Walter Alexander Mata López
Mónica Cobián Alvarado
Armando Román Gallardo
José Román Herrera Morales

PENSAMIENTO ALGORÍTMICO Y PROGRAMACIÓN CON JAVA

Aplicaciones para ingeniería

UNIVERSIDAD DE COLIMA

Pensamiento algorítmico y programación con Java

Aplicaciones para ingeniería

En buen plan

Pensamiento algorítmico y programación con Java

Aplicaciones para ingeniería

Walter Alexander Mata López
Mónica Cobián Alvarado
Armando Román Gallardo
José Román Herrera Morales

Universidad de Colima

Avenida Universidad 333
C.P. 28040, Colima, Colima, México
Dirección General de Publicaciones
Teléfonos: (312) 316 1081 y 316 1000, extensión 35004
Correo electrónico: publicaciones@ucol.mx
www.ucol.mx

ISBN: 978-607-8984-60-2
5E.1.1/317000/093/2024 Edición de publicación no periódica
DOI: 10.53897/LI.2024.0056.UCOL

Proceso editorial certificado con normas Iso desde 2005
Dictaminación doble ciego y edición registradas en el Sistema Editorial Electrónico Pred

Registro: CU-013-24
Recibido: Agosto de 2024
Publicado: Diciembre de 2024

Índice

Prefacio

La programación en Java ha sido, desde su creación en la década de 1990, un pilar fundamental en el desarrollo de *software* a nivel mundial. Este libro se ha concebido con el propósito de servir como una guía exhaustiva y accesible específicamente para el alumnado de las carreras de Ingeniería en Tecnologías de Internet, Ingeniería en Software, Ingeniería en Computación Inteligente, Ingeniería Mecatrónica y carreras afines, así como para estudiantes de licenciaturas de la Universidad de Colima en general.

Java se caracteriza por su portabilidad, robustez y versatilidad, atributos que le han permitido mantenerse vigente y ampliamente utilizado en diversos campos de la tecnología, desde aplicaciones móviles hasta sistemas empresariales complejos. Este texto busca no sólo enseñar los fundamentos del lenguaje, sino también fomentar una comprensión profunda de los principios subyacentes que hacen de Java una herramienta tan poderosa.

Diseñado para servir como un libro de texto en las materias donde se utilice Java como parte del programa de estudios, cada capítulo ha sido cuidadosamente elaborado para llevar al lector a través de un viaje progresivo de aprendizaje. Desde los conceptos básicos hasta técnicas avanzadas, el contenido cubre una amplia gama de temas, incluyendo estructuras de datos, manejo de excepciones, y programación orientada a objetos. Además, se incluyen ejemplos prácticos y ejercicios al final de cada sección, que permiten al lector aplicar lo aprendido y desarrollar sus habilidades de programación de manera efectiva.

Es nuestro deseo que este libro sirva como un recurso valioso para estudiantes de la Universidad de Colima y otras instituciones educativas. Esperamos que, a través de estas páginas, las y los lectores puedan descubrir la elegancia y eficiencia de Java, y que esta comprensión

les inspire a explorar y crear soluciones innovadoras en el vasto mundo del desarrollo de *software*.

Agradecemos a quienes han contribuido a la realización de este libro, desde expertas y expertos que compartieron su conocimiento hasta las personas que revisaron y aportaron sus valiosas críticas. Sin su dedicación y esfuerzo, este proyecto no hubiera sido posible.

Capítulo 1. Introducción a Java

Introducción

Java es uno de los lenguajes de programación más utilizados en la actualidad, ha sobrevivido al paso del tiempo incorporando nuevas características haciéndolo que se mantenga moderno y con la habilidad de dar soporte a versiones anteriores. Es importante conocer sus orígenes y evolución, así como la composición de su plataforma. La plataforma Java incluye el lenguaje de programación, la máquina virtual Java (JVM), el entorno de ejecución Java (JRE) y el kit de desarrollo de Java (JDK). La JVM es fundamental para la portabilidad del *software*, permitiendo que el código compilado se ejecute en cualquier dispositivo con una JVM compatible. El JDK proporciona herramientas para desarrollar, depurar y ejecutar aplicaciones Java, mientras que el JRE es necesario para ejecutar aplicaciones Java. Java se destaca por su orientación a objetos, seguridad, portabilidad y un ecosistema rico y maduro. Ha evolucionado significativamente desde su primera versión en 1995, incorporando mejoras importantes como las expresiones lambda y el sistema de módulos. Su robustez y capacidad de adaptación lo han mantenido como una de las opciones preferidas por desarrolladores a nivel mundial.

Orígenes de Java

Java, desarrollado inicialmente por James Gosling junto con un equipo de ingenieros en Sun Microsystems en 1991, fue concebido como una respuesta a la necesidad de un lenguaje de programación que pudiera ejecutarse en diferentes tipos de dispositivos y plataformas sin necesidad de ser modificado para cada uno. Originalmente llamado Oak, en honor a un árbol que estaba afuera de la oficina de Gosling, y más tarde

renombrado a Java, inspirado por el café que consumían los desarrolladores, el lenguaje fue diseñado con la filosofía de "*Write Once, Run Anywhere*" (WORA), es decir, escribir el código una vez y ejecutarlo en cualquier dispositivo. Este principio se convirtió en la piedra angular de Java, permitiéndole ganar una popularidad masiva en la industria del *software*.

Gestión y evolución bajo Oracle

Desde la adquisición de Sun Microsystems por parte de Oracle en 2010, Oracle ha continuado la gestión y evolución de Java, manteniendo un compromiso con la innovación y el desarrollo comunitario. Oracle ha implementado un nuevo modelo de lanzamiento de seis meses para Java, lo cual significa que nuevas características y mejoras son introducidas de manera más frecuente y predecible. Este modelo permite a la comunidad de Java adaptarse rápidamente a los cambios y aprovechar las nuevas características, sin tener que esperar años por actualizaciones mayores. La evolución continua de Java demuestra su capacidad para adaptarse a los cambiantes paradigmas de programación y las crecientes demandas del desarrollo de *software* moderno. Con cada versión, Java refuerza su posición como un lenguaje de programación robusto, seguro y versátil, elegido por millones de desarrolladores en todo el mundo.

La plataforma Java

Java no es sólo un lenguaje de programación, es una plataforma completa que incluye el lenguaje de programación Java, el entorno de ejecución Java (JRE por sus siglas en inglés), la máquina virtual Java (JVM, por sus siglas en inglés), y un amplio conjunto de bibliotecas estándar. La JVM es una pieza clave en la plataforma Java, ya que permite que el código compilado a *bytecode* se ejecute en cualquier dispositivo o sistema operativo que cuente con una JVM compatible, facilitando así la portabilidad del *software*. A continuación, se detallan los principales componentes y características de la plataforma Java:

- **Máquina Virtual de Java (JVM: *Java Virtual Machine*):** la JVM es el corazón de la plataforma Java. Es el componente que permite que los programas Java se ejecuten en cualquier dispositivo o sistema operativo sin necesidad de ser reescritos o recompilados para cada plataforma específica. La JVM es una máquina virtual de propósito general que se ha diseñado para ejecutar *bytecode* Java, un código intermedio al que el código fuente Java se compila. La independencia de la plataforma se logra a través de este proceso de compilación a *bytecode*, seguido de la interpretación o compilación just-in-time (JIT) del *bytecode* a código nativo por la JVM en el dispositivo de destino.
- **Kit de Desarrollo de Java (JDK: *Java Development Kit*):** el JDK es un conjunto de herramientas de desarrollo para escribir, depurar y ejecutar programas Java. Incluye el compilador Java (`javac`), que convierte el código fuente Java en *bytecode*, diversas herramientas de utilidad para empaquetar y gestionar aplicaciones Java y una copia del JRE (ver abajo). Además, el JDK proporciona bibliotecas estándar de Java que ofrecen funcionalidades predefinidas para tareas comunes de programación, como manipulación de cadenas, gestión de redes, desarrollo de interfaces gráficas de usuario y acceso a bases de datos.
- **Entorno de Ejecución Java (JRE: *Java Runtime Environment*):** el JRE es el conjunto de *software* necesario para ejecutar aplicaciones Java. Incluye la JVM, las bibliotecas de clases del núcleo de Java y otros componentes que soportan la ejecución de programas Java. El JRE es lo que se instala en un dispositivo para permitir que las aplicaciones Java se ejecuten en él. Sin embargo, para desarrollar aplicaciones Java, los programadores necesitan el JDK, que incluye el JRE y las herramientas de desarrollo.
- **Bibliotecas de clases Java:** Java viene con un conjunto extenso y completo de bibliotecas de clases, también conocidas como la API de Java. Estas bibliotecas proporcionan una amplia va-

riedad de funcionalidades, desde operaciones básicas de entrada/salida hasta complejos servicios de red y herramientas de desarrollo web. Las bibliotecas están organizadas en paquetes que cubren diferentes áreas de la computación, facilitando a las y los desarrolladores la tarea de construir aplicaciones ricas y eficientes sin tener que reinventar la rueda.

- **Seguridad:** la plataforma Java fue diseñada con un fuerte enfoque en la seguridad. Incorpora un modelo de seguridad de arena de ejecución (*sandbox*) que restringe las capacidades de las aplicaciones Java, evitando que realicen operaciones no autorizadas que podrían ser perjudiciales para el sistema anfitrión. Además, la plataforma ofrece un robusto mecanismo de gestión de seguridad que incluye la verificación de *bytecode*, la carga segura de clases y las políticas de seguridad que controlan el acceso a los recursos del sistema.
- **Portabilidad:** Una de las premisas fundamentales de Java es su portabilidad. Las aplicaciones Java pueden ejecutarse en cualquier dispositivo que tenga instalado el JRE compatible, sin necesidad de modificaciones. Esto se logra gracias a la arquitectura neutral del *bytecode* y la JVM, que abstrae las diferencias entre los sistemas operativos y el *hardware.*

La plataforma Java es una tecnología robusta y madura que ha sido refinada a lo largo de los años para ofrecer un entorno de desarrollo y ejecución de alto rendimiento, seguro y portátil. Gracias a su diseño modular y extensible, Java sigue siendo una elección popular entre desarrolladores para todo tipo de aplicaciones, desde el desarrollo web y móvil hasta sistemas empresariales de gran escala.

Características clave de Java

Java está considerado como un lenguaje de programación de alto nivel, se distingue por su enfoque en la orientación a objetos, una metodología que permite a los desarrolladores estructurar el *software* como

una colección de objetos distintos que encapsulan sus propios datos y métodos. Esta característica no solo facilita el diseño de programas complejos mediante la división en componentes más manejables y reutilizables, sino que también mejora la legibilidad y mantenibilidad del código. Además de su paradigma orientado a objetos, Java incorpora una serie de características clave que lo convierten en un lenguaje versátil y potente:

- **Portabilidad y plataforma cruzada:** gracias a la JVM, los programas Java pueden ejecutarse en cualquier dispositivo o plataforma que disponga de una JVM, cumpliendo con el principio de "escribir una vez, ejecutar en cualquier lugar". Esta portabilidad elimina los problemas de compatibilidad y simplifica el despliegue de aplicaciones en diversos entornos.
- **Seguridad y robustez:** Java fue diseñado con un fuerte énfasis en la seguridad, incluyendo características como el recolector de basura (*garbage collector*) y un riguroso sistema de gestión de tipos, que ayudan a prevenir problemas comunes como fugas de memoria y violaciones de acceso. Estas medidas de seguridad hacen que Java sea particularmente adecuado para aplicaciones en entornos críticos.
- **Concurrencia integrada:** el soporte integrado para programación multihilo en Java permite a desarrolladores construir aplicaciones interactivas y con un rendimiento optimizado, aprovechando los modernos procesadores multinúcleo para realizar múltiples tareas de forma simultánea.
- **Ecosistema rico y maduro:** Java se beneficia de un extenso ecosistema que incluye un vasto conjunto de bibliotecas estándar, *frameworks* y herramientas de desarrollo que abarcan desde el desarrollo web y móvil hasta la computación en la nube y el análisis de grandes volúmenes de datos. Este ecosistema, apoyado por una activa comunidad de desarrolladores, facilita y acelera el desarrollo de aplicaciones robustas y escalables.

- **Rendimiento:** aunque el rendimiento de Java fue una preocupación en sus primeras versiones, mejoras significativas en la JVM, como la compilación Just-In-Time (JIT) y optimizaciones del *garbage collector*, han hecho que el rendimiento de las aplicaciones Java sea competitivo con otros lenguajes compilados nativamente. Java ofrece un equilibrio entre rendimiento y portabilidad, adecuado para una amplia gama de aplicaciones, desde sistemas embebidos hasta aplicaciones empresariales de gran escala.
- **Interoperabilidad:** a través de la Java Native Interface (JNI) y otras tecnologías de interoperabilidad, Java puede interactuar con código y bibliotecas escritas en otros lenguajes, como C y C++. Esto permite a las y los desarrolladores aprovechar las bibliotecas existentes y el código nativo del sistema para mejorar la funcionalidad y el rendimiento de las aplicaciones Java.
- **Programación funcional:** desde Java 8, el lenguaje ha incorporado características de programación funcional, como expresiones lambda, *streams*, y la API Optional, que facilitan el desarrollo de código más limpio, conciso y expresivo. Estas características promueven un estilo de programación que puede reducir errores y mejorar la legibilidad del código.
- **Gestión de dependencias y modularidad:** con la introducción del sistema de módulos en Java 9 (Project Jigsaw), Java ofrece un sistema de gestión de dependencias y modularidad robusto que permite a los desarrolladores encapsular funcionalidades en módulos con interfaces bien definidas. Esto mejora la seguridad, el mantenimiento y el rendimiento de las aplicaciones al asegurar que sólo se carguen las partes del código y las bibliotecas necesarias para la ejecución.
- **Comunidad y estándares:** Java cuenta con una de las comunidades de desarrolladores más grandes y activas del mundo. Existen numerosos recursos de aprendizaje, foros de discu-

sión, conferencias y grupos de usuarios disponibles para ayudar a quienes desarrollan a resolver problemas, compartir conocimientos y avanzar en sus proyectos. Además, Java se rige por un proceso de evolución transparente y colaborativo (Java Community Process - JCP) que permite a la comunidad contribuir a la dirección y desarrollo del lenguaje.

Estas características, combinadas con una constante evolución y actualización del lenguaje, consolidan a Java como una de las opciones preferidas para desarrolladores en todo el mundo, capaz de adaptarse a las cambiantes demandas del desarrollo de *software* moderno.

Evolución de Java

Desde su lanzamiento inicial como Java 1.0 en 1995, su evolución ha sido un testimonio de su adaptabilidad y relevancia sostenida en el mundo de la tecnología y el desarrollo de *software*. Cada nueva versión de Java ha incorporado mejoras significativas, no s**ó**lo al lenguaje en sí, sino también a la plataforma en general, asegurando que Java permanezca al frente de las innovaciones tecnológicas y satisfaga las necesidades cambiantes de desarrolladores y las empresas. A continuación, se detalla cómo Java ha evolucionado a lo largo de los años, destacando algunas de las adiciones más importantes hasta el año 2024:

- **Java 1.0 (1995):** la primera versión de Java se introdujo con el lema "*Write Once, Run Anywhere*", enfatizando su capacidad de portabilidad. Java 1.0 ya ofrecía un modelo de programación orientado a objetos y un conjunto de bibliotecas que facilitaban el desarrollo de aplicaciones de red y gráficas.
- **Java 5 (2004): añadió genéricos, bucles «***for-each*", enumeraciones (enum), y anotaciones para mejorar la legibilidad del código.
- **Java 8 (2014):** introducción de expresiones lambda y *streams*, que facilitaron la programación funcional. También se agregó la nueva API de *date and time*.

- **Java 9 (2017):** introducción del sistema de módulos (Project Jigsaw), mejoras en la API de Process y JShell (REPL).
- **Java 10 (2018):** Local-Variable Type Inference (var) y mejoras en la recolección de basura.
- **Java 11 (2018):** soporte de larga duración (LTS), se eliminaron módulos obsoletos como Java EE y CORBA, y nuevas funciones de *string*.
- **Java 12 (2019):** expresiones *switch* en vista previa y mejoras en el recolector de basura.
- **Java 13 (2019):** bloques de texto (*text blocks*) como característica previa.
- **Java 14 (2020):** bloques de texto se hicieron estándar, se introdujo la coincidencia de patrones para instancias de *instanceof*.
- **Java 15 (2020):** sellos de clases (*sealed classes*) en vista previa.
- **Java 16 (2021):** registro de clases y coincidencia de patrones para *instanceof*.
- **Java 17 (2021):** LTS, incluyendo sellos de clases, patrones de coincidencia mejorados y un nuevo recolector de basura (G1).
- **Java 18 (2022):** API de interconexión (*Foreign Function and Memory* API) y mejoras de JVM.
- **Java 19 (2022):** mejoras adicionales en el API de interconexión y mejoras en la programación concurrente.
- **Java 20 (2023):** innovaciones de rendimiento, estabilidad y seguridad.
- **Java 21 (2023):** LTS, con nuevas características de lenguaje y mejoras en la plataforma.
- **Java 22 (2024):** nuevas funciones en vista previa y mejoras en las bibliotecas existentes y recolector de basura.

Java se mantiene como uno de los lenguajes de programación más populares y ampliamente utilizados en el mundo. Su robustez, portabilidad y el extenso ecosistema de desarrollo hacen de este lenguaje una excelente alternativa para todo tipo de proyectos, desde aplicaciones móviles hasta soluciones empresariales a gran escala. A medida que se avance en este libro, se explorarr+á en profundidad los fundamentos del lenguaje, sus características y cómo es posible aprovechar estas para construir aplicaciones robustas, eficientes y efectivas.

Instalación del entorno de desarrollo

La instalación del entorno de desarrollo de Java implica varios pasos clave, que varían ligeramente dependiendo del sistema operativo (Windows, macOS o Linux). A continuación, se ofrece una guía detallada para instalar Java, específicamente el Kit de Desarrollo de Java (JDK), que es esencial para desarrollar aplicaciones Java.

Paso 1: Descarga del JDK

1) **Visita la página oficial de Oracle JDK** en Oracle Java o la página de AdoptOpenJDK para obtener una versión de código abierto. Oracle JDK es más utilizado en ambientes empresariales, mientras que AdoptOpenJDK es una opción popular y gratuita para desarrolladores.
2) **Elige la versión de JDK** que necesitas. Java 8 y Java 11 son las versiones de soporte a largo plazo (LTS) más comunes, pero es posible que quieras instalar la última versión para acceder a las características más recientes.
3) **Descarga el instalador** adecuado para tu sistema operativo (Windows, macOS, Linux).

Paso 2: Instalación del JDK

En Windows:

1) **Ejecuta el archivo descargado** para iniciar el instalador.

2) **Sigue las instrucciones** en pantalla. Es recomendable dejar las opciones de instalación por defecto, incluida la ubicación del directorio de instalación.

3) **Finaliza la instalación** y cierra el instalador una vez completado.

En macOS:

1) **Abre el archivo .dmg** descargado para montar el instalador.

2) **Ejecuta el paquete .pkg** dentro del disco montado para iniciar el asistente de instalación.

3) **Sigue las instrucciones** en pantalla, aceptando la licencia y seleccionando el disco de destino si se solicita.

4) **Completa la instalación** y cierra el instalador.

En Linux:

La instalación en Linux varía según la distribución, pero generalmente se realiza a través de la terminal. Para un sistema basado en Debian como Ubuntu, puedes seguir estos pasos:

1) Abre una terminal.

2) **Actualiza el paquete de índices** ejecutando sudo apt-get update.

3) **Instala el JDK** con el comando sudo apt-get install openjdk-11-jdk para instalar Java 11, por ejemplo. Ajusta el número de versión según tus necesidades.

Paso 3: Configuración de la variable de entorno

Configurar la variable de entorno JAVA_HOME es un paso de suma importancia ya que permite que otras aplicaciones, como IDEs y herramientas de desarrollo, localicen el JDK instalado.

En Windows:

1) **Busca "Variables de entorno"** en el menú de inicio y selecciona "Editar las variables de entorno del sistema".

2) **Haz clic en "Variables del sistema …".**

3) **Bajo "Variables del sistema", haz clic en "Nueva…"** para crear una nueva variable.

4) **Establece el nombre de la variable** como JAVA_HOME y el valor como el camino al directorio donde está instalado el JDK, por ejemplo, C:\Program Files\Java\jdk-11.0.1.

5) **Agrega el directorio bin del JDK al PATH** para que puedas ejecutar los comandos de Java desde cualquier lugar. Hazlo seleccionando la variable Path bajo "Variables del sistema", haciendo clic en "Editar…" y agregando %JAVA_HOME%\bin al final.

En macOS y Linux:

4) **Abre tu archivo de perfil** (~/.bash_profile, ~/.zshrc, ~/.profile, etc.) en un editor de texto.

5) **Agrega la siguiente línea** al final del archivo: `export` JAVA_HOME=$(/usr/libexec/java_home) en macOS, o export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64 en Linux (ajusta la ruta según corresponda).

6) **Guarda y cierra** el archivo de perfil.

7) **Aplica los cambios** ejecutando `source` ~/.bash_profile (o el archivo de perfil correspondiente) en la terminal.

Paso 4: Verificación de la Instalación

Para verificar que Java se ha instalado correctamente, abre una terminal o línea de comandos y ejecuta:

```
java -version
```

Si la instalación fue exitosa, verás la versión del JDK, por ejemplo:

```
openjdk version "21.0.1" 2023-10-17 LTS
OpenJDK Runtime Environment Temurin-21.0.1+12 (build 21.0.1+12-LTS)
OpenJDK 64-Bit Server VM Temurin-21.0.1+12 (build 21.0.1+12-LTS, mixed mode, sharing)
```

Tu primer programa en Java

Crear tu primer programa en Java, comúnmente llamado "Hola Mundo", es un paso emocionante hacia el aprendizaje del lenguaje de programación Java. Este programa simplemente imprimirá el mensaje "Hola Mundo" en la consola, demostrando los conceptos básicos de sintaxis y ejecución en Java. Los pasos detallados para crear y ejecutar este programa son los siguientes:

Paso 1: Crear el archivo fuente de Java

8) **Abre un editor de texto:** puedes usar cualquier editor de texto simple como Notepad en Windows, TextEdit en macOS configurado en modo de texto plano, o editores más avanzados como Visual Studio Code, Sublime Text, o Atom.

9) **Escribe el código del programa:** Copia el siguiente código en tu editor. Este es el código fuente para un simple programa de "Hola Mundo" en Java.

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

Este código define una clase pública llamada HolaMundo, que contiene un método *main*. Este método *main* es el punto de entrada (*entry point*) de cualquier aplicación Java. Dentro del *main*, utiliza la instrucción System.out.println que sirve para imprimir el mensaje "Hola Mundo" en la consola.

1. **Guarda el archivo:** Guarda el archivo con el nombre HolaMundo.java. Es importante que el nombre del archivo coincida exactamente con el nombre de la clase pública en tu código.

Paso 2: Compilar el programa

Para compilar el programa, necesitas usar el compilador javac, que viene con el JDK. Sigue estos pasos:

10) **Abre una terminal o línea de comandos:** en Windows, puedes buscar "cmd" en el menú de inicio. En macOS o Linux, puedes abrir la terminal.

11) **Navega hasta el directorio donde guardaste el archivo:** Usa el comando cd para cambiar al directorio donde guardaste HolaMundo.java. Por ejemplo, si lo guardaste en un directorio llamado "Java" en tu escritorio, deberías usar un comando similar a:

```
cd Desktop/Java
```

12) **Compila el programa:** Ejecuta el siguiente comando para compilar tu programa. El compilador javac leerá tu archivo HolaMundo.java, lo compilará y, si no hay errores, creará un archivo de clase ejecutable llamado HolaMundo.class.

```
javac HolaMundo.java
```

Si el proceso de compilación se completa sin mostrar mensajes de error, significa que tu programa se ha compilado con éxito.

Paso 3: Ejecutar el programa

Una vez compilado el programa, puedes ejecutarlo con el intérprete de Java. Sigue estos pasos para ejecutar tu programa:

- **Ejecuta el programa:** asegurándote de que aún estás en el mismo directorio donde se encuentra el archivo HolaMundo.class, ejecuta el siguiente comando:

```
java HolaMundo
```

- **Verifica la salida:** si todo ha ido bien, verás el mensaje "Hola Mundo" impreso en la consola. Esto indica que tu programa se ha ejecutado correctamente.

¡Has completado tu primer programa en Java! Este es un paso fundamental en tu camino para aprender este lenguaje. A partir de aquí, vamos a comenzar a explorar más características como variables, tipos de datos, estructuras de control, clases y objetos. Experimenta modificando tu programa para imprimir diferentes mensajes o realizar operaciones simples para familiarizarte más con la sintaxis del lenguaje.

Conclusiones

Java ha demostrado ser un lenguaje de programación excepcionalmente versátil y robusto desde su creación en 1991. Su filosofía "*Write Once, Run Anywhere*" ha permitido una amplia adopción en diversas industrias, facilitando la creación de aplicaciones multiplataforma. Bajo la gestión de Oracle, Java ha continuado evolucionando con actualizaciones frecuentes, asegurando su relevancia en el desarrollo de *software* moderno. La combinación de su orientación a objetos, seguridad inherente, portabilidad y un ecosistema de herramientas y bibliotecas extensas, lo convierte en una elección preferida por desarrolladores para una amplia gama de aplicaciones, desde sistemas empresariales a aplicaciones móviles.

Documentos consultados

Gosling, J., Joy, B., Steele, G., Bracha, G. y Buckley, A. (2014). *The Java Language Specification, Java SE* (8 Edition). Addison-Wesley. https://docs.oracle.com/javase/specs/jls/se8/html/index.html

Oracle. (s.f.a). *The Java™ Tutorials. Oracle*. https://docs.oracle.com/javase/tutorial/

Oracle. (s.f.b). *Java SE Documentation*. https://docs.oracle.com/en/java/javase/11/

Oracle. (s.f.c). *Java Platform, Standard Edition Installation Guide*. https://docs.oracle.com/en/java/javase/11/install/

Capítulo 2. Elementos básicos de un programa en Java

Introducción

Antes de adentrarnos en la creación de programas complejos en Java, es necesario comprender los elementos básicos que sustentan cualquier aplicación: la sintaxis, los comentarios y las variables con sus tipos de datos. Java es un lenguaje orientado a objetos y de tipado estático, lo que asegura que el código sea claro, seguro y conforme a reglas específicas para la ejecución en la JVM. Este capítulo abordará la sintaxis fundamental de Java, la relevancia de los comentarios para la legibilidad del código, y el uso de variables y tipos de datos, proporcionando ejemplos y ejercicios prácticos para fortalecer tu comprensión de estos conceptos esenciales.

Sintaxis básica de los programas en Java

La sintaxis de un lenguaje de programación es el conjunto de reglas que define cómo escribir instrucciones que serán interpretadas por la computadora. Java, siendo un lenguaje de programación de alto nivel, ofrece una sintaxis que permite a los desarrolladores escribir programas de manera clara y concisa. Comprender la sintaxis básica es fundamental para iniciar en la programación con Java. En esta sección, exploraremos los elementos fundamentales de la sintaxis de Java.

Estructura de un programa Java

Un programa Java típico consta de al menos una clase que contiene un método *main*. Este método es el punto de entrada del programa, donde la ejecución de este inicia. La estructura básica de un programa Java se ve así:

```
public class NombreDeLaClase {
    public static void main(String[] args) {
        // Código del programa
    }
}
```

- **Clases:** en Java, todo reside dentro de clases. Una clase es una especie de plantilla que define las propiedades (variables) y comportamientos (métodos) que los objetos de esa clase tendrán. El nombre de la clase debe coincidir con el nombre del archivo. Por ejemplo, si tu clase se llama MiPrimerPrograma, el archivo debe llamarse MiPrimerPrograma.java.
- **El método *main*:** es el punto de entrada de cualquier programa Java. La firma del método debe ser siempre public static void main(String[] args). Java es un lenguaje fuertemente tipado, lo que significa que debes declarar el tipo de todas las variables. Aquí, String[] args declara un parámetro args, que es un arreglo de objetos String, utilizado para recibir argumentos de la línea de comandos. En Java 21 están considerando realizar cambios en esta sintaxis, pero lo más conveniente es conservarla.
- **{ }:** las llaves delimitan bloques de código en Java. En este ejemplo, encierran el cuerpo de la clase NombreDeLaClase y el método *main*, definiendo los límites de donde comienza y termina cada bloque de código. Todo el código relevante para la clase y el método debe estar contenido dentro de estas llaves.
- **//código del programa:** Los símbolos de comentario // se utilizan para agregar notas dentro del código que no serán ejecu-

tadas por la JVM. En este caso, indica que aquí es donde iría el código que ejecutará el programa. Los comentarios son útiles para explicar la lógica del código o para dejar recordatorios hacia el futuro.

Sensibilidad y nomenclatura de clases

- **Sensibilidad a mayúsculas y minúsculas:** Java es sensible a mayúsculas y minúsculas, lo que significa que *Main*, *main*, y *MAIN* son identificadores diferentes.
- **Nombres de clases:** por convención, los nombres de clases en Java comienzan con mayúscula y siguen el estilo camelCase (la primer letra de cada palabra es mayúscula).
- **Nombres de métodos y variables:** los nombres de métodos y variables deben comenzar con una letra minúscula y seguir el estilo camelCase.

Bloques de código

En Java, los bloques de código son secciones de código que están delimitadas por llaves {}. Por ejemplo, el cuerpo de un método, un bucle o una condición se define utilizando llaves.

```
if (condicion) {
    // bloque de código
}
```

Punto y coma (;)

Cada instrucción en Java termina con un punto y coma (;). Este signo marca el fin de la declaración, permitiendo al compilador saber que esa línea de código ha concluido.

```
int num = 10; // Declaración de una variable con asignación
System.out.println(num); // Llamada al método para imprimir el
valor de la variable
```

La sintaxis básica de Java establece la estructura sobre la cual se construyen todos los programas Java. Comprender estos elementos fundamentales es esencial para empezar a escribir programas eficaces y eficientes en Java. A medida que se avance, se explorará cómo utilizar estas estructuras básicas para crear programas más complejos y funcionales.

Comentarios

Los comentarios en el código son líneas que el compilador ignora, pero que proporcionan información valiosa para las y los desarrolladores. En Java, existen principalmente tres tipos de comentarios: de línea, de bloque y de documentación. Usar comentarios es importante para mantener el código legible, mantenible y amigable para otros desarrolladores, incluido uno mismo en el futuro.

Comentarios de línea

Los comentarios de línea comienzan con // y se extienden hasta el final de la línea. Son útiles para breves anotaciones o para desactivar temporalmente una línea de código.

```
// Esto es un comentario de línea que explica la siguiente instrucción
int num = 10;
```

Comentarios de bloque

Los comentarios de bloque se delimitan con /* al inicio y */ al final. Son adecuados para descripciones más extensas o para comentar múltiples líneas de código simultáneamente.

```
/*
 Este es un comentario de bloque. Puede abarcar varias líneas y es útil
para
 proporcionar descripciones detalladas del código siguiente o para
 comentar secciones completas de código durante las pruebas o la
depuración.
*/
int num = 10;
```

Comentarios de documentación

Los comentarios de documentación, o comentarios Javadoc, comienzan con /** y terminan con */. Estos se utilizan para generar documentación automática del código y deben colocarse antes de las definiciones de clases, interfaces, constructores, métodos y campos (en capítulos posteriores aprenderás sobre estos). Los comentarios Javadoc pueden incluir etiquetas que proporcionan información estructurada, como descripciones de parámetros de método, valores de retorno y excepciones.

```
/**
 * Calcula y devuelve la suma de dos números.
 *
 * @param num1 El primer número a sumar.
 * @param num2 El segundo número a sumar.
 * @return La suma de num1 y num2.
 */
public int sumar(int num1, int num2) {
    return num1 + num2;
}
```

Mejores prácticas para el uso de comentarios

- **No sobrecomentar:** evita comentar código que es claro por sí mismo. El uso excesivo de comentarios puede hacer que el código sea difícil de leer. En su lugar, escribe código autoexplicativo mediante el uso de nombres de variables y métodos descriptivos.
- **Mantén los comentarios actualizados:** un comentario desactualizado puede ser más perjudicial que no tener comentarios. Asegúrate de revisar y actualizar los comentarios cuando cambies el código.
- **Utiliza comentarios para explicar el "por qué":** Mientras que el código explica el "qué" y el "cómo", los comentarios deben

usarse para explicar el "por qué". Esto es especialmente útil para decisiones de diseño no obvias o complejas.

- **Documenta las interfaces p**úblicas: utiliza comentarios Javadoc para documentar las interfaces públicas de tus clases. Esto incluye clases, interfaces, métodos públicos y campos públicos. Proporciona una descripción clara de lo que hace el elemento, sus parámetros, valores de retorno y cualquier excepción que pueda lanzar.
- **Evita comentarios obsoletos:** refactoriza el código para hacerlo más claro en lugar de explicar lógica complicada con comentarios. Si encuentras bloques de código comentados durante la depuración o pruebas, elimínalos antes de considerar el trabajo completo.

Los comentarios son una herramienta valiosa para mejorar la legibilidad y mantenibilidad del código. Al seguir estas prácticas recomendadas, puedes asegurarte de que estos contribuyan positivamente al desarrollo y mantenimiento de tus aplicaciones.

Variables

Una variable es un contenedor de memoria que almacena datos que pueden cambiar durante la ejecución del programa. Cada variable en Java tiene un tipo de dato que determina el tamaño y la forma de su contenido de memoria, así como el conjunto de operaciones que se pueden realizar sobre ella. En el siguiente capítulo se abordarán de manera detallada los distintos tipos de datos de Java.

Declaración de variables

Para utilizar una variable en Java, primero debes declararla especificando su tipo de dato seguido por un nombre único:

```
tipo nombreVariable;
```

Por ejemplo:

```
int edad;
double salario;
char letraInicial;
String nombre;
```

Aquí int, double, char y String son los tipos de datos.
Para nombrar una variable hay que tener algunas consideraciones:

- **Nombres de variables:** Los nombres de las variables deben comenzar con una letra (a-z, A-Z) o un guion bajo (_), seguido de letras, dígitos (0-9) o guiones bajos. Los nombres de variables son sensibles a mayúsculas y minúsculas. Recordemos que la recomendación siempre inicie con una letra minúscula.
- **Palabras reservadas:** No puedes usar palabras reservadas de Java como nombres de variables. Por ejemplo, int, for, class, etc.

Inicialización de variables

Puedes inicializar una variable en su declaración, asignándole un valor inicial:

```
int edad = 30;
double salario = 4550.50;
char letraInicial = 'J';
String nombre = "Juan";
```

O declararla y posteriormente asignarle su valor:

```
int edad;
.
.
.
edad = 30;
```

Este capítulo establece las bases para entender cómo se estructuran y funcionan los programas en Java, enfatizando la importancia de la claridad, la organización y el manejo eficiente de datos. Con ejemplos prácticos y ejercicios, esperamos que te familiarices con estos conceptos

fundamentales, preparándote para temas más avanzados en la programación Java.

Conclusiones

Este Capítulo proporciona una comprensión esencial de los elementos básicos que constituyen un programa en Java, desde la sintaxis hasta las variables y tipos de datos. La clara estructura de la sintaxis de Java facilita la creación de código legible y mantenible, mientras que el uso efectivo de comentarios mejora la comprensión y colaboración en el desarrollo de *software*. La importancia de las variables y los tipos de datos se subraya como fundamentales para la manipulación de información dentro de un programa. Este capítulo sienta una base sólida para avanzar hacia conceptos más complejos en la programación Java, asegurando que las y los desarrolladores puedan construir aplicaciones eficientes y efectivas con una comprensión clara de los bloques de construcción fundamentales.

Documentos consultados

Bloch, J. (2018). *Effective Java* (3rd Edition). Addison-Wesley. https://www.pearson.com/store/p/effective-java/P100000281093/9780134686097

Gosling, J., Joy, B., Steele, G., Bracha, G. y Buckley, A. (2014). *The Java Language Specification, Java SE* (8th Edition). Addison-Wesley. https://docs.oracle.com/javase/specs/jls/se8/html/index.html

Oracle. (s.f.a). *The Java™ Tutorials - Learning the Java Language*. https://docs.oracle.com/javase/tutorial/java/index.html

Oracle. (s.f.b). *Java SE Documentation*. https://docs.oracle.com/en/java/javase/11/

Oracle. (s.f.c). *Java Platform, Standard Edition Installation Guide*. https://docs.oracle.com/en/java/javase/11/install/

OpenJDK. (s.f.). OpenJDK Documentation. https://openjdk.java.net/
Schildt, H. (2018). Java: The Complete Reference (11th Edition). McGraw-Hill Education. https://www.mhprofessional.com/9781260440232-usa-ja-va-the-complete-reference-eleventh-edition-group
Sierra, K. y Bates, B. (2005). Head First Java (2nd Edition). O'Reilly Media. https://www.oreilly.com/library/view/head-first-java/0596009208/

Capítulo 3. Tipos de datos

Introducción

En este capítulo, se exploran los fundamentos de los tipos de datos en Java, esenciales para cualquier desarrollo en este lenguaje. Java clasifica sus tipos de datos en dos categorías principales: primitivos y no primitivos (o tipos de referencia). Los tipos de datos primitivos incluyen enteros, números de punto flotante, caracteres y booleanos, cada uno con sus propias características y usos específicos. Estos tipos básicos son importantes para el almacenamiento y manipulación de información en un programa. Por otro lado, los tipos de datos no primitivos, como clases, interfaces, arreglos y enumeraciones, permiten la creación de estructuras de datos más complejas y son fundamentales para la programación orientada a objetos. A lo largo de este capítulo, profundizaremos en cada uno de estos tipos, proporcionando ejemplos claros y concisos para ilustrar su uso y funcionalidad en Java, sentando así una base sólida para el desarrollo de aplicaciones más avanzadas.

Tipos de datos primitivos

Antes de estudiar los tipos de datos primitivos en Java, es útil comprender los conceptos básicos de *bits* y *bytes*. Un *bit* es la unidad más pequeña de datos en la computación, también conocida como la unidad mínima de información, que puede tener un valor de 0 o 1. Un *byte* es un conjunto de 8 *bits* y es la unidad estándar para medir la memoria en los sistemas computacionales. Los tipos de datos en Java se definen en términos de *bits* y *bytes*, determinando así la cantidad de memoria que ocupan y su rango de valores.

Los tipos de datos primitivos de Java incluyen:

- **Numéricos enteros:** *byte* (8 bits), *short* (16 *bits*), *int* (32 *bits*), *long* (64 *bits*).
- **Numéricos de punto flotante:** *float* (32 *bits*), *double* (64 *bits*).
- **Caracter:** *char* (16 *bits*, para almacenar caracteres Unicode).
- **Booleano:** boolean (para valores verdaderos o falsos).

Numéricos enteros

- ***Byte*:** es el tipo de dato más pequeño, con un tamaño de 8 *bits* (1 *byte*), con un rango de -128 a 127. Es útil para ahorrar memoria en matrices grandes, por ejemplo.

```
byte edad = 30;
System.out.println("Edad: " + edad);
```

- ***Short*:** un tipo de dato de 16 *bits* (2 *bytes*) con un rango de -32,768 a 32,767. Adecuado para números pequeños que no caben en un *byte*.

```
short year = 2020;
System.out.println("Año: " + year);
```

- ***Int*:** es el tipo de dato más utilizado para números enteros, con un tamaño de 32 bits (4 bytes) y un rango de -2^31 a 2^31-1.

```
int poblacion = 150000;
System.out.println("Población: " + poblacion);
```

- ***Long*:** para números enteros muy grandes, utiliza 64 *bits* (8 *bytes*) y tiene un rango de -2^63 a 2^63-1. Los literales *long* se denotan con una L al final.

```
long distanciaALaLuna = 384400000L;
System.out.println("Distancia a la Luna: " + distanciaALaLuna + "
metros");
```

Numéricos de punto flotante

- ***Float*:** representa números con decimales, usa 32 *bits* (4 *bytes*). Los literales *float* se denotan con una f al final.

```
float salario = 12345.67f;
System.out.println("Salario: " + salario);
```

- ***Double*:** También representa números con decimales pero es más preciso, utilizando 64 *bits* (8 *bytes*). Es el tipo predeterminado para decimales.

```
double pi = 3.141592653589793;
System.out.println("Valor de Pi: " + pi);
```

Caracter

- ***Char*:** utilizado para almacenar un único carácter Unicode. Ocupa 16 *bits* (2 *bytes*) y puede representar un rango de 0 a 65,535.

```
char letraInicial = 'J';
System.out.println("Letra Inicial: " + letraInicial);
```

Booleano

- **Boolean:** representa dos valores: *true* o *false*, sólo almacena estos dos valores.

```
boolean estaAprobado = true;
System.out.println("¿Está aprobado? " + estaAprobado);
```

Tipos de datos no primitivos

Los tipos de datos no primitivos en Java, también conocidos como tipos de referencia, son fundamentales para la construcción de aplicaciones complejas y modulares. A diferencia de los tipos primitivos, que almacenan valores directamente, los tipos no primitivos almacenan re-

ferencias a un espacio en la memoria donde se guarda el valor. Esto permite a los tipos no primitivos, como clases, interfaces, arreglos y enumeraciones, representar estructuras de datos más complejas y relaciones entre datos. Entre estos, *String* es probablemente el tipo de objeto más utilizado y merece una atención especial.

Clases

Las clases son el núcleo de la programación orientada a objetos en Java. Permiten a los desarrolladores crear objetos que encapsulan datos y comportamientos, facilitando la modularidad y la reutilización del código. Cada objeto es una instancia de una clase y puede tener atributos (para almacenar datos) y métodos (para definir comportamientos o funciones que puede realizar).

Interfaces

Las interfaces definen un contrato que las clases pueden implementar. Son colecciones de métodos abstractos (sin implementación) que especifican qué se debe hacer, pero no cómo hacerlo. Las interfaces son fundamentales para lograr un alto nivel de abstracción y para separar la definición de las operaciones de sus implementaciones concretas.

Arreglos

Los arreglos son contenedores que almacenan una cantidad fija de elementos de un solo tipo. Permiten agrupar múltiples elementos en una única estructura de datos, accesibles por un índice. Aunque los arreglos tienen una longitud fija una vez creados, son útiles para manejar colecciones de elementos donde la longitud se conoce de antemano.

Enumeraciones (Enums)

Las enumeraciones son un tipo especial de clase en Java que define un conjunto de constantes. Mejoran la legibilidad del código y reducen los errores al restringir los valores que una variable puede tomar a los definidos en la enumeración.

El tipo *String*

El *String* en Java es un objeto que representa una secuencia de caracteres. A diferencia de los tipos primitivos, *String* es inmutable, lo que significa que, una vez creado, el valor de un objeto *String* no puede cambiar. Java ofrece una amplia gama de operaciones para trabajar con cadenas de texto, como concatenación, comparación, búsqueda y extracción de subcadenas.

Ejemplo de uso de *String*:

```
String saludo = "Hola, ";
String nombre = "Mundo";
String mensaje = saludo + nombre; // Concatenación de cadenas
System.out.println(mensaje); // Imprime "Hola, Mundo" en la
terminal
```

Aunque *String* es técnicamente un tipo de dato no primitivo, su uso es tan integrado en el lenguaje Java que a menudo se manipula con la misma facilidad que los tipos primitivos. Además de *String*, Java ofrece otras clases envolventes para representar tipos primitivos como objetos, como *Integer* para *int*, *Double* para *double*, y así sucesivamente. Estas clases envolventes son útiles para utilizar en colecciones genéricas y para acceder a métodos útiles para manipular los datos.

Comprender los tipos de datos y saber cómo declarar e inicializar variables son fundamentos esenciales de la programación. A medida que te familiarices más con estos conceptos, serás capaz de utilizar variables para almacenar y manipular datos de manera efectiva en tus programas. Para profundizar en los tipos de datos primitivos de Java, exploraremos cada uno con ejemplos individuales, destacando sus características y cómo se utilizan en la programación Java.

Conclusiones

Este capítulo ofrece una comprensión detallada de los tipos de datos en Java, clasificándolos en primitivos y no primitivos. Los tipos de datos primitivos proporcionan la base para manejar datos numéricos, caracteres y booleanos, esenciales para las operaciones básicas en cualquier programa. Los tipos de datos no primitivos, como clases, interfaces, arreglos y enumeraciones, permiten una mayor complejidad y modularidad en el diseño de *software*, facilitando la programación orientada a objetos. Comprender y utilizar adecuadamente estos tipos de datos es fundamental para escribir código eficiente, mantenible y robusto en Java. Esta base sólida es de suma importancia para avanzar hacia el desarrollo de aplicaciones más complejas y sofisticadas.

Documentos consultados

Bloch, J. (2018). *Effective Java* (3rd Edition). Addison-Wesley. https://www.pearson.com/store/p/effective-java/P100000281093/9780134686097

Gosling, J., Joy, B., Steele, G., Bracha, G. y Buckley, A. (2014). *The Java Language Specification, Java SE* (8th Edition). Addison-Wesley. https://docs.oracle.com/javase/specs/jls/se8/html/index.html

Oracle. (n.d.a). *The Java™ Tutorials - Learning the Java Language: Primitive Data Types*. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Oracle. (n.d.b). *Java SE Documentation*. https://docs.oracle.com/en/java/javase/11/

Oracle. (n.d.c). *Java Platform, Standard Edition: Interfaces and Inheritance*. https://docs.oracle.com/javase/tutorial/java/IandI/

Oracle. (n.d.d). *Java™ Platform, Standard Edition API Specification.* https://docs.oracle.com/en/java/javase/11/docs/api/

Schildt, H. (2018). *Java: The Complete Reference* (11th Edition). McGraw-Hill Education. https://www.mhprofessional.com/9781260440232-usa-java-the-complete-reference-eleventh-edition-group

Sierra, K. y Bates, B. (2005). *Head First Java* (2nd Edition). O'Reilly Media. https://www.oreilly.com/library/view/head-first-java/0596009208/

Capítulo 4. Entrada y salida de datos desde la consola

Introducción

La capacidad de interactuar con las personas usuarias a través de la Entrada y Salida (E/S) de datos es fundamental en la mayoría de los programas. Java ofrece varias maneras de manejar la entrada y salida, y una de las más comunes y directas es a través de la consola. Este capítulo se centra en cómo leer datos del teclado y cómo mostrar información al usuario/a, utilizando tanto tipos de datos primitivos como del tipo *String*.

Entrada de datos desde el teclado

Para leer la entrada de datos desde el teclado en una aplicación de consola de Java, se utiliza la clase *Scanner* del paquete java.util. La clase *Scanner* proporciona métodos que pueden leer diferentes tipos de datos primitivos y *strings*.

Primero, debes importar la clase *Scanner* en tu programa:

```
import java.util.Scanner;
```

Luego, puedes crear una instancia de *Scanner* (variable) asociada con la entrada estándar (en este caso, el teclado):

```
Scanner scanner = new Scanner(System.in);
```

A continuación, se muestran ejemplos de cómo leer diferentes tipos de datos desde el teclado y cómo mostrar esa entrada en la pantalla.

Ejemplo con **int**

```
import java.util.Scanner;

public class EntradaInt {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Introduce un número entero: ");
        int num = scanner.nextInt();
        System.out.println("Has introducido: " + num);
    }
}
```

Este programa lee un número entero desde el teclado y luego muestra ese número en la pantalla. Este proceso se realiza en varios pasos clave:

1) **Importación de la clase *Scanner*:** import java.util.Scanner; Este código importa la clase *Scanner* del paquete java.util, que es necesaria para leer la entrada del usuario/a desde la consola.

2) **Creación de un objeto *Scanner*:** Scanner scanner = new Scanner(System.in). Aquí se crea una instancia de la clase Scanner llamada scanner. Esta línea asocia el nuevo objeto Scanner con la entrada estándar del sistema (el teclado), permitiendo leer datos introducidos por la o el usuario.

3) **Solicitud de entrada al usuario/a:** System.out.print("Introduce un número entero: "). Antes de leer la entrada del usuario/a, el programa muestra un mensaje en la pantalla solicitándole que introduzca un número entero. Esto se hace para hacer que la interfaz sea interactiva y para indicar al usuario/a qué tipo de dato debe ingresar.

4) **Lectura de un número entero:** int num = scanner.nextInt(); Esta línea utiliza el método nextInt() del objeto scanner para leer un número entero introducido por el usuario/a. El número leído se almacena en la variable num.

5) **Mostrar el número introducido:** System.out.println("Has introducido: " + num). Finalmente, el programa imprime el valor de la variable num, mostrando el número entero que la o el usuario ha introducido.

Ejemplo con *double*

```
import java.util.Scanner;

public class EntradaDouble {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Introduce un número decimal: ");
        double decimal = scanner.nextDouble();
        System.out.println("Has introducido: " + decimal);
        scanner.close()
    }
}
```

Al igual que en el programa anterior, en este se lee un número desde el teclado, pero ahora es *double*, por lo que es necesario ajustar la lectura del número mediante el método nextDouble() del objeto *Scanner*. El número leído se almacena en la variable decimal y se muestra en la consola el número *double* leído.

Ejemplo con *String*

```
import java.util.Scanner;

public class EntradaString {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Introduce tu nombre: ");
        String nombre = scanner.nextLine();
        System.out.println("Hola, " + nombre + "!");
    }
}
```

Este programa emplea el método nextLine() de la clase *Scanner* para leer una línea completa de entrada proporcionada por el usuario a través de la consola. Este método captura todo el texto ingresado hasta

que se presiona *Enter*, incluyendo espacios, permitiendo así a la o el usuario introducir su nombre completo si así lo desea. Después de leer el nombre, el programa lo utiliza para saludar al usuario/a, mostrando un mensaje personalizado en la consola que incluye el nombre introducido.

A continuación, se presenta una tabla que resume los métodos más comunes de la clase *Scanner* en Java, que son útiles para la entrada de datos de diferentes tipos desde la consola:

Tabla 1. Métodos comunes de la clase *Scanner*

Método	Descripción
next()	Lee el siguiente *token* de entrada como una cadena de texto (*String*).
nextLine()	Lee el resto de la línea actual, incluyendo el salto de línea.
nextInt()	Lee el siguiente *token* de entrada como un *int*.
nextLong()	Lee el siguiente *token* de entrada como un *long*.
nextDouble()	Lee el siguiente *token* de entrada como un *double*.
nextFloat()	Lee el siguiente *token* de entrada como un *float*.
nextByte()	Lee el siguiente *token* de entrada como un *byte*.
nextShort()	Lee el siguiente *token* de entrada como un *short*.
nextBoolean()	Lee el siguiente *token* de entrada como un boolean.
hasNext()	Comprueba si hay un siguiente *token* de entrada disponible.
hasNextLine()	Comprueba si hay una siguiente línea en la entrada.
hasNextInt()	Comprueba si el siguiente *token* de entrada puede interpretarse como un *int*.
hasNextLong()	Comprueba si el siguiente *token* de entrada puede interpretarse como un *long*.
hasNextDouble()	Comprueba si el siguiente *token* de entrada puede interpretarse como un *double*.
hasNextFloat()	Comprueba si el siguiente *token* de entrada puede interpretarse como un *float*.

Continúa en la página 41...

Método	Descripción
hasNextByte()	Comprueba si el siguiente *token* de entrada puede interpretarse como un *byte*.
hasNextShort()	Comprueba si el siguiente *token* de entrada puede interpretarse como un *short*.
hasNextBoolean()	Comprueba si el siguiente *token* de entrada puede interpretarse como un boolean.

Fuente: Elaboración propia.

Estos métodos facilitan la lectura de diferentes tipos de datos desde la entrada estándar, permitiendo la creación de aplicaciones interactivas que requieren datos del usuario o usuaria. Es importante cerrar el objeto *Scanner* cuando ya no se necesite, llamando al método close(), para liberar los recursos que pudiera estar ocupando.

La interacción con la o el usuario mediante la entrada y salida de datos es una parte importante de muchos programas. Utilizando la clase Scanner para la entrada y los métodos de System.out para la salida, puedes crear programas Java que sean interactivos y capaces de comunicarse efectivamente con el usuario. Los ejemplos proporcionados en este capítulo te darán una base sólida para manejar diferentes tipos de datos en tus aplicaciones Java.

Conclusiones

El Capítulo IV proporcionó una guía integral sobre cómo manejar la entrada y salida de datos desde la consola en Java, utilizando la clase *Scanner* para capturar datos del usuario/a y los métodos de System.out para mostrar información. La capacidad de interactuar con el usuario es esencial en muchas aplicaciones, y este capítulo cubre cómo leer y mostrar diferentes tipos de datos primitivos y cadenas de texto, proporcionando ejemplos claros y prácticos. Además, se resaltan los métodos más comunes de la clase *Scanner* y sus usos específicos, así como la importancia de cerrar el objeto *Scanner* después de su uso para liberar recursos. Con esta comprensión, las y los desarrolladores están mejor

equipados para crear programas interactivos y eficaces que requieran la entrada del usuario/a, sentando una base sólida para el desarrollo de aplicaciones más avanzadas en Java.

Documentos consultados

Barclay, K. y Savage, J. (2007). Simple input and output. En B. Krämer (Ed.), *Groovy Programming* (pp. 39-45). https://consensus.app/papers/simple-input-output-barclay/b66085ba643a5389b0f573d618c0f12b/

Cross, J. H. II. (2012). Using the new jGRASP canvas of dynamic viewers for program understanding and debugging in Java courses. *Journal of Computing Sciences in Colleges*, *28*(3), 141-142. https://consensus.app/papers/using-viewers-program-understanding-debugging-java-cross/d4610c3876365349ac96fed9a08c8a47/

Huimin, L. (2009). Java console data capture methods and comparison. Computer Programming Skills & Maintenance. *Consensus*. https://consensus.app/papers/java-console-data-capture-methods-comparison-huimin/8905f40b1b2256889d359d3311f0ae00/

Li, W., Gao, K. y Geng, X. (2018). *Java*程序设计中的键盘输入数据的方法分析 *[Method analysis of input data by keyboard in Java programming]. Computer Science and Application*. https://consensus.app/papers/java程序设计中的键盘输入数据的方法分析-method-analysis-input-data-李蔚妍/c89ccfee1a815bd8be99b2a1d7194899/

Parsons, D. (2020). Input and output streams. En *Texts in Computer Science*. https://consensus.app/papers/input-output-streams-parsons/3d727a80553b509a9176070753f04d4f/

Capítulo 5. Operadores en Java

Introducción

En el desarrollo de *software*, los operadores son fundamentales para manipular variables y valores. Java ofrece una variedad de operadores que se agrupan en diferentes categorías: aritméticos, relacionales, lógicos y condicionales. Los operadores aritméticos realizan cálculos matemáticos básicos, mientras que los relacionales comparan valores y devuelven resultados booleanos. Los operadores lógicos combinan o invierten condiciones booleanas, y el operador condicional permite decisiones rápidas dentro de expresiones. Este capítulo profundiza en cada tipo de operador, mostrando cómo utilizarlos eficazmente para controlar la lógica y manipular datos en programas realizados con Java.

Operadores aritméticos

Estos operadores son utilizados para realizar operaciones matemáticas básicas entre variables y/o valores, pueden operar con cualquier tipo de datos numéricos, incluyendo *int*, *long*, *float*, y *double*. La lista de operadores aritméticos incluye la adición (+), la sustracción (-), la multiplicación (*), la división (/), y el módulo (%). Además, el operador de incremento (++) y el operador de decremento (--) son utilizados para incrementar o disminuir el valor de una variable en una unidad, respectivamente.

A continuación, se presentan ejemplos que ilustran el uso de cada uno de estos operadores:

Adición (+)

Suma dos valores.

```
int suma = 5 + 3; // suma es 8
```

Sustracción (-)

Resta el segundo valor del primero.

```
int resta = 5 - 3; // resta es 2
```

Multiplicación (*)

Multiplica dos valores.

```
int producto = 5 * 3; // producto es 15
```

División (/)

Divide el primer valor entre el segundo. Es importante recordar que la división entre enteros trunca el resultado.

```
int divisionEntera = 5 / 3; // divisionEntera es 1
double divisionDecimal = 5.0 / 3.0; // divisionDecimal es
1.6666666666666667
```

Módulo (%)

Devuelve el residuo de dividir el primer valor entre el segundo.

```
int modulo = 5 % 3; // modulo es 2
```

Incremento (++)

Incrementa el valor de una variable en una unidad. Puede ser usado en forma prefija (++variable) o sufija (variable++), afectando cuándo se realiza el incremento durante la evaluación de expresiones.

```
int contador = 0;
contador++; // contador es 1
++contador; // contador es 2
```

Decremento (--)

Disminuye el valor de una variable en una unidad. Al igual que el incremento, puede ser prefijo o sufijo.

```
int contador = 2;
contador--; // contador es 1
--contador; // contador es 0
```

Prioridad de operadores

La prioridad de operadores en Java determina el orden en el que se evalúan las expresiones que contienen múltiples operadores. Los operadores con mayor prioridad se evalúan antes que aquellos con menor prioridad. En caso de operadores con la misma prioridad, la evaluación se realiza de izquierda a derecha, excepto para los operadores de asignación, que se evalúan de derecha a izquierda.

Aquí tienes una tabla de prioridad de operadores en Java, comenzando por los de mayor prioridad:

Tabla 1. Prioridad de operadores en Java

Prioridad	Operador	Descripción
1	++, --, +, -, ~, !	Incremento y decremento unario, más y menos unario, complemento a uno, negación lógica
2	*, /, %	Multiplicación, división, módulo
3	+, -	Suma, resta
4	<<, >>, >>>	Desplazamiento binario
5	<, <=, >, >=, *instanceof*	Comparaciones, *instanceof*
6	==, !=	Igualdad, desigualdad
7	&	AND binario
8	^	XOR binario
9	|	OR binario
10	&&	AND lógico

Continúa en página 46...

Prioridad	Operador	Descripción
11	||	OR lógico
12	? :	Operador ternario (condicional)
13	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=	Asignaciones

Fuente: Elaboración propia.

Es importante considerar esta tabla de prioridad al escribir expresiones complejas para asegurar que se evalúen como se espera. En caso de duda o para hacer más clara la intención, se recomienda utilizar paréntesis para agrupar explícitamente las operaciones y sobrescribir la prioridad predeterminada.

Ejemplo integrado

Véase un ejemplo que integra varios operadores aritméticos en una operación más compleja:

```
int resultado = ((8 + 2) - (3 * 2)) / 2; // Primero multiplica, luego suma, resta y finalmente divide.
// resultado es 2
```

Para resolver esta expresión podemos descomponerla de la siguiente manera:

1. **Operaciones entre paréntesis:** primero, se evalúan las operaciones dentro de los paréntesis debido a su mayor prioridad o precedencia.
   - (8 + 2) evalúa a 10.
   - (3 * 2) evalúa a 6.
2. **Sustracción:** después, se realiza la operación de sustracción - entre los resultados de las operaciones dentro de los paréntesis.
   - 10 - 6 evalúa a 4.

3. **División:** finalmente, se divide el resultado de la sustracción entre 2.
   - 4 / 2 evalúa a 2.

Este ejemplo demuestra cómo se pueden combinar los operadores aritméticos para realizar cálculos complejos, siguiendo las reglas de precedencia de operadores (primero multiplicación y división, luego adición y sustracción) y cómo se pueden utilizar paréntesis para modificar esta precedencia y agrupar operaciones.

Operadores relacionales

Los operadores relacionales en Java se utilizan para comparar dos valores devolviendo un valor booleano (*true* o *false*) dependiendo de si la relación es verdadera. Estos operadores son fundamentales para la lógica de control y las decisiones en la programación, permitiendo que el flujo del programa responda a diferentes condiciones. Los operadores relacionales disponibles en Java incluyen:

- **Igual a (`==`)**: verifica si dos valores son iguales.
- **No igual a (`!=`)**: verifica si dos valores no son iguales.
- **Mayor que (`>`)**: verifica si el valor de la izquierda es mayor que el de la derecha.
- **Menor que (`<`)**: verifica si el valor de la izquierda es menor que el de la derecha.
- **Mayor o igual que (`>=`)**: verifica si el valor de la izquierda es mayor o igual que el de la derecha.
- **Menor o igual que (`<=`)**: verifica si el valor de la izquierda es menor o igual que el de la derecha.

A continuación, se presentan ejemplos que demuestran el uso de cada operador relacional:

Igual a (==)

```
int a = 5;
int b = 5;
System.out.println(a == b); // Imprime true
```

No igual a (!=)

```
int a = 5;
int b = 3;
System.out.println(a != b); // Imprime true
```

Mayor que (>)

```
int a = 5;
int b = 3;
System.out.println(a > b); // Imprime true
```

Menor que (<)

```
int a = 2;
int b = 3;
System.out.println(a < b); // Imprime true
```

Mayor o igual que (>=)

```
int a = 5;
int b = 5;
System.out.println(a >= b); // Imprime true
```

Menor o igual que (<=)

```
int a = 2;
int b = 3;
System.out.println(a <= b); // Imprime true
```

Es importante hacer notar que, aunque estos ejemplos usan números enteros (*int*), los operadores relacionales pueden ser aplicados a cualquier tipo de dato numérico (*byte, short, int, long, float, double*). Sin embargo, se debe tener cuidado al usarlos con tipos de punto flotante (*float* y *double*) debido a la precisión de la representación de estos valores en la memoria.

Además, el operador de igualdad (==) y el de no igualdad (!=) también pueden ser usados para comparar referencias de objetos, no su contenido. Para comparar el contenido de objetos como cadenas de texto (*String*), se debería usar el método .equals() en lugar del operador ==.

Los operadores relacionales son una herramienta necesaria para el control de flujo en Java, permitiendo que los programas tomen decisiones y realicen acciones basadas en la comparación de valores.

Operadores lógicos

Los operadores lógicos en Java son utilizados para combinar expresiones booleanas y retornar un valor booleano como resultado (*true* o *false*). Estos operadores son fundamentales para la construcción de condiciones complejas en estructuras de control como *if*, *while*, y *for*, entre otras. Los operadores lógicos principales en Java son:

- ***AND lógico (&&)***: retorna *true* si ambas expresiones booleanas son verdaderas.
- ***OR lógico (||)***: retorna *true* si al menos una de las expresiones booleanas es verdadera.
- ***NOT lógico (!)***: invierte el valor de una expresión booleana; si la expresión es *true*, el resultado será *false*, y viceversa.

A continuación, se presentan ejemplos que ilustran el uso de cada uno de estos operadores:

AND lógico (&&)

```
boolean a = true;
boolean b = false;
System.out.println(a && b); // Imprime false, porque ambas condiciones deben ser verdaderas
```

OR lógico (||)

```
boolean a = true;
boolean b = false;
System.out.println(a || b); // Imprime true, porque al menos una condición es verdadera
```

NOT lógico (!)

```
boolean a = true;
System.out.println(!a); // Imprime false, porque invierte el valor de a
```

Ejemplo integrado

Para entender mejor cómo se pueden combinar estos operadores para formar condiciones más complejas, consideremos el siguiente ejemplo que utiliza operadores lógicos en una estructura de control *if*:

```
int edad = 20;
boolean tienePermiso = true;

// Verificar si la persona es mayor de 18 años y tiene permiso
if (edad > 18 && tienePermiso) {
    System.out.println("Acceso concedido.");
} else {
    System.out.println("Acceso denegado.");
}
```

```
// Verificar si la persona es menor de 18 años o no tiene permiso
if (edad < 18 || !tienePermiso) {
    System.out.println("Acceso denegado.");
} else {
    System.out.println("Acceso concedido.");
}
```

Este ejemplo demuestra cómo los operadores lógicos permiten evaluar condiciones más detalladas y específicas al combinar múltiples expresiones booleanas. En el primer *if*, se verifica si la persona es mayor de 18 años **y** tiene permiso para conceder el acceso. En el segundo *if*, se verifica si la persona es menor de 18 años **o** no tiene permiso para denegar el acceso.

Es importante recordar que Java evalúa las expresiones booleanas de izquierda a derecha y aplicará cortocircuito en operaciones && y ||. Esto significa que, en una operación *AND* (&&), si la primera expresión es *false*, Java no evaluará la segunda expresión porque el resultado final ya no puede ser *true*. De manera similar, en una operación OR (||), si la primera expresión es *true*, la segunda expresión no se evaluará porque el resultado ya es *true*.

Los operadores lógicos también son herramientas para controlar el flujo de programas en Java, permitiendo realizar decisiones complejas basadas en múltiples condiciones.

Operador condicional (ternario)

El operador ternario en Java es una forma abreviada de realizar una operación condicional. Se llama "ternario" porque involucra tres partes: una condición, un resultado si la condición es *true*, y un resultado si la condición es *false*. La sintaxis del operador ternario es la siguiente:

```
condición ? expresión1 : expresión2;
```

Si la condición evaluada es *true*, se ejecuta expresión1 y su valor se convierte en el resultado de la operación; si la condición es *false*, se ejecuta

expresión2 y su valor se convierte en el resultado. Este operador es útil para simplificar el código, especialmente cuando se necesita asignar un valor a una variable dependiendo de una condición.

Ejemplo básico

```
int edad = 20;
String mensaje = edad >= 18 ? "Mayor de edad" : "Menor de edad";
System.out.println(mensaje); // Imprime "Mayor de edad"
```

En este ejemplo, se evalúa si la variable edad es mayor o igual a 18. Si la condición es *true*, se asigna el texto "Mayor de edad" a la variable mensaje. Si es *false*, se asigna "Menor de edad".

Ejemplo con operaciones

```
int a = 10, b = 5;
int max = a > b ? a : b;
System.out.println("El mayor es: " + max); // Imprime "El mayor es: 10"
```

Aquí, se compara a con b para determinar cuál es mayor. El resultado de la comparación determina si el valor de a o b se asigna a la variable max.

Uso en asignaciones condicionales

El operador ternario es particularmente útil para asignaciones condicionales donde una declaración *if-else* podría ser más verbosa:

```
boolean llueve = true;
String actividad = llueve ? "Quedarse en casa" : "Ir al parque";
System.out.println("Actividad: " + actividad); // Imprime "Actividad: Quedarse en casa"
```

En este caso, se decide la actividad a realizar en función de si está lloviendo o no, demostrando cómo el operador ternario puede hacer que el código sea más conciso y legible. Es importante hacer notar que el operador ternario puede hacer que el código sea más compacto, pero

su uso excesivo o en condiciones complejas puede reducir la legibilidad del código.

Debe utilizarse con moderación, especialmente en expresiones que podrían complicarse demasiado al combinar múltiples operadores ternarios. Además, aunque el operador ternario puede simplificar el código al reemplazar estructuras condicionales simples, las estructuras *if-else* pueden ser más adecuadas para condiciones más complejas o cuando se necesitan realizar múltiples operaciones en las ramas condicionales.

Por supuesto que el operador ternario ofrece una sintaxis elegante y concisa para realizar operaciones condicionales, permitiendo tanto la asignación condicional de valores como la toma de decisiones simplificada en una sola línea de código.

Conclusiones

Los operadores en Java son componentes indispensables para la manipulación de datos y el control de la lógica en la programación. Comprender cómo y cuándo utilizar estos operadores permite a los desarrolladores/as escribir código más eficiente y claro, facilitando la creación de programas que puedan realizar cálculos complejos, tomar decisiones basadas en condiciones y manejar múltiples operaciones de manera efectiva. El dominio de estos operadores es vital para cualquier programador/a que aspire a desarrollar aplicaciones Java robustas y funcionales.

Documentos consultados

Afolayan, A., Ezugwu, E. A. y Kwanashie, A. (2011). Arithmetic logic design with color-coded ternary for ternary computing. *International Journal of Computer Applications, 26*(11), 31-37. https://doi.org/10.5120/3162-2929 [Este documento introduce el diseño lógico aritmético utilizando una codificación de color para la lógica ternaria, que es útil para entender la aplicación de operadores en Java].

Barak, A. (1977). Multiplicative algorithms for ternary arithmetic using binary logic. *IEEE Transactions on Computers,* (8), pp. 823-826. https://

doi.org/10.1109/TC.1977.1674922 [Este documento describe algoritmos multiplicativos para la aritmética ternaria, que son útiles para entender los operadores aritméticos en Java].

Daliri, M. S., Mirzaee, R. F., Navi, K. y Bagherzadeh, N. (2017). High-performance ternary operators for scrambling. *Integration*, *59*, 1-9. https://doi.org/10.1016/j.vlsi.2017.03.010 [Este trabajo presenta nuevos operadores ternarios que pueden ser utilizados en algoritmos de criptografía, lo que puede ayudar a entender la eficiencia de los operadores lógicos y aritméticos en Java].

Hanson, W. (1963). Ternary threshold logic. *IEEE Transactions on Electronics and Computers, 12*(3), 191-197. https://doi.org/10.1109/PGEC.1963.263530 [Este documento explora una nueva lógica de umbral ternaria, que es esencial para comprender cómo los operadores lógicos pueden aplicarse en la programación de Java].

Kawashima, I. (2023). Symmetric ternary logic and its systematic logic composition methodology. *ArXiv.* [Este artículo propone una nueva metodología para la composición lógica en sistemas ternarios, relevante para el uso de operadores lógicos en Java]. https://doi.org/10.48550/arXiv.2305.04115

Mohammadi, S. D., Mirzaee, R. F. y Navi, K. (2019). Partial product generation for unbalanced ternary signed multiplication. *International Journal of High Performance Systems Architecture, 8*(4), 238-249. https://dx.doi.org/10.1504/IJHPSA.2019.104952 [Este documento discute la generación de productos parciales en la multiplicación ternaria, proporcionando una visión sobre cómo los operadores aritméticos pueden ser implementados eficientemente].

Pozinkevych, R. (2021). Logical principles in ternary mathematics. *Asian Journal of Research in Computer Science, 7*(3), 49-54. https://doi.org/10.9734/ajrcos/2021/v7i330181 [Este artículo establece la conexión entre los principios lógicos y matemáticos en la aritmética ternaria, proporcionando una base para entender los operadores lógicos en Java].

Sualim, S. A., Mohamad, R. y Saadon, N. (2018). Ontology of mutation testing for Java operators. *International Journal of Innovative Computing, 8*(2). https://doi.org/10.11113/ijic.v8n2.172 [Esta investigación proporciona una ontología para definir la especificación formal de los conceptos y la documentación de los operadores en Java].

Yamamoto, Y. (2004). Extended regular ternary logic functions and majority functions capable of synthesizing any ternary logic function. *Systems and*

Computers in Japan, 35(1), 79-90. https://doi.org/10.1002/scj.10226 [Este artículo define funciones lógicas ternarias extendidas, lo que ayuda a comprender la aplicación de operadores lógicos y aritméticos en Java]

Zaidi, R. (2017). Operators in JavaScript. En *JavaScript Development*. https://consensus.app/papers/operators-javascript-zaidi/76a8ecbcec-8d580eb276bdd83a9d7d7f/ [Este capítulo explica en detalle los operadores en JavaScript, incluyendo operadores aritméticos, de comparación y lógicos, que también son relevantes para Java]

Capítulo 6. Estructuras de control

Introducción

Las estructuras de control en Java se dividen en dos categorías principales: selectivas y repetitivas, cada una con un propósito específico en la programación. Las **estructuras selectivas** (como *if-else* y *switch*) permiten al programa tomar decisiones y ejecutar diferentes bloques de código según condiciones específicas, esenciales para manejar distintos estados o resultados. Por otro lado, las **estructuras repetitivas** (*for*, *while* y *do-while*) permiten ejecutar repetidamente un bloque de código mientras se cumpla una condición, facilitando tareas como la iteración sobre datos. Ambas estructuras son fundamentales para construir aplicaciones dinámicas y adaptables en Java.

Estructuras selectivas

Las estructuras selectivas, como mencionamos previamente, permiten tomar decisiones en el código, ejecutando diferentes bloques de código basados en una o más condiciones.

if-else

El *if-else* es la estructura de control más básica, que ejecuta un bloque de código si una condición es *true* y otro bloque si la condición es *false*.

```
public class Main {
    public static void main(String[] args) {
        int edad = 18;
        if (edad >= 18) {
            System.out.println("Eres mayor de edad");
```

```
        } else {
            System.out.println("No eres mayor de edad");
        }
    }
}
```

En este pequeño programa se verifica si la variable "edad" es mayor o igual a 18. Si esta condición es verdadera, imprime el mensaje "Eres mayor de edad" en la consola. Si la condición no se cumple (lo que significa que "edad" es menor que 18), entonces imprime "No eres mayor de edad." Este proceso de decisión se logra utilizando una estructura de control *if-else*, donde *if* verifica la condición (edad >= 18) y *else* maneja el caso contrario. Cambia los valores de la variable edad y observa las diferentes salidas del programa.

if-else-if

Para múltiples condiciones, se puede usar una serie de *if-else* anidados, la cual permite múltiples verificaciones y acciones. Veamos el siguiente ejemplo:

```
public class Main {
    public static void main(String[] args) {
        int calificacion = 85;
        if (calificacion >= 90) {
            System.out.println("Excelente");
        } else if (calificacion >= 80) {
            System.out.println("Muy bien");
        } else if (calificacion >= 70) {
            System.out.println("Bien");
        } else {
            System.out.println("Necesitas estudiar más");
        }
    }
}
```

Este programa evalúa una calificación numérica y muestra un mensaje basado en el rango en el que esta calificación se encuentra. Si la calificación es 90 o más, imprime "Excelente". Si está entre 80 y 89, imprime "Muy bien". Para una calificación entre 70 y 79, muestra "Bien". Y para calificaciones menores de 70, el mensaje será "Necesitas estudiar más". Utiliza una estructura de control *if-else* encadenada para verificar cada rango de calificaciones y seleccionar el mensaje apropiado para imprimir. De igual forma que con el anterior, prueba distintos valores para la variable "calificación" a fin de que veas los distintos resultados dependiendo de dicho valor.

Switch

El *switch* es útil para casos en los que se necesita comparar una variable contra una serie de valores. Es más limpio y fácil de leer que múltiples bloques *if-else* cuando se trata de muchas condiciones. Veamos ahora el ejemplo con esta estructura condicional:

```
public class Main {
    public static void main(String[] args) {
        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;
            // Puedes añadir más casos aquí
            default:
                System.out.println("Otro día");
        }
    }
}
```

Como podemos ver, este programa utiliza un *switch* para imprimir el nombre del día de la semana basado en el valor numérico de la variable "dia" [sic]. Si dia es 1, imprime "Lunes"; si es 2, "Martes"; y si es 3, "Miércoles". Si dia tiene otro valor que no está definido en los casos (por ejemplo, 4, 5, 6, 7, etc.), entonces imprimirá "Otro día". Para ver los nombres de otros días, puedes modificar el valor de la variable dia a otro número entre 1 y 7 (asumiendo que agregues los casos correspondientes para cada día de la semana) y volver a ejecutar el programa para ver el resultado.

Cuando intentamos adaptar el programa de calificaciones para usar switch en lugar de *if-else*, nos enfrentamos a una limitación clave: *switch* en Java no admite la evaluación de condiciones basadas en rangos de valores. A diferencia de *if-else*, que puede verificar si un valor se encuentra dentro de un intervalo específico (por ejemplo, si una calificación está entre 80 y 89), *switch* requiere valores exactos para cada caso. Para manejar calificaciones en rangos definidos utilizando *switch*, una estrategia es dividir la calificación por 10, reduciendo así la calificación a un dígito representativo que se puede usar en los casos del *switch*. Este enfoque, aunque efectivo, simplifica la evaluación a menos categorías y podría necesitar ajustes para alinearse perfectamente con el sistema de calificaciones original implementado con *if-else*. Este ejemplo lo podemos desarrollar de la siguiente forma:

```
public class Main {
    public static void main(String[] args) {
        int calificacion = 85;
        switch (calificacion / 10) {
            case 10:
            case 9:
                System.out.println("Excelente");
                break;
            case 8:
                System.out.println("Muy bien");
                break;
            case 7:
```

```
                System.out.println("Bien");
                break;
            default:
                System.out.println("Necesitas estudiar más");
                break;
        }
    }
}
```

En esta versión del programa de calificaciones, al dividir calificación por 10, reducimos la calificación a un dígito que refleja su decena (por ejemplo, 85 se convierte en 8). Luego, utilizamos este resultado como la expresión para el *switch*. Los casos 10 y 9 se tratan juntos para cubrir calificaciones de 90 a 100, reflejando un "Excelente". Un caso de 8 refleja calificaciones de "Muy bien", 7 para "Bien", y cualquier cosa por debajo de 7 cae en el *default* "Necesitas estudiar más".

Este enfoque asume que la calificación máxima es 100. Para calificaciones por encima de 100, este código todavía imprimiría "Excelente" y para calificaciones negativas, se mostraría "Necesitas estudiar más", por lo que es necesario realizar más validaciones, pero este es un buen punto de partida para comprender la estructura condicional *switch*.

Estructuras repetitivas

Las estructuras repetitivas, ciclos o bucles, permiten ejecutar un bloque de código varias veces según se cumplan determinadas condiciones. Aunque existen diferentes tipos de estructuras repetitivas en Java (como *for*, *while*, y *do-while*), todas comparten tres elementos esenciales que definen su funcionamiento: el inicio, la condición de término, y el incremento o decremento. Estos elementos aseguran que el bucle pueda ejecutarse de manera controlada.

1. **Inicio**: este es el punto de partida del bucle, donde se inicializan las variables que controlan la ejecución del bucle. En un

bucle *for*, por ejemplo, este sería el lugar donde se establece el valor inicial de la variable de control del bucle.

2. **Condición de término**: es la expresión booleana que se evalúa antes de cada iteración del bucle. Si la condición es *true*, el cuerpo del bucle se ejecuta. Si es *false*, el bucle termina. Esta condición es importante porque previene los bucles infinitos al establecer un criterio de salida claro.
3. **Incremento/Decremento**: este paso ajusta la variable de control del bucle después de cada iteración. Su propósito es acercar el bucle a su condición de término, ya sea incrementando o decrementando la variable. Este paso es importante para asegurar que el bucle eventualmente termine al modificar la variable de control, de manera que la condición de término pueda convertirse en *false*.

Es importante aclarar que, aunque los elementos de inicio, condición de término e incremento/decremento son comunes en las estructuras repetitivas para controlar las iteraciones, en las estructuras *while* y *do-while* no siempre se presentan de manera explícita o estructurada como en un bucle *for*, estos elementos pueden estar dispersos o incluso algunos pueden estar ausentes dependiendo de la lógica específica del programa. Veamos la estructura repetitiva *for*.

For

```
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            System.out.println("Iteración " + i);
        }
    }
}
```

donde: 1. El **inicio** es int i = 0 2. La **condición de término** es i < 5 3. El **incremento** es i++

La salida en la consola es:

Iteración 0
Iteración 1
Iteración 2
Iteración 3
Iteración 4

While

Este bucle es útil cuando quieres repetir un bloque de código mientras se cumpla una condición, pero el número de iteraciones no es conocido de antemano.

```
public class Main {
    public static void main(String[] args) {
        int i = 0; // Inicio
        while (i < 5) { // Condición de término
            System.out.println("Iteración " + i);
            i++; // Incremento
        }
    }
}
```

En este caso:

1. El **inicio** se establece antes del bucle.
2. La **condición de término** controla el acceso al bucle.
3. El **incremento** se realiza dentro del cuerpo del bucle.

La salida es exactamente la misma que con el bucle *for*, es decir:

Iteración 0
Iteración 1
Iteración 2
Iteración 3
Iteración 4

Do-While

Este bucle es parecido al *while*, pero al evaluar al final, garantiza que el bloque de código se ejecute al menos una vez.

```
public class Main {
    public static void main(String[] args) {
        int i = 0; // Inicio
        do {
            System.out.println("Iteración " + i);
            i++; // Incremento
        } while (i < 5); // Condición de término
    }
}
```

Los elementos del bucle son similares al *while*, pero la **condición de término** se evalúa después de ejecutar el cuerpo del bucle, asegurando como se mencionó anteriormente, que el cuerpo se ejecute al menos una vez. La salida es la misma que en los dos bucles anteriores. Independientemente del tipo de estructura repetitiva, el inicio, la condición de término, y el incremento o decremento son importantes para el diseño y funcionamiento eficaz de un programa, permitiendo la ejecución repetida de bloques de código de manera controlada y predecible.

Las estructuras de control son herramientas que te permiten crear programas con lógica compleja de manera eficiente y legible. Ya sea que necesites tomar decisiones simples con *if-else*, evaluar múltiples condiciones con *switch*, o repetir operaciones con bucles *for*, *while* o *do-while*, Java ofrece la flexibilidad para manejar diversas situaciones de control de flujo. La clave es entender cuándo y cómo usar cada una de estas estructuras para desarrollar códigos que no sólo cumplen con los requisitos funcionales, sino que también son fáciles de comprender y mantener.

Conclusiones

Las estructuras de control en Java proporcionan una comprensión detallada de cómo las estructuras selectivas y repetitivas son esenciales para la construcción de programas flexibles y robustos. A través de ejemplos prácticos, se ha ilustrado cómo las instrucciones *if-else, if-else-if* y *switch* permiten a los programadores tomar decisiones lógicas en función de las condiciones evaluadas, mientras que los bucles *for*, *while* y *do-while* facilitan la ejecución repetida de bloques de código. Este capítulo resalta la importancia de elegir la estructura de control adecuada para cada situación, asegurando que el código sea eficiente, legible y capaz de adaptarse a diferentes escenarios de ejecución. Al dominar estas herramientas, las y los desarrolladores pueden crear aplicaciones con una lógica sólida y un flujo de control bien definido, lo que es fundamental para el éxito en la programación en Java y en otros lenguajes de programación.

Documentos consultados

Libros de texto

Eckel, B. (2006). *Thinking in Java* (4th ed.). Prentice Hall.

Horstmann, C. S. y Cornell, G. (2016). *Core Java Volume I--Fundamentals (10th ed.)*. Prentice Hall.

Documentación oficial de Java

Oracle. (n.d.). *The Java™ Tutorials*. https://docs.oracle.com/javase/tutorial/

Sitios web educativos y tutoriales

GeeksforGeeks. (2021). *Control Statements in Java*. https://www.geeksforgeeks.org/control-statements-in-java/

W3Schools. (n.d.). *Java For Loop*. https://www.w3schools.com/java/java_for_loop.asp

Foros y comunidades de programación

Stack Overflow. (n.d.). *Java tag*. https://stackoverflow.com/questions/tagged/java

Cursos en línea

Coursera. (n.d.). *Java Programming and Software Engineering Fundamentals*. https://www.coursera.org/specializations/java-programming

Udemy. (n.d.). *Java Programming Masterclass for Software Developers*. https://www.udemy.com/course/java-the-complete-java-developer-course/

Capítulo 7. Colecciones en Java

Introducción

En el ámbito del desarrollo de *software*, la gestión eficaz de conjuntos de datos, desde su organización hasta la realización de operaciones como búsquedas y ordenamientos, es una tarea recurrente y crítica. Para abordar estas necesidades de manera eficiente y siguiendo buenas prácticas, Java ofrece el *Framework* (marco de trabajo) de colecciones. Este *Framework* se distingue por proporcionar un conjunto estandarizado de clases e interfaces que facilitan el trabajo con grupos de objetos. Al emplear este *Framework* de colecciones, las y los desarrolladores se benefician al utilizar código estandarizado, lo cual promueve la adopción de buenas prácticas al garantizar que las implementaciones de las estructuras de datos y algoritmos sean consistentes y confiables; además, evita la necesidad de desarrollar soluciones propias para problemas comunes, permitiendo a desarrolladores y desarrolladoras concentrarse en las necesidades específicas de sus aplicaciones, al tiempo que asegura una base sólida y bien probada para la manipulación de colecciones de datos.

Framework de colecciones

El *Framework* de colecciones en Java es una arquitectura unificada que permite manipular y acceder a colecciones de objetos de manera coherente. Ofrece varias interfaces, cada una con múltiples implementaciones, para satisfacer diferentes necesidades de almacenamiento y manipulación de datos. Las principales colecciones son *List*, *Set* y *Map*, entre otras, y cada una sirve a un propósito específico:

- ***List***: representa una colección ordenada que puede contener elementos duplicados. Las y los usuarios pueden acceder a los elementos por su posición en la lista (indexados). Ejemplos de implementaciones incluyen ArrayList, LinkedList, entre otros.
- ***Set***: define una colección que no puede contener elementos duplicados. Es útil para crear colecciones de elementos únicos. HashSet y TreeSet son implementaciones comunes de esta interfaz.
- ***Map***: almacena pares clave-valor y no puede contener claves duplicadas. *HashMap* y *TreeMap* son ejemplos de implementaciones de *Map*.

Ventajas de usar el *Framework* de colecciones

- **Eficiencia**: las implementaciones proporcionadas por el *Framework* están optimizadas para diversas operaciones, como la inserción, eliminación y búsqueda de elementos.
- **Flexibilidad**: las interfaces del *framework* permiten que el código sea flexible y reutilizable, ya que puedes cambiar las implementaciones subyacentes sin modificar el código que las utiliza.
- **Consistencia**: al usar estas interfaces y clases estandarizadas, el código se vuelve más coherente y fácil de entender para otros desarrolladores.
- **Potencia**: el *Framework* incluye métodos que realizan operaciones complejas, como ordenamiento y búsqueda, de manera muy sencilla.

Listas (*List*)

La interfaz *List* en Java, como se mencionó anteriormente, representa una colección ordenada que puede contener elementos duplicados. Permite a las personas usuarias acceder a los elementos por su índice, lo que hace que las listas sean muy útiles para escenarios donde el orden de inserción de los elementos debe preservarse o cuando es necesa-

rio acceder a los elementos por su posición. A continuación, se presenta un ejemplo para la creación y agregación de elementos a una lista:

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

public class EjemploListSimple {
    public static void main(String[] args) {
        List<String> nombres = new ArrayList<>();
        nombres.add("Hugo");
        nombres.add("Paco");
        nombres.add("Luis");

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

Este ejemplo se puede observar cómo crear una *List* de *String* usando *ArrayList* como su implementación. Se añaden tres nombres a la lista y luego se imprime la lista completa. Demuestra la operación básica para agregar elementos, así como la capacidad de la lista para mantener un orden. La salida de este programa es:

```
[Hugo, Paco, Luis]
```

Ahora usaremos *LinkedList* para crear una lista de números enteros.

```
import java.util.List;
import java.util.LinkedList;

public class EjemploListAcceso {
    public static void main(String[] args) {
        List<Integer> nums = new LinkedList<>();
        nums.add(10);
        nums.add(20);
        nums.add(30);
```

```
        int primerElemento = nums.get(0); // Acceso
        System.out.println("Primer elemento: " + primerElemento);

        nums.set(1, 25); // Modificación
        System.out.println("Lista modificada: " + nums);
    }
}
```

El ejemplo muestra cómo acceder a un elemento mediante su índice y cómo modificar el valor de un elemento. Seleccionamos el primer elemento usando `get(0)` y modificamos el segundo elemento (índice 1) con un nuevo valor usando `set(1, 25)`, donde `25` es el valor modificado. La salida es:

```
Primer elemento: 10
Lista modificada: [10, 25, 30]
```

Ahora usaremos iteradores para recorrer la lista.

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

public class Main {
    public static void main(String[] args) {
        List<Double> calificaciones = new ArrayList<>();
        calificaciones.add(8.5);
        calificaciones.add(7.3);
        calificaciones.add(9.4);

        System.out.println("Calificaciones antes de eliminar: " +
calificaciones);
        // Iteración usando un iterador
        Iterator<Double> iterator = calificaciones.iterator();
        while (iterator.hasNext()) { //recorre todos los elementos de la lista
```

```
        hasta el último
                double calificacion = iterator.next(); //devuelve el siguiente
        elemento de la colección
                if (calificacion < 8.0) {
                    iterator.remove(); // Eliminación durante la iteración
                }
            }
            System.out.println("Calificaciones después de eliminar: " +
        calificaciones);
        }
    }
```

En este ejemplo ilustramos la iteración sobre una lista de calificaciones usando un Iterator, lo que permite modificar la colección durante la iteración al eliminar elementos que cumplen cierto criterio (en este caso, calificaciones menores a 8.0). Este método es seguro y evita ConcurrentModificationException que podría surgir si se modifica la lista mientras se itera directamente sobre ella. La salida es:

```
Calificaciones antes de eliminar: [8.5, 7.3, 9.4]
Calificaciones después de eliminar: [8.5, 9.4]
```

Para complementar el ejemplo anterior, un Iterator es un objeto que permite recorrer una colección, elemento por elemento, de manera secuencial. Se utiliza especialmente cuando se necesita iterar sobre una colección, como una lista (*List*) o un conjunto (*Set*), y se desea realizar operaciones como obtener o eliminar elementos durante la iteración.

*Métodos importantes de las listas (**List**)*

Como se ha comentado, las listas son una colección ordenada que puede contener elementos duplicados que son ideales para una amplia gama de situaciones donde se requiere mantener un orden específico. A continuación, se presenta una tabla que resume algunos de los métodos más importantes proporcionados por la interfaz *List*, cada uno

con su propósito y uso básico. Se presentan los métodos esenciales que necesitas conocer para aprovechar al máximo las capacidades de las listas en tus proyectos de programación.

Tabla 1. Métodos de la interfaz *List*

Método	Descripción
add(E e)	Añade el elemento especificado al final de la lista.
add(int index, E element)	Inserta el elemento especificado en la posición indicada.
get(int index)	Retorna el elemento en la posición especificada en la lista.
set(int index, E element)	Reemplaza el elemento en la posición especificada con el elemento dado.
remove(int index)	Elimina el elemento en la posición especificada.
remove(Object o)	Elimina la primera ocurrencia del elemento especificado.
size()	Retorna el número de elementos en la lista.
clear()	Elimina todos los elementos de la lista.
indexOf(Object o)	Retorna el índice de la primera ocurrencia del elemento especificado, o -1 si la lista no contiene el elemento.
lastIndexOf(Object o)	Retorna el índice de la última ocurrencia del elemento especificado, o -1 si la lista no contiene el elemento.
contains(Object o)	Verifica si la lista contiene el elemento especificado.
isEmpty()	Verifica si la lista está vacía.
iterator()	Retorna un iterador sobre los elementos en la lista.
listIterator()	Retorna un iterador de lista para la lista.
subList(int fromIndex, int toIndex)	Retorna una vista de la porción de esta lista entre los índices especificados.

Fuente: Elaboración propia.

Conjuntos (*Set*)

La interfaz *Set* en Java representa una colección que no permite elementos duplicados. Esto la hace especialmente útil para almacenar colecciones de elementos únicos donde el orden de los elementos no es importante. Al igual que las listas, los conjuntos ofrecen operaciones para añadir, eliminar y verificar la presencia de elementos, pero cada elemento debe ser único dentro de un *Set*.

Haciendo uso de *Sets*, vamos a proceder a la creación de un *Set* y agregación de elementos:

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

public class Main {
    public static void main(String[] args) {
        Set<String> frutas = new HashSet<>();
        frutas.add("Manzana");
        frutas.add("Banana");
        frutas.add("Cereza");
        frutas.add("Manzana"); // Este elemento no se añadirá porque ya
existe

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

En este ejemplo se muestra cómo crear un *Set* de *String* usando *HashSet* como su implementación. Se intenta añadir "Manzana" dos veces, pero sólo aparecerá una vez en el *Set* debido a la regla de unicidad o que no se pueden repetir los elementos. La salida es:

```
[Cereza, Manzana, Banana]
```

Al igual como hicimos en las listas, sobre los *Sets* también se puede iterar.

```
import java.util.LinkedHashSet;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        Set<Integer> nums = new LinkedHashSet<>();
        nums.add(10);
        nums.add(20);
        nums.add(30);
        nums.add(10); // No se añadirá

        for (Integer num : nums) {
            System.out.println(num);
        }
    }
}
```

Ahora, en este otro ejemplo, se utiliza *LinkedHashSet*, que, a diferencia de *HashSet*, mantiene un orden de inserción. La iteración se realiza con un bucle for-*each*, imprimiendo cada elemento. Al igual que en *HashSet*, los elementos duplicados no se añaden. La salida ahora es:

```
10
20
30
```

Ahora veamos algunas operaciones con *Sets:*

```
import java.util.Set;
import java.util.TreeSet;

public class Main {
    public static void main(String[] args) {
        Set<String> frutas = new TreeSet<>();
        frutas.add("Manzana");
        frutas.add("Banana");
```

```
        frutas.add("Cereza");

        System.out.println("Total de frutas: " + frutas.size());
        System.out.println("Contiene 'Manzana': " + frutas.
    contains("Manzana"));
        frutas.remove("Banana");
        System.out.println("Conjunto después de eliminar 'Banana': "
    + frutas);
    }
}
```

Por último, en este ejemplo se utiliza *TreeSet*, que automáticamente ordena los elementos en orden ascendente. Se demuestran operaciones básicas como obtener el tamaño del conjunto con size(), verificar la presencia de un elemento con contains() y eliminar un elemento con remove(). La salida de este programa es:

```
Total de frutas: 3
Contiene 'Manzana': true
Conjunto después de eliminar 'Banana': [Cereza, Manzana]
```

*Métodos importantes de los conjuntos (**Set**)*

Al contrario de *List*, *Set* no mantiene un orden específico de sus elementos. A continuación, se destacan algunos de los métodos más importantes proporcionados por la interfaz *Set*, facilitando operaciones como la adición, eliminación y consulta de elementos.

Tabla 2. Métodos de la interfaz *Set*

Método	Descripción
add(E e)	Añade el elemento especificado al conjunto si aún no está presente.
remove(Object o)	Elimina la especificada instancia del conjunto si está presente.

Continúa en la página 75...

Método	Descripción
contains(Object o)	Verifica si el conjunto contiene la instancia del elemento especificado.
size()	Retorna el número de elementos en el conjunto.
isEmpty()	Verifica si el conjunto está vacío.
clear()	Elimina todos los elementos del conjunto.
iterator()	Retorna un iterador sobre los elementos en el conjunto.
addAll(Collection<? extends E> c)	Añade todos los elementos en la colección especificada al conjunto si no están ya presentes.
removeAll(Collection<?> c)	Elimina del conjunto todos sus elementos que están contenidos en la colección especificada.
retainAll(Collection<?> c)	Retiene sólo los elementos en el conjunto que están contenidos en la colección especificada.
containsAll(Collection<?> c)	Verifica si el conjunto contiene todos los elementos de la colección especificada.

Fuente: Elaboración propia.

Mapas (*Map*) en Java

La interfaz *Map* es una estructura de datos que almacena pares tipo clave-valor, donde cada clave única se asocia con un valor específico. Esta capacidad hace de los *Map* una herramienta indispensable para operaciones que requieren buscar, actualizar o eliminar elementos mediante una clave. Además, es importante destacar que los valores dentro de un Map pueden ser extremadamente versátiles, abarcando desde tipos simples hasta colecciones complejas como listas o conjuntos.

Lo anterior permite que los *Map* sean utilizados en una amplia gama de aplicaciones, desde el mapeo simple de objetos hasta estructuras de datos más complejas que requieren agrupar múltiples elementos bajo una clave común, facilitando así el manejo avanzado de datos

y relaciones dentro de las aplicaciones. Veamos algunos ejemplos de su uso. Se observan en el siguiente ejemplo la creación y agregación de elementos a un *Map*:

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

public class Main {
    public static void main(String[] args) {
        Map<String, Integer> edades = new HashMap<>();
        edades.put("Ana", 28);
        edades.put("Luis", 34);
        edades.put("Carlos", 23);

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

El programa comienza importando las clases necesarias del paquete java.util: *Map* y *HashMap*. *Map* es una interfaz que define un objeto que mapea claves a valores, y *HashMap* es una implementación concreta de esta interfaz que utiliza una tabla hash para almacenar los pares clave-valor.

En este ejemplo se muestra cómo crear un *Map* usando *HashMap*. Este almacena las edades asociadas a nombres de personas. Se utiliza el método *put* para añadir pares clave-valor al mapa. Si se intenta añadir otra entrada con una clave que ya existe en el mapa, el valor anterior asociado a esa clave se sobrescribe. La salida de este programa es:

```
{Ana=28, Luis=34, Carlos=23}
```

Ahora veremos cómo se accede a los elementos del *Map*:

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

public class Main {
```

```
    public static void main(String[] args) {
        Map<String, String> capitales = new HashMap<>();
        capitales.put("España", "Madrid");
        capitales.put("Francia", "París");
        capitales.put("Italia", "Roma");

        String capitalEspaña = capitales.get("España");
        System.out.println("La capital de España es: " +
capitalEspaña);

        System.out.println("La capital de Alemania es: " + capitales.
getOrDefault("Alemania", "No encontrada"));
    }
}
```

En el ejemplo, se muestra cómo acceder a los valores almacenados en un *Map* utilizando el método *get* y proporcionando la clave correspondiente. También se utiliza *getOrDefault* para obtener un valor asociado a una clave, especificando un valor por defecto en caso de que la clave no esté presente en el mapa. La salida es:

```
La capital de España es: Madrid
```

```
La capital de Alemania es: No encontrada
```

Por último, veamos también cómo iterar sobre los elementos de un *Map*.

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

public class Main {
    public static void main(String[] args) {
        // Crear un HashMap para almacenar los nombres como claves y las
edades como valores.
        Map<String, Integer> edades = new HashMap<>();
```

```
        // Agregar pares clave-valor al mapa.
        edades.put("Hugo", 8);
        edades.put("Paco", 9);
        edades.put("Luis", 10);

        // Iterar sobre el conjunto de entradas del mapa usando un bucle
    for-each.
        // Cada elemento del conjunto es una entrada del mapa (Map.
    Entry), que contiene una clave y un valor.
        for (Map.Entry<String, Integer> entrada : edades.entrySet()) {
            // Obtener la clave de la entrada, que en este caso es el nombre de
    la persona.
            String nombre = entrada.getKey();
            // Obtener el valor asociado a la clave, es decir, la edad de la
    persona.
            Integer edad = entrada.getValue();
            // Imprimir en consola el nombre de la persona y su edad.
            System.out.println(nombre + " tiene " + edad + " años");
        }
    }
}
```

El programa presenta un ejemplo de cómo utilizar la interfaz *Map* en Java, específicamente a través de su implementación *HashMap*, para almacenar y manipular un conjunto de pares clave-valor. En este caso, el programa utiliza un mapa para almacenar las edades de diferentes personas, identificadas por su nombre. A continuación, se detalla el funcionamiento del programa con comentarios en el código y una explicación general.

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

public class Main {
```

```
    public static void main(String[] args) {
        // Crear un HashMap para almacenar los nombres como claves y las
edades como valores
        Map<String, Integer> edades = new HashMap<>();

        // Agregar pares clave-valor al mapa, donde la clave es un String
con el nombre de la persona,
        // y el valor es un Integer que representa su edad.
        edades.put("Hugo", 8);
        edades.put("Paco", 9);
        edades.put("Luis", 10);

        // Iterar sobre el conjunto de entradas del mapa usando un bucle
for-each.
        // Cada elemento del conjunto es una entrada del mapa (Map.
Entry), que contiene una clave y un valor.
        for (Map.Entry<String, Integer> entrada : edades.entrySet()) {
            // Obtener la clave de la entrada, que en este caso es el nombre de
la persona.
            String nombre = entrada.getKey();
            // Obtener el valor asociado a la clave, es decir, la edad de la
persona.
            Integer edad = entrada.getValue();
            // Imprimir en consola el nombre de la persona y su edad.
            System.out.println(nombre + " tiene " + edad + " años");
        }
    }
}
```

En el método *main*, se crea una instancia de *HashMap* llamada "edades", donde: a) las claves son de tipo *String* (nombres de personas) y b) los valores son de tipo *Integer* (las edades correspondientes). Luego, el programa utiliza el método *put* para añadir tres pares clave-valor al mapa, representando los nombres de tres personas y sus edades.

Posteriormente, el programa itera sobre el conjunto de entradas (*entrySet*) del mapa edades. Para cada entrada, se obtiene la clave (*getKey*) y el valor (*getValue*), que representan, respectivamente, el nombre de la persona y su edad. Finalmente, el programa imprime un mensaje en la consola para cada persona, mostrando su nombre y edad.

```
Hugo tiene 8 años

Luis tiene 10 años

Paco tiene 9 años
```

Métodos importantes de la interfaz Map en Java

A diferencia de *Set* y *List*, *Map* maneja dos conjuntos: uno de claves y otro de valores asociados a esas claves. A continuación, se presentan algunos de los métodos más importantes proporcionados para trabajar con mapas, mostrando la amplia gama de operaciones disponibles.

Tabla 3. Métodos para trabajar con mapas

Método	Descripción
put(K key, V value)	Asocia el valor especificado con la clave especificada en el mapa.
get(Object key)	Retorna el valor al que se asocia la clave especificada, o *null* si el mapa no contiene la clave.
remove(Object key)	Elimina la asociación de la clave especificada si está presente.
containsKey(Object key)	Verifica si el mapa contiene una asociación para la clave especificada.
containsValue(Object value)	Verifica si el mapa mapea una o más claves a este valor.
keySet()	Retorna un *Set* de las claves contenidas en este mapa.
values()	Retorna una colección de los valores contenidos en este mapa.

Continúa en página 81...

Método	Descripción
entrySet()	Retorna un conjunto de las asociaciones de clave-valor contenidas en este mapa.
size()	Retorna el número de asociaciones de clave-valor en el mapa.
isEmpty()	Verifica si el mapa está vacío.
clear()	Elimina todas las asociaciones de clave-valor del mapa.
putAll(Map<? extends K, ? extends V> m)	Copia todas las asociaciones del mapa especificado al mapa actual.

Fuente: Elaboración propia.

Conclusiones

El *Framework* de colecciones de Java representa una herramienta básica para el manejo eficiente y estructurado de datos en el desarrollo de *software*. Ofrece una gama versátil de interfaces y clases que permiten a las y los desarrolladores abordar diversas necesidades de almacenamiento y manipulación de datos, desde listas ordenadas y conjuntos de elementos únicos hasta mapas de pares clave-valor. La estandarización que proporciona este *Framework* no sólo promueve buenas prácticas de programación, sino que también mejora la legibilidad, mantenibilidad y eficiencia del código.

Al proporcionar implementaciones optimizadas y bien probadas de estructuras de datos comunes, el *Framework* de Colecciones libera a las personas que desarrollan de la necesidad de reinventar soluciones para problemas recurrentes, permitiéndoles concentrarse en los aspectos específicos de sus aplicaciones. En última instancia, el dominio de estas colecciones es esencial para cualquier programador Java, ya que su uso adecuado puede mejorar significativamente la calidad y el rendimiento del *software* generado.

Documentos consultados

Libros

Bloch, J. (2017). *Effective Java* (3rd Edition). Addison-Wesley Professional.

Deitel, P. J., Deitel, H. M., Deitel, A. y Deitel, P. T. (2021). *Java: How to program* (11th Edition). Pearson.

Artículos de revista

Ciucciola, B., Donato, F. y Di Modica, G. (2016). Exploring the Performance of Java Collections. *ACM SIGPLAN Notices, 51*(6), 77-88. https://dl.acm.org/doi/10.1145/3030207.3030221

Popescu, A. y Vasilescu, V. (2014). Java Collections Framework: A Tutorial. *International Journal of Software and Systems Engineering (IJSSE), 8*(2), 15-28. https://docs.oracle.com/javase/tutorial/collections/

Sitios web

GeeksforGeeks (n.d.). *Java Collections Framework*. https://www.geeksforgeeks.org/tag/java-collections/ (Consultado el: 20 de junio de 2024).

Oracle (n.d.). *Java Documentation: Collections Framework*. https://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html (Consultado el: 20 de junio de 2024).

Capítulo 8. Paquetes

Introducción

A lo largo de este capítulo, examinaremos la definición y el propósito de los paquetes, cómo se estructuran y nombran adecuadamente para maximizar su utilidad y cómo importar correctamente paquetes y clases para aprovechar la funcionalidad que ofrecen. Este conocimiento no solo es fundamental para organizar adecuadamente tus propios proyectos, sino también para entender y trabajar eficientemente con las numerosas bibliotecas y *frameworks* disponibles en el ecosistema de Java.

La organización y modularidad del código son importantes para que las aplicaciones se puedan mantener, escalar y comprender de una mejor manera. Los paquetes son fundamentales pues contribuyen significativamente a estos objetivos, permitiendo a los desarrolladores: a) agrupar clases relacionadas y, b) reutilizar código de manera eficiente en diferentes partes de una aplicación o entre diferentes proyectos.

Los paquetes en Java no son más que contenedores que agrupan clases, interfaces y otros elementos relacionados, ofreciendo una estructura organizada para el código fuente y facilitando su gestión. Esta organización no sólo ayuda a evitar conflictos de nombres al separarlos en sus propios espacios, sino que también mejora la legibilidad y el mantenimiento del código. Por otro lado, el mecanismo de importación permite acceder a clases y paquetes externos desde el código actual, habilitando la reutilización de componentes y bibliotecas de manera simple y eficaz.

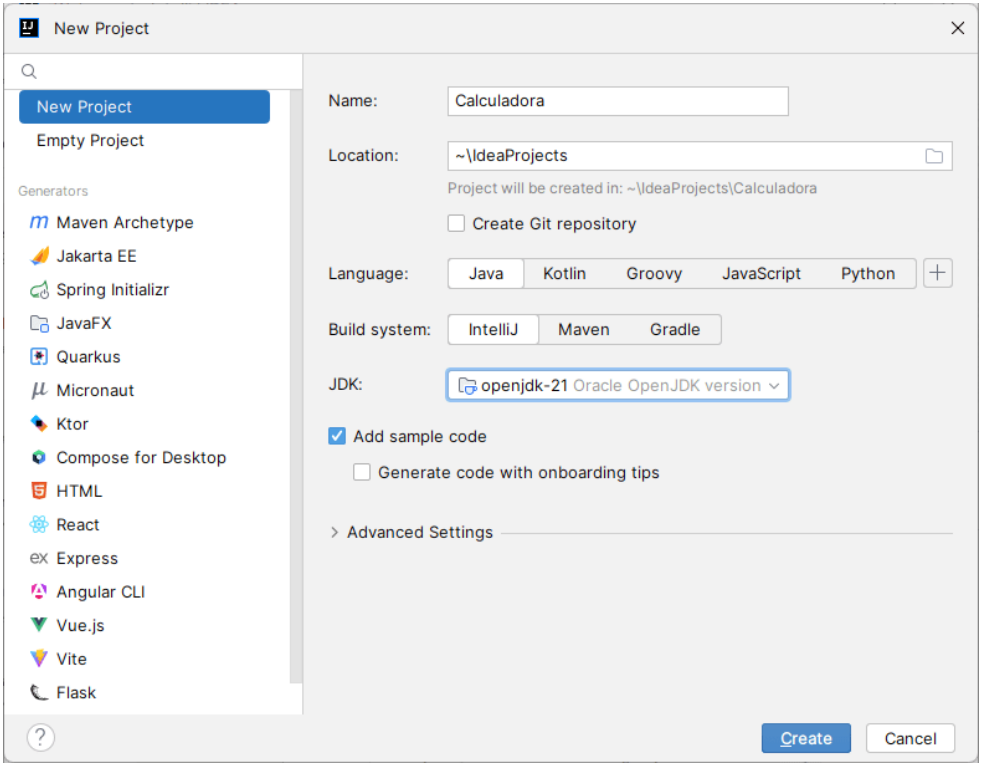
Con estos conceptos bien establecidos vas a saber cómo estructurar tus aplicaciones de manera que promuevan la reutilización, faciliten la navegación por el código y minimicen los conflictos entre nombres

de clases, contribuyendo a la creación de sistemas de *software* robustos, mantenibles y escalables.

Definición y propósito de los paquetes

Como se menciona de manera general en esta guía, un paquete es una forma de agrupar clases, interfaces y subpaquetes relacionados. Los paquetes actúan como contenedores de nombres y ofrecen un espacio de nombres único para las clases que contienen, ayudando a organizar el código de manera lógica y modular. En términos de estructura de archivos, un paquete en Java se representa como directorios en el sistema de archivos, donde cada paquete corresponde a un directorio y cada clase contenida en un paquete es un archivo dentro de este directorio. Para comprender mejor qué es un paquete, vamos a crear un proyecto nuevo, al que le llamaremos Calculadora como se puede ver en la siguiente figura.

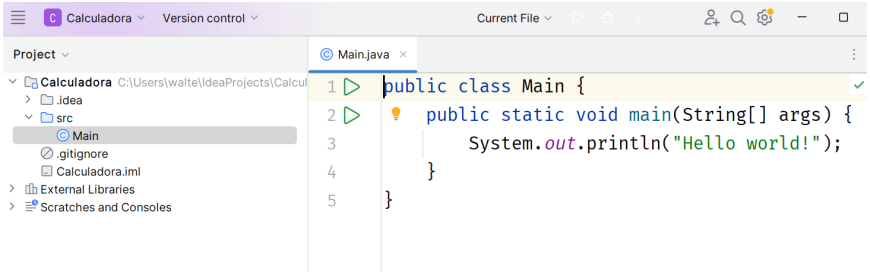
Figura 1. Nuevo proyecto



Fuente: Captura de pantalla.

A continuación, se puede ver cómo aparece en el explorador del IDE nuestro proyecto con un directorio src y el programa de base.

Figura 2. Vista del directorio



Fuente: Captura de pantalla.

Para comprender mejor esta estructura empecemos con una clase *Main* y dentro de ella la función *main* como punto de entrada a nuestro programa. El siguiente código corresponde a una calculadora simple con cuatro operaciones aritméticas básicas:

```
public class Main {
   public static void main(String[] args) {
      System.out.println("Mi Calculadora");
      System.out.println("Suma: " + (5 + 3));
      System.out.println("Resta: " + (5 - 3));
      System.out.println("Multiplicación: " + (5 * 3));
      System.out.println("División: " + ((double) 5 / 3));
   }
}
```

La salida del programa es:

```
Mi Calculadora
Suma: 8
Resta: 2
Multiplicación: 15
División: 1.6666666666666667
```

Como se puede observar, este programa imprime el resultado de las cuatro operaciones aritméticas básicas: suma, resta, multiplicación y división, utilizando los números 5 y 3. Al ejecutarlo, primero muestra un mensaje de introducción ("Mi Calculadora") y luego procede a calcular y mostrar los resultados de sumar (5 + 3), restar (5 - 3), multiplicar (5 * 3) y dividir (5 / 3) estos dos números. Cada operación se realiza dentro de los paréntesis para asegurar que se calcula antes de concatenar el resultado con el texto que se va a imprimir, garantizando así que la salida muestre correctamente los resultados de las operaciones matemáticas.

La implementación previa de la calculadora, aunque funcional, se ve limitada por el uso fijo de los números 5 y 3 en las operaciones. Para crear una versión más versátil y adaptativa, podemos almacenar los operandos en variables. Este enfoque permite modificar fácilmente los valores sin necesidad de ajustar el código de las operaciones. Aunque una versión aún más avanzada podría incluir la lectura de estos números directamente desde la entrada del usuario/a, en el contexto de este ejemplo específico, nos centraremos únicamente en la utilización de variables para los operandos. El código correspondiente es el siguiente:

```
public class Main {
    public static void main(String[] args) {
        // Definición de variables para almacenar los números
        int num1 = 5;
        int num2 = 3;

        // Imprime el título de la calculadora
        System.out.println("Mi Calculadora");

        // Realiza y muestra el resultado de las operaciones básicas
utilizando las variables
        System.out.println("Suma: " + (num1 + num2));
        System.out.println("Resta: " + (num1 - num2));
        System.out.println("Multiplicación: " + (num1 * num2));
```

```
        System.out.println("División: " + ((double) num1 / num2));
    }
}
```

La salida del programa es:

```
Mi Calculadora
  Suma: 8
  Resta: 2
  Multiplicación: 15
  División: 1.6666666666666667
```

Este código mejora la flexibilidad de la calculadora al definir num1 y num2 como variables, lo que facilita la modificación de los valores con los que se realizarán las operaciones. Se mantiene la simplicidad del ejemplo, omitiendo la lectura de entradas del usuario, pero se abre la puerta a futuras expansiones, como solicitar estos números de manera interactiva.

Para mejorar la modularidad de nuestra calculadora, procederemos a encapsular la lógica de cada operación aritmética en métodos de clase distintos dentro de la misma clase *Main*. Esto significa crear un método dedicado para la suma, resta, multiplicación y división, respectivamente. Esta estructuración no sólo hace que el código sea más organizado y legible, sino que también facilita su mantenimiento y la expansión futura de la calculadora. A continuación, veamos cómo implementar esta versión mejorada:

```
public class Main {
    public static void main(String[] args) {
        int num1 = 5;
        int num2 = 3;

        System.out.println("Mi Calculadora Modular");
        System.out.println("Suma: " + sumar(num1, num2));
        System.out.println("Resta: " + restar(num1, num2));
        System.out.println("Multiplicación: " + multiplicar(num1,
```

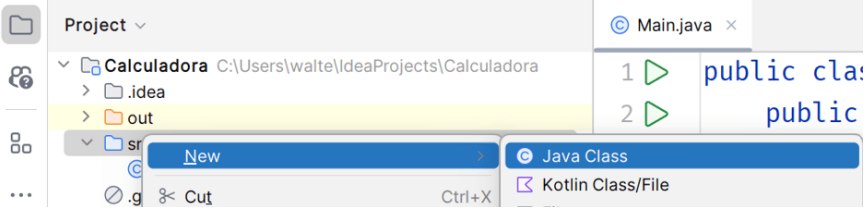
```
        num2));
            System.out.println("División: " + dividir(num1, num2));
        }

        public static int sumar(int a, int b) {
            return a + b;
        }

        public static int restar(int a, int b) {
            return a - b;
        }

        public static int multiplicar(int a, int b) {
            return a * b;
        }

        public static double dividir(int a, int b) {
            if (b == 0) {
                System.out.println("Error: División por cero.");
                return 0;
            }
            return (double) a / b;
        }
    }
```

En este código, cada operación matemática se realiza en su propio método: sumar, restar, multiplicar, y dividir, lo que, como mencionamos previamente, mejora la claridad y la organización del programa. La llamada a cada uno de estos métodos se hace desde el método *main*, pasando num1 y num2 como argumentos. Este enfoque modular facilita la comprensión del flujo del programa y simplifica la adición de nuevas funcionalidades o la modificación de las existentes.

Para continuar avanzando en la organización y modularidad de nuestra calculadora, la siguiente mejora implica crear una clase externa que contenga las operaciones matemáticas. Para esto vamos a crear un

nuevo archivo (nueva clase) que se llame Operaciones dentro de nuestra carpeta src, para esto presionamos clic derecho encima de la carpeta y seleccionamos la clase java como en la siguiente figura:

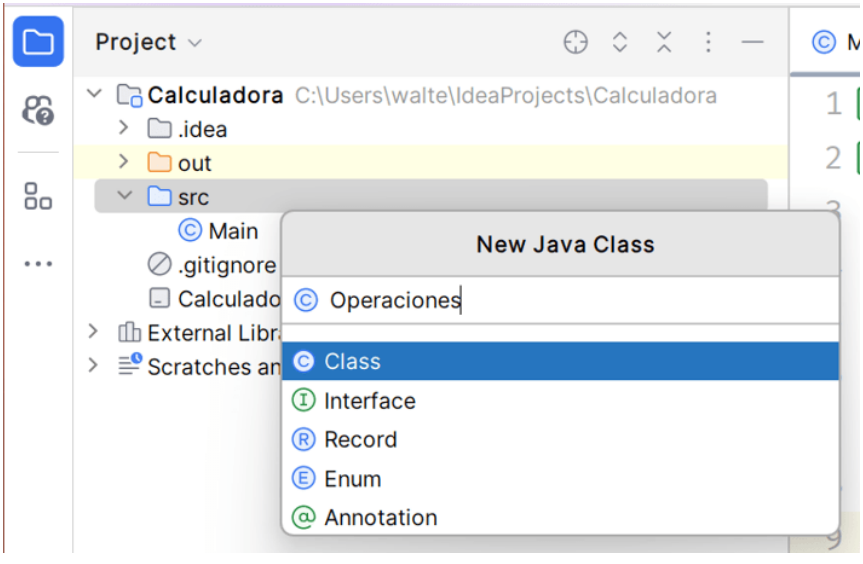
Figura 3. Selección de clase



Fuente: Captura de pantalla.

Ahora establecemos el nombre de nuestra clase (Operaciones), como se aprecia a continuación:

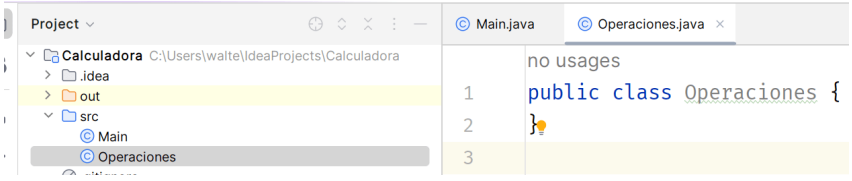
Figura 4. Nombre de clase



Fuente: Captura de pantalla.

Esto nos generará el código base para nuestra nueva clase:

Figura 5. Generación de código



Fuente: Captura de pantalla, elaboración propia.

Al ubicar esta nueva clase dentro del mismo proyecto, se facilita su invocación desde la clase principal. Para utilizar las operaciones definidas en la clase externa, primero se crea una instancia de esta clase en la clase principal. Luego, se accede a cada uno de sus métodos mediante el operador "." (punto) siguiendo las buenas prácticas de programación orientada a objetos. A continuación, se muestra la implementación de esta versión mejorada:

Primero, definimos la clase que contendrá las operaciones matemáticas:

```
//Archivo: Operaciones.java
public class Operaciones {
    public int sumar(int a, int b) {
        return a + b;
    }

    public int restar(int a, int b) {
        return a - b;
    }

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

```
    public double dividir(int a, int b) {
        if (b == 0) {
            System.out.println("Error: División por cero.");
            return 0;
        }
        return (double) a / b;
    }
}
```

La clase Operaciones es una colección de métodos diseñados para realizar operaciones matemáticas básicas: suma, resta, multiplicación y división. Cada método acepta dos parámetros de entrada y retorna el resultado de la operación correspondiente. A continuación, se detalla la funcionalidad de cada método dentro de la clase:

- **sumar(int a, int b)**: Acepta dos enteros como parámetros y retorna su suma. Es una operación directa que no requiere validación adicional.
- **restar(int a, int b)**: Toma dos enteros, `a` y `b`, y retorna la diferencia entre `a` y `b`. Al igual que el método `sumar`, esta operación se realiza sin necesidad de validaciones adicionales.
- **multiplicar(int a, int b)**: Recibe dos enteros y retorna su producto. Este método calcula la multiplicación de los parámetros proporcionados.
- **dividir(int a, int b)**: Este método es ligeramente diferente de los anteriores debido a que implica una validación específica para evitar la división por cero, una operación indefinida que resultaría en un error en tiempo de ejecución. Acepta dos parámetros, `a` (numerador) y `b` (denominador), y retorna un *`double`* que es el resultado de dividir `a` por `b`. Si `b` es 0, imprime un mensaje de error y retorna 0 para evitar la excepción de división por cero.

La clase Operaciones encapsula la lógica de las operaciones matemáticas básicas de manera que pueda ser reutilizada fácilmente en diferentes partes de una aplicación. Al separar estas operaciones en métodos individuales dentro de una clase dedicada, se promueve la claridad del código y se facilita su mantenimiento y expansión en el futuro. Ahora hacemos uso de esta clase dentro de la clase principal (Main) creando una instancia de Operaciones y utilizando cada uno de sus métodos:

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

        int num1 = 5;
        int num2 = 3;

        System.out.println("Mi Calculadora con Clase Externa");
        System.out.println("Suma: " + operaciones.sumar(num1,
num2));
        System.out.println("Resta: " + operaciones.restar(num1,
num2));
        System.out.println("Multiplicación: " + operaciones.
multiplicar(num1, num2));
        System.out.println("División: " + operaciones.dividir(num1,
num2));
    }
}
```

Ahora la salida del programa es la siguiente:

```
Mi Calculadora con Clase Externa
  Suma: 8
  Resta: 2
  Multiplicación: 15
  División: 1.6666666666666667
```

Esta estructura no sólo separa claramente la lógica de las operaciones matemáticas de la lógica de la interfaz de usuario/a, sino que también promueve la reutilización del código y facilita las pruebas y el mantenimiento de la aplicación. Al encapsular las operaciones en su propia clase, se adopta un enfoque modular que es fundamental para la construcción de aplicaciones escalables y mantenibles.

El uso de paquetes en Java sirve a varios propósitos importantes que son vitales para el desarrollo de *software*:

- **Evitar conflictos de nombres**: al agrupar clases en paquetes con espacios de nombres únicos, se minimiza el riesgo de conflictos de nombres. Esto es especialmente útil en proyectos grandes o cuando se utilizan bibliotecas de terceros, donde la posibilidad de que dos clases tengan el mismo nombre es significativa.
- **Mejorar la mantenibilidad**: los paquetes permiten organizar el código de manera lógica, agrupando clases que comparten una funcionalidad común o pertenecen a una misma categoría de tareas. Esto facilita la navegación por el código y mejora su comprensión y mantenimiento.
- **Control de acceso**: los paquetes en Java también definen un nivel de acceso. Las clases pueden limitar el acceso a sus miembros (como métodos y variables) a otras clases dentro del mismo paquete usando modificadores de acceso, como `protected` y el acceso de paquete predeterminado. Esto ayuda a encapsular y proteger los datos, exponiendo sólo lo que es necesario al exterior.
- **Reutilización de c**ódigo: la organización en paquetes fomenta la reutilización de clases y componentes en diferentes partes de una aplicación o incluso entre diferentes proyectos. Al tener clases bien organizadas en paquetes, las y los desarrolladores pueden reutilizar fácilmente el código mediante la importación de estos.

Ejemplo básico

La declaración de un paquete en Java se realiza en la primera línea del archivo de código fuente, utilizando la palabra clave *package*, seguida del nombre del paquete, es decir:

```
package com.miempresa.miaplicacion;

public class MiClase {
    // Detalles de implementación
}
```

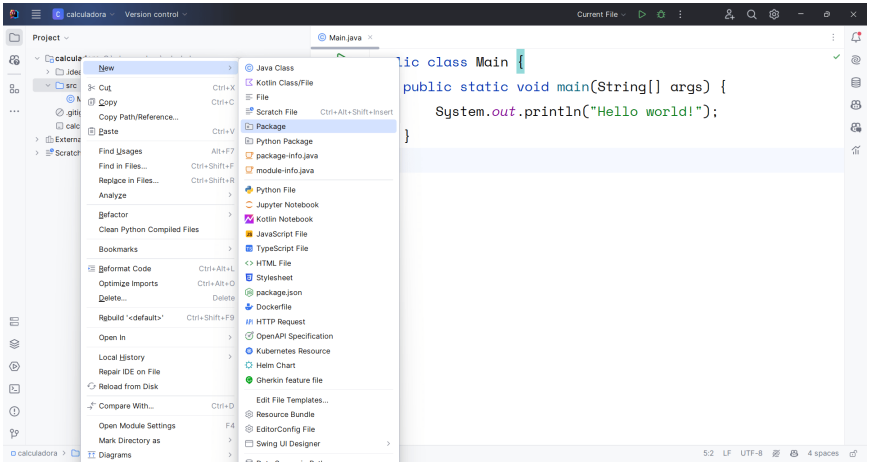
En este ejemplo, MiClase se coloca dentro del paquete com.miempresa.miaplicacion. La estructura de directorios reflejaría esta organización, creando una carpeta com con una subcarpeta miempresa, que a su vez contiene una subcarpeta miaplicacion, donde se encontraría el archivo MiClase.java.

Creación de un paquete para la calculadora

Paso 1: Estructura de paquetes

Para este ejemplo, crearemos paquete de operaciones que va a contener las clases para las operaciones matemáticas de nuestra calculadora:

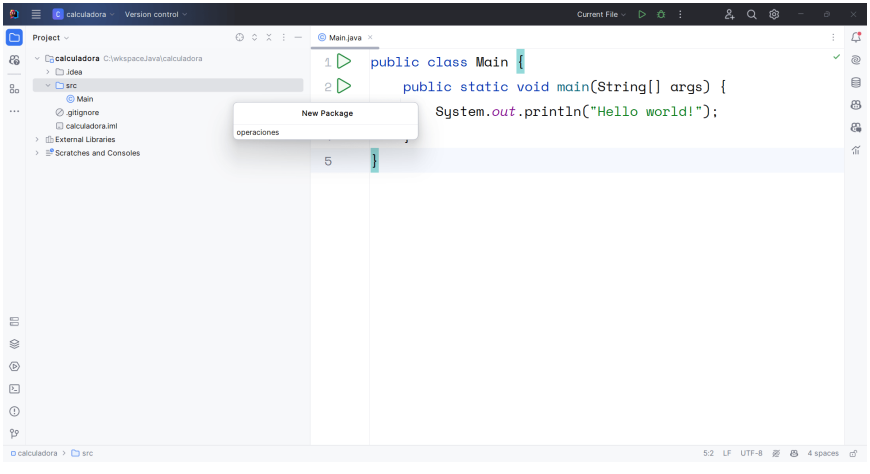
Figura 6. Creación de paquete



Fuente: Captura de pantalla, elaboración propia.

Y se le asigna el nombre del paquete, en este caso *operaciones*:

Figura 7. Asignación de nombre

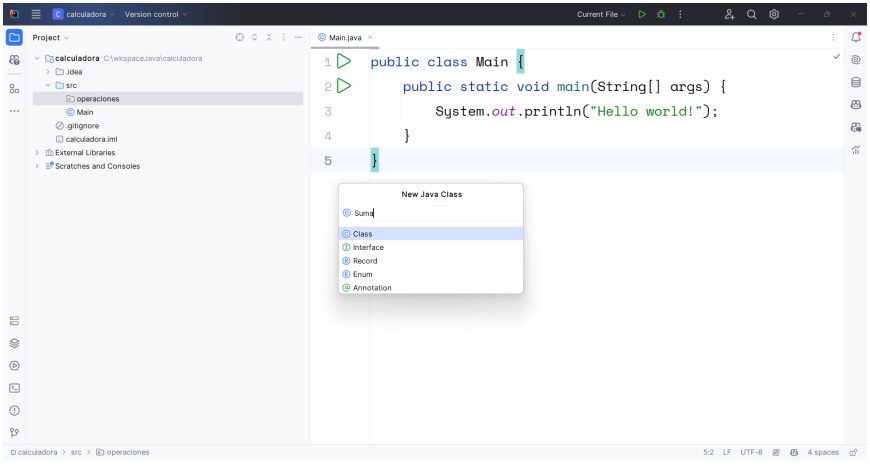


Fuente: Captura de pantalla, elaboración propia.

Paso 2: Creación de clases de operaciones

Dentro del paquete operaciones, definiremos cuatro clases: Suma, Resta, Multiplicación, y División.

Figura 8. Definición de clases



Fuente: Captura de pantalla, elaboración propia.

Archivo de la clase Suma (Suma.java)

```
package operaciones;

public class Suma {
}
```

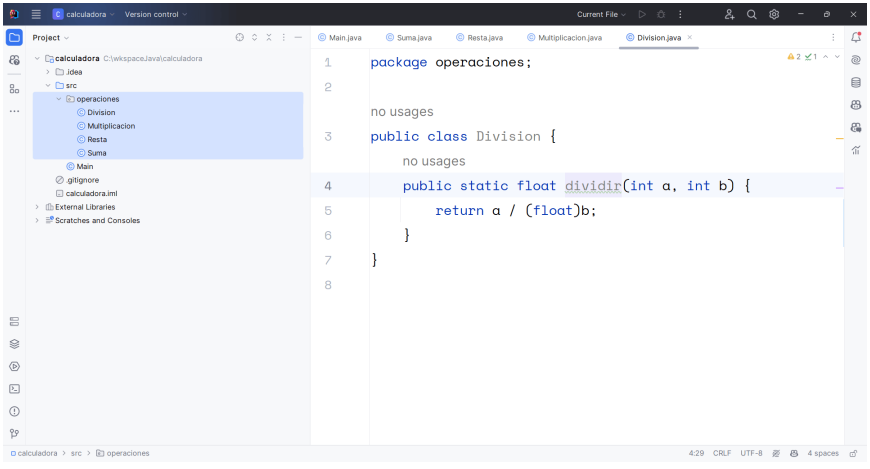
Ahora crea el método correspondiente a la operación aritmética de suma, el código quedaría de la siguiente forma:

```
package operaciones;

public class Suma {
    public static int sumar(int a, int b) {
        return a + b;
    }
}
```

Crea archivos similares para Resta, Multiplicación, y División, ajustando el método correspondiente a realizar la operación matemática apropiada.

El árbol de archivos del proyecto debería verse parecido a la siguiente pantalla:

Figura 9. Árbol de archivos del proyecto



Fuente: Captura de pantalla, elaboración propia.

El código final de las tres clases faltantes podría ser:

Archivo de la clase División (Division.java)

```
package operaciones;

public class Division {
    public static float dividir(int a, int b) {
        return a / (float)b;
    }
}
```

Archivo de la clase Multiplicación (Multiplicacion.java)

```
package operaciones;

public class Multiplicacion {
    public static int multiplicar(int a, int b) {
        return a * b;
    }
}
```

Archivo de la clase Resta (Resta.java)

```
package operaciones;

public class Resta {
    public static int restar(int a, int b) {
        return a - b;
    }
}
```

Paso 3: Clase principal de la aplicación

En nuestro directorio src ya tenemos un archivo Main.java, podemos usar este mismo o crear un nuevo paquete donde encapsulemos el programa principal. En el que ya está creado usemos nuestro nuevo paquete operaciones.

Archivo: CalculadoraApp.java

```
import operaciones.*;

public class Main {
    public static void main(String[] args) {
        System.out.println("Suma: " + Suma.sumar(5, 3));
        System.out.println("Resta: " + Resta.restar(5, 3));
        System.out.println("Multiplicación: " + Multiplicacion.
multiplicar(5, 3));
```

```
        System.out.println("División: " + Division.dividir(5, 3));
        }
}
```

Al ejecutar nuestro programa la salida es:

```
Suma: 8
Resta: 2
Multiplicación: 15
División: 1.6666666
```

Este ejemplo ilustra cómo los paquetes en Java pueden ayudar a organizar una aplicación simple en componentes lógicos y reutilizables. Separar las operaciones matemáticas en su propio paquete no sólo hace que el código sea más claro y mantenible, sino que también facilita la expansión o modificación futura de la calculadora, demostrando la potencia y flexibilidad que ofrecen los paquetes en Java.

Calculadora con métodos y propiedades

Vamos a desarrollar una calculadora que opere utilizando dos propiedades internas, para esto se puede diseñar una clase que contenga estas propiedades como atributos y métodos para realizar operaciones aritméticas básicas con ellas. Esta estructura no sólo organiza el código de manera lógica y modular, sino que también encapsula la funcionalidad de la calculadora, permitiendo fácilmente la expansión o modificación en el futuro.

Diseño de la clase calculadora

Pongamos nuestra clase en el paquete operaciones.

1) **Definición de propiedades**: la clase Calculadora tendrá dos propiedades numéricas que almacenarán los valores sobre los cuales se realizarán las operaciones. Estas propiedades serán de tipo *double* para permitir operaciones con decimales.
2) **Métodos para operaciones**: la clase incluirá métodos para sumar, restar, multiplicar y dividir estos valores. Cada método accederá a las propiedades de la instancia para realizar la operación correspondiente.

3) **Constructor**: la clase proporcionará un constructor para inicializar las dos propiedades con valores específicos al crear un objeto de Calculadora.

Implementación en Java

Veamos el código de la clase Calculadora:

```
public class Calculadora {
    // Propiedades de la clase para almacenar los dos números.
    private double num1;
    private double num2;

    // Constructor que inicializa los números con los valores
proporcionados.
    public Calculadora(double num1, double num2) {
        this.num1 = num1;
        this.num2 = num2;
    }

    // Método para sumar los dos números.
    public double sumar() {
        return num1 + num2;
    }

    // Método para restar el segundo número del primero.
    public double restar() {
        return num1 - num2;
    }

    // Método para multiplicar los dos números.
    public double multiplicar() {
        return num1 * num2;
     }

    // Método para dividir el primer número por el segundo.
```

```
    public double dividir() {
        if (num2 == 0) {
            System.out.println("Error: División por cero.");
            return 0; // Retornar 0 o manejar de otra forma según el diseño
requerido.
        }
        return num1 / num2;
    }
}
```

- **Propiedades (num1, num2)**: Estas variables de instancia almacenan los operandos con los que la calculadora realizará las operaciones. Son privadas para proteger su acceso directo desde fuera de la clase. Si fueran públicas se pudieran acceder desde fuera de la clase.
- **Constructor (public Calculadora(double num1, double num2))**: Este método especial se llama al crear un objeto de la clase. Inicializa num1 y num2 con los valores proporcionados como argumentos.
- **Métodos de operación (sumar, restar, multiplicar, dividir)**: Cada uno de estos métodos realiza una operación específica utilizando las propiedades num1 y num2. La división incluye una verificación para evitar la división por cero, un caso especial que debe manejarse para evitar errores en tiempo de ejecución.

Uso de la clase Calculadora

Para utilizar esta clase, se crea una instancia de Calculadora con valores específicos y luego se llaman los métodos para realizar operaciones:

```
import operaciones.Calculadora;

public class Main {
    public static void main(String[] args) {
```

```
        Calculadora calc = new Calculadora(5, 3);
        System.out.println("Suma: " + calc.sumar());
        System.out.println("Resta: " + calc.restar());
        System.out.println("Multiplicación: " + calc.multiplicar());
        System.out.println("División: " + String.format("%.2f",calc.
dividir()));
    }
}
```

Este ejemplo ilustra cómo la clase Calculadora encapsula la funcionalidad aritmética y cómo puede ser reutilizada para operar con diferentes pares de números de manera clara y eficiente.

Modificación de propiedades de instancia

En Java, para modificar las propiedades de un objeto después de haber creado una instancia, las propiedades deben ser accesibles, preferiblemente a través de métodos específicos dentro de la clase. Esto es parte de un principio de diseño en la programación orientada a objetos llamado **encapsulamiento**, que se trata de ocultar los detalles de la implementación de un objeto y exponer solo métodos seguros para interactuar con ese objeto.

Implementando *Getters* y *Setters*

Para modificar las propiedades de una calculadora después de haber creado una instancia, se deberían implementar métodos ***setter*** para cada propiedad que quieras poder modificar, y para poder leer cada valor de las propiedades, es necesario implementar métodos ***getter***. En el siguiente código se presenta la modificación necesaria para implementar estos dos tipos de métodos de instancia.

```
public class Calculadora {
    private double num1;
    private double num2;
```

```
    // Constructor
    public Calculadora(double num1, double num2) {
        this.num1 = num1;
        this.num2 = num2;
    }

    // Getters
    public double getNum1() {
        return num1;
    }

    public double getNum2() {
        return num2;
    }

    // Setters
    public void setNum1(double num1) {
        this.num1 = num1;
    }

    public void setNum2(double num2) {
        this.num2 = num2;
    }

    // Métodos de operaciones
    public double sumar() {
        return num1 + num2;
    }

    public double restar() {
        return num1 - num2;
    }

    public double multiplicar() {
```

```
        return num1 * num2;
    }

    public double dividir() {
        if (num2 == 0) {
            System.out.println("Error: División por cero.");
            return 0;
        }
        return num1 / num2;
    }
}
```

Ejemplo de uso

Una vez definidos los métodos ***setter*** y ***getter***, es posible modificar las propiedades de una instancia de Calculadora en cualquier momento después de su creación. Veamos cómo podríamos utilizar estos métodos:

```
import operaciones.Calculadora;

public class Main {
    public static void main(String[] args) {
        // Crear una instancia con valores iniciales
        Calculadora calc = new Calculadora(12.0, 3.0);

        // Utilizar la calculadora con los valores iniciales
            System.out.println("Valores iniciales: " + calc.getNum1()
+ ", " + calc.getNum2());
        System.out.println("División inicial: " + calc.dividir());

            // Modificar las propiedades después de la creación
        calc.setNum1(15.0);
        calc.setNum2(5.0);
        System.out.println("Nuevos valores: " + calc.getNum1() + ", "
```

```
        + calc.getNum2());

            // Usar la calculadora con los nuevos valores
            System.out.println("Nueva división: " + calc.dividir());
        }
    }
```

Al permitir la modificación de propiedades de un objeto, es importante considerar las implicaciones en términos de integridad y consistencia de los datos. Los métodos ***setter*** proporcionan un punto de control centralizado donde puedes validar los datos antes de asignarlos a las propiedades, asegurando que el objeto siempre se mantenga en un estado válido. Por ejemplo, podrías prevenir la asignación de un valor negativo a una propiedad que debe ser siempre positiva, o asegurarte de que el denominador en una operación de división nunca sea cero antes de que ocurra un intento de división.

Implementar *getters* y *setters* es una manera efectiva de mantener el control sobre cómo se accede y se modifica el estado interno de los objetos, haciendo tu código más seguro y fácil de mantener.

Sobrecarga de constructores y métodos

Sobrecarga de constructores

La sobrecarga de constructores en Java permite definir múltiples constructores en una clase, cada uno con una firma diferente. Esto significa que los constructores pueden tener diferentes listas de parámetros (en número y tipo), lo que permite inicializar objetos de múltiples maneras, dependiendo de la información disponible en el momento de la creación.

La sobrecarga de constructores es una técnica útil para mejorar la flexibilidad y legibilidad del código, permitiendo a las y los desarrolladores proporcionar diferentes maneras de configurar objetos según las necesidades específicas de cada situación.

Ejemplo de Calculadora con sobrecarga de constructores

Consideremos nuestra clase Calculadora que puede realizar operaciones básicas como suma, resta, multiplicación y división. Además, queremos que esta calculadora tenga la capacidad de calcular el cuadrado de un número o el factorial de un número utilizando diferentes constructores.

Clase Calculadora (Calculadora.java)

```
public class Calculadora {
    private double operando1;
    private double operando2;

    // Constructor para operaciones binarias
    public Calculadora(double operando1, double operando2) {
        this.operando1 = operando1;
        this.operando2 = operando2;
    }

    // Constructor para operaciones unarias (como cuadrado o factorial)
    public Calculadora(double operando1) {
        this.operando1 = operando1;
        // Utilizamos operando2 como un flag o marcador, -1 podría
indicar que no se usa
        this.operando2 = -1;
    }

    // Métodos para realizar operaciones
    public double sumar() {
        return operando1 + operando2;
    }

      public double restar() {
        return operando1 - operando2;
    }
```

```
    public double multiplicar() {
        return operando1 * operando2;
    }

    public double dividir() {
        if (operando2 == 0) {
            throw new IllegalArgumentException("División por cero no
permitida.");
        }
        return operando1 / operando2;
    }

    // Método para calcular el cuadrado del primer operando
    public double cuadrado() {
        if (operando2 == -1) {
            return operando1 * operando1;
        }
        throw new IllegalStateException("Operación no válida para dos
operandos.");
    }

    // Método para calcular el factorial del primer operando
    public double factorial() {
        if (operando2 == -1) {
            return factorial((int) operando1);
        }
        throw new IllegalStateException("Operación no válida para dos
operandos.");
    }

    private double factorial(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("Factorial de número
```

```
        negativo no definido.");
            }
            double result = 1;
            for (int i = 2; i <= n; i++) {
                result *= i;
            }
            return result;
        }
    }
```

Uso de la clase Calculadora con constructores sobrecargados

```
    public class Main {
        public static void main(String[] args) {
            Calculadora calcBinaria = new Calculadora(5, 3);
            System.out.println("Suma: " + calcBinaria.sumar());
            System.out.println("Producto: " + calcBinaria.multiplicar());

            Calculadora calcUnaria = new Calculadora(5);
            System.out.println("Cuadrado: " + calcUnaria.cuadrado());
            System.out.println("Factorial: " + calcUnaria.factorial());
        }
    }
```

La sobrecarga de constructores en la clase Calculadora permite que los objetos se inicialicen para distintos tipos de operaciones matemáticas, ya sean binarias o unarias. Esta flexibilidad facilita el uso de la clase en diferentes contextos, sin necesidad de establecer condiciones iniciales no aplicables. Además, los métodos adecuados garantizan que las operaciones sólo se realicen en el contexto correcto, mejorando la seguridad y la integridad del programa.

Sobrecarga de métodos

La sobrecarga de métodos, también conocida como *overloading*, es una característica de la programación orientada a objetos que permite a una clase tener varios métodos con el mismo nombre, pero con diferentes listas de parámetros (en tipo, número o ambos). Esto permite realizar tareas similares con diferentes tipos de datos o con diferente número de argumentos, mejorando la legibilidad del código y la usabilidad de la clase.

Características de la sobrecarga de métodos

- **Mismo nombre de m**étodo: los métodos sobrecargados comparten el mismo nombre dentro de una clase.
- **Lista de parámetros diferentes**: los métodos deben diferir en el tipo de dato y/o número de sus parámetros.
- **Retorno y acceso**: los métodos sobrecargados pueden tener diferentes tipos de retorno y modificadores de acceso, aunque estos no son factores para diferenciar métodos sobrecargados.

Ejemplo de Calculadora con métodos sobrecargados

Vamos a implementar ahora que la calculadora pueda realizar operaciones aritméticas básicas como suma y multiplicación, aceptando tanto enteros como números de punto flotante. Esto mostrará cómo la sobrecarga de métodos puede ser utilizada para manejar diferentes tipos de datos de entrada.

Clase Calculadora con métodos sobrecargados (Calculadora.java)

```
public class Calculadora {

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

```
    // Sobrecarga de método para sumar números de punto flotante
    public double sumar(double a, double b) {
        return a + b;
    }

    // Sobrecarga de método para multiplicar enteros
    public int multiplicar(int a, int b) {
        return a * b;
    }

    // Sobrecarga de método para multiplicar números de punto flotante
    public double multiplicar(double a, double b) {
        return a * b;
    }
}
```

En este ejemplo, cada operación aritmética (suma y multiplicación) está implementada dos veces: una para trabajar con *int* y otra con *double*. Esto permite que la calculadora maneje adecuadamente las operaciones sin pérdida de precisión dependiendo del tipo de datos que se utilice, ofreciendo así flexibilidad en el uso de la clase:

- **Sumar y multiplicar con *int***: Estos métodos se utilizan cuando los números involucrados son enteros. El resultado también es un entero.
- **Sumar y multiplicar con *double***: Estos métodos se utilizan para números de punto flotante. El resultado es también un número de punto flotante, permitiendo precisión decimal.

Uso de la clase Calculadora (Main.java)

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

```
        // Usar métodos para enteros
        System.out.println("Suma de enteros: " + calc.sumar(5, 3));
        System.out.println("Multiplicación de enteros: " + calc.
multiplicar(4, 2));

        // Usar métodos para punto flotante
        System.out.println("Suma de punto flotante: " + calc.
sumar(5.5, 3.3));
        System.out.println("Multiplicación de punto flotante: " + calc.
multiplicar(4.5, 2.0));
    }
}
```

La sobrecarga de métodos en la clase Calculadora demuestra cómo se pueden manejar múltiples tipos de datos de forma eficiente dentro de una misma clase. Esta técnica es extremadamente útil para la creación de APIs intuitivas y fáciles de usar, permitiendo que los métodos realicen operaciones similares en diferentes contextos de datos sin requerir nombres de método distintos para cada acción. Esto simplifica el aprendizaje y la utilización de la clase, haciendo el código más limpio y organizado.

Herencia

La herencia es uno de los pilares fundamentales de la programación orientada a objetos (POO). Permite que una clase (llamada subclase) herede campos y métodos de otra clase (llamada superclase o clase base). Esto facilita la reutilización de código común y promueve una jerarquía lógica dentro del diseño del *software*.

Propósitos de la herencia

1. **Reutilización de código**: permite que las nuevas clases adopten propiedades y comportamientos de las clases existentes.
2. **Simplificación del código**: reduce la complejidad al eliminar

el código redundante y manejar la variabilidad de comportamientos a través de la jerarquía de clases.

3. **Extensibilidad**: facilita la modificación y extensión de la implementación de clases existentes sin alterarlas.

*Sintaxis b*ásica

En Java, la herencia se implementa usando la palabra clave *extends*. Una clase sólo puede extender a una superclase, ya que Java no permite herencia múltiple directamente (aunque se puede lograr un efecto similar utilizando interfaces).

Ejemplo

Consideremos un sistema donde necesitamos modelar diferentes tipos de vehículos. Podemos tener una clase base llamada Vehículo y varias subclases como Automóvil, Motocicleta y Camión (omitir los acentos).

Clase base: Vehículo (Vehiculo.java)

```
public class Vehiculo {
    private String marca;
    private int año;

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

    public void mostrarDetalles() {
        System.out.println("Marca: " + marca + ", Año: " + año);
    }

        // Getters y Setters
    public String getMarca() {
        return marca;
    }
```

```
    public void setMarca(String marca) {
        this.marca = marca;
    }

    public int getAño() {
        return año;
    }

    public void setAño(int año) {
        this.año = año;
    }
}
```

Subclase: Automóvil (Automovil.java)

```
public class Automovil extends Vehiculo {
    private String modelo;

    public Automovil(String marca, int año, String modelo) {
        super(marca, año);  // Llama al constructor de la clase base
        this.modelo = modelo;
    }

    @Override
    public void mostrarDetalles() {
        super.mostrarDetalles();  // Llama al método de la clase base
        System.out.println("Modelo: " + modelo);
    }
}
```

Subclase: Motocicleta (Motocicleta.java)

```
public class Motocicleta extends Vehiculo {
    private boolean tieneSidecar;
```

```
    public Motocicleta(String marca, int año, boolean tieneSidecar) {
        super(marca, año);
        this.tieneSidecar = tieneSidecar;
    }

    @Override
    public void mostrarDetalles() {
        super.mostrarDetalles();
        System.out.println("Tiene sidecar: " + (tieneSidecar ? "Sí" :
"No"));
    }
}
```

Subclase: Camión (Camion.java)

```
public class Camion extends Vehiculo {
    private double capacidadCarga; // Capacidad de carga en toneladas
    private boolean tieneRemolque; // Indica si el camión tiene
remolque

    public Camion(String marca, int año, double capacidadCarga,
boolean tieneRemolque) {
        super(marca, año);  // Llama al constructor de la clase base
Vehiculo
        this.capacidadCarga = capacidadCarga;
        this.tieneRemolque = tieneRemolque;
    }

    // Método para mostrar los detalles específicos del camión
    @Override
     public void mostrarDetalles() {
        super.mostrarDetalles();  // Llama al método de la clase base
        System.out.println("Capacidad de Carga: " + capacidadCarga
+ " toneladas");
```

```
        System.out.println("Tiene Remolque: " + (tieneRemolque ? "Sí" : "No"));
    }

    // Getters y Setters
    public double getCapacidadCarga() {
        return capacidadCarga;
    }

    public void setCapacidadCarga(double capacidadCarga) {
        this.capacidadCarga = capacidadCarga;
    }

    public boolean isTieneRemolque() {
        return tieneRemolque;
    }

    public void setTieneRemolque(boolean tieneRemolque) {
        this.tieneRemolque = tieneRemolque;
    }
}
```

Uso de las clases

```
public class Main {
    public static void main(String[] args) {
        Automovil miAuto = new Automovil("Toyota", 2021, "Corolla");
        Motocicleta miMoto = new Motocicleta("Harley Davidson", 2019, true);
        Camion miCamion = new Camion("Volvo", 2020, 5.0, true);

        miAuto.mostrarDetalles();
        miMoto.mostrarDetalles();
        miCamion.mostrarDetalles();
    }
}
```

Este ejemplo ilustra cómo se puede usar la herencia para especializar y expandir clases en sistemas de objetos, facilitando la reutilización de código y reduciendo la redundancia. Las subclases Automóvil, Motocicleta y Camión extienden Vehículo y se benefician de sus propiedades y métodos comunes, al tiempo que añaden sus propias características únicas. Esto ilustra cómo la herencia puede ser usada para crear una estructura de clases eficiente y organizada, facilitando la extensibilidad y mantenimiento del código en aplicaciones más grandes.

Polimorfismo

El polimorfismo es uno de los conceptos fundamentales en la programación orientada a objetos (POO). El término polimorfismo proviene del griego "*poli*" (muchos) y "*morphos*" (forma), y en el contexto de la POO, se refiere a la capacidad de un objeto de adoptar muchas formas. Esencialmente, permite que las subclases de una clase puedan definir sus propios comportamientos únicos, mientras comparten la misma interfaz, es decir, los mismos métodos de la clase base.

Tipos de polimorfismo

1) **Polimorfismo de inclusión (o subtipado)**

   Ocurre cuando una subclase es tratada como su superclase. Los objetos de una subclase pueden ser tratados como objetos de la superclase en lugares como argumentos de métodos, en colecciones que aceptan tipos de la superclase, etc.

12) **Polimorfismo de sobrecarga (overloading)**

   Sucede cuando dos o más métodos en una clase tienen el mismo nombre, pero parámetros diferentes (en número o tipo). Esto permite realizar una función similar de diferentes maneras, proporcionando flexibilidad en la manera de interactuar con un objeto.

Ejemplo de polimorfismo de inclusión

Vamos a definir una clase base Animal y algunas subclases como Perro y Gato. Cada subclase tendrá una implementación específica de un método hacerSonido().

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

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

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

Uso del polimorfismo de inclusión

```
public class TestPolimorfismo {
    public static void main(String[] args) {
        Animal miAnimal = new Animal();
        Animal miPerro = new Perro();
```

```
        Animal miGato = new Gato();

        miAnimal.hacerSonido(); // Imprime "Algún sonido..."
        miPerro.hacerSonido(); // Imprime "Guau Guau" -
Polimorfismo
        miGato.hacerSonido(); // Imprime "Miau Miau" - Polimorfismo

        hacerSonidoAnimal(miGato); // Se puede pasar cualquier
subclase de Animal
    }

    public static void hacerSonidoAnimal(Animal animal) {
        animal.hacerSonido();
    }
}
```

Ejemplo de polimorfismo de sobrecarga

Ahora vamos a ver un ejemplo de cómo se puede sobrecargar métodos dentro de una clase.

```
public class Calculadora {
    // Método para sumar dos enteros
    public int sumar(int a, int b) {
        return a + b;
    }

    // Método para sumar tres enteros
    public int sumar(int a, int b, int c) {
        return a + b + c;
    }

    // Método para sumar dos doubles
    public double sumar(double a, double b) {
        return a + b;
    }
}
```

Uso del polimorfismo de sobrecarga

```
public class TestCalculadora {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();
        System.out.println(calc.sumar(5, 3));  // Usa el primer método
        System.out.println(calc.sumar(5, 3, 2));  // Usa el segundo
método
        System.out.println(calc.sumar(5.0, 3.0));  // Usa el tercer
método
    }
}
```

El polimorfismo en Java permite diseñar y escribir código que es más flexible y fácil de extender. La capacidad de utilizar la misma interfaz para tipos diferentes facilita la reutilización de código y la implementación de sistemas más complejos de manera más sencilla y mantenible. Además, la sobrecarga de métodos ofrece versatilidad en cómo se pueden realizar las operaciones, ajustándose a las necesidades específicas del contexto en que se utilizan.

Conclusiones

En este capítulo hemos explorado en profundidad el concepto de paquetes en Java, destacando su importancia fundamental para la organización y modularidad del código. A lo largo del capítulo, se ha demostrado cómo los paquetes contribuyen significativamente a evitar conflictos de nombres, mejorar la mantenibilidad del código, controlar el acceso a clases y métodos, y fomentar la reutilización de código. Se han presentado ejemplos prácticos que ilustran la creación y uso de paquetes, desde una simple calculadora hasta estructuras más complejas que involucran herencia y polimorfismo. Además, se han abordado conceptos avanzados como la sobrecarga de constructores y métodos, que permiten una mayor flexibilidad en el diseño de clases. En conjunto, este capítulo ha proporcionado una base sólida para comprender

cómo los paquetes y los principios de programación orientada a objetos pueden ser utilizados para crear aplicaciones Java robustas, mantenibles y escalables.

Documentos consultados

Oracle. (s.f.a). *Definición y propósito de los paquetes en Java*. https://docs.oracle.com/javase/tutorial/java/package/packages.html

Oracle. (s.f.b). *Estructura y organización de paquetes en Java*. https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html

Oracle. (s.f.c). *Cómo importar paquetes y clases en Java*. https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html

Oracle. (s.f.d). *Conceptos de modularidad y reutilización de código en Java*. https://docs.oracle.com/javase/tutorial/java/concepts/

Oracle. (s.f.e). *Control de acceso y encapsulamiento con paquetes en Java*. https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Capítulo 9. Manejo de excepciones en Java

Introducción

El manejo de excepciones en Java es un mecanismo fundamental para tratar errores en tiempo de ejecución, permitiendo que el programa reaccione adecuadamente y continúe su ejecución o se cierre de manera controlada. Las excepciones ayudan a manejar situaciones anormales que pueden ocurrir durante la ejecución de un programa, como errores de entrada/salida, problemas de red, errores de formato de datos, entre otros.

Conceptos básicos

1) ***Exception:*** es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de instrucciones del programa.
2) ***Try-Catch:*** es la estructura básica para manejar excepciones. El bloque *try* contiene el código que puede generar una excepción, y el bloque *catch* define cómo manejar la excepción si esta ocurre.
3) ***Finally:*** un bloque opcional que se ejecuta después de los bloques *try* y *catch*, independientemente de si se lanzó una excepción o no. Es útil para cerrar recursos o limpiar el estado, como cerrar un archivo o liberar un recurso de red.
4) ***Throw:*** se utiliza para lanzar explícitamente una excepción, permitiendo que el programa pase el control a uno de los manejadores de excepciones.

5) ***Throws:*** se utiliza en la declaración de un método para indicar que el método puede lanzar una excepción. Obliga a los invocadores del método a manejar o propagar la excepción.

Ejemplo con el manejo de excepciones en operaciones aritméticas

Supongamos que tenemos un método para dividir dos números. La división por cero es un error común que puede causar la terminación abrupta de un programa si no se maneja adecuadamente.

Archivo: Calculadora.java

```
public class Calculadora {

    // Método que realiza la división y maneja la excepción de división por
cero.
    public double dividir(int numerador, int denominador) {
        try {
            return numerador / denominador;
        } catch (ArithmeticException e) {
            System.err.println("Error: Intento de división por cero");
            return Double.NaN; // Retorna "Not a Number" para indicar
un resultado indefinido.
        } finally {
            System.out.println("Operación de división evaluada");
        }
    }

    public void metodoRiesgoso() throws IOException {
        throw new IOException("Se ha producido un error de E/S");
    }
}
```

Archivo: Main.java

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

        // División segura
        double resultado = calc.dividir(10, 0);
        System.out.println("Resultado: " + resultado);

        try {
            calc.metodoRiesgoso();
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Explicación del código

- **División segura**: el método dividir intenta realizar una división y maneja ArithmeticException, que puede lanzarse si el denominador es cero. El bloque *finally* asegura que se imprime un mensaje cada vez que se intenta la operación, independientemente de si fue exitosa o no.
- **Método riesgoso**: declara que puede lanzar una IOException. Al no manejar esta excepción dentro del mismo método, cualquier llamador del método debe manejar o declarar esta excepción. En *main*, el método se llama dentro de un bloque *try* y la excepción se maneja en un bloque *catch* correspondiente.

Otros ejemplos

Vamos a explorar otras técnicas para el manejo de excepciones en Java. Estos ejemplos ayudarán a ilustrar cómo puedes implementar estas técnicas en tu código para gestionar errores de manera efectiva y elegante.

1. Excepciones personalizadas

Crear una excepción personalizada para manejar casos específicos, como un error de saldo insuficiente en una cuenta bancaria.

```
public class SaldoInsuficienteException extends Exception {
    public SaldoInsuficienteExcepción(String mensaje) {
        super(mensaje);
    }
}

public class CuentaBancaria {
    private double saldo;

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

    public void retirar(double cantidad) throws
SaldoInsuficienteException {
        if (cantidad > saldo) {
            throw new SaldoInsuficienteExcepción("Fondos
insuficientes. Intento de retirar: " + cantidad + ", Saldo disponible: "
+ saldo);
        }
        saldo -= cantidad;
        System.out.println("Retiro exitoso. Saldo actual: " + saldo);
    }

    public static void main(String[] args) {
        CuentaBancaria cuenta = new CuentaBancaria(1000);
            try {
```

```
            cuenta.retirar(1500);
        } catch (SaldoInsuficienteException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Explicación del código

- **Clase SaldoInsuficienteException**: Esta es una excepción personalizada que extiende *Exception*. Se usa para indicar que una operación de retiro no puede completarse debido a fondos insuficientes.
- **Clase CuentaBancaria**: Contiene un método retirar que lanza una SaldoInsuficienteException si el monto a retirar es mayor que el saldo disponible. Este método encapsula la lógica de negocio relativa al manejo del saldo de una cuenta bancaria.
- **Manejo de la excepción**: En el método *main*, intentamos retirar una cantidad mayor al saldo disponible, lo que dispara la excepción personalizada. La excepción se captura y se maneja mostrando un mensaje de error al usuario.

2. Cadenas de excepciones (exception chaining)

Este ejemplo muestra cómo encadenar excepciones para agregar contexto o cambiar el tipo de excepción sin perder la información original.

```
public class ProcesadorDeDatos {
    public void procesar() throws Exception {
        try {
            // Simula un error
            throw new IOException("Error de entrada/salida");
        } catch (IOException e) {
            throw new Exception("Error procesando datos", e);
        }
    }
```

```
    public static void main(String[] args) {
        ProcesadorDeDatos procesador = new ProcesadorDeDatos();
        try {
            procesador.procesar();
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.err.println("Causa original: " + e.getCause());
        }
    }
}
```

Explicación del código

- **Método procesar**: intenta ejecutar una operación que puede lanzar una IOException. En el catch, la IOException se atrapa y se lanza una nueva excepción genérica (*Exception*), incluyendo la excepción original como causa.
- **Propósito**: encadenar excepciones ayuda a mantener la pila de errores original mientras se añade más información o se cambia el tipo de excepción lanzada para adaptarse mejor al contexto del error.

3. Excepciones de tiempo de ejecución (unchecked exceptions)

Ejemplo de cómo lanzar y manejar excepciones en tiempo de ejecución, que no necesitan ser declaradas ni capturadas explícitamente.

```
public class Validador {
    public static void validarEdad(int edad) {
        if (edad < 18) {
            throw new IllegalArgumentException("Acceso denegado -
Debes ser mayor de 18 años.");
        }
        System.out.println("Acceso concedido.");
    }
```

```
        public static void main(String[] args) {
            try {
                validarEdad(16);
            } catch (IllegalArgumentException e) {
                System.err.println(e.getMessage());
            }
            }
    }
```

Explicación del código

- **Lanzamiento de IllegalArgumentException**: se lanza si la edad es menor de 18, indicando que la o el usuario no cumple con un requisito de edad.
- **Características**: al ser una excepción de tiempo de ejecución, IllegalArgumentException no requiere ser declarada ni capturada obligatoriamente, simplificando el código y evitando la necesidad de bloques de código *try-catch* innecesarios.

4. Propagación de excepciones

Un método propaga una excepción a su llamador, que debe manejarla o propagarla aún más.

```
    public class GeneradorDeErrores {
        public void hacerAlgoRiesgoso() throws IOException {
            throw new IOException("Algo salió mal con el archivo.");
        }

        public static void main(String[] args) {
            GeneradorDeErrores generador = new GeneradorDeErrores();
            try {
                generador.hacerAlgoRiesgoso();
            } catch (IOException e) {
                System.err.println("Error capturado: " + e.getMessage());
            }
        }
    }
```

Explicación del código

- **Método hacerAlgoRiesgoso**: lanza una IOException para simular un error en una operación de archivo.
- **Propagación de excepciones**: el método declara que lanza IOException, lo que significa que cualquier código que lo llame debe manejar esta excepción, ya sea con un bloque *try-catch* o declarando a su vez que lanza la excepción.

Nota: El manejo de excepciones es importante para construir aplicaciones robustas y confiables en Java. Permite controlar el flujo del programa incluso cuando ocurren errores, proporcionando una manera de recuperarse de situaciones inesperadas. Utilizar adecuadamente *try, catch, finally, throw*, y *throws* es fundamental para escribir código seguro y fácil de mantener.

El manejo de excepciones en Java es una técnica vital para la creación de aplicaciones robustas y confiables. Este mecanismo permite a los programas manejar adecuadamente los errores en tiempo de ejecución, asegurando que el flujo del programa pueda continuar de manera controlada o finalizar de forma segura en situaciones imprevistas.

Conclusiones

En este capítulo, hemos cubierto los conceptos básicos del manejo de excepciones, incluyendo la estructura *try-catch-finally*, y las palabras clave *throw* y *throws*. Estos elementos constituyen la base para gestionar errores comunes, como los de entrada/salida y de formato de datos. Además, hemos visto cómo aplicar estos conceptos en ejemplos prácticos, como la división por cero y la gestión de excepciones en métodos que pueden lanzar errores específicos.

También exploramos técnicas avanzadas como la creación de excepciones personalizadas, lo que permite una gestión más precisa de errores específicos en el contexto del negocio, como en el caso de saldo insuficiente en una cuenta bancaria. La técnica de encadenamiento de excepciones proporciona una manera eficaz de mantener la información del error original mientras se añade contexto adicional. Asimis-

mo, vimos la importancia de las excepciones en tiempo de ejecución, que simplifican el código al no requerir declaración explícita, y la propagación de excepciones, que obliga a los llamadores a manejar o propagar errores.

Documentos consultados

Baeldung. (s.f.a). *A Guide to Java Exceptions*. https://www.baeldung.com/java-exceptions

Baeldung. (s.f.b). *Custom Exceptions in Java*. https://www.baeldung.com/java-new-custom-exception

Baeldung. (s.f.c). *Java Exception Chaining*. https://www.baeldung.com/java-exception-chaining

GeeksforGeeks. (s.f.). *Exception Handling in Java with Examples*. https://www.geeksforgeeks.org/exception-handling-in-java/

Oracle. (s.f.a). *Excepciones (Java Platform SE 7)*. https://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html

Oracle. (s.f.b). *Excepciones en Java: Gestión y Manejo*. https://docs.oracle.com/javase/tutorial/essential/exceptions/

Oracle. (s.f.c.). *Java Try...Catch...Finally*. https://docs.oracle.com/javase/tutorial/essential/exceptions/try.html

Oracle. (s.f.d). *How to Throw Exceptions in Java*. https://docs.oracle.com/javase/tutorial/essential/exceptions/throwing.html

Oracle. (s.f.e). *How to Use the Throws Keyword in Java*. https://docs.oracle.com/javase/tutorial/essential/exceptions/declaring.html

Oracle. (s.f.f). *Unchecked Exceptions – The Controversy*. https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html

Capítulo 10. Reforzando tus conocimientos en programación Java y temas clave para convertirte en un experto

Introducción

Convertirse en un experto en programación Java requiere una combinación de aprendizaje teórico, práctica constante y la adquisición de competencias demandadas por la industria del *software*. Este capítulo te guiará sobre cómo reforzar tus conocimientos en Java y te proporcionará una lista de temas esenciales que debes dominar para destacarte en el campo de la programación y satisfacer las exigencias del sector.

Estrategias para reforzar tus conocimientos en Java

Estudio continuo y actualización

El estudio continuo es vital para mantenerse relevante en el campo de la programación, especialmente con un lenguaje tan dinámico como Java. Las nuevas versiones de Java se lanzan regularmente, cada una introduciendo nuevas características y mejoras. Para estar al día, es esencial leer la documentación oficial proporcionada por Oracle, que ofrece información detallada sobre las nuevas características y cambios. Además, los libros avanzados escritos por expertos de la industria, como *Effective Java* de Joshua Bloch, son recursos invaluables que proporcionan conocimientos profundos y mejores prácticas.

Participar en conferencias y seminarios web también es una excelente manera de mantenerse actualizado. Estos eventos no sólo ofrecen la oportunidad de aprender de los líderes de la industria, sino que también permiten conectarse con otros profesionales y compartir conocimientos. Finalmente, suscribirse a revistas y *blogs* técnicos puede ofrecer *insights* sobre las tendencias emergentes y casos de uso innovadores de Java.

Práctica constante

La práctica constante es crucial para solidificar tus conocimientos y habilidades en programación. Plataformas como LeetCode, HackerRank y Codewars ofrecen desafíos de programación que varían en dificultad, lo que te permite aplicar conceptos teóricos en situaciones prácticas. Estos ejercicios ayudan a mejorar tu lógica y habilidades para resolver problemas.

Además, trabajar en proyectos de código abierto es una excelente manera de ganar experiencia práctica. Contribuir a repositorios en GitHub no sólo mejora tus habilidades de codificación, sino que también te enseña sobre la colaboración en equipo, el control de versiones y las revisiones de código. Los proyectos de código abierto a menudo abordan problemas del mundo real, proporcionando una experiencia valiosa que se puede traducir directamente al entorno laboral.

Desarrollo de proyectos

El desarrollo de proyectos personales te permite aplicar lo que has aprendido de manera creativa y autónoma. Al trabajar en proyectos desde cero, puedes enfrentar desafíos únicos que requieren soluciones innovadoras. Estos proyectos pueden variar desde pequeñas aplicaciones de consola hasta aplicaciones web complejas o aplicaciones móviles.

Trabajar en proyectos colaborativos, por otro lado, simula un entorno de trabajo profesional. Aprender a utilizar herramientas de gestión de proyectos como JIRA, y sistemas de control de versiones como Git,

es esencial. Además, desarrollar habilidades blandas como la comunicación efectiva y la colaboración en equipo es igualmente importante.

Participación en la comunidad

Ser parte de la comunidad de desarrolladores de Java te ofrece una red de apoyo y aprendizaje. Foros como Stack Overflow y grupos de discusión en Reddit son excelentes lugares para hacer preguntas y compartir conocimientos. Estos foros están llenos de desarrolladores y desarrolladoras con experiencia que pueden ofrecer consejos y soluciones a problemas específicos.

Asistir a conferencias, talleres y *meetups* también es beneficioso. Estos eventos proporcionan oportunidades para aprender sobre las últimas tendencias y tecnologías, participar en sesiones prácticas y conocer a otros profesionales en el campo. Las conferencias anuales de Java, como JavaOne, son eventos destacados donde los expertos de la industria presentan nuevas ideas y tecnologías emergentes.

Cursos y certificaciones

Inscribirse en cursos avanzados y obtener certificaciones reconocidas es una excelente manera de validar tus habilidades. La certificación Oracle Certified Professional Java Programmer (OCPJP) es particularmente valiosa, ya que es reconocida mundialmente y demuestra un alto nivel de competencia en Java.

Los cursos en línea ofrecidos por plataformas como Coursera, edX y Udacity también son recursos valiosos. Estos cursos a menudo incluyen proyectos prácticos que te permiten aplicar lo que has aprendido. Además, muchos de estos cursos son impartidos por expertos de la industria, lo que garantiza que estás aprendiendo las mejores prácticas y técnicas actualizadas.

Temas clave para convertirte en un experto en Java

Fundamentos avanzados de Java

Un conocimiento profundo de los fundamentos de Java es esencial para cualquier desarrollador avanzado. Esto incluye una comprensión sólida de las estructuras de control como bucles, condicionales y operadores. Además, es crucial dominar los tipos de datos y sus conversiones, así como el manejo de excepciones.

El manejo de excepciones es particularmente importante, ya que permite a las y los desarrolladores escribir código robusto que puede manejar errores de manera elegante. Aprender a depurar código eficazmente también es fundamental. Herramientas de depuración como Eclipse y IntelliJ IDEA ofrecen características avanzadas que pueden ayudarte a identificar y solucionar problemas de manera más eficiente. Programación orientada a objetos (POO)

La programación orientada a objetos (POO) es un paradigma central en Java. Comprender y aplicar los principios de POO, como encapsulación, herencia, polimorfismo y abstracción, es esencial. Estos principios no sólo ayudan a organizar el código de manera más eficiente, sino que también facilitan su mantenimiento y reutilización.

Los patrones de diseño son otra área crucial en la POO. Patrones como Singleton, Factory, Observer y Decorator proporcionan soluciones probadas a problemas comunes de diseño de *software*. Implementar estos patrones en Java te permite escribir código más limpio, modular y escalable.

Colecciones y genéricos

La API de colecciones de Java proporciona una variedad de estructuras de datos predefinidas que facilitan el almacenamiento y la manipulación de grupos de objetos. Dominar el uso de listas, conjuntos, mapas y colas es crucial para escribir código eficiente. Además, comprender cómo funcionan las implementaciones internas de estas colecciones puede ayudarte a elegir la estructura de datos más adecuada para cada situación.

Los genéricos son otra característica poderosa de Java. Permiten la creación de clases, interfaces y métodos que operan sobre tipos especificados por el usuario, lo que aumenta la reutilización del código y mejora la seguridad en tiempo de compilación. Aprender a usar genéricos de manera efectiva es esencial para escribir bibliotecas y APIs flexibles y seguras.

Entradas/salidas y serialización

El manejo de entradas y salidas (I/O) es una habilidad fundamental para cualquier persona desarrolladora de Java. Esto incluye la lectura y escritura de archivos, así como el manejo de flujos de datos. Java proporciona una API de I/O robusta que permite trabajar con diferentes tipos de flujos, como *byte streams* y *character streams*.

La serialización es el proceso de convertir un objeto en un formato que pueda ser almacenado o transmitido y luego reconstruido más tarde. Esto es crucial para la persistencia de datos y la comunicación entre aplicaciones. Aprender a serializar y deserializar objetos de manera eficiente te permitirá desarrollar aplicaciones que pueden guardar y recuperar su estado, lo cual es esencial en aplicaciones distribuidas y sistemas complejos.

Concurrencia y multithreading

La programación concurrente permite que un programa realice múltiples tareas simultáneamente, lo que es crucial para aprovechar al máximo las capacidades de *hardware* moderno. Java proporciona una API de concurrencia robusta que incluye clases y interfaces como *Thread, Runnable, Executor* y *CompletableFuture*.

Dominar las técnicas de programación multihilo y aprender a manejar correctamente los problemas de concurrencia, como la sincronización, los bloqueos y la consistencia de memoria, es esencial para desarrollar aplicaciones eficientes y seguras. Las herramientas de *profiling* y *debugging* de concurrencia pueden ayudarte a identificar y solucionar problemas relacionados con el rendimiento y la concurrencia.

Desarrollo de interfaces de usuario/a

El desarrollo de interfaces de usuario/a (UI) es una habilidad importante para crear aplicaciones que sean intuitivas y fáciles de usar. Java ofrece dos principales *frameworks* para el desarrollo de UIs: Swing y JavaFX. Swing es un *toolkit* más antiguo que aún es ampliamente utilizado, mientras que JavaFX es más moderno y proporciona capacidades avanzadas para el diseño de interfaces gráficas.

Aprender a manejar eventos y a diseñar interfaces de usuario de manera efectiva es crucial. Esto incluye la comprensión de los patrones de diseño de interfaces y las mejores prácticas para crear UIs que sean tanto funcionales como atractivas. Las herramientas de diseño visual, como Scene Builder para JavaFX, pueden facilitar el proceso de crea-ción de interfaces.

Desarrollo web con Java

El desarrollo web es una de las áreas más dinámicas y en demanda en la industria del *software*. Con Java, puedes desarrollar aplicaciones web robustas y escalables utilizando tecnologías como *servlets*, JavaServer Pages (JSP) y *frameworks* avanzados como Spring y Hibernate.

Los *servlets* y JSP permiten la creación de aplicaciones web dinámicas, mientras que *frameworks* como Spring facilitan la gestión de la configuración, el control de la inversión y la creación de aplicaciones de nivel empresarial. Hibernate, por otro lado, es un *framework* de mapeo objeto-relacional que simplifica la persistencia de datos. Domi-nar estas tecnologías te permitirá desarrollar aplicaciones web eficien-tes y mantenibles.

Bases de datos y persistencia

La interacción con bases de datos es una parte fundamental del desarrollo de aplicaciones. Java proporciona la API JDBC para conectarse y trabajar con bases de datos relacionales. Aprender a utilizar JDBC de manera eficiente es esencial para realizar operaciones CRUD (Crear, Leer, Actualizar y Borrar) en bases de datos.

La Java Persistence API (JPA) y *frameworks* como Hibernate proporcionan una capa de abstracción sobre JDBC, facilitando la persistencia de objetos Java en bases de datos. Comprender y utilizar estas herramientas te permitirá desarrollar aplicaciones que manejan datos de manera eficiente y segura.

Servicios web y APIs

Los servicios web permiten la comunicación entre aplicaciones a través de la red, utilizando estándares como HTTP y SOAP. Con Java, puedes desarrollar servicios web RESTful y SOAP que facilitan la integración entre sistemas.

Los *frameworks* como Spring Boot simplifican el desarrollo de microservicios, permitiéndote crear aplicaciones modulares y escalables. Aprender a diseñar e implementar APIs efectivas es crucial para el desarrollo de aplicaciones modernas y la integración con otros sistemas y servicios.

Seguridad en aplicaciones Java

La seguridad es un aspecto crítico del desarrollo de *software*. Implementar autenticación y autorización correctamente es esencial para proteger aplicaciones Java. *Frameworks* como Spring Security proporcionan herramientas robustas para gestionar la seguridad en aplicaciones web.

Además, es importante seguir las mejores prácticas de seguridad, como el manejo seguro de datos sensibles, la protección contra vulnerabilidades comunes como inyecciones SQL y *cross-site scripting* (XSS), y el uso de cifrado para proteger datos en tránsito y en reposo. Dominar estos conceptos te permitirá desarrollar aplicaciones seguras y proteger la información de las y los usuarios.

Pruebas y calidad del código

Escribir pruebas es una parte fundamental del desarrollo de *software* de calidad. Herramientas como JUnit y Mockito te permiten crear pruebas unitarias y de integración que aseguran que tu código funcione correctamente y se mantenga robusto ante cambios. Además, aprender

técnicas de refactorización y utilizar herramientas para el análisis de código estático te ayudará a mantener y mejorar la calidad de tu código a lo largo del tiempo. Un enfoque riguroso en pruebas y calidad del código es esencial para desarrollar *software* confiable y mantenible.

Optimización y rendimiento

Optimizar el rendimiento de las aplicaciones es crucial para asegurar que sean rápidas y eficientes. Esto incluye la identificación de cuellos de botella, el uso eficiente de recursos y la optimización del uso de memoria. Herramientas de *profiling* como VisualVM y JProfiler te permiten analizar el rendimiento de tus aplicaciones, así como identificar áreas para mejorar. Aprender técnicas de optimización y cómo aplicar estas herramientas te permitirá desarrollar aplicaciones que no sólo funcionen bien, sino que también proporcionen una experiencia de usuario superior.

Conclusiones

Reforzar tus conocimientos en Java y convertirte en un experto requiere dedicación y un enfoque integral en tu aprendizaje. Dominar los temas clave mencionados no sólo te ayudará a desarrollar competencias técnicas avanzadas, sino que también te preparará para satisfacer las demandas de la industria del *software*. Con esfuerzo continuo y una actitud proactiva hacia el aprendizaje, podrás destacarte en el campo de la programación en Java y contribuir de manera significativa al desarrollo de soluciones innovadoras en el mundo del *software*.

Autoras | Autores

Walter A. Mata López

ORCID: 0000-0002-8107-2182

Ingeniero en sistemas computacionales por el Instituto Tecnológico de Colima, Maestro en ciencias, área computación por la Facultad de Ingeniería Mecánica y Eléctrica de la Universidad de Colima; tiene un Diploma de Estudios Avanzados (DEA) en Métodos y técnicas avanzadas de desarrollo de *software* por parte de la Universidad de Granada, España; es Doctor en socioformación y sociedad del conocimiento por el Centro Universitario CIFE, México, y está en proceso de la realización de la tesis del Doctorado en tecnología educativa por el Centro Escolar Mar de Cortés, México. Actualmente es profesor investigador de tiempo completo en la Facultad de Ingeniería Mecánica y Eléctrica de la Universidad de Colima. Sus intereses se centran en el desarrollo de *software*, analítica de datos, la inteligencia artificial aplicada a la educación y la creación de herramientas de aprendizaje asistidas por IA.

Mónica Cobián Alvarado

ORCID: 0009-0007-8441-0873

Ingeniera en sistemas computacionales por parte del Instituto Tecnológico de Colima, Maestra en ciencias, área computación por la Facultad de Ingeniería Mecánica y Eléctrica de la Universidad de Colima; tiene un Diploma de Estudios Avanzados (DEA) en Métodos y técnicas avanzadas de desarrollo de *software* por parte de la Universidad de Granada, España. Tiene más de 20 años de experiencia como docente

impartiendo materias como: matemáticas discretas, análisis y diseño de sistemas, seminario de investigación, entre otras. Actualmente es profesora por asignatura en la Facultad de Telemática de la Universidad de Colima. Su área de interés se enfoca en la aplicación de tecnología educativa en la ingeniería, con especial atención en la inteligencia artificial (IA), la analítica de datos y el desarrollo de herramientas de aprendizaje asistidas por IA.

Armando Román Gallardo

ORCID: 0000-0001-5912-4428

Actualmente se desempeña como profesor investigador de tiempo completo en la Facultad de Telemática de la Universidad de Colima. Su formación académica reside en la Ingeniería en sistemas computacionales y la Maestría en ciencias computacionales por la Universidad de Colima, así como el Doctorado en educación por la Universidad de Baja California. Sus intereses de investigación se centran en los procesos de desarrollo de *software*, el internet de las cosas e IAs generativas.

José Román Herrera Morales

ORCID: 0000-0003-0811-4187

Profesor investigador de tiempo completo en la Facultad de Telemática de la Universidad de Colima. Cursó la carrera de Ingeniería en comunicaciones y electrónica, y la Maestría en ciencias, área telemática, por la Universidad de Colima. Realizó su Doctorado en tecnologías de la información en la Universidad de Guadalajara. Sus intereses de investigación incluyen los sistemas de búsqueda y recuperación de información, los sistemas inteligentes, la tecnología web, minería y las bases de datos.

Pensamiento algorítmico y programación con Java. Aplicaciones para ingeniería, de Walter Alexander Mata López, Mónica Cobián Alvarado, Armando Román Gallardo y José Román Herrera Morales, fue editado en la Dirección General de Publicaciones de la Universidad de Colima, avenida Universidad 333, Colima, Colima, México, www.ucol.mx. La edición se terminó en diciembre de 2024. En la composición tipográfica se utilizó la familia Adobe Garamond Pro. El tamaño del libro es de 22.5 cm de alto por 16 cm de ancho. Programa editorial no periódico: Eréndira Cortés Ventura. Gestión administrativa: María Inés Sandoval Venegas. Cuidado de la edición: Irma Leticia Bermúdez Aceves. Diseño de portada: Adriana Minerva Vázquez Chávez.

Pensamiento algorítmico y programación con Java. Aplicaciones para ingeniería se ha concebido con el propósito de servir como una guía exhaustiva para el estudiantado de ingeniería, enfocada en enseñar los principios básicos y avanzados del lenguaje de programación Java, al permitirles descubrir su elegancia y eficiencia, a la par de promover una comprensión que les inspire a explorar y crear soluciones innovadoras en el vasto mundo del desarrollo de *software*.

Java se caracteriza por su portabilidad, robustez y versatilidad, atributos que le han permitido mantenerse vigente y ser un software ampliamente utilizado en diversos campos de la tecnología, desde aplicaciones móviles hasta sistemas empresariales complejos. Este texto busca no sólo enseñar los fundamentos del lenguaje, sino también fomentar una comprensión profunda de los principios subyacentes que hacen de Java una herramienta tan poderosa.

El contenido se estructura progresivamente, comenzando con una introducción a los orígenes y evolución de Java, pasando por la instalación del entorno de desarrollo y la creación de programas básicos. Se exploran los elementos esenciales como la sintaxis, tipos de datos, operadores y manejo de excepciones. Además, se abordan temas avanzados como la programación orientada a objetos, polimorfismo y manejo de paquetes, proporcionando ejemplos prácticos y ejercicios para reforzar el aprendizaje.

ISBN: 978-607-8984-60-2
9 786078 984602

UNIVERSIDAD DE COLIMA