

INTRODUCCIÓN A LA PROGRAMACIÓN

MANUAL DEL LENGUAJE JAVA

Autor: Hugo Pelayo Aseko

Fecha: 1 de abril de 2023

Asignatura: Programación

Este manual cubre aspectos básicos y generales relacionados con la programación, con un enfoque alrededor del lenguaje de programación Java. Se empieza por introducir la programación en general abstrayendo de los detalles de los lenguajes de programación.

Al final del manual se agrega un apéndice donde se pueden consultar las definiciones de términos relevantes al manual que sean de difícil comprensión.

Índice

Introducción	4
Desarrollo de software	5
El programa	5
Programación en Java.....	8
Un poco de historia	8
Utilidades necesarias para empezar a programar en Java	9
Configuración de entorno de desarrollo para Java en Visual Studio Code	9
IntelliJ IDEA para la programación en Java	13
Estructura de un programa	13
Elementos básicos de Java	14
Máquina virtual Java.....	14
Sintaxis	15
Generics.....	21
Entrada y salida de datos.....	22
Salida de datos	22
Entrada de datos	25
Mediante el BufferedReader	26
Mediante Scanner	27
Estructuras de control	29
Estructuras secuenciales	29
Estructuras selectivas	29
Estructuras iterativas.....	33
Estructuras de salto	35
Arrays	37
Colecciones	41
Otras clases	44
Clases envoltorio	46
Programación Orientada a Objetos	47
Apéndice	52
Bibliografía.....	54

Introducción

Se entiende por software a aquel sistema dotado de componentes lógicos que posibilitan la realización de tareas bien definidas, constituye entonces el conjunto de instrucciones que debe ser ejecutado por el **hardware**.

El software por lo general se desarrolla utilizando lenguajes de programación de alto nivel. Son lenguajes que utilizan una estructura semántica más próxima a los lenguajes humanos y, por tanto, más fáciles de entender para los programadores. Debido a que el hardware entiende únicamente el lenguaje máquina, que está formado exclusivamente por cadenas de unos y ceros, el software escrito en lenguaje de alto nivel debe ser compilado primero antes de ser ejecutado por nuestro hardware, interpretado o, en el caso de algunos lenguajes de alto nivel, ambos a la vez.

Aunque sea poco común, para el desarrollo de software se utiliza también el lenguaje ensamblador, que es un lenguaje que constituye una capa inmediatamente superior al lenguaje máquina. Una de las diferencias principales entre estos dos lenguajes (que también motiva su uso) y los de alto nivel se basa en que estos últimos abstraen al programador de la arquitectura para la cual se desarrolla la aplicación, de tal modo que podemos desarrollar aplicaciones en lenguajes de alto nivel sin tener mucho conocimiento sobre la arquitectura destino. Sin embargo, este no es el caso para el lenguaje ensamblador que al igual que el lenguaje máquina, va de la mano con la arquitectura.

El software por lo general se clasifica en tres tipos acorde a las funciones las cuales este ha de realizar, tenemos entonces el software de sistema, software de programación y el software de aplicación.

El software de sistema está destinado a administrar el sistema informático, esto es, proveer al programador de una interfaz de alto nivel o conjunto de herramientas para su diagnóstico y mantenimiento aislando al mismo tiempo al programador de los detalles internos del sistema.

El software de programación es el conjunto de aplicaciones que permiten al programador desarrollar programas informáticos, es decir, desarrollar software, en este grupo se incluyen los editores de texto, los compiladores, entornos de desarrollo integrados (IDE), entre otros. Este software dota al programador de herramientas prácticas para el desarrollo de aplicaciones permitiendo el uso de lenguajes de programación tanto de alto nivel como de bajo nivel mencionados con anterioridad.

Por último, tenemos el software de aplicación. Se trata de un conjunto de software destino a un usuario final para la realización de tareas específicas que pueden ser automatizadas o realizadas mediante la asistencia del software. Se incluye dentro de este grupo las aplicaciones ofimáticas como el Microsoft Word, aplicaciones para la manipulación de bases de datos como MySQL Workbench, entre otros.

Para el desarrollo de software se sigue normalmente un conjunto de pasos para llegar al producto final. Es un proceso que llega a ser muy complejo y durar mucho tiempo dependiendo de la complejidad del producto, de hecho, una aplicación mala de esta metodología puede llevar el desarrollo de un producto al fracaso o imponer retrasos prolongados sobre este.

Desarrollo de software

El desarrollo de software acostumbra a involucrar varias fases: análisis de requisitos, diseño, implementación o codificación, pruebas unitarias y de integración, implantación del producto en el entorno de uso y finalmente el mantenimiento, etapa que acostumbra a ser más longeva.

La fase de análisis de requisitos es la primera que se realiza siempre en un proyecto de desarrollo de software (en realidad esto se aplica a cualquier proyecto). En esta fase se especifican las características funcionales (qué debe hacer nuestro software) y las no funcionales (requisitos de nuestro sistema, problemas de escalabilidad, confiabilidad, entre otros). En esta fase normalmente participa un analista junto a varios programadores para analizar los detalles del producto que se desea obtener. Acostumbra a ser una fase muy difícil ya que sienta las bases del proyecto, una fase de análisis cadente puede dificultar el avance del proyecto.

En el diseño es una fase posterior a la fase de análisis donde se define cómo se pretende cumplir con los requisitos especificados en la fase de análisis. Conviene entonces, identificar una variedad de soluciones posibles para el problema que se está intentando solucionar, evaluarlas y determinar de estas soluciones cuál es la que más nos conviene. Habiendo optado por una solución, se procede a optar por las herramientas de desarrollo necesarias para el proyecto.

En la implementación se realizan las tareas de programación, en esencia pasamos a implementar los esquemas resultantes de la fase de diseño. En esta fase participan mayoritariamente los programadores.

El período de pruebas se puede separar en dos partes, pruebas unitarias o pruebas de integración. Esta fase básicamente se encarga de poner a prueba el producto y determinar si cumple con las especificaciones establecidas anteriormente. En el caso de pruebas unitarias, probamos las piezas pequeñas de nuestro software que pueden ser procedimientos, módulos, funciones, clases, entre otros; las pruebas de integración se realizan una vez se han realizado de manera exitosa las pruebas unitarias, en este caso nos aseguramos que el sistema completo funciona correctamente.

Finalmente pasamos a la producción, donde el producto es transferido al entorno de uso del usuario final (o el cliente que lo haya solicitado). Después de esta fase tenemos el mantenimiento que es un proceso de control, mejora y optimización de nuestro producto que ya se encuentra en los equipos de los usuarios finales. Este período acostumbra a ser el más longevo del proceso de desarrollo de software. Durante esta fase, se pueden realizar revisiones e incluso mejoras sobre el producto si hubiese la necesidad.

El programa

Como ya se había mencionado anteriormente todo software consta de programas. Existe cierta confusión entre programa y el software en sí, un programa es una pieza del software. Se considera programa a una secuencia de instrucciones u órdenes escritas en un lenguaje de programación cualquiera que han de ser interpretadas por la computadora para realizar una

tarea específica, esta secuencia de pasos ha de ser finita, bien definida y concisa, se conocen generalmente con el nombre de algoritmo.

El conjunto de expresiones que componen un programa se conoce como el código del programa. Este se escribe en lenguajes de programación acorde a un paradigma de programación que se adapte a las necesidades del programador. Los paradigmas de programación más comunes se clasifican en dos grandes grupos: el imperativo y el declarativo. En el paradigma imperativo el programador instruye a la máquina a través del código a cómo cambiar su estado; en el paradigma declarativo se instruye a la máquina sobre las propiedades del resultado esperado de ciertas operaciones y no a cómo realizarlas.

Entre las herramientas más comunes para el diseño de algoritmos destacan los diagramas de flujo y el pseudocódigo:

El pseudocódigo, también conocido con el nombre de lenguaje algorítmico, es una forma de describir con lenguaje natural el flujo de ejecución de un algoritmo o programa de forma compacta. Normalmente utiliza las construcciones de un lenguaje de programación real, pero está diseñado para la interpretación por parte de humanos ya que abstrae las particularidades de los lenguajes de programación facilitando el enfoque en la lógica y funcionamiento del programa. Por lo general se utiliza el pseudocódigo en libros, textos científicos que exponen algoritmos y también es usado para la planificación de algoritmos para programas complejos. Esto permite, por ejemplo, descubrir posibles errores de lógica en nuestros algoritmos antes de proceder a su implementación. Como se ha mencionado previamente, el pseudocódigo sigue una sintaxis arbitraria y no existe cierta convención ya que el objetivo principal es simplemente exponer la solución a un problema de la forma más sencilla posible. Sin embargo, cabe recalcar que ciertos IDE como PSeInt utilizan pseudocódigo con sintaxis propia para el desarrollo de algoritmos.

```
AlgoritmoEjercicio4

INICIO
# Fecha:          16 Septiembre 2022
# Autor:          Hugo Pelayo
# Descripción:    Este programa pide al usuario su nombre y muestra por pantalla un mensaje con él

# Parte declarativa:
nombre_usuario: cadena de caracteres

# Cuerpo del algoritmo
mostrarPorPantalla("Entre su nombre, por favor: ")
leer nombre_usuario

mostrarPorPantalla("Hola ", nombre_usuario)
FIN
```

Figura 1. Ejemplo de pseudocódigo. Fuente: elaboración propia

El cuerpo del pseudocódigo está constituido normalmente por estructuras de control y estructuras e iterativas o bucles y declaraciones de variables, entre otros aspectos. Las estructuras de control se dividen principalmente en dos grupos, estructuras secuenciales donde tenemos un flujo de ejecución de sentencias continua o, pueden ser selectivas, en cuyo caso el flujo de ejecución de las sentencias varía acorde al resultado de evaluar ciertas expresiones lógicas.

```
instrucción1;  
instrucción2;  
instrucción3;  
...  
instrucciónn;
```

Figura 2 Estructura secuencial. Fuente: Wikipedia

```
Si condición Entonces  
    instrucciones1;  
Si no Entonces  
    instrucciones2;  
Fin Si
```

Figura 3 Estructura selectiva con alternativa. Fuente: Wikipedia

El otro grupo de grupo elementos está constituido por las estructuras iterativas, en las cuales se ejecuta un cierto grupo de instrucciones mientras se cumpla cierta condición, tenemos entonces: el **bucle mientras** donde ejecutamos su cuerpo mientras una condición se satisfaga; tenemos el **bucle hacer** donde, a diferencia del bucle anterior, se ejecuta el cuerpo al menos una vez, cabe destacar que en el caso del bucle anterior si la condición especificada en la cabecera del bucle no se cumple ni en la primera iteración entonces nunca se entra al cuerpo de este; para finalizar tenemos el **bucle repetir** donde ejecutamos un grupo de sentencias un número finito de veces.

```
Mientras condición Hacer  
    instrucciones;  
Fin Mientras
```

Figura 4. Formato del bucle mientras. Fuente: Wikipedia

```
Hacer  
    instrucciones;  
Mientras condición
```

Figura 5. Formato del bucle hacer. Fuente: Wikipedia

```
instrucciones;  
Mientras ¬(condición) Hacer  
    instrucciones;  
Fin Mientras
```

Figura 6. Formato del bucle repetir. Fuente: Wikipedia

Programación en Java

Java es un lenguaje de programación muy utilizado en el desarrollo de aplicaciones web, es multiplataforma y soporta principalmente el paradigma de la programación orientada objetos. Se utiliza también para el desarrollo de aplicaciones móviles y de escritorio. Java en sí es considerado una plataforma por ser un sistema que sirve como base para hacer funcionar ciertos módulos de hardware y software con los que es compatible. Fue comercializado por primera vez en 1995 por Sun Microsystems y desarrollado originalmente por James Gosling. Su sintaxis deriva en gran parte de C y C++, pero ofrece menos soporte para utilidades de bajo nivel que ambos. Las aplicaciones en Java se compilan a bytecode que puede interpretar cualquier máquina virtual de Java indistintamente de la arquitectura del computador en cuestión.

Un poco de historia

Sun definió por primera vez la implementación de referencia original para los compiladores de Java, máquinas virtuales y librerías, trabajo que se publicó en 1995.

Originalmente Java se creó para utilizar en un proyecto sobre un decodificador de señales de televisión en la operación The Green Project en el año 1991. En un principio se llamó Oak, se pasó a llamar Green tras darse cuenta de que Oak ya existía y pertenecía a una marca comercial de adaptadores de tarjetas gráficas, entonces se acabó llamando Java. Hay muchas teorías sobre la elección de este último nombre, sin embargo, destaca más el hecho de que el nombre tiene origen en un tipo de café de una cafetería que frecuentaban sus desarrolladores. El objetivo en un principio era diseñar un lenguaje de programación con una estructura similar a la de C++, aun así, en 1994, sus desarrolladores acabaron orientándolo al entorno Web, porque creyeron que el navegador web Mosaic haría del internet un medio más interactivo, concepto que se asimilaba mucho a la televisión por cable, idea inicial del proyecto de Java. Naughton, uno de los desarrolladores de Java, creó entonces un prototipo de navegador, WebRunner, que más tarde se conocería con el nombre de HotJava.

La promesa inicial de Gosling era “Write once, Run Anywhere” (Escríbelo una vez, ejecútalo en cualquier plataforma), un concepto que promovía la transportabilidad, es decir, la capacidad de ejecutar los mismos códigos en cualquier plataforma que ofreciese soporte para el entorno de ejecución de Java, pudiendo compartir de esta forma los mismos binarios si tener que reescribir el código fuente.

Desde la versión 1.0 del JDK (Java Development Kit), lanzada en enero de 1996, Java ha experimentado números cambios sobre todo en las clases que ofrece su librería estándar. En diciembre de 1998 se lanzó el JDK 1.2 bajo los nombres de Java 2 y J2SE (Java 2 Platform, Standard Edition), esto con el objetivo de distinguir la plataforma base de las ediciones J2EE (Java 2 Platform, Enterprise Edition) y J2ME (Java 2 Platform, Micro Edition).

Entre las características de Java podemos destacar que es fuertemente orientado a objetos, en efecto, el programa principal es un objeto que viene encapsulado en forma de clase. Aquí los objetos son generalmente clases que constan de atributos y los métodos que nos sirven para poder operar sobre estos atributos. Otro aspecto importante a destacar en este aspecto es el soporte para objetos genéricos que se pueden reaprovechar para otras aplicaciones, cosa que agiliza el desarrollo de estas.

Java ofrece independencia de plataforma, por tanto, programas escritos en este lenguaje se pueden ejecutar en cualquier plataforma indistintamente del hardware, siempre y cuando haya soporte para la máquina virtual de Java en la plataforma destino. Para ello los compiladores de Java, traducen los archivos de código fuente de Java a binarios con extensión .class, los cuáles contienen bytecode, un tipo de instrucciones interpretables por la máquina virtual de Java.

A diferencia de C++, lenguaje del cuál Java hereda muchos aspectos sobre el paradigma de la programación orientada a objetos, Java proporciona un recolector de basura para solucionar el problema de las fugas de memoria, error muy común entre programadores de C++. Las fugas de memoria suceden cuando alojamos memoria de forma dinámica (en tiempo de ejecución) y no la desalojamos acabado su uso, esto se convierte entonces en memoria ocupada pero no aprovechable. El *garbage collector* (recolector de basura) de Java se ocupa de este problema.

Utilidades necesarias para empezar a programar en Java

Para empezar a desarrollar aplicaciones en Java lo primero que debemos hacer es decidirnos por una edición, la que más se ajuste a nuestras necesidades (mencionadas con anterioridad). Tenemos entonces a elegir: J2ME, J2SE o J2EE. Esta última se reserva para aplicaciones web más complejas con accesos a bases de datos, entre otros aspectos. Para aplicaciones menos complejas y más o menos completas que se puedan ejecutar en PC, la edición J2SE sería suficiente.

Ya escogido J2SE, debemos descargarnos el SDK (Software Development Kit) de Java que incluye herramientas para desarrollo de aplicaciones Java como el compilador, el depurador, entre otras.

Configuración de entorno de desarrollo para Java en Visual Studio Code

Para configurar un entorno de desarrollo para Java en el editor de código Visual Studio Code deberemos primero instalar el editor en cuestión, seguido de un JDK y finalmente las extensiones que el editor ofrece para la programación con Java. Nos dirigimos entonces al enlace de descarga de VS Code: [Visual Studio Code](#).

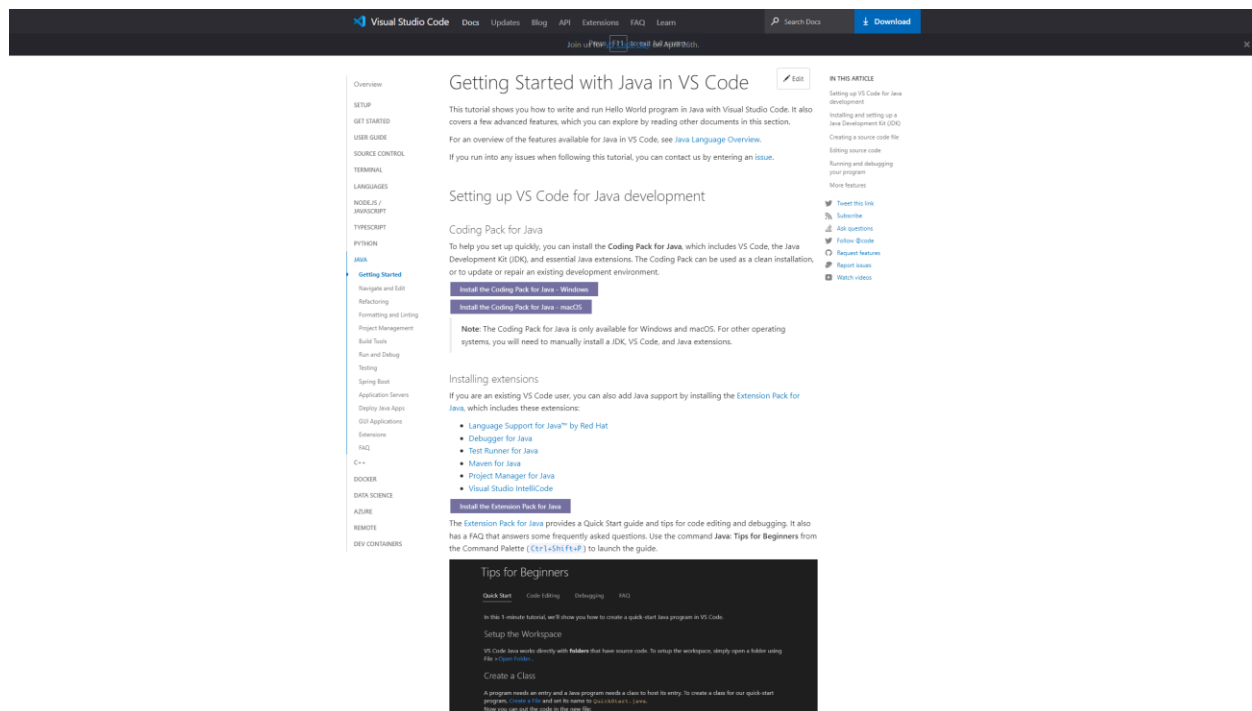


Figura 7. Tutorial Java VS Code. Fuente: Elaboración propia

Debemos asegurarnos primero de que tenemos el JDK instalado, para ello abrimos una terminal (Tecla Windows y escribimos “cmd”), ahora escribimos el siguiente comando sin las comillas: “java --version”, nos debería salir un mensaje como el siguiente:

```
java 17.0.4.1 2022-08-18 LTS
Java(TM) SE Runtime Environment (build 17.0.4.1+1-LTS-2)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.4.1+1-LTS-2, mixed mode, sharing)
```

Figura 8. Captura terminal versión de JDK. Fuente: Elaboración propia

Si no nos sale el mensaje, muy probablemente no tengamos el JDK instalado. Podemos conseguir uno de forma gratuita desde la página mencionada anteriormente. Deslizando un poco hacia abajo, encontraremos un listado de JDKs que podemos descargar, hacemos clic sobre el enlace que está señalado con la marca verde en la imagen de abajo, nos redirigirá a la página a través de la cuál podremos descargar e instalar nuestro JDK.

Installing a Java Development Kit (JDK)

If you have never installed a JDK before and need to install one, we recommend you to choose from one of these sources:

- [Amazon Corretto](#)
- [Azul Zulu](#)
- [Eclipse Adoptium's Temurin](#)
- [Microsoft Build of OpenJDK](#)
- [Oracle Java SE](#) ✓
- [Red Hat build of OpenJDK](#)
- [SapMachine](#)

Figura 9. Listados JDK para VSCode. Fuente: Elaboración propia

Una vez seguros de que tenemos nuestro JDK instalado, tendremos dos vías de instalación para el editor y las herramientas necesarias para desarrollar aplicaciones con Java: podemos descargarnos el ejecutable Coding Pack for Java (disponible sólo para Windows y MacOS), a través de los botones en magenta: Install the Coding Pack for Java - Windows e Install the Coding Pack for Java - macOS, y proceder con la instalación mediante esta aplicación; o podemos realizarla manualmente siguiendo los siguientes pasos:

Hacemos clic sobre el botón azul en la esquina superior derecha de la pantalla que dice “Download” y descargamos el instalador de Visual Studio Code:

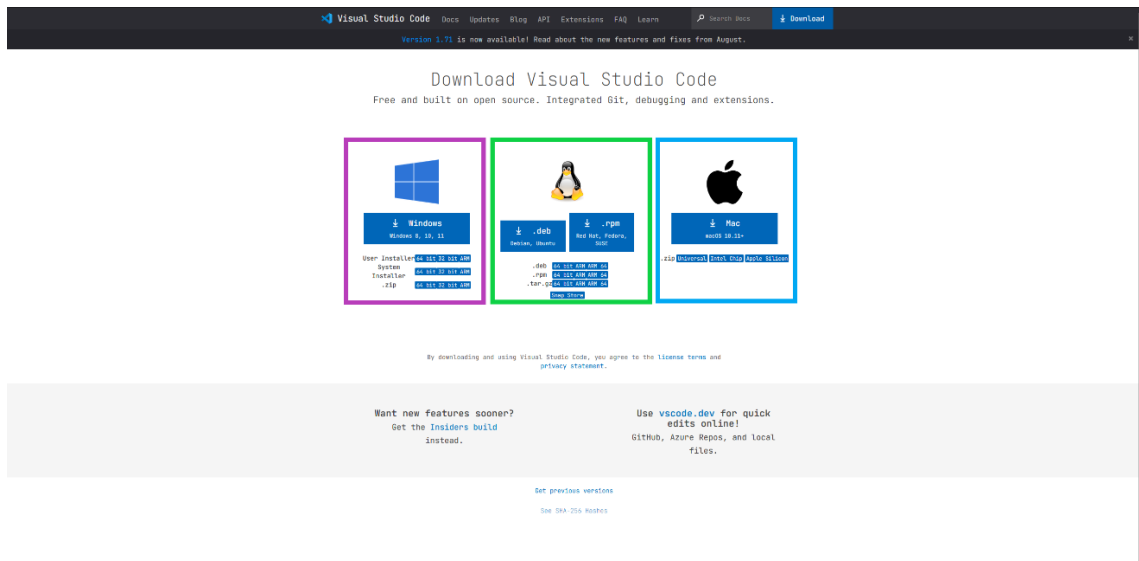


Figura 10. Plataformas para VSCode. Fuente: Elaboración propia

Aquí escogemos el instalador dependiendo de nuestro sistema operativo: el primero es para sistemas operativos Windows, el segundo para sistemas operativos Linux y el tercero para Mac, hacemos clic sobre los recuadros en azul grandes para descargar nuestro instalador e instalamos Visual Studio Code.

Una vez instalado Visual Studio Code, volvemos a nuestra página [VS Code Tutorial](#), e instalamos las extensiones de Java para Visual Studio Code, para ello hacemos clic sobre el botón de color magenta que dice “Install the Extension Pack for Java”.

Installing extensions

If you are an existing VS Code user, you can also add Java support by installing the [Extension Pack for Java](#), which includes these extensions:

- [Language Support for Java™](#) by Red Hat
- [Debugger for Java](#)
- [Test Runner for Java](#)
- [Maven for Java](#)
- [Project Manager for Java](#)
- [Visual Studio IntelliCode](#)

[Install the Extension Pack for Java](#)

The [Extension Pack for Java](#) provides a Quick Start guide and tips for code editing and debugging. It also has a FAQ that answers some frequently asked questions. Use the command `Java: Tips for Beginners` from the Command Palette (`Ctrl+Shift+P`) to launch the guide.

Figura 11. Extensiones de Java para VSCode. Fuente: Elaboración propia

Nos abrirá una ventana de Visual Studio Code que contiene información similar a la siguiente:

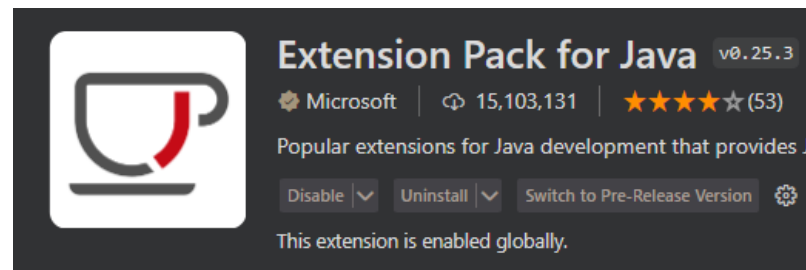


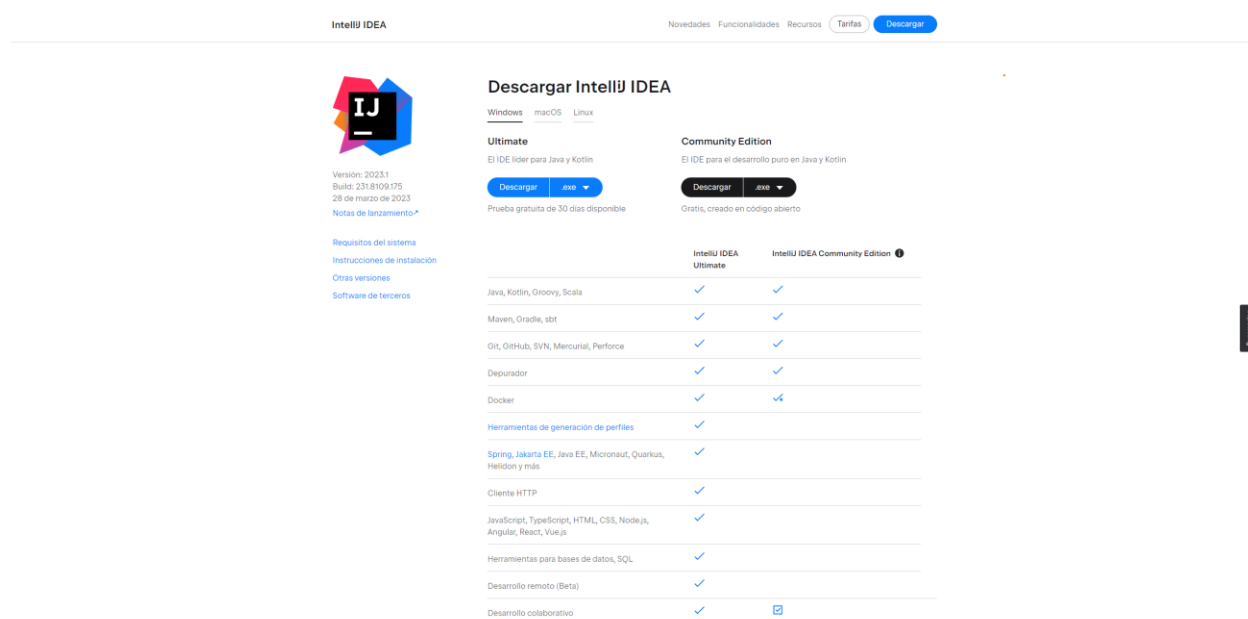
Figura 12. Paquete de extensiones de Java para VSCode. Fuente: Elaboración propia

Cuando se realice una instalación por primera vez, nos saldrá un botón diciendo “**Instalar**” en esta ventana, hacemos clic sobre él y esperamos a que se acaben de instalar las extensiones de Java.

Finalizado el paso anterior ya estaremos listos para crear nuestro programa en Java y ejecutarlo.

IntelliJ IDEA para la programación en Java

Para el desarrollo de aplicaciones Java, se recomienda IDEs como IntelliJ IDEA, NetBeans o Eclipse. IntelliJ IDEA es utilizado incluso por Google como la estructura base para Android Studio, IDE de desarrollo de aplicaciones para Android. IntelliJ IDEA ofrece una versión gratuita mantenida por cierta comunidad de usuarios del producto que se llama IntelliJ IDEA Community, y tenemos la versión de pago que es IntelliJ IDEA Ultimate que ofrece varias tarifas dependiendo del uso que le vayamos a dar y que tipo de entidad formen los usuarios, ya sea uso personal, empresa, etcétera. Estas dos versiones se pueden obtener de su [página oficial](#). Abajo se recoge un listado de diferencias principales entre ambas versiones:



The screenshot shows the IntelliJ IDEA download page. On the left, there's a sidebar with the IntelliJ logo, version information (2023.1, Build: 231.8109.175, 28 de marzo de 2023), and links for system requirements, installation instructions, other versions, and third-party software. The main content area is titled 'Descargar IntelliJ IDEA' and has tabs for Windows, macOS, and Linux. It compares two editions: 'Ultimate' and 'Community Edition'. The Ultimate edition is described as 'El IDE líder para Java y Kotlin' and offers a 30-day free trial. The Community Edition is described as 'El IDE para el desarrollo puro en Java y Kotlin' and is free and open-source. A table below compares features between the two editions.

	IntelliJ IDEA Ultimate	IntelliJ IDEA Community Edition
Java, Kotlin, Groovy, Scala	✓	✓
Maven, Gradle, sbt	✓	✓
Git, GitHub, SVN, Mercurial, Perforce	✓	✓
Depurador	✓	✓
Docker	✓	✓
Herramientas de generación de perfiles	✓	
Spring, Jakarta EE, Java EE, Micronaut, Quarkus, Helidon y más	✓	
Cliente HTTP	✓	
JavaScript, TypeScript, HTML, CSS, Node.js, Angular, React, Vue.js	✓	
Herramientas para bases de datos, SQL	✓	
Desarrollo remoto (Beta)	✓	
Desarrollo colaborativo	✓	✗

Figura 13. Comparativa Ultimate y Community. Fuente: sitio web oficial JetBrains de IntelliJ IDEA

Estructura de un programa

Java es un lenguaje que sigue estrictamente el paradigma de la programación orientado a objetos, por tanto, todo programa consta de una colección de clases. La estructura de un programa en Java es muy similar a la de un programa en C++ o C. Las clases son agrupaciones de datos sobre los cuales se puede operar a través de un conjunto de métodos. Todo programa Java debe contener un sólo método *main()* entre todas las clases que lo forma, aunque pueda estar constituido de varias clases. En el método principal *main()* se inicia la ejecución de nuestro programa.

La estructura estándar de los ficheros de nuestro programa Java siempre empieza por indicar el paquete de la clase que define dicho fichero. A continuación, se importan todas las clases que vaya a utilizar la clase que está definiendo el fichero. Para importar clases de otros paquetes o proyectos java utilizamos la directiva `import`. Y finalmente, tenemos la definición de nuestra clase.

```

1 package poo;
2
3 import java.util.ArrayList;
4 import java.io.InputStreamReader;
5 import java.io.BufferedReader;
6 import java.io.IOException;
7
8 import poo.agencia.Empresa;
9 import poo.agencia.Vehiculo;
10 import poo.agencia.Vehiculo.VehiculoType;
11 import poo.utils.Pair;
12 import poo.agencia.Furgoneta;
13 import poo.agencia.Coche;
14 import poo.agencia.Moto;
15
16 public class PracticaAlquilerVehiculos {
17     private static final InputStreamReader input = new InputStreamReader(System.in);
18     private static final BufferedReader reader = new BufferedReader(input);
19
20     private static ArrayList<Pair<VehiculoType, Vehiculo>> noAlquilados;
21     private static ArrayList<Empresa> empresas;
22
23
24     public static void main(String[] args) {
25         // índice de Empresa
26         int seleccion;
27         // Cuent los días
28         int contadorDias;
29         // Días de alquiler de tipo de Vehiculo elegido
30         int cantidadDeDias;

```

Figura 14. Ejemplo estructura programa en Java. Fuente: elaboración propia

Elementos básicos de Java

Java es un lenguaje con enfoque muy fuerte sobre el paradigma orientado a objetos, los conceptos sobre los cuales este paradigma se apoya son seguidos estrictamente por este lenguaje de programación.

El lenguaje ofrece muchas herramientas para el desarrollo de aplicaciones en forma de librerías con variedad de clases e interfaces. Este lenguaje, a diferencia de otros como C++, ofrece gran seguridad sobre la gestión de nuestro sistema a coste de reducir la flexibilidad a la hora del manejo de recursos del mismo. Java no ofrece instrucciones que nos permitan acceder directamente a memoria como es el caso de los punteros en C++ o C. La máquina virtual, que es el entorno sobre el cual se ejecutan nuestras aplicaciones permite que nuestras aplicaciones se ejecuten de forma segura en el sistema. También, gracias a que todas las aplicaciones Java requieren de la máquina virtual para poder ser ejecutadas, esto mejora la portabilidad, permitiendo que las aplicaciones Java se puedan ejecutar en cualquier sistema, siempre y cuando este tenga una implementación válida de la máquina virtual.

Máquina virtual Java

Una herramienta fundamental en la plataforma Java es la Máquina Virtual Java (JVM, por sus siglas en inglés), que se ejecuta en una plataforma específica y es capaz de interpretar y ejecutar instrucciones escritas en un código binario especial llamado bytecode Java. Es generado por el compilador del lenguaje Java y es considerado un código máquina de bajo nivel, capaz de ser ejecutado incluso por un microprocesador físico.

La JVM actúa como un puente entre el hardware del sistema y la aplicación Java que se está ejecutando, convirtiendo el código bytecode en código nativo del dispositivo final. De esta manera, se logra que las aplicaciones Java sean portables y puedan ser ejecutadas en diferentes plataformas y sistemas operativos, siempre y cuando se cuente con la máquina virtual Java correspondiente.

La máquina virtual Java puede ser implementada en diferentes formas, ya sea en software, hardware, herramientas de desarrollo o navegadores web. La JVM provee definiciones para un conjunto de instrucciones, registros, formato de archivos de clases, pila, heap con recolector de basura y área de memoria. Cualquier implementación de la JVM aprobada por los creadores de Java debe ser capaz de ejecutar cualquier clase que cumpla con la especificación.

El uso de la JVM permite que el código Java sea portable y pueda ejecutarse en cualquier plataforma que cuente con la máquina virtual Java correspondiente. Es por ello que se popularizó el axioma "Write once, run anywhere" o "Escríbelo una vez, ejecútalo en cualquier lugar". Sin embargo, aunque se han intentado construir microprocesadores que acepten el bytecode de Java como lenguaje de máquina, estos intentos no han sido exitosos.

Sintaxis

En cuanto a la sintaxis de Java, es importante tener en cuenta que el lenguaje es sensible a mayúsculas y minúsculas. Esto significa que el compilador distingue entre palabras escritas con diferentes combinaciones de mayúsculas y minúsculas, por lo que escribir "public" no es lo mismo que escribir "Public". De hecho, el segundo caso provocaría un error de compilación al no ser reconocido como una palabra reservada. Este aspecto se aplica no solo a las palabras clave del lenguaje, sino también a los nombres de variables y métodos.

Otro aspecto importante a considerar es que, en Java, cada sentencia debe finalizar con el carácter ";" (punto y coma). Este requisito se puede apreciar en el ejemplo "Holamundo.java", donde cada línea de código termina con este símbolo.

Además, los bloques de instrucciones se delimitan con llaves {..}. Este aspecto se puede observar en los ejemplos anteriores, donde el contenido de una clase y el código de un método se encontraban entre estas llaves.

Por último, cabe destacar que en Java existen dos tipos de comentarios: los de una línea, que se inician con "//", y los de varias líneas, que se delimitan por "/" y "/". Estos comentarios no afectan al código compilado y se utilizan para hacer anotaciones y explicaciones en el código fuente.

Las sentencias en Java están compuestas por expresiones, que son combinaciones de elementos y literales que se evalúan para producir un resultado. En programación, los elementos más pequeños que son significativos y entendidos por el compilador se denominan tokens. En el lenguaje de programación Java, estos tokens se dividen en cinco categorías principales.

Una de estas categorías son los identificadores, que son nombres asignados por el programador a variables, clases, paquetes, métodos y constantes en el código Java. Los identificadores en

Java son case sensitive, lo que significa que una variable llamada "miVariable" es diferente de una variable llamada "MIVARIABLE" o "mivariable". Los identificadores también pueden incluir números y el signo "_" para crear nombres únicos y descriptivos. Java consta de palabras clave las cuales no podemos utilizar como identificadores para nuestras variables ya que están reservadas para el lenguaje, a continuación, vienen listadas:

abstract	boolean	break	byte	case
catch	char	class	continue	default
do	double	else	extends	false
final	finally	float	for	if
implements	import	instanceof	int	interface
long	native	new	null	package
private	protected	public	return	short
static	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while	var	rest
byvalue	cast	const	future	generic
goto	inner	operator	outer	strictfp

Figura 15. Palabras clave. Fuente: Wikipedia

Toda la información en Java se almacena en variables que ya se han descrito con anterioridad, estas variables deben venir descritas por un tipo en el momento de su declaración, esto es necesario ya que Java, o más específicamente, el compilador necesita saber previamente al proceso de compilación el tamaño necesario para guardar estas variables, tenemos de este modo varios tipos básicos que ya vienen incluidos en el lenguaje:

TYPE	EXPLANATION
<code>byte</code>	8-bit signed integer within the range of -128 to 127
<code>short</code>	16-bit signed integer within the range of -32,768 to 32,767
<code>int</code>	32-bit signed integer within the range of -2147483648 to 2147483647
<code>long</code>	64-bit signed integer within the range of -9223372036854775808 to 9223372036854775807
<code>float</code>	single-precision 32-bit floating point within the range of 1.4E-45 to 3.4028235E38
<code>double</code>	double-precision 64-bit floating point within the range of 4.9E-324 to 1.7976931348623157E308
<code>boolean</code>	It can be either <code>true</code> or <code>false</code>
<code>char</code>	single 16-bit Unicode character within the range of <code>\u0000</code> (or 0) to <code>\uffff</code> (or 65,535)

Figura 16. Tabla que muestra los tipos básico de Java con el valor máximo que pueden almacenar. Fuente: freeCodeChamp

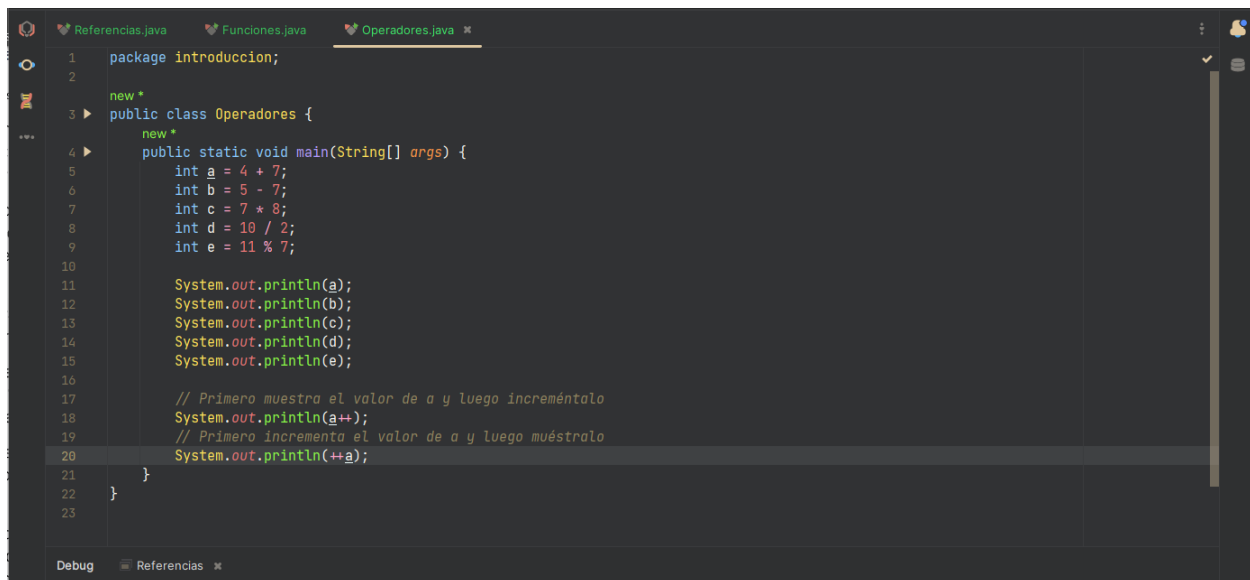
Los operadores en Java son símbolos que permiten realizar diferentes tipos de operaciones, como aritméticas, lógicas, de comparación, de asignación, entre otras. A continuación, se detallan algunos de los operadores más comunes en Java:

Aritméticos: permiten realizar operaciones matemáticas básicas como suma (+), resta (-), multiplicación "*", división "/", módulo "%" e incremento "++" o decremento "--", de estos dos últimos tenemos dos variantes para cada uno, postincremento o postdecremento, y preincremento o predecremento.

Lógicos: se utilizan para comparar dos valores booleanos y obtener un resultado verdadero o falso. Los operadores lógicos son "&&" (AND), "||" (OR) y "!" (NOT).

De comparación: permiten comparar dos valores y obtener un resultado verdadero o falso. Los operadores de comparación son "==" (igual a), "!=" (distinto de), "<" (menor que), ">" (mayor que), "<=" (menor o igual que) y ">=" (mayor o igual que).

De asignación: se utilizan para asignar un valor a una variable. El operador de asignación es el signo igual "=". También existen operadores de asignación compuestos como "+=", "-=", "*=", "/=" y "%=".



```
1 package introduccion;
2
3 public class Operadores {
4     new *
5     public static void main(String[] args) {
6         int a = 4 + 7;
7         int b = 5 - 7;
8         int c = 7 * 8;
9         int d = 10 / 2;
10        int e = 11 % 7;
11
12        System.out.println(a);
13        System.out.println(b);
14        System.out.println(c);
15        System.out.println(d);
16        System.out.println(e);
17
18        // Primero muestra el valor de a y luego incrementalo
19        System.out.println(a++);
20        // Primero incrementa el valor de a y luego muéstralo
21        System.out.println(++a);
22    }
23 }
```

Figura 17. Ejemplo de uso de operadores. Fuente elaboración propia

Las funciones, por otro lado, se conocen más bien como métodos (ya que se definen dentro de clases) y son bloques de código que realizan una tarea específica. Los métodos pueden recibir cero o más parámetros y pueden devolver un valor o ser de tipo void (sin valor de retorno) en cuyo caso se acostumbra a denominar procedimientos. La sintaxis de un método es la siguiente:

```
[modificador] tipoDeRetorno nombreDelMetodo (tipoParametro1 parametro1,
tipoParametro2 parametro2, ...) {
    // Cuerpo del método
    return valorDeRetorno; // necesario si el tipoDeRetorno no es void
}
```

Donde el modificador es una palabra clave que indica el nivel de acceso y otros atributos del método, como public, private, protected, static, final, abstract, entre otros. También puede ser varios de ellos a la vez; el tipoDeRetorno es el tipo de dato que devuelve el método, o void si no devuelve nada; el nombreDelMetodo es el nombre del método en sí, que debe ser descriptivo y seguir las convenciones de nomenclatura de Java; los tipos de parámetros y los nombres de los

misimos se indican entre paréntesis y separados por comas. Los parámetros son variables locales que reciben un valor al momento de llamar al método.

Para llamar a un método, se utiliza el nombre del método seguido de los valores de los parámetros entre paréntesis. Por ejemplo:



```
1 package introduccion;
2
3 public class Funciones {
4     new *
5     public static void main(String[] args) {
6         // Llamamos al método
7         int resultado = suma(a: 3, b: 5);
8
9         System.out.print("El resultado es: " + resultado);
10    }
11
12    1 usage new *
13    public static int suma(int a, int b) {
14        return a + b;
15    }
16 }
```

Figura 18. Ejemplo de llamada a método. Fuente: elaboración propia



```
1 package introduccion;
2
3 public class Variable {
4     public static void main(String[] args) {
5         int variable1 = 11;
6         double variable2 = 4.5;
7         float variable3 = .5f; // igual que 0.5
8         char variable4 = 0x20AC; // símbolo unicode euro
9     }
10 }
11
12 }
```

Figura 19. Ejemplo declaración variables. Fuente: elaboración propia

En Java, existen dos tipos de datos: los tipos primitivos que ya se habían mencionado previamente, y los tipos objeto o referencia. Los tipos primitivos son datos básicos que representan valores simples como enteros, decimales, caracteres y booleanos, mientras que

los tipos objeto son datos más complejos que representan objetos y se almacenan en memoria dinámica.

Los tipos primitivos en Java son ocho: byte, short, int, long, float, double, char y boolean. Estos tipos de datos se definen por sí mismos y no necesitan ser instanciados mediante la palabra clave "new". Los tipos primitivos son almacenados en la memoria de manera directa, lo que significa que su valor real es el que se guarda en la variable.

Por otro lado, los tipos objeto o referencia son aquellos que se crean mediante la palabra clave "new" y representan objetos más complejos, como clases, arrays y otros tipos definidos por el programador. Estos tipos de datos se almacenan en memoria dinámica y se acceden a ellos a través de referencias o punteros que apuntan a la dirección de memoria donde están almacenados los objetos. Al utilizar tipos objeto, se puede acceder a una gran cantidad de funcionalidades y métodos que no están disponibles para los tipos primitivos.

Los tipos primitivos son datos simples y se almacenan directamente en la memoria, mientras que los tipos objeto son más complejos y se almacenan en memoria dinámica a través de referencias.

En Java, cuando se llama a un método que tiene parámetros, se pasa una copia de la referencia a ese objeto como parámetro. Esto significa que la variable que se pasa como parámetro es simplemente otra referencia al mismo objeto que se encuentra en la memoria, por lo tanto, si el método modifica el objeto, esos cambios se reflejarán en todas las referencias a ese objeto en el programa. En otras palabras, el objeto en sí mismo no se copia, solo la referencia a él.

Si se pasa un tipo primitivo como un parámetro, se pasa una copia de su valor en lugar de una referencia. Esto significa que cualquier modificación realizada en la variable dentro del método no afectará el valor original que se pasó al método.

A screenshot of an IDE window titled 'Referencias.java'. The code defines a package 'introduccion', imports 'java.util.Arrays', and creates a class 'Referencias'. Inside the class, there is a 'main' method that initializes an array 'misNumeros' with values {1, 2, 3, 4, 5}, prints it, and then calls a method 'duplicar' with 'misNumeros' as an argument. The 'duplicar' method is defined as 'public static void duplicar(int[] lista)' and contains a loop that increments each element of the 'lista' array by 2. The IDE interface includes a sidebar with icons for Explorer, Search, and Run and Debug, and a top bar with window management icons.

```
1 package introduccion;
2
3 import java.util.Arrays;
4
5 new *
6 public class Referencias {
7     new *
8     public static void main(String[] args) {
9         int[] misNumeros = { 1, 2, 3, 4, 5 };
10        System.out.println(Arrays.toString(misNumeros));
11
12        duplicar(misNumeros);
13        System.out.println(Arrays.toString(misNumeros));
14    }
15
16    1 usage new *
17    public static void duplicar(int[] lista) {
18        for (int indice = 0; indice < lista.length; ++indice)
19            lista[indice] += 2;
20    }
21 }
```

Figura 20. Ejemplo paso de referencias como parámetros. Fuente: elaboración propia

Generics

En Java, los Generics son una característica que permite especificar tipos de datos parametrizados. Es decir, permiten definir clases, interfaces y métodos que puedan trabajar con diferentes tipos de datos de manera genérica, sin necesidad de especificar el tipo concreto al momento de la implementación.

Los Generics se utilizan para asegurar la seguridad de tipos de datos y para hacer que el código sea más reutilizable y legible. Además, también permiten detectar errores de tipo en tiempo de compilación y evitar errores de ejecución.

La sintaxis básica de los Generics en Java es utilizando los caracteres '<' y '>' para delimitar los tipos de datos genéricos. Por ejemplo, una clase genérica que trabaje con cualquier tipo de datos se puede definir de la siguiente manera:



```
3
4 6 usages
5 public class Pair<T, U> {
6     3 usages
7     T m_First;
8     3 usages
9     U m_Second;
10
11     3 usages
12     public Pair(T first, U second) {
13         m_First = first;
14         m_Second = second;
15     }
16
17     1 usage
18     public T getFirst() { return m_First; }
19     1 usage
20     public U getSecond() { return m_Second; }
21
22     no usages
23     public void setFirst(T first) { m_First = first; }
24     no usages
25     public void setSecond(U second) { m_Second = second; }
26 }
```

Figura 21. Generics. Fuente: elaboración propia



```
1 package generics;
2
3 public class PruebaPair {
4     public static void main(String[] args) {
5         // silenciar advertencia unchecked assignment
6         @SuppressWarnings("unchecked")
7         Pair<Integer, String> numerous = new Pair[3];
8
9         numerous[0] = new Pair<>(1, "Uno");
10        numerous[1] = new Pair<>(2, "Dos");
11        numerous[2] = new Pair<>(3, "Tres");
12
13        for (Pair<Integer, String> par : numerous)
14            System.out.println(par.getFirst() + " con representación " + par.getSecond());
15    }
16 }
```

Figura 22. Ejemplo uso Generics. Fuente: elaboración propia

Entrada y salida de datos

Una de las operaciones más habituales que tiene que realizar un programa en Java es intercambiar datos con otros sistemas o con el mismo en el cual se está ejecutando. Para este fin, el SE de Java ofrece el paquete `java.io` que contiene una serie de clases que nos permiten realizar operaciones de lectura y escritura sobre todo tipo de dispositivos abstrayendo al programador de estos mediante el uso de una interfaz común para ellos. Las principales clases para la entrada de datos son `InputStreamReader` y `BufferedReader`, ambas especializaciones de `InputStream`; y para la salida estándar de datos tenemos la clase `PrintStream` de la cual se especializan muchas otras, también tenemos `BufferedWriter`, `InputStreadWriter`, entre otros.

La salida estándar está asociada a la de un proceso en cuestión, es decir, la salida de datos por defecto del proceso sobre el que se ejecuta nuestro programa, en Linux, por ejemplo, la salida estándar de todos los procesos se asocia al terminal, sin embargo, este comportamiento se puede cambiar redireccionando la salida del proceso en cuestión a dispositivos como sockets, ficheros, etcétera.

Salida de datos

Para enviar datos por la salida estándar debemos crear un objeto de tipo `PrintStream`, la clase `System` ya ofrece un atributo llamado `out` para mostrar datos por la salida por defecto. Para el envío de datos esta clase nos ofrece los métodos `print(String)` o `println(String)` que envían cadenas de caracteres por la salida de datos, la principal diferencia entre ambos es que el segundo añade un salto de línea al final (carácter de control `\n`) tras haber enviado el contenido que recibe como parámetro. Cabe destacar que estos métodos ofrecen varias sobrecargas que reciben muchos otros tipos de parámetros, a parte la clase madre de todas, `Object`, ofrece el método `toString()` el cual puede ser redefinido por todas las clases de Java para ofrecer una mejor representación de sí.

Métodos de la clase `PrintStream`:

Firma	Descripción
<code>PrintStream append(...)</code>	Concatena un carácter o secuencia de caracteres a este <code>PrintStream</code>
<code>boolean checkError()</code>	Comprueba el estado de este <code>PrintStream</code>
<code>void clearError()</code>	Restaura el estado de errores de este <code>PrintStream</code>
<code>void close()</code>	Cierra el flujo de datos de este <code>PrintStream</code>
<code>void flush()</code>	Limpia el buffer de datos de este <code>PrintStream</code> una vez que los caracteres se han enviado por la salida de datos
<code>PrintStream format(...)</code>	Formatea los datos que se pasa como parámetro antes de enviarlos por la salida de datos
<code>void print(...)</code>	Envía la cadena de caracteres que se pasa como parámetro por la salida de datos
<code>void printf(...)</code>	Envía la cadena de caracteres que se pasa como parámetro por la salida de datos y permite formatear la cadena

<code>void println(...)</code>	Envía la cadena de caracteres que se pasa como parámetro por la salida de datos y al final añade un salto de línea
<code>void setError(...)</code>	Activa el estado de error de este PrintStream
<code>void write(...)</code>	Escribe bytes por la salida de datos

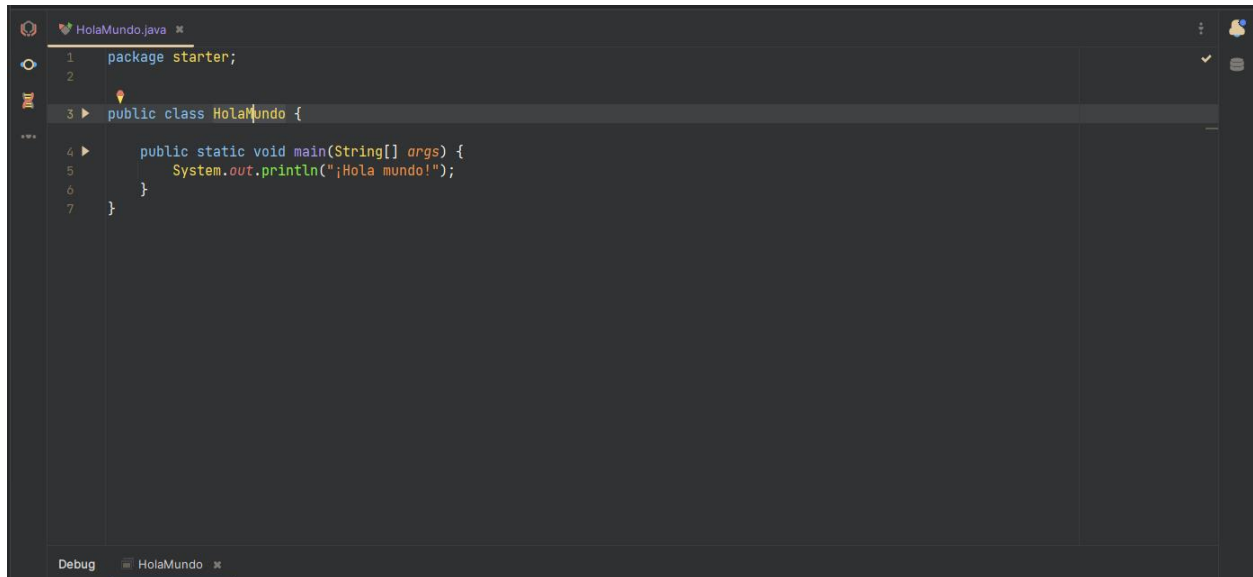


Figura 23. Ejemplo de salida de mensaje. Fuente: elaboración propia

La clase `PrintStream` también ofrece los métodos `printf()` y `format()` desde la versión 5 de Java, estos nos permiten formatear nuestra cadena de caracteres antes de mostrarla por la salida de datos. La cadena que nosotros enviamos a estos métodos está formada por nuestro texto acompañado de una serie de especificadores de formateo en aquellas posiciones donde queramos un formato de muestreo específico, la sintaxis para los especificadores de formateo es estricta y la mínima equivocación en esta puede llevar a Java interpretarlo como texto normal o lanzar una excepción.

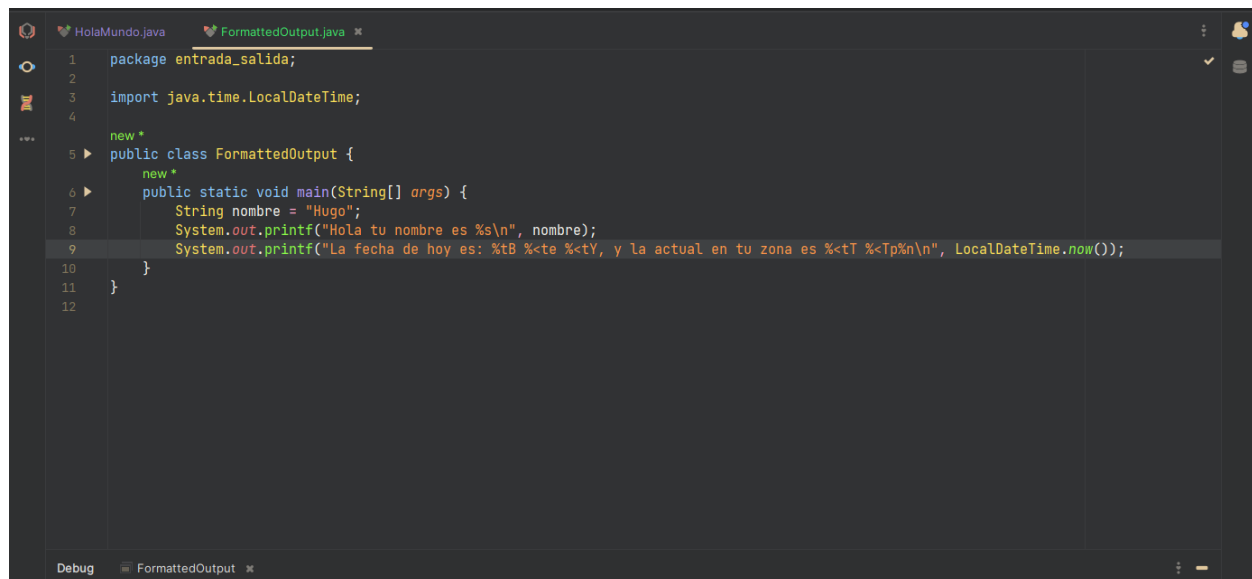
El formato para los especificadores:

```
% [posición_argumento$] [indicador] [mínimo] [.num_decimales] conversión
```

El signo de porcentaje al principio indica que el contenido a continuación representa un especificador de formato, si queremos el signo de porcentaje en sí en nuestra salida utilizamos entonces el `"%%"`. *posición_argumento* es un valor entero positivo que indica cuál de los datos que pasamos como parámetro a nuestro método se asocia al especificador en cuestión, al igual que el resto de especificadores que están entre corchetes, este es opcional. El indicador especifica un formato de salida. *mínimo* representa un número mínimo de caracteres a ser representados. *num_decimales* es un entero positivo que indica el número de decimales con que mostrar un valor en coma flotante, importante destacar que este valor debe venir precedido de un punto (ejemplo `%3f`). Para finalizar tenemos la conversión que indica el tipo de argumento

(String, char, double, int, etcétera). A continuación, se recogen los especificadores de conversión más comunes:

Especificador	Categoría	Descripción
'h' o 'H'	general	Muestra valor retornado por Integer.toHexString(arg.hashCode()) Donde "arg" es nuestro dato
's' o 'S'	general	Muestra valor de un String
'c' o 'C'	carácter	Muestra el resultado de un carácter en Unicode
'x' o 'X'	entero	Muestra valor entero en hexadecimal
'e' o 'E'	coma flotante	Decimal en notación científica
'g' o 'G'	coma flotante	Decimal en notación científica con precisión específica
'a' o 'A'	coma flotante	Decimal en notación científica con un dígito significativo y el exponente correspondiente
'd'	entero	Muestra el valor de un entero en decimal
'f'	coma flotante	Muestra un valor en coma flotante con
'o'	entero	Muestra el valor de un entero en octal



```

1 package entrada_salida;
2
3 import java.time.LocalDateTime;
4
5 new *
6 ▶ public class FormattedOutput {
7     new *
8     ▶ public static void main(String[] args) {
9         String nombre = "Hugo";
10        System.out.printf("Hola tu nombre es %s\n", nombre);
11        System.out.printf("La fecha de hoy es: %tB %<te %<tY, y la actual en tu zona es %<tI %<Tp%n", LocalDateTime.now());
12    }
13 }
  
```

Figura 24. Ejemplo de formateo con especificadores. Fuente: elaboración propia

El ejemplo anterior muestra el resultado siguiente:

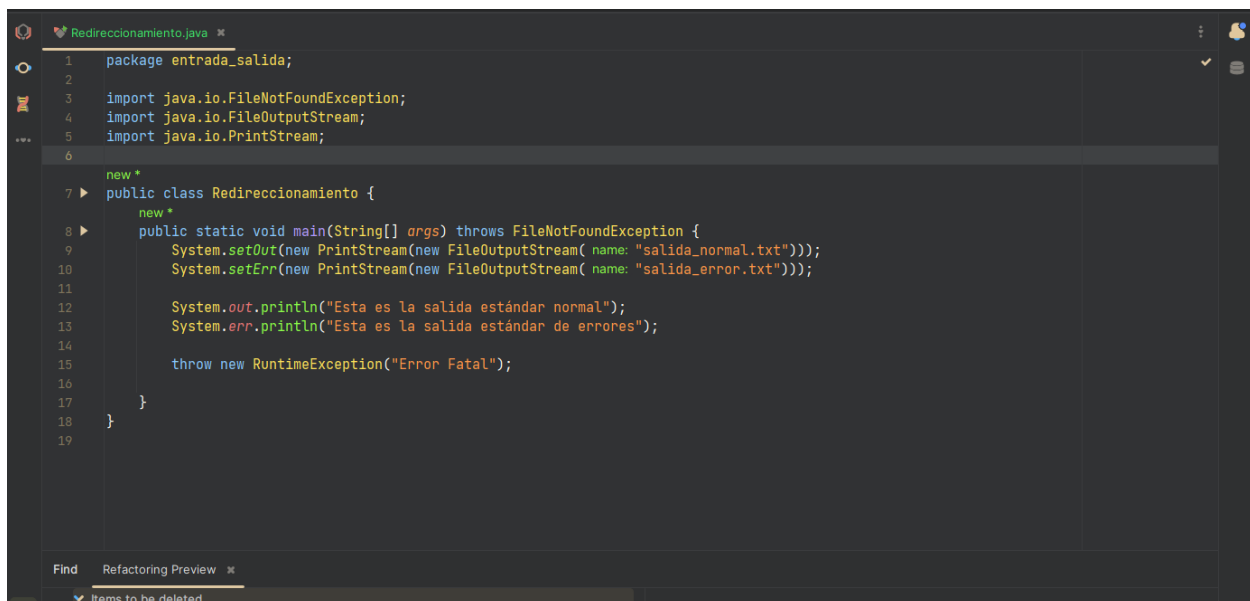
```

Hola tu nombre es Hugo
La fecha de hoy es: April 22 2023, y la actual en tu zona es 16:23:00 PM
  
```

Figura 25. Resultado ejemplo Figura 24. Fuente: elaboración propia

A parte de la salida estándar de datos existe la salida estándar de error de un proceso, que tenemos disponible a través del atributo *err* de la clase *System*. Al igual que el atributo *out*, *err* es un objeto de tipo *PrintStream*. La ventaja de esta diferenciación es que podemos redireccionar la salida estándar de un proceso y la salida estándar de error a ficheros diferentes, por ejemplo, podríamos tener nuestra salida estándar vinculada a la terminal o un fichero de texto con nombre "salida.txt" y en nuestra salida estándar de error a otro fichero con nombre "errores.log" o a la misma terminal. En algunos IDEs como NetBeans, la salida estándar de error se suele mostrar con un color rojo llamativo ya que se va más alarmante, sin embargo, la salida estándar suele estar en un color negro.

Desde java podemos cambiar la salida estándar de nuestro programa o su salida estándar de error utilizando los métodos *setOut()* o *setErr()* que nos ofrece la clase *System*, debemos primero abrir un flujo de datos o stream al fichero que queremos asociar cierta salida mediante la clase *FileOutputStream(File)*.



```
1 package entrada_salida;
2
3 import java.io.*;
4 import java.io.*;
5 import java.io.*;
6
7 new *
8 public class Redireccionamiento {
9     new *
10     public static void main(String[] args) throws FileNotFoundException {
11         System.setOut(new PrintStream(new FileOutputStream( name: "salida_normal.txt")));
12         System.setErr(new PrintStream(new FileOutputStream( name: "salida_error.txt")));
13
14         System.out.println("Esta es la salida estándar normal");
15         System.err.println("Esta es la salida estándar de errores");
16
17         throw new RuntimeException("Error Fatal");
18     }
19 }
```

Figura 26. Ejemplo redireccionamiento salida. Fuente: elaboración propia

Entrada de datos

En Java, para poder recibir datos del usuario a través del teclado se utiliza, por un lado, la clase *Scanner* que se encuentra en el paquete *java.util*. Esta clase permite leer diferentes tipos de datos, como enteros, flotantes, caracteres, cadenas, entre otros. Para poder utilizarla, primero debemos crear un objeto de la clase *Scanner* y luego utilizar sus métodos para leer los datos que necesitamos. Por ejemplo, para leer un entero, utilizamos el método *nextInt()* del objeto *Scanner* y lo almacenamos en una variable de tipo *int*. Es importante tener en cuenta que el método *nextInt()* solo lee el entero y no consume el salto de línea, por lo que puede ser necesario utilizar el método *nextLine()* para consumir ese salto de línea antes de leer la siguiente línea de datos. También es posible utilizar otros métodos de la clase *Scanner* para validar los datos de entrada y evitar errores en la lectura.

Por otro lado, tenemos la clase `BufferedReader`. Esta es una clase de Java que permite leer datos de entrada de una manera más eficiente en comparación con otras clases de lectura de datos como `Scanner`. Esta clase almacena los datos en un buffer temporal antes de leerlos, lo que reduce la cantidad de acceso al disco o a la red y mejora el rendimiento de la aplicación. Además, `BufferedReader` proporciona métodos para leer texto y caracteres de una variedad de fuentes, incluyendo archivos, entradas del usuario y conexiones de red. En general, si se espera leer grandes cantidades de datos de entrada, se recomienda utilizar `BufferedReader` en lugar de otras clases de lectura de datos.

Mediante el `BufferedReader`

Para la lectura de datos de disponemos de la clase `InputStream`. Este objeto está asociado a un fichero. La clase `System` nos ofrece un atributo público y estático al cual podemos acceder llamado `in`. A este atributo está asociado por defecto el teclado.

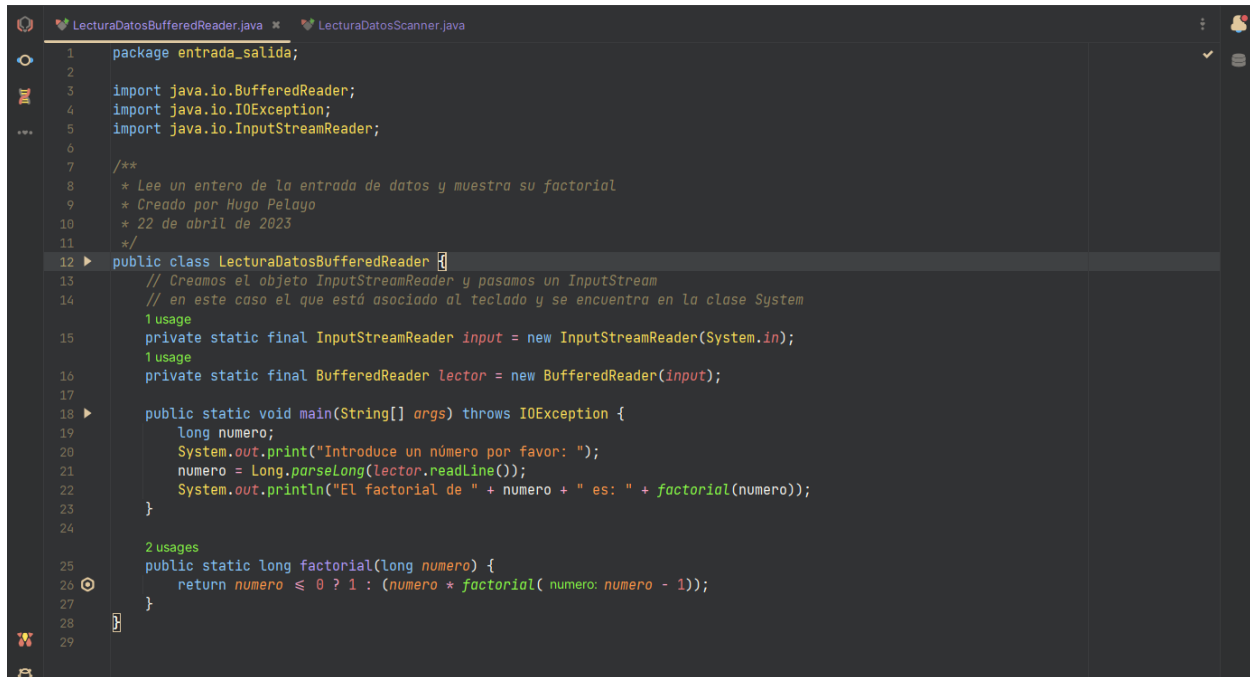
Firma	Descripción
<code>int available()</code>	Retorna el número estimado de bytes que pueden ser leídos sin ser bloquearse en la lectura de datos
<code>void close()</code>	Cierra el flujo de datos de este <code>InputStream</code>
<code>void mark()</code>	Marca la posición actual de este <code>InputStream</code>
<code>boolean markSupported()</code>	Prueba si este <code>InputStream</code> soporta los métodos <code>read()</code> o <code>reset()</code>
<code>void read()</code>	Lee el siguiente byte en el fichero que tiene asociado este <code>InputStream</code>
<code>void reset()</code>	Reposiciona el punto lector de datos de este <code>InputStream</code> a la posición donde estaba en el momento de última llamada a este método
<code>long skip(...)</code>	Salta una cantidad determinada de bytes del flujo de datos de este <code>InputStream</code>

La clase `InputStream` nos ofrece el método `read()` que sólo devuelve el último carácter leído del fichero que está asociado al objeto, por lo tanto, consecutivas llamadas a este método son necesarias para leer varias cadenas de caracteres, lo cual es ineficiente. Para agilizar esta tarea disponemos de la clase `BufferedReader`, cuyo constructor recibe un `InputStream` y nos aporta métodos con los cuales leer cadenas de caracteres del fichero asociado al `InputStream` que recibe en el momento de construcción.

Para leer una cadena de caracteres de la entrada de datos, la clase `BufferedReader` nos ofrece el método `readLine()`, el cual lee una secuencia de caracteres hasta el primer salto de línea y la devuelve en formato `String`. Un punto muy importante de este método es que lanza una excepción de tipo `IOException` que está marcada y por tanto es necesario capturar o volver a lanzar desde el método que se invoque.

Es importante destacar que el método `readLine()` devuelve una cadena de caracteres en un objeto `String`, si necesitamos recuperar un valor entero o decimal de la entrada de datos,

tendremos que formatear la cadena devuelta por *readLine()* utilizando los métodos estáticos que ofrecen ciertas clases envoltorio. Los nombres de estos métodos empiezan por *parseXxx()* donde Xxx equivale al nombre de la clase envoltorio, tenemos como ejemplos: *Integer.parseInt(cadena)*, *Double.parseDouble(cadena)*, *Long.parseLong(cadena)*, etcétera.



```
1 package entrada_salida;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6
7 /**
8  * Lee un entero de la entrada de datos y muestra su factorial
9  * Creado por Hugo Pelayo
10  * 22 de abril de 2023
11  */
12 public class LecturaDatosBufferedReader {
13     // Creamos el objeto InputStreamReader y pasamos un InputStream
14     // en este caso el que está asociado al teclado y se encuentra en la clase System
15     1 usage
16     private static final InputStreamReader input = new InputStreamReader(System.in);
17     1 usage
18     private static final BufferedReader lector = new BufferedReader(input);
19
20     public static void main(String[] args) throws IOException {
21         long numero;
22         System.out.print("Introduce un número por favor: ");
23         numero = Long.parseLong(lector.readLine());
24         System.out.println("El factorial de " + numero + " es: " + factorial(numero));
25     }
26
27     2 usages
28     public static long factorial(long numero) {
29         return numero <= 0 ? 1 : (numero * factorial(numero - 1));
30     }
31 }
```

Figura 27. Ejemplo lectura de datos. Fuente: elaboración propia

Mediante Scanner

La clase Scanner funciona de manera similar a la clase BufferedReader. La clase Scanner ofrece métodos como *nextLine()* para leer una cadena de caracteres, *nextInt()* o *nextDouble()* para leer un valores enteros o decimales respectivamente. Cabe destacar que estos dos últimos lanzan una excepción si la cadena leída de la entrada de datos no se válida, es decir, no se puede formatear a un valor decimal correctamente.

```
LecturaDatosScanner.java
1 package entrada_salida;
2
3 import java.util.Scanner;
4
5 /**
6  * Lee un entero de la entrada de datos y muestra su factorial
7  * Creado por Hugo Pelayo
8  * 22 de abril de 2023
9  */
10 public class LecturaDatosScanner {
11     // Creamos el objeto Scanner y le pasamos como parámetro un InputStream
12     // en este caso el que está asociado al teclado y se encuentra en la clase System
13     2 usages
14     private static final Scanner lector = new Scanner(System.in);
15
16     public static void main(String[] args) {
17         long numero;
18         System.out.print("Introduce un número por favor: ");
19         numero = lector.nextInt();
20         System.out.println("El factorial de " + numero + " es: " + factorial(numero));
21
22         // cerramos el flujo de datos
23         lector.close();
24     }
25
26     2 usages
27     public static long factorial(long numero) {
28         return numero <= 0 ? 1 : (numero * factorial(numero - 1));
29     }
30 }
```

Figura 28. Lectura de datos con Scanner. Fuente: elaboración propia

Cierto problema que presenta la lectura con un objeto Scanner es que al cuando lee una cadena de caracteres de la entrada de datos, cuando se encuentra un salto de línea, este es abandonado en el buffer interno que el mismo objeto monitoriza, de tal manera que próximas llamadas a métodos como *nextInt()* o *nextLine()* devolverían una cadena vacía. Una forma de solucionar este problema es añadir una llamada intermedia a *nextLine()* que no nos sea necesaria. De todos modos, existe variedad de soluciones para este inconveniente.

```
LecturaDatosScanner.java  ProblemaScanner.java
1 package entrada_salida;
2
3 import java.util.Scanner;
4
5 /**
6  * Muestra por pantalla el nombre y edad del usuario
7  * Creado por Hugo Pelayo
8  * 22 de abril de 2023
9  */
10 public class ProblemaScanner {
11     4 usages
12     private static final Scanner scanner = new Scanner(System.in);
13     public static void main(String... args) {
14         System.out.print("¿Cómo te llamas? ");
15         String nombre = scanner.nextLine();
16
17         System.out.print("¿Y cual es tu edad? ");
18         int edad = scanner.nextInt();
19
20         // consumir salto de línea extra
21         scanner.nextLine();
22
23         System.out.printf("Hola %s. Tienes %d años\n", nombre, edad);
24         scanner.close();
25     }
26 }
```

Figura 29. Problema lectura con Scanner. Fuente: elaboración propia

Estructuras de control

En la programación con Java se emplea principalmente tres tipos de estructuras para el control del flujo de ejecución de un programa. Con la introducción de estructuras de datos y ciertas características (no únicamente en Java) estas muestran formas sintácticas ligeramente variadas pero la idea principal es la misma. Podemos agruparlas en tres campos diferentes: estructuras secuenciales, estructuras selectivas y estructuras iterativas:

Estructuras secuenciales

Están compuestas por un grupo de instrucciones o ninguna que se ejecutan en el orden en que aparecen una detrás de otra. Es la estructura más básica de ejecución de un programa y de la cual se nutren el resto. Como ejemplo tenemos básicamente cualquiera de los programas que se han mostrado en las figuras anteriores. Las estructuras secuenciales en esencia representan un paquete de instrucciones que se ejecutan como una unidad.

Las estructuras secuenciales vienen agrupadas en los llamados bloques. Estos bloques normalmente viene delimitados por llaves “{ }” para poder separarlos de otros, aunque también hay bloques sin llaves, esto pasa cuando este está compuesto por una única instrucción.

Estructuras selectivas

En estas estructuras evaluamos una expresión cuyo resultado debe ser del tipo booleano o convertible a dicho tipo de dato. Las estructuras selectivas se dividen en estructuras de selección simples, estructuras de selección compuestas, estructuras con el operador condicional y estructuras con el operador switch.

Las estructuras selectivas también se suelen denominar estructuras de decisión, básicamente nos permite bifurcar el flujo de ejecución de un programa mediante la evaluación de una condición cuyo resultado debe ser un valor booleano, es decir, puede ser cierto o falso. En otras palabras, estas estructuras nos permiten ejecutar un grupo de instrucciones u otro en base a si una condición se da o no.



```
1 package estructuras_control.selectivas;
2
3 import java.util.Scanner;
4
5 public class EjemploSelectiva1 {
6     2 usages
7     private static final Scanner lector = new Scanner(System.in);
8
9     public static void main(String[] args) {
10         final int MAYOR_DE_EDAD = 18;
11
12         System.out.print("Mi edad: ");
13         int miEdad = lector.nextInt();
14         if (miEdad < MAYOR_DE_EDAD)
15             System.out.println("Todavía soy mayor de edad :(");
16
17         lector.close();
18     }
19 }
```

Figura 30. Ejemplo sentencia selectiva simple. Fuente: elaboración propia

La estructura selectiva if ofrece una variante con un bloque que de instrucciones que se puede ejecutar en caso de que la condición sea falsa, o simplemente evaluar otras condiciones en caso de que la primera resulte ser falsa. Para indicar que queremos un bloque alternativo utilizamos la cláusula else. Esta cláusula no es obligatoria, pero es recomendable si se sabe de antemano que es necesario realizar ciertas acciones en caso de no ser cierta la condición inicial.



```
1 package estructuras_control.selectivas;
2
3 import java.util.Scanner;
4
5 public class EjemploSelectiva2 {
6     2 usages
7     private static final Scanner lector = new Scanner(System.in);
8
9     public static void main(String[] args) {
10         final int MAYOR_DE_EDAD = 18;
11
12         System.out.print("Mi edad: ");
13         int miEdad = lector.nextInt();
14         if (miEdad < MAYOR_DE_EDAD)
15             System.out.println("Todavía soy mayor de edad :(");
16         else
17             System.out.println("¡Ya soy mayor de edad! Por fin :)");
18
19         lector.close();
20     }
21 }
```

Figura 31. Estructura selectiva con bloque alternativo. Fuente: elaboración propia

Figura 32. Estructura selectiva con evaluación múltiple. Fuente: elaboración propia

En ciertas ocasiones, queremos ejecutar un bloque de instrucciones, pero tenemos un seguido de instrucciones que se basa en evaluar si una variable es igual a cierto valor entero, esta operación acaba siendo innecesaria ya que realizamos la misma evaluación varias veces sobre la misma variable, para solucionar este inconveniente, tenemos las sentencias switch. Esta sentencia tiene dos variantes, una que se puede considerar normal y que hereda de otros lenguajes de programación y otra mejora también denominada Enhanced Switch Statement en inglés, que es una versión con mejora sintáctica que se hizo pública a partir de la versión 14 de Java, aunque en versiones previas, más concretamente en la 12 y 13, esta característica ya era se estaba poniendo a prueba.

```
EjemploSelectiva3.java SwitchStatement.java
1 package estructuras_control.selectivas;
2
3 import java.util.Scanner;
4
5 public class SwitchStatement {
6     2 usages
7     private static final Scanner lector = new Scanner(System.in);
8     public static void main(String[] args) {
9         String valorEnString;
10        System.out.print("Introduce un número entre el 1 y el 4: ");
11        int userInput = lector.nextInt();
12
13        switch (userInput) {
14            case 1:
15                valorEnString = "Uno";
16                break;
17            case 2:
18                valorEnString = "Dos";
19                break;
20            case 3:
21                valorEnString = "Tres";
22                break;
23            case 4:
24                valorEnString = "Cuatro";
25                break;
26            default:
27                valorEnString = "Desconocido";
28                break;
29        }
30
31        System.out.println("Has introducido el valor: " + valorEnString);
32        lector.close();
33    }
34 }
```

Figura 33. Ejemplo sentencia switch. Fuente: elaboración propia


```
EjemploSelectiva3.java SwitchStatement.java EnhancedSwitch.java
1 package estructuras_control.selectivas;
2
3 import java.util.Scanner;
4
5 public class EnhancedSwitch {
6     private static final Scanner lector = new Scanner(System.in);
7     public static void main(String[] args) {
8         String valorEnString;
9         System.out.print("Introduce un número entre el 1 y el 4: ");
10        int userInput = lector.nextInt();
11
12        valorEnString = switch (userInput) {
13            case 1 → "Uno";
14            case 2 → "Dos";
15            case 3 → "Tres";
16            case 4 → "Cuatro";
17            default → "Desconocido";
18        };
19
20        System.out.println("Has introducido el valor: " + valorEnString);
21        lector.close();
22    }
23 }
24 }
```

Figura 34. Sentencia switch mejorada. Fuente: elaboración propia

Para acabar tenemos el operador ternario, es una forma simplificada de redactar una sentencia selectiva compuesta (un if con bloque else) la sintaxis es la siguiente:

```
expresion ? valor1 : valor2
```


Donde expresion es la expresión a ser evaluada, si esta es cierta el resultado global del operador ternario sería el valor1 y valor2 en caso contrario.



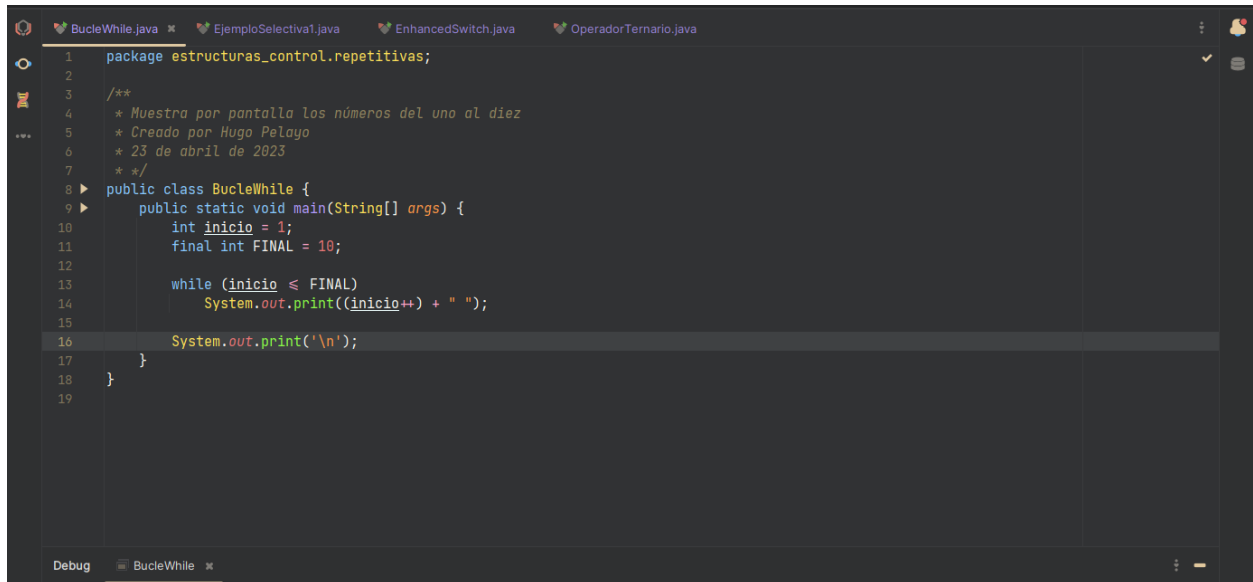
```
1 package estructuras_control.selectivas;
2
3 import java.util.Scanner;
4
5 /**
6  * Muestra por pantalla si la edad que el usuario ha introducido por
7  * pantalla corresponde a la mayoría de edad o no
8  * Creado por Hugo Pelayo
9  * 23 de abril de 2023
10  */
11 public class OperadorTernario {
12     private static final Scanner lector = new Scanner(System.in);
13
14     public static void main(String[] args) {
15         final int MAYOR_DE_EDAD = 18;
16         String siMayorEdad = "¡Ya soy mayor de edad! Por fin :)";
17         String noMayorEdad = "Todavía soy mayor de edad :(";
18
19         System.out.print("Mi edad: ");
20         int miEdad = lector.nextInt();
21
22         System.out.println(miEdad < MAYOR_DE_EDAD ? noMayorEdad : siMayorEdad);
23         lector.close();
24     }
25 }
```

Figura 35. Ejemplo operador ternario. Fuente: elaboración propia

Estructuras iterativas

También llamadas bucles, son una combinación de las anteriores, estas estructuras nos permiten ejecutar un grupo de instrucciones de manera reiterada mientras se cumpla una condición.

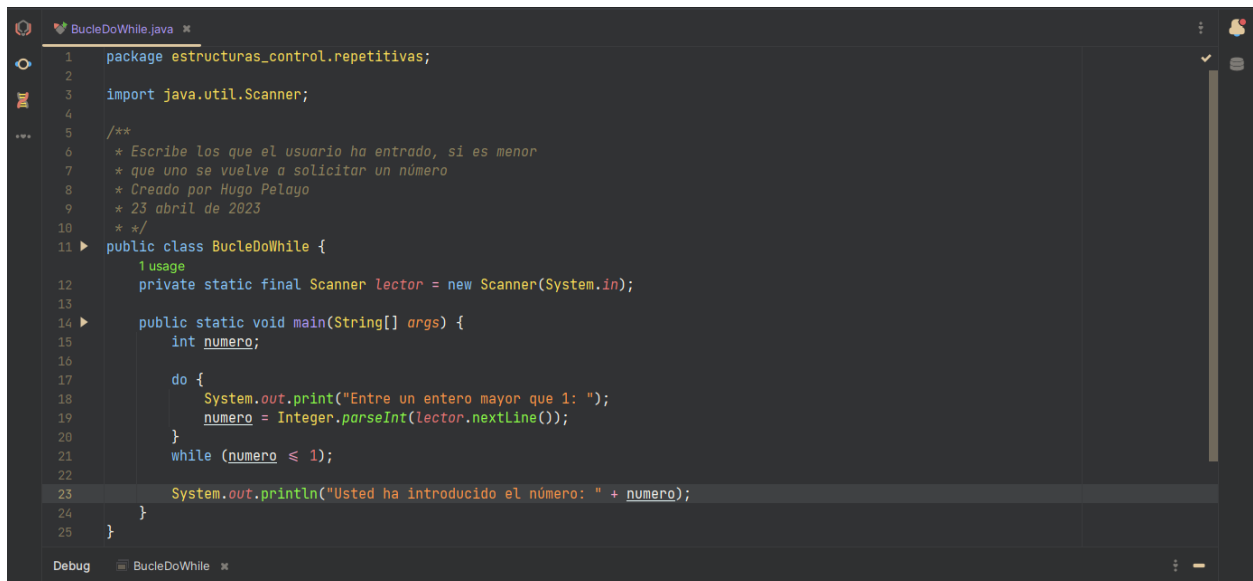
La primera estructura iterativa es el bucle while. Esta estructura nos permite repetir una serie de instrucciones mientras se cumpla una condición. Esta condición es evaluada incluso la primera vez que se intenta iterar sobre el cuerpo del bucle, de modo que, si la condición no se cumple el cuerpo no llega a ejecutarse nunca. Es imprescindible que el bucle while controlemos la condición de fin, de lo contrario se convertiría en un bucle infinito.



```
1 package estructuras_control.repetitivas;
2
3 /**
4  * Muestra por pantalla los números del uno al diez
5  * Creado por Hugo Pelayo
6  * 23 de abril de 2023
7  */
8 public class BucleWhile {
9     public static void main(String[] args) {
10         int inicio = 1;
11         final int FINAL = 10;
12
13         while (inicio ≤ FINAL)
14             System.out.print((inicio++) + " ");
15
16         System.out.print('\n');
17     }
18 }
19
```

Figura 36. Bucle while. Fuente: elaboración propia

La segunda estructura repetitiva es el bucle do-while. Este bucle es similar al anterior, sin embargo, evalúa la condición después de ejecutar el cuerpo del bucle, de modo, se garantiza que nuestras instrucciones se ejecutarán al menos una vez.



```
1 package estructuras_control.repetitivas;
2
3 import java.util.Scanner;
4
5 /**
6  * Escribe los que el usuario ha entrado, si es menor
7  * que uno se vuelve a solicitar un número
8  * Creado por Hugo Pelayo
9  * 23 abril de 2023
10 */
11 public class BucleDoWhile {
12     private static final Scanner lector = new Scanner(System.in);
13
14     public static void main(String[] args) {
15         int numero;
16
17         do {
18             System.out.print("Entre un entero mayor que 1: ");
19             numero = Integer.parseInt(lector.nextLine());
20         } while (numero ≤ 1);
21
22         System.out.println("Usted ha introducido el número: " + numero);
23     }
24 }
25
```

Figura 37. Bucle do-while. Fuente: elaboración propia

Por último, tenemos el bucle for, este nos permite ejecutar un conjunto de instrucciones un número conocido y fijo de veces. Es ideal cuando sabemos exactamente cuántas veces queremos iterar sobre un bloque de sentencias.



Figura 38. Bucle for. Fuente: elaboración propia

Estructuras de salto

Las estructuras de salto son estructuras que nos sirven para romper el flujo normal de nuestro código o simplemente retorna de funciones o procedimientos, éstas compuestas por las sentencias `break`, `continue`, `return`, Java no ofrece soporte para la sentencia `goto`, sin embargo, permite implementar su funcionalidad a través de la sentencia `break`. El uso de esta última (en estructuras iterativas), acostumbra a ser desaconsejada por muchos programadores por la dificultad que genera a la hora de seguir el flujo de ejecución y a lógica de nuestro programa.

La sentencia `break` nos permite terminar inmediatamente la ejecución de un bucle, es decir, cualquier instrucción que haya después de la sentencia `break` no se ejecuta, a parte, también nos permite saltar a cualquier etiqueta que tengamos en nuestro código siempre que no sea fuera del método en que se utiliza. La sentencia `continue` nos permite omitir la iteración actual, funciona de manera similar a la sentencia `break`, pero nos permite volver a evaluar la condición del bucle y si esta se cumple entonces se vuelve a iterar. Cuando estas sentencias se encuentran en bucles anidados, afectan al bucle más interno.




```

1 package estructuras_control.de_salto;
2
3 import java.util.Scanner;
4
5 public class SentenciaBreak {
6     1 usage
7     private static final Scanner lector = new Scanner(System.in);
8
9     public static void main(String[] args) {
10         int[] elementos = { 1, 0, 4, 5, 6, 7, 8, 9, 10, 22, 11 };
11         System.out.print("Introduce un entero: ");
12         int target = lector.nextInt();
13
14         int indice = 0;
15         boolean encontrado = false;
16
17         while (indice < elementos.length) {
18             if (elementos[indice] == target) {
19                 encontrado = true;
20                 break;
21             }
22             else
23                 ++indice;
24         }
25
26         if (encontrado)
27             System.out.println("El elemento se ha encontrado en la posición: " + (indice - 1));
28         else
29             System.out.println("El elemento no existe...");
30     }
31 }
32

```

Figura 39. Sentencia break. Fuente: elaboración propia



```

1 package estructuras_control.de_salto;
2
3 public class SentenciaContinue {
4     public static void main(String[] args) {
5         int[] elementos = { 2, 4, 6, 1, 22, 34, 11, 9, 7, 15, 4 };
6
7         for (int elemento : elementos) {
8             // mostramos solo los elementos pares
9             if (elemento % 2 == 0) {
10                 continue;
11             }
12
13             System.out.print(elemento + " ");
14         }
15
16         System.out.print('\n');
17     }
18 }
19

```

Figura 40. Sentencia continue. Fuente: elaboración propia

Por último, tenemos la sentencia return. La primera nos permite retornar de un procedimiento y la segunda nos permite saltar a una etiqueta que indicamos en nuestro código, esta funcionalidad también es admitida por la sentencia break. La sentencia return también sirve para devolver valores de funciones:



```
1 package estructuras_control.de_salto;
2
3 public class SentenciaReturn {
4     public static void main(String[] args) {
5         int[] elementos = { 2, 4, 6, 1, 22, 34, 11, 9, 7, 15, 4 };
6
7         for (int elemento : elementos) {
8             // mostramos solo los elementos pares
9             if (!esPar(elemento)) {
10                 continue;
11             }
12
13             System.out.print(elemento + " ");
14         }
15
16         System.out.print('\n');
17     }
18
19     1 usage
20     public static boolean esPar(int numero) {
21         if (numero % 2 == 0)
22             return true;
23         else
24             return false;
25     }
26 }
```

Figura 41. Ejemplo sentencia return. Fuente: elaboración propia

Arrays

Un array o arreglo en castellano es un objeto en el cual podemos almacenar una serie de datos del mismo tipo. Es un tipo de contenedor de elementos secuencial ya que almacena los datos de manera secuencial. Cada elemento ocupa una sola posición o índice que indica donde se sitúa un dentro del array, siendo 0 el primer índice y la máxima posición siendo el total de elementos menos uno. Para declarar un array utilizamos la siguiente sintaxis:

```
tipo_dato[] identificador_array;
tipo_dato identificador_array[];
```

Los arrays se declaran en los mismos bloques que las variables de tipos de datos básicos, es decir, los podemos declarar como atributos de una clase o como variables locales de un método. Al declarar un array, las posiciones de este se inicializan al valor por defecto del tipo de dato, es decir, en el caso de tipos básicos como enteros o decimales, se inicializan con el valor 0, si se trata de un array de valores booleanos, todas las posiciones se inicializan a falso, y en el caso de referencias, estas se inicializan todas con el valor nulo.

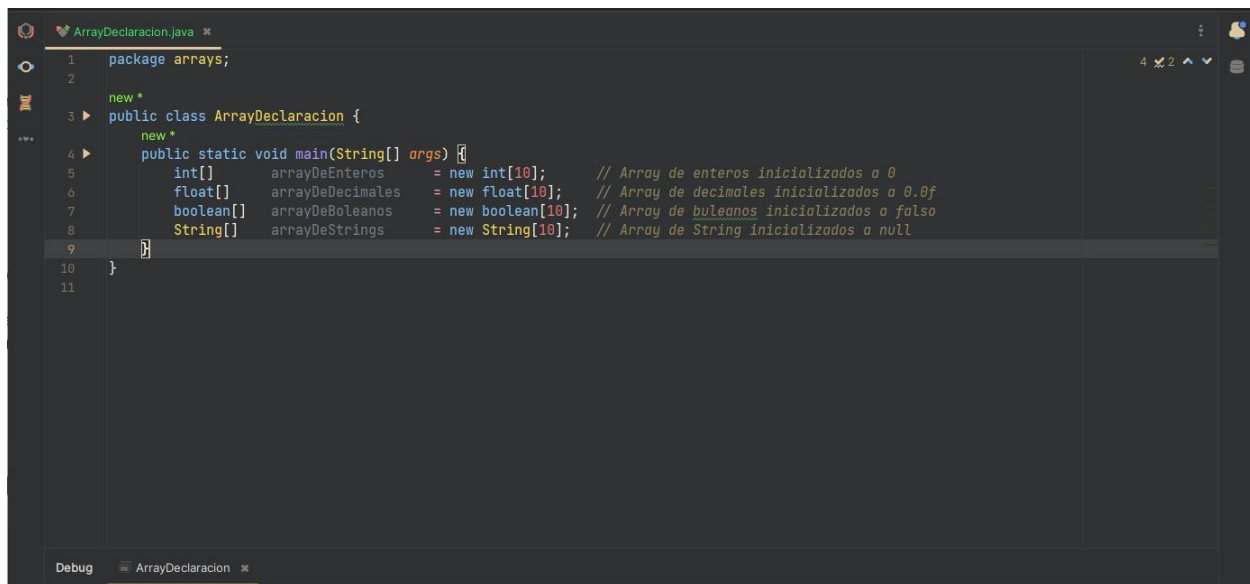


Figura 42. Ejemplo declaración arrays. Fuente: elaboración propia

Para asignar valores a las posiciones de un array utilizamos la sintaxis:

```
identificador_array[indice] = valor;
```

También es posible darle valor inicial a nuestro array en el momento de la creación con la siguiente sintaxis.

```
tipo_dato[] identificador_array = { valor1, valor2, valor3 .. valorN };
```

```
tipo_dato identificador_array[] = { valor1, valor2, valor3 .. valorN };
```

```
tipo_dato[] identificador_array = new tipo_dato[]{ valor1, valor2, valor3 .. valorN };
```

```
tipo_dato identificador_array[] = new tipo_dato[]{ valor1, valor2, valor3 .. valorN };
```

Cabe destacar que es incorrecto declarar el array con valores iniciales y con tamaño inicial a la vez indiferentemente del tamaño inicial que le demos y los valores que asignemos, es decir, las siguientes sintaxis son erróneas:

```
tipo_dato[] identificador_array = new tipo_dato[tamano]{ valor1, valor2,
valor3 .. valorN };

tipo_dato identificador_array[] = new tipo_dato[tamano]{ valor1, valor2,
valor3 .. valorN };
```

Los arrays son objetos de tipo referencia (no son tipos básicos). Como todos los objetos, podemos pasar los arrays como argumentos a funciones. Al pasar un array a un método, este crea una copia local que es una referencia que apunta a la dirección de memoria donde se localiza el array original, de tal modo que podemos modificar nuestro array desde cualquier método que lo reciba como parámetro.



Figura 43. Pasando arrays como argumentos. Fuente: elaboración propia

Para recorrer los elementos de un array podemos utilizar los bucles vistos con anterioridad, de tal modo que podemos acceder a todos sus elementos o a unos específicos. A parte, tenemos el bucle for-each que está disponible desde la versión 5 de Java y nos permite recorrer todos los elementos de un objeto que esté marcado como iterable, tal es el caso del array y otros contenedores de la librería de Java. La sintaxis para el bucle for-each es la siguiente:

```
for (tipo_dato identificador_variable : contenedor)
    sentencias...
```

Contenedor es el identificador del contenedor o iterable que queremos recorrer:

```

1 package arrays;
2
3 new *
4 public class ArrayDeclaracion {
5     new *
6     public static void main(String[] args) {
7         int indice;
8         String[] palabra = { "Agitar", "Comer", "Romper", "Destruir" };
9
10        indice = 0; // empezamos a recorrer desde la posición inicial
11        while (indice < palabra.length) {
12            System.out.printf("Palabra en la posición %d: %s\n", indice + 1, palabra[indice]);
13            ++indice; // avanzamos a la siguiente posición
14        }
15
16        for (indice = 0; indice < palabra.length; ++indice)
17            System.out.printf("Palabra en la posición %d: %s\n", indice + 1, palabra[indice]);
18
19        indice = 0; // inicializamos para mostrar la posición
20        for (String item : palabra)
21            System.out.printf("Palabra en la posición %d: %s\n", ++indice, item);
22    }
23 }

```

Figura 44. Ejemplo recorridos array. Fuente: elaboración propia

Los arrays también pueden ser multidimensionales, de tal manera que tenemos estructuras que se asemejan a las matrices. La sintaxis de declaración es similar a la de los arrays unidimensionales:

```

tipo_dato[] identificador_array = new tipo_dato[tamano1][tamano2];
tipo_dato identificador_array[] = new tipo_dato[tamano1][tamano2];

```

La estructura en memoria se asemeja al siguiente esquema:

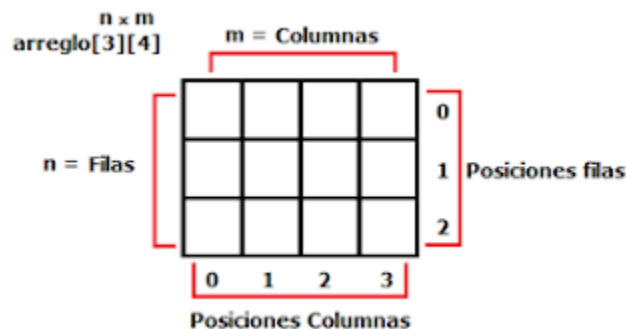
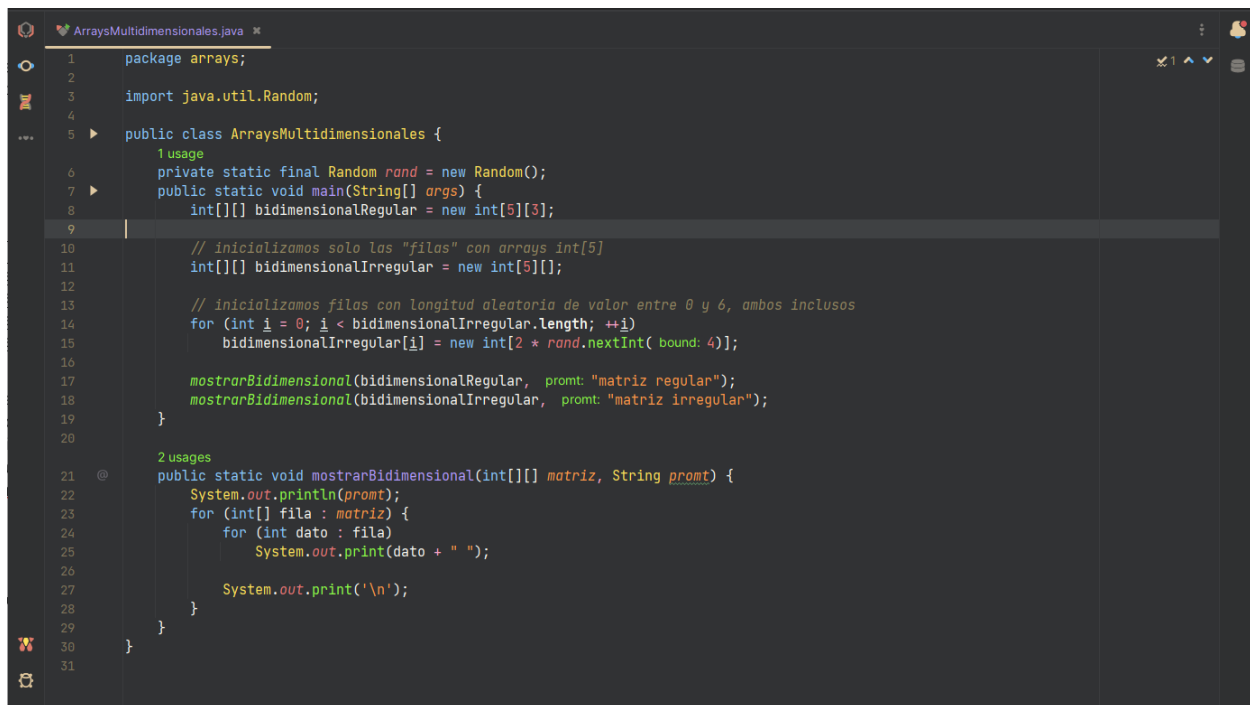


Figura 45. Estructura array bidimensional. Fuente: codeJaVu.blogspot.com

La idea es tener un array de arrays, de tal modo que, con el primer índice, accedemos un array en concreto y con el segundo accedemos a un elemento específico. También es posible realizar el mismo recorrido con el bucle `for-each`. Cabe destacar que no necesariamente debe construirse un array bidimensional con la estructura de una matriz. La estructura puede ser irregular, para esto tendríamos que manualmente indicar el tamaño de los subarrays.

A screenshot of an IDE window titled 'ArraysMultidimensionales.java'. The code is in Java and defines a class 'ArraysMultidimensionales'. It includes a package declaration 'package arrays;', an import 'import java.util.Random;', and a public class definition. Inside the class, there is a private static final 'Random' object 'rand' and a 'main' method. The 'main' method initializes two 2D arrays: 'bidimensionalRegular' (5x3) and 'bidimensionalIrregular' (5x5). It then calls 'mostrarBidimensional' for both. The 'mostrarBidimensional' method is a static method that takes a 2D array and a prompt string, and prints the array's contents. The code is well-commented, explaining the initialization of the irregular array with random values.

```
1 package arrays;
2
3 import java.util.Random;
4
5 public class ArraysMultidimensionales {
6     1 usage
7     private static final Random rand = new Random();
8     public static void main(String[] args) {
9         int[][] bidimensionalRegular = new int[5][3];
10
11         // inicializamos solo las "filas" con arrays int[5]
12         int[][] bidimensionalIrregular = new int[5][];
13
14         // inicializamos filas con longitud aleatoria de valor entre 0 y 6, ambos inclusos
15         for (int i = 0; i < bidimensionalIrregular.length; ++i)
16             bidimensionalIrregular[i] = new int[2 * rand.nextInt( bound: 4)];
17
18         mostrarBidimensional(bidimensionalRegular, prompt: "matriz regular");
19         mostrarBidimensional(bidimensionalIrregular, prompt: "matriz irregular");
20     }
21
22     2 usages
23     public static void mostrarBidimensional(int[][] matriz, String prompt) {
24         System.out.println(prompt);
25         for (int[] fila : matriz) {
26             for (int dato : fila)
27                 System.out.print(dato + " ");
28             System.out.print('\n');
29         }
30     }
31 }
```

Figura 46. Inicialización de arrays multidimensionales. Fuente: elaboración propia

Colecciones

Las colecciones en programación son estructuras de datos dinámicas que permiten almacenar objetos de forma flexible, es decir, se pueden aumentar o disminuir en tamaño durante la ejecución del programa. A diferencia de los arrays, que tienen un tamaño fijo, las colecciones pueden ser ordenadas, permiten la inserción y eliminación de elementos y son más eficientes en el manejo de grandes cantidades de datos. Sin embargo, a diferencia de los arrays, las colecciones no pueden almacenar tipos de datos primitivos.

En Java, se utiliza la interfaz genérica *Collection* para trabajar con colecciones. Esta interfaz permite almacenar cualquier tipo de objeto y utilizar una serie de métodos comunes como añadir, eliminar y obtener el tamaño de la colección. Además, existen varias subinterfaces que extienden la interfaz *Collection* y que aportan distintas funcionalidades a la hora de trabajar con las colecciones.

Existen diferentes tipos de colecciones en Java, cada una con su propia estructura y funcionalidad. Por ejemplo, las listas son adecuadas para enumeraciones de registros de datos que pueden ser eliminados o aumentados en cualquier momento, mientras que las pilas y colas son útiles para almacenar datos de forma LIFO o FIFO, respectivamente. Las tablas hash se utilizan para asociar claves con valores, mientras que los árboles se emplean para mantener datos permanentemente ordenados. En resumen, las colecciones son herramientas fundamentales para gestionar grandes cantidades de datos en Java. A continuación, se muestra la jerarquía de clases de la interfaz *Collection*:

Collection Interface

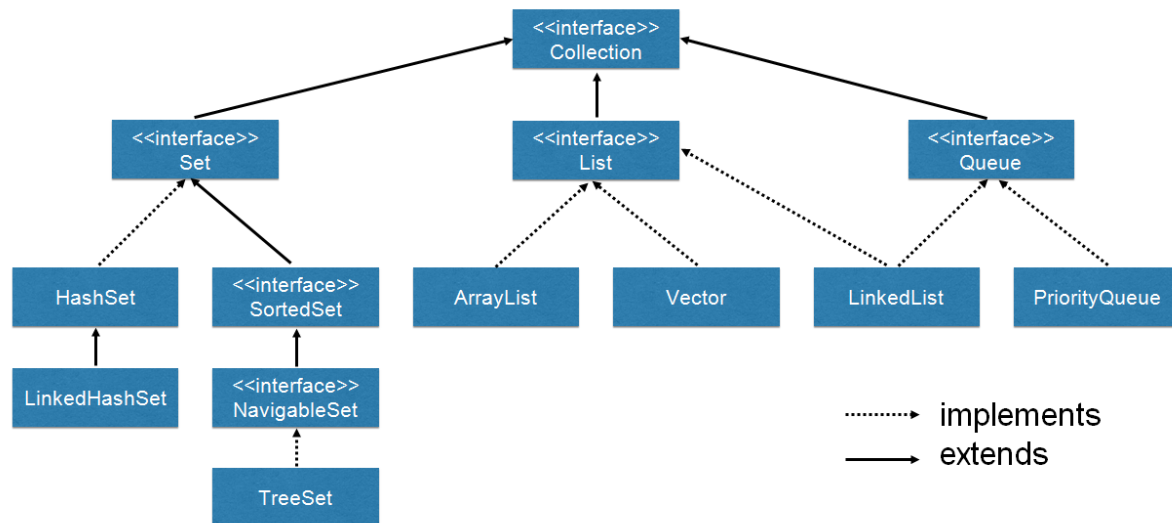


Figura 47. Interfaz Collection. Fuente: Marcus Biel

La interfaz List en Java es una subinterfaz de la interfaz Collection y se utiliza para almacenar una secuencia ordenada de elementos, es decir, una lista. Las implementaciones de la interfaz List en Java proporcionan una serie de métodos para agregar, eliminar, buscar y obtener elementos de una lista. Además, los elementos en una lista pueden ser indexados y se pueden acceder mediante su posición en la lista.

Existen varias clases que implementan la interfaz List en Java, cada una con diferentes características y usos. Algunas de las clases más comunes son:

ArrayList: Esta clase almacena los elementos en una matriz de tamaño dinámico y proporciona un acceso rápido a los elementos mediante su índice. Sin embargo, la inserción y eliminación de elementos en posiciones intermedias de la lista puede ser costosa debido a la necesidad de reorganizar los elementos de la matriz.

LinkedList: Esta clase utiliza una estructura de datos enlazada para almacenar los elementos y proporciona una inserción y eliminación más eficiente que ArrayList en posiciones intermedias de la lista. Sin embargo, el acceso a los elementos mediante su índice es más lento debido a que la lista debe recorrerse desde el principio para llegar al elemento deseado.

Vector: Esta clase es similar a ArrayList, pero proporciona operaciones seguras para subprocesos, lo que significa que puede ser utilizada por varios hilos de ejecución simultáneamente sin riesgo de corrupción de datos.


La interfaz Queue define una colección que mantiene el orden de inserción de sus elementos y permite la inserción y eliminación de elementos en ambas extremidades de la colección. La interfaz Queue extiende la interfaz Collection y define varios métodos específicos para la gestión de colas, como *offer()* para añadir un elemento al final de la cola, *poll()* para eliminar y devolver

el elemento en el frente de la cola, y *peek()* para devolver el elemento en el frente de la cola sin eliminarlo.

Existen varias clases en Java que implementan la interfaz *Queue*, como *PriorityQueue* y *ArrayDeque*. *LinkedList* es una implementación de lista doblemente enlazada que también puede funcionar como cola. *PriorityQueue* es una cola de prioridad en la que los elementos se ordenan según un comparador o según su orden natural. *ArrayDeque* es una implementación de doble cola que también puede funcionar como pila.

La interfaz *Set* define una colección que no permite elementos duplicados y no mantiene un orden específico de los elementos. La interfaz *Set* extiende la interfaz *Collection* y define métodos para añadir elementos a la colección *add()*, eliminar elementos *remove()*, comprobar si un elemento está presente *contains()* y obtener el tamaño de la colección *size()*.

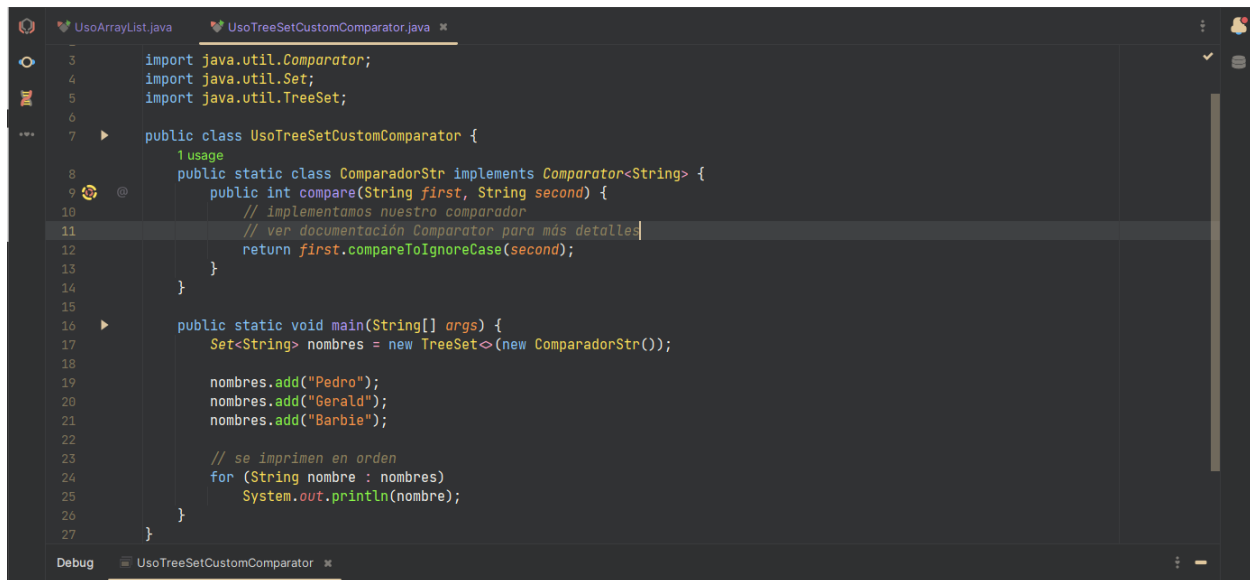
Existen varias clases en Java que implementan la interfaz *Set*, como *HashSet*, *LinkedHashSet* y *TreeSet*. *HashSet* es una implementación de tabla hash que no mantiene un orden específico de los elementos. *LinkedHashSet* es una implementación de tabla hash que mantiene el orden de inserción de los elementos. *TreeSet* es una implementación de árbol balanceado que mantiene los elementos ordenados según un comparador o según su orden natural.

A screenshot of an IDE window showing three Java files: 'UsoArrayList.java', 'UsoTreeSet.java', and 'UsoTreeSetCustomComparator.java'. The 'UsoTreeSet.java' file is active and contains the following code:

```
1 package collections;
2
3 import java.util.Set;
4 import java.util.TreeSet;
5
6 public class UsoTreeSet {
7     public static void main(String[] args) {
8         Set<String> nombres = new TreeSet<>();
9
10        nombres.add("Pedro");
11        nombres.add("Gerald");
12        nombres.add("Barbie");
13
14        // se imprimen en orden
15        for (String nombre : nombres)
16            System.out.println(nombre);
17    }
18 }
19
```

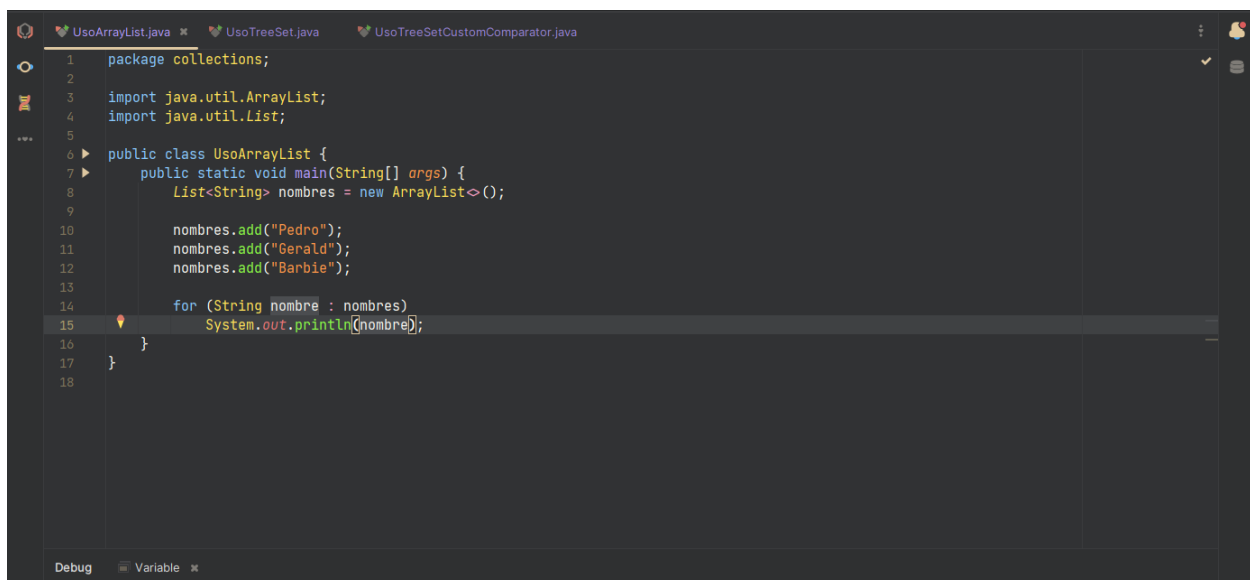
The IDE interface includes a sidebar on the left with icons for Explorer, Search, and Run and Debug. The bottom status bar shows 'Debug' and the active file 'UsoTreeSet.java'.

Figura 48. Utilización Set. Fuente: elaboración propia



```
1 import java.util.Comparator;
2 import java.util.Set;
3 import java.util.TreeSet;
4
5 public class UsoTreeSetCustomComparator {
6     // usage
7     public static class ComparadorStr implements Comparator<String> {
8         public int compare(String first, String second) {
9             // implementamos nuestro comparador
10            // ver documentación Comparator para más detalle
11            return first.compareToIgnoreCase(second);
12        }
13    }
14
15    public static void main(String[] args) {
16        Set<String> nombres = new TreeSet<>(new ComparadorStr());
17
18        nombres.add("Pedro");
19        nombres.add("Gerald");
20        nombres.add("Barbie");
21
22        // se imprimen en orden
23        for (String nombre : nombres)
24            System.out.println(nombre);
25    }
26 }
```

Figura 49. Uso Set con un comparador custom. Fuente: elaboración propia



```
1 package collections;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class UsoArrayList {
7     public static void main(String[] args) {
8         List<String> nombres = new ArrayList<>();
9
10        nombres.add("Pedro");
11        nombres.add("Gerald");
12        nombres.add("Barbie");
13
14        for (String nombre : nombres)
15            System.out.println(nombre);
16    }
17 }
```

Figura 50. Ejemplo uso List. Fuente: elaboración propia

Otras clases

En Java existen variedad de clases para todo tipo de funcionalidad que se nos ocurra, es mejor por lo general, consultar primero si es posible que haya alguna clase que ofrezca una funcionalidad que deseamos utilizar antes de implementarla. A continuación, se presentan varias de las clases más utilizadas en Java:

La clase String en Java es una clase fundamental que se utiliza para representar una cadena de caracteres. Las cadenas de caracteres son secuencias de caracteres Unicode que se utilizan comúnmente en la programación para representar texto. La clase String es inmutable, lo que

significa que una vez que se crea una cadena, no se puede modificar. Sin embargo, se pueden crear nuevas cadenas a partir de la cadena original. La clase `String` también proporciona una variedad de métodos para trabajar con cadenas, como la concatenación, la búsqueda, la extracción y la comparación.

Date: La clase `Date` se utiliza para representar una fecha y hora específicas en Java. Esta clase se basa en un tiempo en milisegundos transcurridos desde la medianoche del 1 de enero de 1970 (conocido como el "epoch"). La clase `Date` se ha quedado obsoleta y se recomienda utilizar la clase `LocalDate` o `LocalDateTime` en su lugar para trabajar con fechas en Java.

ChronoUnit: es una enumeración que define las unidades de tiempo utilizadas en el nuevo paquete `java.time` de Java 8. Estas unidades de tiempo incluyen años, meses, días, horas, minutos, segundos y nanosegundos. La clase `ChronoUnit` proporciona métodos para calcular la diferencia entre dos instantes de tiempo en una unidad de tiempo específica.

StringTokenizer: se utiliza para dividir una cadena en tokens (subcadenas) utilizando un delimitador específico. Esta clase es útil para analizar cadenas de texto que tienen un formato estructurado. La clase `StringTokenizer` proporciona una variedad de métodos para obtener los tokens y el número total de tokens en la cadena.

LocalDate: se utiliza para representar una fecha sin una zona horaria específica en Java. Esta clase proporciona métodos para trabajar con fechas, como obtener el año, el mes y el día, así como para realizar operaciones matemáticas con fechas, como sumar o restar días.

LocalDateTime: se utiliza para representar una fecha y hora sin una zona horaria específica en Java. Esta clase es similar a la clase `LocalDate`, pero también incluye información de hora. La clase `LocalDateTime` proporciona métodos para trabajar con fechas y horas, así como para realizar operaciones matemáticas con fechas y horas, como sumar o restar días y horas.

StringBuilder: se utiliza para construir cadenas de caracteres de forma dinámica en Java. A diferencia de la clase `String`, la clase `StringBuilder` es mutable, lo que significa que se puede modificar una vez creada. La clase `StringBuilder` proporciona una variedad de métodos para agregar, eliminar o modificar caracteres en la cadena.

DateTimeFormatter: se utiliza para formatear y analizar fechas y horas en Java. Esta clase proporciona métodos para crear patrones de formato de fecha y hora personalizados, así como para analizar fechas y horas a partir de cadenas de texto.

En el paquete `otras_clases` se proveen ejemplos de usos de estas clases.

Clases envoltorio

Las clases envoltorio o Wrapper Classes en Java son un conjunto de clases que permiten convertir los tipos primitivos en objetos y viceversa. Esto es útil en situaciones en las que necesitamos trabajar con tipos de datos primitivos, pero también necesitamos acceder a las funcionalidades proporcionadas por los objetos.

Las clases envoltorio son:

- Integer: representa un número entero de 32 bits.
- Long: representa un número entero de 64 bits.
- Float: representa un número de coma flotante de 32 bits.
- Double: representa un número de coma flotante de 64 bits.
- Short: representa un número entero de 16 bits.
- Byte: representa un número entero de 8 bits.
- Character: representa un carácter Unicode de 16 bits.
- Boolean: representa un valor booleano (true o false).

De este modo, si queremos almacenar un valor entero en una estructura de datos que solo admite objetos, podemos crear un objeto de la clase Integer que contenga ese valor entero. A su vez, si queremos obtener el valor entero de ese objeto, podemos llamar al método *intValue()* que nos devolverá el valor original.

Otra utilidad de las clases envoltorios es que nos permiten utilizar los tipos primitivos en contextos que requieren objetos, como por ejemplo en colecciones o en parámetros de métodos que solo aceptan objetos.

A partir de Java 5, se introdujo el concepto de “autoboxing” y “unboxing” para simplificar la manipulación de tipos primitivos y clases envoltorio. El “autoboxing” permite la conversión automática de tipos primitivos en sus clases envoltorio correspondientes y viceversa, lo que simplifica el código y hace que sea más legible.

Además, desde Java 5, se introdujeron nuevos métodos en las clases envoltorio para simplificar aún más la manipulación de tipos primitivos, como por ejemplo los métodos *valueOf()* para crear un objeto envoltorio a partir de un valor primitivo, y los métodos *intValue()*, *doubleValue()*, *floatValue()*, *longValue()*, etcétera, para obtener el valor primitivo de un objeto envoltorio.

También se introdujeron mejoras en la forma en que se comparan los objetos envoltorios, permitiendo la comparación de objetos envoltorios utilizando operadores de comparación como `==` y `!=`, sin tener que preocuparse por las diferencias entre los objetos y los valores primitivos.

Programación Orientada a Objetos

La programación orientada a objetos (POO) es un paradigma de programación que se basa en la idea de encapsular los datos y comportamientos dentro de objetos que interactúan entre sí para llevar a cabo la funcionalidad del programa.

En la POO, los objetos se modelan a partir de clases, que son estructuras que definen la estructura y comportamiento de los objetos. Una clase puede contener atributos, que son las variables que definen el estado de un objeto, y métodos, que son las funciones que definen su comportamiento.



```
1 package poo;
2
3 no usages new *
4 public abstract class Animal {
5     3 usages
6     private String m_Nombre;
7
8     no usages new *
9     public Animal(String nombre) {
10         m_Nombre = nombre == null ? "Desconocido" : nombre;
11     }
12
13     1 usage new *
14     public String getNombre() { return m_Nombre; }
15
16     no usages new *
17     public void setNombre(String nombre) { m_Nombre = nombre == null ? "Desconocido" : nombre; }
18
19     1 usage new *
20     public abstract void greet();
21
22     no usages new *
23     public void presentar() {
24         greet();
25         System.out.println(" Me llamo " + getNombre());
26     }
27 }
28 }
```

Figura 51. Ejemplo básico de clase. Fuente: elaboración propia

La herencia es un concepto clave en la programación orientada a objetos que permite crear nuevas clases a partir de otras existentes. Se basa en la idea de que una nueva clase (la clase derivada o subclase) puede heredar propiedades y comportamientos de una clase existente (la clase base o superclase).

La herencia se representa en un diagrama de clases mediante una línea que conecta la clase base con la subclase. En términos generales, una subclase es una versión más especializada de su superclase. Por ejemplo, una clase "Vehículo" podría ser la superclase de las subclases "Coche", "Motocicleta" y "Camión".

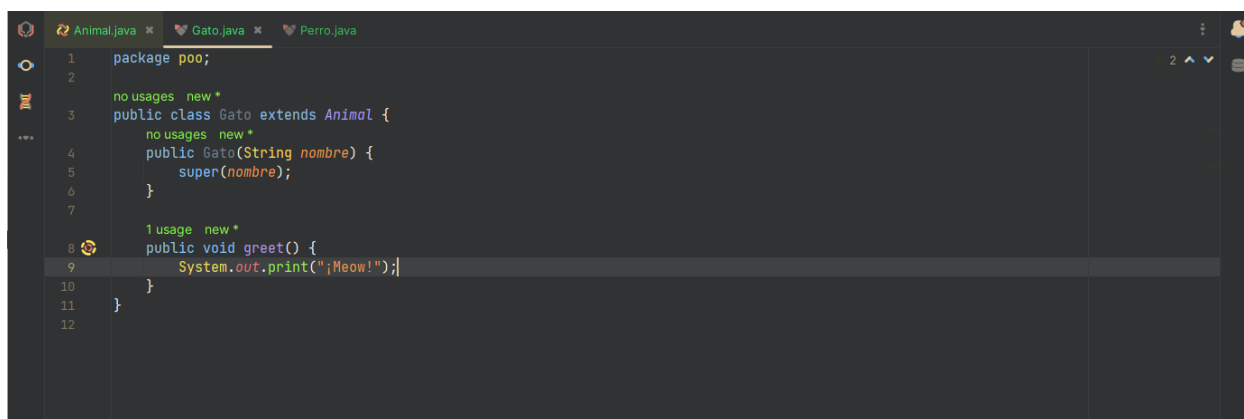
Al heredar de una clase, una subclase tiene acceso a todos los atributos (variables de instancia) y métodos públicos y protegidos de la superclase. Además, puede agregar nuevos atributos y métodos, o redefinir (anular) los existentes. Este proceso se llama extensión de la clase base y permite que la subclase tenga su propia identidad y funcionalidad.

La herencia también permite la reutilización de código, ya que las clases base pueden ser compartidas por varias subclases. Por ejemplo, si varias subclases necesitan ciertos métodos en común, estos pueden ser definidos en la superclase y heredados por todas las subclases.

Es importante tener en cuenta que la herencia no siempre es la mejor solución para la reutilización de código. A veces, es mejor utilizar la composición, que consiste en crear objetos de otras clases dentro de una clase y utilizar sus métodos y atributos en lugar de heredarlos.

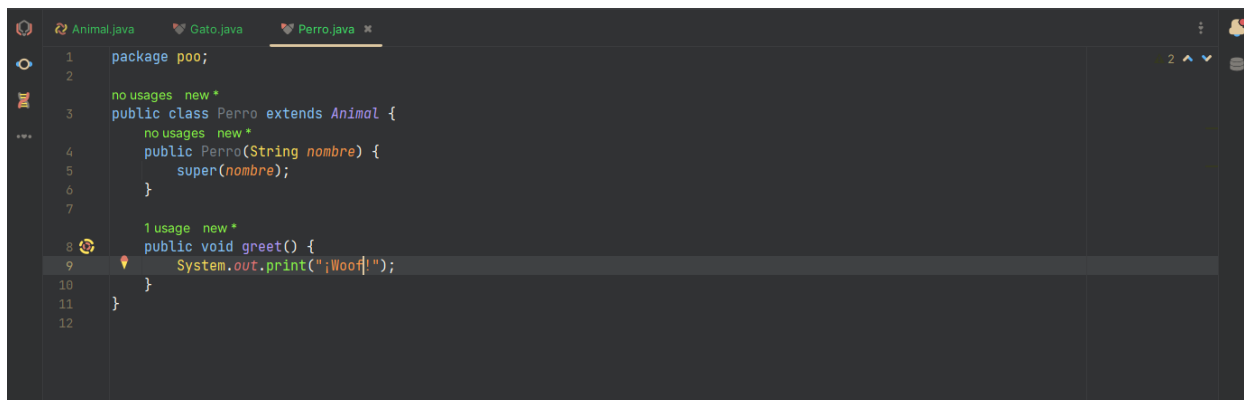
Por último, cabe destacar que Java soporta la herencia simple, es decir, una clase sólo puede heredar de una única superclase. Esto se debe a que la herencia múltiple puede dar lugar a problemas de ambigüedad y complejidad en el código. Sin embargo, Java ofrece una alternativa para la herencia múltiple a través de las interfaces, que permiten a una clase implementar múltiples interfaces y heredar su comportamiento.

Para indicar que una clase hereda de otra utilizamos la palabra clave `extends`, que indica que queremos que cierta clase herede las funcionalidades de otra. Si no queremos que se pueda especializar una clase (que no se pueda heredar de ella), la marcamos con la palabra clave `final`.



```
1 package poo;
2
3 no usages new *
4 public class Gato extends Animal {
5     no usages new *
6     public Gato(String nombre) {
7         super(nombre);
8     }
9
10    1 usage new *
11    public void greet() {
12        System.out.print("Meow!");
13    }
14 }
```

Figura 52. Ejemplo especialización de Animal a Gato. Fuente: elaboración propia



```
1 package poo;
2
3 no usages new *
4 public class Perro extends Animal {
5     no usages new *
6     public Perro(String nombre) {
7         super(nombre);
8     }
9
10    1 usage new *
11    public void greet() {
12        System.out.print("Woof!");
13    }
14 }
```

Figura 53. Ejemplo especialización de Animal a Perro. Fuente: elaboración propia

Otro concepto importante es el polimorfismo, que se refiere a la capacidad de los objetos de una subclase para ser tratados como objetos de su superclase. Esto permite que las diferentes subclases puedan tener diferentes implementaciones de los métodos heredados de la superclase.

La palabra "polimorfismo" proviene del griego "poly" que significa "muchos" y "morphos" que significa "formas". En el contexto de la programación, esto significa que una entidad

(generalmente una clase o un objeto) puede tomar muchas formas o comportarse de diferentes maneras.

En términos simples, el polimorfismo permite a los objetos de diferentes clases responder al mismo mensaje o método de manera diferente. En otras palabras, una misma operación puede ser realizada por diferentes objetos de diferentes clases de maneras distintas.

Hay dos tipos principales de polimorfismo: polimorfismo de sobrecarga y polimorfismo de sobreescritura.

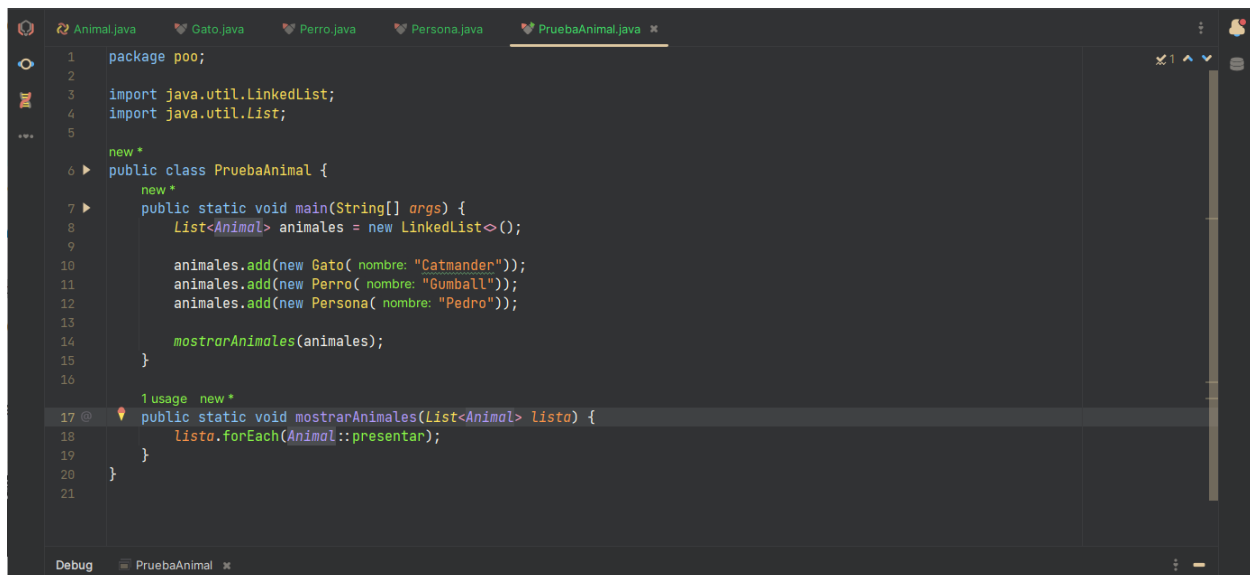
El polimorfismo de sobrecarga (también conocido como sobrecarga de método) ocurre cuando una clase tiene varios métodos con el mismo nombre, pero diferentes parámetros. En este caso, la clase puede usar diferentes implementaciones del método según los argumentos que se pasen.

Por ejemplo, en una clase "Calculadora", puede haber varios métodos llamados "sumar" con diferentes parámetros, como "sumar(int a, int b)", "sumar(double a, double b)" y "sumar(int a, int b, int c)". Cuando se llama al método "sumar", la clase utilizará la versión correspondiente según los argumentos que se le pasen.

El polimorfismo de sobreescritura (también conocido como sobreescritura de método) ocurre cuando una clase hija (o subclase) redefine un método que ha sido definido en su clase padre (o superclase). En este caso, la subclase proporciona su propia implementación del método, que anula la implementación del método de la superclase.

Por ejemplo, en una clase "Animal", puede haber un método "hacerSonido()" que devuelve una cadena que representa el sonido que hace el animal. La subclase "Perro" podría entonces anular este método y proporcionar su propia implementación del mismo que devuelve la cadena "¡Woof!", mientras que otra clase que hereda de Animal llamada Gato, retorna la cadena "¡Meow!".

El polimorfismo permite una mayor flexibilidad y modularidad en el diseño de programas orientados a objetos. Permite que diferentes objetos se comporten de maneras diferentes según el contexto en el que se usen. Esto hace que el código sea más fácil de mantener y modificar a medida que los requerimientos cambian.



```
1 package poo;
2
3 import java.util.LinkedList;
4 import java.util.List;
5
6 new *
7 public class PruebaAnimal {
8     new *
9     public static void main(String[] args) {
10         List<Animal> animales = new LinkedList<>();
11
12         animales.add(new Gato( nombre: "Catmander"));
13         animales.add(new Perro( nombre: "Gumball"));
14         animales.add(new Persona( nombre: "Pedro"));
15
16         mostrarAnimales(animales);
17     }
18
19     1 usage new *
20     public static void mostrarAnimales(List<Animal> lista) {
21         lista.forEach(Animal::presentar);
22     }
23 }
```

Figura 54. Ejemplo de polimorfismo implementando método abstracto. Fuente: elaboración propia

La encapsulación es otro concepto fundamental de la P00, y se refiere a la ocultación de la complejidad del objeto mediante el uso de métodos para acceder y modificar los atributos de la clase. De esta manera, se evita que los datos internos del objeto sean modificados de forma inadecuada desde fuera de la clase. Esto ofrece la capacidad de ocultar los detalles internos de una clase u objeto, de manera que sólo se muestre una interfaz pública para su uso.

La encapsulación también permite proteger los datos y comportamientos internos de una clase, evitando que otros objetos los manipulen directamente y provocando posibles errores o inconsistencias. De esta manera, se logra un mayor control y seguridad en la manipulación de los datos.

Para implementar la encapsulación, se utilizan modificadores de acceso para controlar el acceso a los miembros de una clase. Estos modificadores de acceso son: public, private, protected y default.

- Public: los miembros públicos de una clase son accesibles desde cualquier parte del código.
- Private: los miembros privados sólo son accesibles desde dentro de la misma clase.
- Protected: los miembros protegidos son accesibles desde la misma clase y desde clases hijas o subclases.
- Default: los miembros con acceso por defecto sólo son accesibles desde clases del mismo paquete.

Al definir los miembros de una clase, se deben elegir los modificadores de acceso adecuados para garantizar una buena encapsulación. Es decir, los miembros que no deban ser modificados desde fuera de la clase, se deben declarar como privados y se deben proporcionar métodos públicos (getters y setters) para acceder a ellos.

La encapsulación también permite la abstracción, otro de los pilares de la programación orientada a objetos. Al ocultar los detalles internos de una clase, se puede mostrar una interfaz más abstracta y sencilla para su uso, lo que facilita su comprensión y utilización.

En la POO, también se utilizan conceptos como la abstracción, que permite modelar objetos de manera simplificada, y la interfaz, que define un conjunto de métodos que una clase debe implementar para cumplir con un contrato específico. La clase Animal expuesta en la figura 47 es una clase abstracta que simplemente define un conjunto de métodos comunes a los animales.

Apéndice

A continuación, se lista una serie de definiciones de aquellos términos que se introdujeron en el manual y que puedan haber resultados confusos o desconocidos para el lector o lectora.

- Algoritmo: Un algoritmo es un conjunto de instrucciones bien definidas, ordenadas y finitas que permiten resolver un problema o realizar una tarea determinada.
- Bytecode: También llamado *bytecode Java*, es el tipo de instrucciones capaces de ser interpretadas por la máquina virtual de Java (JVM). Se encuentra en los ficheros con extensión `.class` que puede generar un compilador de java. Estas instrucciones son interpretadas por un traductor JIT (Just-In-Time) en tiempo de ejecución el cual las traduce al lenguaje máquina de la plataforma subyacente.
- Clase: plantilla que representa atributos de una entidad junto con operaciones sobre como operar sobre los mismos, abstrayéndonos de detalles irrelevantes de estos.
- Expresión: combinación de valores constantes o literales, funciones y variables que se interpretan de acuerdo a las normas de precedencia y asociación de un lenguaje.
- Expresión lógica: expresión cuya evaluación retorna un valor que puede ser cierto o verdadero.
- Firma (Function Signature): también conocida como su "function signature" en inglés, es la combinación de su nombre y los tipos de sus parámetros en orden. Es una forma de identificar de manera única una función en un programa. Por ejemplo, la firma de una función llamada "sumar" que toma dos parámetros enteros sería *sumar(int, int)*.
- Función: Una función es un bloque de código que realiza una tarea específica y devuelve un resultado. En Java, las funciones también se llaman métodos y se definen dentro de una clase. Una función puede tomar cero o más parámetros, realizar operaciones y devolver un valor. El valor devuelto por una función se especifica en su declaración y se llama tipo de retorno.
- Hardware: Es el conjunto de elementos físicos y palpables que constituyen una computadora o sistema informático. Destacan entre ellos los dispositivos de entrada como el teclado o ratón y los de salida como el monitor.
- IDE: Un IDE (Integrated Development Environment) es un entorno de desarrollo integrado que proporciona herramientas y servicios para simplificar y acelerar el desarrollo de software. Por lo general, un IDE combina un editor de código, un depurador, un compilador o intérprete, y otras herramientas de productividad para facilitar la programación.
- Iterable: es cualquier objeto que puede ser iterado, lo que significa que se puede recorrer su contenido elemento por elemento. Para hacer esto, un objeto iterable debe implementar la

interfaz "Iterable", que define un método *iterator()* que devuelve un objeto Iterator que se utiliza para recorrer los elementos del objeto iterable.

- **Módulo:** Un módulo en Java es una unidad de código que encapsula un conjunto relacionado de clases, interfaces y recursos.
- **Paradigma:** Un paradigma de programación es un enfoque o modelo de programación que establece cómo se deben estructurar y organizar los programas para resolver problemas. En Java, los paradigmas de programación más comunes son la programación orientada a objetos (OOP), la programación funcional y la programación procedimental.
- **Procedimiento:** Un procedimiento en Java es una sección de código que realiza una tarea específica y devuelve un resultado o efecto secundario. Los procedimientos también se conocen como funciones o métodos y son una parte fundamental de la programación estructurada y orientada a objetos en Java.
- **Proceso:** en un sistema operativo, representa un programa en ejecución. Los procesos normalmente son las entidades que mantienen en funcionamiento nuestro sistema ofreciendo servicios como lectura de datos, o control sobre nuestro hardware, de ellos el sistema operativo generalmente guarda información como el identificador, el consumo de memoria, tiempo en ejecución, entre otros.

Bibliografía

- [1 Wikipedia, «Software,» Wikipedia, 2023 Marzo 31. [En línea]. Available:
] <https://es.wikipedia.org/wiki/Software>. [Último acceso: 2023 Abril 3].
- [Wikipedia, «Programa informático,» Wikipedia, 24 Marzo 2023. [En línea]. Available:
2 https://es.wikipedia.org/wiki/Programa_inform%C3%A1tico. [Último acceso: 6 Abril 2023].
]
- [Wikipedia, «Pseudocódigo,» Wikipedia, 20 Febrero 2023. [En línea]. Available:
3 <https://es.wikipedia.org/wiki/Pseudoc%C3%B3digo>. [Último acceso: 8 Abril 2023].
]
- [F. H. Chowdhury, «The Java Handbook - Learn Java,» FreeCdeChamp, 7 Septiembre 2022.
4 [En línea]. Available: <https://www.freecodecamp.org/news/the-java-handbook/>. [Último
] acceso: 10 Abril 2023].
- [visure, «Qué son los requisitos no funcionales: ejemplos, definición, guía completa,»
5 VISURE, [En línea]. Available: <https://visuresolutions.com/es/blog/requerimientos-no-funcionales/#:~:text=Los%20requisitos%20no%20funcionales%20son%20las%20restricciones%20o%20los%20requisitos,la%20confiabilidad%20y%20muchos%20m%C3%A1s..> [Último
] acceso: 10 Abril 2023].
- [6 Oracle, «Formatter,» [En línea]. Available:
] <https://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html#syntax>. [Último acceso:
2023 abril 22].
- [7 Wikipedia, «Máquina virtual Java,» 14 abril 2023. [En línea]. Available:
] https://es.wikipedia.org/wiki/M%C3%A1quina_virtual_Java. [Último acceso: 2023 abril 23].
- [Wikipedia, «Java Lenguaje de programación(),» 2023 marzo 17. [En línea]. Available:
8 [https://es.wikipedia.org/wiki/Java_\(lenguaje_de_programaci%C3%B3n\)#Programaci%C3%B3n](https://es.wikipedia.org/wiki/Java_(lenguaje_de_programaci%C3%B3n)#Programaci%C3%B3n)
] n. [Último acceso: 23 abril 2023].
- [9 M. Biel, «Java Collections Framework Video Tutorial,» [En línea]. Available: <https://marcus-biel.com/java-collections-framework/>. [Último acceso: 23 abril 2023].
- [1 Oracle, «Java™ Platform, Standard Edition 7,» [En línea]. Available:
0 <https://docs.oracle.com/javase/7/docs/api/>. [Último acceso: 23 abril 2023].
]
- [1 Wikipedia, «Programación orientada a objetos,» 2023 marzo 2023. [En línea]. Available:
1] https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos. [Último acceso: 23
] abril 2023].

- [1 codejavu, «Que son las matrices en Java,» 26 junio 2015. [En línea]. Available:
2 <http://codejavu.blogspot.com/2015/06/que-son-las-matrices-en-java-arreglos.html>.
] [Último acceso: 23 abril 2023].
- [1 R. Marín, «Arrays: Dimensionalidad en varios lenguajes de programación,» 27 febrero 2019.
3 [En línea]. Available: <https://www.inesem.es/revistadigital/informatica-y-tics/arrays/>.
] [Último acceso: 23 abril 2023].
- [1 J. M. Alarcón, «Los conceptos fundamentales sobre Programación Orientada Objetos
4 explicados de manera simple,» 26 julio 2021. [En línea]. Available:
] <https://www.campusmvp.es/recursos/post/los-conceptos-fundamentales-sobre-programacion-orientada-objetos-explicados-de-manera-simple.aspx>. [Último acceso: 23
abril 2023].