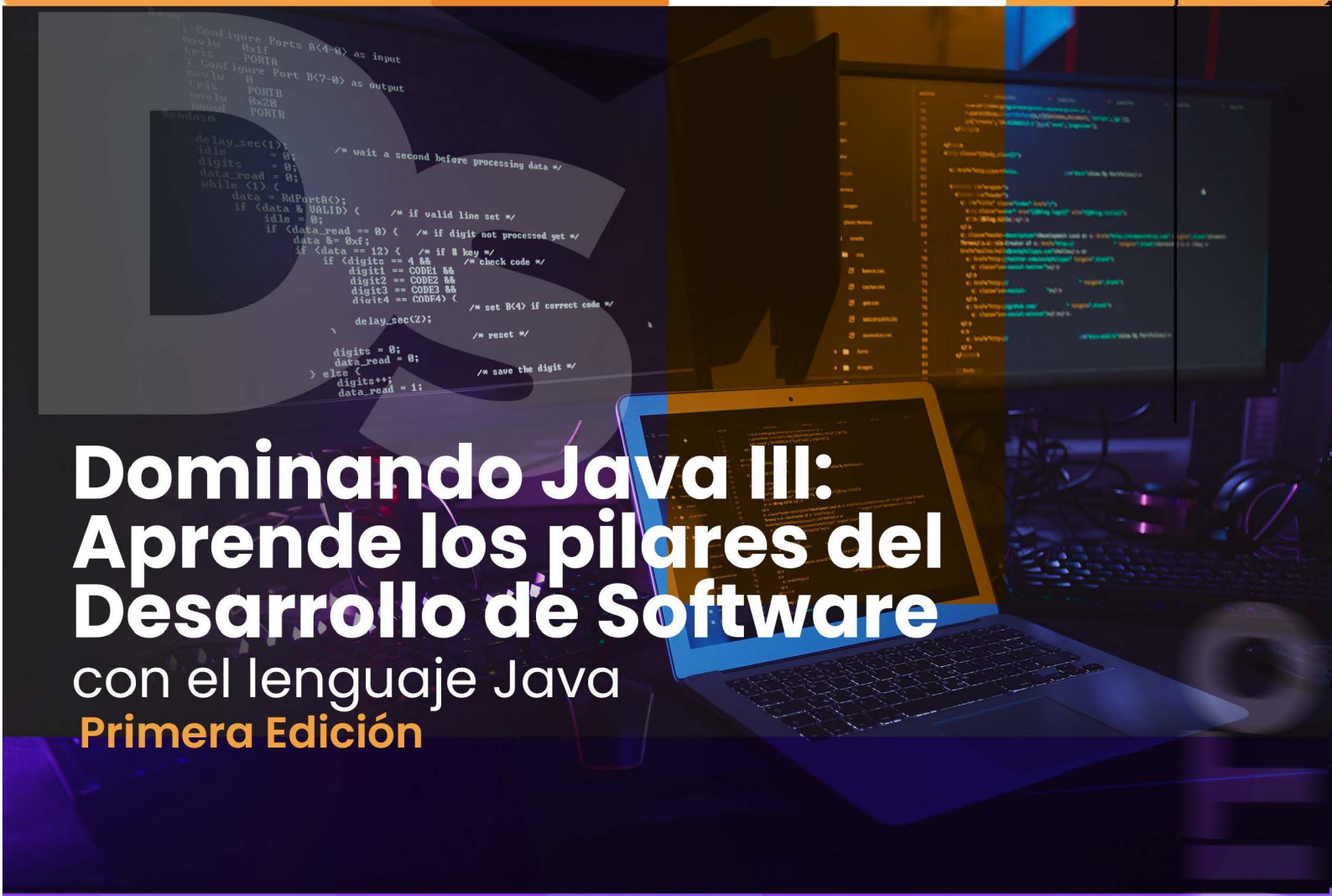
DOXA EDITION

Autor:
ELVIS PACHACAMA C.

2024



Dominando Java III: Aprende los pilares del Desarrollo de Software

con el lenguaje Java

Primera Edición

ITQ
WWW.ITQ.EDU.EC
INVESTIGACIÓN

ISBN: 978-9942-7328-1-1

INSTITUTO SUPERIOR TECNOLÓGICO QUITO
Excelencia en Educación Superior

DOMINANDO JAVA III:

APRENDE LOS PILARES DEL DESARROLLO DE SOFTWARE CON EL LENGUAJE JAVA

AUTOR:

ELVIS DAVID PACHACAMA CABEZAS

PRIMERA EDICIÓN

AÑO: 2024

TRABAJO EN EDICIÓN:

DOXA EDITION

DIRECCIÓN EDITORIAL: DIEGO JAVIER BASTIDAS LOGROÑO

EDITOR EXTERNO: DAVID FABIAN CEVALLOS SALAS

ISBN:978-9942-7328-1-1

DEDICATORIA

Esta obra está dedicada a los dos pilares de mi vida, Grace Amparo Cabezas Salazar y José Ramón Pachacama Tipán, que sin ellos no hubiera podido lograr todo lo que tengo. Gracias por todos sus esfuerzos lágrimas desvelos por verme triunfar.

Y para ti mi Keylita que estas en el cielo sé que me vas a cuidar y guiar cada paso que dé.

AGRADECIMIENTO

Quiero expresar mi más sincero agradecimiento a todas aquellas personas que hicieron posible la creación de mi primer libro.

En primer lugar, quiero agradecer a mi familia por su amor, apoyo y paciencia durante todo el proceso. Gracias por haberme brindado el espacio y el tiempo necesario para poder concentrarme en esta tarea y por haber sido mi fuente de inspiración en cada página.

También quiero agradecer a mis amigos y colegas por su constante motivación, por haberme brindado retroalimentación y sugerencias valiosas durante la escritura de este libro. Gracias por su amistad y por haber compartido conmigo momentos de alegría y distracción.

Un agradecimiento especial a mi editor, por su profesionalismo, dedicación y paciencia en la revisión y edición de este libro. Gracias por haberme guiado en cada paso del proceso de publicación y por haber hecho posible la materialización de este proyecto.

Finalmente, quiero agradecer a mis lectores por su interés en mi trabajo y por brindarme la oportunidad de compartir mis ideas, pensamientos y pasión por la educación. Espero que este libro les brinde una experiencia única y significativa.

SOBRE EL AUTOR

Elvis Pachacama Cabezas es un educador y profesional con un título de tercer nivel en Ingeniería en Tecnologías de la Información, obtenido en la Universidad Estatal de Sumy en Ucrania (SUMDU). Se destaca por su participación en la vida estudiantil, ya que fue representante de los estudiantes ecuatorianos en el Consejo Estudiantil de la universidad, lo que sugiere un compromiso activo con los asuntos estudiantiles.

Además, Elvis se graduó con honores y recibió reconocimiento por su destacado desempeño académico en su título universitario. Esta distinción refleja su dedicación y excelencia en sus estudios. Actualmente, Elvis Pachacama Cabezas es docente en el Instituto Superior Tecnológico Quito, donde imparte cátedra en la Carrera de Desarrollo de Software. Su posición como educador indica una dedicación al campo de la enseñanza y sugiere que posee conocimientos prácticos en el desarrollo de software, que comparte con los estudiantes.

Adicionalmente, es relevante mencionar que Elvis está próximo a cursar la maestría en inteligencia artificial en la Universidad Internacional de Valencia. Este detalle indica su interés en profundizar sus conocimientos en un campo avanzado y en constante evolución, como es la inteligencia artificial.

CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN AL CONTENIDO DEL LIBRO

Bienvenido al apasionante mundo de la programación en Java. Este libro está diseñado para ayudarlo a dar sus primeros pasos en el aprendizaje del lenguaje de programación Java y proporciona una base sólida para convertirse en un programador competente y versátil.

Java es un lenguaje de programación ampliamente utilizado y reconocido en la industria del desarrollo de software. Su popularidad se debe a su enfoque en la portabilidad, la flexibilidad y la facilidad de uso, lo que la convierte en la opción preferida para desarrollar de aplicaciones empresariales, aplicaciones móviles, sistemas integrados y muchas otras soluciones tecnológicas.

En este libro, nos sumergiremos en el mundo de Java desde cero, por lo que no se requiere experiencia previa en programación. Empezaremos con los conceptos fundamentales y poco a poco avanzaremos hacia temas más avanzados. El objetivo es que, al final de este curso, pueda crear programas funcionales y comprender cómo funciona Java.

¿Qué encontraras en este libro?

Introducción a Java. Este capítulo presenta técnicas de programación a programadores novatos y recuerda a los programadores experimentados estas técnicas básicas, y cubre técnicas utilizadas para resolver problemas informáticos.

Las etapas del método clásico de desarrollo de programas incluyen: análisis de problemas, desarrollo de algoritmos, codificación o implementación de algoritmos en un lenguaje de programación de alto nivel, compilación, ejecución del programa original, verificación y prueba del programa; seguido del mantenimiento y documentación del programa.

Aunque este libro se ocupa principalmente del desarrollo práctico de programas, técnicas y métodos, se introducen brevemente las fases de programación clásicas, como el análisis de requisitos y la especificación del dominio del problema a resolver; para ello cabe mencionar que sí, los dos modelos de programación más utilizados en el ámbito educativo y profesional son: la programación estructurada y la programación orientada a objetos; porque Java es un lenguaje completamente orientado a objetos.

Metodología de programación, creación y desarrollo de programas en Java.

Este capítulo presenta técnicas de programación a programadores novatos y recuerda a los programadores experimentados estas técnicas básicas, y cubre técnicas utilizadas para resolver problemas informáticos.

Las etapas del método clásico de desarrollo de programas incluyen: análisis de problemas, desarrollo de algoritmos, codificación o implementación de algoritmos en un lenguaje de programación de alto nivel, compilación, ejecución del programa original, verificación y prueba del programa; seguido del mantenimiento y documentación del programa.

Aunque este libro se ocupa principalmente del desarrollo práctico de programas, técnicas y métodos, se introducen brevemente las fases de programación clásicas, como el análisis de requisitos y la especificación del dominio del problema a resolver; para ello cabe mencionar. Es importante señalar que los dos modelos de programación más utilizados en el ámbito educativo y profesional son: la programación estructurada y la programación orientada a objetos; Debido a que Java es un lenguaje totalmente orientado a objetos, este patrón se utiliza a lo largo del libro y sus aplicaciones son universales y están orientadas a Internet y a la Web.

Elementos básicos de Java. Hemos visto cómo crear nuestros propios programas, ahora analizaremos los conceptos básicos de Java.

Dado que este capítulo es importante en el desarrollo de aplicaciones, este capítulo cubre los conceptos teóricos y prácticos relacionados con la estructura del programa. Cubierto en el capítulo anterior, incluidos los siguientes temas:

- Estructura general de un programa en Java.
- Creación del programa.
- Elementos básicos que lo componen.
- Tipos de datos en Java y como se declaran.
- Concepto y declaración de variables.
- Operaciones básicas de entrada y salida.

Operadores y expresiones. Un programa escrito secuencialmente ejecutará declaraciones una tras otra; del primero al último, cada declaración se ejecutará solo una vez; El modo secuencial es adecuado para resolver problemas simples.

Pero para resolver problemas generales, es necesario poder comprobar el contenido de la notificación, siempre se debe hacer.

Una estructura o construcción de control determina la secuencia de ejecución o flujo de declaraciones; se divide en tres categorías según el flujo de ejecución: secuencia, selección y repetición.

Este capítulo analiza las sentencias if y switch, la selectividad o las construcciones condicionales que controlan si una sentencia o lista de sentencias se ejecuta en función de si se cumple una condición; Para soportar estas construcciones, Java tiene el tipo booleano.

Programación Orientada a Objetos. En este capítulo, aprenderás los principios básicos de la programación orientada a objetos (POO), que es el paradigma de programación más utilizado en Java. Nos enfocaremos en entender qué son las clases y objetos, y cómo se utilizan para representar entidades del mundo real en el código. Exploraremos conceptos fundamentales como:

- Clases y Objetos: Qué es una clase como modelo o plantilla de un objeto y cómo crear instancias (objetos) a partir de esta.
- Atributos y Métodos: Cómo los atributos representan las propiedades de un objeto y los métodos definen su comportamiento.
- Encapsulamiento: La técnica de restringir el acceso directo a los atributos de un objeto, proporcionando métodos públicos para su manipulación.
- Herencia: Cómo crear nuevas clases derivadas de clases existentes, reutilizando código y simplificando el mantenimiento.
- Polimorfismo: Permite a los objetos ser tratados como instancias de su superclase, mejorando la flexibilidad y extensibilidad del código.

Arreglos: En este capítulo, exploraremos los arreglos en Java, que son estructuras fundamentales para almacenar y manipular colecciones de datos. Veremos cómo declarar, inicializar y utilizar arreglos tanto de una dimensión (vectores) como de múltiples dimensiones (matrices).

- Declaración e Inicialización de Arreglos: Cómo crear arreglos en Java, tanto estáticos como dinámicos.
- Operaciones Básicas: Acceso, modificación, y recorrido de elementos en arreglos utilizando bucles for y for-each.
- Arreglos Multidimensionales: Introducción a los arreglos de dos dimensiones (matrices) y cómo se utilizan para representar tablas de datos.
- Manejo de Excepciones con Arreglos: Cómo manejar errores comunes, como índices fuera de rango, utilizando bloques try-catch.

Java Swing: En este capítulo, aprenderás a crear interfaces gráficas utilizando Java Swing, una de las bibliotecas más utilizadas para la creación de aplicaciones de escritorio

en Java. Aprenderemos a manejar componentes básicos y a diseñar ventanas interactivas.

- Componentes Básicos de Swing: Cómo utilizar componentes como JFrame (ventana principal), JPanel (paneles de contenedor), JLabel (etiquetas), JButton (botones), y JTextField (campos de texto).
- Contenedores y Gestores de Diseño: Cómo organizar los componentes en una ventana utilizando gestores de diseño (FlowLayout, BorderLayout, GridLayout, etc.) para crear interfaces visualmente agradables y funcionales.
- Eventos y Listeners: Manejo de eventos de usuario (como clics de botón y entradas de texto) utilizando ActionListener y otros oyentes de eventos.

Ejemplo Práctico: Crearemos una calculadora gráfica simple que permita a los usuarios realizar operaciones matemáticas básicas. La calculadora incluirá botones para los números y operadores, un campo de texto para mostrar resultados, y un ActionListener para manejar los clics de botones.

CAPÍTULO 1
INTRODUCCIÓN A JAVA

1.1. INTRODUCCIÓN A JAVA

1.1.1. HISTORIA Y CARACTERISTICAS DE JAVA

Java[1] fue desarrollado por un equipo de ingenieros de Sun Microsystems a principios de la década de los 90, el proyecto fue liderado por James Gosling. Inicialmente, el proyecto se llamó “Green Project” y tenía como objetivo desarrollar una plataforma de software para dispositivos electrónicos de consumo. Sim embargo, pronto se dieron cuenta que las limitaciones de los dispositivos de la época no permitirían ejecutar aplicaciones complejas en ellos.

En 1995, Sun Microsystems lanzó oficialmente Java como una plataforma de desarrollo de software para aplicaciones empresariales y de escritorio. Su lema principal era” Write once, run anywhere” (Escribe una vez, ejecuta en cualquier lugar), lo que significa que el código Java puede ser compilado en u n formato llamado “bytecode” y ejecutado en cualquier máquina virtual Java (JMV), independientemente del sistema operativo subyacente.

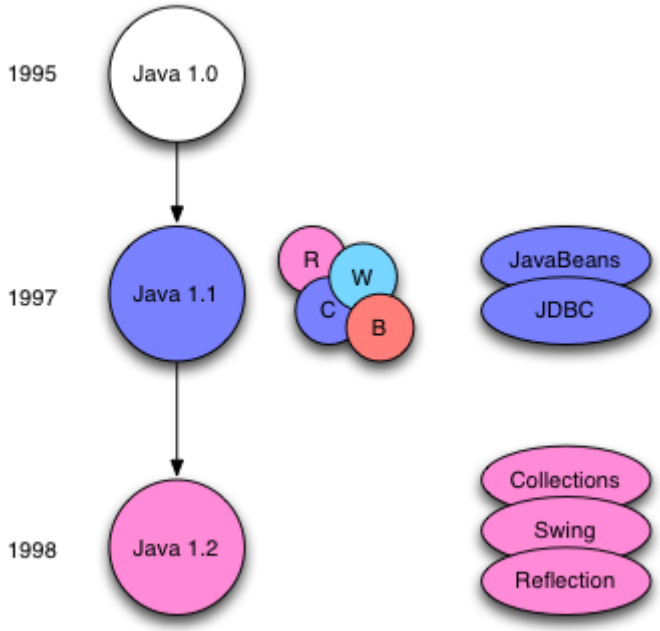
Java rápidamente ganó popularidad debido a su enfoque en la portabilidad, la seguridad y su capacidad para ejecutar aplicaciones en múltiples plataformas sin necesidad de reescribir el código fuente. En 2009, Sun Microsystems fue adquirida por Oracle Corporation, quien se convirtió en el principal patrocinador y desarrollador de Java.

A lo largo de los años, Java ha experimentado varias versiones principales y actualizaciones, cada una introduciendo nuevas características y mejoras. A continuación, se presenta un resumen de la historia de Java y sus versiones principales.

1.1.2. EVOLUCIÓN DE JAVA

[1] La fuente cuenta con una extensa documentación sobre la historia de Java y la evolución en el tiempo https://www.netec.com/post/historia-y-curiosidades-de-java

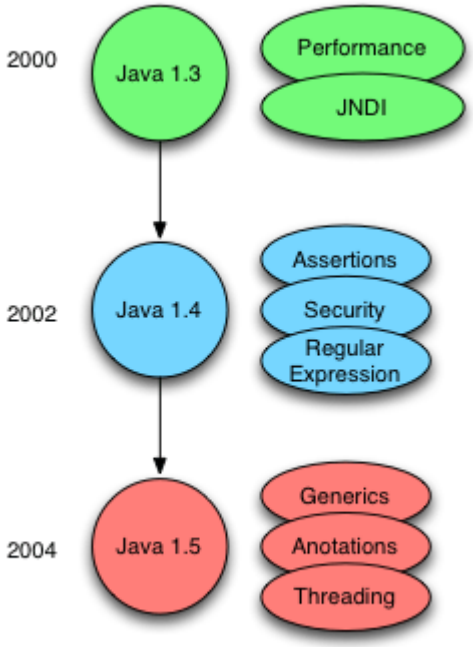
Figura 1
Versión Java (1-2)



Nota. El gráfico representa los cambios de java desde la versión 1.0 hasta la versión 1.2. Tomada de https://www.arquitecturajava.com/las-versiones-de-java/

- **JDK 1.0(enero de 1996):** Fue la primera versión oficial de Java, Introdujo el lenguaje de programación Java, la máquina virtual de Java (JMV) y las bibliotecas básicas de clases.
- **JDK 1.1(febrero de 1997):** Esta versión agregó nuevas bibliotecas y funcionalidades como, soporte para aplicaciones de red y acceso a base de datos mediante JDBC (Java Database Connectivity).
- **J2SE 1.2(diciembre de 1998):** Se introdujo la nueva marca “Java2”.

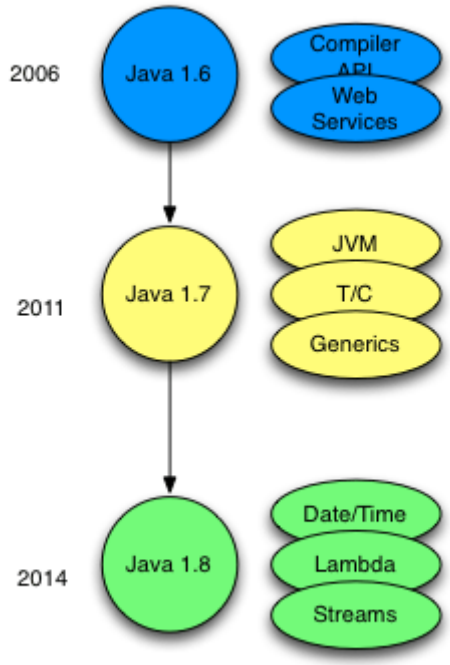
Figura 2
Versión de Java (1.3-1.5)



Nota. El gráfico representa los cambios de java desde la versión 1.3 hasta la versión 1.5. Tomada de https://www.arquitecturajava.com/las-versiones-de-java/

- **Versión 1.3:** Avances pequeño en cuanto a APIs, se añade soporte JNDI. Sin embargo, el avance en cuanto a la arquitectura de la maquina virtual es importante ya que aparece la máquina HotSpot con compilación JIT (**Just-in Time**).
- **Versión 1.4:** Se produce un salto importante en cuanto a nuevas Apis. Se incorpora un fuerte soporte XML, Expresiones Regulares, criptografía, etc.
- **Versión 1.5:** También denominada Java 5 se producen dos altos importantes a nivel del core del lenguaje. Por una parte, la inclusión de tipos genéricos que no tenía el mundo de las colecciones. Por otro lado, la inclusión del **concepto de metadatos con el uso de anotaciones.** Se amplía el soporte de APIs orientadas a la programación concurrente.

Figura 3
Versión de Java (1.6- 1.8)



Nota. El gráfico representa los cambios de java desde la versión 1.6 hasta la versión 1.8. Tomada de https://www.arquitecturajava.com/las-versiones-de-java/

- **Versión 1.6:** Esta versión contiene avances muy puntuales con la inclusión de un API de compilación "on-the-fly" que permitirá gestionar servicios web de forma cómoda.
- **Versión 1.7:** Otra versión cuyos cambios a nivel del lenguaje son imitados. Se produce una mejora de la máquina virtual incluyendo nuevos recolectores de basura.
- **Versión 1.8:** Llega Java 8 el gran salto en cuanto al lenguaje se refiere. Se abren las puertas a la programación funcional con el uso **de expresiones Lambda y Streams.** Se realiza una revisión de APIS y se actualiza de forma importante la gestión de fechas.
- **Versión 1.9:** En este reléase Java incluye el proyecto Jigsaw que permite modularizar el JDK. Y la distribución de packages. Se trata de un proyecto clave para el futuro enfoque de MicroServicios.
- **Versión 1.10:** Quizás la mejora más significativa es el añadido de la palabra **var** al lenguaje simplificado la inferencia de tipos cuando programamos de tal forma que a partir de ese momento es válido el uso de la siguiente estructura:

```
var lista= new ArrayList<Persona>();
```

Aparte de esto mejoraron temas de recolector de basura y programación concurrente.

- **Versión 1.12:** Añade la suite de MicrosoftBenchMark que permite al desarrollador realizar pruebas de rendimiento dentro del propio JDK.
- **Versión 1.15:** Fue lanzada el 15 de septiembre de 2020. Esta versión continúa la evolución de la plataforma Java y presenta diversas características y mejoras que mejoran la productividad de los desarrolladores y la eficacia del lenguaje.
- **Versión 1.17:** Fue lanzada el 14 de septiembre de 2021 y es una versión de soporte a largo plazo.
- **Versión 20:** La última actualización de Java, JDK 20, presenta versiones preliminares o de incubación de siete capacidades nuevas, incluidos subprocesos virtuales y concurrencia estructurada.

  Las siete funciones marcadas oficialmente para el lanzamiento, todas las cuales están en etapa de incubación o de vista previa, también incluye una API de vector, valores de ámbito, patrones de registros, coincidencia de patrones para declaraciones y expresiones de cambio, y una API de función y memoria externa.

1.1.3. CONFIGURACIÓN DEL ENTORNO DE DESARROLLO

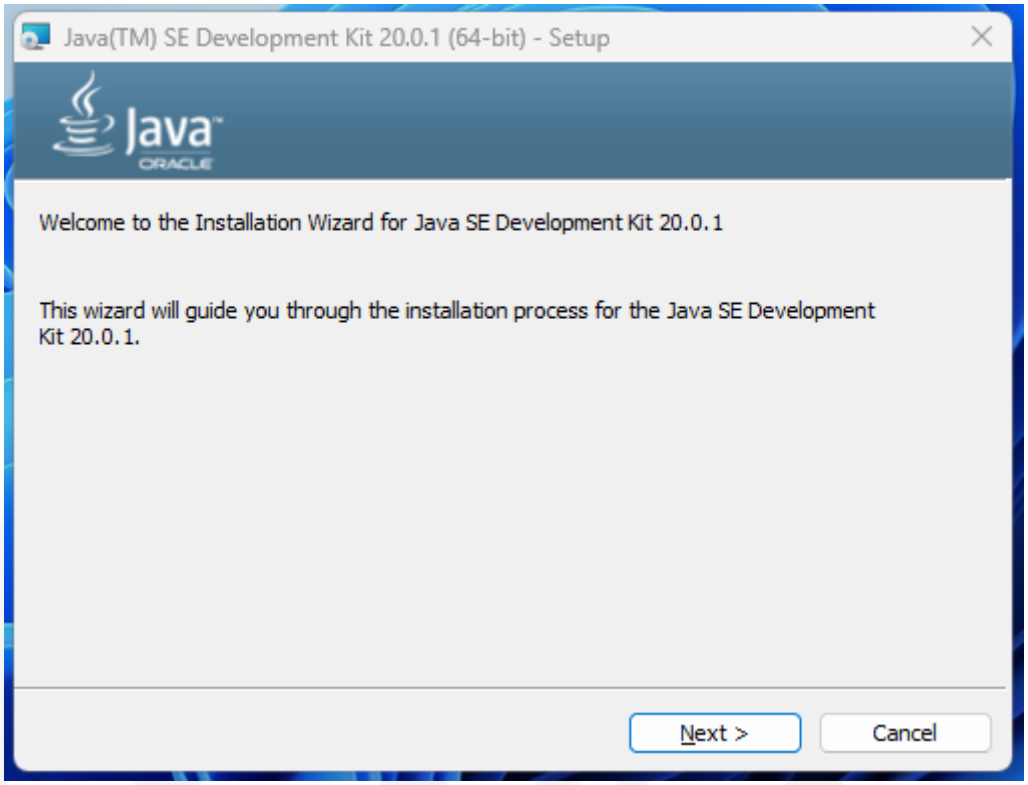
Java requiere algunas configuraciones para poder usarlo en nuestro equipo. Tenemos que tener en cuenta que puede haber.

La configuración del entorno de Java implica instalar y configurar el Java Development Kit (JDK) y, opcionalmente, un entorno de desarrollo integrado (IDE) para facilitar el desarrollo de aplicaciones en Java. A continuación, se presenta una guía para configurar el entorno de Java:

a. Instalación del JDK

Visita el sitio web oficial del Oracle https://download.oracle.com/java/20/latest/jdk-20_windows-x64_bin.exe (sha256) y descarga la versión más reciente del JDK compatible con tu sistema operativo.

Figura 4
Instalación de JDK

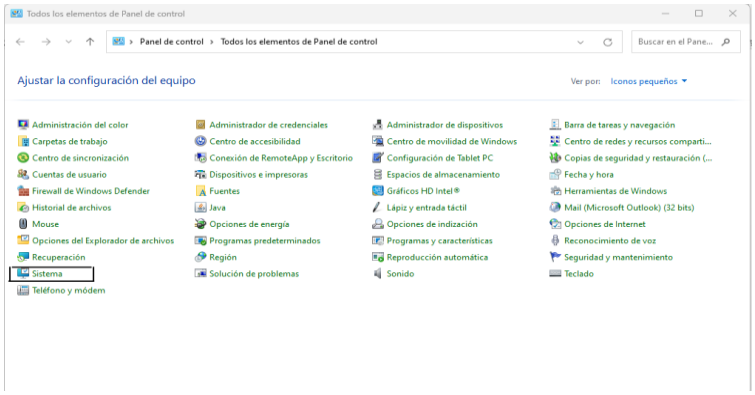


Nota. El gráfico representa el inicio de la instalación del JDK. Captura de pantalla.

b. Configuración de las variables de entorno:

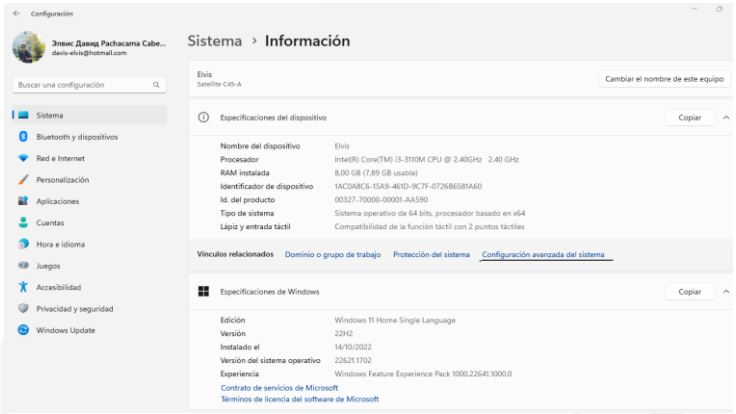
- Abre las variables de entorno del sistema en tu SO. En Windows, puedes hacerlo a través del Panel de control ->Sistema-> Configuración avanzada del sistema->Variables de entorno. En Linux y macOS, puedes editar el archivo. bashrco o .bash_profile en tu directorio de inicio.

Figura 5
Panel de control-Sistema



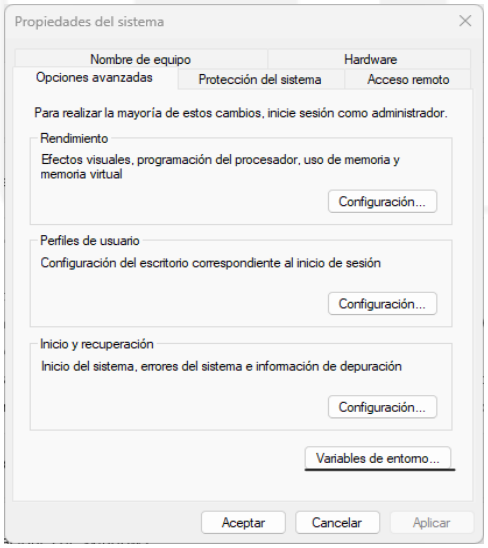
Nota. El gráfico representa el Panel de control de Windows para la configuración de las variables de entorno. Captura de pantalla.

Figura 6
Configuración avanzada



Nota. El gráfico representa la configuración de variables de entorno, mediante configuración avanzada. Captura de pantalla.

Figura 7
Variables de entorno



Nota. El gráfico representa donde se debe configurar las variables de entorno en el sistema operativo Windows. Captura de pantalla.

- Agregamos la ruta de instalación del JDK a la variable de entorno “PATH”. Esto permite que el sistema encuentre los comando y herramientas del JDK. Por ejemplo, en Windows, podrías agregar.

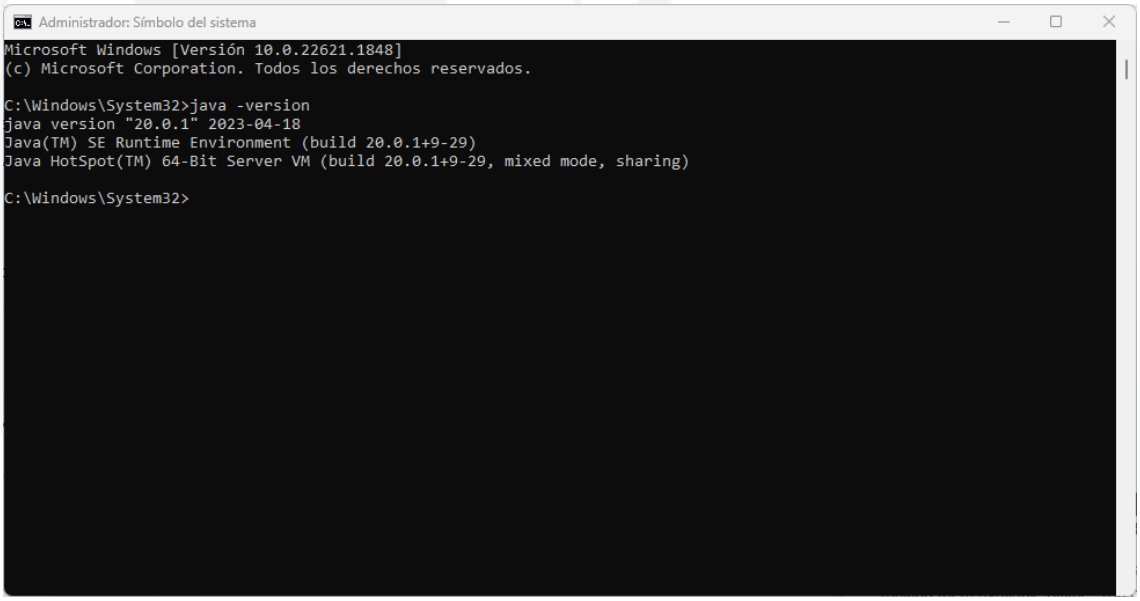
- Instalación de un IDE. Hay varios IDEs populares para el desarrollo de Java, como Eclipse, IntelliJIDEA y NetBeans. Elije el IDE que prefieras y descárgalo desde su sitio web oficial.

Sigue las instrucciones del instalador del IDE para completar la instalación en tu sistema.

Una vez instalado el IDE, puedes configurar la ruta JDK en la configuración del IDE para asegurarte que estas utilizando la versión correcta del JDK.

Después de completar estos pasos, tu entorno de Java debería estar configurado y listo para desarrollar aplicaciones en Java Puedes verificar la configuración ejecutando el comando "java -version" en la línea de comandos, que debería mostrar la versión del JDK instalado.

Figura 8
Versión de Java



Nota. El gráfico representa la versión de Java que tenemos instalado en nuestro entorno de trabajo. Captura de pantalla.

1.1.4. LA MÁQUINA VIRTUAL JAVA, ESTRUCTURA DE UN PROGRAMA EN JAVA.

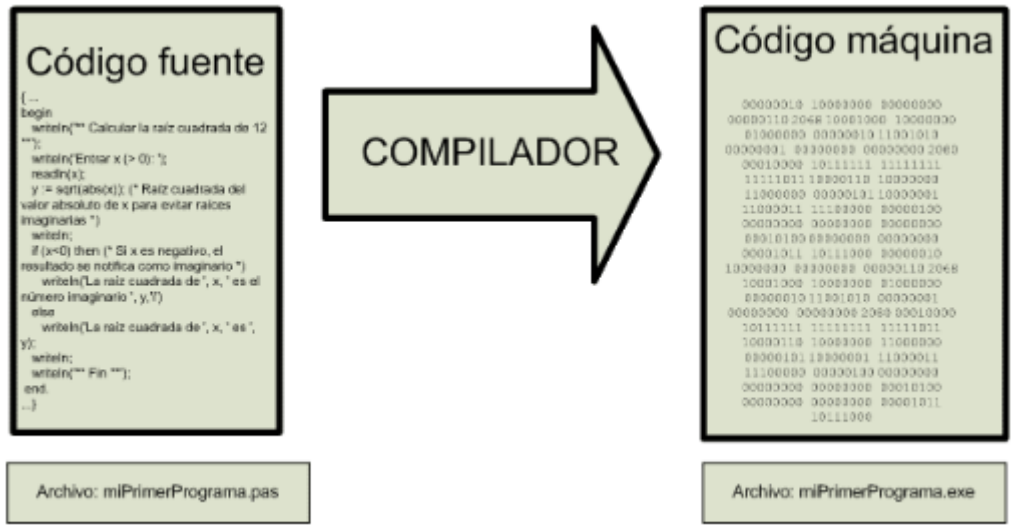
Vamos a crear nuestro primer programa, que nos servirá para entender y comprobar si hemos instalado y configurado correctamente Java antes vamos a reparar algunas definiciones importantes que nos permitirá comprender lo que vamos haciendo.

- **Compilación.** "Compilar" significa traducir el código escrito en "Lenguaje entendible por humanos", por ejemplo, Java, C, Pascal, etc. A un código que

entienda la computadora o código de máquina, este lenguaje no es entendido por nosotros. Se hace esto porque para nosotros nos seria imposible trabajar directamente con el lenguaje de máquina. Es por esta razón que en este libro vamos a utilizar el lenguaje de programación Java y luego emplearemos un traductor (compilador). La creación de programas en muchos lenguajes de programación se base en el siguiente proceso: escribir código fuentes, este código fuente se compila y se obtiene un programa ejecutable.

El compilador se encarga de evitar que un programa se pueda traducir su código fuente con errores y de hacer otras verificaciones previas, de modo que el código de máquina va a tener ciertas garantías de que cumplen cuando menos con los estándares de sintaxis obligatorios de un lenguaje.

Figura 9
Compilación de Programa en Pascal



Uno de los grandes problemas al compilar este programa es que si lo desarrollaste en un Sistema Operativo de Windows solo se va a ejecutar en Windows y no lo podrás hacer en Unix, Linux u otro Sistema Operativo. Sin embargo, una aplicación creada en Java no corresponde a la figura 9. Esta es una de las características novedosas que tenía Java en relación a otros lenguajes de programación cuando se creó la primera versión. Lo novedoso de Java fue que se hizo independiente del hardware y del Sistema Operativo en el que se ejecutaba.

RESUMEN DEL CAPÍTULO 1

En el capítulo de introducción a Java, se abordan los fundamentos esenciales de este lenguaje de programación. Java es un lenguaje versátil y orientado a objetos que se utiliza para desarrollar una amplia variedad de aplicaciones, desde aplicaciones web hasta aplicaciones móviles y sistemas embebidos.

El capítulo comienza explicando una breve historia del lenguaje de programación Java desde su primera versión hasta la versión final que hoy en día se está utilizando, como fue evolucionando cada una de las versiones.

El capítulo también se centra como configurar el entorno de desarrollo de Java, que incluye la instalación del Kit de Desarrollo de Java (JDK) y la configuración de las variables de entorno.

CAPITULO 2

METODOLOGÍA DE PROGRAMACIÓN, CREACIÓN Y DESARROLLO DE PROGRAMAS EN JAVA

2.1. RESOLUCIÓN DE PROBLEMAS CON JAVA.

En Java el proceso de resolución de problemas utilizando una computadora implica la escritura de programas y su posterior ejecución; por lo tanto, la programación es un proceso de resolución de problemas y preguntas. Existen diferentes técnicas para solucionar estos problemas, aunque el diseño y las propuestas arquitectónicas de software son creativas y se pueden considerar diferentes fases durante la programación. Las fases de resolución de problemas y sus características más importantes son:

- **Análisis del problema.** En el análisis del problema se examina todas las especificaciones de los requisitos entregados por el cliente, respecto al programa.
- **Diseño del algoritmo.** Una vez que se haya analizado el problema a resolver, se diseña una solución de un conjunto de instrucciones llamado algoritmo que o resuelva.
- **Codificación.** Para la solución del algoritmo se escribe en la sintaxis en cualquier lenguaje de programación de alto nivel, en este caso Java, el cual se obtendrá un código fuente que después será compilado.
- **Compilación y ejecución.** Es este punto el programa se empaqueta, en el caso de Java se interpreta y se ejecuta.
- **Verificación y depuración.** Una vez ejecutado nuestro programa, se verifica rigurosamente su sintaxis y se elimina todos los errores que aparezcan, a estos errores se los llama *bugs*.
- **Mantenimiento.** En este punto nuestro programa se actualiza y se modifica cada vez que sea necesario para que cumplan todas las necesidades y requerimientos del usuario, en esta fase se utilizan y mejoran los algoritmos realizando los cambios que sean necesarios.
- **Documentación.** En esta última fase del desarrollo de un programa vamos a documentar, todas las fases del ciclo de vida del software, esencialmente el análisis del problema, diseño y codificación; junto con los manuales de usuarios y manuales técnicos, así como las normas para el mantenimiento del software.

2.1.1. ANALISIS DEL PROBLEMA

En esta fase se requiere especificar el problema y definir claramente las tareas que el programa debe realizar y el resultado o solución que se necesita; en esta fase se divide en diferentes etapas.

- Entender el problema lo más claro posible.
- Comprender y enumerar los requerimientos o los requisitos del problema. Es necesario aclarar si nuestro software requiere interactuar con el usuario para la lectura de datos de entrada y también para poder especificar el formato de salida o los resultados.
- Cuando hablamos de especificar los datos supone que debemos representarlos en su formato correspondiente.
- Cuando nuestro programa realiza una salida de datos, se debe especificar cómo generar y dar formato a los resultados de salida.

Cuando el problema es complejo es necesario dividirlo o descomponerlo en subproblemas o módulos, una vez dividido el problema vamos a aplicar los pasos anteriores para poder analizar individualmente los requerimientos necesarios, así como la posible relación o conexión entre cada uno de los módulos. El análisis del problema requiere una definición clara, considerando exactamente lo que hará el programa y la solución deseada; dentro de esta fase se realizaran algunas preguntas a tener en cuenta.

- ¿Qué datos de entrada se requiere?
- ¿Cuál es la información de salida que se requiere?

2.1.2. DISEÑO DEL ALGORITMO

Después de haber analizado nuestro problema, la descripción y las especificaciones necesarias del mismo, el paso a seguir es diseñar un algoritmo que nos permita resolver dicho problema. Para esto nuestra computadora necesita que se le indique las tareas o acciones que se van a ejecutar en un orden sucesivo. Un algoritmo[2] es un método para resolver un problema mediante una serie de pasos precisos, definidos y finitos.

[2] La fuente cuenta con una extensa documentación sobre los conceptos, técnicas y métodos de diseño y construcción de algoritmos y programas: Joyanes L. Fundamentos de programación Algoritmos estructuras de datos y objetos 4a edición Madrid McGraw-Hill 2008

Los pasos sucesivos que indican las acciones o instrucciones a ejecutar por la máquina constituyen el algoritmo; para completarlo se requiere el diseño previo del mismo, lo cual es independiente, lo cual es independiente tanto del lenguaje de programación en que se expresará como de la computadora que lo ejecutará. En la resolución del problema, el algoritmo se puede expresar en un lenguaje de programación diferente y ejecutarse en una computadora distinta, pero será siempre el mismo; por ejemplo: en una analogía de la vida diaria, una receta de cocina se puede expresar en español, ingles o francés, pero independientemente del lenguaje que hable el cocinero, los pasos de la receta serán los mismos.

La especificación del orden del algoritmo en donde se realizan las instrucciones o acciones del programa se llama control del programa, este control se realiza con ordenes secuenciales o repetitivas.

Estos controles se estudiarán en capítulos más adelante. A estas instrucciones que realizamos en lenguajes de alto nivel también se los llama sentencias.

En la etapa del análisis del proceso de programación se va a determinar lo que el programa hará, en la siguiente etapa de diseño se definirá como se va a realizar la tarea solicitada.

Uno de los métodos más fáciles para resolver un problema complejo es dividiendo, eso quiere decir que nosotros vamos a subdividir nuestro problema es subproblemas y luego fraccionando estos subproblemas en otros de nivel más bajo hasta que se pueda implementar una solución, este método es conocido como método de diseño descendente o modular.

Cualquier *software* bien diseñado consta de un programa principal, este programa viene siendo el módulo de más alto nivel, el cual llama a otros módulos o subprogramas de nivel más bajo, que a su vez estos pueden llamar a otros subprogramas. Estos módulos pueden diseñarse, codificarse, comprobarse y depurarse independientemente, incluso pueden ser desarrollados por distintos programadores para luego ser acoplados en un solo software.

Como ya sabemos el proceso para el desarrollo de software hasta que se concluya son los siguientes.

1. Programar un módulo.
2. Comprobarlo.
3. Depurarlo, si es necesario.
4. Integrarlos con los módulos anteriores.

El proceso que va a convertir el análisis del problema en un diseño modular con refinamientos sucesivos que nos va a permitir una futura traducción a un lenguaje de programación de alto nivel se llama diseño del algoritmo, este diseño del algoritmo es muy aparte del lenguaje de programación en el que se va a codificar el programa.

El último de los pasos es el diseño del algoritmo en el cual se debe comprobar y verificar la exactitud de la solución del problema, esto quiere decir que vamos a obtener resultados exactos en un tiempo finito.

Los algoritmos se pueden representar de distintas formas, gráficamente por medio de fórmulas, diagramas de flujo N-S y seudocódigos, esta última técnica es la más utilizada en el mundo de la programación moderna y es la que más se recomienda para trabajar en Java. Por ejemplo, algoritmos en el cual nosotros podemos realizar tareas, "ir al estadio a ver un partido de futbol "Nacional-Barcelona"".

Ejemplo 2.1

1. Inicio.

2. Ver la programación del campeonato de futbol en la página oficial de la LigaPro.

3. **Si** no juega "Nacional-Barcelona" **entonces.**

3.1. Decidir otra actividad.

3.2. Bifurcar al paso 7.

4. Si hay fila entonces.

4.1. Formarse.

4.2. **Mientras** haya personas delante **hacer.**

4.2.1. Avanzar en la fila.

Fin de mientras.

Fin de si.

5. **Si** hay localidades **entonces.**

5.1. Compramos una entrada.

5.2. Ingresamos a nuestra localidad.

5.3. Localizamos nuestro asiento.

Mientras se juega el partido hacer.

5.3.1. Ver y alentar al Barcelona.

Fin_mientras.

Abandonar el estadio.

6. Volver a casa.

7. Fin.

Como podemos ver en el algoritmo anterior tenemos algunos aspectos a considerar. Primero tenemos algunas palabras reservadas que están escritas en negritas como por ejemplo (entonces, mientras, si_no). Estas palabras son fundamentales para el desarrollo de nuestro problema, ya que estas van a estar encargadas en la toma de decisiones de los procesos de automatizaciones de nuestro software, dentro del algoritmo también tenemos conceptos importantes de selección como estas (si-entonces, si_no, if-then-else), también tenemos palabra que me van a permitir repetir o realizar ciclos a secciones de nuestro algoritmo estas palabras pueden son (mientras-hacer, hacer-mientras o repetir-hasta o iterar- fin de iterar), como gran parte de los lenguajes de programación son desarrollados en ingles las palabras reservadas que vamos a utilizar son while-do y do-while que se encuentra en la mayoría de los algoritmos, especialmente los que se procesan datos. Cuando usamos bucles de decisión nos va a permitir seleccionar distintas alternativas de procesos o acciones a seguir o indicar la repetición una y otra vez las operaciones básicas.

2.1.3. CODIFICACIÓN

Una vez terminado las dos primeras fases empezamos con la codificación del algoritmo desarrollado anteriormente, para esto elegimos el lenguaje de programación que nosotros deseemos en este caso vamos a utilizar el lenguaje de programación Java. Como ya sabemos el algoritmo es independiente del lenguaje de programación utilizado, el código puede escribirse de una manera más fácil.

Como dijimos anteriormente para codificar nuestros algoritmos lo que debemos hacer es sustituir las palabras reservadas en español por su equivalente en inglés, recordemos que los lenguajes de programación son desarrollados en inglés por ese motivo vamos a reemplazar nuestras palabras reservadas, y todas las operaciones e instrucciones indicadas en nuestros algoritmos de lenguaje natural a lenguaje de programación de alto nivel, aquí vamos a seguir todas las reglas de sintaxis de este.

Al terminar de diseñar nuestro algoritmo y verificar que funcione correctamente, se procede a traducir esas instrucciones a un lenguaje de alto nivel como ya lo habíamos mencionado a Java en algunas de sus versiones actuales como ya lo vimos en el capítulo anterior, preferiblemente versiones actuales. Una que hemos escrito el código fuente lo que tenemos que hacer el pasarlo a un computador, mediante un editor de texto

siguiendo las reglas de sintaxis de Java, uno de los objetivos de este libro es que el lector nunca omita este pequeño detalle.

En el caso de Java, el código fuente se guarda en un archivo de texto el cual tiene una extensión .java, por ejemplo nombreDelPrograma.java después de haber generado el código fuente el siguiente paso es la compilación del código fuente. Si en el código fuente existe algún fallo, el compilador envía un mensaje de error, el programador debe encontrar y solucionar esos errores para luego enviar nuevamente al compilador estos pasos se deben realizar las veces que sean necesarias hasta que el compilador no envíe ningún error. Una vez que el compilador no envié ningún mensaje de error el compilador genera el código de máquina o el bytecode en el caso de Java.

2.1.4. COMPILACIÓN-INTERPRETACIÓN DE UN PROGRAMA EN JAVA

Una vez escrito el código fuente este código se debe traducir a un lenguaje de máquina, este proceso lo realiza el compilador y el sistema operativo. La ejecución del programa lo hace la Máquina Virtual de Java una vez que el compilador de paso al código fuente eso quiere decir que ya no tenga errores de sintaxis.

La etapa siguiente a la compilación es la ejecución de nuestro programa, en esta etapa se verifica la compilación exitosa eso quiere decir que el programa no tenga ningún error de sintaxis, esto no quiere decir que el programa funcione correctamente. Puede ocurrir que el programa interrumpa su ejecución y termina de forma inesperada por la existencia de errores lógicos, como por ejemplo una división de un número entre 0 o una mala identificación de una variable o un dato. Cuando sucede este tipo de errores lo que toca hacer es revisar todo el código fuente el algoritmo y en muchos casos es necesario revisar el análisis y las especificaciones del problema, esto se debe a que un análisis incorrecto o un mal diseño de especificaciones puede producir un mal algoritmo.

Este proceso se repite una y otra vez hasta que nuestro programa no produzca ningún error, después de la compilación se obtiene un programa todavía no ejecutable directamente llamado programa objeto. Suponiendo que no existan errores de compilación en nuestro programa fuente, se debe indicar al sistema operativo que se ejecute la fase de ensamblaje o vinculación, cargando el programa objeto con las bibliotecas de compilación, esta operación se realiza con un cargador.

2.1.5. VERIFICACIÓN Y DEPURACIÓN DE UN PROGRAMA EN JAVA

La verificación o depuración de un programa es el proceso de la ejecución con una amplia variedad de datos de entrada llamados de prueba que determinaran si el software tiene errores bugs, para verificar la funcionalidad de nuestro software se debe utilizar una gran

variedad de datos de distintos tipos para dichas pruebas y de esta forma se comprueba los límites y aspectos especiales de nuestro programa.

Los errores que se pueden encontrar en la verificación del software son:

1. Los errores de compilación son problemas detectados por el compilador de un programa informático al intentar traducir el código fuente escrito por un programador a un lenguaje que la computadora puede entender. Estos errores representan discrepancias o problemas en la sintaxis o estructura del código, impidiendo que el programa se compile correctamente y, por lo tanto, se ejecute. En otras palabras, los errores de compilación son fallos que deben corregirse antes de que el programa pueda funcionar adecuadamente, ya que el compilador no puede crear un programa ejecutable a partir del código fuente que contengan estos errores.
2. Los errores de ejecución se refieren a problemas que surgen cuando un programa informático está en proceso de ejecución o funcionamiento. Estos errores no se detectan durante la compilación del programa, sino que ocurren mientras el programa se está ejecutando. Como por ejemplo una división entre 0, estos resultados de situaciones inesperadas o condiciones inadecuadas que el programa no está preparado para manejar.
3. Los errores lógicos, también conocidos como errores de lógica, son fallos de un programa informático que no causan problemas al momento de su compilación o de ejecución, pero lo que si afecta negativamente el funcionamiento correcto del software debido a una lógica incorrecta o mal implementada en el código. Estos errores no suelen generar mensajes de error o alertas, por lo que pueden ser difíciles de detectar y suelen requerir un análisis cuidadoso del código para ser identificados y corregidos, si los errores persisten toca revisar el algoritmo y si persisten toca revisar el análisis del problema.

2.1.6. DOCUMENTACIÓN Y MANTENIMIENTO

La documentación se refiere al proceso de crear registros detallados y descripciones escritas que explican el funcionamiento de un sistema, programa o proceso. La documentación se utiliza para facilitar la comprensión, el uso y la gestión eficiente de estos elementos, así como para proporcionar información relevante a quienes trabajan con ellos.

La documentación de un sistema puede ser interna y externa: La primera se encuentra en las líneas de código dentro de nuestro programa, aquí se detalla que hace cada sección del código del programa, la segunda se encarga de un análisis, diagramas de flujo o

seudocódigo, manuales de usuarios, manuales técnicos con instrucciones para ejecutar el programa e interpretar los resultados que obtenemos.

La documentación es muy importante para poder corregir errores futuros en nuestra aplicación o bien realizar actualizaciones de este, este último proceso se llama mantenimiento del programa y con cada actualización la documentación también se debe actualizar de esta manera se facilita ajustes posteriores.

Algo común es realizar el manejo de versiones[3] mediante 3 números: X.Y.Z y cada uno indica una cosa diferente

- El primero (X) se le conoce como versión mayor y nos indica la versión principal del software.

  Ejemplo: 1.0.0, 3.0.0.0.

- El segundo (Y) se le conoce como versión menor y nos indica nuevas funcionalidades. Ejemplo 1.2.0, 3.3.0.

- El tercero (Z) se le conoce como revisión y nos indica que se hizo una revisión del código por algún fallo. Ejemplo 1.2.2, 3.3.4.

Ahora que conocemos el significado de cada número, viene una pregunta importante: ¿cómo sabemos cuándo cambiarlos y cual cambiar?

- Versión mayor o X, cuando agreguemos nuevas funcionalidades importantes, puede ser como un nuevo módulo o característica clave para la funcionalidad.

- Versión menor o Y, cuando hacemos correcciones menores, cuando arreglamos un error y se agregan funcionalidades que no son cruciales para el proyecto.

- Revisión o Z, cada vez que entregamos el proyecto.

Dentro del control de versión por números podemos agregar una clasificación por estabilidad del proyecto.

Las opciones que tenemos para este control son: Alpha, Beta

- **Alpha** es una versión inestable que es muy probable que tenga muchas opciones que mejorar, pero queremos que sea probada para encontrar errores y poder poner a prueba funcionalidades en la mayoría de los casos podemos decir que está casi listo el producto. Ejemplo 1.0Alpha, 1.0.a.1, 1.0a2.

[3] La fuente cuenta con una extensa información sobre el control de versiones en el desarrollo de aplicaciones informáticas: ¿Cómo se deciden las versiones del software? | EDteam

- **Beta** es una versión más estable que Alpha en la que contamos con el producto es su totalidad, y se desea realizar pruebas de rendimiento, usabilidad y funcionamiento en algunos módulos para ver cómo funciona bajo un ambiente no tan controlado. Aquí aparece el nombre de Beta Tester que escuchamos en el mundo del software. Ejemplo 2.0Beta, 2.0b,2.0b1.

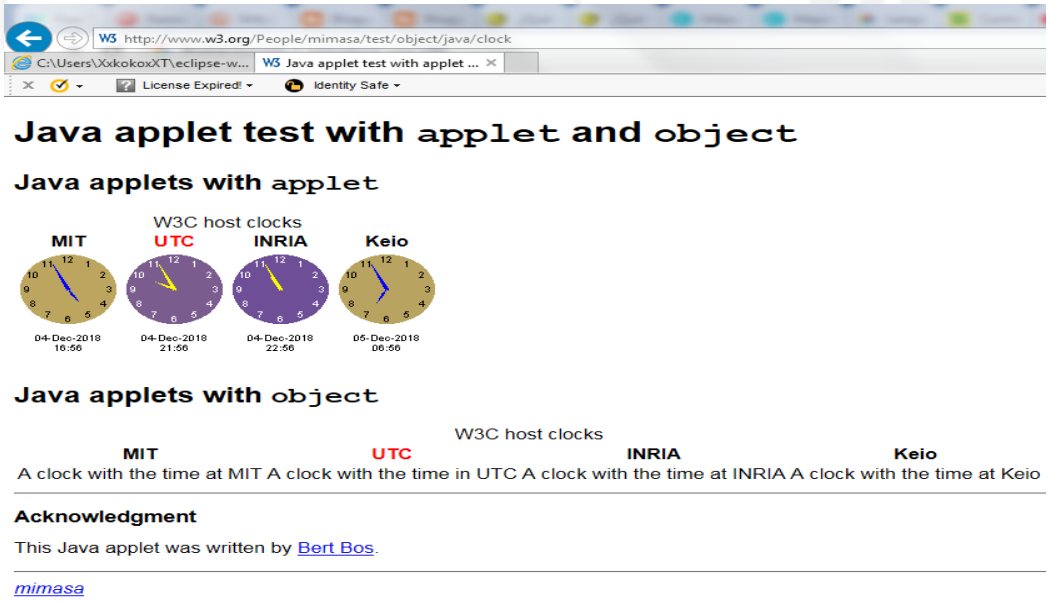
- El siguiente paso es **RC (Release Candidate)**, que es el último toque fino del software antes de salir y después de pasar por Beta. Ejemplo: 3.0-RC o también 3.0-RC1.

2.2. CREACION DE UN PROGRAMA EN JAVA

Java es un lenguaje de programación y plataforma informática con en cual podemos crear programas de cualquier tipo en sus diferentes plataformas. Esto hace de Java rápido, seguro, fiable y portable. En java podemos escribir básicamente 2 tipos de programas: applets y aplicaciones: y ambos requieren que nuestra computadora tenga instalado una versión del JMV (Java Virtual Machine) (Toro, 2018).

- **Applets.** Los applets son programas Java que se transmiten por internet y que se ejecutan incrustada dentro de una página web como se observa en la figura 10. Algo que destacar es que desde el 2014en adelante los diferentes navegadores como por ejemplo Google Chrome y Firefox han quitado el soporte al plugin de NPAPI.

Figura 10.
Ejemplo de un applet ejecutándose en un navegador web

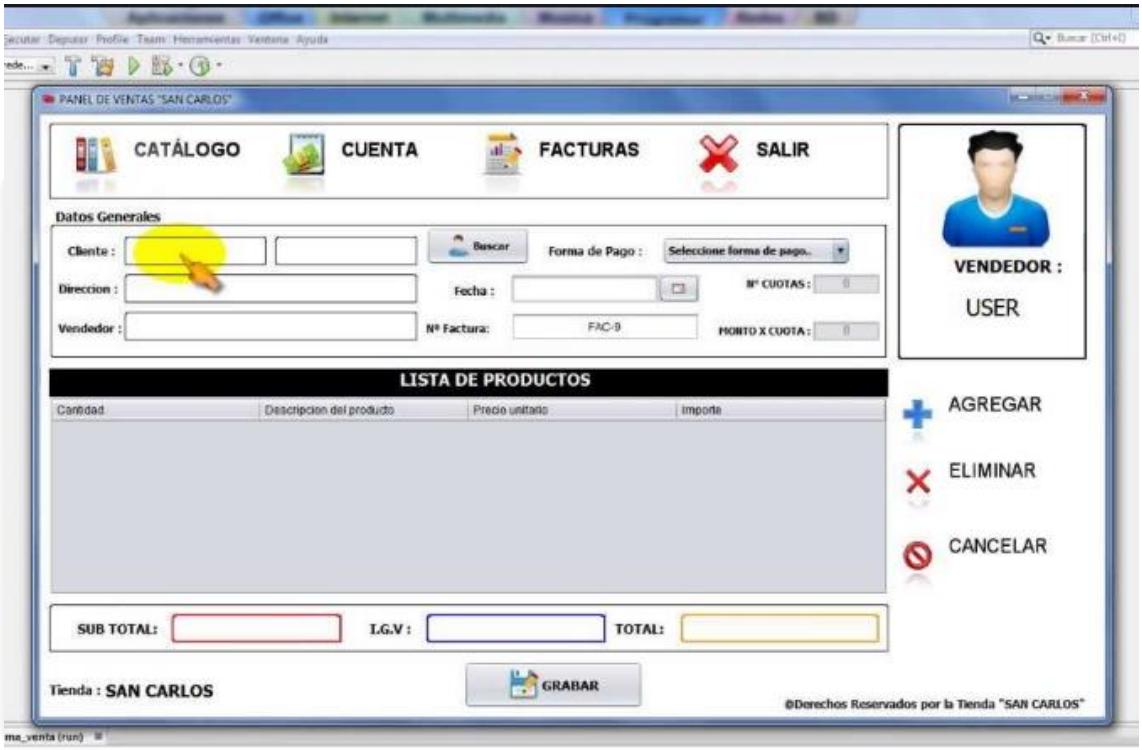


Nota. El gráfico representa como se muestra un applet desarrollado en Java. https://1.bp.blogspot.com/-

BjGnHF2jYNU/Xat7jd5wWzI/AAAAAAAAF_w/wVu3AvDbc20xgNsZzQC1WUrkU8Q1eZLbwCKgBGAsYHg/s1600/image.png

- **Aplicaciones.** Las aplicaciones son programas Java que son independientes de un navegador web, poseen ventana propia a diferencia de los applets pero que también necesitan de una versión de la JMV para poder se ejecutados en el sistema operativo (Toro, 2018).

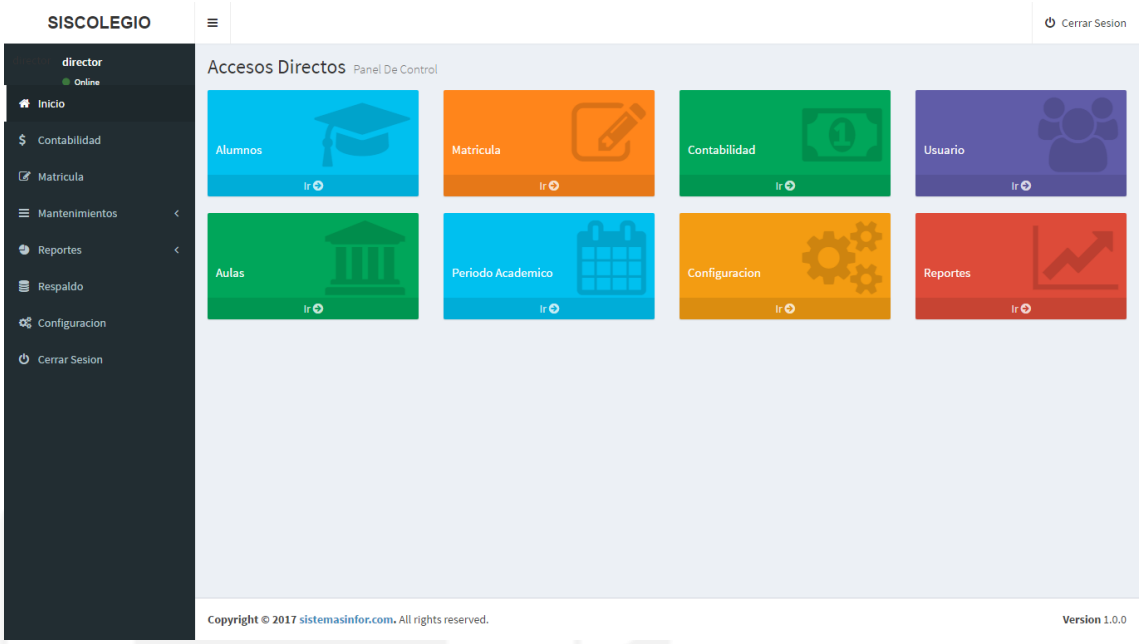
Figura 11
Aplicación Java de un sistema de ventas



Nota. El gráfico representa como se muestra una aplicación de escritorio desarrollado en Java SE. https://2.bp.blogspot.com/-LyoSzLLL0aI/XauDScW05WI/AAAAAAAAGAY/sMLUjJUzlCkNWE6cfSQflyAEXW30qdIQgCKgBGAsYHg/s1600/image.png

- **Aplicaciones Web.** Para desarrollar aplicaciones Web en Java se debe unir un conjunto de tecnologías como, por ejemplo. Servlets, JavaBeans, Java Server Page (JSP), Java Server Face (JSF) entre otras. Cuando se desarrolla aplicaciones Web en Java se trabaja con el paradigma **request-response** a diferencia de los applets las aplicaciones Webs se ejecutan del lado del servidor que interactúan con los clientes.

Figura 12.
Sistema Web de matrícula escolar



Nota. El gráfico representa como se muestra una aplicación web desarrollada en Java usando un distinto conjunto de tecnologías (imagen tomada del sistemainfor). https://2.bp.blogspot.com/-rn3ZXtX3SuE/WP5V84bGF7I/AAAAAAAAA8E/odZokM_eoFUTxGkNxHbWw2bm6lba5secACLcB/s1600/principal.png

- **Aplicaciones para dispositivos móviles, PDAs, electrodomésticos.** Para el desarrollo de aplicaciones también se debe unir distintas tecnologías estas aplicaciones están desarrolladas para ser ejecutadas en dispositivos móviles inteligentes, y electrodomésticos.

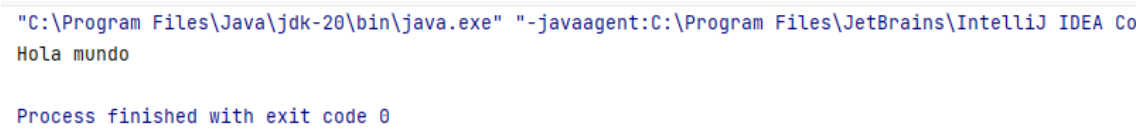
Como ya hemos detallado los dos tipos de programas que se pueden desarrollar en Java, en este libro nos vamos a centrar en el desarrollo de aplicaciones en lugar de applets, eso nos permitirá centrarnos y dirigir esfuerzos hacia los conceptos importantes de programación y en la resolución de problemas durante el proceso de aprendizaje.

Cuando se realiza un programa en Java este consta de una colección de clases, esta característica se cumple en los tipos de programas que se pueden desarrollar en Java; las clases contiene datos y métodos para su manipulación. En el siguiente ejemplo vamos a detallar un ejemplo de una aplicación simple en Java detallando como se ejecuta el mismo.

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

Al momento que ejecutamos el programa y si el compilador no muestra ningún error, en consola se visualizara lo siguiente:

Figura 13
Salida en pantalla del programa en Java



Nota. El gráfico representa la ejecución de un programa en Java la salida se lo realiza en consola. Captura de pantalla.

La salida del programa es un Hola Mundo esto se debe a la instrucción println lo que hace esta instrucción es enviar el mensaje contenido entre comillas a la pantalla. El proceso de la ejecución del programa lo veremos en la ilustración de la figura 14.

1. **Edición del programa.** Para este paso se puede utilizar cualquier editor de texto como Word, Notepad entre otros dependiendo el Sistema Operativo que se utilice siempre y cuando sigamos las reglas de sintaxis de Java que se describirá en los siguientes capítulos. Todo el código de este programa se le llamara programa fuente. En el caso del ejemplo anterior se lo llamo **HolaMundo.java**

   Normalmente para la codificación de las aplicaciones en Java se utiliza entorno de desarrollo integrados que facilita la codificación y desarrollo del software. Estos entornos de desarrollo integrados cuentan con editores donde podemos editar el código, compiladores, depuradores para la detección de errores de sintaxis y un programa que carga el código para ser ejecutado.

2. **Compilación de un programa en Java.** Después de haber escrito el código fuente y antes de compilar el programa debemos revisar la sintaxis del código y verificar si cumple con todas las reglas de Java. Una vez verificado, se procederá a compilar nuestro programa con el compilador **javac**, en donde se encarga de buscar errores, si no encuentra ningún error traduce el programa a bytecode; el programa se guarda en un archivo con la extensión **.class.** Tomando el ejemplo anterior, el archivo fuente HolaMundo.java se convirtió y se almaceno como HolaMundo.class.

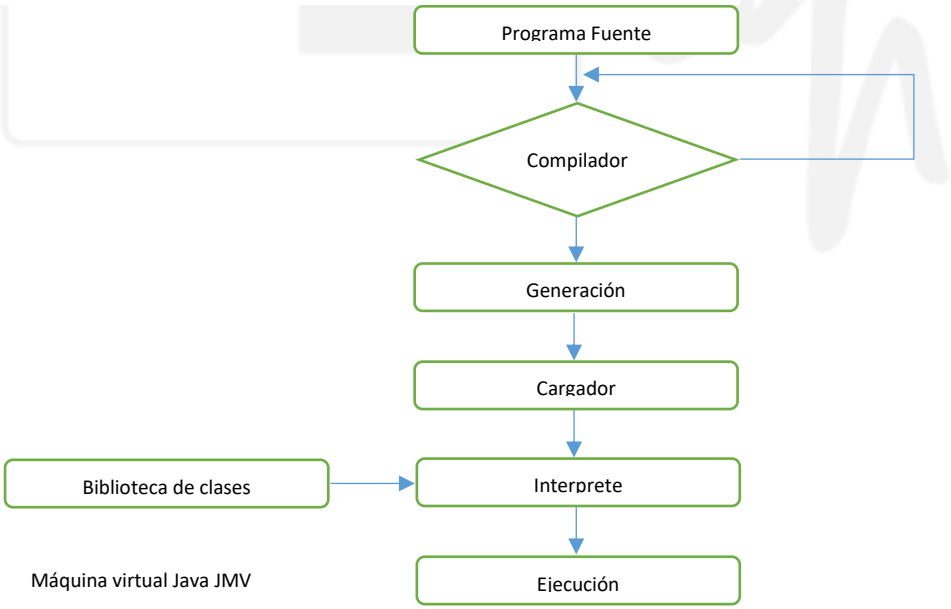
3. **Carga de un programa en la memoria.** Para ejecutar una aplicación en Java, lo que se hace es cargar el archivo en la memoria de la computadora. Cuando se escriben programas en Java como ya lo habíamos dicho se necita un entorno de desarrollo integrado llamados (IDE, por sus siglas en inglés), estos entornos

contienen bibliotecas de clases estas clases son programas que realizan tareas específicas, y también se cargan las bibliotecas desarrolladas por el usuario, que en Java se denomina paquetes(**packages**) que son un conjunto de clases que se relacionan entre sí. Para que la aplicación en Java se ejecute con éxito es necesario contar con otro programa llamado cargador (*loader*) este se encarga de enlazar y combinar todas las clases en bytecodes.

4. **Verificación de bytecodes y ejecución.** Ya que el programa cargador haya conectado los bytecode de las diferentes clases y el de su programa se carga en la memoria RAM. A medida que las clases se vayan cargando en la memoria principal un verificador de bytecodes aprueba que las clases están correctas y validas y no rompen ninguna restricción de seguridad de Java. Por último, el intérprete traduce cada una de las instrucciones de bytecode al lenguaje de máquina de su computadora independientemente del Sistema Operativo que se esté utilizando y enseguida se ejecuta.

Hoy en día las aplicaciones en Java se ejecutan más rápido ya que las máquinas virtuales Java han mejorado rotundamente y por lo general realizan simultáneamente combinaciones de interpretación, compilación inmediata, por tal motivo el proceso de ejecución se redujo considerablemente.

Figura 14.
Diagrama de Flujo de un programa en Java



Nota: El gráfico de representa el diagrama de flujo de la ejecución de un programa en Java.

2.3. METODOLOGÍA DE LA PROGRAMACIÓN

Cuando se desarrolla y se diseña software en Java, existen dos enfoques para diseñarlos: programación estructurada y programación orientada a objetos.

2.3.1. PROGRAMACIÓN ESTRUCTURADA

El método de la programación estructurada utiliza métodos tradicionales de programación, estos métodos se van utilizando desde la década de los 60 y 70, especialmente desde la creación de lenguajes de programación como Pascal por Niklaus Wirth. Con este tipo de programación se tiene un enfoque especifico en el cual se puede crear programas muy bien escritos y legibles. Con este tipo de método se escriben programas de acuerdo con ciertas reglas y técnicas.

Las reglas de programación4 o diseño estructurado se basan en la modularización, a su vez cada módulo se analiza para obtener una solución individual, lo cual significa que la programación estructurada tiene un diseño descendente.

Las técnicas de programación estructurada incluyen construcciones, estructuras o instrucciones básicas de control estos conceptos lo revisaremos en capítulos siguientes, los cuales son:

- Secuencias.
- Decisiones.
- Bucles.

La estructura básica de este método de programación indica el orden en el cual se van a ejecutar las instrucciones de un algoritmo o un programa.

Este método de programación se utiliza para programas pequeños, estos paradigmas o principios de organización resultan muy eficientes, lo que hace el programador es crear un conjunto de instrucciones en cierto lenguaje de programación, compilar y ejecutar ese conjunto de instrucciones, como ya lo habíamos dicho este método de programación solo funciona en programas pequeños, pero a medida que la dificultad de la solución del problema va siendo más complejo se va a utilizar otros tipos de métodos como la creación de métodos o funciones, en el cual se desglosa en unidades más pequeños.

[4] LA siguiente fuente con tiene un capítulo completo (1) donde se analiza y comparan con detalle ambos tipos de métodos de programación: Joyanes, L. y Sánchez, L. Programación en C++. Un enfoque práctico. Madrid MacGraw-Hill, 2006

2.3.2. PROGRAMACIÓN ORIENTADA A OBJETOS

La programación orientada a objetos (POO)5 es un paradigma de la programación que se basa en el concepto de "objetos" para moldear el mundo real y resolver problemas de programación.

La Programación Orientada a Objetos (POO) es un enfoque de desarrollo de software que se basa en la creación y manipulación de objetos, que son instancias de clases. En POO, los objetos representan entidades del mundo real y tiene atributos(datos) y métodos(funciones) que operan en esos datos.

El objetivo principal de la POO es organizar el código de manera modular y reutilizable al permitir la encapsulación de datos y la abstracción de comportamientos. Esto facilita la creación de sistemas complejos al dividirlos en objetos interconectados que colaboran que colaboran para cumplir con sus responsabilidades y al mismo tiempo, proporciona un alto nivel de modularidad, flexibilidad y mantenibilidad en el desarrollo del software.

En Java, la creación de operaciones requiere la escritura de los algoritmos correspondientes y su implementación. En el enfoque orientado a objetos, las múltiples operaciones necesarias de un programa utilizan métodos para implementar los algoritmos, los cuales, a su vez utilizarán instrucciones o sentencias de control, de selección, repetitivas o iterativas (Joyanes Aguilar & Zahonero Martinez, Programación en Java 6, 2011).

2.4. METODOLOGÍA DE DESARROLLO BASADA EN CLASES

En Java, una metodología de desarrollo basada en clases se refiere a la práctica de utilizar clases como el componente central y fundamental para organizar y estructura el código de un programa. La programación orientada a objetos (POO) es la base de esta metodología en Java. Detallaremos algunos elementos los cuales vamos a utilizar en esta metodología de programación.

- **Clases.** Las clases son plantillas o moldes que define la estructura y el comportamiento de los objetos.
- **Objetos.** Los objetos son instancias concretas de una clase. Se crean a partir de una clase y contiene daros específicos y pueden ejecutar métodos definidos en esa clase.
- **Encapsulamiento.** La encapsulación es el concepto de ocultar los detalles internos de una clase y proporcionar un interfaz publica para interactuar con los objetos.

[5] OOP. Object Oriented Programming

- **Herencia.** La herencia es un mecanismo que permite crear nuevas clases basadas en clases existentes.
- **Polimorfismo.** El polimorfismo permite que los objetos de diferentes clases se comprometen de manera similar al responder a un mismo conjunto de mensajes o métodos.
- **Abstracción.** La abstracción implica la creación de clases y objetos que representan conceptos abstractos y generalizados, ocultando los detalles innecesarios para centrarse em lo esencial.
- **Modularidad.** La modularidad se refiere a la división del programa en módulos o paquetes, donde cada módulo se relaciona con una clase o un conjunto de clases relacionadas.
- **Políticas de buenas prácticas.** Se siguen pautas y convenciones de codificación, como el uso de nombres de clases y métodos descriptivos, comentarios adecuados y diseño coherente, para asegurar que el código sea legible y mantenible.

Todos estos componentes lo vemos con más detalle en capítulos siguientes.

2.5. ENTORNOS DE PROGRAMACIÓN EN JAVA

Aprender a programar en Java se necesita aprender y conocer ciertas técnicas y metodologías para realizar el análisis, diseño y construcción del software.
Para realizar el aprendizaje practico se realiza de diferentes maneras y con diferentes herramientas que hacen la programación más fácil como:

- JDK Java Development Kit.
- IDE entono de desarrollo integrado.
- Editores de texto, compiladores/depuradores.

Estas herramientas hacen más fácil al programador al realizar un programa en Java. Lo que el programador debe hacer es conocer a profundidad la herramienta que va a utilizar es por eso que en este capítulo terminaremos de ver que es o que necesitamos para poder programar en Java.

2.5.1. EL KIT DE DESARROLLO JAVA: JDK 20

El JDK es una herramienta que sirve para crear programa en Java fue creada por Sun Microsystem, que consta de un conjunto de programas de líneas de ordenes que se

utilizan para crear, compilar y ejecutar programas en Java, cada nueva versión de Java se acompaña por el kit de desarrollo, al momento que de escribir este capítulo, la última versión del JDK es la 20.

"Durante más de 20 años, Java[6] ha permitido a los desarrolladores diseñar y construir la próxima generación de aplicaciones robustas, escalables y seguras", afirmó Goeorges Saab, vicepresidente senior de desarrollo de la plataforma Java y presidente de la Junta Directiva de OpenJDK. "Las nuevas e innovadoras mejoras en Java 20 reflejan la visión y los invaluables esfuerzos que la comunidad global de Java a contribuido a lo largo de la existencia de Java".

El ultimo kit de desarrollo de Java (JDK) proporciona actualizaciones y mejoras con siete propuestas de mejora de JDK(JEP). La mayoría de las actualizaciones son funciones de seguimiento que mejoran la funcionalidad introducida en versiones anteriores.

JDK 20 ofrece mejoras de lenguaje del proyecto OpenJDK Amber (Record Patterns and Pattern Matching for Switvh); mejoras de OpenJDK Project Panama para nterconectar Java Virtual Machine (JVM) y código nativo (Foreign Function & Memory API y Vector API); y características relacionadas con Project Loom(Valores con alcance, subprocesos virtuales y concurrencia estructurada), que agilizarán drásticamente el proceso de escritura, mantenimiento y observación de aplicaciones concurrentes de alto rendimiento.

"Hoy en día, las organizaciones enfrentan una presión cada vez mayor para utilizar sus recursos de la manera más inteligente y eficiente posible, lo que requiere que los desarrolladores busquen herramientas que agilicen el desarrollo de aplicaciones y al mismo tiempo ayuden a garantizar que sus organizaciones alcancen sus objetivos de cumplimiento y seguridad de TI", dijo Jay Lyman, analista de investigación senior de S&P.

Las actualizaciones más importantes entregadas en Java 20 son:

Actualizaciones y mejoras del idioma

- **JEP 432: Patrones de registro (segunda vista previa):** mejora el lenguaje Java al permitir a los usuarios anidar patrones de registro y tios de patrones para crear una forma poderosa, declarativa y componible de navegación y procesamiento de datos. Esto ayuda a aumentar la productividad de los

[6] La fuente tiene como referencia la página oficial de Oracle sobre la actualización de JDK 20 para el desarrollo de aplicaciones. https://www.oracle.com/news/announcement/oracle-releases-java-20-2023-03-21/

desarrolladores al permitirles ampliar la coincidencia de patrones para permitir consultas de datos más sofisticados y componible.

- **JEP 443: Coincidencia para patrones para Switch(cuarta vista previa):** al extender la coincidencia de patrones para switch, se puede probar una expresión con varios patrones, cada uno con una acción especifica, de modo que las consultas complejas orientadas a datos se puedan expresar de manera concisa y segura. Ampliar la expresividad y aplicabilidad de las expresiones y declaraciones de cambio ayuda a aumentar la productividad de los desarrolladores.

Características de la incubadora/vista previa del telar del proyecto

- **JEP 429: Valores de ámbito (incubadora):** permite compartir datos inmutables dentro y entre subprocesos, que se prefieren a las variables locales de subprocesos, especialmente cuando se utilizan una gran cantidad de subprocesos virtuales. Esto aumenta la facilidad de uso, la comprensibilidad, la solidez y el rendimiento.
- **JEP 436: Subprocesos virtuales (segunda vista previa):** agilice significativamente el proceso de escritura, mantenimiento y observación de aplicaciones concurrentes de alto rendimiento mediante la introducción de subprocesos virtuales livianos en la plataforma Java. Al permitir a los desarrolladores solucionar problemas, depurar y perfilar fácilmente aplicaciones simultáneas con herramientas y técnicas JDK existentes, los subprocesos virtuales ayudan a acelerar el desarrollo de aplicaciones.
- **JEP 437: Concurrencia estructurada (segunda incubadora):** simplifica la programación multiprocesos al tratar multitareas que se ejecutan en diferentes subprocesos como una sola unidad de trabajo. Esto ayuda a los equipos de desarrollo a optimizar el manejo y la cancelación de errores, mejorar la confiablidad y mejorar la observabilidad.

Características de la vista previa del proyecto Panamá.

- **JEP 434: API de memoria y funciones externas (segunda vista previa):** permite a los programas Java interoperen con código y datos fuera del tiempo de ejecución de Java. Al invocar eficientemente funciones externas (es decir, código fuera de la máquina virtual Java [JVM]) y al acceder de forma segura a memoria externa (es decir memoria no administrada por la JVM), esta característica permite a los programadores Java llamar a bibliotecas nativas y procesar datos nativos sin que requiere la interfaz nativa de Java. Esto aumenta

- la facilidad de uso, el rendimiento y la seguridad.
- **JEP 438: API vectorial (Quinta incubadora):** expresa cálculos vectoriales que se compilan de manera confiable en tiempo de ejecución en instrucciones vectoriales en arquitecturas de CPU compatibles. Esto aumenta el rendimiento en comparación con los cálculos escalares equivalente.

El lanzamiento de Java 20 es el resultado de una amplia colaboración entre los ingenieros de Oracle y otros miembros de la comunidad mundial de desarrolladores de Java a través de OpenJDK y Java Community Process (JCP). Además de las nuevas mejoras, Java 20 cuenta con el respaldo de Java Management Service , un servicio nativo de Oracle Cloud Infrastructure (OCI), que proporciona un panel único para ayudar a las organizaciones a administrar tiempos de ejecución y aplicaciones de Java en las instalaciones o en cualquier nube.2.6. ENTORNOS DE DESARROLLO INTEGRADO (IDE)

2.6. HERRAMIENTAS PARA DESARROLLO EN JAVA

Antes de empezar a practicar y desarrollar aplicaciones en Java en su computadora, debemos disponer de todas las herramientas adecuadas, para poder editar, compilar y ejecutar los programas en Java, muchas de las herramientas que vamos a utilizar son gratuitas y las puede descargar desde el sitio oficial de los desarrolladores.

Existen distintas versiones y populares de entornos de desarrollo para Java como Apache Netbeans, Eclipse JBuilder, IntellijIDEA entre otros para el propósito de nuestro libro vamos a utilizar IntelliJIDEA.

2.6.1. APACHE NETBEANS

Apache NetBeans, anteriormente conocido simplemente como “NeatBeans”, es un entorno de desarrollo integrado IDE de código abierto que se utiliza principalmente para desarrollar aplicaciones en Java, pero también admite varios otros lenguajes de programación, incluyendo PHP, HTML, JavaScript, c/C++, y más. Fue originalmente desarrollado por Sun Mycrosystems y más tarde adquirido por Oracle Corporation cuando esta última compró a Sun Mycrosystems. Sin embargo, en el 2016, Orcle donó el proyecto a la Apache Software Foundation, y desde entonces se conoce como Apache NetBeans.

Algunas de las principales características de Apache NetBeans incluyen:

1. Editor de código.
2. Gestión de Proyectos.
3. Depuración.

4. Soporte de Lenguaje múltiples.
5. Creación de interfaces gráficas.
6. Administración de dependencias.
7. Extensiones y complementos.
8. Compatibilidad y multiplataforma.

2.6.2. ECLIPSE

Eclipse es un entorno de desarrollo integrado IDE de código abierto ampliamente usado en la programación de software en Java. Fue inicialmente desarrollado por IBM en 2001 y luego donado a la Eclipse Foundation, una organización sin fines de lucro que se encarga de su desarrollo y mantenimiento. Eclipse es conocido por su flexibilidad y extensibilidad, lo que hace adecuado para una amplia variedad de lenguajes de programación y tipos de proyectos. Algunas de sus características son:

1. Editor de código.
2. Gestión de proyectos.
3. Depuración
4. Control de versiones.
5. Extensibilidad.
6. Soporte para múltiples lenguajes.
7. Desarrollo de aplicaciones empresariales
8. Comunidad activa.

2.6.3. INTELLIJ IDEA

Intellij IDEA es un entorno de desarrollo integrado (IDE) desarrollado por JetBrains, una empresa de software con sede en Republica Checa. IntelliJ IDEA es uno de los IDE más populares y ampliamente utilizados en la industria del desarrollo de software, y se utiliza principalmente para la programación en lenguajes como Java, Kotlin, Groovy, Scala y otros.

Algunas de las características y ventajas clase de IntelliJ IDEA incluyen:

1. Potente editor de código.
2. Análisis estático.
3. Soporte para múltiples lenguajes.

4. Depuración avanzada
5. Gestión de proyectos
6. Pruebas integradas
7. Integración con control de versiones
8. Extensibilidad
9. Herramientas específicas de desarrollo
10. Interfaz de usuario amigable

RESUMEN DEL CAPÍTULO 2

Este capítulo se centra en proporcionar una guía para el desarrollo de programas en el lenguaje de programación Java, haciendo énfasis en alas mejores prácticas y una metodología efectiva.

- **Metodología de Desarrollo:** Se presentan las etapas típicas de desarrollo de software, como el análisis, diseño, implementación, pruebas y mantenimiento del software. Se enfatiza la importancia de seguir un proceso estructurado.
- **Programación estructurada:** La programación estructurada es un paradigma de programación que se basa en la organización lógica y ordenada del código fuente.
- **Programación Orientada a Objetos:** La programación Orientada a Objetos POO es un paradigma de programación que se basa en el concepto de la creación de objetos y la interacción entre ellos.
- **Kit de desarrollo JDK 20:** JDK suele introducir mejoras en el lenguaje de Java. Estas mejoras pueden incluir nuevas palabras claves, funcionalidades de lenguaje, mejoras en la sintaxis, etc.
- **Entorno de desarrollo:** Se describe que tipos de entrono de desarrollo utilizar para la programación en Java, dando una clara perspectiva que tipo de IDE utilizar para el desarrollo de software.

CAPITULO 3

ELEMENTOS BÁSICOS EN JAVA

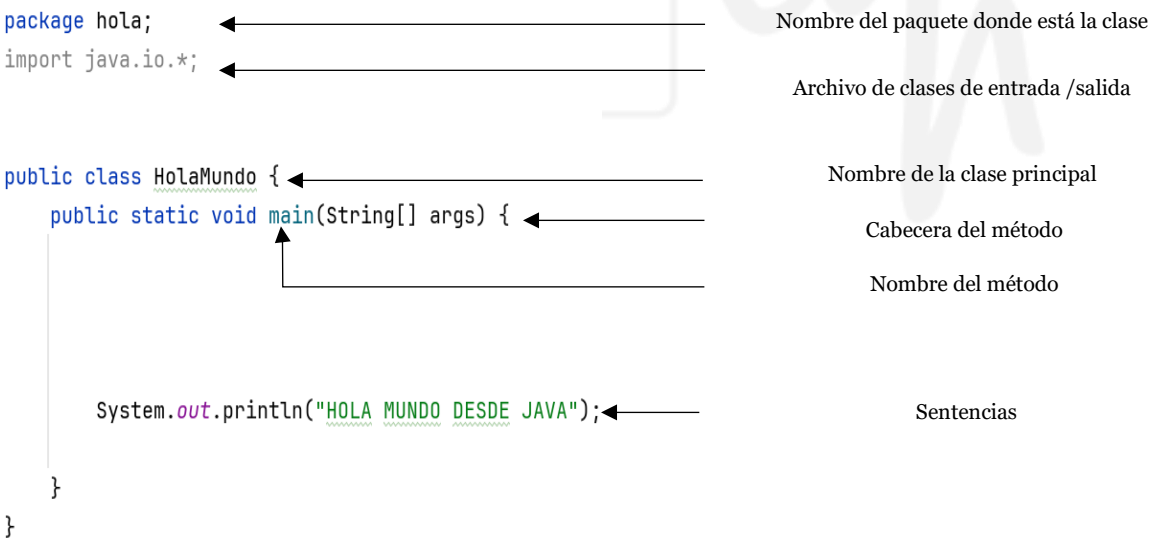
3.1. ESTRUCTURA GENERAL DE UN PROGRAMA EN JAVA

En este capítulo vamos hablar sobre la estructura y los elementos de un programa en Java, describiendo ideas relativas a su estructura; cada programa en Java se compone de una o más clases obligatoriamente un programa debe tener una clase **main**, esta clase debe tener un método main(). Por otro lado, un programa en Java debe tener un conjunto de declaraciones **import** esto me permite importar bibliotecas o archivos de Java. De modo que un programa en Java debe incluir lo siguiente.

- Declaraciones para importar paquetes.
- Declaraciones de clases.
- El método main().
- Métodos definidos por el programador dentro de la clase.
- Comentarios del programa (se utilizan en todo el programa).

La estructura completa de un programa en Java se muestra en la en la figura 15 que a continuación se detalla con un ejemplo sencillo de Java.

Figura 15.
Código sencillo en Java



Nota: El gráfico representa el código de un programa sencillo en Java explicando la estructura y a que representa cada una de las líneas de código. Captura de pantalla

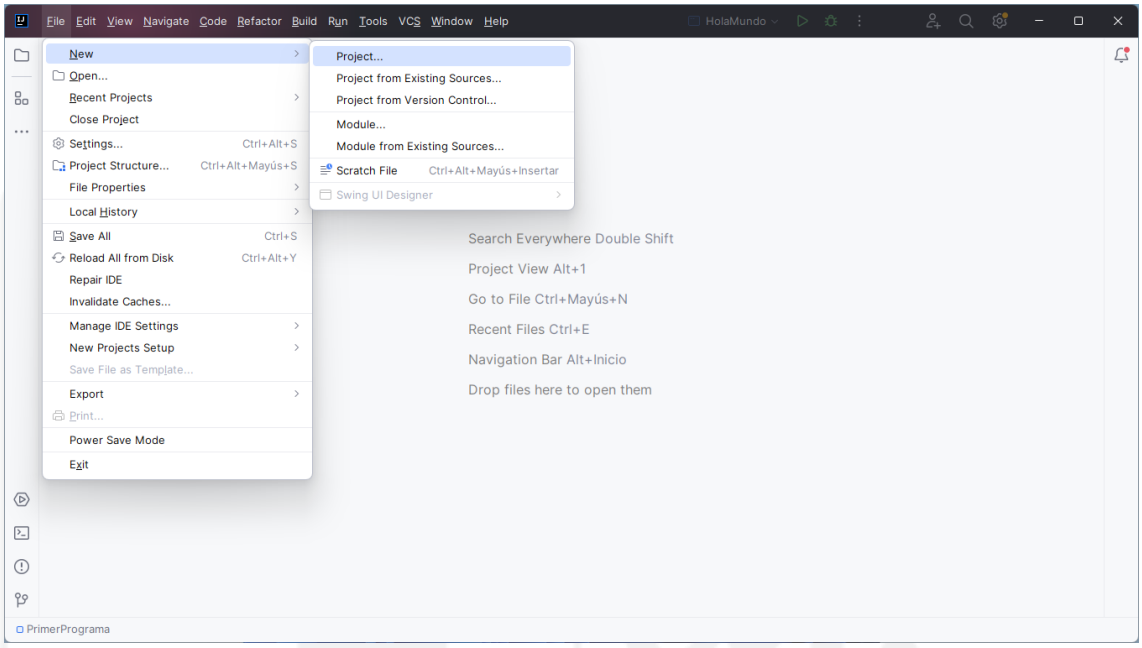
3.1.1. CREANDO MI PRIMER PROGRAMA EN INTELLIJ IDEA

Para la parte práctica del libro utilizaremos el entorno de desarrollo integrado INTELLIJ IDEA la versión Community Edition que se lo descarga de la página oficial, https://www.jetbrains.com/es-es/idea/download/?section=windows.

1. Abrimos el software instalado, presionamos en File, new Project.

Figura 16.

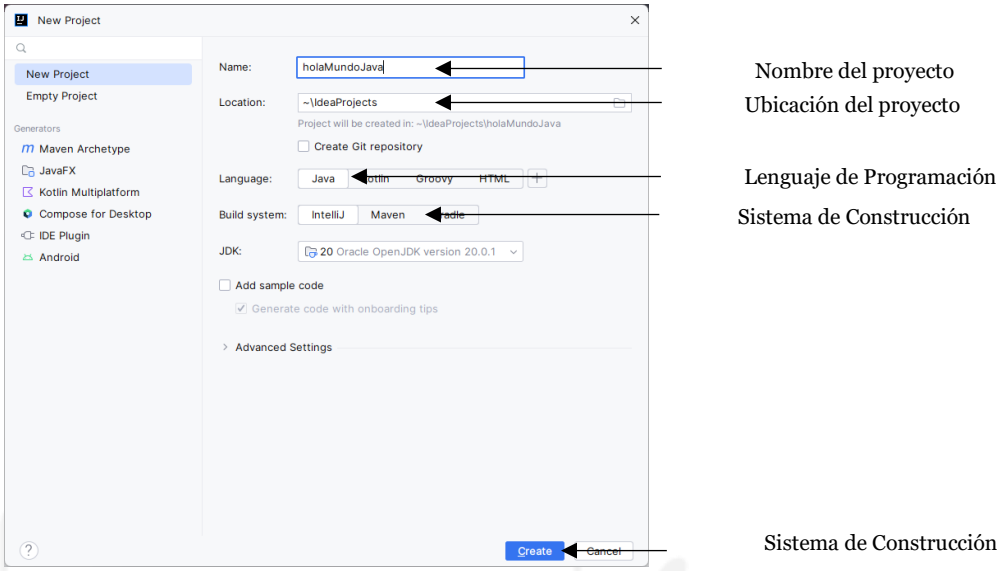
Creando programa en IntelliJ IDEA



Nota: El gráfico representa como se crea un proyecto en Java con IntelliJ IDEA. Captura de pantalla.

2. Colocamos el nombre del proyecto y la ubicación donde se va a guardar el proyecto para el ejemplo holaMundoJava.

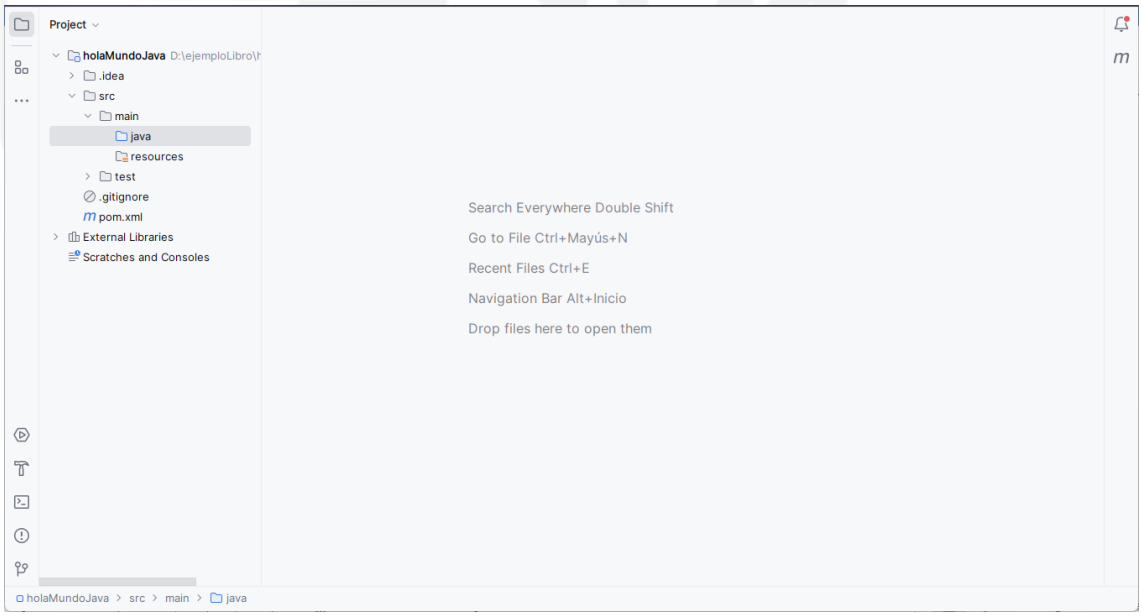
Figura 17.
Creando Primer Programa



Nota: La Figura representa como crear mi primer proyecto en IntelliJ IDEA con lenguaje de programación en Java. Captura de pantalla.

3. Al Crear el proyecto, nos va a crear la estructura de nuestra aplicación de la siguiente manera.

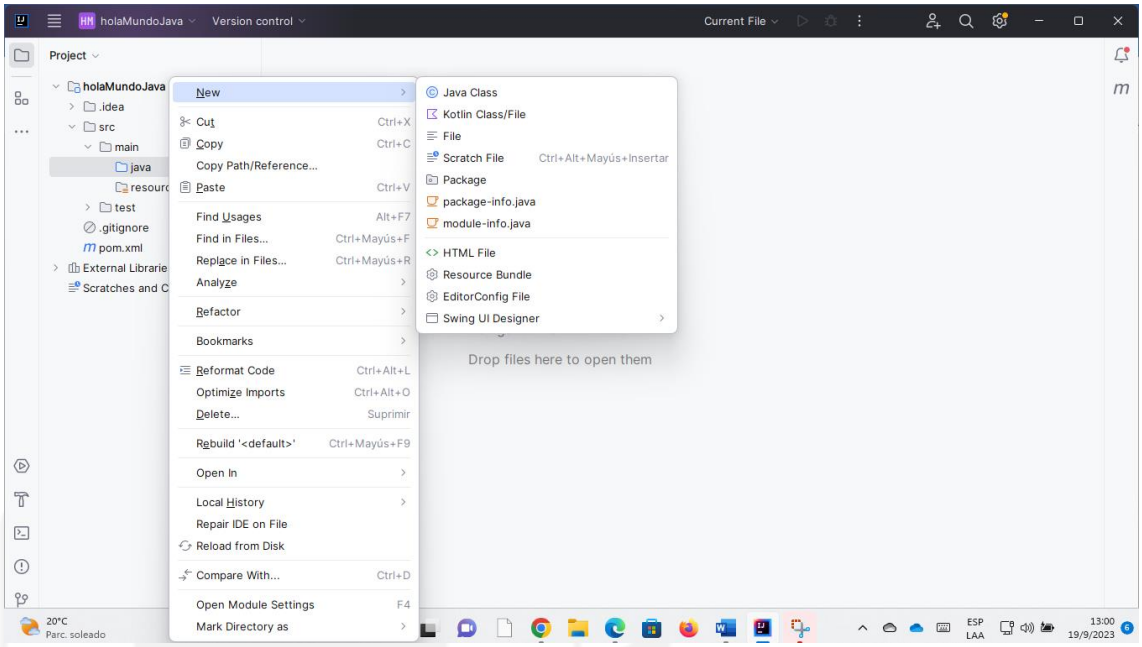
Figura 18.
Estructura de un Programa en Java



Nota: La figura corresponde a la estructura de nuestro programa. Captura de pantalla

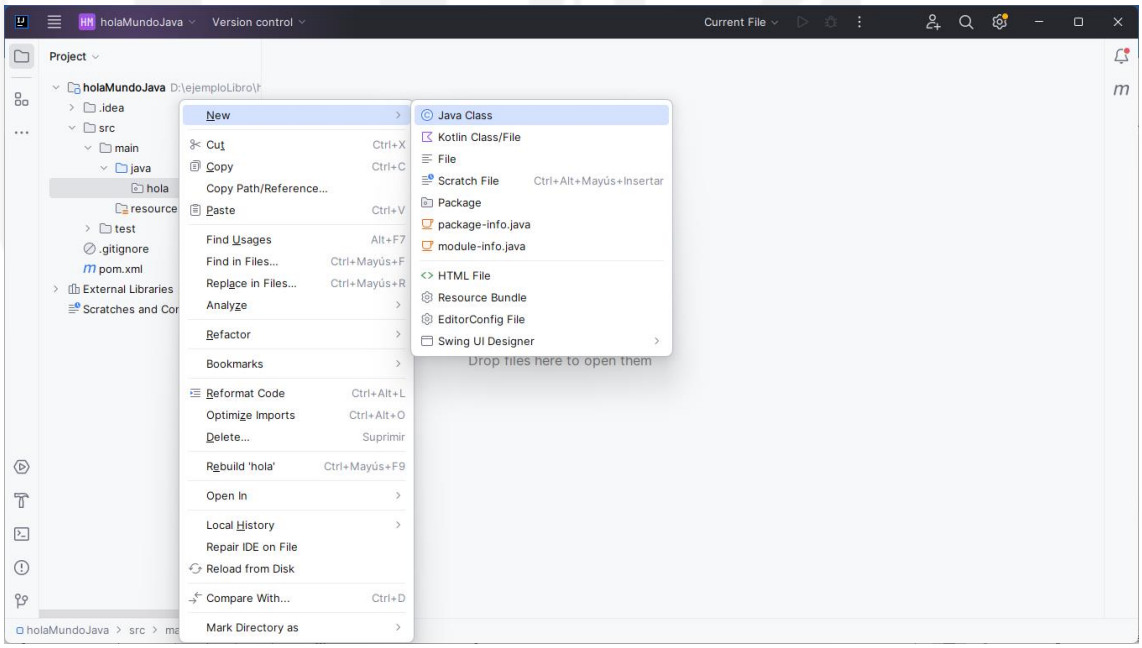
Dentro de la carpeta src, se encuentra la carpeta java ahí vamos a crear un paquete y una clase main().

Figura 19.
Creación de un paquete



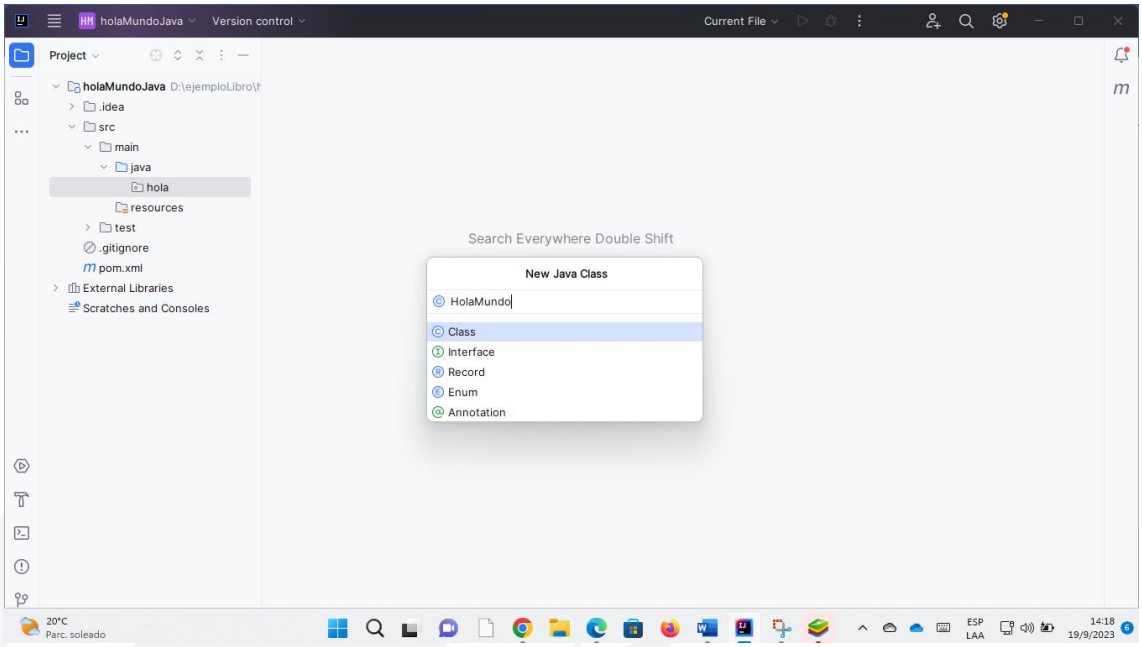
Nota: La figura representa la creación de un paquete. Captura de pantalla.

Figura 20.
Creación clase Main



Nota: La figura representa como crear una clase. Captura de pantalla.

Figura 21.
Clase Hola Mundo



Nota: La presente figura se crea una clase de nombre HolaMundo. Captura de pantalla.

Una de las características al crear una clase en Java, esta se la debe nombrar con una letra mayúscula como se mira en la figura 19. No es una regla de Java, pero para los programadores es muy importante que las clases de nombren con una letra mayúscula al principio si el nombre de la clase contiene dos palabras cada palabra empieza con una letra mayúscula, tampoco debe llevar signos de puntuación.

La declaración del import que tenemos en la primera línea me permite importar librerías o paquetes de Java en este caso java.io proporciona entrada y salida del sistema a través de flujos de datos, serializados y el sistema de archivos[7] como se puede observar el (*) en la importación eso quiere decir que importamos todos los elementos que contiene el paquete java.io.

Cuando hablamos de comentarios en los lenguajes de programación, nos referimos a líneas en nuestro código que no se van a leer y no van a tener ningún cambio nuestro programa, estas líneas sirven para explicar lo que hace esa sección de código como podemos ver en la figura21.

[7] La siguiente fuente tiene como referencia sobre paquetes en Java
https://docs.oracle.com/javase/8/docs/api/java/io/package-summary.html

Figura 22.
Comentarios en Java

```
package hola;
import java.io.*;
/*Cuando escribimos un comentario en Java utilizamos
* dos // para comentar una sola linea
* pero para comentar una seccion utilizamos los signos  que
* pueden observar*/

public class HolaMundo {
    //aqui tenemos nuestro método main que va a permitir ejecutar mi programa
    public static void main(String[] args) {
        System.out.println("HOLA MUNDO DESDE JAVA");

    }
}
```

Nota: Como podemos ver en la figura en Java se puede representar los comentarios de dos formas una sección de código o comentarios de una sola línea. Captura de pantalla.

Un programa en Java se puede considerar un conjunto de clases, en la que al menos en una de esas clases debe tener un método main(), este método me permite ejecutar mi programa sin este método no tendríamos nada en salida de nuestra aplicación.

Figura 23.
Método main()

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


}
```

Nota: La figura me muestra el código del método main() y dentro de las llaves va todo lo que se ejecuta.

3.1.2. Import

En Java, todas las clases se agrupan en paquetes o packages que definen utilidades o grupos temáticos y que se encuentran en directorios del disco con su mismo nombre. Para incorporar y utilizar las clases de un paquete en un programa se utiliza la declaración ***import***: por ejemplo, si le queremos indicar al compilador que queremos importar la clase Math del paquete lang se debe escribir:

Figura 24
Sentencia import

```
import java.lang.Math;
```

Nota: La figura representa como se debe importar una clase de Java de un paquete que contiene dicha clase. Captura de pantalla.

La sintaxis general de la declaración import es:

```
import nombrePaquete.nombreClase;
```

nombreClase es el identificador de una clase del paquete; en un programa se puede incorporar varias clases con una secuencia de sentencias import; por ejemplo, para importar la clase BufferedREader, IOException, InputStreamReader del paquete java.io se debe escribir:

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

Se puede especificar que se incorporen todas las clases publicas de un paquete, en este caso el nombreClase se sustituye por el *, de esta manera se puede utilizar todas las clases públicas del paquete en el programa.

```
import java.io.*;
```

EL programador puede definir sus propios paquetes y dentro podemos agrupar las clases que el programador desarrolle, de igual manera si en alguna parte del programa se necesita de esa clase simplemente se utiliza la sentencia import nombrePaquete.nombreClase. Ejemplo si en un proyecto tenemos un paquete de nombre auto y dentro del paquete se tiene un conjunto de clases como Automático, Manual, de esta manera hacemos las declaraciones.

```
import auto.Automatico;
import auto.Manual;
```

Si deseamos llamar a todas las clases del paquete auto lo instanciamos de la siguiente manera:

```
import auto.*;
```

Las declaraciones de las importaciones de clases se lo realizan al inicio de la programación en la cabecera de nuestro programa y de esa manera se pueden utilizar esas clases en todo el programa sin necesidad de volverlas a llamar.

3.1.3. DECLARACIÓN DE CLASES

Como ya se ha dicho un programa en Java debe tener por lo menos una clase, la clase principal donde se incluya el método main(), y otros método y variables que el programador vea necesario; para declarar una clase debe llevar el nombre con una palabra clave el nombre de la clase debe empezar con una letra mayúscula y si el nombre de la clase lleva dos palabras deben estar unidas y cada palabra debe empezar con una

letra mayúscula, generalmente indicando el acceso, seguida por su indicador, la palabra reservada class, su nombre y sus miembro por ejemplo:

```
package suma;

public class Suma {
    public static void main(String[] args) {
        int inicio, fin, suma;
        suma=0;
        inicio= 1;
        fin=100;
        for (int i = inicio; i < fin; i++) {
            System.out.print(i + " + " + suma + " = " );
             suma = suma + i;
            System.out.println(suma);
        }
    }
}
```

El archivo donde se va a guardar el programa del ejemplo anterior debe tener como nombre Suma.java, el nombre del archivo fuente siempre debe ser el mismo de la clase principal, es decir, la que contiene main(), y la extensión java.

3.1.4. MÉTODO (main)

El método main() es el punto de entrada esencial para la ejecución de programas en Java. En esencia, es donde inicia la acción de un programa en Java. Como habíamos hablado anteriormente en un programa desarrollado en Java, por lo menos debe haber una clase que tenga una método main().

```
public static void main(String[] args) {
    ...  bloque de sentencias
}
```

El método main() se declara como público *public,* lo que significa que es accesible desde cualquier parte del programa, se lo debe declarar como estático *static,* lo que implica que no necesitas crear una instancia de la clase que contiene el método *main* para utilizarlo. Puedes invocarlo directamente en la clase.

El método *main()* tiene un tipo de retorno *void*, lo que indica que no devuelve ningun valor. En lugar de retornar algo, su propósito es iniciar la ejecución del programa, el método toma un parámetro especial llamado *args,* que es un arreglo (array) de cadena de texto. Este parámetro permite que el programa reciba argumentos desde la línea de comandos, lo que facilita la interacción con el usuario o la transmisión de información adicional al programa.

El cuerpo del método *main se* define entre llaves *{}* y es donde se coloca las instrucciones que componen el funcionamiento del programa.

3.1.5. MÉTODOS DEFINIDOS POR EL USUARIO

Los métodos definidos por el usuario en Java son funciones personalizadas creadas por el programador dentro de un programa Java para realizar tareas específicas. Estos métodos son una manera de organizar y reutilizar el código al encapsular un conjunto de instrucciones bajo un nombre único. Pueden aceptar parámetros como entrada de datos y devolver resultados, lo que permite una modularidad y claridad efectivas en la programación. Estos métodos son esenciales para dividir un programa en partes más pequeñas y manejables, lo que facilita su comprensión y mantenimiento.

Los métodos definidos por el usuario se invocan, dentro de la clase donde se definieron, por su nombre y los parámetros opcionales que pudieran tener, después de que el método se invoca, el código asociado se ejecuta, y a continuación se retorna el método llamador. Si la llamada es desde un objeto de la clase, se invoca el método precedido del objeto y el selector punto (.). En capítulos más adelante se verá a fondo como instanciar y llamar a un método de una clase en Java, mientras tanto a título de ejemplo tomando el ejemplo anterior vamos a crear un método de nombre suma y lo llamamos en nuestro método *main()*.

```
package suma;
public class Suma {
    public static void main(String[] args) {
        //Instanciamos el objeto Suma
        Suma op= new Suma();
        /*
        * LLamamos al método suma para poder
        * ejecutar la operación de sumar los números
        * del 1 al 100
        * */
        op.operacion();
    }

    //método donde estamos realizando la operación
    public void operacion(){
        int inicio, fin, suma;
        suma=0;
        inicio= 1;
        fin=100;
        for (int i = inicio; i < fin; i++) {
            System.out.print(i + " + " + suma + " = " );
            suma = suma + i;
            System.out.println(suma);
        }
    }
}
```

Como se ve en el ejemplo el programador ha creado un método llamado operación(), este método no devuelve nada ya que utiliza la palabra reservada *void* dentro del método declaramos tres variables de tipo entero, inicio, fin y suma; cada una de estas están

inicializada después generamos un bucle *for* que inicia en 1 y termina en 100 y se va recorriendo 1 a 1, dentro del for vamos a sumar cada uno de los números del 1 al 100.

Los métodos en Java se especifican en la clase a la cual pertenecen, la estructura de un método es la siguiente:

```
tipo_retorno nombreMetodo(lista_de_parametros){  principio del método
     sentencias                    cuerpo del método
    return expresión              valor que devuelve el método
}                                 fin del método
tipo_retorno                      Es el tipo de valor o void devuelto por la función
nombreMetodo                      Nombre del método
lista_de_parametros              Lista de parámetros, o void pasados    al método
                                  se los conoce también como argumentos
```

Ejemplo 3.1

Realizar un programa en Java formado por una clase que contenga dos métodos además del *main(): calcula(), mostrar(),* el primer método determina el valor de una función matemática para un valor dado de x, el segundo método muestra el resultado, *el método main()* llama a cada uno de los métodos auxiliares, los métodos deben ser estáticos por las restricciones de *main()*, la función evalúa utilizando el coseno de la clase Math, que esta se encuentra en el paquete java.lang.

```
package evaluar;

public class Evaluar {
    public static void main(String[] args) {
        double f;
        f=calcular();
        mostrar(f);

    }
    public static double calcular(){
        double x=3.141516/4;
        return x*Math.cos(x) + 0.5;
    }
    public static void  mostrar(double r){
        System.out.println(" Valor de la función r = " + r);
    }

}
```

3.1.6. COMENTARIOS

Como ya hemos explicado anteriormente, los comentarios es información que no va a leer el programa, los comentarios son líneas en nuestro código donde explicamos que hace la sección de código, estas líneas son ignorados por le compilador y no realiza ningún proceso.

En programación la implementación de comentarios es una buena practica al momento de programar, como desarrolladores debemos comentar el código tanto como sea posible, de esta manera usted mismo y otros programadores pueden leer fácilmente lo

que se esta desarrollando. En Java se suele comentar los programas al inicio de cada archivo fuente, estos comentarios al inicio se detalla que hace el programa, quien es la persona que la desarrollo y la fecha que fue creado el programa e información de revisión.

En Java los comentarios de un programa se pueden introducir de dos formas:

- Con los siguientes caracteres */*....cometarios....*/* para insertar más de una línea
- Con la secuencia de dos backslash o barras (//) me sirve sola para incorporar el comentario en una sola línea.

Comentarios con /* */

Los comentarios comienzan y terminan con la secuencia /* */; el texto va entre los caracteres y todo lo que esta dentro es ignorado por el compilador

/*Saludo.java Primer programa en java*/

Si se necita introducir varias líneas de comentarios de programas se puede hacer de la siguiente manera.

```
/*
* Programa: Evaluar.java
* Programador: Elvis Pachacama
* Descripción: Primer programa en Java
* Fecha de creación: 25 septiembre del 2023
* Revisión: Ninguna
* */
```

También los comentarios lo podemos introducir de la siguiente manera.

```
System.out.println(" Valor de la función r = " + r);/*Imprime el valor de la función*/
```

Cometario en una línea con //

Con este tipo de comentarios de igual manera el compilador no lo va a ejecutar y se inserta en una sola línea.

```
//Evaluar.java
```

Ejemplo 3.2

Crear un programa en Java donde se guarde un mensaje en una variable de tipo String y lo imprima en la pantalla, el programa.

```
package mensaje;
/*
* Programa: Mensaje.java
* Programador: Ing. Elvis Pachacama
* Descripción: Escribir un programa donde se guarde una mensaje en una variable de tipo String
*              imprimir el valor de la variable en pantalla.
* Fecha de creación: 25 Septiembre del 2023
* Revisión: ninguna
```

```
* */
public class Mensaje {
    public static void main(String[] args) {
        String mensaje = "Escribiendo un programa en Java...";
        System.out.println(mensaje);
    }
}
```

3.2. ELEMENTOS DE UN PROGRAMA EN JAVA

Todos los programas escritos en Java constan de un archivo donde se encuentran las clases y métodos que escribe el programador, además de otros archivos que se encuentran incorporadas en otras clases, el compilador traduce todos programas y además incorporan todas las clases solicitadas al programa y se encarga de analizar la secuencia de los tokens.

3.2.1. ELEMENTOS LEXICOS DEL PROGRAMA

Cuando se programa en Java encontraremos 5 clases de tokens como son: identificadores, palabras reservadas, literales, operadores y otros separadores, los cuales vamos air identificando con el pasar del libro.

3.2.1.1. IDENTIFICADORES

Los identificadores es un conjunto de caracteres que se utilizan para representar las variables, métodos, clases y otros elementos del código fuente. Los identificadores siguen ciertas reglas y convenciones que deben cumplirse.

- **Caracteres permitidos.** Los identificadores pueden contener letras mayúsculas, minúsculas, dígitos y el carácter especial "_" (guion bajo).
- **No pueden empezar con un número.** Un identificador no puede empezar con un número, pero si puede contener un número después del primer carácter.
- **Sensible a mayúsculas y minúsculas.** Java distingue entre mayúsculas y minúsculas en los identificadores, ejemplo **"miVariable"** y **"MiVariable",** como se ve en el ejemplo el nombre es el mismo, pero en Java son distintos identificadores.
- **No se pueden usar palabras claves.** No puedes utilizar palabras claves reservadas de Java como identificadores. Ejemplo **"public"**, **"class", "int", "if",** entre otras clases.

Cuando se usa identificadores estos pueden ser creados de cualquier longitud, Java no te limita en el número de caracteres que puede contener. Consejos que te pueden servir de reglas para escribir un identificador.

1. Identificadores de variables con minúsculas.

2. Constantes con mayúsculas.
3. Métodos en minúsculas.
4. Clases con el primer carácter en mayúsculas.

3.2.1.2. PALABRAS RESERVADAS

Cuando se programa en Java hay palabras reservadas, las cuales no puedes ser usadas para nombrar una clase, método o variable, esta es una característica de Java ya que estas palabras se asocian con algún significado especial ejemplo.

```
//error no se puede nombrear un método con una palabra reservada
void void(){   //error
    //error no se puede nombrar una variable con una palabra reservada
    int char;  //error
}
```

En Java actualmente existen 53 palabras[8] reservadas.

Tabla 1.
Palabras reservadas en Java

abstract	**continue**	**for**	**new**	**switch**
assert	**default**	**goto**	**package**	**synchronized**
boolean	**do**	**if**	**private**	**this**
break	**double**	**implements**	**protected**	**throw**
byte	**else**	**import**	**public**	**thows**
case	**enum**	**instanceof**	**return**	**transient**
catch	**extends**	**int**	**short**	**try**
char	**final**	**interface**	**static**	**void**
class	**finally**	**long**	**strictfp**	**volatile**
const	**float**	**native**	**super**	**while**
true	**false**	**null**		

[8] La siguiente fuente tiene como referencia las palabras reservadas en Java las que no se pueden usar en un programa para nombrar un método, clase, etc.
https://codigofacilito.com/articulos/palabras_reservadas_java

Dos de las palabras que tenemos en la tabla anterior hoy en día ya no son utilizadas en Java; *const y goto,* estas palabras son reservadas del lenguaje de programación C++, estas palabras se mantuvieron para generar mejores mensajes de error.

3.2.2. PAQUETES

Como ya lo habíamos dicho Java agrupa las clases e interfaces que tienen cierta relación mediante archivos especiales las cuales contienes declaraciones de clases con sus métodos, estas agrupaciones lógicas de clases y otros tipos relacionados que nos permiten organizar y estructurar un programa en Java de manera más eficiente. Estos paquetes ayudan a evitar conflictos de nombres y hacen que el código se más modular y fácil de mantener. Cada paquete en Java actúa como un contenedor que contiene clases y otros recursos relacionados. Los paquetes se organizan en una jerarquía, y el nombre completo de una clase incluye la ruta del paquete en la que se encuentra.

Java también proporciona paquetes ya desarrollados, y los más importantes son:

Java.lang, java.applet, java.awt, java.io y java.util.

java.lang, es un conjunto de clases y recursos esenciales que proporcionan funcionalidades, básicas y fundamentales para la programación en Java. Estas clases incluyen objetos fundamentales como ***String, Integer, Boolean, Math.***

Java.io, es un conjunto de clases y utilidades que proporciona funcionalidades para la entrada y salida de datos (E/S). Ofrece clases como: ***FileInputStream, FileOutputStream, BufferesReader, BufferedWriter.***

Java.util, es un conjunto amplio y diverso de clases y utilidades que proporciona funcionalidades adicionales y utilidades de propósito general para la programación en Java. Contiene clases para estructuras de datos como listas, conjuntos, mapas, así como clases para manejar fechas, tiempos, y muchas otras utilidades útiles para tareas comunes de programación.

Java.awt, que significa "Abstract Window Toolkit" en inglés, es un conjunto de clases y utilidades en Java diseñado para facilitar la creación de interfaces gráficas de usuario (GUI). Este paquete proporciona una variedad de componentes gráficos, como botones, ventanas, cuadros de texto y otros elementos, que permiten a los desarrolladores construir aplicaciones Java con interfaces de usuario interactivas y visualmente atractivas.

En esencia, java.awt es una parte esencial de la biblioteca estándar de Java que facilita la creación de aplicaciones con GUI, ayudando a los programadores a diseñar interfaces de

usuario intuitivas y portables que pueden ejecutarse en diferentes sistemas operativos sin cambios significativos en el código fuente.

No olvidemos que el programador puede crear sus propios paquetes y después puede utilizarlo en cualquier parte de la aplicación que desee. Ejemplo tenemos una clase *Auto* y se va almacenar en el paquete *vehículo*.

```
package vehiculo;

public class Auto {
    double precio;
    public static void conducirVehiculo(){

    }

}
```

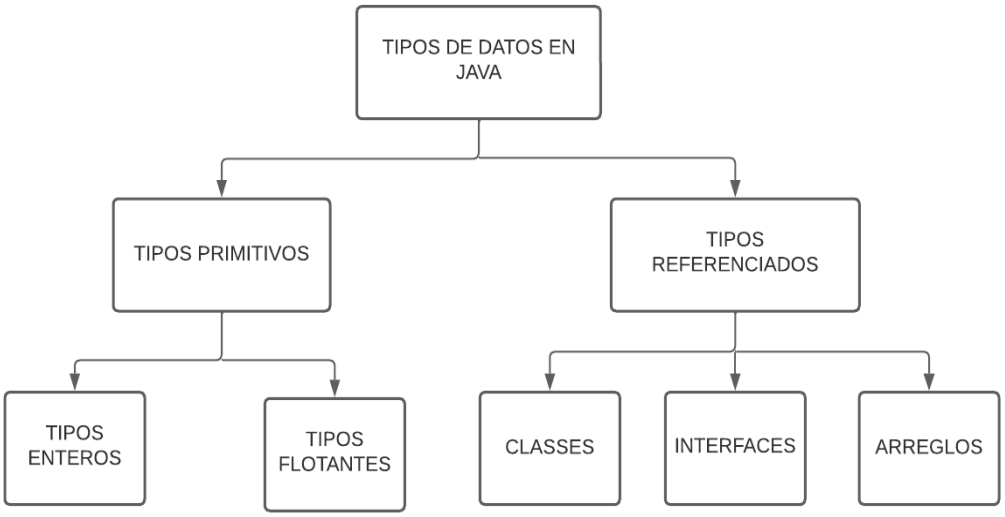
Ya creado el directorio, Java generar un subdirectorio con el mismo nombre del paquete *vehiculo,* dentro de este paquete tenemos la clase *Auto,* el programador puede utilizar la clase creada del paquete *vehiculo,* anteponiendo en el nombre del paquete al nombre de la clase. El compilador lo que hace es buscar el archivo de la clase en el paquete: *vehiculo.Auto.conducirVehiculo();*

3.3. TIPOS DE DATOS EN JAVA

Los tipos de datos en Java se refieren a las categorías de valores de valores que las variables pueden contener. Estos tipos definen qué tipo de información puede almacenarse en una variable y cómo se almacena y procesa. Los tipos de datos en Java pueden dividirse en dos categorías principales: tipos de datos primitivos y tipos de datos de referencia.

Existe una clasificación amplia respecto a los tipos de que se manejan en Java, sin embargo, podemos resumirla como se muestra en la figura 25.

Figura 25.
Tipos de datos en Java

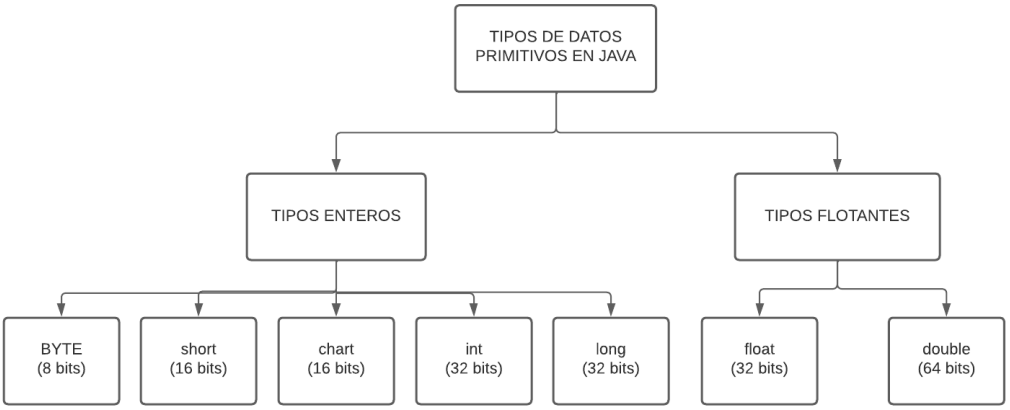


Nota: La imagen representa a los tipos de datos que se maneja en Java.

Por un lado, tenemos los tipos de datos primitivos y por otro lado tenemos los tipos que se consideran como extensiones de la clase Object, también conocidos como referencia de objetos.

Los tipos de datos primitivos se dividen en tipos de datos enteros y flotantes y estos se dividen en otros tipos de datos que lo mostraremos en la figura 26.

Figura 26.
Tipos de Datos primitivos en Java



Nota: La figura representa a los datos primitivo en Java

Como se muestra en la figura anterior por un lado tenemos los tipos de datos enteros que se divide:

Tabla 2.
Tipos de Datos en Java

TIPO DATOS	TAMAÑO
Byte	8 bits
Short	16 bits
Chart	16 bits
Int	32 bits
Long	64 bits

Nota: En la tabla se muestra los tipos de datos primitivos enteros con sus respectivos tamaños en bits.

Por otro lado, también tenemos los datos de tipo flotante que se divide en:

Tabla 3.
Tipos de datos Flotantes

TIPO DE DATO	TAMAÑO
Float	64 bits
Double	64 bits

Nota: En la tabla se muestra los tipos de datos primitivos flotantes con sus tamaños en bits

Estos tipos de datos en Java son los más básicos y son los que utilizaremos si necesitamos la mayor rapidez y ahorro en recurso, con el objetico de que nuestro programa sea más eficiente. Sin embargo, en la práctica también utilizaremos también funciones ya creadas que pueden hacer uso de manera indirecta de estos tipos de datos primitivos.

3.3.1 TIPOS DE DATOS ENTEROS.

Los tipos de datos enteros en Java son categorías que representan valores numéricos enteros, es decir, números sin parte decimal. Estos tipos permiten almacenar y manipular números enteros positivos y negativos en programas Java. Los tipos de datos enteros se utilizan para variables que necesitan contener números enteros, como recuentos, índices, edades y otros valores que no requieren fracciones o decimales. Algunos ejemplos de tipos de datos enteros en Java incluyen int, byte, short y long, cada uno de los cuales tiene un rango de valores que puede representar, lo que afecta a la cantidad de memoria que ocupan y su precisión en la representación de números

enteros. Estos tipos de datos enteros son fundamentales en programación y se utilizan ampliamente en una variedad de aplicaciones Java.

3.3.2. DECLARACIONES DE VARIABLES

Una de las formas más simples de declarar variables en Java es primero se escribe el tipo de variable seguido por el nombre de la variable, si se desea asignar un valor inicial a la variable, y el formato de declaración es el siguiente.

```
<tipo de dato> <nombre de variable = <valor inicial>
```

También en Java se puede declarar múltiples variables en la misma línea siempre y cuando sean del mismo tipo de variable.

```
<tipo de dato> <nom_var1>,<nom_var2>...<nom_varn>
```

Ejemplo:

```
int valor;
int valor1=99;
int valor3, valor2;
int num_parte = 1233, num_items=23;
long num4, num5=991144222;
```

3.3.3. TIPOS DE COMA FLOTANTE

Los tipos de datos de coma flotante o punto flotante como lo llaman representan números reales que contiene una coma o punto decimal, tal como 3.14159, o número grandes como $1.34 * 10^{15}$. La declaración de las variables de tipo flotante es igual a la declaración de las variables de tipo entero.

```
float valor;         //declara una variable real
float valor1=99.99f; //asigna 99.99 a la variable valor1
float valor3, valor2; //declara varios valores de coma flotante
double prod;
```

Java soporta dos formatos de coma flotante: float, esta tipo de datos requiere 4 bytes de memoria mientras que el tipo double requiere 8 bytes.

3.3.4. CARACTERES

Un carácter en Java se refiere a un tipo de dato que representa un solo símbolo o letra de un conjunto de caracteres. Este tipo de dato se utiliza para almacenar caracteres individuales, como letras, números, signos de puntuación y otros símbolos, en un programa Java. Los caracteres se representan mediante el tipo de dato *char* y pueden utilizarse para procesar y manipular texto en una aplicación Java.

3.3.5. BOOLEAN

Un tipo de dato booleano en Java se refiere a una categoría que representa dos valores posibles: verdadero o falso. Este tipo de dato se utiliza para realizar evaluaciones lógicas y controlar el flujo de un programa. En Java, el tipo de dato booleano se declara con la

palabra clave boolean y se utiliza para almacenar resultados de comparaciones o expresiones lógicas, como condiciones en sentencias if, while, for, entre otras. Los valores booleanos, verdadero o falso, son fundamentales para la toma de decisiones en la programación, ya que permiten controlar el comportamiento del programa en función de condiciones específicas.

```
boolean dulce;
dulce=true;
boolean encontrar, bandera;
```

3.4. VARIABLES

Una variable en Java es un contenedor que se utiliza para almacenar y manipular datos en un programa. Cada variable tiene un nombre único y un tipo de dato que especifica qué tipo de información puede almacenar, como números, texto o valores lógicos. Las variables permiten a los programadores guardar y gestionar información en la memoria del programa, lo que facilita la creación de aplicaciones dinámicas y flexibles, por ejemplo.

```
char valor;
```

el ejemplo significa que el programa va a reservar un espacio de memoria para *valor*, en este caso el espacio de memoria reservado será de dos bytes. El identificador de la variable deber ser valido, y debe representar a lo que se está programando, por ejemplo:

```
sueldo      sueldo_diario      edad_empleado   $sueldo
```

3.4.1. DECLARACIÓN

Cuando se declara una variable se necita una sentencia que proporciona información de esta al compilador, la sintaxis que se utiliza es la siguiente:

```
tipo variable
```

tipo es el nombre del tipo de dato que se va a utilizar.
variable, es el identificador o nombre valido den Java.

Ejemplo:

```
int numero;
double sueldo_diario,
        horas_trabajadas,
        sueldo_semanal;
short diaSemana;
```

Antes de utilizar las variables es preciso declararlas, las declaraciones se les puede hacer en tres lugares del programa:

- En una clase, como miembro de esta.
- Al principio de un método o un bloque de código.
- En el lugar donde se va a utilizar.

3.4.1.1 EN UNA CLASE

Las variables se declaran como un miembro de una clase esto quiere decir que estas variables se pueden utilizar en cualquier parte del programa.

```
public class Auto {
    double velocidad;
    void entrada(){
        velocidad=56.56;
    }
    void salida(){
        System.out.println("La velocidad el auto es: " + velocidad);
    }

}
```

En esta clase, la variable *velocidad* puede utilizarse en cualquier método, como en el ejemplo lo podemos utilizar en el método *entrada()* y *salida()*.

3.4.1.2. AL PRINCIPIO DE UN BLOQUE DE CÓDIGO

Las variables también se pueden declarar en un método o un bloque de código dentro de un método.

```
double velocidad(int n){
    int r;
    double c;
    c=1L;
    for (r = 1; r < n ; r++) {
        double aux=20;

    }

    return c;
}
```

En el ejemplo, las variables *r* y *c* están definidas dentro del método *velocidad* y estas variables son locales esto quiere decir que se pueden utilizar solo dentro del método, la variable *aux* esta declara dentro del ciclo *for* por la cual esta solo se le puede utilizar dentro de este.

3.4.1.3. EN EL PUNTO DE UTILIZACIÓN

Java es un lenguaje de programación que proporciona una gran flexibilidad al momento de declarar variables, ya que se puede declarar variables donde se les va a utilizar, este tipo de declaraciones se utiliza mucho cuando se utiliza bucles.

```
for (int i = 0; i < 10; i++) {
    //...
}
```

En la sección del código se declara una variable dentro del ciclo *for* en la cual hace un recorrido desde el 0 hasta el 10 y se va aumentando de uno en uno.

3.4.1.4. INICIALIZACIÓN DE VARIABLES

Al momento de declarar una variable esta no tiene valor lo que podemos hacer es darle un valor inicial cuya sintaxis es la siguiente.

```
tipo nombre_variable= expresion;
```

expresión, es cualquier declaración valida cuyo valor es el mismo modelo del *tipo,* por ejemplo.

```
char genero = 'F';
int acumulador= 0;
int anio=2023;
```

En el ejemplo anterior vemos que al crear las variables se les da un valor, que se guardan en la memoria, una variable inicializada o no pueden cambiar de valor utilizando sentencias de asignación como, por ejemplo:

```
int anio;
anio=2024;
```

3.5. DURACIÓN DE UNA VARIABLE

Dependiendo donde se usa las variables de Java, estas se pueden utilizar en todo el programa, dentro de un método o dentro de un bloque de código de manera temporal, el lugar del programa donde se activa la variable se llama alcance o (scope), por lo general este alcance se extiende desde donde se define la variable hasta el final del bloque, de acuerdo con esto existen dos tipos de alcen que son:

- Variables Locales
- Variables de clase

3.5.1. VARIABLES LOCALES

Estas variables son definidas dentro de un método, estas solo están disponible o visibles dentro de esa sección del programa o método especifico, para este tipo de variables rige algunas reglas que son.

1. En el interior del método no se pueden modificar por ninguna sentencia externa a aquel.
2. Sus nombres no son únicos, eso quiere decir que dos o más métodos pueden declarar variables con el mismo nombre, y cada una de ellas son distintas y solo pertenecen a ese método.
3. Las variables no existen en memoria hasta que se ejecute el método donde están declaradas, de esta forma me permite ahorrar memoria ya que deja que varios

métodos compartan la misma memoria para sus variables locales, pero no de manera simultánea.

Como ya hemos mencionado por esta última razón las variables locales también se llaman automáticas ya que se crean de manera predeterminada al inicio del método y termina cuando finaliza la ejecución del método.

```
public static void sumar(){
    double n1,n2,n3;
    //declaramos una variable local
    int resultado;
    n1=Math.random()*999;
    n2=Math.random()*999;
    n3=Math.random()*999;
    resultado= (int) ((int)n1+n2+n3);
    System.out.println("La suma de los números aleatorios son " +
resultado);

}
```

3.5.2. VARIABLES DE CLASES

Sabemos que los miembros de una clase son los métodos y variables, cuando se declara un variable fuera de los métodos estas están disponible y visibles para cualquier método, y se lo utiliza o referencia solo con su nombre.

```
public class Auto {

    //declramos variables globales o variables
    //de la clase
    double precio, cilindraje, interes;

    double costo()
    {
        //declaración de variables locales
        double x;
        precio= 25.000;
        return precio;
    }
    //declaración de variable local
    double valor;
    double cilindraje(){
        //inicializamos la variable que solo se puede utilizar en esta
clase
        float x;
        //esta variable es visible porque esta declarado en la clase
        cilindraje=1.8;
        //inicializamos la variable valor que solo se puede
        //visualizar en esta clase
        valor = 0;
        valor = valor+precio;
        return valor;
    }


}
```

3.6. ENTRADA Y SALIDA

La entrada y salida de datos en Java se refiere al proceso de trasferir información hacia y desde un programa de Java. Esto permite que los programas interactúen con los usuarios o con otros sistemas para recibir datos y enviar resultados.

La clase *System* define dos referencias a objetos *static* para la gestión de entrada y salida por consola, estos son:

```
System.in     //para la entrada por teclado
System.out    //para la salida por pantalla
```

En Java *System.in* es una entrada estándar predefinida que representa el flujo de entrada de datos desde el teclado o desde otro dispositivo de entrada. Es una instancia de la clase *InputStream* y se utiliza para recibir información introducida por el usuario durante la ejecución de un programa. Esta entrada estándar es ampliamente utilizada para interactuar con el usuario y recopilar datos en tiempo real en aplicaciones de consola, además hace referencia a un objeto de la clase *BufferedInputStream* en la cual hay diversos métodos para captar caracteres tecleados.

La segunda referencia a un flujo de salida estándar predefinido que representa la forma en que los programas pueden mostrar información y resultados al usuario. Se trata de una instancia de la clase *PrintStream* y cumple un rol fundamental en la comunicación entre el programa y el usuario en aplicaciones de consola.

Este mecanismo permite que los programas emitan mensajes, datos y resultados de cálculos directamente en la pantalla del usuario. Es ampliamente utilizado para mostrar mensajes informativos, resultados de operaciones, mensajes de error o cualquier otro tipo de salida que sea relevante para el usuario.

Cuando se emplea *System.out*, el programa escribe datos en forma de secuencias de caracteres, lo que facilita la presentación de información de manera legible y comprensible para el usuario. Puede utilizarse para imprimir texto, valores numéricos, datos procesados o cualquier información relevante para el flujo de trabajo de la aplicación.

Es importante mencionar que *System.out,* está vinculado a la consola por defecto, pero se puede redirigir hacia otros destinos de salida, como archivos o dispositivos externos, para lograr una mayor flexibilidad en la gestión de la información generada por el programa.

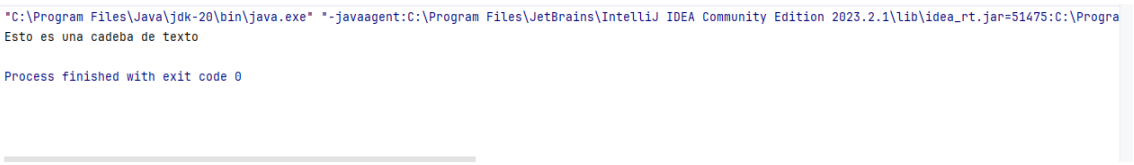
3.6.1. SALIDA(System.out)

Cuando se quiere imprimir un dato en consola Java tiene un objeto *out* que ese encuentra definido en la clase *System,* y permite visualizar los datos en la pantalla de su equipo, por ejemplo:

```
public static void main(String[] args) {
    System.out.println("Esto es una cadeba de texto");
}
```

Figura ***27***

Salida de información en consola



Nota: La imagen representa la salida de pantalla del código donde muestra el mensaje “Eso es una cadena de texto”. Captura de pantalla.

System.out es una referencia al objeto de la clase *PrintStream,* este objeto tiene diferentes métodos los cuales lo podemos utilizar con frecuencia.

```
print()  //Transfiere una cadena de caracteres al buffer de la
pantalla
println() //transfiere uan cadena de caracteres y el caracter de fin
de línea al buffere de pantalla
flush() //el buffer con las cadenas almacenadas se imprime en pantalla
```

3.6.2. ENTRADA

La clase System define un objeto de la clase BufferedInputStream cuya referencia resulta en in. El objeto se asocia al flujo estándar de entrada, que por defecto es el teclado, los elementos básicos de este flujo son caracteres individuales y no cadenas como ocurre con el objeto out; entre los métodos de la clase se encuentra read() que devuelve el carácter actual en el buffer de entrada; por ejemplo:

Para realizar entrada de datos por teclado con la clase BuffereReader se importa la clase

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
```

importada la clase lo que se hace es crear un objeto de esta clase BufferedReader y de la clase *InputStreamReader*.

```
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
```

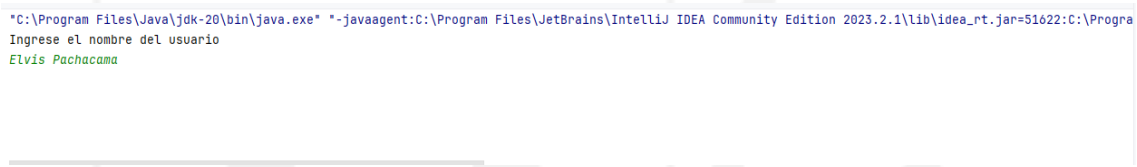
Inicializados los objetos *InputStreamReader* y *BufferedReader* nos permite capturar líneas de caracteres mediante el teclado con el método *readLine()*.

```
package vehiculo;
```

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

public class Auto {

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        String nombre;
        System.out.println("Ingrese el nombre del usuario");
        nombre = br.readLine();
    }

}
```

Como podeos ver en el código anterior además de las dos clases importadas, se debe importar un control de excepciones para el manejo de caracteres *IOException*.

Figura 28.
Salida en consola



Nota: La figura representa la salida de nuestro código en consola donde pide que se ingrese el nombre del usuario por teclado. Captura de pantalla

Cuando se ingresa datos por consola como vemos en el ejemplo del código anterior, no solo se puede ingresar cadena de caracteres, sino Java permite ingresar cualquier tipo de datos, por ejemplo:

Ejemplo 3.3

Crear un programa en Java donde el usuario ingrese su nombre completo y la edad en números enteros.

```
package nombre;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
/*
* Programa: Alumno.java
* Programador: Ing. Elvis Pachacama
* Descripción. Crear un programa donde el usuario ingrese
*              por teclado el nombre del alumno y la edad los tipos de
*              ingresar son nombre tipo String, edad tipo entero
* Fecha de creación: 26 de septiembre del 2023
* Revisón: ninguna*
* */
public class Alumno {
    //método main
    public static void main(String[] args) throws IOException {
        //Inicializamos el objeto de la clase BufferedREader y de la
```

```
clase InputStreamReader
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        //declaramos las variables a utilizar nombre de tipo String y
edad de tipo entero
        String nombre;
        int edad;
        //Imprimimos un mensaje en consola pidiendo al usuriao
ingresar el nombre
        System.out.println("Ingrese el nombre del Alumno:");
        //Ingresamos y guardamos en la variable nombre lo que ingrese
el usuario
        nombre=br.readLine();
        //Imprimimos un menaje en consola pidiendo al usuario ingrese
al edad del Alumno
        System.out.println("Ingrese la edad delLAumno:");
        //Ingresamos y transformamos a entero la edad
        edad = Integer.parseInt(br.readLine());
    }
}
```

Algo para recordar cuando usamos la clase *BufferedReader*, todos los datos que se ingresan lo hacen de tipo *String*, como se muestra en el código anterior, se ingresa por teclado y se guarda en una variable de tipo *int* la eda del alumno, para eso debemos convertir esa cadena de tipo *string* a *entero*, y de esa manera se puede ingresar cualquier tipo de dato.

Tabla 4.
Tipo de conversión de datos

Tipo dato	Conversión
int edad	edad=Integer.*parseInt*(br.readLine());
double sueldo	`sueldo= Double.parseDouble(br.readLine());`
float sueldo	`sueldo= Float.parseFloat(br.readLine());`

Nota: La tabla muestra como convertir un dato en Java cuando se utiliza la clase BufferedReader

3.6.3. ENTRADA CON LA CLASE *Scanner*

La entrada de datos con la clase Scanner en Java es una técnica comúnmente utilizada para obtener información ingresada por el usuario a través del teclado o para leer datos desde otros flujos de entrada, como archivos. La clase Scanner se encuentra en el paquete java.util y proporciona métodos para analizar y procesar diferentes tipos de datos de entrada, como números enteros, números de punto flotante, cadenas de caracteres y

más. Aquí te muestro un ejemplo de cómo usar la clase Scanner para obtener entrada de datos desde el teclado:

```
package nombre;
import java.util.Scanner;
/*
* Programa: Alumno.java
* Programador: Ing. Elvis Pachacama
* Descripción. Crear un programa donde el usuario ingrese
*              por teclado el nombre del alumno y la edad los tipos de
*              ingresar son nombre tipo String, edad tipo entero
* Fecha de creación: 26 de septiembre del 2023
* Revisón: ninguna*
* */
public class Alumno {
    //método main
    public static void main(String[] args) {
        //instanciamos el objeto de la clse Scanner
        Scanner entrada = new Scanner(System.in);
        //imprimimos un mensaje en la pamtalla para que el usuario
ingrese
        //el nombre del alumno
        String nombre;
        int edad;
        double matricula;
        System.out.println("Ingrese el nombre del Alumno: ");
        nombre = entrada.nextLine();
        System.out.println("Ingrese la edad del Alumno");
        edad=entrada.nextInt();
        System.out.println("Ingrese el valor de la matricula");
        matricula = entrada.nextDouble();

    }
}
```

Una vez que se crea el objeto *Scanner,* se puede utilizar diferentes métodos para leer la entrada de datos: *nextInt* o *nextDouble,* leen datos de tipo enteros y de coma flotante.

Cuando el compilador llama a uno de los métodos anteriores, el programa espera hasta que el usuario teclee un número y pulse Enter.

El método *nextLine()* lo que hace es leer una línea de entrada, cuando se utiliza el método *next()* este método sirve para leer una palabra sin espacios.

En Java para usar la clase *Scanner* se debe importar el paquete *java.util* y siempre que se utiliza una clase que no está definida en el paquete básico *java.lang* se necita utilizar la directiva *import*. La primera línea del código indica a Java dónde encontrar la definición de la clase *Scanner*.

RESUMEN DEL CAPÍTULO 3

Este capítulo nos introdujo a los componentes básicos de un programa en Java, en los siguientes capítulos se analizará con detalle cada uno de estos componentes, de igual manera se analizó el modo de utilizar las características mejoradas en Java que nos permitirá escribir programas orientados a objetos. En este capítulo además de aprender la estructura general de un programa en Java y que:

- **Main:** Cuando se ejecuta un programa en Java, el sistema comienza por buscar y ejecutar el método *main* en la clase principal. Todo el código que desees que se ejecute de residir dentro de este método.
- **Métodos definidos por el usuario.** Los métodos definidos por el usuario son una característica fundamental de Java que mejora la estructura y eficiencia del código.
- **Paquetes.** Los paquetes en Java son una característica fundamental que facilita la organización y la gestión de código en proyectos de programación Java, mejorando la claridad, la reutilización y el modularidad del código.
- **Tipos de datos enteros.** Un tipo de dato entero se utiliza para representar números enteros sin parte decimal, como enteros positivos y negativos.
- **Tipos de datos flotantes.** Los datos de coma flotante se utilizan para representar números con decimales, como números reales o fracciones y puede incluir *float y double.*
- **Caracteres.** Un carácter es una representación de un símbolo o letra individual, y el tipo de datos *char* se utiliza para almacenar y trabajar con este tipo de caracteres en aplicaciones Java
- **Boolenanos.** En Java un tipo de dato booleano es una categoría que representa valores lógicos de verdadero o falso y se utiliza para realizar evaluaciones lógicas y controlar el flujo de un programa
- **Entrada y salida de datos.** La entrada y salida de datos son componentes cruciales en la mayoría de las aplicaciones Java, ya que permiten que los programas interactúen con los usuarios y con otros sistemas de manera efectiva. Java ofrece una variedad de clases y métodos que facilitan estas operaciones, lo que hace que sea posible crear aplicaciones interactivas y que puedan leer y escribir datos de manera eficiente.
- **System.in,** en Java es el mecanismo que permite la entrada de datos desde el teclado o dispositivos de entrada estándar en tiempo real

- **System.out,** en Java es un componente esencial para la salida de datos en aplicaciones de consola, permitiendo que los programas muestren información y resultados al usuario de manera efectiva y legible.

Ejercicios

3.1. ¿Cuál es la salida del siguiente programa?

```
public class Alumno {
    //método main
    public static void main(String[] args) {
        System.out.println("Hola Mundo desde Java!\n" + "Programando
soluciones");

    }
}
```

3.2. Identificar el error del siguiente programa

```
public class Alumno {
    //método main
    main() {
        System.out.println("Hola mundo!\n");

    }
}
```

3.3. Escribir un programa donde el usuario ingrese por teclado el nombre y la dirección de una persona y la imprima en consola, el programador debe utilizar la clase *Scanner*. Para resolver el problema.

3.4. Escribir un programa donde el usuario ingrese los datos de la compra de un vehículo utilizando la clase *InputStreamReader* e imprimirlos en pantalla, los datos a ingresar son:

- Nombre del comprador.
- CI
- Dirección
- Teléfono
- Correo electrónico
- Precio del vehículo
- Valor del Iva 12%

Los datos ingresados deben ser en el formato que se los pide.

3.5. Escribir un programa en Java que imprima el mapa de tu país con asteriscos.

CAPITULO 4

OPERADORES Y EXPRESIONES

4.1. OPERADORES Y EXPRESIONES

En Java, los operadores y las expresiones son elementos fundamentales para realizar cálculos, tomar decisiones y realizar diversas operaciones en un programa.

Los programas Java constan de datos, declaraciones y expresiones; estas últimas suelen representar ecuaciones matemáticas; por ejemplo: 7*x 2*z; donde los símbolos más y de producto (+, *) son operadores de suma y de producto; los números 7 y 2 y las variables x y z son operandos; En pocas palabras, una expresión es una secuencia de operaciones y operandos que especifican un cálculo.

Cuando se usa entre números o variables, este operador se llama operador binario porque suma dos números. Otro tipo de operador en Java es el operador unitario o *unario*, que opera sobre un único valor; si la variable x es igual a 5, entonces -x será igual a= -5; el signo menos (-) representa la resta del operador de valor unitario. Java admite un potente conjunto de operadores unarios, binarios y otros.

Sintaxis

```
variable=expresión
```

variable identificador valido para Java, que es declarado como variable

expresión constante, otra variable con un valor previamente asignado o una fórmula evaluada de tipo *variable*

4.1. OPERADORES DE ASIGNACIÓN

Los operadores de asignación en Java son símbolos especiales utilizados para asignar un valor a una variable. Estos operadores desempeñan un papel fundamental en la manipulación y gestión de datos en programas escritos en Java. A través de estos operadores, podemos actualizar el contenido de variables de manera eficiente y concisa.

El operador de asignación básico es el signo igual (=). Su función principal es asignar el valor a la derecha del operador al nombre de variable a la izquierda. Por ejemplo:

```
int numero = 23;
```

Como se ve en el ejemplo el signo = esta asignando un valor a la variable número en el que se guarda el 23, este valor se puede cambia. Sin embargo, Java también ofrece operadores compuestos de asignación que combinan una operación con la asignación en una sola expresión. Estos operadores compuestos permiten realizar operaciones

aritméticas y luego asignar el resultado a la variable en un solo paso. Los operadores compuestos de asignación incluyen:

- : Suma y asignación. Por ejemplo, x += 5; es equivalente a x = x + 5;.
- -=: Resta y asignación. Por ejemplo, y -= 3; es equivalente a y = y - 3;.
- *=: Multiplicación y asignación. Por ejemplo, z *= 2; es equivalente a z = z * 2;.
- /=: División y asignación. Por ejemplo, w /= 4; es equivalente a w = w / 4;.

Estos operadores compuestos son especialmente útiles cuando se trabaja con variables y se necesita actualizar su valor en función de operaciones previas.

4.2. OPERADORES ARTIMÉTICOS

Los operadores aritméticos en Java son símbolos que se utilizan para realizar operaciones matemáticas en valores numéricos. Estos operadores permiten realizar cálculos como suma, resta, multiplicación, división y módulo. A continuación, se proporciona una definición más detallada de los operadores aritméticos en Java:

- **Suma (+).** El operador de suma se utiliza para agregar dos valores numéricos y producir un resultado que es la suma de ambos. Por ejemplo:

```
int resultadoSuma = 23+25; //El resultado sera 48
```

- **Resta (-).** El operador de resta se utiliza para restar un valor numérico de otro y producir un resultado que es la diferencia entre ellos. Por ejemplo:

```
int resultadoResta=65-34;
```

- **Multiplicación (*).** El operador de multiplicación se utiliza para multiplicar dos valores numéricos y producir un resultado que es el producto de ambos. Por ejemplo:

```
int resultadoMultiplicacion=5*23;
```

- **División (/).** El operador de división se utiliza para dividir un valor numérico por otro y producir un resultado que es el cociente de la división. Es importante notar que, si los operandos son números enteros, la división se realizará como una división entera, lo que significa que se truncará cualquier parte decimal.

```
double resultadoDivision=34/6;
```

- **Módulo (%).** El operador de módulo se utiliza para encontrar el resto de la división entre dos números enteros. Es útil para verificar si un número es divisible por otro y para realizar operaciones relacionadas con ciclos y patrones repetitivos.

```
double resultadoModulo= 10%3; // el módulo es 1
```

4.3. OPERADOS DE INCREMENTO Y DECREMENTO.

Los operadores de incremento y decremento en Java son operadores unarios que se utilizan para aumentar o disminuir el valor de una variable en una unidad. Estos operadores son extremadamente útiles en bucles y en situaciones donde es necesario actualizar una variable de manera iterativa. A continuación, se proporciona una definición de los operadores de incremento y decremento en Java:

1. **Operador de Incremento (++):** El operador de incremento (++), a veces llamado "incremento de uno", se utiliza para aumentar el valor de una variable en una unidad. Puede aplicarse a variables numéricas (como enteros o números de punto flotante) y también a variables de tipo char, que representan caracteres.
    a. **Prefijo (++variable):** Cuando se utiliza el operador de incremento en forma de prefijo, primero se incrementa el valor de la variable y luego se utiliza ese valor en la expresión actual.

```
int contador =6;
int resultado = ++contador;//Incrementamos al contador más 1 y el resultado sera 7
```

    b. **Postfijo (variable++):** Cuando se utiliza el operador de incremento en forma de postfijo, primero se utiliza el valor actual de la variable en la expresión actual y luego se incrementa.

```
int contador =6;
int resultado = contador++;//Utiliza el valor actual del contador que 6 luego incrementa el contador en 1
```

2. **Operador de Decremento (--):** El operador de decremento (--), similar al operador de incremento, se utiliza para disminuir el valor de una variable en una unidad. Al igual que con el operador de incremento, el operador de decremento puede aplicarse en forma de prefijo o postfijo.
    a. **Prefijo (--variable):** En esta forma, primero se decrementa el valor de la variable y luego se utiliza ese valor en la expresión actual.

```
int contador =6;
int resultado = --contador;//Decrementa el contador en 1 y luego lo asigana a resultado que es 5
```

    b. **Postfijo (variable--):** En esta forma, primero se utiliza el valor actual de la variable en la expresión actual y luego se decrementa.

```
int contador =6;
int resultado = contador--;//Utiliza el contador con el valor actual en resultado y luego lo decrementa en 1 valor 5
```

4.4. OPERADOS RELACIONALES

Debido a que el tipo de datos booleano de Java tiene los valores falso y verdadero, una expresión booleana es una secuencia de operandos y operadores que se combinan para producir uno de sus valores posibles.

Los operadores que prueban la relación entre dos operandos (como >= o ==) se denominan operadores relacionales y se utilizan en expresiones de la siguiente forma:

Los operadores relacionales se utilizan normalmente en declaraciones selección (if) o las iteraciones (while, for) se utilizan para probar condiciones; a través de ellos se realizan las operaciones de igualdad, desigualdad y diferencia relativa; la tabla 5 enumera los operadores relacionales que se pueden aplicar a cualquier operando de tipo de datos estándar: char, int, float, double, etc.; cuando se usa en una expresión, se evalúa como verdadero o falso según el resultado de la condición; Por ejemplo.

```
boolean c;
c= 4<8;
```

la variable c se asigna a verdadero (true), puesto que 4 es menor que 8, entonces la operación < devuelve un valor verdadero (true), que se asigna a c.

Tabla 5.
Operadores relacionales en Java

Operador	Significado	Ejemplo
==	Igual a	x==z
¡=	No igual a	A ¡=b
>	Mayor a	a>b
<	Menor a	B<a
>=	Mayor igual	a>=b
<=	Menor igual	C<=b

Nota: La tabla representa los operadores relacionales que se utiliza en Java con su significado y ejemplos.

4.5. OPERADORES LÓGICOS

Además de operadores matemáticos, Java tiene operadores lógicos: if, while o for, ya mencionados que se estudiarán más adelante; éstos, denominados booleanos, en honor a George Boole, creador del álgebra que lleva su apellido, son: not (!), and (&&), or (||) y or exclusivo (). not produce falso si su operando es verdadero y viceversa; and resulta en verdadero sólo si ambos operandos son verdaderos; en caso contrario, deriva en falso; or produce verdadero si cualquiera de los operandos tiene ese valor y resulta en falso sólo si ambos lo son; or exclusivo deriva en verdadero si ambos operandos son distintos: verdadero-falso o falso-verdadero, y produce falso sólo si ambos operandos son iguales: verdadero-verdadero o falso-falso. Además, hay que considerar que Java permite utilizar &, | como operadores and y or respectivamente, con el significado mencionado, salvo la evaluación en cortocircuito; la tabla 6 muestra los operadores lógicos de Java.

Tabla 6.
Operadores lógicos

Operador	Operación lógica	Ejemplo
Negación(!)	No lógica	!(x>=y)
o, exclusiva(^)	Operando_1 ^Operando_2	X<n ^ n>9
Y, lógica (&&)	Operando_1 && Operando_2	m<n&&k>l
O, lógica ||	Operando1||operando2	m=5||n!=10

Nota: La tabla muestra los operadores lógicos que se utilizan en las condicionales con sus respectivos ejemplos.

RESUMEN DEL CAPÍTULO 4

En este capítulo se examinó los diferentes tipos de operadores que utiliza Java y cual es su sintaxis y ejemplos.

- **Los operadores en Java** son herramientas esenciales para realizar operaciones matemáticas, comparar valores, tomar decisiones lógicas y controlar el flujo de un programa. Combinados con variables y estructuras de control, los operadores permiten crear programas funcionales y eficientes en Java. Su comprensión y uso adecuado son fundamentales para cualquier programador de Java.
- **Operadores de asignación,** en Java son elementos esenciales para asignar valores a variables y actualizarlas de manera eficiente
- **Operadores aritméticos,** Estos operadores aritméticos en Java son fundamentales para realizar cálculos matemáticos en programas y son ampliamente utilizados en aplicaciones que involucran manipulación de datos numéricos.
- **Operadores de incremento y decremento,** estos operadores son muy útiles para realizar tareas que involucran recorridos y manipulación de índices en arreglos y colecciones, así como para llevar un seguimiento de valores en bucles y otras estructuras de control.
- **Operadores relacionales,** Estos operadores de relación son fundamentales para la programación en Java, ya que permiten tomar decisiones basadas en comparaciones de valores.

Ejercicios

4.1. Determinar el valor de las siguientes expresiones aritméticas.

14/12

24/14

123/34

200/50

4.2. Escribir un programa en Java que lea por teclado dos números enteros de tres dígitos calcule e imprima producto, cociente y resto cuando el primero se divide entre el segundo.

4.3. Escribir un programa en Java donde el usuario ingrese la temperatura en Celcius por teclado y convertir a Farenheit utilizando la siguiente formula.

$$f = \frac{9}{5}C + 32$$

4.4. Escribir un programa en Java para obtener la Hipotenusa y los ángulos de un triángulo rectángulo a partir de las longitudes de los catetos, todos los datos se ingresan por teclado.

4.5. Escribir un programa en Java donde el Usuario ingrese un número por teclado y verifique si el número es primo o no.

CAPITULO 5

ESTRUCTURAS DE SELECCIÓN

5.1. ESTRUCTURAS DE CONTROL.

Las estructuras de control en Java son herramientas esenciales que permiten a los programadores administrar cómo se ejecutan las instrucciones en un programa Java. Estas estructuras permiten tomar decisiones, repetir acciones y controlar el flujo de ejecución de manera que las instrucciones se ejecuten en el orden y bajo las condiciones deseadas. En esencia, las estructuras de control en Java son elementos cruciales que brindan flexibilidad y lógica a la programación, permitiendo la automatización de tareas y la toma de decisiones basadas en condiciones específicas.

```
{
    senticia 1;
    sentencia 2,
    .
    .
    .
    sentencia n
}
```

El flujo de control va de la declaración 1 a la declaración 2 y así sucesivamente; sin embargo, algunos problemas requieren pasos con dos o más opciones o alternativas elegidas en función del valor de una condición o expresión.

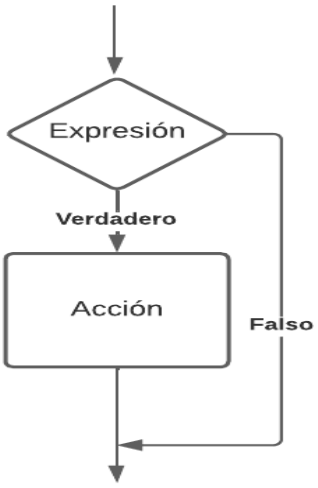
5.2. SENTECIA if

La principal estructura de control de selección de Java es la declaración if; tiene dos alternativas o formatos posibles, el más simple tiene la siguiente sintaxis:

```
if(expresion) Acción
```

La declaración if funciona así: cuando se alcanza, se evalúa la siguiente expresión entre paréntesis; si la expresión es verdadera, la acción se ejecuta, de lo contrario no se ejecuta y la ejecución continúa con la siguiente declaración. Acción es una sentencia simple o compuesta, en la figura xx muestra un diagrama de flujo que indica el proceso de ejecución del programa.

Figura 29
Diagrama de flujo de una sentencia básica if



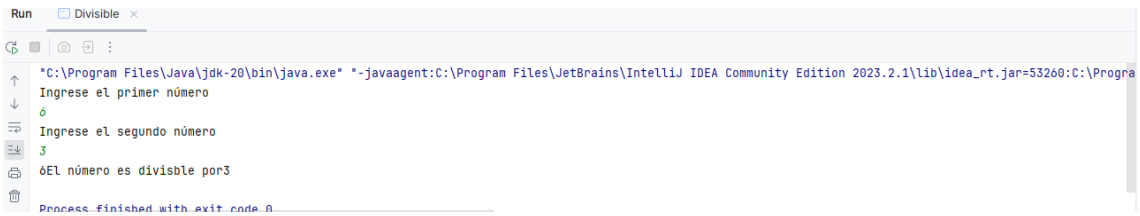
Nota. La figura representa el diagrama de flujo de una sentencia básica.

Ejemplo 5.1

Escribir un programa en Java donde el usuario ingrese dos números enteros por teclado y determinar si el primer número es divisible para el segundo número.

```
package divisible;

import java.util.Scanner;

public class Divisible {
    public static void main(String[] args) {
        int n,d;
        Scanner entrada = new Scanner(System.in);
        System.out.println("Ingrese el primer número");
        n=entrada.nextInt();
        System.out.println("Ingrese el segundo número");
        d=entrada.nextInt();
        if(n%d==0){
            System.out.println(n + "El número es divisble por" + d);
        }
    }
}
```

Figura 30
Ejecución del programa



Nota: La imagen muestra la ejecución del código anterior donde el primer número es divisible para el segundo número ingresado.

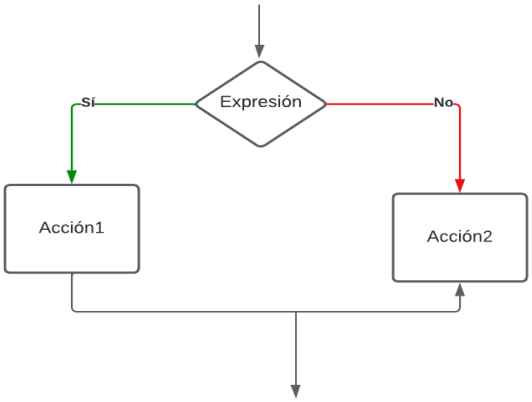
5.3. SENTECIA if-else.

Un segundo formato del if es la expresión if-else, cuyo formato tiene la siguiente sintaxis:

```
if(expresion)
    accion1
else
    accion2
```

En este formato acción 1 y acción 2 son, de forma individual, una única sentencia que termina en un punto y coma, o un grupo de sentencias entre llaves; expresión se evalúa cuando se ejecuta la sentencia: si es verdadera, se efectúa acción 1; en caso contrario se ejecuta acción 2, la figura 5.2 muestra su semántica.

Figura 31
Diagrama de flujo sentencia if-else



Nota: La figura representa el diagrama de flujo de la sentencia if-else

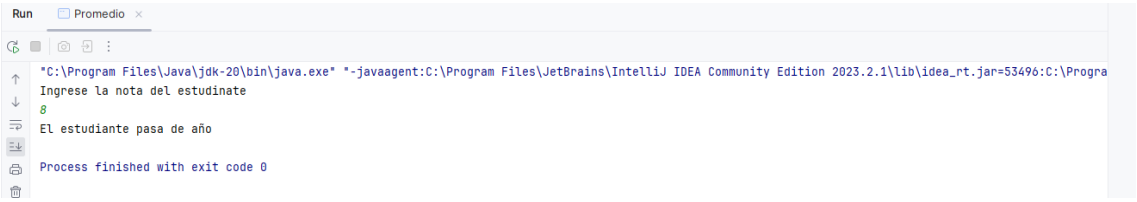
Ejemplo 5.2

Escribir un programa donde el usuario ingrese una nota del examen, si la nota del examen es mayor a 7 mostrar un mensaje pasa de año, caso contrario si la nota es menor igual a 7 mostrar el mensaje el estudiante pierde el año.

```
package notas;
import java.util.Scanner;
public class Promedio {
    public static void main(String[] args) {
        int nota;
        Scanner entrada = new Scanner(System.in);
        System.out.println("Ingrese la nota del estudinate");
        nota=entrada.nextInt();
        if (nota > 7) {
            System.out.println("El estudiante pasa de año");
        }else{
            System.out.println("El estudiante no pasa el año");
        }
```

```
    }
}
```

Figura 32
Salida del programa if-else



Nota: La figura muestra la salida en pantalla del programa de notas, donde si el estudiante saca más de 7 pasa de año, caso contrario pierde

5.4. SENTENCIA DE CONTROL SWITCH

En Java, switch es una sentencia que se utiliza para elegir una de entre múltiples opciones; es especialmente útil cuando la selección se basa en el valor de una variable simple o de una expresión simple denominada expresión de control o selector; el valor de dicha expresión puede no de tipo double.

Syntaxis ser de tipo int o char pero

```
switch(selector)
{
    case etiquea1 : sentencia1;
        break;
    case etiqueta2 : setencia2;
        break;
    .
    .
    .
    case etiquetan : sentenciasn;
        break;
    default: sentenciasd ;
}
```

a expresión de control o selector se evalúa y compara con cada una de las etiquetas de case; además, debe ser un tipo ordinal, por ejemplo, int, char, bool pero no float o string; cada etiqueta es un valor único, constante y debe tener un valor diferente de los otros. Si el valor de la expresión es igual a una de las etiquetas case, por ejemplo, etiqueta1, entonces la ejecución comenzará con la primera sentencia de sentencias1 y continuará hasta encontrar break o el final de switch. El tipo de cada etiqueta debe ser el mismo que la expresión de selector; las expresiones están permitidas como etiquetas, pero sólo si cada operando de la expresión es por sí misma una constante; por ejemplo, 4, +8 o m*15, siempre que hubiera sido definido anteriormente como constante nombrada. Si el valor del selector no está listado en ninguna etiqueta case no se ejecutará ninguna de las opciones a menos que se especifique una acción predeterminada. La omisión de una etiqueta default puede crear un error lógico difícil de prever; aunque ésta es opcional, se

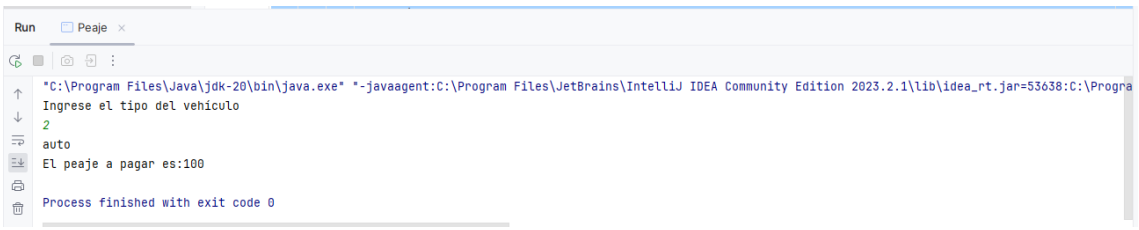
recomienda su uso, a menos de estar absolutamente seguro de que todos los valores de selector están incluidos en las etiquetas case.

Ejemplo 5.3

Escribir un programa donde el programa simule un peaje donde ingrese el tipo de vehículo y me muestre el valor de pagar en el peaje.

```
package peaje;

import java.util.Scanner;

public class Peaje {
    public static void main(String[] args) {
        int tipoVehiculo, peaje;
        Scanner entrada= new Scanner(System.in);
        System.out.println("Ingrese el tipo del vehículo ");
        tipoVehiculo=entrada.nextInt();
        switch (tipoVehiculo){
            case 1:
                System.out.println("Pesado");
                peaje=500;
                System.out.println("El peaje a pagar es:" + peaje);
                break;
            case 2:
                System.out.println("auto");
                peaje=100;
                System.out.println("El peaje a pagar es:" + peaje);
                break;
            case 3:
                System.out.println("moto");
                peaje=50;
                System.out.println("El peaje a pagar es:" + peaje);
                break;
            default:
                System.out.println("Opción no encontrada");
        }
    }
}
```

Figura 33
Salida de pantalla peaje



Nota: La figura representa a la salida de pantalla donde el usuario ingresa el tipo de vehículo y muestra cuanto debe pagar.

RESUMEN DEL CAPÍTULO 5

Estructuras de Control en Java es fundamental para comprender cómo se diseña la lógica de un programa Java. Permite a los programadores tomar decisiones, crear aplicaciones interactivas y controlar el flujo de ejecución de manera efectiva. La comprensión y el dominio de estas estructuras son esenciales para escribir programas funcionales y lógicos en el lenguaje de programación Java.

- **Estructuras de control,** Las estructuras de control en Java son esenciales para escribir programas complejos y lógicos. Permiten que los programas realicen cálculos, tomen decisiones basadas en condiciones y ejecuten acciones de manera eficiente.
- **If,** es una herramienta fundamental en programación que permite realizar bifurcaciones en la ejecución del programa, ejecutando ciertas instrucciones solo cuando se cumple una condición dada.
- **If-else,** es una herramienta fundamental en programación que permite gestionar el flujo del programa, ejecutando diferentes acciones en función de si una condición es verdadera o falsa.
- **Switch,** es útil cuando se necesita tomar decisiones basadas en múltiples opciones y es una alternativa a las estructuras if y else if cuando se tienen varios casos para evaluar.

Ejercicios

5.1. ¿Cuáles son los errores de sintaxis que tiene la siguiente sentencia?

```
if x> 26.0
    y=x
else
    y=z;
```

5.2. Escribir un programa que determine si un año es bisiesto, para este programa se sabe que cuando es múltiplo de 4, por ejemplo, 1984, sin embargo, los años que son múltiplos de 100 sólo son bisiestos cuando también son múltiplos de 400, por ejemplo 1800 no es bisiesto, mientras que 2000, si lo es.

5.3. Escribir un programa en Java que resuelva la ecuación cuadrática ($ax^2 + bx + c = 0$)

5.4. Escribir un programa que calcule el número de días de un mes, dados los valores numéricos del mes y el año.

CAPITULO 6

ESTRUCTURA DE REPETICIÓN

6.1. ESTRUCTURA DE REPETICIÓN.

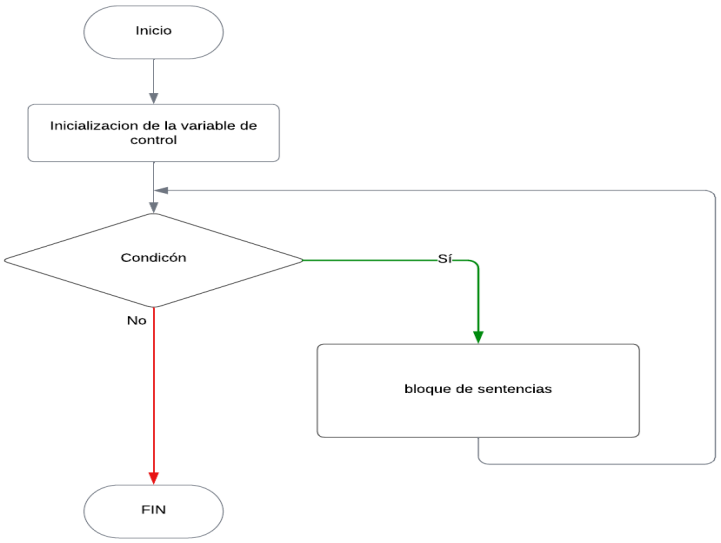
Las estructuras de repetición en Java son herramientas cruciales para automatizar procesos repetitivos en la programación. Estas se dividen principalmente en dos categorías: el bucle “for” y el bucle “while”. Exploraremos detalladamente cada una de ellas, brindando claridad sobre sus conceptos fundamentales y proporcionando una guía visual mediante diagrama de flujos.

6.1.1. BUCLE FOR EN JAVA

El bucle **For** es una estructura de control que permite repetir un bloque de código un número especifico de veces. Este tipo de bucle es especialmente útil cuando se sabe de antemano cuántas veces se desea repetir una determinada tarea.

El diagrama de flujo de una estructura “for” es similar al diagrama de flujo de la estructura “while” que la estudiaremos más adelante. Un “for” verifica si la condición se cumple entra al bucle y se repite las veces necesarias que la condición le indica, caso contrario no entra y termina la ejecución.

Figura 34
Diagrama de flujo ciclo for



Nota: La figura representa al diagrama de flujo del ciclo repetitivo “for”.

Un “for” tiene la siguiente sintaxis en Java.

```
for (incicializacion; condicion; incremento/decremento) {
    bloque de codigo

}
```

Cuando el programa ejecuta el ciclo “for”, lo primero que hace es evaluar la condición si la condición es verdadera entra al ciclo, en el caso de que la condición sea false termina el ciclo “for”.

Una vez que entra al ciclo se va a ejecutar todo el bloque de código las veces que se cumpla la condición, con cada iteración se va evaluar la condición convirtiéndose en un ciclo finito. Cuando la condición llega a su fin el ciclo “for” termina el flujo y continua con las sentencias siguientes al “for”.

Ejemplo 6.1

Crear un programa en Java donde se imprima los números del 1 al 10 utilizando la sentencia de control “for”.

```
package ciclofor;

public class Main {
    public static void main(String[] args) {
        for (int i = 0; i <= 10; i++) {
            System.out.println("El número es: " + i);

        }
    }
}
```

Figura 35.
Salida de pantalla ciclo “for”

Nota: La figura representa la salida de pantalla de la ejecución del código de la sentencia “for”, mostrando los números del 0 al 10.

Normalmente la variable de control se declara y se inicializa dentro de la condición del “for”. En este ejemplo se declara una variable **int i=0,** es decir que nuestro ciclo de repetición va a empezar en 0 y la condición a evaluar es **i<=10**. Por último, la variable **i** se incrementa en 1 por cada iteración, el incremento se realiza con el operador **++.**

Dentro del ciclo "for", se imprime la variable de iteración las veces que la condición se cumpla.

Ejemplo 6.2

Crear una aplicación en Java donde utilizando el ciclo **"for"** sume los números impares del 1 al 50 la suma de los números se debe guardar en una variable de tipo entero llamado suma.

```
package org.elvis;

public class SumaImpares {
    public static void main(String[] args) {
        //Declaramos la variable de tipo entero de nombre suma
        int suma;
        //inicializamos la vatiable suma
        suma=0;
        //Implementamos el ciclo for
        for (int i = 0; i < 50 ; i+=2) {
            suma = suma+i;
        }
        System.out.println("La suma de los números impares  del 1 al
50: " + suma);
    }
}
```

Figura 36.
Salida de Pantalla

Nota: La figura representa a la ejecución del ejercicio 6.2, en donde al ejecutar muestra la suma de los números impares del 1 al 50.

6.1.2. CICLO FOR ANIDADOS

Los ciclos **"for"** anidados son una técnica poderosa en la programación que permite realizar múltiples iteraciones dentro de otra iteración. Este enfoque es útil para operaciones que involucran estructura de datos bidimensional, (tema que lo trataremos en capítulos más adelante) o para aplicar un conjunto de instrucciones a cada par de elementos en dos o más conjuntos de datos. Vamos a explorar en detalle cómo funcionan estos ciclos, su sintaxis en Java.

Sintaxis del ciclo For Anidados

La sintaxis general de un ciclo **for anidado** es la siguiente:

```
//sintaxis
for(inicialización1; condición1; expresión_de_actualización1){
    for (inicialización2; condicion2; expresion_de_actualización2) {
        //bloque de código del bucle interno
    }
    //bloque de código del bucle externo
}
```

- **Inicialización1:** Se inicializa las variables de control del bucle externo
- **Codición1:** Es la condición que se evalúa antes de cada iteración del bucle externo. Si es verdadera, el bucle se ejecuta; de lo contrario, el bucle se termina.
- **Expresión_de_actializacion1:** Se ejecuta después de cada iteración del bucle externo.
- **Inicialización2:** Se inicializan las variables de control del bucle interno
- **Condición2:** Es la condición que se evalúa antes de cada iteración del bucle interno. Si es verdadera, el bucle se ejecuta; de lo contrario, el bucle se termina.
- **Expresión_de_actualización2:** Se ejecuta después de cada iteración del bucle interno.

Funcionamiento del ciclo for anidado

Los ciclos for anidados ejecutan el bucle interno completo para cada iteración del bucle exterior. Esto significa que, para cada iteración del bucle externo, se realizan todas las iteraciones del bucle interno. Este proceso continua hasta que se alcanza la condición de terminación del bucle externo.

Ejemplo 6.3.

Realizar un programa en Java donde se genere una matriz 3x3, y mediante un for anidado imprimir los datos de la matriz.

```
package org.elvis;

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

        int[][] matriz={{1, 2, 3},{4, 5, 6}, {7, 8, 9}};
         for (int i = 0; i < matriz.length; i++) {
             for (int j = 0; j < matriz[i].length ; j++) {
                 System.out.print(matriz[i][j]);
             }
             System.out.println("");
         }

    }
}
```

Figura 37.
Salida de Pantalla de una matriz

Nota: La figura representa la ejecución del ejercicio 6.3, en donde al ejecutar muestra en consola lo datos que contiene la matriz.

Para este ejemplo, el bucle externo recorre las filas de la matriz mediante la variable **"i"**, y el bucle interno recorre las columnas **"j".** En cada iteración, se imprime el elemento correspondiente de la matriz.

Explicación del código.

1. Declaramos e inicializamos una variable de tipo entero donde se va a guardar los valores de la matriz de nombre **matriz,** con los siguientes elementos **1, 2, 3; 4, 5, 6; 7, 8, 9**.
2. Implementamos un ciclo **for,** que va a controlar la filas, donde la variable de iteración es **i,** el ciclo va a empezar en 0 y terminara hasta llegar al final de la longitud de la matriz en este caso es 3, y se va a incrementar de uno en uno, una vez que entra al primer ciclo se ejecuta el segundo **for,** este ciclo controlara las columnas, donde la variable de iteración es **j,** este ciclo empezara en 0 y terminara en el tamaño de la longitud de la variable **i,** y se va incrementando de uno en uno.
3. Se imprime el primer valor de la matriz que es **1,** termia el ciclo y vuelve a revisar al condición y si se cumple vuelve a entrar al bloque de código del **for interno** imprime el segundo valor de la primera fila y segunda columna que es **2,** termina el ciclo y vuelve a evaluar la condición del **for**, como la condición se cumple vuelve a ingresar al bloque de código e imprime el valor de la primer fila tercera columna que es **3,** termina el ciclo y vuelve a evaluar la condición esta vez la condición no se cumple termina el ciclo y entra a la sección del **for** externo, se imprime un salto de fila y evalúa la condición del **for** externo, como la condición si se cumple se repite por segunda vez y vuelve a repetir el ciclo interno las veces que la condición se cumpla.

El ciclo "foreach" en Java, también conocido como "enhanced for", es una característica introducida en Java 5 que simplifica la iteración a través de elementos de una colección

o array. En esta sección, exploraremos en detalle cómo funciona el ciclo "foreach", proporcionando información detallada.

La información de colecciones la estudiaremos con más profundidad en capítulos más adelante.

La sintaxis es más simple que la de un bucle **"for"** tradicional y es ideal cuando se necesita recorrer todos los elementos de una colección sin preocuparse por los índices.

La sintaxis general del ciclo **"foreach"** es la siguiente.

```
for(tipo elemento : coleccion){
    //Bloque de código a ejecutar para cada elemento
}
```

- **Tipo.** Es el tipo de dato de los elementos en la colección
- **Elemento.** Es una variable que representa cada elemento de la colección en cada iteración.
- **Colección.** Es la colección (array, lista, conjunto, etc) sobre la que se está iterando.

El ciclo **"foreach"** itera sobre cada elemento de la colección, asignando cada elemento sucesivamente a la variable **"elemento"**, y ejecuta el bloque de código dentro del bucle para cada elemento. Este ciclo se detiene automáticamente cuando todos los elementos de la colección han sido procesados.

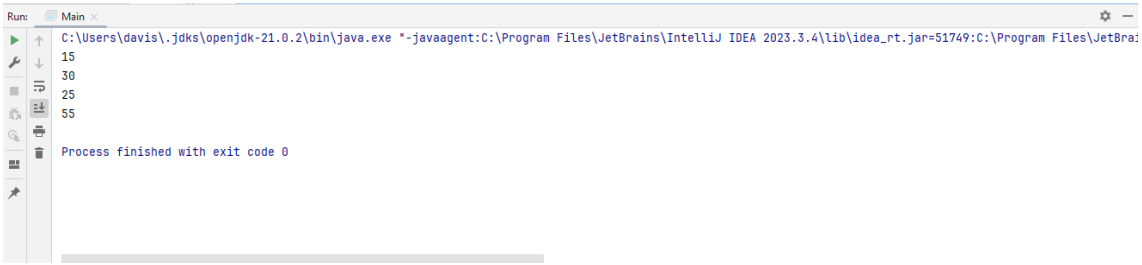
Ejemplo 6.4

Crear un programa en Java donde se cree una lista de tipo Integer, añadir una lista de 4 números enteros, a la lista, recorrer cada elemento de la lista con un **"foreach"**, e imprimirla.

```
package org.example;

import java.util.ArrayList;

//TIP To <b>Run</b> code, press <shortcut actionId="Run"/> or
// click the <icon src="AllIcons.Actions.Execute"/> icon in the
gutter.
public class Main {
    public static void main(String[] args) {

        //Creamos una lista de cadenas
        ArrayList<Integer> lista = new ArrayList<>();
        lista.add(15);
        lista.add(30);
        lista.add(25);
        lista.add(55);

        //Iteramos la lista con el foreach
        for (Integer numero : lista){
```

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

    }
}
```

Figura 38.
Salida de Pantalla ciclo "foreach"



Nota: La salida en pantalla nos mostrara el recorrido de toda lista imprimiendo en pantalla todos los elementos de la lista.

El ciclo de iteración **"foreach"** no tiene un diagrama de flujo especifico ya que su estructura es bastante simple y se basa en la iteración de elementos de una colección. Sin embargo, se puede representar de manera abstracta utilizando un diagrama de flujo genérico que muestra el proceso de iteración sobre una colección y la ejecución del bloque de código para cada elemento.

Este diagrama muestra cómo el ciclo **"foreach"** itera sobre los elementos de una colección y ejecuta el bloque de código asociado para cada elemento.

El ciclo **"foreach"** en Java proporciona una forma simple y legible de iterar sobre colecciones de elementos sin la necesidad de gestionar manualmente los índices o los contadores de bucles. Es útil para realzar operaciones repetitivas en colecciones de datos, como listas, conjuntos y matrices.

RESUMEN DEL CAPITULO 6

Los ciclos **"for" y "foreach"** son estructuras fundamentales en Java utilizadas para iterar sobre colecciones de datos, como matrices, listas y conjuntos. Aunque comparten el propósito de repetir un bloque de código múltiples veces, cada uno tiene sus propias características y aplicaciones específicas.

For: El ciclo **"for"** es una estructura de control de flujo que permite ejecutar un bloque de código un número especifico de veces. Su sintaxis consta de tres partes:

- La inicialización.
- La condición de continuación
- Expresión de actualización.

Estas partes determinan cómo se realiza la iteración y permiten personalizar el comportamiento del bucle según las necesidades del programa.

El ciclo **"for"** es especialmente útil cuando se conoce de antemano el número exacto de iteraciones que se deben realizar, como en el caso de iteraciones sobre matrices o en situaciones que quieren un conteo especifico.

For anidados: Los ciclos **for** anidados son una herramienta valiosa en la programación, permitiendo realizar operaciones complejas sobre estructuras de datos multidimensionales o aplicar un conjunto de instrucciones a cada para de elementos en dos o más conjuntos de datos. Su sintaxis clara y concisa en Java facilita su implementación, mientras que su versatilidad los hace adecuados para una amplia gama de aplicaciones en el desarrollo de software.

Foreach: El ciclo **"foreach"**, también es conocido como "enhanced for loop", es una construcción más reciente en Java que simplifica la iteración sobre colecciones de datos. Se utiliza principalmente para recorrer matrices y colecciones sin la necesidad de mantener un índice de iteración explicito.

Su sintaxis es más compacta y legible que la del ciclo **"for"** convencional.

EJERCICIOS PROPUESTOS.

1. Escribir un programa en Java donde se calcule la suma de los primeros N números naturales. El usuario deberá ingresar N por teclado, se debe usar un for para el cálculo de la suma.
2. Escribir un programa en Java donde por teclado se ingrese un número natural positivo, calcular la factorial de dicho número, para la realización del programa se debe utilizar el ciclo for.

3. Escribir un programa en Java donde el usuario muestre la tabla de multiplicar de un número, el número debe ser ingresado por teclado. Para realizar el programa se debe utilizar el ciclo for.
4. Escribir un programa donde se imprima un patrón de asteriscos como se muestra formando un triángulo rectángulo, para realizar el programa se debe utilizar for anidados.

   *

   **

   ***

   ****

   *****
5. Escribir un programa en Java donde se calcule la suma de una matriz, para realizar el programa se debe utilizar for anidados.
6. Escribir un programa en Java donde se encuentre número mayor de una lista, el usuario deberá ingresar los elementos de la lista por teclado y deberá usar el foreach para realizar el ejercicio.
7. Escribir un programa en Java donde se pueda encontrar los números pares que se encuentran en una lista, para realizar el programa se debe utilizar foreach.

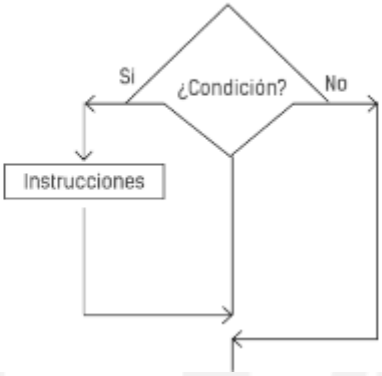
CAPITULO 7

CICLOS REPETITIVOS 2

7.1. CICLO WHILE

El ciclo **"while"** en Java es una estructura de control de flujo que permite repetir un bloque de código mientras una condición booleana sea verdadera. Se utiliza el ciclo **"while"** cunado no sabemos cuántas veces queremos que se repita nuestro bloque de código.

Figura 39.
Diagrama de flujo ciclo while



Nota: La figura representa al diagrama de fujo del ciclo while.

Sintaxis del ciclo while

```
while (condición de finalización){
    //bloque de instrucciones
}
```

- **While:** Es la palabra reservada en Java para utilizar el ciclo traduciendo al español significa **mientras.**
- **Condición:** En esta sección de código implementamos la condición acompañada con los operadores lógicos, "==", ">"," <", ">=", "<=", entre otros. Esta condición si se llega a cumplir permite entrar al bloque de código y ejecutara todas las instrucciones que tenemos.

  Algo muy importante del ciclo **"while",** es que para entrar al ciclo tiene que si o si cumplirse la condición para ejecutar las instrucciones que se encuentran dentro del "**while".**

  Como hemos comentado, un ciclo podemos definirlo como una estructura que nos permite repetir o iterar un conjunto de instrucciones o sentencias. Como

podemos observar en la **figura 39.** Las veces que se va a repetir el código depende de la condición que se ejecute.

7.1.1. Ventajas del Ciclo While

- **Flexibilidad en la condición de salida.** El ciclo **"while"** es útil cuando la cantidad de iteraciones no es conocida de antemano y depende de una condición especifica. Esto proporciona flexibilidad para manejar situaciones donde la cantidad de repeticiones puede variar.
- **Simplicidad.** El ciclo **"while"** tiene una sintaxis simple y clara, lo que facilita su comprensión y uso, especialmente para principiantes en programación.
- **Eficiencia en ciertos casos.** En situaciones donde se requiere que un bloque de código se ejecute repetitivamente mientras se cumple una condición, el ciclo **"while"** puede ser más eficiente que otros bucles, ya que evita la evaluación de una condición al final de cada iteración.

7.1.2. Desventajas de ciclo while.

Cuando utilizamos el ciclo **"while"** existe ciertas desventajas que puede afectar al desarrollo de la aplicación.

- **Posibilidad de ciclos infinitos.** Si la condición del ciclo **"while" nunca** se vuelve falsa, puede causar un ciclo infinito, lo que resulta en un rendimiento ineficiente o incluso en el bloqueo del programa.
- **Requiere inicialización fuera del bucle.** A diferencia de otros bucles como el bucle **for**, el ciclo **"while"** requiere que la variable de control se inicialice antes del bucle, lo que puede aumentar la complejidad y el riesgo de errores si no se maneja correctamente.
- **Potencial para errores de lógica.** Si la condición de salida no se actualiza correctamente dentro del bloque de código del ciclo **"while",** puede resultar en un comportamiento inesperado o incorrecto del programa.

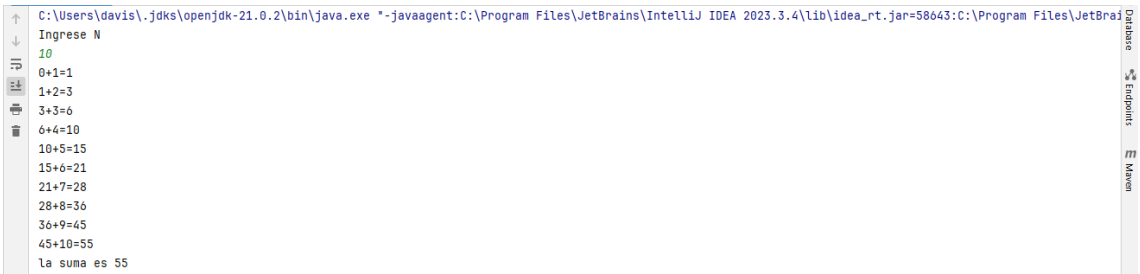
El ciclo **"while"** es una herramienta poderosa en Java que ofrece flexibilidad y simplicidad en situaciones donde la cantidad de iteraciones es desconocida o depende de una condición especifica. Sin embargo, su uso requiere de precaución para evitar bucles infinitos y errores de lógica de programación. Al comprender las ventajas y desventajas del ciclo **"while",** los programadores pueden utilizarlos de manera efectiva en sus proyectos para mejorar la eficiencia y la claridad del código.

Ejemplo 7.1

Escribir un programa en Java que calcule la suma de los primeros N números naturales utilizando un ciclo **"while"**, el usuario debe ingresar N por teclado, y mostrar la suma de cada uno de los ciclos y al final mostrar la suma total.

```
package org.elvis;
import java.io.*;
public class SumaWhile {
    public static void main(String[] args) throws IOException {
        //Declaramos e inicializamos una variable BufferedReader para
leer datos
        //por teclasdo
        BufferedReader br= new BufferedReader(new
InputStreamReader(System.in));
        //Inicializamos una variable de tipo entero para controlar el
ciclo while
        int ciclo, contador,suma, sumaAnterior;
        contador=1;
        suma=0;
        //Ingresamos N por teclado
        System.out.println("Ingrese ");
        ciclo = Integer.parseInt(br.readLine());
        //Mediante el ciclo while controlamos la iteraciones de las
operaciones
        while (contador<=ciclo){
            //guardamos en una variable la suma anaterior para poder
comtrolar la impreison
            sumaAnterior=suma;
            //Realizamos la suma
            suma=suma+contador;
            //imprimimos lo que necesita el cliente
            System.out.println(sumaAnterior + "+" + contador + "=" +
suma);
            //Incrementamos el contador en 1
            contador++;
        }
        //imprimimos la suma total
        System.out.println("la suma es "+ suma);
    }
}
```

Figura 40.
Salida de pantalla suma N números



Nota: En la figura se muestra la salida de pantalla del ejemplo 7.1 donde se ingresa por teclado N y se muestra en pantalla la suma de cada una de las iteraciones.

Explicación del código

1. Primero declaramos una variable de la clase **BufferedReader** es una clase en Java diseñada para leer texto de fuentes de entrada de caracteres, como archivos de texto o la entrada estándar por teclado. Ayuda a mejorare el rendimiento almacenando internamente los caracteres en un búfer, lo que minimiza las operaciones de entrada y salida. Al mismo tiempo declaramos la clase **InputStreamReader**, que es una clase que se utiliza para convertir un flujo de entrada de bytes en un flujo de entrada de caracteres.

2. Declaramos e inicializamos las variables de tipo entero que van a manejar el ciclo **"while".**

    a. **Ciclo.** La variable ciclo de tipo entero va a controlar hasta que número se va a repetir el ciclo esta va a tomar el valor que le indique el usuario ingresando por teclado.

    b. **Contador.** Esta variable de tipo entero es la que se encargara de ir contando y aumentando las veces que se va repitiendo el código,

    c. **Suma.** La variable de tipo entero suma, es la que va ir guardando la suma de cada una de las iteraciones que va hacer el ciclo while.

    d. **sumaAnterior.** Esta variable de tipo entero me va a permitir guardar la suma anterior antes que realice la suma por cada iteración y de esa manera poder imprimir en pantalla.

    e. Inicializamos la variable **contador** en **1** y la variable **suma** en **0**

    f. Leemos la variable N por teclado para controlar el límite del ciclo.

    g. Implementamos el ciclo **"while"** donde la condición es mientras el **contador** sea <= a **N,** N número ingresado por teclado, el ciclo se va a repetir, una vez entrado al ciclo se realiza el siguiente bloque de código.

h. Inicializamos la variable **sumaAnterior** con un valor de **suma**, en este caso se inicializa con **0,** en la primera ejecución del ciclo.

i. En la línea 20 se realiza la suma, donde se va acumulando el valor de suma más el valor del contador. En la primera iteración seria **0+1=1**

j. En la línea 22 realizamos una impresión en consola donde nos muestra el valor de la **sumaAnterior + contador = suma.** Imprimiendo en consola el siguiente valor. **0+1=1.**

k. En la línea 24 al contador se le sumará 1, teniendo en la primera pasa el valor del contador cambiará y será **1.**

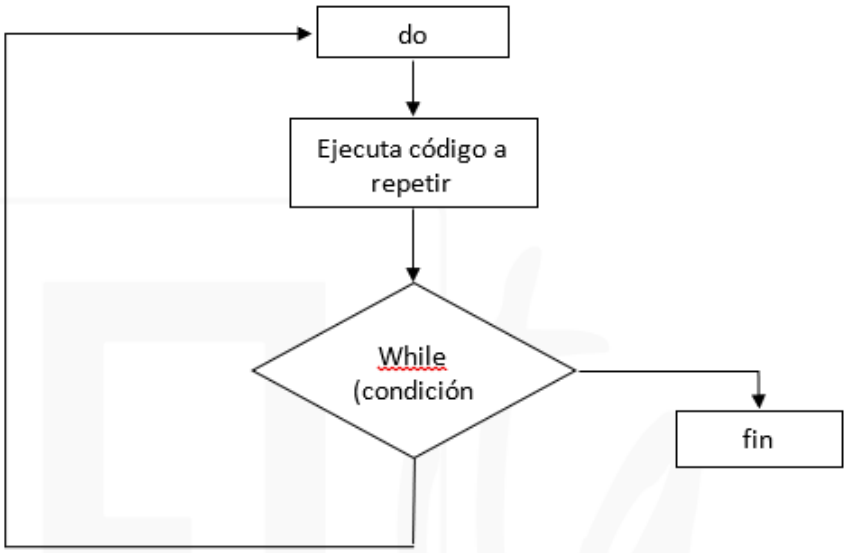
l. Regresa al **while** y la condición a verificar es la siguiente si **contador** que tiene el valor de **1** es **<= N,** como la condición se cumple regresamos al paso **g.** Hasta que la condición sea falsa. Una vez la condición sea falsa se termina el ciclo.

m. En la línea **27** imprimimos en pantalla la suma total de los primeros **N** números.

7.2. CICLO DO WHILE

El ciclo **"do while"** en Java es una estructura de control de flujo que ejecuta un bloque de código al menos una vez y luego repite la ejecución mientras una condición booleana especifica sea verdadera. A diferencia del ciclo **"while"**, que para entrar al ciclo debe cumplirse la condición, el ciclo **"do while",** ejecuta por lo menos una vez la sección a repetir, y luego evalúa la condición y si es falsa termina la ejecución del ciclo **"do while".**

Figura 41.
Diagrama de flujo ciclo do while



Nota: La figura representa al diagrama de flujo del ciclo **"do while".**

SINTAXIS DEL CICLO DO WHILE

```
do{
    //Bloque de código
    //y sección que se repite si se cumple la condición
}while(condicion);
```

- La palabra clave **"do"** marca el inicio del bloque de código que se ejecutará al menos una vez.
- El bloque de código se ejecuta primero antes de que se evalué la condición.
- La palabra clave **"while"** se utiliza para especificar la condición de salida del ciclo.
- Después de cada iteración del bloque de código se evalúa la condición y se determina si el ciclo debe continuar o no ejecutándose.

Es importante tener en cuenta que el punto y coma; es necesario al final de la declaración del ciclo **"do while".** Esto diferencia al ciclo **"do while"** de otras estructuras de control de flujo en Java.

7.2.1. VENTAJAS DEL CICLO WHILE

- **Ejecución garantizada al menos una vez.** La principal ventaja del ciclo **"do while"** es que garantiza que el bloque d código se ejecute al menos una vez, independientemente de la condición booleana. Esto lo hace útil en situaciones

donde se necesita ejecutar un bloque de código inicial antes de evaluar la condición.

- **Sintaxis clara y concisa.** La sintaxis del ciclo **"do while"** es simple y clara, lo que facilita su comprensión y uso, especialmente para principiantes en programación.
- **Flexibilidad en la condición de salida.** Al igual que con el ciclo **"while"**, el ciclo **"do while"** ofrece flexibilidad en la condición de salida, lo que permite manejar situaciones donde la cantidad de repeticiones puede variar.

7.2.2. DESVENTAJAS DE CICLO DO-WHILE

- **Menor eficiencia en ciertos casos.** En comparación con otros bucles como el **"for"** y el **"while"** el ciclo **"do while"** puede ser menos eficiente en términos de rendimiento, ya que siempre ejecuta el bloque de código al menos una vez antes de verificar la condición.
- **Posibilidad de bucles infinitos.** Si la condición del ciclo **"do while"** nunca se vuelve falsa, puede causar un ciclo infinito, lo que resulta en un rendimiento ineficiente o incluso en el bloqueo del programa.
- **Requiere inicialización fuera del bucle:** Al igual que el ciclo **"while"**, el ciclo **"do while"** requiere que la variable de control se inicialice antes del bucle, lo que puede aumentar la complejidad y el riesgo de errores si no se maneja correctamente.

El ciclo **"do while"** es una estructura útil en Java que garantiza la ejecución de un bloque de código al menos una vez y luego repite la ejecución mientras se cumpla una condición especifica. Si bien ofrece ventajas en términos de claridad y flexibilidad, los programadores deben tener cuidado con el riesgo de bucle infinitos y la posible menor eficiencia en comparación con otros bucles. Al comprender las ventajas y desventajas del ciclo **"do while"**, los programadores pueden utilizarlo de manera efectiva en sus proyectos para mejorar la eficiencia y la claridad del código.

Ejercicio 7.2.

Escribir un programa en Java que solicite al usuario ingresar un número utilizando un ciclo **"do while".** El programa debe seguir solicitando la entrada hasta que el usuario ingrese un número valido, un número valido es un número entero positivo.

```
package org.elvis;
import java.io.*;

public class NumeroValido {
```

```
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        int numero;
        do{
            System.out.println("Ingrese un número");
            numero=Integer.parseInt(br.readLine());
            if (numero <= 0) {
                System.out.println("Vuelva a ingresar el número");
            }
        }while(numero<=0);
        System.out.println("Felicidades ");
    }
}
```

Figura 42.
Ejecución del código Ejercicio 7.2

Nota: La figura nos muestra la ejecución del código del ejercicio 7.2. Utilizando ciclo **do while**

Explicación del código ejercicio 7.2

En esta sección explicaremos línea a línea cómo funciona el ejercicio.

1. Declaramos e inicializamos una variable de la clase **BufferedReader** y la clase **InputStreamReader,** para ingresar datos por teclado la variable en donde vamos a guardar el buffer es **br.**
2. En loa línea 7 declaramos una variable te tipo entero donde guardamos el valor que ingresemos por teclado.
3. En la línea 8 a la línea 14 implementamos el ciclo **"do while",** en la línea 8 nos dice **hacer.**
4. Entramos al ciclo, en la línea 9 mostramos un mensaje al usuario diciendo que ingrese el número por teclado.
5. En la línea 10, en la variable número leemos y guardamos los datos ingresados por teclado.
6. En la línea 11 realizamos una condicional donde le preguntamos, que si el número ingresado por teclado es **menor igual a 0.** Si la condicional nos da verdadero quieres decir que la información ingresada es verdadera y nos muestra un

mensaje pidiendo que volvamos a ingresar el número, caso contrario termina la condicional. Esta es la sección del código que se va a iterar o repetir si la condición es verdadera desde el punto 3.

7. En la línea 14 se evalúa la condición diciendo, **mientras** que el número ingresado sea **menor igual a 0,** toda la sección desde la línea 8 a la 14 se repetirá las veces necesarias hasta que le usuario ingrese un número entero positivo.
8. Si el usuario ingresa un número entero positivo termina el ciclo **do while** y nos muestra el mensaje de la línea 15, "**Felicidades a ingresado un número entero positivo.**"

RESUMEN DEL CAPITULO 7

En este capítulo hemos estudiado dos de los ciclos más importante de la programación en Java que nos van a ayudar a mejorar la calidad de desarrollo de nuestras aplicaciones, además a entender como la iteración o a la repetición de una sección de código es importante al momento de desarrollar las aplicaciones.

El ciclo **"while "en** Java es una estructura de control de flujo que permite repetir un bloque de código mientras una condición especifica sea verdadera. Esta estructura se utiliza cuando el número de repeticiones no se conoce o depende de una condición particular. La sintaxis básica del ciclo **"while"** es la siguiente.

```
while (condicion){
    //Bloque de código a repetirse
}
```

El bloque de código dentro del ciclo **"while"** se ejecuta repetidamente mientras la condición especifica sea evaluada como verdadera. Si la condición es falsa no entra al ciclo y en si jamás se llega a ejecutar el bloque de código.

A diferencia del ciclo **"while"**, que para entrar a la iteración si o si se debe cumplir la condición dada es decir debe ser verdadera, en cambio el ciclo **"do while",** se ejecuta primero el bloque de código y de ahí realiza le evaluación de la condición si la condición es verdadera vuelve a repetir el bloque de código de la iteración, eso quiere decir que, si o si se va a ejecutar por lo menos una vez, la sección del código.

La sintaxis del ciclo **"do while"** es la siguiente.

```
do {
    //bloque de código a repetir
}while(condicion);
```

EJERCICIOS PROPUESTOS

1. Escribir un programa en Java donde se solicite al usuario ingresar un número entero positivo por teclado, luego calcule la suma de sus dígitos utilizando el ciclo **"while".**
2. Escribir un programa en Java donde el usuario ingrese dos números enteros positivos, el primer número ingresado debe ser menor al segundo número ingresado, mediante el ciclo **"while"**, contar y mostrar cuantos números primos existe entre los dos números.
3. Escribir un programa en Java que solicite al usuario ingresar un número entero positivo N y que genere y muestre los primeros N términos de la secuencia de Fibonacci utilizando el ciclo **"while"**

4. Desarrollar un programa en Java donde el programa elija un número aleatorio entre 1 y 100, el usuario debe adivinar dicho número. El programa debe dar pistas si el número a adivinar es mayor o menor al número ingresado. Para realizar el programa se debe utilizar el ciclo **"do while"**, el programa solo debe dejarte realizar 3 intentos. SI después de los tres intentos no logro adivinar el número, el programa debe mostrar un mensaje que diga usted perdió el juego.

5. Desarrollar un programa en Java donde se pida al usuario un numero por teclado al usuario el número debe ser entero positivo y luego calcule el factorial del número ingresado. Para desarrollar el programa se debe utilizar el ciclo **"do while".**

6. Desarrollar un programa en Java que permita al usuario convertir entre diferentes unidades de longitud mediante un menú, el usuario debe ingresar la opción a desarrollar, todos los datos deben ser ingresados por teclado. El programa se termina cuando el usuario decida ya no continuar. Para desarrollar el programa se debe utilizar el ciclo **do while.**

CAPITULO 8

PROGRAMACIÓN ORIENTADA A OBJETOS

8.1. PROGRAMACIÓN ORIENTADA A OBJETOS

Hoy en día en desarrollo de aplicaciones ha evolucionado a pasos gigantescos, que la programación estructurada a quedado en segundo plano, y lo que se pretende lograr es el diseño de sistemas que estén integrados por componentes independientes, que son diseñados y probados antes de ser incorporados en un solo proyecto global, sencillo y fácil construcción. Todo esto con la única idea que si se necesita cambiar algún modulo o componente solo se cambia o se remplaza el módulo necesario sin tocar el resto de la aplicación. Este tipo de desarrollo es denominado como desarrollo modular, lo que se busca con este tipo de implementación es la reutilización del código, componentes o módulos de la aplicación, como lo hacen los distintos lenguajes de programación con sus bibliotecas.

En base a la programación estructura y su complejidad al desarrollar aplicaciones nace lo que hoy en día es la Programación Orientada a Objetos (POO), como una manera de facilitar al programador en sus tareas de desarrollo.

Para tener una buena práctica de la programación orientada a objetos, hay que enfocarse que el desarrollo de las aplicaciones debe ser modular, esto quiere decir que un problema complejo se debe dividir en distantitas partes aplicando el principio de **abstracción,** que se basa en obtener la información esencial de los objetos y cosas simples para representar la complejidad (Garnica, s.f.).

La Programación Orientada a Objetos (POO) es un paradigma de la programación, que está basada en la programación de objetos, esto implica que vamos a envolver sus características y comportamientos relacionados entre ellos.

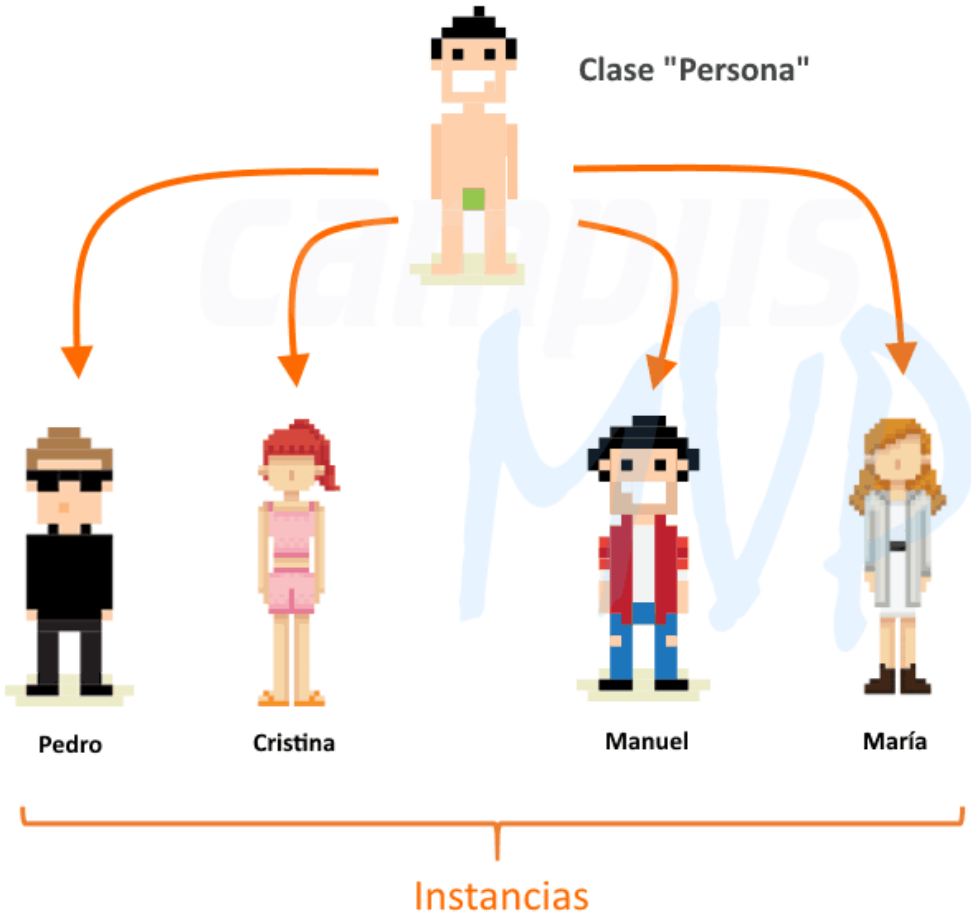
8.2. OBJETOS Y CLASES.

¿Nos podemos imaginar realizar un programa con cada uno de los objetos que nos rodea? Pues si eso es posible con la **Programación Orientada a Objetos,** podemos programar cualquier objeto que tenemos a nuestro alrededor, dándole sus características y su funcionalidad.

Los objetos son como las piezas de un rompecabezas en una computadora. Imagina que cada objeto es como una pieza individual que representa algo real o abstracto, como una persona, un carro o una idea. Estos objetos se crean siguiendo un patrón llamado clase, que es como un molde para crear objetos. Las clases nos ayudan a definir las características y acciones de los objetos.

Las características de un objeto se llaman atributos, y son como las propiedades que tiene un objeto. Por ejemplo, si hablamos de un objeto que representa a una persona, sus atributos podrían ser su nombre, su peso su velocidad. Por otro lado, las acciones que un objeto puede realizar se llaman comportamientos. Siguiendo con el ejemplo de persona, los comportamientos podrían ser hablar, correr o andar en bicicleta.

Figura 43.
Objeto tipo Persona

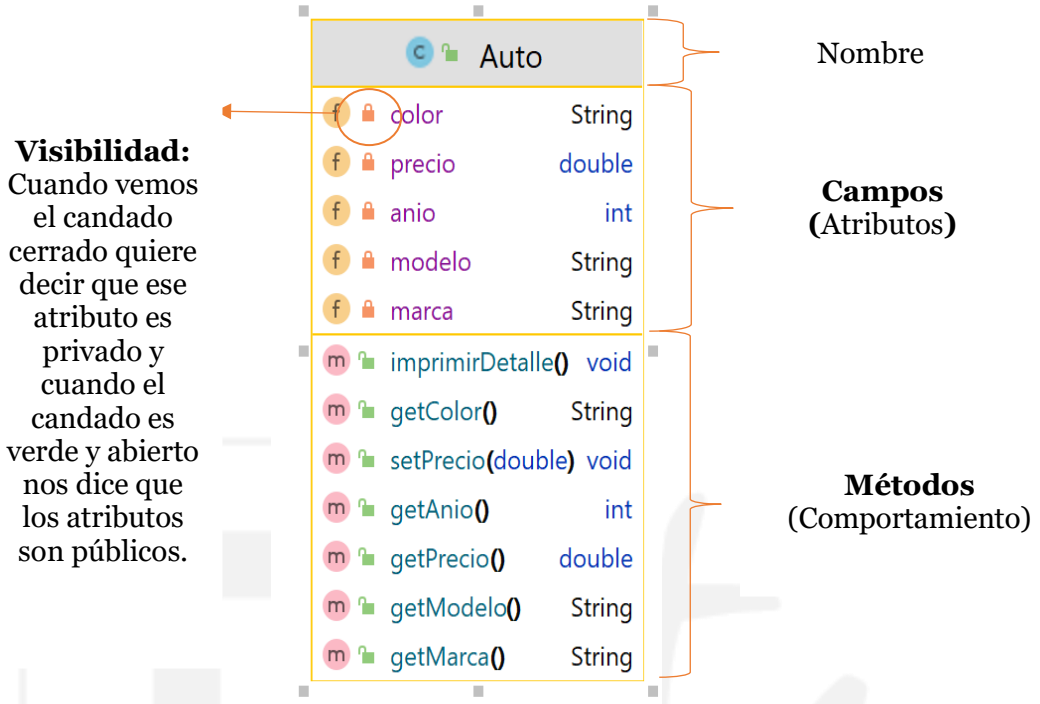


Nota: La imagen representa al objeto Persona que al principio tiene una clase de tipo persona y se instancia en distintas Personas. Para más información consultar la siguiente información. https://www.campusmvp.es/recursos/post/los-conceptos-fundamentales-sobre-programacion-orientada-objetos-explicados-de-manera-simple.aspx.

Para entender con más detalle el concepto de **Programación Orientada a Objetos (POO)** haremos un ejemplo más detallado.

De seguro te gustan los autos verdad vamos a ver cómo podemos mediante POO explicar de mejor manera este tipo de programación.

Figura 44.
Diagrama de clase UML



Nota: Esto es un diagrama de la clase UML. Encontraremos más diagramas en el libro. Aquí se está explicando cada uno de los componentes de la clase tipo Auto.

Digamos que tienes un auto este auto de marca **Chevrolet. Chevrolet** es una instancia de nuestra clase **Auto.** Cada auto tiene varios atributos que son estándar: color, precio, año, modelo, marca, estos son los atributos de mi clase **Auto**.

Además de los atributos, los autos tienen similares comportamientos, en nuestro tenemos **imprimirDetalles, getColor,** entre otros métodos de la clase. Toda la información que se guarda dentro de los atributos suele denominarse estados, y todos los métodos del objeto definen su comportamiento.

Figura 45.
Objetos de la clase Auto

Ferrari: Auto
Color: rojo
Precio: 100000
Año:2024
Modelo: SF90 STRADALE
Marca: Ferrari

Audi: Auto
Color: gris
Precio: 65000
Año:2023
Modelo: Audi Q5
Marca: Audi

Nota: La imagen representa a objetos de la instancia de la clase **Auto,** como se observa con una sola clase se puede crear dos objetos del mismo tipo.

Como podemos ver en la Figura 45, Audi el auto de tu amigo, también es una instancia de la clase **Auto.** Este objeto tiene los mismos atributos de Ferrari, la diferencia esta en los valores de los atributos: su color, su precio, año, y modelo.

Por lo tanto, la clase es un molde de n objeto.

8.3 CLASE:

Ya que sabemos el concepto de lo que es un objeto ahora vamos a entrar a el concepto de clase, una clase es un plano o un molde para poder crear un objeto. La clase va a definir los atributos o propiedades del objeto y los métodos va a definir las acciones que puede realizar dicho objeto.

En el amplio mundo de la programación, especialmente en **Java**, las clases son como los cimientos fundamentales. Imagina que estamos diseñando una casa, antes de ponernos a construir la casa real necesitamos los planos, ¿verdad? Bueno, en la **programación orientada a objetos**, una clase es como ese plano. Es una plantilla o un modelo que define como se estructurara un objeto.

8.4 JERARQUIA DE CLASES

Mientras estemos trabajando con una sola clase no vamos a tener problemas. Pero en el mundo real cuando realicemos un software robusto vamos a tener que heredar nuestras clases de otras clases.

La jerarquía de clases en Java es un concepto fundamental en la **POO,** nos va a permitir organizar y estructurar las clases de manera jerárquica, estableciendo relaciones de herencia entre ellas como se muestra en la **figura 46.** Esta jerarquía facilita la reutilización de código y la creación de una estructura modular y coherente en cada una de las aplicaciones que desarrollemos.

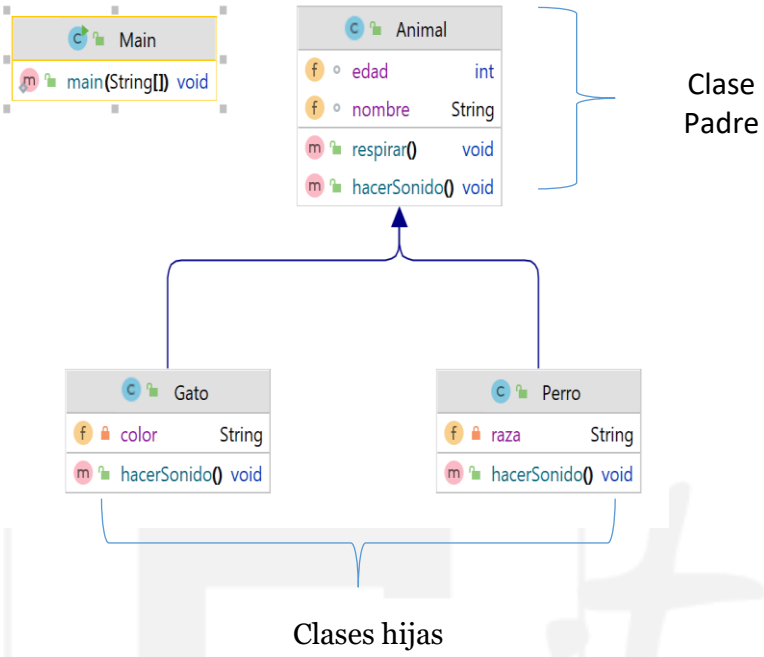
En la POO, una clase puede heredar atributos y métodos de otra clase en Java una clase no puede heredad los atributos o métodos de dos clases, lo que significa que se puede extender funcionalidad al agregar nuevas características o comportamientos específicos. La clase que hereda se conoce como **subclase o clase hija**, mientras que la clase de la que se hereda se llama **super clase o clase padre.**

La relación de herencia entre clases se representa mediante la palabra clave **"extends",** en Java. Al extender una clase, la subclase hereda todos los miembros (atributos y métodos) de la clase padre y se puede agregar nuevos miembros so sobrescribir los existentes según sea necesario.

Como se muestra en **figura 46** tenemos un ejemplo de jerarquía de clases en la cual la **clase padre** es la **clase Animal,** que contiene los atributos y métodos comunes de todos los animales, como: **nombre, edad, hacerSonido() y respirar().** Luego tenemos las subclases o clases hijas **Perro y Gato,** que se extiende la clase **Animal,** y se agregan características específicas para cada una de las clases hijas como: **raza** para el objeto perro y **color** para el objeto **gato.**

La jerarquía de las clases en Java proporciona una formula intuitiva y flexible de organizar el código fuente, permitiendo la creación de aplicaciones complejas con estructura modular y muy fácil de mantener y permitiendo a la reutilización del código.

Figura 46
Diagrama UML jerarquía de clases.

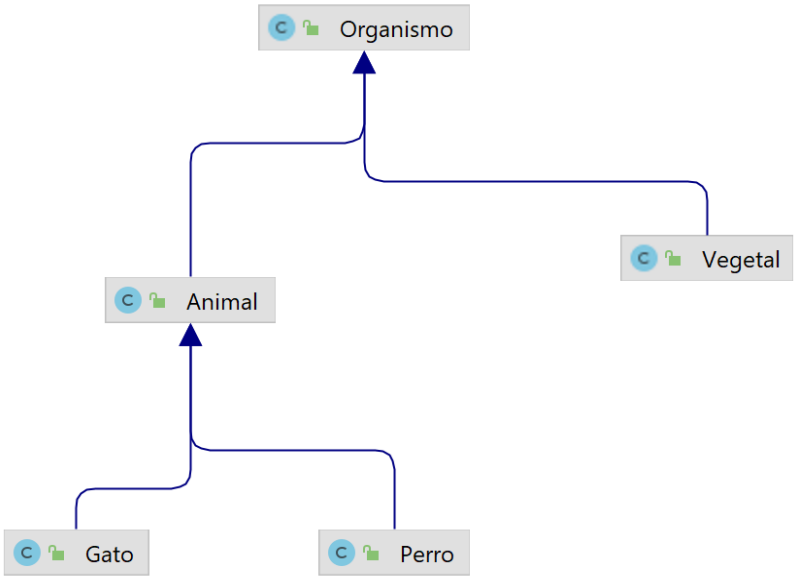


Nota: La figura representa la jerarquía de clase en un diagrama UML. Donde la clase padreo superclase es la clase Animal y las clases hijas son Gato y Perro estas dos clases heredan atributos de la clase padre.

Vamos un poco más allá digamos que tenemos que realizar una tarea crear una clase **Organismo,** esta es nuestra super clase, luego tendremos dos subclases que van a heredad los atributos de la clase padre, estas clases son: **Animal** y **Plantas** y a su vez la clase **Animal** será una clase padre de las clases **Gato y Perro.**

Implementando el concepto de jerarquía de clases nuestro diagrama quedaría de la siguiente manera.

Figura 47.
Diagrama UML



Nota: La figura representa un diagrama UML las clases se pueden simplificar si es más importante mostrar sus relaciones que sus contenidos.

Las subclases pueden sobrescribir el comportamiento de los métodos que heredan de las clases padres. Una subclase puede sustituir completamente el comportamiento por efecto o limitarse a mejorarlo con información adicional.

8.5. PILARES FUNDAMENTALES DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

Cuando empezamos a desarrollar aplicaciones basándonos en la POO debemos conocer 4 pilares fundamentales que proporcionan un marco sólido para el diseño y desarrollo de software. Estos pilares son: **encapsulamiento, abstracción, herencia y polimorfismo.** Cada uno de estos conceptos lo estudiaremos de manera teórica y práctica.

8.5.1. Encapsulamiento

Encapsulación o encapsulamiento significa reunir en cierta estructura todos los elementos que, a determinado nivel de abstracción, se pueden considerar de una misma entidad, y es el proceso de agrupamiento de datos y operaciones relacionadas bajo una misma unidad de programación, lo que aumenta la cohesión de los componentes del sistema.

En este caso, los objetos que poseen las mismas características y comportamiento se agrupan en clases que son unidades de programación que encapsulan datos y

operaciones; la encapsulación oculta lo que hace un objeto de lo que hacen otros objetos del mundo exterior, por lo que se denomina también ocultación de datos. Un objeto tiene que presentar una "cara" al mundo exterior, de modo que pueda iniciar operaciones; la televisión tiene un conjunto de botones, bien en ella misma o incorporados en un control remoto. Una máquina lavadora tiene un conjunto de mandos e indicadores que establecen la temperatura y el nivel del agua. Los botones de la TV y los mandos de la lavadora constituyen la comunicación con el mundo exterior, las interfaces.

En esencia, la interfaz de una clase representa un contrato de prestación de servicios entre ella y los demás componentes del sistema; de este modo, los clientes de un componente sólo necesitan conocer los servicios que éste ofrece y no cómo están implementados internamente. Por consiguiente, se puede modificar la implementación de una clase sin afectar a las restantes relacionadas con ella. Existe una separación entre la interfaz y la implementación: la primera establece qué se puede hacer con el objeto; de hecho, la clase actúa como una caja negra; es estable, la implementación se puede modificar.

Otro ejemplo lo encontramos en los automóviles: no se necesita conocer el funcionamiento de la caja de cambios, el sistema de frenos o la climatización para que el conductor utilice todos estos dispositivos. (Joyanes Aguilar & Zahonero Martinez, Programación en Java 6, 2011)

En resumen, el encapsulamiento es ocultar información de nuestra aplicación que puede ser cambiado y al ser cambiado puede causar un mal funcionamiento de nuestra aplicación.

8.5.2. ABSTRACCIÓN.

La abstracción se refiere a la capacidad de representar objetos del mundo real como modelos simplificados en el software. Por ejemplo, si está desarrollando un sistema de gestión de bibliotecas, puede abstraer un libro como un objeto con atributos, como título, autor y año de publicación, y métodos como prestar y devolver. Entonces, la abstracción nos permite centrarnos en las partes esenciales de un objeto y ocultar lo que no es necesario.

8.5.3. HERENCIA.

Una de las formas en que entendemos la herencia, son las características que se mantienen de generación en generación en una familia. Por ejemplo, los abuelos tenían cabello rubio, y uno de los hijos hereda esta característica, a su vez el hijo tiene a su vez un hijo que también hereda el cabello rubio. Por otro lado, los mismos abuelos pudieron haber tenido una hija con cabello castaño y lacio, y los nietos pueden heredar también

esta misma característica de tener el cabello castaño y lacio. En la programación orientada a objetos, el concepto de herencia es exactamente igual. Definiremos una jerarquía de clases que nos permitirán heredar características entre clases Padre y clases Hijas.

La herencia es uno de los conceptos más importantes en la programación orientada a objetos (POO), y mencionaremos algunas de las razones más importantes por la cuales aplicaremos este concepto en el diseño de nuestras clases.

- La herencia nos permitirá representar características o comportamiento en común entre clases, permitiendo definir en la clase Padre los atributos o métodos que sean comunes a las clases hijas, las cuales heredarán estos atributos o métodos definidos en la clase Padre.
- Lo anterior permite evitar duplicar el código entre la clase Padre y las clases Hijas, por lo que cumplimos con la reutilización de código que es uno de los principales objetivos de la POO.
- Si aplicamos el concepto de Herencia, podemos modelar de una manera más sencilla sistemas del mundo real. Si ya la definición de objetos es un gran aporte, diseñar una jerarquía de clases, nos facilitará aún más el modelado de clases.

8.5.3.1. SINTAXIS HERENCIA EN JAVA.

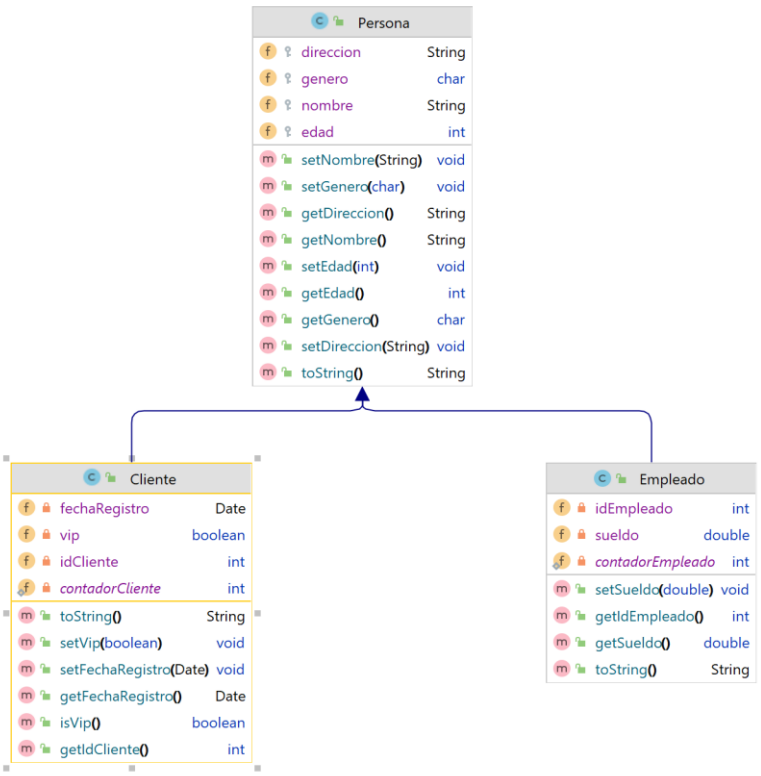
Cuando utilizamos herencia en nuestras aplicaciones vamos a empezar a crear nuestra clase padre y luego vamos creando las clases hijas.

Ejercicio 8.1

Dado el siguiente diagrama de clases UML crear un programa en Java, donde se aplique los conceptos vistos sobre herencia.

Escribir un programa en Java donde se cree una Clase padre **Persona,** con dos clases hijas las cuales van a heredar atributos y métodos, las clases hijas serán. **Cliente** y **Empleado.** Para ejecutar el programa se creará una clase **TestHerencia** imprimir en consola los datos del objeto **Cliente** y el objeto **Empleado.**

Figura 48.
Diagrama UML aplicando Herencia



Nota: El diagrama UML nos muestra como dos clases hijas pueden heredad atributos y métodos de una clase padre.

Clase Persona

```
package org.elvis.domain;

public class Persona {
    //Declaramos las variables de la clase persona
    //que heredaran las clases hijas
    //los atributos son  de tipo protected para que pueden
    //ser heredados desde las clases hijas
    protected String nombre;
    protected char genero;
    protected int edad;
    protected String direccion;
    //Implementamos un contructor vacio
    public Persona() {
    }
    //Implementamos un constructor que recibe
    //e inicializa el argumento de nombre
    public Persona(String nombre) {
        this.nombre = nombre;
    }
    //Implementamos un contrcutor que recibe
    //e inicializa todos los parametros de la clase padre

    public Persona(String nombre, char genero, int edad, String
direccion) {
        this.nombre = nombre;
```

```
        this.genero = genero;
        this.edad = edad;
        this.direccion = direccion;
    }
    //Implementamso los métodos get y set
    public String getNombre() {
        return nombre;
    }

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

    public char getGenero() {
        return genero;
    }

    public void setGenero(char genero) {
        this.genero = genero;
    }

    public int getEdad() {
        return edad;
    }

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

    public String getDireccion() {
        return direccion;
    }

    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }
    //Sobre escribimos el método toString
    //donde se imprime el estado del objeto
    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder("Persona{");
        sb.append("nombre='").append(nombre).append('\'');
        sb.append(", genero=").append(genero);
        sb.append(", edad=").append(edad);
        sb.append(", direccion='").append(direccion).append('\'');
        sb.append(", ").append(super.toString());
        sb.append('}');
        return sb.toString();
    }
}
```

Explicación del código de la clase Persona.

La clase Persona, es una clase padre la cual tiene algunos atributos de tipo **protected**, por qué en nuestras clases hijas se necesita heredar estos atributos, los atributos para la clase padre son:

Tabla 7.
Argumentos clase Persona

Modificador de acceso	Tipo de argumento	Nombre del argumento
protected	String	nombre
protected	char	genero
protected	int	edad
protected	String	dirección

Nota: La tabla representa los atributos de la clase **Persona** que serán heredadas por las clases hijas.

Después de declarar los argumentos vamos a implementar los **constructores,** un constructor es un método que lleva el mismo nombre de la clase y sirve para inicializar los valores de un objeto, en el caso nuestro vamos a sobrecargar nuestro **constructor.** El tema de constructores lo explicaremos más adelante.

Constructor vacio

Este constructor no inicializa ningún valor.

```
    //Implementamos un contructor vacio
    public Persona() {
    }
```

Constructor inicializando un valor

Este constructor inicializa a la variable **nombre,** este valor cambiara dependiendo del valor que reciba**.**

```
//Implementamos un constructor que recibe
    //e inicializa el argumento de nombre
    public Persona(String nombre) {
        this.nombre = nombre;
    }
```

Constructor que inicializa distintos valores.

Este constructor va a inicializar todos los argumentos de la clase **Persona.**

```
//Implementamos un contrcutor que recibe
    //e inicializa todos los parametros de la clase padre

    public Persona(String nombre, char genero, int edad, String
direccion) {
```

```
        this.nombre = nombre;
        this.genero = genero;
        this.edad = edad;
        this.direccion = direccion;
    }
```

La palabra reservada **this** la usamos para hacer referencia a los miembros de la clase y no para los parámetros del constructor de esa manera no nos confundimos a que parámetro estamos haciendo referencia.

Métodos get and set

En la **POO** los métodos **get** y **set** son esenciales cuando programamos objetos, ya que estos nos permiten obtener información y setear información.

```
//Implementamso los métodos get y set
    public String getNombre() {
        return nombre;
    }

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

    public char getGenero() {
        return genero;
    }

    public void setGenero(char genero) {
        this.genero = genero;
    }

    public int getEdad() {
        return edad;
    }

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

    public String getDireccion() {
        return direccion;
    }
```

```
    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }
```

Por último tenemos que sobrescribir el método **toString(),** este método ya está programado en **Java,** lo que hacemos es sobrescribir este método para poder obtener los valores del objeto. Más adelante estudiaremos todos estos temas a profundidad.

```
//Sobre escribimos el método toString
    //donde se imprime el estado del objeto
    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder("Persona{");
        sb.append("nombre='").append(nombre).append('\'');
        sb.append(", genero=").append(genero);
        sb.append(", edad=").append(edad);
        sb.append(", direccion='").append(direccion).append('\'');
        sb.append(", ").append(super.toString());
        sb.append('}');
        return sb.toString();
    }
```

Una vez programado nuestra clase padre de nombre **Persona,** implementamos las clase hijas las que heredaran los atributos y métodos de la clase principal.

Clase Empleado

Esta clase tendrá sus propios atributos y métodos.

```
package org.elvis.domain;
/*
* Utilizamos la palabra reservada extends la que en Java
* nos permite decir que esta clase está heredando de otra clase
* */
public class Empleado extends Persona{
    /*Declaranos los atributos de la clase
    * Empleado de tipo private los declaramos de esta manera
    * ya que esta clase no sera una clase padre*/
    private int idEmpleado;
    private double sueldo;
    private static int contadorEmpleado;
    /*Implementamos un constructor con parametros propios de la clase
    y parametros herados de la clase padre
    */
    public Empleado(String nombre, double sueldo) {
        super(nombre);
        this.idEmpleado=++Empleado.contadorEmpleado;
        this.sueldo = sueldo;
    }
    /*Implementamos lo métodos get y set*/
```

```
    public int getIdEmpleado() {
        return idEmpleado;
    }

    public double getSueldo() {
        return sueldo;
    }

    public void setSueldo(double sueldo) {
        this.sueldo = sueldo;
    }
    /*Implementamos el método toString() para obtner el
    * estado del objeto Empleado propios y heredados*/

    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder("Empleado{");
        sb.append("idEmpleado=").append(this.idEmpleado);
        sb.append(", ").append(super.toString());
        sb.append(", sueldo=").append(this.sueldo);
        sb.append('}');
        return sb.toString();
    }
}
```

Explicación del código.

En la clase hija de nombre **Empleado,** primero para saber que esta clase esta heredando una clase padre usamos la palabra reservada **extends,** esto me permite decirle a Java que estamos heredando de otra clase atributos y métodos que se puedan heredar.

```
public class Empleado extends Persona{
```

Declaramos los atributos propios de la clase en ese caso ya no son de tipo **protected** sino de tipo **private,** ya que esta clase no heredara los atributos a otra clase.

```
/*Declaranos los atributos de la clase
    * Empleado de tipo private los declaramos de esta manera
    * ya que esta clase no sera una clase padre*/
    private int idEmpleado;
    private double sueldo;
    private static int contadorEmpleado;
```

Tabla 8.
Atributos clase Empleado

Modificador de Acceso	Tipo de variable	Nombre
private	int	idEmpleado
private	double	sueldo

private	int	contadorEmpleado

Nota: La tabla representa a los atributos con sus respectivos modificadores de acceso tipo de dato y nombre de la clase Empleado.

Luego de declarar las variables de nuestra clase implementamos el constructor aquí inicializamos los atributos del objeto en este caso tenemos un constructor con parámetros el cual recibe de la clase padre y de su propia clase ejemplo. En este constructor inicializamos el parámetro **nombre** este parámetro no lo tenemos en nuestra clase **Empleado,** pero lo vamos a heredad de nuestra clase padre. Para poder heredar este atributo utilizamos la palabra reservada **super,** que se utiliza para invocar al constructor de una superclase o clase padre en nuestro ejemplo la clase **Persona** recordaran que tenemos un constructor que solo inicializa el nombre, y es útil cuando nuestra clase hija necesita inicializar parte de la super clase.

```
/*Implementamos un constructor con parametros propios de la clase
y parametros herados de la clase padre
*/
public Empleado(String nombre, double sueldo) {
    super(nombre);
    this.idEmpleado=++Empleado.contadorEmpleado;
    this.sueldo = sueldo;
}
```

Implementamos los métodos get y set para poder obtener la información y modificarlas.

```
/*Implementamos lo métodos get y set*/

public int getIdEmpleado() {
    return idEmpleado;
}

public double getSueldo() {
    return sueldo;
}

public void setSueldo(double sueldo) {
    this.sueldo = sueldo;
}
```

Por último, sobrescribimos el método **toString,** para obtener el estado del objeto **Empleado.**

```
@Override
public String toString() {
    final StringBuilder sb = new StringBuilder("Empleado{");
    sb.append("idEmpleado=").append(this.idEmpleado);
    sb.append(", ").append(super.toString());
    sb.append(", sueldo=").append(this.sueldo);
    sb.append('}');
    return sb.toString();
}
```

Clase Main.

Para ejecutar nuestro programa vamos a crear una clase **TestHerencia,** implementando el método **Main,** que nos permite ejecutar el ejercicio.

```
package org.elvis.test;

import org.elvis.domain.Cliente;
import org.elvis.domain.Empleado;

import java.util.Date;

public class TestHerencia {
    public static void main(String[] args) {
        //Instanciamos el objeto de la clase Empleado
        // e inicializamos el constructor con los datos necesario
        Empleado empleado1= new Empleado("Juan",500);
        System.out.println("Empleado : " + empleado1);
           }
}
```

El método **main,** permite ejecutar el programa, luego instanciamos la clase **Empleado** dándole un nombre al objeto **empleado1**, enviando los siguientes parámetros **nombre= Juan** y el sueldo 500.

Empleado empleado1= new Empleado("Juan",500);

Por último imprimimos el estado del objeto **empleado1.**

System.*out*.println("Empleado : " + empleado1);

Figura 49.
Ejecución del ejercicio Persona

Nota: La imagen muestra la ejecución del programa donde estamos heredando algunos parámetros de la clase Persona en la clase Empleando.

Clase Cliente

```
package org.elvis.domain;

import java.util.Date;
//La clase cliente hereda atributos de la clase persona
//por medio de la palabra extends
```

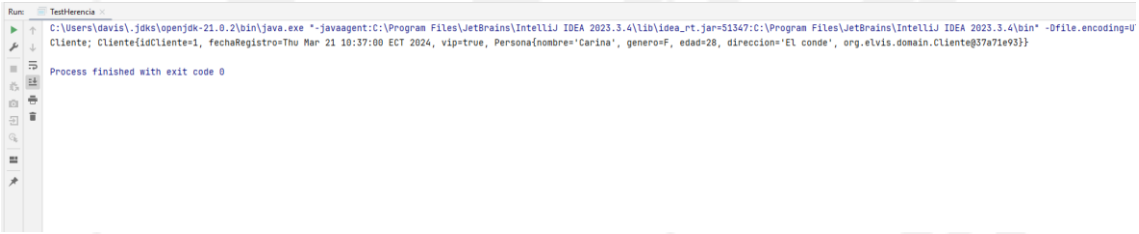
```
public class Cliente extends Persona {
    /*
    * Declaramos los atributos
    * propios de la clase Cliente
    * */
    private int idCliente;
    private Date fechaRegistro;
    private boolean vip;
    private static  int contadorCliente;

    /*implementamos un constructor para inicializar los atributos de
la clase
        y de la clase padre en este caso inicializaremos todos los
atriburtos de la
        clase PAdre mediante el contructor
        Utilizamos la palabra reservada super para decirle que son los
atributos heredados
        de la clase padre
    */

    public Cliente(Date fechaRegistro, boolean vip, String nombre,
                   char genero, int edad, String direccion) {
        super(nombre, genero, edad, direccion);
        this.idCliente = ++Cliente.contadorCliente;
        this.fechaRegistro = fechaRegistro;
        this.vip = vip;
    }
    /*Implementamos los métodos get y set*/

    public int getIdCliente() {
        return idCliente;
    }


    public Date getFechaRegistro() {
        return fechaRegistro;
    }

    public void setFechaRegistro(Date fechaRegistro) {
        this.fechaRegistro = fechaRegistro;
    }

    public boolean isVip() {
        return vip;
    }

    public void setVip(boolean vip) {
        this.vip = vip;
    }
    /*Sobrescribimos el mpetodo toString
    * Para mostrar el estado del objeto
    * Cliente
    */

    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder("Cliente{");
        sb.append("idCliente=").append(idCliente);
        sb.append(", fechaRegistro=").append(fechaRegistro);
        sb.append(", vip=").append(vip);
        sb.append(", ").append(super.toString());
```

```
        sb.append('}');
        return sb.toString();
    }
}
```

Clase Main

```
package org.elvis.test;

import org.elvis.domain.Cliente;
import org.elvis.domain.Empleado;

import java.util.Date;

public class TestHerencia {
    public static void main(String[] args) {
        //Instanciamos el objeto de la clase Cliente
        // y pasamos los valores el constructor con lso datos
necesario
        //Empleado empleado1= new Empleado("Juan",500);
        //System.out.println("Empleado : " + empleado1);
        Date fecha= new Date();
        Cliente cliente = new Cliente(fecha, true, "Carina", 'F', 28,
"El conde");
        System.out.println("Cliente; " + cliente);
    }
}
```

Figura 50.
Ejecución de la clase Main



Nota: Aquí muestra la ejecución del objeto **Cliente,** el cual la clase hereda todos los atributos y método de la clase padre **Persona.**

8.5.4. POLIMORFISMO.

Polimorfismo es la habilidad de ejecutar métodos sintácticamente iguales en tipos distintos.

Un objeto tiene una sola forma y un tipo, y esto nunca cambia durante toda la vida del objeto creado. Sin embargo, una variable de un tipo puede hacer referencia a objetos de diferentes tipos, siempre y cuando haya una relación entre estos tipos, como puede ser una relación de herencia (Clase padre hijo).

Una variable de tipo de una clase padre puede almacenar referencias de clases hijas o del mismo tipo de la clase padre, y mandar a llamar los métodos que tiene en común utilizando polimorfismo, es decir, ejecuta los métodos de la clase hija, con la variable de tipo de la clase padre.

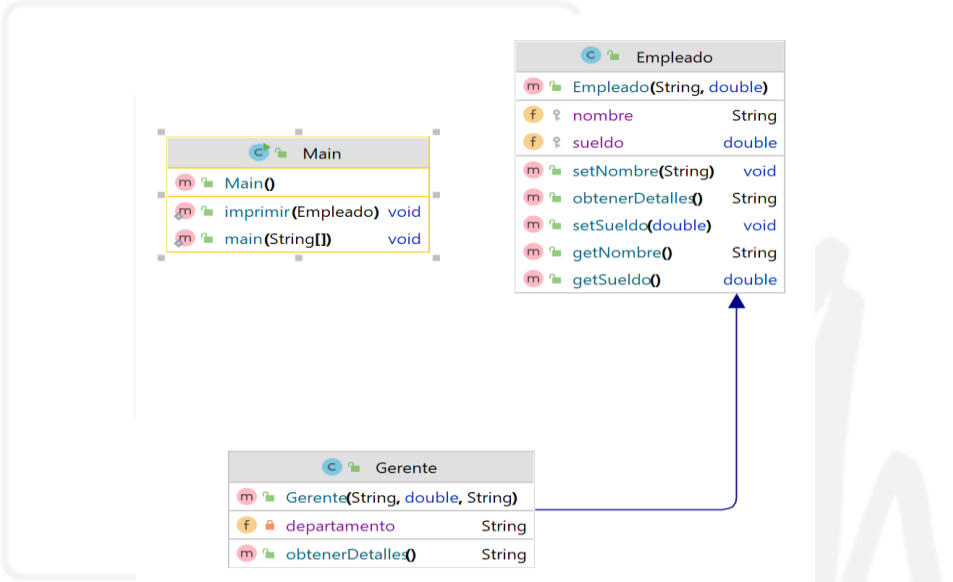
Esto es precisamente el concepto de polimorfismo. La importancia del polimorfismo es que podemos generalizar un método que reciba distintos tipos en la jerarquía de clases definida (clases padre e hijas), por ejemplo, un método que imprima los detalles de cada clase, sin importar si es un objeto es de tipo padre o de tipo hijo, y todo esto, con una variable de tipo padre.

Una vez entendido el concepto de polimorfismo vamos a la parte práctica.

Ejercicio 8.2.

Dado el siguiente diagrama de clases UML. Desarrollar una clase **Empleado** y una clase **Gerente.** Escribir un programa en Java donde se aplique el concepto de polimorfismo.

Figura 51.
Diagrama UML polimorfismo



Nota: El diagrama UML representa una clase padre y una clase hija la cual se va aplicar el concepto de polimorfismo.

Clase Empleado.

```
package org.elvis.domain;

public class Empleado {
    //declaramos los atributos de la clase
    //y poder heredarlos
    protected String nombre;
    protected double sueldo;

    public Empleado(String nombre, double sueldo) {
        this.nombre = nombre;
        this.sueldo = sueldo;
    }

    public String obtenerDetalles(){
        return "Nombre: " +this.nombre + ", sueldo: "+ this.sueldo;
```

```
    }

    public String getNombre() {
        return nombre;
    }

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

    public double getSueldo() {
        return sueldo;
    }

    public void setSueldo(double sueldo) {
        this.sueldo = sueldo;
    }
}
```

Explicación del código clase padre Empleado.

La clase **Empleado,** tiene los siguientes atributos que son declarados de tipo **protected,** para que los atributos puedan ser heredados desde las clases hijas que se implementen.

Tabla 9.
Atributos clase Empleado

Modificador de acceso	Tipo de variable	Nombre
protected	String	Nombre
protected	double	sueldo

Nota: La tabla representa los atributos que tiene la clase padre detallando el tipo de acceso, el tipo de variable y el nombre de la variable.

Luego de declarar todos los atributos, implementamos los constructores en este caso solo un constructor que va a recibir dos parámetros, uno de tipo **String nombre** y el otro **double sueldo**. El constructor va a inicializar el atributo nombre y sueldo.

```
public Empleado(String nombre, double sueldo) {
    this.nombre = nombre;
    this.sueldo = sueldo;
}
```

Luego implementamos un método **obtenerDetalles,** este método va a devolvernos el estado de nuestro objeto empleado que la clase hija se podrá sobrescribir dicho método para obtener el estado de los dos objetos.

```
public String obtenerDetalles(){
    return "Nombre: " +this.nombre + ", sueldo: "+ this.sueldo;
}
```

Y por último implementamos los métodos **set** y **get,** como ya lo hemos dicho estos métodos me permiten setear los parámetros y obtener los datos de los atributos.

```
public String getNombre() {
    return nombre;
}

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

public double getSueldo() {
    return sueldo;
}

public void setSueldo(double sueldo) {
    this.sueldo = sueldo;
}
```

Clase hija Gerente.

La clase gerente es la clase hija la que va a heredar los atributos y métodos de la clase padre.

```
package org.elvis.domain;

public class Gerente extends Empleado{
    private String departamento;

    public Gerente(String nombre, double sueldo, String departamento)
{
        super(nombre, sueldo);
        this.departamento=departamento;
    }

    @Override
    public String obtenerDetalles(){
        return super.obtenerDetalles() + ",
departamento:"+this.departamento;
    }



}
```

En la clase hija lo primer que hacemos es heredar con la palabra **extends** los atributos y métodos de la clase padre **Empleado,** luego se declara los atributos propios de la clase con sus respectivos modificadores de acceso. Para la clase hija los modificadores de acceso serán de tipo **privado** estos atributos no podrán ser manipulados ni heredados desde otra clase.

Tabla 10.
Atributos clase Gerente

Modificador de acceso	Tipo de variable	Nombre
private	String	departamento

Nota: En la tabla tenemos los atributos propios de la clase **Gerente** que solo se podrán utilizar dentro de la clase.

Luego implementamos el método constructor donde inicializaremos los atributos de la clase y heredaremos los atributos de la clase padre con la palabra **super,** en esta clase hija hereda los atributos **nombre y sueldo.**

```
public Gerente(String nombre, double sueldo, String departamento) {
    super(nombre, sueldo);
    this.departamento=departamento;
}
```

Y por último sobrescribimos el método **obtenerDetalles**, que se hereda de la clase padre y se obtiene el estado del objeto **Gerente.**

```
@Override
public String obtenerDetalles(){
    return super.obtenerDetalles() + ",
departamento:"+this.departamento;
}
```

Clase Main

```
package org.elvis.test;

import org.elvis.domain.Empleado;
import org.elvis.domain.Gerente;

public class Main {
    public static void main(String[] args) {
        Empleado empleado = new Empleado("Carina", 250);
        imprimir(empleado);
        Gerente gerente1= new Gerente("Elvis", 125.00 ,"Desarrollo" );
        imprimir(gerente1);
    }

    public static void imprimir(Empleado empleado){
        String detalles = empleado.obtenerDetalles();
        System.out.println("empleado " + detalles);
    }
}
```

La clase **Main** contiene el método main que es necesario en cualquier programa de Java, este método te permite ejecutar cualquier programa en Java sin este método el programa no se podrá ejecutar.

La importancia del polimorfismo del polimorfismo es que podemos generalizar un método que reciba distintos tipos en la jerarquía de clases definidas (clase padre e hijas),

por ejemplo, un método que imprima los detalles de cada clase, sin importar si un objeto es de tipo padre o tipo hijo, y todo esto con una variable de tipo padre.

```
public static void imprimir(Empleado empleado){
    String detalles = empleado.obtenerDetalles();
    System.out.println("empleado " + detalles);
}
```

Como podemos observar dentro del método **Main**, en nuestro código creamos una variable de tipo **Empleado** instanciando el objeto de la clase llamando **empleado.** Esta variable se almacena en la memoria stack, ya que es una variable local. Y a la variable **empleado,** se le asigna una referencia de un objeto de tipo padre, es decir de tipo **Empleado**, con el atributo de nombre **Carina**, y el atributo sueldo de este empleado es de $**250.** Y cuando se manda a llamar al método **obtenerDetalles** con la variable **empleado,** el método que se ejecuta es el de la clase **Empleado.**

```
Empleado empleado = new Empleado("Carina", 250);
imprimir(empleado);
```

Hasta aquí nada fuera de lo normal ni nada nuevo, ya que estamos llamando a un método del mismo tipo de la clase Padre. Sin embargo, luego con la misma variable de tipo padre, le asignamos una referencia de tipo hijo es decir la clase **Gerente** con el atributo de nombre **Elvis,** sueldo **$125** y posteriormente llamamos al método **imprimir.**

```
Gerente gerente1= new Gerente("Elvis", 125.00 ,"Desarrollo" );
imprimir(gerente1);
```

De esta forma hemos estudiado los 4 pilares que engloba la **POO.** NO te olvides de realizar los ejercicios de la parte final de este capítulo.

RESUMEN CAPITULO 8

En el vibrante universo de la programación, nos adentramos en un fascinante viaje hacia la Programación Orientada a Objetos **POO**, donde los cimientos del código se moldean a través de objetos y clases. Esta travesía nos revela un enfoque tan ingenioso como artísticos para dar forma a nuestras creacioones digitales. Estudiando los 4 pilares fundamentales de la POO, como los objetos que son los cimientos de este tipo de programación, estos nos son simplemente líneas de código, sino son entidades vivas que se emulan aspectos del mundo real. Cada uno de los objetos ya sea un avión, una persona o un animal, cada uno de estos objetos poseen una identidad única, cada uno de los objetos tiene sus propias características y habilidades.

Abstracción: La abstracción es una herramienta poderosa y esencial de cualquier programador. Nos permite simplificar la complejidad, crear conceptos reutilizables y construir un enlace entre el mundo real y el digital.

Encapsulamiento. En todo nuestro capitulo, hemos descubierto el concepto de encapsulamiento, que nos permite ocultar los detalles internos de nuestros objetos poniendo solo a flote la información esencial. En poca palabra encapsular nos permite resguardar la información de nuestras aplicaciones.

Herencia. Uno de los conceptos fascinantes de la **POO** es el concepto de herencia, este concepto nos permite que las clases hijas hereden atributos y métodos de una clase padre. Una de las dificultades de herencia en Java es que no permite herencia múltiple eso quiere decir que solo se puede heredar los atributos de una sola clase padre.

Polimorfismo. El concepto de polimorfismo es donde los objetos cambian de forma y función según el contexto. Este concepto es como presenciar la metamorfosis de una mariposa donde un objeto puede adoptar múltiples roles.

EJERCICIOS PROPUESTOS

1. Escribir un programa en Java donde se implemente una clase Libro con los siguientes atributos.
    - ISBN.
    - Titulo.
    - Autor.
    - Número de páginas.

    Crear sus respectivos métodos get y sets correspondientes para cada atributo. Crear un método **toString** para mostrar la información relativa al libro con el siguiente formato:
    - El libro con ISBN creado por el autor tiene páginas.
    - En la clase Main, crear 3 objetos Libro (los atributos que quieran) y mostrarlos por pantalla.
    - Indicar cuál de los 3 objetos tiene más páginas.
    - Todos los datos deben ser ingresados por teclado.
2. Vamos a realizar una clase llamada Raíces, donde representaremos los valores de una ecuación de 2º grado. Tendremos los 3 coeficientes como atributos, llamémosles a, b y c. Hay que insertar estos 3 valores para construir el objeto. Las operaciones que se podrán hacer son las siguientes:
    - obtenerRaices(): imprime las 2 posibles soluciones.
    - obtenerRaiz(): imprime única raíz, que será cuando solo tenga una solución posible.
    - getDiscriminante(): devuelve el valor del discriminante (double), el discriminante tiene la siguiente formula (b^2)-4*a*c.
    - tieneRaices(): devuelve un booleano indicando si tiene dos soluciones, para que esto ocurra el discriminante debe ser mayor o igual que 0
    - tieneRaiz(): devuelve un booleano indicando si tiene una única solución, para que esto ocurra, el discriminante debe ser igual que 0.
    - calcular(): mostrara por consola las posibles soluciones que tiene nuestra ecuación, en caso de no existir solución, mostrarlo también.

- Para realizar el cálculo el usuario debe ingresar todos los datos por teclado.

3. Realizar un programa en Java donde se desea representar con programación orientada a objetos, un aula con estudiantes y un profesor.

   Tanto de los estudiantes como de los profesores necesitamos saber su nombre, edad y sexo. De los estudiantes, queremos saber también su calificación actual (entre 0 y 10) y del profesor que materia da.
   Las materias disponibles son matemáticas, filosofía y física.
   Los estudiantes tendrán un 50% de hacer novillos, por lo que si hacen novillos no van a clase pero aunque no vayan quedara registrado en el aula (como que cada uno tiene su sitio).
   El profesor tiene un 20% de no encontrarse disponible (reuniones, baja, etc.)
   Las dos operaciones anteriores deben llamarse igual en Estudiante y Profesor (polimorfismo).
   El aula debe tener un identificador numérico, el número máximo de estudiantes y para que esta destinada (matemáticas, filosofía o física). Piensa que más atributos necesita.
   Un aula para que se pueda dar clase necesita que el profesor esté disponible, que el profesor de la materia correspondiente en el aula correspondiente (un profesor de filosofía no puede dar en un aula de matemáticas) y que haya más del 50% de alumnos.
   El objetivo es crear un aula de alumnos y un profesor y determinar si puede darse clase, teniendo en cuenta las condiciones antes dichas.
   Si se puede dar clase mostrar cuantos alumnos y alumnas (por separado) están aprobados de momento (imaginad que les están entregando las notas).
   NOTA: Los datos pueden ser aleatorios (nombres, edad, calificaciones, etc.) siempre y cuando tengan sentido (edad no puede ser 80 en un estudiante o calificación ser 12 el rango de calificaciones debe ser de 0 a 10).

CAPÍTULO 9

ARREGLOS (ARRAYS)

9.1. ARREGLOS

Un arreglo en programación es una estructura de datos del mismo tipo que se llama elementos y pueden ser datos simples en Java, esta estructura de datos nos permite almacenar múltiples valores siempre y cuando sean del mismo tipo en una sola entidad o variable. A diferencia de las variables ordinarias m que pueden contener un solo valor a la vez, los arreglos nos permiten manejar colecciones de datos de manera eficiente. Esta capacidad es crucial para desarrollar aplicaciones complejas que necesiten manipular grandes conjuntos de datos de forma estructurada.

Un arreglo puede contener, por ejemplo, la estatura de cada uno de los alumnos de una clase, las temperaturas de cada día durante un mes de una determinada ciudad en este caso la ciudad de Quito, también se puede guardar el de personas que residen en las 24 provincias del Ecuador, cada uno de estos ítems se denomina elemento.

Los elementos en arreglo se enumeran consecutivamente 0, 1, 2, 3, 4, ... Estos elementos se denominan índices y siempre un arreglo empieza desde el índice 0. Estos números sirven para localizar la posición de cada elemento dentro del arreglo, que permite proporcionar un acceso directo a este.

Si el arreglo se llama a, entonces para acceder al primer elemento seria en esta posición a [0], y si queremos acceder al segundo elemento seria en esta posición a [1].

El arreglo **a** tiene 6 elementos: a [0] contiene 18, a [1] contiene 17, a [2] contiene 15, a [3] contiene 16, a [4] contiene 14, a [5] contiene 17; el diagrama de la **tabla 11** representa realmente una región de la memoria de la computadora, ya que un arreglo es un espacio de memoria donde se puede guardar distintos números de datos del mismo tipo. En Java los índices de un arreglo siempre tienen como límite inferior 0 y como límite superior el tamaño del arreglo -1.

Tabla 11.
Arreglo de 6 elementos

a	**18**	**17**	**15**	**16**	**14**	**17**
	0	1	2	3	4	5

Nota: La tabla representa un arreglo con 6 elementos con sus respectivos índices.

9.1.1. DECLARACIÓN DE UN ARREGLO

En Java, un arreglo se declara especificando el tipo de datos que almacenará, seguido de corchetes (‘[]’) para indicar que se trata de un arreglo. La creación del arreglo se realiza utilizando la palabra reservada **‘new’,** seguida por el tipo de datos y el tamaño del arreglo entre corchetes. Es importante destacar que, una vez creado, el tamaño del arreglo es fijo y no se puede cambiar.

Se declara el módulo similar a otros tipos de datos, excepto que en este caso se debe indicar al compilador que es un arreglo y esto se hace con corchetes.

```
int [] v;
float a[];
```

Los corchetes se pueden colocar de dos maneras.

- A continuación del tipo de dato
- Después del nombre del arreglo.

La sintaxis para la declaración de un arreglo en Java es.

```
tipo [] identificador;
tipo identificador[];
```

9.1.2. CREACIÓN DE UN ARREGLO.

La creación de un arreglo se realiza utilizando la palabra clave **“new”,** junto al tipo de los elementos del arreglo y su número. Ejemplo para crear un arreglo donde guarde las notas de una asignatura de programación de 15 alumnos.

```
float [] notas;
notas = new float [15];
```

Para poder también crear un arreglo se puede escribir en una sola línea de sentencia.

```
float [] notas = new float[15];
```

Cuando se crea un arreglo en Java, Java considera a este arreglo como un objeto, en consecuencial al crearlo, se usa el operador **new** junto al tipo de elementos y su tamaño.

9.1.3. INICIALIZACIÓN DE UN ARREGLO

Los arreglos en Java pueden ser inicializados al momento de su declaración. Existen dos formas comunes de hacerlo: utilizando un bloque de inicialización o mediante un bucle para asignar valores.

```
double [] notas = {9, 1, 2, 8, 9, 9.50, 8, 9.5, 7.3, 8.4, 5, 9.3,
7, 8.3, 8.2};
```

En este caso, el arreglo se crea con 15 elementos, y cada uno se inicializa con los valores proporcionados. Java automáticamente determina el tamaño del arreglo basado en la cantidad de elementos.

9.1.4. INICIALIZANDO UN ARREGLO CON UN BUCLE.

Otra de la forma más común de inicializar un arreglo es utilizando un bucle, lo cual es especialmente útil cuando se necesita establecer valores de forma dinámica.

```
//Declaramos el arreglo con tamaño de 5
 double[] notas = new double[5];
 //Mediante un blucle asignamos un valor al arreglo
 for (int i = 0; i < notas.length; i++) {
     //Inicializa cada elemento con un valor random
     notas[i] = Math.random();
 }
```

En el código anterior, se utiliza el ciclo **"for"** para recorrer cada posición del arreglo, asignado un valor basado en variable de iteración **"i".**

9.1.5. ACCESO Y MODIFICACIÓN DE ELEMENTOS DE UN ARREGLO.

Una vez que se ha creado el arreglo e inicializado, cada uno de los elementos del arreglo pueden ser accedidos utilizando el índice correspondiente. En Java, el índice de un arreglo comienza en 0, lo que significa que el primer elemento se encuentra en la posición 0, lo que significa que el primer elemento se encuentra en la posición 0 y el último elemento se encuentra en la posición" n**-1"** donde **"n"** es el tamaño del arreglo.

```
//Accediendo al primer elemento del arreglo notas
double primerElemento= notas[0];
//Accediendo al último elemento del arreglo notas
double ultimoElemento= notas[notas.length-1];
```

9.1.6. MODIFICACIÓN DE ELEMENTO

En Java para modificar un elemento de un Array es tan sencillo que solo le asignamos un nuevo valor al índice especifico.

```
//Cambio el valor del primer elemento a 5
notas[0]=5;
//Cambia el valor del último elemento a 10
notas[4]=10;
```

Ejercicio 10.1

Crear un programa en Java donde el usuario ingrese por teclado un tamaño de un arreglo, multiplica cada uno de los elementos del arreglo y muestra en pantalla el producto del arreglo. Los datos deben ser ingresado por teclado.

```
package com.elvis.arreglos;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Main {
    public static void main(String[] args) throws IOException {
        //Inicializamos la clase BufferedReader e InputStreamReader
para ingresar datos por consola
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        //Declaramos la variable para ingresar el tamaño del vector
        int tamanio ;
        //Declaramos e inicializamos un acumulador donde se guardara
la multi´plicación de la matriz
        int producto=1;
        //Ingresamos por teclado el tamaño del vector inicializando la
variable tamaño
        System.out.println("Ingrese el tamaño del vector");
        //Convertimos la variable ingresada de tipo String a integer
        tamanio= Integer.parseInt(br.readLine());
       //Declaramos el arreglo con el tamño qeu inicializamos en la
variable tamaño
        int num[]=new int[tamanio];
        //Recorremos el tamaño del vector ingresando por teclado cada
uno de lso elementos
        for (int i = 0; i < tamanio; i++){
            System.out.print("Ingrese el valor del vector num
["+(i)+"]: " );
            num[i]= Integer.parseInt(br.readLine());
        }
        //Imprimimos cada uno de los valores del vectos
        System.out.println("Los elementos del arreglo es: ");
        for (int i = 0; i < tamanio; i++){
            System.out.print(num[i]+",");
```

```
        }
        //Recorremos y multiplcamos cada uno de los elementos de la
matriz
        for (int i = 0; i < tamanio; i++){
            producto=producto*num[i];
        }
        System.out.println(" ");
        //Imprimimos el producto de la matriz
        System.out.println("El producto de la vector es: "+producto);

    }
}
```

9.1.7. BUCLE FOR EACH PARA RECORRIDO DE ARREGLO Y COLECCIONES.

En la programación en, iterar sobre colecciones de datos es una tarea común. Java proporciona formas para realizar esto, y una de las más simples es el bucle **"for-each".** Este tipo bucle, se introdujo en Java 5, está diseñado específicamente para recorrer elementos de un arreglo o de cualquier estructura que implemente la interfaz **"iterable"**, como las listas o conjuntos.

Sintaxis

```
for (tipo de dato identificador ; nombreArreglo)
    sentencia
```

La sintaxis establece cada elemento del arreglo o colección a la variable dada, a continuación, ejecuta la sentencia o sentencias, las cuales naturalmente pueden ser un bloque, la colección debe ser un arreglo o un objeto de una clase que implementa la interfaz **iterable** tal como un **ArrayList.**

Sintaxis

```
for (tipo de dato identificador ; nombreArreglo)
    sentencia
```

Ejemplo 9.2

Realizar un programa en Java donde se cree una matriz de n elementos calcular la suma del array.

```
package com.elvis.arreglos;
```

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

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

        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        //Solicitamos el tamaño del arreglo
        System.out.println("Ingrese el tamaño del arreglo ");
        int tamanio = Integer.parseInt(br.readLine());

        //Creamos el areglo con el tamalo ingresado

        int [] num = new int[tamanio];

        //Usamos el for-each con un índice externo para ingresar
los elementos del array

        System.out.println("Ingrese los elementos del array");
        //declaros u nindece externo

        int index = 0;
        for (int elemento : num){
            System.out.print("Elemento [" + index + "]:");
            //Ingresamos los elementos al array
            num[index]=Integer.parseInt(br.readLine());
            //incrementamos el indice
            index++;
        }

        //Mostramos los elementos del arreglo para verificar que
fueron ingresados
        for (int elemento : num) {
            System.out.print(elemento + " ,");
        }
```

```
        int suma = 0;
        //sumamos los elementos del arreglo
        for (int elemento : num){
            suma=suma+elemento;
        }
        //Mostramos la suma del array
        System.out.println("");
        System.out.println("La suma del array es : "+ suma);

    }
}
```

Figura 52.
Array usando for-each

Nota: La imagen representa la salida de pantalla del código donde se ingresa una matriz usando for-each

9.2. ARREGLOS MULTIDIMENSIONALES.

Hasta aquí hemos implementado arreglos unidimensionales o de una sola dimensión, estos se caracterizan por tener un solo subíndice.

Los arreglos multidimensionales tienes más de una dimensión eso quiere decir que están conformados por filas y columnas conocido también como tablas o matrices. Pero también podemos crear estructura de datos de más dimensiones.

Un arreglo de dos dimensiones equivale a una tabla con múltiples filas y múltiples columnas, como se observa en la **Figura. 53,** obsérvese que las filas se etiquetan de 0 a m y las columnas de 0 a n, el número de elementos que tendrá la matriz será el resultado del producto (m+1) *(n+1), para poder acceder a un elemento de la matriz debemos hacerlo por las coordenadas representadas por su número de fila y de columna (a, b).

Sintaxis

```
<tipo de datoElemento> <nombre arreglo> [][];
```

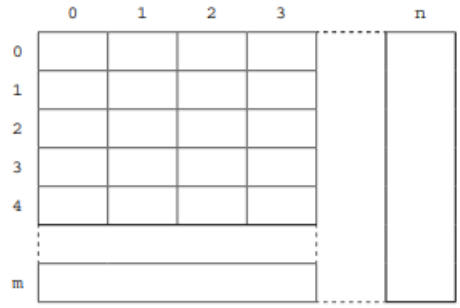
O bien

```
<tipo de datoElemento>[][] <nombre arreglo>;
```

Ejemplo como se declara una matriz

```
//declaramos el tipo de matriz de se va a crear tipo char
char pantalla[][];
//declaramos una matriz de tipo entero
int puestos[][];
//declaramos una matriz de tipo double
double[][] matriz;
```

Figura 53.
Estructura de un arreglo de dos dimensiones



Las declaraciones no reservan memoria para los elementos de la matriz, en realidad son referencias; para reservar memoria y especificar el tamaño de las filas y las columnas.

9.2.1. CREACIÓN DE UNA MATRIZ

Una vez que se haya declarado la matriz ahora nos toca crear la matriz para eso se utiliza la palabra **new.**

```
//Creamos las matrices
pantalla= new char[80][24];
```

```
//Creamos las matrices con un tamaño 80x24
pantalla= new char[80][24];
//creamos la matriz puestos de tipo entero y tamaño 10x5
puestos=new int[10][5];
//Declaramos una matriz de tamaño estático 5x5
final  int n=5;
matriz=new double[n][n];
```

También se puede inicializar un arreglo multidimensional con valores específicos al momento de la declaración.

```
//declaramos e inicializamso la matriz de tipo entero de nombre
matrz
int[][] matriz ={
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
};
```

En el ejemplo anterior, se crea una matriz de tamaño 3x4 con valores predefinidos. Cuando se crea un matriz lo primero que se va a leer son las filas y después las columnas.

9.2.2. INICIALIZACIÓN DE ARREGLOS BIDIMENSIONALES.

Como se observa en el ejemplo anterior la inicialización de una matriz con elementos constantes se inicializará encerrando entre llaves la lista de constantes de cada fila se separa con comas, como se observa en el ejercicio siguiente.

```
//declaramos e inicializamos la matriz de tipo entero de nombre
tabla
int[][] tabla= {{52, 54, 31},{41, 35, 37}};
```

Lo que se hace es definir una matriz de tamaño de 2 filas y 3 columnas por cada una de las filas.

9.2.3. ACCESO Y MANIPULACIÓN DE ELEMENTOS

Para acceder o modificar un elemento de una matriz, es necesario especificar tanto el numero de la fila y de la columna del elemento.

```
//Asigna el valor 8 al elemento de la fila 3 columna 4
matriz[2][3]=8;
//Obtenemos el valor de la fila 1 columna 3
int valor=matriz[0][2];
```

9.2.4. RECORRIDO DE UN ARREGLO BIDIMENSIONAL.

Para poder recorrer un arreglo bidimensional, es común utilizar bucles anidados en este caso utilizamos el **“for”** anidados. Lo primero que se va a recorrer con el primer **“for”** son la filas y con el **“for”** interno se recorrerá las columnas.

```
//Recorremos la matriz con bucles for
for (int i=0; i< matriz.length;i++){
    for (int j =0;j<matriz[i].length;j++){
```

```
            System.out.print(matriz[i][j]+" ");
        }
        //Imprimimos un espacio para separar cada una de las filas
        System.out.println();
}
```

El código implementado imprime todos los elementos de la matriz en forma de tabla como muestra en la

Figura 54.
Salida de pantalla Matriz 3x4

Nota: La figura representa a la salida de pantalla de la matriz en tipo tabla.

Ejercicio 9.3

Desarrolla un programa en Java que realice las siguientes operaciones con matrices, el ejercicio debe implementar con distintos métodos.

1. Inicialización de la matriz. Crear dos matices cuadras el tamaño debe ser ingresada por teclado.
2. Suma de matrices. Sumar las matrices A y B para obtener una matriz C
3. Producto de matrices: Calcular el producto de las matrices A y B, el resultado debe ser almacenado en una matriz D.
4. Mostrar las matrices: Imprimir en consola las matrices A, B, C y D.
5. Determinante de una matriz: Calcular e imprimir el determinante de la matriz.

Resolviendo cada ítem del ejercicio.

1. Implementamos un método para poder ingresar los datos a la matriz, la cual recibe dos parámetros, un parámetro de tipo entero llamado matriz y también recibe la variable br que permite ingresar datos por teclado.

```
//Implementamos el método para rellenar la matriz
public static void rellenarMatriz(int[][] matriz, BufferedReader
br) throws IOException {
    for (int i = 0; i < matriz.length; i++) {
```

```
            for (int j = 0; j < matriz[i].length; j++) {
                System.out.println("Ingrese el valor para la
posicion [ " + i + " ][ " + j + " ]: ");
                matriz[i][j] = Integer.parseInt(br.readLine());
            }
        }
}
```

2. Implementamos un método de tipo int de nombre **sumarMatrices** donde recibe dos parámetros en este caso la matriz A y la matriz B. Como el método es de tipo entero va a retornar el resultado de la suma A + B y se guardara el resultado en una nueva matriz de tipo entero de nombre C.

```
// Método para sumar dos matrices

public static int[][] sumarMatrices(int[][] A, int[][] B) {

      int filas = A.length;

      int columnas = A[0].length;

      int[][] C = new int[filas][columnas];

      for (int i = 0; i < filas; i++) {

            for (int j = 0; j < columnas; j++) {

                  C[i][j] = A[i][j] + B[i][j];

            }

      }

       return C;

}
```

3. Implementamos un método para multiplicar las matrices A y B y se guardara en una matriz D, este método es de tipo **int** de nombre multiplicarMatrices y recibe dos parámetros de tipo entero en este caso la matriz A y la matriz B.

   El método como es de tipo entero debe retornar un valor que es D.

```
//Implementamos un método para multiplicar las matices A * B
public static int[][] multiplicarMatrices(int[][] a, int[][] b)
{
    int filasA = a.length;
    int columnasA = a[0].length;
    int columnasB = b[0].length;
    int[][] d = new int[filasA][columnasB];
    for (int i = 0; i < filasA; i++) {
        for (int j=0;j<columnasB;j++){
            d[i][j]=0;
```

```
                for (int k=0;k<columnasA;k++){
                    d[i][j]=a[i][k]*b[k][j];
                }
            }
        }
        return d;
    }
```

4. Implementamos un método donde vamos a imprimir las dos matrices ingresadas y las matrices resultantes para este método utilizamos dos **for each** anidados donde van a recorrer las filas y las columnas de cada una de las matrices.

   El primer **for,** recorre las filas una vez que entra a la primera fila de la matriz ua vez que entra el segundo **for** comienza a recorrer las columnas va imprimiendo cada uno de los elementos de la primera fila y las columnas, sale del segundo for y se imprime un salto de línea y continua con la segunda fila, etc…

```
//Implementamos un método para imprimir

public static void imprimirMatriz(int[][] matriz) {
    for (int[] fila : matriz) {
        for (int elemento : fila) {
            System.out.print(elemento + " ");
        }
        System.out.println();
    }
}
```

5. Implementamos un método **main** para ejecutar cada uno de los métodos implementados.

```
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

    //Declaramos e inicializamos las variables para ingresar el
tamaño de la matriz
    int fila, columna;
    System.out.println("Ingrese el tamaño de fila");
    fila = Integer.parseInt(br.readLine());
    //Ingresamos por teclado el tamaño de la columna
    System.out.println("Ingresar el tamaño de la columna");
    columna = Integer.parseInt(br.readLine());
```

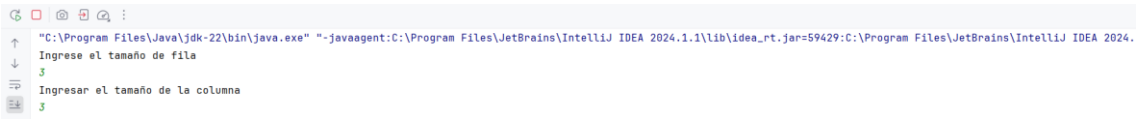
```
    if (fila == columna) {
        //Declaramos e inicializamos las matrices  A Y B
        int[][] a = new int[fila][columna];
        int[][] b = new int[fila][columna];
        //LLenamos las matrices con los elementos ingresando por
teclado
        System.out.println("Ingrese los valores de A");
        rellenarMatriz(a, br);
        //Llenamos la matriz b con los elementos ingresados por
teclado
        System.out.println("Ingrese los valores de la matriz
B");
        rellenarMatriz(b, br);

        //sumar las matrices ingresadas
        int[][] c = sumarMatrices(a, b);

        //Multiplicar matrices A y B (Solo si el número de
columas de A es iagual a las filas de B)
        int [][] d = null;
        if(a[0].length==b.length){
            d=multiplicarMatrices(a,b);
        }else{
            System.out.println("No es posible multiplicar las
matrices A y B debido a la incompatibilidad de tamaño");
        }


        //Mostrar las matriz A, B, C y D
        System.out.println("\n Matriz A");
        imprimirMatriz(a);
        System.out.println("\n Matriz B");
        imprimirMatriz(b);
        System.out.println("Matriz C=(A+B)");
        imprimirMatriz(c);
        if(d!=null){
            System.out.println("\n Matriz D=(A*B)");
            imprimirMatriz(d);
        }
```

```
        } else {
            System.out.println("los datos de la fila y columna den
ser iguales ");
        }
    }
```

Salida de Pantalla

Primero ingresamos por teclado el tamaño de las filas y columnas de la matriz.

Figura 55.
Ingreso del tamaño por teclado



Nota: La imagen representa al ingreso por teclado de tamaño de las filas y columnas de la matriz.

Ingresamos los elementos de cada una de las matrices cada vez que pide un ingreso nos muestra el numero de la fila y el numero de la columna donde se está ingresando.

Figura 56.
Ingreso por teclado matriz A

Nota: La imagen nos muestra el ingreso de la matriz a en su respectivo espacio de la matriz.

Ingresamos los elementos de la matriz B.

Figura 57.
Ingreso elementos matriz B

Nota. La imagen nos muestra el ingreso de cada uno de los elementos de la matriz B.

Por último, se muestra en pantalla las matrices resultantes.

Figura 58.
Impresión de las matrices A, B y resultantes

Nota. La imagen nos muestra las matrices que ingresamos por teclado y las matrices resultantes de las operaciones indicadas.

9.3. ARREGLOS IRREGULARES O TRIANGULARES

Como se estudió en la sección anterior Java puede crear arreglos regulares que quiere decir que tienen el mismo número de filas y columnas, Java también puede crear arreglos irregulares en pocas palabras que el número de filas y columnas son distintas por ejemplo se puede crear el siguiente modelo del binomio de Newton:

Figura 59.
Triangulo de Pascal

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Nota: La imagen representa al triangulo de Pascal desarrollado con matrices en Java.

En Java, las matrices irregulares, son una característica que permite crear matrices multidimensionales donde las filas pueden tener diferentes longitudes. A diferencia de las matrices bidimensionales tradicionales, que son tablas rectangulares donde cada fila tiene el mismo número de columnas, las matrices irregulares ofrecen una estructura más flexible.

9.3.1. CREACIÓN DE MATRICES IRREGULARES.

Para crear una matriz irregular en Java, primero debes declarar la matriz especificado solo el número de filas. Posteriormente, puedes inicializar cada fila con un tamaño diferente.

```
//Creamos una matriz con 3 filas.
int[][] matrizIrregular = new int[3][];
//Primer fila con dos columnas
matrizIrregular[0]=new int[2];
//Segunda fila con 4 columnas
matrizIrregular[1]=new int[4];
//Tercer fila con 3 columnas
matrizIrregular[2]=new int[3];
```

El código nos muestra cómo se crea una matriz irregular esta matriz tiene tres filas, pero cada fila tiene un numero distinto de columnas.

9.3.2. VENTAJAS Y DESVENTAJAS.

Ventajas.

- **Flexibilidad:** Las matrices irregulares permiten trabajar con datos que no encajan en una estructura uniforme, como listas de listas, donde cada lista interna puede tener una longitud diferente.
- **Eficiencia en la memoria:** Al no estar obligadas a tener la misma longitud en cada fila, estas matrices pueden utilizar menos memoria si las filas varían en tamaño.

Desventajas.

- **Complejidad.** Manipular matrices irregulares pude ser más complejo que trabajar con matrices bidimensionales estándar, ya que es necesario verificar la longitud de cada fila antes de acceder a sus elementos.
- Rendimiento. En algunos casos, el acceso a los elementos de una matriz irregular puede ser menos eficiente debido a la falta de uniformidad en la estructura de datos.

Ejercicio 9.4

Crear un programa en Java donde el usuario ingrese el tamaño una matriz irregular, imprimir en pantalla en triangulo de Pascal como se observa en la **figura 59.**

1. Crear un Método para generar el triángulo de Pascal.

2. Crear un método para imprimir el triángulo de Pascal.
3. Crear un método **main** para ejecutar el programa.

Implementamos el método para generar el triángulo de Pascal este método es de tipo **int** que va a devolver la matriz creada, el método recibe dos parámetros el tamaño de la matriz.

```
//método para generar el triángulo de Pascal
public static int[][] trianguloPascal(int n){
    int[][] triangulo = new int[n][n];
    for(int i=0; i<n; i++){
        //el primer y último valor de cada fila es 1
        triangulo[i][0] = 1;
        triangulo[i][i] = 1;
        //Los valores intermedios se calculan como la suma de
los dos valores de arriba
        for(int j=1; j<i; j++){
            triangulo[i][j]=triangulo[i-1][j-1]+triangulo[i-
1][j];
        }
    }
    return  triangulo;
}
```

El siguiente método para implementar es para imprimir el triangulo de Pascal este método es de tipo **void** el cual no va a devolver ningún parámetro, este método recibe un parámetro que es la matriz que se acaba de crear en el método anterior.

```
//Método para imprimir el triángulo de Pascal
public static void  imprimirTriangulo(int[][] triangulo){
    int n=triangulo.length;
    for(int i=0; i<n; i++){
        //Imprimimos los espacios para centrar la pirámide
        for(int j=0; j<n-i; j++){
            System.out.print(" ");
        }
        //Imprimimos los valores de la fila actual
        for(int j=0; j<=i; j++){
            System.out.print(triangulo[i][j]+" ");
        }
```

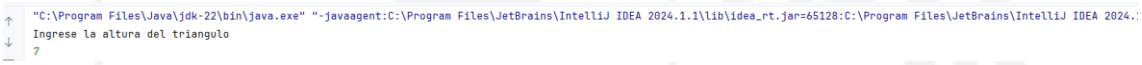
```
            System.out.println();
        }
}
```

Por último, se implementa un método **main** para poder ejecutar nuestro programa. Lo primero que se hace es ingresar por teclado el tamaño de la matriz mediante el método **trianguloPascal()** y por último llamamos al método **imprimirTriangulo(),** para mostrar en pantalla la matriz creada.

```
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
    System.out.println("Ingrese la altura del triángulo");
    int n = Integer.parseInt(br.readLine());
    //Generamos el triángulo de Pascal
    int[][] trianguloPascal= trianguloPascal(n);
    //Imprimimos el triángulo de Pascal en consola
    imprimirTriangulo(trianguloPascal);
}
```

Al ejecutar el programa lo primero ingresamos el tamaño de la matriz que es de tipo **int.**

Figura 60.
Tamaño del triangulo



Nota: La imagen nos muestra el ingreso del tamaño del triángulo por teclado.

Luego el programa muestra el resultado el triángulo de Pascal con una matriz irregular.

Figura 61.
Salida del programa triangulo de Pascal

Nota: La imagen muestra la salida de pantalla donde se ha generado el triángulo de Pascal.

9.4. CLASE Vector y ArrayList

En Java, tanto la clase **"vector",** como la clase **"ArrayList"** forman parte del marco de colecciones y se utilizan para almacenar grupos de objetos. Aunque ambos ofrecen funcionalidades similares al manejar listas dinámicas, hay ciertas diferencias claves en cómo se comportan y se utilizan en diferentes contextos.

Estas clases almacena secuencias de objetos de cualquier tipo: las colecciones este tipo de estructura de datos se diferencian en la forma de organizar los objetos y, por consecuencia en la manera en las que se puede recuperar.

9.4.1 CLASE Vector

La clase Vector se encuentra en el paquete `java.util` este colección tiene un comportamiento similar a un arreglo unidimensional, guarda objetos o referencias de cualquier tipo, este tipo de estructura de datos crece dinámicamente, sin necesidad de que se programe operaciones adicionales, el arreglo donde almacena los elementos es de tipo `Object`.

La clase **"Vector"** fue introducida en Java 1.0 y aunque sigue siendo útil es ciertos casos, el uso de esta clase ha disminuido en favor a la clase **"ArrayList".**

9.4.2. Características

La clase **"Vector"** tiene ciertas características que permite crear las estructuras de datos de una forma más fácil.

- **Sincronización:** La clase **"Vector"** es sincronizado, lo que significa que es seguro para su uso en entornos multihilos sin necesidad de medidas adicionales para la sincronización. Sin embargo, esta sincronización introduce una sobrecarga en el rendimiento.
- **Crecimiento dinámico:** Al igual que un **"ArrayList"**, un **"vector"** tambien puede crecer dinámicamente cuando alcanza su capacidad de almacenamiento. Sin embargo un "vector", suplica su tamaño por defecto cuando necesita más espacio, lo que puede llevar a un uso ineficiente de la memoria en algunos caso.
- **rden de inserción:** Otra de las características que el **"Vector"** inserta sus elementos de forma ordenada y los mantiene de esa forma.
- **Métodos heredados:** La clase **"Vector"** incluye algunos métodos heredados de la clase **"stack"**
    - **"addElemet":** Este método nos permite añadir un elemento al vector**,**

- **elemetAt":** Este método se utiliza para buscar o recuperar un elemento en índice especifico de un vector.

La declaración de un vector es muy sencilla

```
Object[] elementData;
```

Desde Java en su versión 5 se pude establecer el tipo concreto de elemento, el cual, de ser una clase, la cual puede guardar una colección, en este caso un vector por ejemplo un vector de tipo String.

```
//Declaramos una variable nombre de tipo vector
Vector<String> nombres=new Vector<>();
```

9.4.3. CREACION DE UN VECTOR

Al igual que un Objecto para crear un Vector se utiliza la palabra clave **new,** la clase **Vector** dispone de diversos constructores, por ejemplo.

```
//Creamos un vector vacio
 public Vector();
 //Creamos un vector con una capacidad inicial
 public Vector(int capacidad);
```

Inicializando los vectores.

```
Vector v1=new Vector();
Vector v2=new Vector();
//Este vector contiene los mismos elementos del v2
Vector v3=new Vector(v2);
```

9.4.4. Insertando elementos al Vector

Cuando se inserte elementos a un vector existen diferentes formas para insertar los elementos, lo que no se puede hacer es insertar datos de tipo primitivo, por ejemplo, **int, char, etc.**

Los métodos que se utiliza para insertar un elemento en un Vector son los siguientes.

- **boolean add(Object ob):** Este método permite insertar un objeto después delltimo elemento del vector.
- **void addElement(Objecto b):** Este método permite añadir el objeto después del último elemento del vector.

- **void insertElement(Objecto b, int p):** inserta el objeto en la posición p, y los elementos posteriores a p se desplazan.

Se debe tomar en cuenta que, al momento de insertar un elemento al vector con un tipo concreto, ese elemento debe ser del mismo tipo, o de una derivado, por ejemplo.

```
//Declaramos una variable dias de tipo vector

Vector<String> dias = new Vector<>();
//Añadimos el elemento Jueves al vector
dias.add("Jueves");
//Se añade un elemento primitivo al vector dias error por
incompatibilidad de tipos
dias.addElement(new Integer(1));//error de tipo
```

9.4.5. Acceso a un elemento del vector

Al igual que un **Array,** para acceder a un elemento de un **Vector** se lo realiza mediante la posición que el elemento ocupa, los métodos de acceso devuelven el elemento con el tipo Objects, el cual es necesario hacer una conversión al tipo objeto.

```
//El método elementAt devuelve el elemento del vector en la
posición p
Object elementAt(int p);
//el método get  devuelve el elemento cuya posición es p
Object get(int p);
//El método size devuelve el número de elementos del vector
int size();
```

9.4.6. Eliminar un elemento del vector

A diferencia de un **Array** un vector es una estructura de datos dinámica, que crece y decrece dinámicamente si se añade o elimina objetos, eliminar un elemento de un vector en Java utilizando la clase **Vector** es un proceso esencial para manipular datos almacenados en esta estructura. La clase **Vector** es parte de la vieja biblioteca de colecciones de Java que es sincronizada por defecto, lo que significa que es segura para su uso en entornos multihilo.

En la clase **Vector,** hay varias formas de eliminar elementos:

- **Eliminar por el índice:** Elimina el elemento en la posición especificada.
- **Eliminar por Objeto:** Elimina la primera aparición de un elemento especifico

- **Eliminar todos los elementos que cumplan con una condición:** Utilizando un **Iterator** o el método **removeIf.**

```
//eliminar elemento índice y el resto se reenumera
void removeElementAt(int indice);
//Elimina la primera aparición de op, devuelve true si se elimina
correctamente
boolean removeElement(Object op);
//Elimina todos los elementos
void removeAllElements()
```

9.4.7. Búsqueda de un elemento en un Vector

La búsqueda de un elemento dentro de un **Vector** es una operación fundamental en la programación, especialmente cuando trabajamos con estructura de datos que almacena múltiples valores, como la clase **Vector** en Java. Este proceso permite determinar si un elemento existe en la colección, así como también localizar su posición exacta. Esta operación es esencial en aplicaciones donde se requiere verificar la presencia de ciertos datos o acceder a elementos específicos para su manipulación.

9.4.8. Métodos de búsqueda en un Vector

Java ofrece varios métodos en la clase **Vector** que permiten buscar elementos de manera eficiente. Estos métodos incluyen:

- **contains(Object o):** Este método verifica si un elemento especifico existe en el vector. Retorna **true** si el elemento está presente y **false** en caso contrario.
- **indexOf(Object o):** Este método busca el índice de la primera aparición del elemento especificado. Si el elemento está presente, retorna su índice (basado en 0); si no, retorna -1.
- **lastIndexOf(Object o):** Similar al **indexOf**, pero busca la ultima aparición del elemento en el vector.
- **elementAt(int index):** Aunque no es un método de búsqueda en sí, este método permite acceder a un elemento en una posición especifica, lo cual puede ser útil después de encontrar su índice con el método **indexOf.**

Ejercicio 9.5

Crea un programa en Java que simule un sistema de gestión de inventario para una tienda. El sistema debe permitir agregar productos, buscar productos, actualizar la cantidad de un producto específico, eliminar productos del inventario, y mostrar el inventario completo. Además, el programa debe ofrecer la funcionalidad para limpiar el inventario y realizar operaciones de copia y sublista del inventario.

Utiliza la clase Vector para almacenar los productos del inventario. Cada producto debe estar representado por una clase llamada Producto que contenga el nombre del producto, la cantidad disponible y el precio.

El programa debe implementar y demostrar el uso de los siguientes métodos de la clase Vector:

1. add(E e): Agregar un nuevo producto al inventario.
2. remove(Object o): Eliminar un producto del inventario.
3. remove(int index): Eliminar un producto por índice.
4. get(int index): Obtener un producto por índice.
5. set(int index, E element): Actualizar la información de un producto.
6. indexOf(Object o): Encontrar el índice de un producto específico.
7. contains(Object o): Verificar si un producto existe en el inventario.
8. isEmpty(): Verificar si el inventario está vacío.
9. size(): Obtener el número total de productos en el inventario.
10. clear(): Limpiar el inventario.
11. clone(): Crear una copia del inventario.
12. subList(int fromIndex, int toIndex): Crear una sublista del inventario.

Implementamos una clase Producto donde se encapsula cada uno de los parámetros del Producto, implementamos un constructor vacío, implementamos un constructor con todos los parámetros donde inicializaremos cada uno de ellos e implementamos un constructor que solo recibe el parámetro nombre el cual busca un parámetro dentro del vector. Implementamos los métodos get y set que permite obtener y modificar los productos y por último se sobrescribe el método toString par a ver los datos dl producto

Clase Producto

```
package com.elvis.arreglos;

public class Producto {
    //Declaramos las variables del producto
    private String nombre;
    private int cantidad;
    private double precio;
    //implementamos un constructor vacío
```

```
    public Producto() {
    }
    //Implementamos un constructor para inicializar solo el
nombre y poder buscar un producto

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

    //Implementamos un constructor donde inicializamos todas las
variables de producto
    public Producto(String nombre, int cantidad, double precio)
{
        this.nombre = nombre;
        this.cantidad = cantidad;
        this.precio = precio;
    }
    //Implementamos los método Get y Set

    public String getNombre() {
        return nombre;
    }

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

    public int getCantidad() {
        return cantidad;
    }

    public void setCantidad(int cantidad) {
        this.cantidad = cantidad;
    }

    public double getPrecio() {
        return precio;
```

```
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }
    //Implementamos un método toString para obtener los valores
de los parámetros ingresados
    @Override
    public String toString() {
        return "Producto{" +
                "nombre='" + nombre + '\'' +
                ", cantidad=" + cantidad +
                ", precio=" + precio +
                '}';
    }
}
```

Para ejecutar el programa se implementa la clase Main, donde se programará el Menú y se llamará a cada uno de los métodos indicados en el enunciado del ejercicio.

Se declara una variable de tipo entero para elegir y guardar la opción que se debe realizar en el programa.

```
public static void main(String[] args) throws IOException {
    //declaramos un variable de tipo int donde se guardara la
opción a realizar
    int op;
```

Creamos el vector de tipo Producto de nombre inventario.

```
//Creamos el vector de tipo Producto
Vector<Producto> inventario = new Vector<>();
```

Declaramos e inicializamos la clase **BufferedReader e InputStreamReader** que permite ingresar datos por teclado en consola.

```
//Declaramos e inicializamos la clase BufferedReader
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
```

Se implementa un menú con el ciclo **doWhile** donde el usuario elije la opción que se desea ejecutar y mientras no se ingrese la opción 0, el programa se seguirá ejecutando las veces que el usuario desee.

Luego de elegir la opción se ingresa a un **switch** evaluando la opción ingresada y comparando que opción se ingreso y ejecutando el código necesario.

```
do {
    //Creamos un menú para que el usuario elija el proceso a
realizar
    System.out.println("--------Sistema de Inventario--------");
    //Opción 1 agrega un nuevo producto al vector
    System.out.println("1. Agregar Producto");
    //Opción 2 elimina un producto buscando el nombre del
producto
    System.out.println("2. Eliminar Producto por el nombre");
    //Opción 3 elimina un producto ingresando el índice del
producto
    System.out.println("3. Eliminar el producto por índice");
    //Opción 4 actualiza un producto ingresando el índice del
producto
    System.out.println("4. Actualizar producto por índice");
    //Opción 5 Busca un producto por el nombre del producto
    System.out.println("5. Buscar producto por nombre");
    //Opción 6 Verifica si el vector este vacío
    System.out.println("6. Verificar si el inventario este
vacío");
    //Opción 7 Muestra el total de productos del vector
    System.out.println("7. Mostrar número total de productos");
    //Opción 8 Muestra los productos del vector
    System.out.println("8. Mostrar todos los productos");
    //Opción 9 Limpia el vector
    System.out.println("9. Limpiar inventario");
    //Opción 10 Clona un vector
    System.out.println("10. Clonar inventario");
    //Opción 11 Crea una sablista ingresando el inicio del
vector y el fin del vector
    System.out.println("11. Crear una sablista del inventario");
    //Opción 0 me permite salir de la aplicación
    System.out.println("0. Salir");
    System.out.println("Ingrese el número de proceso que desea
realizar");
```

```
        op = Integer.parseInt(br.readLine());
        switch (op) {
            case 1:
                //Ingresamos por teclado los parámetros del producto
                System.out.println("Ingrese el nombre del
producto");
                String nombre = br.readLine();
                System.out.println("Ingrese la cantidad del
producto");
                int cantidad = Integer.parseInt(br.readLine());
                System.out.println("Ingrese el precio del
producto");
                double precio = Double.parseDouble(br.readLine());
                //llenamos el vector con los parámetros del producto
                inventario.add(new Producto(nombre, cantidad,
precio));
                System.out.println("Producto agregado
exitosamente");
                break;
            case 2:
                //Ingresamos el nombre del producto que se va a
eliminar
                System.out.println("Ingrese el nombre del producto a
eliminar");
                String productoEliminar = br.readLine();
                //Removemos el producto que ingresamos si existe en
el Vector
                inventario.removeIf(producto ->
producto.getNombre().equals(productoEliminar));
                System.out.println("Producto eliminado
correctamente");
                break;
            case 3:
                //Ingresamos el índice del producto a eliminar
                System.out.println("Ingrese el índice del producto a
eliminar");
                int indiceEliminar =
```

```
Integer.parseInt(br.readLine());
            //Implementamos una condición para verificar si el
índice ingresado es mayor a 0 y menor al tamaño del vector
            if (indiceEliminar >= 0 && indiceEliminar <
inventario.size()) {
                inventario.remove(indiceEliminar);
                System.out.println("Producto eliminado
correctamente");
            } else {
                System.out.println("Índice fuera de rango");
            }
            break;
        case 4:
            //Ingresamos el índice del producto que se va a
actualizar
            System.out.println("Ingrese el índice del producto a
actualizar");
            int indiceActualizar =
Integer.parseInt(br.readLine());
            //evaluamos si el índice ingresado es correcto y si
esta dentro del rango del tamaño del vector
            if (indiceActualizar >= 0 && indiceActualizar <
inventario.size()) {
                //obtenemos el producto a actualizar mediante el
índice
                Producto prod =
inventario.get(indiceActualizar);
                System.out.println("Ingrese el nuevo nombre del
producto");
                prod.setNombre(br.readLine());
                System.out.println("Ingrese la nueva cantidad
del producto");

prod.setCantidad(Integer.parseInt(br.readLine()));
                System.out.println("Ingrese el nuevo precio del
producto");
```

```
prod.setPrecio(Double.parseDouble(br.readLine()));
                    //seteamos los nuevos datos en el producto
seleccionado
                    inventario.set(indiceActualizar, prod);
                    System.out.println("Producto actualizado");
                } else {
                    System.out.println("Índice fuera de rango");
                }
                break;
            case 5:
                //Ingresamos el nombre del producto a buscar
                System.out.println("Ingrese el nombre del producto a
buscar");
                String nombreBuscar = br.readLine();
                //Buscamos el producto dentro del vector
                int indice = inventario.indexOf(new
Producto(nombreBuscar));
                //Verificamos si la búsqueda devuelve 0 el producto
existe
                if (indice != -1) {
                    System.out.println("Producto encontrado en el
índice " + indice);
                    //Caso contrario el producto no existe
                } else {
                    System.out.println("Producto no encontrado ");
                }
                break;
            case 6:
                //Verificamos si el vector esta vacío
                if (inventario.isEmpty()) {
                    System.out.println("El inventario está vacío");
                } else {
                    System.out.println("El inventario no está
vacio");
                }
                break;
            case 7:
```

```
            //Mostramos el tamaño del vector o el total de
productos
            System.out.println("Número total de productos en el
inventario: " + inventario.size());
            break;
        case 8:
            //Mostramos los productos del inventario
            System.out.println("Productos del inventario");
            //Recorremos el vector para imprimir en pantalla los
productos.
            for (int i = 0; i < inventario.size(); i++) {
                System.out.println(i + ":" + inventario.get(i));
            }
            break;
        case 9:
            //Limpiamos el vector
            inventario.clear();
            System.out.println("El inventario fue limpiado ");
            break;
        case 10:
            //creamos un vector y el cual tenemos que clonar el
vector inventario a inventarioClonado
            Vector<Producto> inventarioClonado =
(Vector<Producto>) inventario.clone();
            System.out.println("Inventario clonado");
            System.out.println("Copia del inventario");
            //Recorremos el vector con un foreach
            for (Producto producto : inventarioClonado) {
                System.out.println(producto);
            }
            break;
        case 11:
            System.out.println("Ingrese el índice de inicio par
la sablista: ");
            int inicioIndice = Integer.parseInt(br.readLine());
            System.out.println("Ingrese el índice de fin para la
sablista: ");
```

```
            int finIndice = Integer.parseInt(br.readLine());
            if (inicioIndice >= 0 && finIndice <=
inventario.size() && inicioIndice < finIndice) {
                System.out.println("Sublista del inventario");
                Vector<Producto> sublista = new
Vector<>(inventario.subList(inicioIndice, finIndice));
                //Recorremos el vector para imprimir la sublista
                for (Producto producto : sublista) {
                    System.out.println(producto);
                }
            } else {
                System.out.println("Índices fuera de rango o
inválidos");
            }
            break;
        case 0:
            System.out.println("Saliendo del sistema...");
            break;
        default:
            System.out.println("Opción no valida, Por favor,
intente nuevamente");
    }

} while (op != 0);
}
```

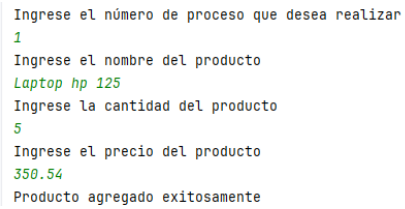
Salida de pantalla ejecutando el programa

Figura 62.
Menú del programa

Nota: La imagen muestra el menú el usuario debe elegir la opción deseada.

Cuando se elige la opción 1 ingresamos productos al vector Producto, los datos a ingresar son nombre del producto, cantidad y valor del producto para nuestro ejemplo se ingresar tres productos.

Figura 63.
Opción 1, ingreso datos



Nota: La imagen muestra al elegir la opción 1 y como se ingresa los datos al vector.

Para cuestión del ejercicio vamos a probar cada uno de los ítems de menú.

9.5. Clase ArrayList

La clase **ArrayList** en Java es parte del paquete **java.util** y es una de las implementaciones más utilizadas de la interfaz **List**. A diferencia de los arreglos tradicionales en Java, los cuales tiene un tamaño fijo, un **ArrayList** es una estructura de datos que puede crecer o reducirse dinámicamente según sea necesario. Esto lo convierte en una opción flexible y poderosa para gestionar colecciones de datos.

Un **ArrayList** es una colección ordenada de elementos donde cada elemento tiene una posición indexada, comenzando desde 0. A diferencia del **Vector**, otra clase de coleccione que también permite el almacenamiento dinámico, **ArrayList** no está sincronizada, lo que significa que no es seguro para su uso en entornos multihilo sin medidas adicionales de sincronización. Esto lo hace más rápido en operaciones de un solo hilo, pero menos seguro en aplicaciones concurrentes.

Internamente, un **ArrayList** utiliza un arreglo para almacenar sus elementos. Cuando el número de elementos supera la capacidad del arreglo interno, el **ArrayList** crea un nuevo arreglo con mayor capacidad y copia los elementos antiguos en el nuevo arreglo. Este comportamiento permite que el **ArrayList** crezca dinámicamente, pero también implica un costo en rendimiento cada vez que se redimensiona.

9.5.1. Métodos de la clase ArrayList

El **ArrayList** ofrece una gran cantidad de métodos para gestionar y manipular los elementos del Array.

- **add(E e):** Añade un elemento al final del **ArrayList**.

- **add(int index, E elemento):** Inserta un elemento un elemento en un posición especifica.
- **get(int index):** Devuelve el elemento en la posición especificada.
- **set(int index, E element):** Reemplaza el elemento en la posición especificada con un nuevo elemento.
- **remove(int index):** Elimina el elemento en la posición indicada.
- **remove(Object o):** Elimina la primera aparición con un elemento especifico.
- **contains(Object o):** Verifica si el **ArrayList** contiene el elemento especifico.
- **size():** Devuelve el número de elementos en el **ArrayList**
- **clear():** Elimina todos los elementos del ArrayList
- **isEmpy():** Verifica si el **ArrayList** está vacío.
- **indexOf(Object o):** Devuelve el índice de la primera aparición de un elemento o **-1** si no se encuentra.
- **lastIndexOf(Object o):** Devuelve el índice de la última aparición de u elemento o **-1** si no se la encuentra.
- **subList(int inicioInidice, int finalIndice):** Devuelve una vista de una parte del **ArrayList** entre los índices ya indicados.

Para crear un **ArrayList** en Java es un proceso sencillo que implica importar la clase **ArrayList** desde el paquete **java.util** y luego instanciar un nuevo objeto de **ArrayList**.

```
//Creamos un ArrayList de tipo String
ArrayList<String> listaNombres=new ArrayList<>();

//Creamos un ArrayList de tipo enteros
ArrayList<Integer> listaNumeros = new ArrayList<>();

//Creamos un ArrayList de un tipo de objeto personalizado
ArrayList<MiObjeto> listaObjetos= new ArrayList<>();
```

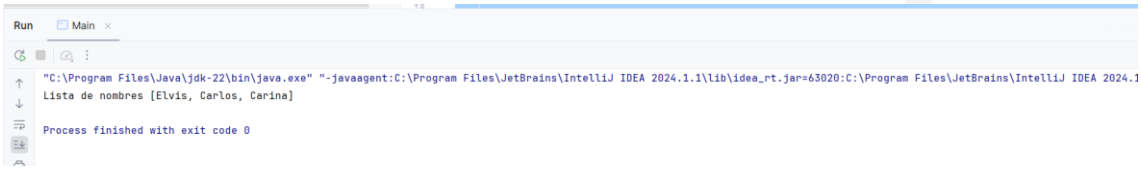
Ejercicio 9.6.

Crear un programa en Java donde se cree un objeto de tipo ArrayList donde se ingres tres nombres e imprimir el ArrayList

```
package com.elvis.arreglos;

//Importamos la biblioteca ArrayList de la clase java.util
import java.util.ArrayList;
public class Main {
    public static void main(String[] args) {
        //Creamos un ArrayList de tipo String para almacenar los
nombres
        ArrayList<String> listaNombres=new ArrayList<>();

        //Agregamos los nombres al ArrayList
        listaNombres.add("Elvis");
        listaNombres.add("Carlos");
        listaNombres.add("Carina");

        //Imprimimos el ArrayList con todos los elementos

        System.out.println("Lista de nombres "+ listaNombres);


    }
}
```

En la salida de pantalla se observa todos los elementos del ArrayList

Figura 64.
Salida pantalla ArrayList



Nota: La imagen muestra la salida de pantalla donde se muestra los elementos de la lista.

RESUME DEL CAPITULO 9

En este capítulo se exploraron lo arreglos en Java, que son una estructura de datos utilizada para almacenar y gestionar colecciones de elementos del mismo tipo. Se discutieron los conceptos claves, incluyendo:

- Un arreglo es una forma estructurada de organizar y acceder a elementos de un tipo de dato especifico.
- Existen arreglos de una, dos o múltiples dimensiones.
- Para declarar un arreglo, se especifica el tipo de dato de sus elementos, el nombre del arreglo y se indican las dimensiones con corchetes. El tamaño del arreglo se establece con el operador **new.** Los elementos de arreglo se pueden acceder mediante asignaciones directas, lecturas, escrituras, o usando bucles como **for, while** o **do-while.**
- En Java los arreglos son considerados objetos, por lo que todos tienen el atributo **length** que indica su longitud.
- Los arreglos de caracteres en Java son secuencias individuales y la clase **String** se utiliza para manipular cadenas de texto.

Además, se revisaron las clases **Vector, ArrayList** del paquete **java.util.** Las clases **Vector y ArrayList** permiten almacenar y recuperar objetos de cualquier tipo, funcionando de manera similar a os arreglos, pero con más flexibilidad, ya que tratan los elementos como objetos de tipo **Object.**

EJERCICIOS PROPUESTOS.

1. Desarrollar un programa en Java que permita almacenar y gestionar las notas de un grupo de estudiantes utilizando un arreglo unidimensional. El programa debe ofrecer las siguientes funcionalidades.
    a. Ingresar el número de estudiantes.
    b. Ingresar las notas de los estudiantes.
    c. Calcular y mostrar la nota promedio de la clase
    d. Mostrar la nota más alta y la nota más baja.
    e. Mostrar cuántos estudiantes tiene una nota superior al promedio.

    Todos los datos deben ser ingresados por teclado y las notas que se ingresan deben estar entre 0.1 y 10.

2. Escribir un programa que permita realizar la suma de dos matrices de números enteros de tamaño **nxm** donde **n y m** son ingresados por teclado. El programa debe tener las siguientes funcionalidades.
    a. Solicitar el tamaño de las matrices **n y m**
    b. Permitir al usuario ingresar los valores de las dos matrices.
    c. Calcular y mostrar la matriz resultante de la suma de las dos matrices ingresadas.
    d. Mostrar el valor mayor y menor de la matriz resultante.

    Todos los datos deben ser ingresado por teclado.

3. Crear un programa en Java que funcione como una agenda de contactos utilizando la clase **Vector.** Cada contacto debe contener un nombre, un número de teléfono y una dirección de correo electrónico. El programa debe permitir:
    a. Agregar un nuevo contacto.
    b. Buscar un contacto por nombre
    c. Eliminar un contacto por nombre.
    d. Mostrar los contactos almacenados.
    e. Salir del programa.

    Todos los datos deben ser ingresados por teclado.

4. Desarrolla un programa en Java que permita gestionar un inventario de productos utilizando la clase **ArrayList.** Cada producto debe tener un nombre, un precio y una cantidad disponible en inventario. El programa debe ofrecer las siguientes funcionalidades.
    a. Agregar un nuevo producto al inventario.
    b. Buscar un producto por nombre.
    c. Actualizar la cantidad de un producto existe.
    d. Mostrar todos los productos disponibles.
    e. Eliminar un producto de inventario.
    f. Salir del programa.

    Todos los datos deben ser ingresados por teclado.

5. Escribir un programa que lea el siguiente arreglo.

4	8	1	3	5
2	0	6	9	7
4	6	2	1	3

Y lo escriba de la siguiente forma

4	2	4
8	0	6
1	6	2
3	9	1
5	7	3

6. Escribir un programa en el que se genere aleatoriamente un arreglo de 20 números enteros. El arreglo debe quedar de tal forma que la suma de los 10 primero elementos sea mayor que la suma de los 10 últimos. Mostrar el arreglo original y el arreglo con la distribución indicada.

CAPITULO 10

GRAFICOS GUI/Swing

Introducción

Hasta ahora, todos los ejemplos en este libro han utilizado entradas desde el teclado y salida en pantalla. Sin embargo, en el desarrollo moderno, es común crear aplicaciones que interactúan a través de páginas web y utilizan interfaces gráficas de usuario (GUI, por sus siglas en inglés). Es esencial que los desarrolladores comprendan cómo manejar elementos GUI, como menús y botones, dentro de sus aplicaciones. Este capítulo presenta la creación de aplicaciones gráficas y el desarrollo de interfaces gráficas de usuario que interactúan con el sistema operativo. Además, se introducirá el concepto fundamental de "eventos" y sus respectivos manejadores. Aunque este tema se explora en mayor profundidad en cursos avanzados de Java, aquí se cubrirán los aspectos básicos.

En el ecosistema actual de Java, existen dos principales conjuntos de componentes GUI: AWT y Swing. En las primeras versiones de Java, hasta Java SE 1.2, las interfaces gráficas se construían usando el *Abstract Window Toolkit* (AWT), disponible en el paquete java.awt. Swing, inicialmente una extensión en Java 1.1, se integró completamente en la biblioteca estándar de Java SE 1.2 como parte de la *Java Foundation Class* (JFC). Swing ha reemplazado en gran medida a AWT al ofrecer componentes más avanzados y flexibles. A pesar de esto, AWT sigue siendo la base de muchas operaciones fundamentales, como el manejo de eventos. En resumen, AWT proporciona los componentes originales de la GUI, mientras que Swing añade una colección más moderna y versátil, ambos incluidos en las versiones actuales del JDK.

En las aplicaciones actuales, especialmente aquellas que buscan integrarse con tecnologías modernas, es recomendable explorar opciones adicionales como JavaFX para interfaces más ricas y Jakarta EE para aplicaciones web robustas. Además, herramientas de construcción como Maven facilitan la gestión de dependencias y el mantenimiento de proyectos, integrando estas bibliotecas de manera eficiente en los entornos de desarrollo contemporáneos.

10.1. Swing

Swing es una parte fundamental del entorno de Java, proporcionando a los desarrolladores una poderosa herramienta para crear interfaces graficas de usuario (GUI) que sean flexibles, robustas y estéticamente agradables. Desde si introducción a Java 1.2 como parte de la Java Foundation Classes(JFC).

Swing sigue siendo una parte integral del desarrollo de aplicaciones de escritorio en Java, incluso en un ecosistema en constante evolución que ahora incluye herramientas como Maven para la gestión de proyectos y Jakarta EE para el desarrollo de aplicaciones empresariales. A continuación, exploraremos como Swing se integra con etas tecnologías modernas y como aprovecharlas para crear aplicaciones robustas y eficientes.

10.2. Maven y la gestión de dependencias en Proyectos Swing

Maven es una herramienta de gestión de proyectos y compresión de software, que simplifica el proceso de construcción, reporte y documentación de proyectos Java. Para los desarrolladores que trabajan con Swing, Maven es especialmente útil para gestionar dependencias y asegurar que las bibliotecas necesarias estén siempre actualizadas.

Al comenzar un proyecto Swing con Maven, puedes definir tus dependencias en el archivo **pom.xml**. Maven se encarga de descargar las bibliotecas necesarias desde repositorios remotos y las incluye automáticamente en tu proyecto. Esto no solo ahorra tiempo, sino también asegura que estés utilizando las versiones más recientes y compatibles de las bibliotecas.

Por ejemplo, si deseas incluir la última versión de Swing con otras bibliotecas como JavaFX o las APIs de Jakarta EE. Simplemente se debe agregar las dependencias relevantes en tu archivo **pom.xml**: Maven se encargará del resto, incluyendo la descarga, instalación y configuración de estas bibliotecas en tu entorno de desarrollo.

10.3. Paquetes de las API de Java.

Swing sigue siendo una herramienta esencial para el desarrollo de interfaces graficas de usuario (GUI) en Java, a pesar de la evolución constante del entorno de desarrollo, Este conjunto de bibliotecas permite a los desarrolladores crear aplicaciones de escritorio robustas y eficientes, utilizando componentes gráficos como marcos, botones y campos de texto.

10.3.1. Componentes Clave de Swing

Swing ofrece un conjunto completo de componentes para construir interfaces gráficas, que incluyen.

- **Marcos (Frames):** Estas son ventanas completas que incluyen elementos estándar como barras de título, menús, y botones de control (maximizar, minimizar y cerrar). Los marcos son la base sobre la cual se construyen las aplicaciones de escritorio en Swing.
- **Contenedores (Containers):** Los contenedores son componentes que pueden albergar otros elementos GUI. Ejemplos de contenedores incluyen paneles

JPanel y ventanas internas **JInternalFrame**, que organizan y estructuran la disposición de otros componentes en la interfaz.

- **Botones (Buttons):** Elementos interactivos que permiten a los usuarios ejecutar comandos con un simple click. Swing ofrece varios tipos de botones, como **Jbutton** para acciones generales y **JToggleButton** para alternar estados.
- **Etiquetas (labels):** Las etiquetas **JLabel** son componentes sencillos que muestran texto o imágenes en la interfaz, útiles para describir otros componentes o mostrar información estática.
- **Campos de texto y áreas de texto (Text Fields y Text Areas): JTextField** permite la entrada de una sola línea de texto, mientras que **TextArea** ofrece un espacio más amplio para la entrada y la visualización de múltiples líneas de texto.
- **Listas Desplegables (Combo Boxes):** Los **JComboBox** permiten al usuario seleccionar una opción entre varias disponibles, mostrado una lista desplegable cuando es necesario.

Además de estos componentes, Swing ofrece una gama completa de herramientas para construir interfaces gráficas, incluyendo contenedores como paneles y ventanas, así como menús y barras de menús que enriquecen la experiencia de usuario.

Java organiza sus clases en grupos de categorías relacionadas, conocidos como paquetes, que forman parte de lo que se llama la Java Application Programming Interface (API) o la biblioteca de clases Java. Swing ha sido una parte integral de Java Standard Edition (JSE) desde la versión Java SE 1.2, y sus clases se encuentran en la jerarquía de paquetes **java.swing**. También forma parte de las Java Foundation Classes(JFC), que son la base para crear interfaces gráficas ricas en aplicaciones Java.

Para utilizar las clases de Swing en tu proyecto, necesitas importar los paquetes correspondientes. Esto se puede hacer mediante instrucciones especificas de importación o utilizando una sentencia general como:

```
//Importamos la librería javax.swing
import javax.swing.*;
```

Si un programa incluye declaración como el ejemplo siguiente:

```
//Importamos la liberia javax.swing y JFrame
import javax.swing.JFrame;
```

10.3.2. Swing en el contexto moderno en Java

Desde su inclusión en Java Standard Edition (JSE)1.2, Swing se ha mantenido relevante gracia a su flexibilidad y capacidad de personalización. Aunque JavaFx ha sido promovido como el sucesor de Swing para el desarrollo de GUIs más modernas y ricas en medios, Swing sigue siendo ampliamente utilizado, especialmente en aplicaciones empresariales donde la estabilidad y la compatibilidad a largo plazo son cruciales.

En el contexto actual, muchos desarrolladores integran Swing con herramientas modernas como Maven para la gestión de dependencias y la construcción de proyectos. Maven facilita la integración de Swing en proyectos complejos, permitiendo un manejo eficiente de bibliotecas y versiones, lo que asegura que los componentes de tu aplicación estén actualizados y compatibles.

En el entorno Java es vasto y rico en funcionalidades, con una enorme cantidas de paquetes disponibles. En las versiones más recientes de Java SE, el número de paquetes supera los 200, cada uno compuesto por una colección de clases o interfaces que ofrecen soluciones a diferentes necesidades de desarrollo. Estos paquetes se organizan en diversas categorías claves:

- **Bibliotecas del lenguaje y utilidades:** Incluyen paquetes fundamentales como **java.lang** y **java.util**, que proporcionan clases esenciales para el lenguaje Java y herramientas comunes para la manipulación de datos, colecciones y más.
- **Bibliotecas base o núcleo:** Paquetes como **java.applet** y **java.io** forman parte del núcleo de Java, ofreciendo capacidades para entrada/salida de datos y la creación de aplicaciones applet.
- **Bibliotecas de integración:** Paquetes como **java.sql** y **javax.sql** están diseñados para facilitar la interacción con base de datos, permitiendo la gestión, consultas y operaciones de base de datos dentro de las aplicaciones Java.
- **Bibliotecas de Interfaces de Usuario y otras APIs esenciales en Java:** El desarrollo de interfaces de usuario y otras funcionalidades clave en Java se apoya en una serie de bibliotecas que proporcionan las herramientas necesarias para crear aplicaciones interactivas y seguras.
- **Bibliotecas de interfaces de Usuario AWT:** El Abstract Windows Toolkt(AWT) es una de la bibliotecas más antiguas en Java, diseñada para crear interfaces gráficas de usuario GUIs. Los paquetes principales incluyen **java.awt** y **java.awt.color,** que ofrecen clases y métodos para manejar gráficos, colores y otros elementos básicos de la interfaz de usuario. Aunque AWT ha sido una gran

medida remplazado por Swing y JavaFX en aplicaciones modernas, sigue siendo la base de muchos componentes gráficos, proporcionando compatibilidad con las interfaces nativas del sistema operativo.

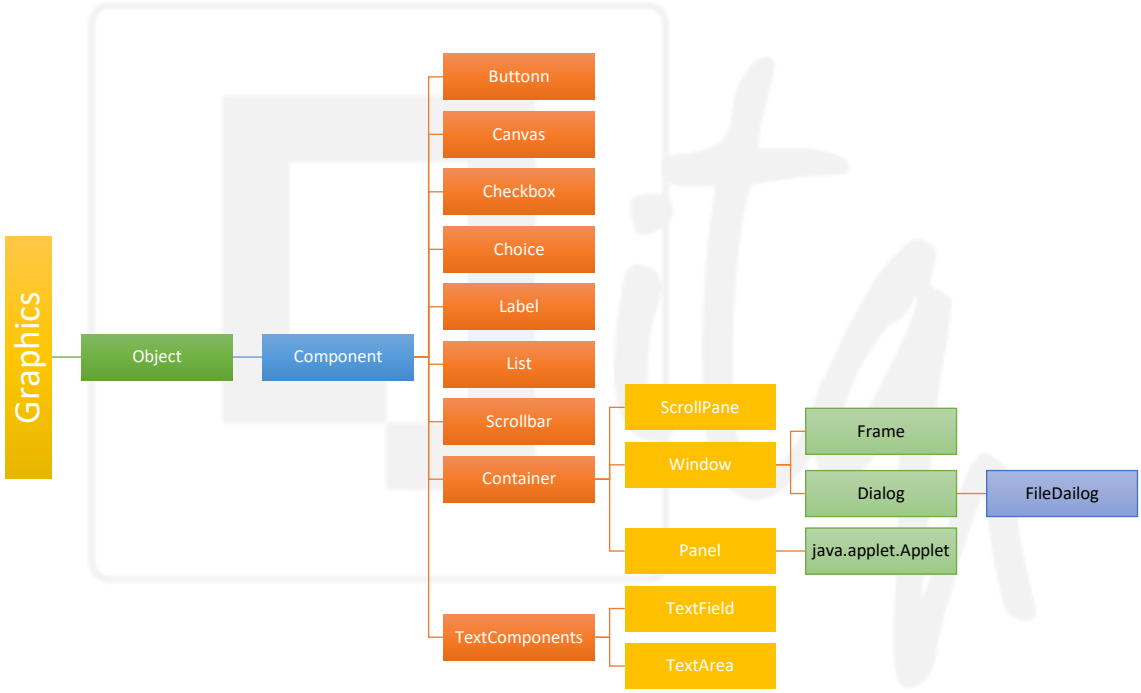
- **Bibliotecas de Interfaces de Usuario Swing:** Swing es la evolución de AWT, introduciendo una mayor flexibilidad y un conjunto más amplio de componentes GUI, Los paquetes **java.swing** y **java.swing.event** contienen todo lo necesario para crear interfaces gráficas ricas y personalizables, desde simples botones y etiquetas hasta menús complejos y paneles avanzados. A pesar de la aparición de JavaFX, Swing sigue siendo popular en aplicaciones de escritorio por su estabilidad y amplio soporte en el mundo Java.
- **Bibliotecas para invocación remota de métodos (RMI) y CORBA:** La invocación remota de métodos (RMI) es una tecnología que permite a los desarrolladores de Java construir aplicaciones distribuidas, donde los objetos pueden interactuar a través de la red. Los paquetes **java,rmi** y **java.rmi.CORBA** proporcionan la infraestructura necesaria para realizar las comunicaciones remotas. Aunque CORBA ha perdido relevancia en favor de tecnologías más modernas como RESTful APIs y microservicios, RMI sigue siendo útil en ciertas aplicaciones distribuidas que requieren un alto nivel de integración con Java.
- **Bibliotecas de seguridad:** Java se toma muy en serio la seguridad, y las bibliotecas de seguridad en Java están diseñadas para proteger aplicaciones de una amplia gama de amenazas. Estas bibliotecas incluyen clases para manejar criptografía, autenticación, autorización y políticas de seguridad. En las versiones actuales Java, se han ampliado para incluir soporte para algoritmos criptográficos más robustos y modernos, así como mejoras en la gestión de claves y certificados.
- **Bibliotecas para manejo XML:** El manejo de datos en formato XML es crucial en muchas aplicaciones empresariales, y Java ofrece una variedad de bibliotecas para este propósito. Estas incluyen clases para la creación, manipulación y validación de documentos XML. Con la evolución de los estándares web y la necesidad de interoperabilidad entre sistemas, las bibliotecas XML en Java ha sido actualizadas para ofrecer un mejor rendimiento y compatibilidad con las últimas especificaciones de XML.

10.4. Entendiendo la Jerarquía de Clases AWT

La jerarquía de clases AWT está diseñada de manera que cada componente gráfico (como botones, cuadros de texto o ventanas) es una subclase de la clase base **Component**. Esta estructura jerárquica permite que todos los elementos gráficos compartan características comunes, como la capacidad de se renderizados en una pantalla o de reaccionar a eventos de usuario.

A continuación, exploraremos las clases clave en esta jerarquía y su relevancia en el desarrollo moderno de aplicaciones Java, como se muestra en la figura 65.

Figura 65.
Jerarquía clase AWT



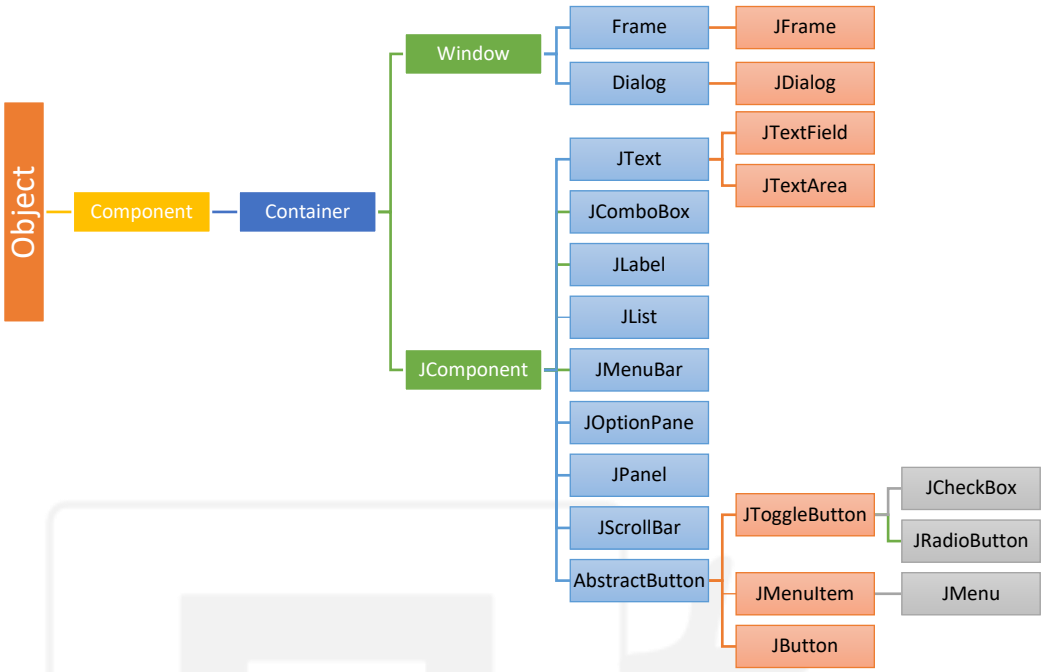
Nota: La imagen me muestra la jerarquía de clases de AWT.

10.5. Entendiendo la Jerarquía de clases Swing

La jerarquía de clases en Swing está diseñada para proporcionar una organización lógica y flexible, que permite a los desarrolladores construir interfaces gráficas de manera modular y escalable. Las clases Swing se derivan de la jerarquía de AWT, pero extienden sus capacidades como componentes más avanzados y personalizables.

Vamos a desglosar las principales clases en la jerarquía de Swing y explorar cómo se utilizan en el desarrollo moderno de aplicaciones Java como se muestra en la figura 66.

Figura 66.
Jerarquía clases Swing



Nota: La imagen muestra la jerarquía de clases Swing

10.6. Crear un marco o clase JFrame

El marco o **JFrame** es uno de los componentes fundamentales en la construcción de aplicaciones de escritorio con Swing en Java. Un **JFrame** actúa como la ventana principal de una aplicación, proporcionando un contenedor donde se alojan todos los demás componentes de la interfaz de usuario, como botones, menús, etiquetas y paneles.

10.6.1. Entendiendo JFrame: La ventana Principal

En el contexto de Swing un **JFrame** es una ventana independiente con características estándar como bordes, una barra de título, y botones para cerrar, minimizar y maximizar. Al ser el punto de partida para la mayoría de las aplicaciones gráficas, el **JFrame** debe ser configurado cuidadosamente para asegurar que la interfaz sea intuitiva y funcional.

Para crear un **JFrame,** se utiliza la clase **JFrame** esta clase se encuentra en el paquete **javax.swing**. Esta clase se extiende **java.awt.Frame** y proporciona toda la funcionalidad necesaria para gestionar una ventana dentro de una aplicación Java.

10.7. Creación básica de un JFrame

Crear un **JFrame** es una aplicación moderna es un proceso sencillo pero fundamental. Los marcos no se pueden visualizar automáticamente, para poder mostrarlos se debe utilizar el método `setVisible(true)` por ejemplo la clase `MarcoHolaMundo` se deriva de **JFrame,** un objeto de esta clase proporciona un lienzo para situar componentes

como un botón, un campo de texto, etc. Para declara la clase se debe realizar de la siguiente manera:

```
public class MarcoHolaMundo extends JFrame
```

Para crear el marco se debe instanciar le nuevo objeto de la clase y decirle a la ventana que se haga visible.

```
MarcoHolaMundo ventana = new MarcoHolaMundo();
ventana.setVisible(true);
```

10.7.1. Métodos propios de JFrame

El **JFrame** es uno de los componentes fundamentales en la creación de aplicaciones de escritorio en Java. Además de actuar como la ventana principal de una aplicación, **JFrame** ofrece una amplia variedad de métodos que permiten personalizar, gestionar y manipular el comportamiento y la apariencia de la ventana.

- **JFrame ():** constructor por defecto, este método crea un marco sin título.
- **JFrame (String titulo):** constructor para crear un marco con un título.
- **setTitle(String titulo):** Uno de los métodos más simples pero muy esencial es **setTitle.** Este método se utiliza para establecer el texto que aparecerá en la barra de título de la ventana. El titulo suele ser lo primero que los usuarios ven, por lo que debe ser claro y representativo de la funcionalidad de la aplicación.

  ```
  //Inicializamos el nuevo objeto de la clase MarcoHolaMundo
  MarcoHolaMundo ventana = new MarcoHolaMundo();
  //Se coloca un título al marco con el método setTitle
  ventana.setTitle("Hola mundo desde un Frame");
  ```

- **setIconImage(Image i):** Este método coloca una imagen i como icono de la ventana. La clase **JFrame** de Java es una herramienta poderosa y sencilla para personalizar la apariencia de una aplicación de escritorio. El uso de un ícono distintivo no solo mejora la identidad visual de la aplicación, sino que también proporciona una mejor experiencia de usuario, al permitir que la aplicación se reconozca fácilmente entre otras ventanas abiertas.

  El método **setIconImage** toma un objeto de tipo **Image** como argumento. Este objeto representa la imagen que deseas utilizar como ícono de la ventana. La clase **Image** forma parte del paquete **java.awt,** y puedes obtener una instancia de **Image** de varias maneras, siendo una de las más comunes la carga de una imagen desde un archivo en el sistema.

```
//Cargamos la imagen desde un archivo
ImageIcon icono=new ImageIcon("Imagenes/icono.png");
Image imagen = icono.getImage();
//Establecer la imagen como ícono del JFrame
ventana.setIconImage(imagen);
```

- **setDefaultCloseOperation(int operación):** Este método es uno de los aspectos fundamentales de la gestión de ventanas es definir lo que sucede cuando el usuario quiere cerrar la ventana. El método **setDefaultCloseOperation**, controla este comportamiento, permitiendo elegir entre varias opciones.
    - **JFrame.EXIT_ON_CLOSE**: Termina la ejecución del programa.
    - **JFrame.DISPOSE_ON_CLOSE**: Cierra la ventana actual pero no termina la aplicación si hay otras ventanas abiertas.
    - **JFrame.DO_NOTHING_ON_CLOSE**: No realiza ninguna acción, dejando el control al programador.
    - **JFrame.Hide_ON_CLOSE**: Oculta la ventana, pero no la destruye.

```
//Cierra la aplicación al cerrar la ventana
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

- **Container getContentPane():** Este método es un componente esencial en el desarrollo de interfaces gráficas de usuario (GUI) en Java, especialmente cuando se trabaja con la clase JFrame. Este método facilita el acceso y la manipulación del contenedor principal de un JFrame, que es donde se colocan todos los componentes visuales de la interfaz, como botones, etiquetas, paneles, entre otros. Entender a fondo cómo utilizar este método es crucial para cualquier desarrollador que desee crear aplicaciones de escritorio robustas y bien organizadas.

- **setUndesorated(boolen b):** Este método por ser de tipo booleano devuelve true o false, si la respuesta es true quita los adornos del marco, es decir los bordes del marco.

- **setResizable(boolean r):** Este método de igual manera devuelve un true o false si al devolver un true el método se puede cambiar el tamaño del marco.

- **add(Component a):** Añadir componentes al **JFrame** es una de las tareas más comunes. El método **add** permite agregar un componente directamente a la ventana, aunque para un control más preciso es recomendable trabajar con el método **ContentPane** del **JFrame.**

```
//Con el método add añadimos componentes al JFrame
ventana.add(new JButton("Aceptar"));
```

- **setVisible(boolean b):** El método **setVisible** controla la visibilidad del **JFrame**. Este método es esencial para que le ventana aparezca en la pantalla después de haber sido configurada. El parámetro **true** muestra la ventana, mientras que el parámetro **false** la oculta.

```
//El método setVisible muestra la ventana en la pantalla
ventana.setVisible(true);
```

- **setLocation(int x, int y) y setLocationRelativeTo(component c):** El método **setLocation** permite especificar la posición de la ventana en la pantalla en coordenadas **x** e **y** en este caso siendo **(0,0)** en este caso en la parte superior izquierda de la pantalla.

```
//Colocamos la ventana en la posición 100 pixeles del borde
izquierdo superior
ventana.setLocation(100,100);
```

El método **setLocationRelativeTo** posiciona el **JFrame** en relación con otro componente o en el centro de la pantalla si se pasa como parámetro **null** como argumento.

```
//Centramos la ventana en la mitad de la pantalla pasando el
parametro null
ventana.setLocationRelativeTo(null);
```

- **setResizable(boolean r):** Con este método se puede controlar si el usuario puede redimensionar la ventana. Establecer este parámetro en **false** puede ser útil cuando la interfaz de usuario está diseñada para funcionar solo un tamaño especifico. Este método es especialmente relevante en aplicaciones con un diseño fijo, donde permitir cambios de tamaño podría distorsionar la interfaz.

```
//El método setResizable impide que la ventana sea
redimensionada
ventana.setResizable(false);
```

- **setJMenuBar(JMenuBar menubar):** Un **JFrame** puede contener una barra de menús, que a su vez puede contener múltiples menús **JMenu** y elementos de menú **JMenuItem**. El método **setMenuBar** s utiliza para añadir esta barra de menús al **JFrame.**

  Incluir menús es fundamental para aplicaciones de escritorio que ofrecen múltiples funciones y configuraciones, proporcionando un acceso organizado y estructurado a estas opciones.

```
//Inicializamos un nuevo objeto de la clase JMenuBar
JMenuBar barraMenu=new JMenuBar();
//Inicializamos el boton del menu Archivo
JMenu menuArchivo=new JMenu("Archivo");
//Añadimos el botón del menú a la barra de menús
barraMenu.add(menuArchivo);

//Seteamos el botón en la barra de menú
ventana.setJMenuBar(barraMenu);
```

- **dispose():** Finalmente el método **dispose** se utiliza para liberar los recursos utilizados por el **JFrame** cuando ya no es necesario. Esto es crucial en aplicaciones complejas o de larga duración, donde el manejo eficiente de los recursos es clave para evitar fugas de memoria y otros problemas de rendimiento. A diferencia de **setVisible(false),** que solo oculta la ventana, **dispose** cierra la ventana y libera todos los recursos asociados, haciendo que el **JFrame** ya nos sea reutilizable.

```
//Libera los recursos del marco
ventana.dispose();
```

Ejercicio 10.1

Crear una aplicación en Java donde se crea una ventana de 250 * 100 pixeles de tamaño.

La clase principal de la ventana se deriva de la clase JFrame, en el constructor se debe incluir el tamaño en pixeles, la posición y el título, si no se indica la posición, asume (0,0). Una vez creada la ventana se debe cerrar la operación que la aplicación realice asea salir de la ejecución (EXIT_ON_CLOSE). En caso contrario, el hilo de la ejecución se seguirá ejecutando, consumiendo recursos del sistema.

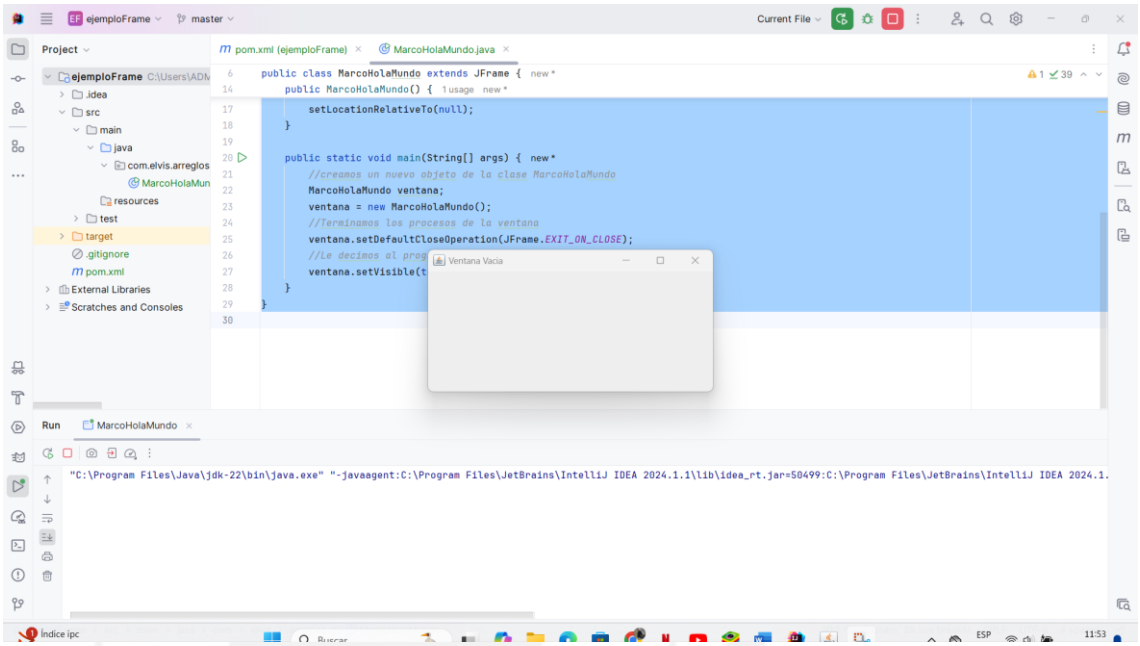
```
package com.elvis.arreglos;

import javax.swing.*;
import java.awt.*;

public class MarcoHolaMundo extends JFrame {
    //Declaramos e inicializamos dos variables de tipo entero
    //ancho y alto donde pondremos las medidas de la ventana
    private static final int ancho=400 , alto=200;
    //Implementamos un método constructor que no recibe ningún
parametro
```

```
        //Aquí seteamos el título de la ventana
        //las medidas de la ventana
        //y la posición de la ventana
        public MarcoHolaMundo() {
            setTitle("Ventana Vacía");
            setSize(ancho, alto);
            setLocationRelativeTo(null);
        }

        public static void main(String[] args) {
            //creamos un nuevo objeto de la clase MarcoHolaMundo
            MarcoHolaMundo ventana;
            ventana = new MarcoHolaMundo();
            //Terminamos los procesos de la ventana
            ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            //Le decimos al programa que me muestre la ventana
            ventana.setVisible(true);
        }
}
```

Ejecución del programa

Al ejecutar el programa nos muestra una ventana vacía con el título centrada a la mitad del monitor.

Figura 67.
Ejecución del programa



Nota: La imagen nos muestra la salida de pantalla de la ejecución del programa.

Ejercicio 10.2.

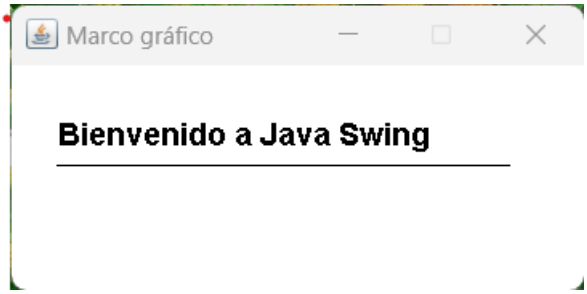
Crear un programa en Java donde se crea una venta se posicione a la mitad de la pantalla, dentro de la ventana dibujar un mensaje con la clase **drawString()** que diga **Bienvenido al mundo de JavaSwing.** Implementar un método **Paint()** de la clase Component que proporciona el contexto gráfico, representado por la clase **Graphics,** la llamada al método **paint()** la realiza directamente el sistema cuando se muestra en pantalla. El constructor de la clase ventana recibe su nombre, lo dimensiona. Establece que no se puede cambiar el tamaño y lo hace visible.

```
import javax.swing.*;
import java.awt.*;

public class Ventana extends JFrame {
    private static final int ancho=300, alto=150;
    /*Implementamos un constructor que recibe un parametro de
tipo String titulo
    * el constructor setea el tamaño de la ventana
    * Le decimos que no se pude cambiar de tamaño
    * Le damos un posición central
    * le pedidos que muestre la ventana*/
    public Ventana(String titulo)  {
```

```
        super(titulo);
        setSize(ancho, alto);
        setResizable(false);
        setLocationRelativeTo(null);
        setVisible(true);
    }
    public void paint(Graphics g){
        //inicializamos y creamos un objeto de la clase Font
        Font tipoLetra= new Font("Courier", Font.BOLD, 15);
        //Seteamos el tipo de letra para la ventana
        g.setFont(tipoLetra);
        //Dibujamos el mensaje en la ventana
        g.drawString("Bienvenido a Java Swing", ancho/10,70);
        //dibujamos una linea debajo del mensaje
        g.drawLine(ancho/10,80, ancho/10 +225,80);

    }

    public static void main(String[] args) {
        //Creamos un objeto de la clase Ventana
        Ventana ventanaConMensaje;
        //instanciamos el objeto de tipo Ventana con el título
de la ventana
        ventanaConMensaje= new Ventana("Marco gráfico");
        //Terminamos el proceso al cerrar la ventana

ventanaConMensaje.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
;
    }
}
```

Salida de pantalla, ejecución del programa.

Figura 68.
Ejecución del programa



Nota: La imagen nos muestra la salida de pantalla una ventana donde se dibuja "Bienvenido a Java Swing" el texto se encuentra dibujado una línea debajo del texto.

10.8. Administradores de diseño Java Swing

Cuando trabajas con interfaces gráficas en Java usando Swing, una de las tareas más comunes es organizar los componentes visuales dentro de un marco o panel. Aquí es donde los administradores de diseño juegan un papel crucial. Imagina que estas diseñando el interior de una casa. Tienes diferentes habitaciones y muebles que debes colocar de manera que no solo se vean bien, sino que también funcionen correctamente. Los administradores de diseño Swing son como los arquitectos de interiores para tus aplicaciones, ayudándote a decidir dónde y cómo colocar cada componente.

Un **administrador de diseño** (o layout manager en inglés) es una herramienta que Swing proporciona para controlar cómo se organizan y dimensionan los componentes dentro de un contenedor. En lugar de posicionar cada componente manualmente, puedes delegar a esta responsabilidad a un administrador de diseño, que se encargará de distribuirlos de manera eficiente según las reglas que le establezcas.

Hay definidos alrededor de siete tipos de layouts que les pasaremos a estudiarlos a continuación (Ahmad Dar, 2020).

10.8.1. FlowLayout

Es el más simple y básico, imagina que tienes un montón de fotos y las colocas una tras otra en una fila. Cuando ya no hay espacio en la fila, comienza una nueva. Esto es exactamente lo que hace **FlowLayout**. Coloca los componentes en una fila horizontal y, si es necesario, pasa a la siguiente fila. Es ideal para formularios sencillos o grupos de botones.

La clase **FlowLayout** dispone de varios constructore, uno de ellos establece la alineación de los componentes: **FlowLayout(int alineación),** la aliniacipon de los componentes de los puede realizar al derecho de la ventana con el método

FlowLayout,RIGHT, al centro de la ventana con el método **FlowLayout.Center** y al lado izquierdo con el método **FlowLayout.LEFT.**

Ejemplo 10.3

```
import javax.swing.*;
import java.awt.*;

public class DisenioFlow extends  JFrame{
    public static void main(String[] args) {
        //creamos un nuevo objeto de la clase JFrame
        JFrame frame= new JFrame("Ejemplo de diseño
FlowLayout");
        frame.setLayout(new FlowLayout());
        //Agregamos los componentes a la ventana
        frame.add(new JButton("Botón 1"));
        frame.add(new JButton("Botón 2"));
        frame.add(new JButton("Botón 3"));
        frame.add(new JButton("Botón 4"));
        frame.add(new JButton("Botón 5"));
        //Seteamos la el tamaño de la ventana
        frame.setSize(300,150);
        //Terminamos el proceso del programa al presionar la X
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //Le decimos a la ventana que se muestre en pantalla
        frame.setVisible(true);
    }
}
```

Ejecución del programa.

Como se muestra en la figura 69 el tamaño de la ventana no permite organizar todos los botones en una sola fila con el diseño **FlowLayout** cuando ya no hay espacio en la primera fila automáticamente los otros dos botones pasan a la segunda fila.

Figura 69.
Diseño FlowLayout

Nota: La imagen muestra cómo se organizan los elementos de frame con el diseño FlowLayout.

10.8.2. BorderLayut

Este administrador divide el contendor en cinco regiones: norte, sur, este, oeste y centro. Es como un esquema de una rosa de los vientos en la que puedes colocar cada componente en cada dirección **BorderLayout** es útil cuando necesitas una estructura básica, como una barra de menú en la parte superior, (Mughal, 2016)un panel de estado en la parte inferior, y contenido principal en el centro. Los componentes distribuidos con este gestor de diseño se ubican en las posiciones: superior(north), inferior(south), derecha (east), izquierda(west) y centro (center), estas posiciones se representan por las constantes:

BorderLayout.CENTER, BorderLayout.NORTH, BorderLayout.EAST, BorderLayout.SOUTH y **BorderLayout.WEST.**

Los componentes se pueden separar de manera horizontal o vertical creando un objeto **BorderLayout** con el constructor:

```
BorderLayout(int separaHorizontal, int separaVertical);
```

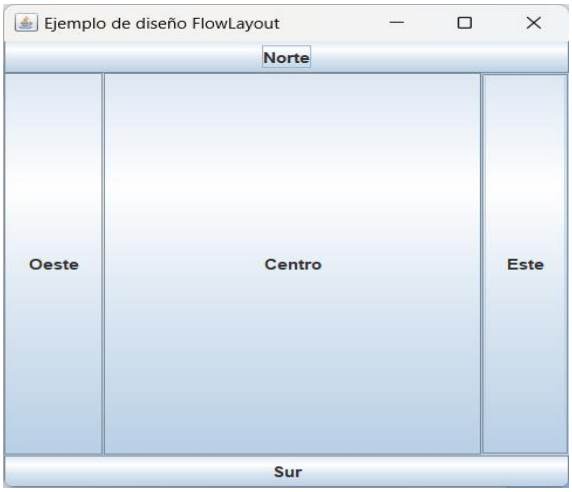
La separación es en pixeles, es necesario tener en cuenta que la unidad de medida de las GUI siempre será ésta.

Ejemplo 10.4

```
import javax.swing.*;
import java.awt.*;

public class DisenioFlow extends  JFrame{
    public static void main(String[] args) {
        //creamos un nuevo objeto de la clase JFrame
        JFrame frame= new JFrame("Ejemplo de diseño
FlowLayout");
        //Seteamos el diseño que vamos a utilizar en el programa
```

```
        frame.setLayout(new BorderLayout());
        //Agregamos los componentes a la ventana y se lo añade
en la parte de arriba de la ventana
        frame.add(new JButton("Norte"), BorderLayout.NORTH);
        //Agregamos los componentes a la ventana y se lo añade
en la parte inferior de la ventana
        frame.add(new JButton("Sur"), BorderLayout.SOUTH);
        //Agregamos los componentes a la ventana y se lo añade
en la parte derecha de la ventana
        frame.add(new JButton("Este"), BorderLayout.EAST);
        //Agregamos los componentes a la ventana y se lo añade
en la parte izquierda de la ventana
        frame.add(new JButton("Oeste"), BorderLayout.WEST);
        //Agregamos los componentes a la ventana y se lo añade
en la parte centro de la ventana
        frame.add(new JButton("Centro"), BorderLayout.CENTER);
        //Seteamos la el tamaño de la ventana
        frame.setSize(400,400);
        //Terminamos el proceso del programa al presionar la X
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //Le decimos a la ventana que se muestre en pantalla
        frame.setVisible(true);
    }
}
```

Figura 70.
Salida de Pantalla BorderLayout



Nota: La imagen nos muestra como ser posicionan cada uno de los elementos en la ventana con diseño BorderLayout.

10.8.3. GridLayout

Imagina una cuadricula de celdas donde cada componente tiene el mismo tamaño. **GridLayout** es perfecto para paneles donde quieres que todos los componentes estén alineados y tengan el mismo espacio, como un tablero de botones en una calculadora.

Al crear el gestor pasa el número de filas/columnas que forman la rejilla en la que se colocan los componentes. Los constructores que se pueden utilizar son:

- **GridLayout():** Coloca los componentes en un única fila y única columna.
- **GridLayout(int f, int c):** Coloca los componentes en cuadrículas de f que corresponde a las filas por c que corresponde a las columnas.
- **GridLayout(int f, int c, int sepHztal, int sepVert),** define la rejilla y establece la separación en pixeles.

Ejemplo 10.5

Crear un programa Java donde se crea una ventana donde se asigna un gestor de tipo **GridLayuot,** en la ventana se coloca seis botones.

El gestor **GridLayout** que se crea define una cuadricula de 4 filas por 3 columnas, con una separación de 25 pixeles, en tanto horizontal y vertical.

```
import javax.swing.*;
import java.awt.*;

public class DisenioGrid extends JFrame {
    //Declaramos e inicializamos la medidas de la ventana en su
ancho y alto
    int ancho=500;
    int alto=400;
    //Implementamos un constructor que no recibe ningún
parámetro

    public DisenioGrid() {
        //Ponemos el titulo a la ventana
        super("Diseño Grid");
        //Creamos el diseño donde se van a colocar los elementos
de la ventana
```

```
        //tiene 4 filas, 3 columnas y va a tener una separación
de 25px
        setLayout(new GridLayout(4, 3,25,25));
        //Añadimos los botones a la ventana según el diseño
creado
        add(new JButton("Primero"));
        add(new JButton("Segundo"));
        add(new JButton("Tercero"));
        add(new JButton("Cuarto"));
        add(new JButton("Quinto"));
        add(new JButton("Sexto"));
        add(new JButton("Septimo"));
        add(new JButton("Octavo"));
        add(new JButton("Noveno"));
        add(new JButton("Decimo"));
        add(new JButton("Decimo Primero"));
        add(new JButton("Decimo Segundo"));
        //Seteamos las dimensiones de la ventana
        setSize(ancho, alto);
        //Mostramos la ventana en la pantalla
        setVisible(true);
    }

    public static void main(String[] args) {
        //Creamos un nuevo objeto de la clase DisenioGrid
        DisenioGrid miVentana= new DisenioGrid();
        //Implementamos un método para poder terminar el proceso
al momento de cerrar la ventana

miVentana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

}
```

Ejecución del programa.

Figura 71.
Diseño GridLayout

Nota: La imagen nos muestra cómo se añaden los elementos en una ventana JFrame con el método GridLayout.

10.8.4. BoxLayout

Este gestor organiza los componentes en una sola fila o columna, similar a una caja que se orienta de manera horizontal o vertical (Garrido Abenza, 2015). Al crear un **BoxLayout**, debes especificar el contenedor donde se ubicarán los componentes y la dirección en la que se alinearán, que puede ser horizontal **BoxLayout.X_AXIS** o vertical **BoxLayout.Y_AXIS**. La sintaxis del constructor se define como:

```
BoxLayout(Container target, int orientación)
```

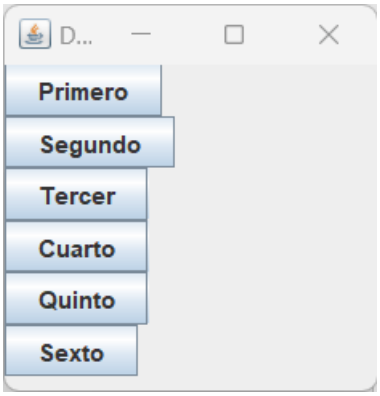
Ejercicio 10.6

Crear un programa en Java donde muestre una venta cuyo gestor de diseño sea tipo BoxLayout. La aplicación crea un gestor de diseño **BoxLayout** asociado a un panel JPanel con orientación vertical, cada etiqueta que se va a insertar, y por último se añade el panel a la ventana.

```
import javax.swing.*;

public class DisenioBoxLayout extends JFrame {
```

```
    int ancho = 200;
    int alto = 200;

    public DisenioBoxLayout()  {
        //Creamos la ventana con un título indicado
        super("Diseño BoxLayout");
        //Creamos un objeto de tipo JPanel
        JPanel panel = new JPanel();
        //Seteamos el panel con el diseño indicado y los
elementos se
        //setean de forma vertical
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
        //Añadimos cada uno de los botenes al panel
        panel.add(new JButton("Primero"));
        panel.add(new JButton("Segundo"));
        panel.add(new JButton("Tercer"));
        panel.add(new JButton("Cuarto"));
        panel.add(new JButton("Quinto"));
        panel.add(new JButton("Sexto"));
        //Añadimos el panel al Frame
        add(panel);
        //Seteamos el ancho del Frame
        setSize(ancho, alto);
        //Pedimos que se muestre la ventana en la pantalla
        setVisible(true);
    }

    public static void main(String[] args) {
        //Creamos un nuevo objeto de la clase DisenioBoxLayout
        DisenioBoxLayout miVentana= new DisenioBoxLayout();
        //Terminamos el proceso del programa al salir de la
aplicación

miVentana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Figura 72.
Diseño BoxLayout



Nota: La imagen nos muestra como se organizan los elementos mediante el diseño BoxLayout.

10.8.5. BoxLayout-Box

El contenedor Box en Java es una herramienta y poderosa que viene con un **BoxLayout** configurando de forma predeterminada. Lo que hace que **Box** sea particularmente útil es qu no necesitas crear un panel por separado, en su lugar, simplemente creas un objeto **Box** y luego agregas directamente los componentes que necesitas (Rodríguez Rancel, 2012).

Además, la clase **Box** proporciona dos métodos estáticos, conocidos como métodos de fábrica, que facilitan la creación de objetos **Box** de manera eficiente. Estos métodos están diseñados para simplifica el proceso de configuración del contenedor, permitiéndote enfocarte más en el diseño y la funcionalidad de la interfaz.

Con las versiones más recientes de Java, como Java 17 y superiores, la funcionalidad de **Box** y **BoxLayout** se han mantenido estable y sigue siendo una opción confiable para el diseño de interfaces gráficas. Java continúa evolucionando con mejoras en el rendimiento, seguridad y nuevas características, como soporte para **records** y **sealed classes**, que facilitan la escritura de código más limpio y seguro. A pesar de estos avances, las herramientas clásicas como **Box** y **BoxLayout** sigue siendo esenciales para los desarrolladores que buscan crear interfaces gráficas flexibles y fácilmente mantenibles (Ernest, 2012).

Ejercicio 10.7

En una ventana principal, se colocan dos Box; uno horizontal y otro de forma vertical cada uno de ellos contiene cuatro botones.

Los botones son objetos JButton, se crearán seis botones, los cuatro primeros se añaden en un contenedor Box horizontal, y los otros cuatro en un contenedor vertical, los

componentes del Box se pueden separar horizontal o verticalmente, llamando a los métodos.

- add(Box.createHorizontalStrut(n))
- add(Box.createVerticalStrut(n))

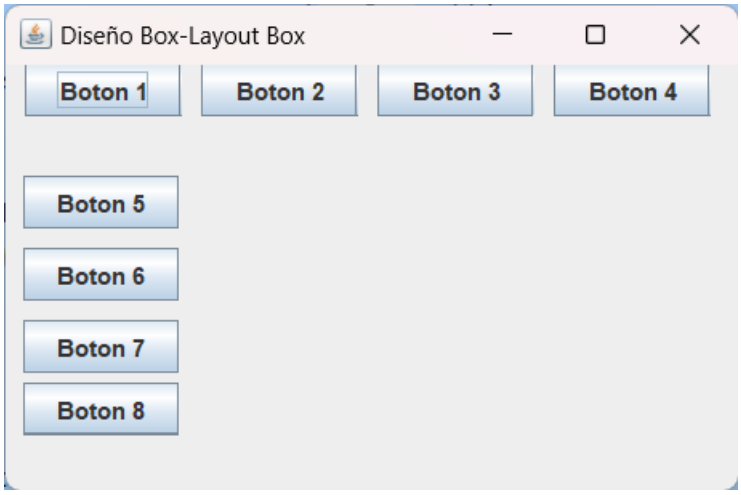
En número de pixeles es n, también, se puede crear una zona rígida de separación entre componentes, la ventana por omisión tiene asociado el gestor de diseño BorderLayout, pues el contendor horizontal tendrá que estar situado en la parte norte de la ventana y el vertical debe estar en la parte central de mi ventana.

```
import javax.swing.*;
import java.awt.*;

public class DisenioDosCajas extends JFrame {
    int ancho=380;
    int alto=250;

    public DisenioDosCajas() {
        super("Diseño Box-Layout Box");
        JButton b1 = new JButton("Boton 1");
        JButton b2 = new JButton("Boton 2");
        JButton b3 = new JButton("Boton 3");
        JButton b4 = new JButton("Boton 4");
        JButton b5 = new JButton("Boton 5");
        JButton b6 = new JButton("Boton 6");
        JButton b7 = new JButton("Boton 7");
        JButton b8 = new JButton("Boton 8");
        Box cajaHorizontal = Box.createHorizontalBox();
        cajaHorizontal.add(Box.createHorizontalStrut(10));
        cajaHorizontal.add(b1);
        //Separamos horizontalmente 10 px
        cajaHorizontal.add(Box.createHorizontalStrut(10));
        cajaHorizontal.add(b2);
        //Separamos horizontalmente 10px
        cajaHorizontal.add(Box.createHorizontalStrut(10));
        cajaHorizontal.add(b3);
        //zona rigida, separación horizontal
        cajaHorizontal.add(Box.createRigidArea(new Dimension(10,
```

```
10)));
        cajaHorizontal.add(b4);
        add(cajaHorizontal, BorderLayout.NORTH);

        //Creamos la caja vertical
        Box cajaVertical = Box.createVerticalBox();
        //Añadimos el espacio
        cajaVertical.add(Box.createVerticalStrut(30));
        //añadimos el boton 5
        cajaVertical.add(b5);
        //separación vertical
        cajaVertical.add(Box.createVerticalStrut(10));
        //Ñdimos en boton 6
        cajaVertical.add(b6);
        cajaVertical.add(Box.createVerticalStrut(10));
        cajaVertical.add(b7);
        cajaVertical.add(Box.createRigidArea(new
Dimension(5,5)));
        cajaVertical.add(b8);
        //añadimos la caja al centro
        add(cajaVertical, BorderLayout.CENTER);
        //Seteamos el tamaño del frame
        setSize(ancho, alto);
        setVisible(true);
    }

    public static void main(String[] args) {
        //creamos el nuevo objeto de la clase DisenoDosCajas
        DisenioDosCajas ventana = new DisenioDosCajas();
        //Al cerrar terminamos el proceso del programa
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Ejecución del programa

Figura 73.
Elementos en dos cajas



Nota: La imagen nos muestra cómo se colocan dos cajas con distintas posiciones.

10.8.6. Desactivar el gestor de posicionamiento.

Cuando se desarrolla interfaces gráficas en Java Swing, los gestores de posicionamiento, como **BorderLayout**, **GridLayout,** y **BoxLayout**, son herramientas fundamentales que facilitan la organización automática de los componentes dentro de un contenedor. Sin embargo, hay situacionesen las que estos gestores pueden resultar limitantes, especialmente cuando necesitas un control absoluto sobre la posición y el tamaño de cada componente en la interfaz.

Es aquí donde entra en juego la posibilidad de desactivar el gestor de posicionamiento. Desactivar el posicionamiento significa optar por no utilizar ninguno de los gestores predefinidos en Java Swing, como **FlowLayout** o **GridBagLayout,** y en su lugar, asumir el control manual de la disposición de los componentes. Al hacerlo, puedes establecer exactamente dónde se colocan los componentes en un contenedor, usando coordenadas absolutas. Esto se conoce comúnmente como posicionamiento absoluto (Шилдт, 2019).

Los métodos utilizados para trabajar con coordenadas absolutas son:

- **setSize(int ancho, int alto)**
- **setLocation(int x, int y)**
- **setBounds(int x, int y, int ancho, int alto)**, fija la posición y el tamaño del componente

Por ejemplo:

```
import javax.swing.*;

public class PosicionamientoAbsoluto {
    public static void main(String[] args) {
        //Creamos la ventana principal de la aplicación con un
título
        JFrame ventana= new JFrame("Posicionamiento Absoluto");
        //Terminamos el proceso de la aplicación al cerrar la
ventana principal
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Desactivamos el gestor de posicionamiento
        ventana.setLayout(null);

        //Creamos y configuramos los componente con posiciones y
tamaños exactos
        JButton boton1= new JButton("Boton 1");
        //Seteamos el botón con un posicionamiento de x, y,
ancho y alto
        //El parámetro x y es el punto donde inicia el elemento
        boton1.setBounds(50, 50, 100, 50);

        //Creamos un segundo botón
        JButton boton2= new JButton("Boton 2");
        boton2.setBounds(200, 50, 100, 50);

        //Creamos un tercer elemento un campo de texto
        JTextField campoTexto = new JTextField();
        campoTexto.setBounds(50,150,250,30);

        //Setemos el tamaño de la ventana
        ventana.setSize(350,300);

        //Añadimos los componentes a la ventana principal
        ventana.add(boton1);
        ventana.add(boton2);
        ventana.add(campoTexto);
```

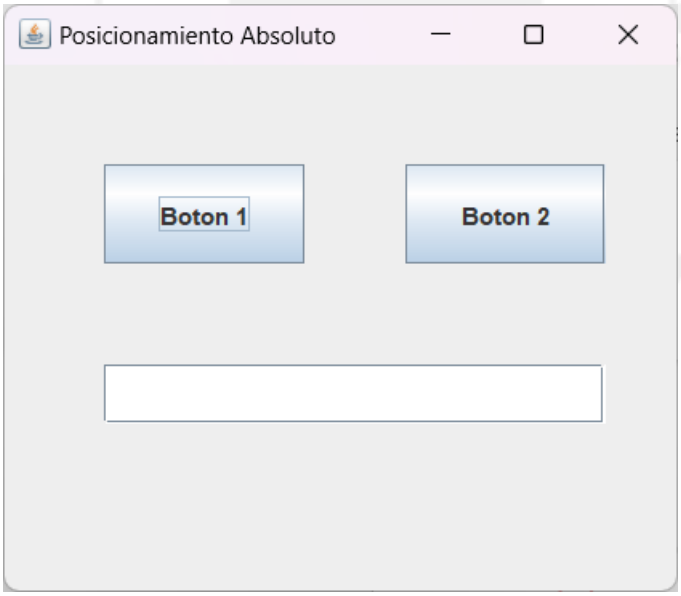
```
        //Mostramos la ventana en pantalla
        ventana.setVisible(true);
    }
}
```

Salida de pantalla

Cuando se desactiva el gestor de posicionamiento te da un control total, también implica asumir la responsabilidad de algunos desafíos.

- Redimensionamiento de la Ventana
- Responsividad
- Mantenimiento

Figura 74.
Posicionamiento Absoluto



Nota: La imagen nos muestra como el programador al desactivar el gestor de posicionamiento toma control de la ventana y puede añadir los elementos en la posición que se desea.

10.9. Botones y etiquetas en Java Swing

Cuando se desarrolla una aplicación en Java Swing, **botones** y **etiquetas** son dos de los componentes más básicos y esenciales que se utiliza para crear una interfaz gráfica de usuario (GUI) funcional e intuitiva. Estos componentes permiten a los usuarios interactuar con tu aplicación de manera sencilla y directa.

10.9.1. Botones en Java Swing.

Los componentes **botones** son elementos interactivos que los usuarios pueden hacer **click** para ejecutar acciones específicas. En Java Swing los botones se representan por la clase **JButton**. Un botón puede contener texto, un ícono o ambos, y su principal función es disparar eventos cuando el usuario interactúa con él, generalmente al hacer click (Грин, 2019).

10.9.2. Métodos de AbstractButton

Los métodos más interesantes de esta clase abstracta son:

- **setText(String texto):** coloca el texto que va a identificar al botón.
- **getText():** obtiene y devuelve el texto que está asociado al botón.
- **isSelect():** Devuelve true si el botón se seleccionó.
- **setSelected(boolean b):** selecciona el botón.
- **doClick(int tiempo):** elige el botón durante un tiempo milisegundos.
- **setIcon(Icon icono):** este método permite asociar un icono al botón.
- **setMnemonic(int mnemonic):** este método permite relacionar una tecla al botón, esto permite seleccionar el botón pulsando la tecla ALT y la tecla asociada.
- **addActionListener(ActionListener oyente):** Asigna un oyente para detectar la acción realizada sobre el botón.

JButton.

La clase **JButton** representa el botón común, este botón se crea especificando una cadena, un icono o aun sin especificar algún elemento, los constructores de la clase son:

- `JButton()`
- `JButton(String texto)`
- `JButton(String texto, Icon icono)`

La clase deriva d **AbstractButton** por los cual todos sus métodos están disponibles en la clase **JButton**.

- JButton boton1, boton2, boton3;
- Boton1= new JButton();
- Boton2=new JButton(“Verde”);
- Boton3=new JButton(new GuardarIcon());

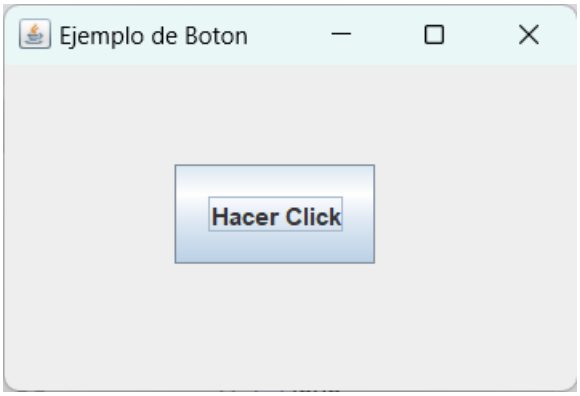
Ejercicio 10.8.

Crear una ventana principal en Java donde se crea e inserte un botón en el centro de la veta, el botón debe tener un texto "**Hacer CLick".**

```
import javax.swing.*;

public class Boton {
    public static void main(String[] args) {
        //Creamos una ventana principal de la aplicación
        JFrame ventana = new JFrame("Ejemplo de Botón");
        //Terminamos el proceso del programa al cerrar la
ventana100
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //Seteamos el tamaño de la ventana
        ventana.setSize(300,200);
        ventana.setLayout(null);
        //Creamos un botón con un texto
        JButton boton = new JButton("Hacer Click");
        boton.setBounds(85,50,100,50);

        //Añadimos el botón a la ventana
        ventana.add(boton);

        //Mostramos la ventana en la pantalla
        ventana.setVisible(true);
    }
}
```

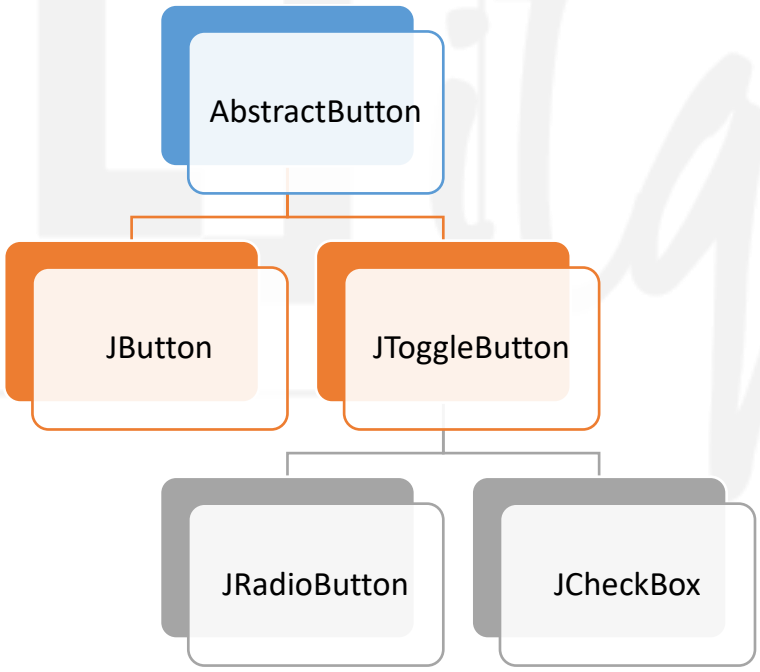
Ejecución del programa

Figura 75.
JButton



Nota: La imagen nos muestra la salida de pantalla del programa donde creamos un botón y lo añadimos a la ventana principal.

Figura 76.
Clase AbstractButton



Nota: La imagen muestra la jerarquía de la clase botón en Java Swing.

10.9.3. Botones con dos estados

JToggleButton es la clase principal para los botones que se pueden alternar entre dos estados, como encendido y apagado. **JRadioButton**, que es una subclase de **JToogleButton,** se usa para crear grupos de botones donde solo se puede seleccionar una opción a la vez. Para agrupar estos botones y asegurarte de que solo uno pueda estar seleccionado a la vez. Para agrupar estos botones y asegurarte de que solo uno pueda

estar seleccionado a la vez, utilizas la clase **ButtonGroup**. Primero se crea un objeto **ButtonGroup** usando su constructor sin parámetros. Luego se añade cada **JRadioButton** al grupo utilizando el método **add(AbstractButton b)** de **ButtonGroup.**

Estos son los métodos que se utilizan:

- JRadioButton()
- JRadioButton(String mensaje)
- JRadioButton(String mensaje, boolean selección), si se selecciona el botón cambia de estado a true

Ejercicio 10.9

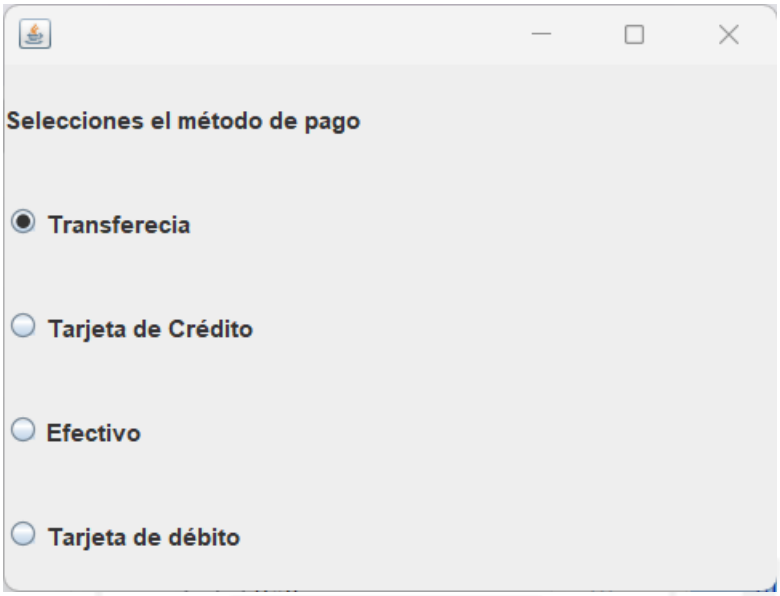
Crear un programa en Java donde se cree una venta principal con cuatro radios botones seleccionadores, estos botones determinan que un usuario debe elegir el método de pago de las compras realizadas: en transferencia, tarjeta de crédito, efectivo y tarjeta de débito.

El constructor de la clase es donde se crea los cuatro botones de radio que se agrupan utilizando un ButtonGroup, mientras se va creando los botones de radio, estos se den añadir al Panel y al ButtonGroup (Toro, 2018).

```
import javax.swing.*;
import java.awt.*;

public class RadioBoton extends JFrame {
    ButtonGroup grb;
    JRadioButton radioButton1, radioButton2, radioButton3,
radioButton4;

    public RadioBoton() {
        //Creamos el nuevo objeto para crear un grupo de botones
        grb = new ButtonGroup();
        //Seteamos el tipo de diseño de pantallas con 4 filas y
una columna
        setLayout(new GridLayout(5, 1));
        //Añadimos un JLabel que diga elegir la opción
        add(new JLabel("Selecciones el método de pago"));
        //Se crea el radio botón, y al mismo tiempo se añade al
panel y a la agrupación
        radioButton1 = new JRadioButton("Transferecia", true);
```

```
        //añadimos el radio botón al panel
        add(radioButton1);
        //añadimos el botón al grupo
        grb.add(radioButton1);
        radioButton2 = new JRadioButton("Tarjeta de Crédito",
false);
        add(radioButton2);
        grb.add(radioButton2);
        //creamos el objeto JRadioBotton
        radioButton3 = new JRadioButton("Efectivo", false);
        add(radioButton3);
        grb.add(radioButton3);

        //Creamso el objeto de radioBotton 4
        radioButton4 = new JRadioButton("Tarjeta de débito",
false);
        add(radioButton4);
        grb.add(radioButton4);

        setSize(400, 300);
        setVisible(true);
    }

    public static void main(String[] args) {
        RadioBoton ventana = new RadioBoton();
    }
}
```

Figura 77.
JRadioButton



Nota: La imagen nos muestra los cuatro JRadioButton para poder seleccionar una opción, que es la ejecución del código superior.

10.9.4. JComboBox

El componente **JComboBox** es un elemento de Java Swing que te permite crear un menú desplegable interactivo. Es una excelente opción cuando quieres ofrecer al usuario una lista de opciones, pero sin ocupar demasiado espacio en la interfaz del usuario. Los usuarios pueden seleccionar una opción de la lista, y en algunos casos, inclusos escribir una entrada personalizada si el cómo box está configurado para ser editable (Joyanes Aguilar, Programación en Java 6 algoritmos, programación orientada a objetos, 2011).

Los distintos constructores que se puede utilizar con este elemento son los siguientes:

- **JComboBox:** Este constructor crea un **JComboBox** vacío, que luego se puede llenar con elementos de manera programática.

```
JComboBox<String> comboBox=new JComboBox<>();
comboBox.addItem("Opción 1");
comboBox.addItem("Opción 2");
```

- **JComboBox(E[] ítems):** Este constructor inicializa un combo box con una lista de elementos proporcionada con un array. Es útil cuando ya conoces las opciones desde el principio.

```
//Creamos una arreglo de tipo String
String[] opciones ={"Azul", "Amarillo", "Rojo"};
```

```
//Creamos el combo box con el arreglo que hemos creamos
JComboBox<String> comboBox=new JComboBox<>(opciones);
```

- **JComboBox(Vector<E> ítems):** Este constructor es similar al constructor anterior, pero en lugar de llenarlo con un arreglo , utiliza con un **Vector**. Es útil si se tiene que manjar la lista de opciones como un **Vector.**

```
//Creamos un vector de tipo String
Vector<String> opciones = new Vector<>();
//Añadimos cada uno de los elementos al vector
opciones.add("Lunes");
opciones.add("Martes");
opciones.add("Miercoles");
//Creamos el comboBox con los elementos del vector
JComboBox<String> combo = new JComboBox<>(opciones);
```

Ejercicio 10.10

Crear un programa en Java donde se muestre un JComboBox y que me muestre 4 ciudades de Ecuador.

Clase Lamina.

La clase Lamina tiene un constructor donde se crea los componentes del JPanel como un Label y un JComboBox (Hennessy & Paterson, 2019).

```
import javax.swing.*;
import java.awt.*;

public class Lamina extends JPanel {
    public Lamina() {
        //Implementamos el tipo de diseño del panel
        setLayout(new BorderLayout());
        //Creamos un objeto de tipo Jlabel con el nombre
"Ciudades del Ecuador"
        texto = new JLabel("Ciudades del Ecuador");
        //Añadimos el tipo de letra que se va a utilizar y el
tamaño
        texto.setFont(new Font("Serif", Font.PLAIN, 20));
        //Ponemos el título en el centro de la ventana
        add(texto, BorderLayout.CENTER);
        //Creamos un objeto de tipo Panel
        JPanel laminaNorte = new JPanel();
```

```
        //Creamos una instancia de un objeto JComboBox
        miCombo = new JComboBox();
        //Añadimos los elemento al comboBox
        miCombo.addItem("Quito");
        miCombo.addItem("Guayaquil");
        miCombo.addItem("Cuenca");
        miCombo.addItem("Riobamba");
        //Añadimos el combo al panel
        laminaNorte.add(miCombo);
        //Ubicamos el JComboBox en el Panel
        add(laminaNorte, BorderLayout.NORTH);

    }

    private JLabel texto;
    private JComboBox miCombo;
}
```

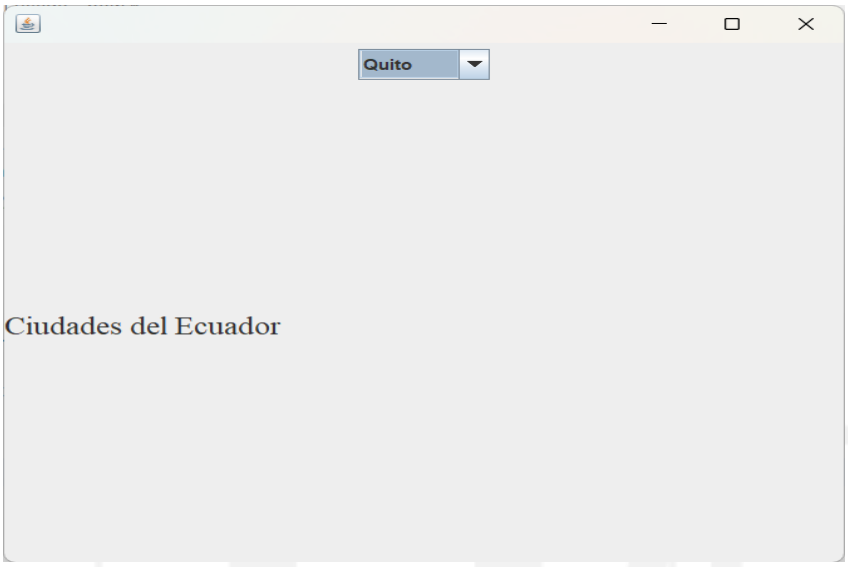
Clase Main

```
import javax.swing.*;
import java.awt.*;

public class ComboBox extends JFrame {

    public ComboBox()  {
        //Visualizamos la Ventana
        setVisible(true);
        //
        setBounds(550,300,550,400);
        //Inicializamos el nuevo objeto Lamina
        Lamina miPanel = new Lamina();
        //añadimos el objeto al Frame
        add(miPanel);

    }

    public static void main(String[] args) {
        //Instanciamos el objeto de la clase ComboBox
```

```
        ComboBox miComboBox = new ComboBox();
    }
}
```

Ejecución del programa

Figura 78.
JComboBox



Nota: La imagen nos muestra la salida de pantalla del código anterior.

10.10. Componentes de texto

Java Swing ofrece una gama de componentes de texto que permiten a los usuarios introducir y editar texto dentro de las aplicaciones gráficas. Estos componentes son esenciales para crear formularios, cuadros de diálogos y otras interfaces interactivas.

Dentro de estos componentes exploraremos **JTextComponent, JTextField, JPasswordField.**

10.10.1. JTextComponent

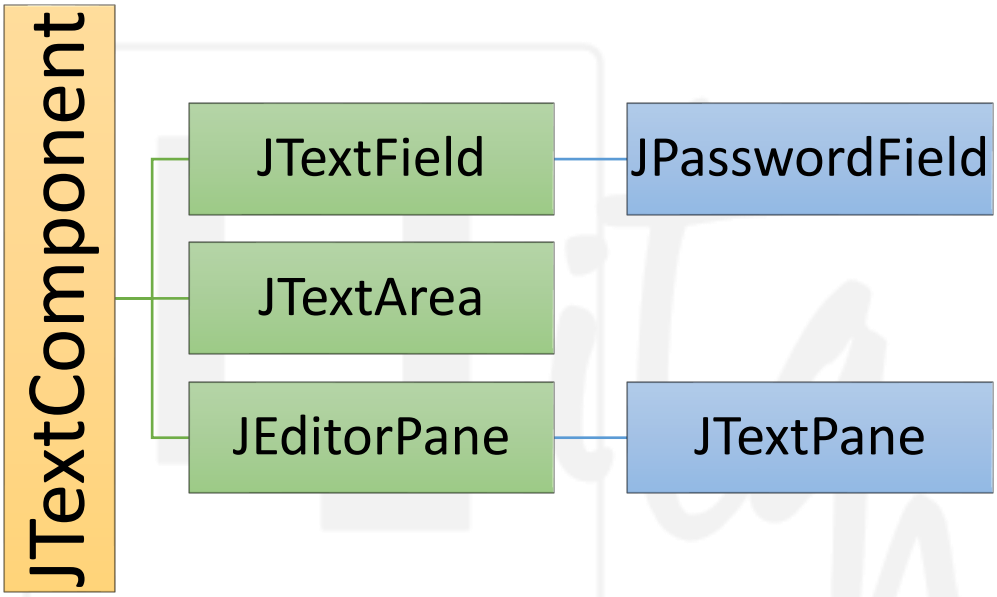
La clase base abstracta de la cual se derivan todos los componentes de texto Swing, estas clases son **JTextField, JTextArea** y **JPasswordField.** Esta clase proporciona la funcionalidad básica y común necesaria para poder manejar texto en Swing, como la selección y manipulación de texto. Aunque no se utiliza directamente para crear

instancias, **JTextComponent** define el marco sobre el cual se construye en otros componentes de texto más específicos (Goling, Joy, & Steele, 2018).

10.10.2. JTextField

El JTextField es uno de los componentes de texto más utilizados en Java Swing, este componente proporciona un cuadro de texto de una sola línea que permite a los usuarios introducir y editar datos. Es ideal para formularios donde se necesitan entradas de texto cortas como nombres, direcciones de correo electrónicos o números de teléfono (Schildt, 2014).

Figura 79.
JTextComponent



Nota: La imagen nos muestra la jerarquía de componentes de texto.

10.10.3. JPasswordField

JPasswordField es una subclase de **JTextField** diseñada específicamente para manejar la entrada de contraseñas. A diferencia de **JTextField**, **JPasswordField** ocultando el texto introducido por el usuario, generalmente mostrando asteriscos o puntos en lugar de caracteres reale (Garnica, s.f.)s.

Los constructores que se utilizan JTextField son:

- **JTextField():** Se crea un campo de texto vacío con 0 columnas.
- **JTextFiel(int cols),:** Se crea un campo de texto vacío con un número de columnas que recibe el constructor
- **JTextField(String mensaje):** Campo de texto que se ajusta a la cadena mensaje.

- **JTextField(String mensaje, int col):** Campo de texto que tiene un mensaje y tamaño de números de columnas que recibe el constructor (Deitel P. , 2014).

Los métodos que se utilizan en estas clases son:

- **setFont(Font tipo):** este método me permite seleccionar el tipo de letra que se va a utilizar en el cuadro de texto
- **setHorizontalAlignment(int alig):** Este método me permite alinear el texto que se va a insertar en el campo de texto. Los valores que me devuelve este método son **Right, left, center, trailing, leading.** Este último viene por defecto.
- **setColumns(int col):** Este método pone el número de columnas preferido para el campo.
- **setEchoChar(char c):** Coloca c para enmascarar los caracteres del campo.
- **getEchoChar():** Devuelve el carácter que enmascara el campo de texto por defecto es *.
- **Char[] getPassword[]:** Devuelve la cadena del campo en un arreglo de caracteres.

Ejercicio 10.11

Crear un programa en Java donde se cree una ventana principal, a la ventana añadir un JLabel y un JPasswordField, una vez ingresado la contraseña mostrar un mensaje donde diga contraseña valida.

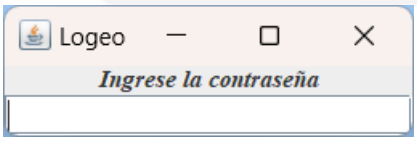
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Login extends JFrame {
    private static final int ancho = 500;
    private static final int alto = 500;
    private JPasswordField clave=null;
    private JLabel lblUsuario = null;
    private JLabel lblContrasea = null;

    public Login(String title) {
        super(title);
```

```
        setSize(ancho, alto);
        creaComponentes();
        pack();
    }

    private void creaComponentes() {
        clave = new JPasswordField(20);
        lblUsuario = new JLabel();
        //listener
        clave.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent evt) {
                procesoAccionUser(evt);
            }

        });
        //pone el campo de texto con la clave
        add(clave, BorderLayout.CENTER);
        //Crea y pone la etiqueta en la ventana
        lblUsuario.setFont(new java.awt.Font("Times new
Roman", 3, 12));

lblUsuario.setHorizontalAlignment(javax.swing.SwingConstants.
CENTER);
        lblUsuario.setText("Ingrese la contraseña");
        lblUsuario.setToolTipText("Ejemplo");
        add(lblUsuario, BorderLayout.NORTH);

    }
    //Método que se ejecuta al actuar el usuario sobre el
campo
    private void procesoAccionUser(ActionEvent evt) {
        char pass[];
        pass = clave.getPassword();
        lblContrasea = new JLabel("");
        lblContrasea.setFont(new Font("Times New Roman", 3,
12));
```

```
            if (pass.length == 0){
                System.out.println("Ingrese el Password");
                lblContrasea.setText("PASSWORD(teclear)");
            }
            else{
                clave.setEditable(false);
                lblContrasea.setText("Se valida la contraseña");
            }
            add(lblContrasea, BorderLayout.SOUTH);
            validate();
            pack();
        }

        public static void main(String[] args) {
                Login login;
                login = new Login("Logeo");
                login.setVisible(true);

        }
    }
```

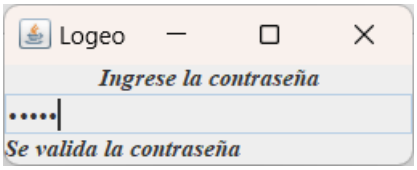
Ejecución del programa.

Figura 80.
Salida en pantalla



Nota: La imagen nos muestra una ventana un JPasswordField donde se ingresa la contraseña

Figura 81.
Validando Contraseña



Nota: La imagen nos muestra, que se ingresa la contraseña y al ingresar se valida mostrando un mensaje valida la contraseña.

10.10.4. JTextArea

El componente JTextArea se utiliza con el fin de mostrar muchas líneas de texto, dispone de métodos para fijar el ancho de cada línea y la acción a realizar si la línea que se inserta es mayor que el ancho prefijado, también permite decidir si se rompen o no las palabras en el salto de línea.

Este componente no dispone de barra de desplazamiento JScrollPane, hay que crear el scroll y asociarlo al componente; por ejemplo: se crea el componente areaTexto: **JTextArea areaTexto = new JTextArea()**, a continuación se crea el scroll y se asocia areaTexto (Deitel & Feitek, 2019):

```
JScrollPane barra= new JScrollPane(área Texto)
```

Por último, el scroll se pone en el marco: add(barra). Sus constructores son:

- **JTextArea():** Crea el componente con cadena nula y 0 filas y columnas.
- **JTextArea(int filas,int cols)**, crea con cadena nula y el número de filas y columnas especificado.
- **JTextArea(String t)** , crea el componente con cadena t y 0 filas y columnas.
- **JTextArea(String t,int filas, int col)**, crea el componente con cadena t y el número de filas y columnas especificado.

Sus métodos son:

- **public void append(String t)**, añade la cadena t al final del documento.
- **public void insert(String t,int p)**, inserta la cadena t a partir de posición p.
- **void replaceRange(String t, int inicio, int fin)**, sustituye el texto del documento en el rango inicio-fin, por la cadena t.
- **public void setColumns(int cols)**, fija el ancho de cada línea.
- **public void setLineWrap(boolean f)** si f es true activa salto automático línea.
- **public void setWrapStyleWord(boolean f)** si f es true no rompe palabras en el salto de línea.

Resumen Capitulo 10

Swing es un conjunto de clases, interfaces y recursos que facilita la creación de interfaces gráficas de usuario, comúnmente conocidas como **GUI** (del inglés "Graphical User Interfaz"). Todas las clases de Swing se encuentran dentro del paquete javax.swing. Al desarrollar un programa con una interfaz gráfica, normalmente interactuarás con cuatro tipos de elementos clave:

1. **Contenedor de Nivel Superior o Ventana Principal**: Este es el marco principal donde se alojan todos los componentes de la interfaz. Puede ser un marco (JFrame), un applet (JApplet), o un cuadro de diálogo (JDialog).
2. **Componentes de la Interfaz Gráfica**: Son los elementos visuales como botones, campos de texto, etc., que se colocan dentro de la ventana principal o en otros contenedores secundarios.
3. **Contenedores Secundarios**: Son contenedores adicionales diseñados para organizar mejor otros componentes de la interfaz, como JPanel y JScrollPane. Estos contenedores también son componentes que pueden contener otros elementos.
4. **Mecanismos para la Gestión de Eventos**: Son necesarios para manejar las acciones del usuario, como clics en botones o entradas de texto.

La ventana principal de una aplicación GUI se representa mediante un objeto de tipo JFrame. Los marcos (frames) no son visibles por defecto; para mostrar un marco en pantalla, se debe llamar al método setVisible(true).

Los componentes de una interfaz se distribuyen utilizando gestores de posicionamiento, que colocan los elementos en posiciones relativas dentro de la ventana principal. Estos gestores implementan la interfaz LayoutManager y cada contenedor tiene un gestor de posicionamiento predeterminado:

- Los marcos (JFrame) y cuadros de diálogo (JDialog) utilizan el gestor BorderLayout de manera predeterminada.
- Los paneles (JPanel) y applets (JApplet) utilizan el gestor FlowLayout por defecto.

Es posible cambiar el gestor de posicionamiento de un contenedor con el método setLayout(LayoutManager mng). Existen siete tipos principales de gestores de posicionamiento en Swing: FlowLayout, BorderLayout, GridLayout, BoxLayout, GridBagLayout, CardLayout y SpringLayout.

Los botones y etiquetas son algunos de los componentes más simples y utilizados en Swing:

- **Etiquetas (JLabel)**: Se utilizan para poner títulos o proporcionar mensajes breves junto a otros componentes.
- **Botones (JButton)**: Permiten la interacción directa del usuario, como un botón común de clic.
- **Botones de Opción (JRadioButton)**: Facilitan la creación de grupos de botones donde solo se puede seleccionar una opción.
- **Listas Desplegables (JComboBox)**: Permiten al usuario seleccionar una opción de una lista desplegable.

Todos los tipos de botones en Swing derivan de la clase abstracta AbstractButton.

Componentes de Texto en Swing

Para incluir campos de texto en una interfaz gráfica, Swing ofrece varios componentes derivados de la clase base JTextComponent:

- **JTextField**: Ideal para una única línea de texto, como cuando se necesita introducir un nombre o una dirección de correo electrónico.
- **JTextArea**: Diseñado para múltiples líneas de texto, útil para áreas de comentarios o descripciones extensas.

Estos componentes permiten la edición y visualización de texto dentro de la aplicación, ofreciendo versatilidad y facilidad de uso para construir interfaces de usuario dinámicas y atractivas.

Ejercicios Propuestos

1. Crea una aplicación en Java Swing que muestre una ventana con un botón. Cuando el usuario haga clic en el botón, debe aparecer un mensaje en la consola que diga "¡Botón clicado!".
    - Utilizar `JFrame` para crear la ventana principal.
    - Añadir un `JButton` al marco.
    - Implementar un `ActionListener` para detectar clics en el botón.
2. Construye una aplicación en Java Swing que incluya:
    - Una lista desplegable (`JComboBox`) con opciones de colores ("Rojo", "Verde", "Azul").

- Un panel (`JPanel`) que cambie su color de fondo según la selección del usuario en la lista desplegable.

El ejercicio tendrá algunos requisitos.

- Utilizar un `JComBox` para crear la lista desplegada.
- Implementar un `ActionListener` para capturar el evento de selección del usuario
- Cambiar el color del panel dinámicamente en función de la selección del usuario.

3. Escribir una aplicación gráfica en la que un usuario teclee el nombre de un archivo de texto y se muestre en un área de texto.
4. Escriba una aplicación gráfica con 6 componentes tipo botón, etiqueta y campo de texto; definir 3 paneles, cada uno con gestor de posicionamiento diferente; poner 2 componentes en cada panel y en el marco los 3 paneles.
5. Situar una etiqueta, 2 botones y un componente de texto en un marco con el gestor de posicionamiento BoxLayout.

Referencias

(s.d.).

Ahmad Dar, M. (2020). *JAVA Programming Simplified.* Walter de Gruyter GmbH.

Deitel, H., & Feitek, P. (2019). *Java: Cómo Programar (11° edición),.* Person Eduacation.

Ernest, M. (2012). *Java SE 7 Programming Essentials.* Wiley.

Garnica, C. C. (s.d.). *Principios Básicos de la POO | POO en JAva*.

Garrido Abenza, P. (2015). *Comenzando a programar con JAVA.* Universidad Miguel Hernández.

Goling, J., Joy, B., & Steele, G. (2018). *The Java Programming Languaje.*

Hennessy, J. L., & Paterson, D. A. (2019). *Arquitectura de Computadoras: Un enfoque Cuantitativo.* Pearson Education.

Joyanes Aguilar, L. (2011). *Programación en Java 6 algoritmos, programación orientada a objetos.* Mexico: McGraw-Hill Interamericana de España S.L.; 1er edición (16 Septiembre 2011).

Joyanes Aguilar, L., & Zahonero Martinez, I. (2011). *Programación en Java 6.* Mexico: McGRAW-HILL/INTERAMERICANA EDITORES, S.A. DE C.V.

Mughal, K. (2016). *A Programmer's Guide to Java SE 8 Oracle Certified Associate (OCA).* Pearson Education.

Rodríguez Rancel, M. L. (2012). *Aprender a programar en Java desde cero.* aprenderaprogramar.com.

Toro, J. (09 de 10 de 2018). *Tipos de Progrmas que se pueden hacer en Java.* Obtido de Applets: http://joseltoro.blogspot.com/2018/10/que-tipos-de-programas-se-pueden-hacer.html

Грин, А. и. (2019). *Java Swing. Создание графического интерфейса.* . Питер.

Шилдт, Г. (2019). *Java. Полное руководство (11-е издание).* Вильямс.

CADENHEAD, R. y LEMAY, L. (2007), Teach Yourself Java 6 in 21 Days, Indianápolis: Sams

Caules, C. Á. (2023). Las versiones de Java y su historia. Arquitectura Java. https://www.arquitecturajava.com/las-versiones-de-java/