena una lista según el orden natural de sus elementos o usando un comparador.
- `Collections.shuffle(lista)`: Mezcla aleatoriamente los elementos de una lista.
- `Collections.reverse(lista)`: Invierte el orden de los elementos de una lista.
- `Collections.max(coleccion)` y `Collections.min(coleccion)`: Obtienen el máximo o mínimo de una colección.
- `Collections.unmodifiableList(lista)`: Devuelve una vista inmutable de la lista.
- `Collections.synchronizedList(lista)`: Devuelve una lista sincronizada para uso seguro en entornos concurrentes.
- `Collections.frequency(coleccion, elemento)`: Cuenta cuántas veces aparece un elemento en una colección.

Estas utilidades permiten manipular colecciones de forma más sencilla y robusta, facilitando tareas como la gestión de datos, la concurrencia y la protección contra modificaciones accidentales.

```
List<String> nombres = Arrays.asList("Ana", "Luis", "Pedro");
Collections.sort(nombres); // Ordena la lista
Collections.shuffle(nombres); // Mezcla aleatoriamente
```

Ejemplo 20.2: Ejemplo de uso de la clase Collections

20.4. La clase `Objects`

La clase `Objects` es una utilidad esencial para trabajar con referencias de objetos en Java, especialmente en lo que respecta a la gestión de valores nulos y la comparación segura. Sus métodos estáticos ayudan a evitar errores comunes, como las excepciones `NullPointerException`, y facilitan la escritura de código más robusto y legible.

Algunas funciones importantes de `Objects` incluyen:

- `Objects.isNull(obj)` y `Objects.nonNull(obj)`: Verifican si una referencia es nula o no.
- `Objects.requireNonNull(obj)`: Lanza una excepción si el objeto es nulo, útil para validar argumentos.
- `Objects.requireNonNullElse(obj, default)`: Devuelve el objeto si no es nulo, o un valor por defecto si lo es.
- `Objects.equals(a, b)`: Compara dos objetos de forma segura, evitando errores si alguno es nulo.
- `Objects.hash(obj1, obj2, ...)`: Calcula el código hash de varios objetos, útil para implementar `hashCode()`.
- `Objects.toString(obj, default)`: Devuelve la representación en texto del objeto, o un valor por defecto si es nulo.

El uso de `Objects` mejora la seguridad y claridad del código, especialmente en métodos que reciben parámetros o manipulan datos que pueden ser nulos.

```
String nombre = null;
System.out.println(Objects.isNull(nombre)); // true
System.out.println(Objects.requireNonNullElse(nombre, "↪
  Desconocido")); // "Desconocido"

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

  @Override
```

```
10    public boolean equals(Object obj) {
11      if (this == obj) return true;
12      if (!(obj instanceof Persona)) return false;
13      Persona otra = (Persona) obj;
14      return Objects.equals(nombre, otra.nombre) && edad == otra.↪
      edad;
15    }
16
17    @Override
18    public int hashCode() {
19      return Objects.hash(nombre, edad);
20    }
21  }
```

Ejemplo 20.3: Ejemplo de uso de la clase Objects

20.5. La clase Random, ThreadLocalRandom y SecureRandom

La clase `Random` es una utilidad fundamental para la generación de números aleatorios en Java. Permite obtener valores enteros, decimales, booleanos y otros tipos, facilitando tareas como simulaciones, juegos, pruebas y selección aleatoria de datos. Por ejemplo, el método `nextInt(n)` devuelve un entero entre 0 (inclusive) y `n` (exclusivo), mientras que `nextDouble()` genera un valor decimal entre 0.0 y 1.0.

Para aplicaciones concurrentes, `ThreadLocalRandom` ofrece mayor rendimiento y seguridad al evitar la contención entre hilos, siendo ideal para entornos multihilo. Su uso es similar al de `Random`, pero se accede mediante `ThreadLocalRandom.curr`

Ejemplo de uso de ambas clases:

```
1  Random rnd = new Random();
2  int valor = rnd.nextInt(100); // Número entre 0 y 99
3
4  int otroValor = ThreadLocalRandom.current().nextInt(100); // ↪
     Número entre 0 y 99 en entorno multihilo
```

Ejemplo 20.4: Uso de Random y ThreadLocalRandom

Por otro lado, `SecureRandom` está diseñada para aplicaciones que requieren mayor seguridad, como generación de contraseñas, claves criptográficas o tokens. Utiliza fuentes de entropía más robustas y algoritmos criptográficos para asegurar la imprevisibilidad de los valores generados.

Ejemplo de uso de ambas clases:

```
1  Random rnd = new Random();
2  int valor = rnd.nextInt(100); // Número entre 0 y 99
```

```
3
4 SecureRandom srnd = new SecureRandom();
5 byte[] bytes = new byte[16];
6 srnd.nextBytes(bytes); // Array de bytes aleatorios seguro
```

Ejemplo 20.5: Uso de Random y SecureRandom

Ejercicio 20.3

Crea una función en Java que calcule la media de n llamadas a otra función, la cual devuelve 4 si la suma del cuadrado de dos números aleatorios (distintos, comprendidos entre 0 y 1) es menor o igual a 1, si no, devuelve 0.
¿Qué resultado obtienes para un valor de n grande?

20.6. Las clases `BigInteger`, y `BigDecimal`

Las clases `BigInteger` y `BigDecimal` permiten realizar cálculos numéricos con precisión arbitraria, superando las limitaciones de los tipos primitivos (`int`, `long`, `double`) en Java. Son esenciales cuando se requiere trabajar con números muy grandes, operaciones matemáticas exactas o cálculos financieros donde los errores de redondeo pueden ser críticos.

`BigInteger` representa enteros de cualquier tamaño, permitiendo operaciones como suma, resta, multiplicación, división, potenciación y cálculo de módulos, sin riesgo de desbordamiento. Por ejemplo, para calcular la secuencia de Fibonacci:

```
1 BigInteger a = BigInteger.ONE;
2 BigInteger b = BigInteger.ZERO;
3 for (int i = 2; i <= 1000; i++) {
4   BigInteger temp = a;
5   a = a.add(b);
6   b = temp;
7   System.out.println(a); // Imprime la secuencia de Fibonacci
8 }
```

Los principales métodos de `BigInteger` incluyen:

- `add(BigInteger val)`: Suma el valor especificado.
- `subtract(BigInteger val)`: Resta el valor especificado.
- `multiply(BigInteger val)`: Multiplica por el valor especificado.
- `divide(BigInteger val)`: Divide por el valor especificado.

- `mod(BigInteger val)`: Calcula el módulo respecto al valor especificado.
- `pow(int exponent)`: Eleva a la potencia indicada.
- `compareTo(BigInteger val)`: Compara dos valores.
- `gcd(BigInteger val)`: Calcula el máximo común divisor.
- `isProbablePrime(int certainty)`: Verifica si el número es probablemente primo.
- `toString()`: Convierte el valor a cadena.

`BigDecimal` se utiliza para representar números decimales con precisión exacta, ideal para aplicaciones financieras, científicas o cualquier contexto donde los errores de coma flotante sean inaceptables. Permite controlar el redondeo y la escala de los resultados:

```
1 BigDecimal precio = new BigDecimal("19.99");
2 BigDecimal cantidad = new BigDecimal("3");
3 BigDecimal total = precio.multiply(cantidad);
4 System.out.println(total); // 59.97
```

Los principales métodos de `BigDecimal` incluyen:

- `add(BigDecimal val)`: Suma el valor especificado.
- `subtract(BigDecimal val)`: Resta el valor especificado.
- `multiply(BigDecimal val)`: Multiplica por el valor especificado.
- `divide(BigDecimal val, MathContext mc)`: Divide por el valor especificado, controlando precisión y redondeo.
- `setScale(int newScale, RoundingMode mode)`: Ajusta la escala (número de decimales) y el modo de redondeo.
- `compareTo(BigDecimal val)`: Compara dos valores decimales.
- `abs()`: Devuelve el valor absoluto.
- `pow(int n)`: Eleva a la potencia indicada.
- `toPlainString()`: Convierte el valor a cadena sin notación científica.
- `stripTrailingZeros()`: Elimina ceros innecesarios al final de la parte decimal.

Ambas clases son inmutables y ofrecen métodos para comparar, convertir y formatear valores, así como para realizar operaciones matemáticas avanzadas. El uso de `BigInteger` y `BigDecimal` garantiza precisión y seguridad en cálculos que exceden las capacidades de los tipos numéricos estándar.

Ejercicio 20.4

Implementa un programa que calcule el factorial de un número grande utilizando `BigInteger`, y otro que lo haga usando la primitiva long. ¿Difiere el resultado?

20.7. Clases para fechas y horas: `LocalDate`, `LocalTime`, `LocalDateTime`, `Duration`, `Period`

Las clases `LocalDate`, `LocalTime`, `LocalDateTime`, `Duration` y `Period` forman parte del paquete `java.time`, introducido en Java 8 para facilitar el manejo de fechas y horas de manera clara, segura y sin los problemas de la antigua API (`Date`, `Calendar`). Estas clases son inmutables y thread-safe, lo que evita errores en aplicaciones concurrentes.

- `LocalDate`: Representa una fecha sin hora (año, mes, día).
- `LocalTime`: Representa una hora sin fecha (hora, minuto, segundo).
- `LocalDateTime`: Combina fecha y hora.
- `Duration`: Modela una cantidad de tiempo en horas, minutos, segundos, útil para medir intervalos temporales.
- `Period`: Modela una cantidad de tiempo en años, meses y días, útil para calcular diferencias entre fechas.

Estas clases permiten realizar operaciones como sumar o restar días, comparar fechas, calcular diferencias, formatear y analizar cadenas de texto, y trabajar con zonas horarias mediante `ZonedDateTime`. Por ejemplo, para calcular cuántos días faltan para una fecha específica, se utiliza `Period.between`. Para medir la duración entre dos instantes, se emplea `Duration.between`.

Ejemplo de uso de `LocalDate`, `Period` y `Duration`:

```
LocalDate hoy = LocalDate.now();
LocalDate cumple = LocalDate.of(2025, 8, 18);
Period periodo = Period.between(hoy, cumple);
System.out.println(periodo.getDays()); // Días hasta el cumpleaños

LocalTime inicio = LocalTime.of(9, 0);
LocalTime fin = LocalTime.of(17, 30);
Duration jornada = Duration.between(inicio, fin);
System.out.println(jornada.toHours()); // Horas trabajadas
```

El uso de estas clases mejora la precisión y legibilidad del código al trabajar con fechas y horas, evitando errores comunes y facilitando el desarrollo de aplicaciones que requieren cálculos temporales.

20.8. La clase `UUID`

La clase `UUID` (Universally Unique Identifier) permite generar identificadores únicos de 128 bits, útiles para distinguir objetos, registros o entidades en sistemas distribuidos, bases de datos y aplicaciones donde se requiere unicidad global. Los UUID se generan de forma aleatoria o basados en información como la hora y la dirección MAC, garantizando que no se repitan incluso en diferentes sistemas.

Algunos métodos importantes de la clase `UUID` son:

- `UUID.randomUUID()`: Genera un UUID aleatorio.
- `UUID.fromString(String uuid)`: Crea un UUID a partir de su representación textual.
- `uuid.toString()`: Devuelve el UUID en formato estándar de texto.
- `uuid.version()`: Indica la versión del UUID (por ejemplo, aleatorio, basado en tiempo, etc.).

El uso de UUID es común en la generación de claves primarias, tokens de sesión, nombres de archivos únicos y cualquier contexto donde la unicidad sea esencial, sin depender de un contador centralizado.

20.9. La clase `Base64`

La clase `Base64`, disponible en `java.util`, es una utilidad para la codificación y decodificación de datos en el formato Base64, ampliamente utilizado para transmitir información binaria como texto legible (por ejemplo, imágenes, archivos o credenciales en protocolos HTTP). Base64 convierte datos binarios en

una cadena de caracteres ASCII, facilitando su almacenamiento y transferencia en sistemas que solo admiten texto.

Las principales funcionalidades de la clase incluyen:

- `Base64.getEncoder()`: Obtiene un codificador para transformar datos binarios en texto Base64.
- `Base64.getDecoder()`: Obtiene un decodificador para revertir el proceso y recuperar los datos originales.
- `Base64.getUrlEncoder()`, `Base64.getMimeEncoder()`: Codificadores especializados para URLs y mensajes MIME.

El uso de `Base64` es común en el intercambio de datos entre sistemas, almacenamiento seguro de información y transmisión de archivos en APIs web. Por ejemplo, para codificar una cadena y luego decodificarla:

```
1 String texto = "Hola mundo";
2 String codificado = Base64.getEncoder().encodeToString(texto.↪
    getBytes());
3 byte[] decodificado = Base64.getDecoder().decode(codificado);
4 System.out.println(new String(decodificado)); // "Hola mundo"
```

Ejemplo 20.6: Codificación y decodificación con Base64

Esta clase simplifica el manejo de datos binarios en aplicaciones Java, asegurando compatibilidad y seguridad en la transferencia de información.

D

Ejercicios

21

Ejercicios de Programación

En este capítulo se presentan ejercicios prácticos para aplicar los conceptos aprendidos en programación. Cada ejercicio está diseñado para reforzar el conocimiento y fomentar la práctica. Deberás aplicar todos los conocimientos adquiridos en los capítulos anteriores.

21.1. Criba de Eratóstenes

La criba de Eratóstenes es un algoritmo clásico para encontrar todos los números primos menores o iguales a un número dado n. El método consiste en ir eliminando los múltiplos de cada número primo comenzando desde el 2, de modo que al finalizar el proceso, los números que no han sido eliminados son los primos.

Ejercicio 21.1

Implementa la criba de Eratóstenes en Java. El programa debe solicitar al usuario un número n y mostrar todos los números primos menores o iguales a n.

Conocimientos requeridos: Estructuras de control, arreglos y algoritmos de búsqueda.

21.2. Balanceo de paréntesis

El balanceo de paréntesis es un problema común en la programación, que consiste en verificar si los paréntesis en una expresión están correctamente balanceados. Esto significa que cada paréntesis de apertura tiene un correspondiente paréntesis de cierre y que están en el orden correcto.

Ejercicio 21.2

Implementa un programa en Java que verifique si una expresión matemática tiene los paréntesis balanceados. El programa debe solicitar al usuario una expresión y mostrar si está balanceada o no.

Una segunda versión de este ejercicio consiste en comprobar el balanceo de otros tipos de paréntesis, como corchetes y llaves.

Ejercicio 21.3

Implementa un programa en Java que verifique si una expresión matemática tiene los paréntesis balanceados, incluyendo corchetes y llaves. El programa debe solicitar al usuario una expresión y mostrar si está balanceada o no.

Conocimientos requeridos: Estructuras de control, manejo de cadenas y pilas.

21.3. Pirateo de contraseñas MD5

El algoritmo de cifrado MD5 es famoso por su rapidez y por generar un *hash* de 128 bits a partir de una entrada de cualquier longitud. Sin embargo, se ha descubierto que es vulnerable a ataques de colisión, lo que significa que es posible encontrar dos entradas diferentes que produzcan el mismo *hash* MD5.

Para generar un *hash* MD5 en Java, puedes utilizar la clase ‘MessageDigest‘ del paquete ‘java.security‘. Aquí tienes un ejemplo de cómo hacerlo:

```
1  import java.security.MessageDigest;
2  import java.security.NoSuchAlgorithmException;
3  import java.util.Scanner;
4
5  public class MD5Example {
6    public static void main(String[] args) throws ↪
     NoSuchAlgorithmException {
7      Scanner scanner = new Scanner(System.in);
8      System.out.print("Introduce la contraseña: ");
9      String input = scanner.nextLine();
10
11     MessageDigest md = MessageDigest.getInstance("MD5");
12     md.update(input.getBytes());
13     byte[] digest = md.digest();
14
15     StringBuilder sb = new StringBuilder();
16     for (byte b : digest) {
17       sb.append(String.format("%02x", b));
18     }
```

```
19      System.out.println("Hash MD5: " + sb.toString());
20    }
21  }
```

Ejemplo 21.1: Generar un hash MD5 en Java

En este caso, el *hash* en MD5 tiene 16 bytes. Para poder atacar este algoritmo se usan las conocidas como *rainbow tables*, que son tablas pre-computadas de *hashes* y sus correspondientes entradas. Estas tablas permiten realizar ataques de colisión de manera más eficiente, ya que evitan la necesidad de calcular el *hash* para cada posible entrada en tiempo real. Estas tablas se guardan en disco en formato binario, y en cada posición se almacena el valor de la contraseña que genera ese valor de *hash*. P.e.: en la posición 0 del fichero está la contraseña cuyo MD5 es 0x0, en la posición 1 aquella que genera el valor 0x1, etc.

Ejercicio 21.4

Crea dos programas.
El primero de ellos deberá generar la tabla *rainbow*, guardando después del espacio de *hashes* cuál fue la última contraseña que se usó para rellenar la tabla. Esto permitirá continuar rellenando la tabla en consecutivas ejecuciones del programa.
El segundo programa, recibirá como entrada un *hash* MD5 y deberá buscar en la tabla *rainbow* la contraseña correspondiente. Si la encuentra, la mostrará; si no, informará que no se encontró.
Consejo: Si cada contraseña puede tener hasta 8 caracteres, al buscar en la tabla la posición n leeremos 8 bytes desde la posición $(n-1)*8$.

Conocimientos requeridos: Estructuras de control, manejo de cadenas y ficheros de acceso aleatorio.

21.4. Agenda personal

Una agenda personal es una aplicación que permite gestionar contactos, incluyendo su nombre, número de teléfono y correo electrónico. El programa debe permitir al usuario agregar, eliminar y buscar contactos. Además, debe ofrecer la opción de listar todos los contactos almacenados.

Ejercicio 21.5

Implementa la agenda personal en Java, teniendo en cuenta las siguientes funcionalidades:

- El usuario podrá acceder a la agenda por consola, o usando una interfaz gráfica.
- La agenda deberá utilizar alguna técnica de persistencia de datos.
- Deberá disponer de una opción para importar, y exportar, contactos desde un fichero de texto, o un fichero XML.

Conocimientos requeridos: Clases avanzadas, manejo de ficheros, persistencia de datos, interfaces gráficas.

21.5. Juego de adivinanza de números

El juego de adivinanza de números es una aplicación que permite a un jugador adivinar un número secreto generado aleatoriamente por la computadora. El programa debe proporcionar pistas al jugador sobre si su suposición es demasiado alta o demasiado baja.

Ejercicio 21.6

Implementa el juego de adivinanza de números en Java, teniendo en cuenta las siguientes funcionalidades:

- El número secreto debe ser generado aleatoriamente en un rango específico (por ejemplo, 1-100).
- El jugador debe poder hacer múltiples intentos para adivinar el número.
- Después de cada intento, el programa debe indicar si la suposición es demasiado alta, demasiado baja o correcta.
- Cuando el jugador adivina el número, el programa debe mostrar un mensaje de felicitación y el número de intentos realizados.

Conocimientos requeridos: Estructuras de control, generación de números aleatorios, manejo de entradas y salidas.

21.6. Juego de ahorcado

El ahorcado es un juego clásico en el que el jugador debe adivinar una palabra secreta letra por letra. El jugador tiene un número limitado de intentos para adivinar la palabra antes de que se complete el dibujo del ahorcado.

Ejercicio 21.7

Implementa el juego de ahorcado en Java, teniendo en cuenta las siguientes funcionalidades:

- La palabra secreta debe ser elegida aleatoriamente de un fichero de texto que contenga una lista predefinida de palabras.
- El jugador debe poder hacer múltiples intentos para adivinar la palabra.
- Después de cada intento, el programa debe mostrar el estado actual de la palabra (letras adivinadas y letras faltantes).
- Cuando el jugador adivina la palabra, el programa debe mostrar un mensaje de felicitación y el número de intentos realizados.

Conocimientos requeridos: Estructuras de control, manejo de cadenas y ficheros de texto.

21.7. Calculadora IRPF

Los impuestos a las personas físicas (IRPF) son un tema importante en la programación financiera. Una calculadora IRPF permite a los usuarios calcular el importe de su impuesto sobre la renta en función de sus ingresos y deducciones.

En este caso, vamos a basarnos en un caso simplificado, ya que existen distintas deducciones, y cada comunidad autónoma puede tener sus propias reglas. Consideraremos que los tramos de IRPF son los siguientes:

- Hasta 12.450 €: 19 %
- De 12.450 € a 20.200 €: 24 %
- De 20.200 € a 35.200 €: 30 %
- De 35.200 € a 60.000 €: 37 %
- Más de 60.000 €: 45 %

Ejercicio 21.8

Crea un programa en Java que calcule el IRPF de un usuario en función de su salario. El programa debe solicitar al usuario que introduzca su salario bruto anual y, a continuación, calcular y mostrar el importe del IRPF a pagar según los tramos establecidos.
Además, mostrará un desglose de cómo se ha calculado el impuesto, indicando el tramo correspondiente y el porcentaje aplicado.
Por último, mostrará cuál es el salario neto del usuario, es decir, el salario bruto menos el IRPF, tanto su valor anual, como su valor para 12 y 14 pagas.
Recuerda que deberás usar precisión de 1 céntimo.

Conocimientos requeridos: Estructuras de control, manejo de entradas y salidas.

Índice de cuadros