

Table of Contents

Algebra Booleana	3
Pseudocódigo	6
Instrucciones de los algoritmos	9
Operadores de los algoritmos	12
Diagramas de flujo	14
Elementos de un diagrama de flujo	16
La condicional	20
Ejemplo de uso de Condicionales	25
En caso de (switch)	30
Ejemplo de en caso de (switch)	32
Estructuras Repetitivas	37
Ejemplos de Bucles	42
Los Subprocesos o Funciones	54
Solicitar Licencia Educativa	58
El Editor IntelliJ IDEA	61
Crear un nuevo Proyecto en IntelliJ IDEA	64
Comprimiendo el Proyecto	67
Introducción a la algoritmia	68
¿Qué es el lenguaje de programación Java?	70
Tipos de Datos en Java	74
Literales, Constantes y Variables	78
Alcance de las variables en Java	87
Los operadores en Java	91
Las Clases y Objetos en Java	96
La función `main`	98
Salidas y Entradas de datos en Java	99
Entendiendo mejor JOptionPane	102
Estructuras Secuenciales	107
La estructura de decisión en Java	110
El Operador Ternario en Java	117
El Ciclo `while` en Java	119
El Ciclo `do-while` en Java	122
El Ciclo `for` en Java y su versión mejorada `for-each`	125

Anidación de ciclos en Java	128
¿Qué son las clases predefinidas en Java?	131
Los Wrappers	134
La clase `Math`	137
StringBuilder	141
Validación de entradas de datos	144
Cadenas con Formato en Java y Bloque de texto	148
Menús de Interacción con JOptionPane	154
Tipos Enumerados	159
Usando `enum` con `JOptionPane`	164
El Modificador de Acceso Static	168
El modificador de acceso final	171
¿Qué es una función en Java?	175
La Recursividad en la programación	179
Los Registros en Java	183
Los niveles de acceso público y privado en Java	188
¿Qué son los arreglos?	191
¿Qué son los arreglos bidimensionales?	205
¿Existen los arreglos de más de dos dimensiones en Java?	208
Práctica 1: Nuestra Primera Clase en Java	210
Práctica 2: Sistema de Cálculo de Descuentos	212
Práctica 3: Calculadora Básica	217
Práctica 4: Creación de anagramas	221
Actividad 1: Calculadora de áreas y perímetros de figuras geométricas	224
Actividad 2: Calculadora extendida	227

Algebra Booleana

La álgebra booleana es una rama de la matemática y la lógica que trata de las operaciones lógicas y aritméticas en los valores binarios 0 y 1. Fue inventada por George Boole en el siglo XIX como un sistema de lógica para modelar el razonamiento humano. La álgebra booleana es fundamental para la informática y la electrónica digital, ya que se utiliza para diseñar circuitos digitales y programación de computadoras.

Operaciones básicas

Las operaciones básicas de la álgebra booleana son las siguientes:

Operación	Descripción	Símbolo
Negación	Invierte el valor de una variable	!
Conjunción	Devuelve verdadero si ambas variables son verdaderas	&&
Disyunción	Devuelve verdadero si al menos una de las variables es verdadera	

Leyes de la álgebra booleana

Las leyes del álgebra booleana son reglas que se utilizan para simplificar y manipular expresiones booleanas. Las principales leyes del álgebra booleana son las siguientes:

1. Ley de la identidad: $A + 0 = A$ y $A * 1 = A$
2. Ley de la dominancia: $A + 1 = 1$ y $A * 0 = 0$
3. Ley de la idempotencia: $A + A = A$ y $A * A = A$
4. Ley de la complementación: $A + !A = 1$ y $A * !A = 0$
5. Ley de la absorción: $A + A * B = A$ y $A * (A + B) = A$

6. Ley de la distribución: $A * (B + C) = A * B + A * C$ y $A + B * C = (A + B) * (A + C)$

7. Ley de De Morgan: $!(A * B) = !A + !B$ y $!(A + B) = !A * !B$

Tablas de verdad

Una tabla de verdad es una representación visual de todas las posibles combinaciones de valores de las variables en una expresión booleana y el resultado de la operación.

La tabla de verdad general es la siguiente:

A	B	A && B	A B	!A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Aplicación en algoritmos

La álgebra booleana se utiliza en algoritmos para realizar operaciones lógicas y de comparación. Por ejemplo, en un algoritmo de búsqueda se puede utilizar la operación de conjunción para verificar si dos condiciones son verdaderas

1. Inicio
2. Leer x, y
3. Si $x > 0 \&\& y > 0$
4. Escribir "Ambos números son positivos"
5. Sino
6. Escribir "Al menos uno de los números es negativo"
7. Fin

En este ejemplo, la operación $x > 0 \&\& y > 0$ verifica si tanto x como y son mayores que cero, y en caso afirmativo muestra un mensaje en pantalla. Si al menos uno de los

números es negativo, se muestra otro mensaje.

El álgebra booleana es una herramienta poderosa para la programación y la lógica, ya que permite realizar operaciones lógicas y de comparación de manera eficiente y precisa.

Pseudocódigo

El pseudocódigo es una forma de representar algoritmos de manera que sean fáciles de entender y de traducir a un lenguaje de programación. Aunque no es un lenguaje de programación en sí mismo, el pseudocódigo es una herramienta muy útil para planificar y diseñar programas antes de escribir el código en un lenguaje de programación específico.

Estructura del pseudocódigo

El pseudocódigo se compone de una serie de instrucciones que describen paso a paso lo que debe hacer un programa. Estas instrucciones se organizan en bloques de código que representan las diferentes partes del algoritmo. Cada bloque de código se inicia con una palabra clave que indica el tipo de instrucción que se va a realizar, seguida de una descripción detallada de la operación que se va a llevar a cabo.

La estructura que usaremos y que debe de seguir el pseudocódigo es la siguiente:

Nombre del algoritmo: <<Nombre del algoritmo>>

Definición de constantes:

 Tipo de Dato: <<Nombre de la constante>> = <<Valor de la constante>>

Definición de variables:

 Tipo de Dato: <<Nombre de la variable>>

1. Inicio

2. <<Instrucciones>>

.

.

n. Fin

Tipos de datos

Los tipos de datos que podemos utilizar en el pseudocódigo son los mismos que en la mayoría de los lenguajes de programación, y nos permiten almacenar diferentes tipos de información, como números, texto, fechas, etc. Algunos de los tipos de datos más comunes son:

Tipo de Dato	Descripción
Entero	Números enteros, positivos o negativos
Real	Números con decimales
Carácter	Letras, números o símbolos
Cadena	Conjunto de caracteres
Lógico	Valores verdadero o falso

Ejemplo de pseudocódigo

A continuación, se muestra un ejemplo de pseudocódigo que calcula el área de un círculo a partir de su radio:

Nombre del algoritmo: Calcular área de un círculo

Definición de constantes:

Real: PI = 3.1416

Definición de variables:

Real: radio, area

1. Inicio

2. Escribir "Ingrese el radio del círculo:"

3. Leer radio

4. Hcer area = PI * radio * radio

5. Escribir "El área del círculo es:", area

6. Fin



Nota: En este ejemplo, PI es una constante que representa el valor de pi, radio es la variable que almacena el radio del círculo y area es la variable que almacena el área calculada. Las instrucciones Escribir y Leer se utilizan para mostrar mensajes en pantalla y leer valores del usuario, respectivamente.

⚠ Nota: Para efectos de las constantes, estas siempre se escribirán en mayúsculas y se les asignará un valor que no cambiará durante la ejecución del algoritmo. Por su parte, las variables se escribirán en minúsculas y se les asignará un valor que puede cambiar durante la ejecución del algoritmo.

Instrucciones de los algoritmos

Para los algoritmos tenemos las siguientes instrucciones:

1. **Inicio/Fin:** Marca el inicio y el fin del algoritmo.
2. **Leer:** Permite leer algún dato y almacenarlo dentro de una variable para su posterior uso.
3. **Escribir:** Permite mostrar en pantalla algún dato.
4. **Hacer:** Permite asignar un valor a una variable, ya sea simple o mediante algún cálculo.

Ejemplo 1

Nombre del algoritmo: SumaDeNumeros

Definición de variables:

Entero: num1, num2, suma

Algoritmo:

1. Inicio
2. Leer num1
3. Leer num2
4. Hacer suma = num1 + num2
5. Escribir suma
6. Fin

Otro ejemplo sería el siguiente:

Nombre del algoritmo: AreaDeUnCírculo

Definición de constantes:

Real: PI = 3.1416

Definición de variables:

Real: radio, area

Algoritmo:

1. Inicio
2. Leer radio
3. Hacer area = PI * radio * radio

4. Escribir area
5. Fin

Leer y Escribir

La instrucción Leer se utiliza para obtener un valor del usuario y almacenarlo en una variable, mientras que la instrucción Escribir se utiliza para mostrar un valor en pantalla. Estas instrucciones son fundamentales para la interacción con el usuario y para la visualización de resultados.

Ambas instrucciones pueden usarse para leer o escribir más de una variable o texto, separándolos por comas. Por ejemplo:

```
Leer a, b, c  
Escribir "El resultado es:", resultado
```

En este caso, se leen tres valores y se almacenan en las variables a, b y c, y luego se muestra en pantalla el mensaje "El resultado es:" seguido del valor de la variable resultado.

⚠ Nota: Es importante tener en cuenta que las instrucciones Leer y Escribir son específicas de pseudocódigo y pueden variar en otros lenguajes de programación.

⚠ Nota: Ambas instrucciones dependen de la definición de variables antes de poder usarlas, en otras palabras, las variables deben ser declaradas antes de poder ser utilizadas en las instrucciones Leer y Escribir.

Hacer

La instrucción Hacer se utiliza para asignar un valor a una variable, ya sea un valor simple o el resultado de una operación matemática. Por ejemplo:

```
Hacer suma = num1 + num2
```

En este caso, se asigna a la variable `suma` la suma de los valores almacenados en las variables `num1` y `num2`.

La instrucción `Hacer` es fundamental para realizar cálculos y operaciones en un algoritmo, ya que permite manipular los valores almacenados en las variables y realizar operaciones matemáticas con ellos.

⚠ Nota: La instrucción `Hacer` puede utilizarse para asignar valores a variables de diferentes tipos de datos, como enteros, reales, caracteres, etc.

⚠ Nota: La instrucción `Hacer` también puede utilizarse para asignar valores arbitrarios a variables, como por ejemplo `Hacer x = 10`, donde se asigna el valor 10 a la variable `x`.

Ejemplo 2

A continuación, se muestra un ejemplo de un algoritmo que calcula el área de un triángulo a partir de su base y altura:

Nombre del algoritmo: `CalcularAreaTriangulo`

Definición de variables:

Real: `base`, `altura`, `area`

Algoritmo:

1. Inicio
2. Leer `base`
3. Leer `altura`
4. Hacer `area = (base * altura) / 2`
5. Escribir "El área del triángulo es:", `area`
6. Fin

Operadores de los algoritmos

Los operadores son símbolos que permiten realizar operaciones matemáticas, lógicas y de comparación. En los algoritmos se utilizan los siguientes operadores:

1. **Aritméticos:** Permiten realizar operaciones matemáticas.
2. **Relacionales:** Permiten comparar dos valores.
3. **Lógicos:** Permiten realizar operaciones lógicas.

Operadores aritméticos

Los operadores aritméticos son los siguientes:

Operador	Descripción	Ejemplo
<code>+</code>	Suma	<code>a + b</code>
<code>-</code>	Resta	<code>a - b</code>
<code>*</code>	Multiplicación	<code>a * b</code>
<code>/</code>	División	<code>a / b</code>
<code>%</code>	Módulo	<code>a % b</code>
<code>++</code>	Incremento	<code>a++</code>
<code>--</code>	Decremento	<code>a--</code>
<code>^</code>	Potencia	<code>a ^ b</code>

Operadores relacionales

Los operadores relacionales son los siguientes:

Operador	Descripción	Ejemplo
<code>==</code>	Igual a	<code>a == b</code>
<code>!= o <></code>	Diferente de	<code>a != b</code>
<code>></code>	Mayor que	<code>'a > b</code>
<code><</code>	Menor que	<code>a < b</code>
<code>>=</code>	Mayor o igual	<code>a >= b</code>
<code><=</code>	Menor o igual	<code>a <= b</code>
<code>!</code>	Negación	<code>!a</code>

Operadores lógicos

Los operadores lógicos son los siguientes:

Operador	Descripción	Ejemplo
<code>&&</code>	Y	<code>a && b</code>
<code> </code>	O	<code>a b</code>

Diagramas de flujo

Los diagramas de flujo son una herramienta visual que se utiliza para representar algoritmos y procesos de forma gráfica. Los diagramas de flujo son una forma eficaz de comunicar ideas y procesos complejos de una manera clara y concisa.

Elementos de un diagrama de flujo

Los diagramas de flujo están compuestos por una serie de elementos que representan diferentes partes de un algoritmo o proceso. Algunos de los elementos más comunes de un diagrama de flujo son:

1. Inicio/Fin:

- Representa el inicio o fin de un algoritmo o proceso.
- Se representa con un óvalo.

2. Proceso:

- Representa una operación o acción que se realiza en el algoritmo.
- Se representa con un rectángulo.
- Usualmente, se usa para asignar valores a variables, realizar cálculos o ejecutar instrucciones.

3. Decisión:

- Representa una condición o pregunta que se evalúa en el algoritmo.
- Se representa con un rombo.
- Usualmente, se usa para tomar decisiones basadas en una condición.

4. Conector:

- Representa la conexión entre diferentes partes de un diagrama de flujo.
- Se representa con un círculo.

5. Flecha:

- Representa la dirección del flujo de un proceso en el diagrama.
- Se representa con una flecha.

6. Entrada/Salida:

- Representa la entrada o salida de datos en el algoritmo.
- Se representa con un paralelogramo.
- Usualmente, se usa para leer o escribir datos desde o hacia una fuente externa.

Elementos de un diagrama de flujo

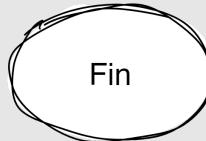
Los diagramas de flujo están compuestos por una serie de elementos que representan diferentes partes de un algoritmo o proceso. Algunos de los elementos más comunes de un diagrama de flujo son:

Inicio/Fin

Representa el inicio o fin de un algoritmo o proceso. Se representa con un óvalo.



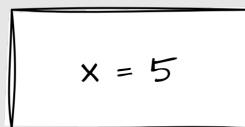
inicio.png



fin.png

Proceso

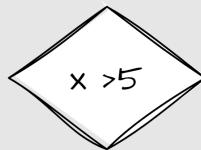
Representa una operación o acción que se realiza en el algoritmo. Se representa con un rectángulo.



hacer.png

Decisión

Representa una condición o pregunta que se evalúa en el algoritmo. Se representa con un rombo.



decision.png

Conecotor

Representa la conexión entre diferentes partes de un diagrama de flujo. Se representa con un círculo pequeño. A small, empty circular connector symbol used for linking flowchart components.

Flecha

Representa la dirección del flujo de un proceso en el diagrama. Se representa con una flecha.



flecha.png

Entrada

Representa la entrada de datos en el algoritmo. Se representa con un paralelogramo.



entrada.png

Salida

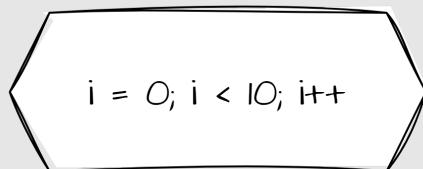
Representa la salida de datos en el algoritmo. Se representa con un documento.



salida.png

Para

Representa un bucle o ciclo en el algoritmo. Se representa con un hexágono. En dónde se deberá indicar el número de veces que se repetirá el proceso.



para.png

Subproceso

Representa un proceso o algoritmo que se ejecuta de forma independiente en el diagrama de flujo. Se representa con un rectángulo con dos líneas verticales.



subproceso.png

Estos elementos son fundamentales para la creación y comprensión de un diagrama de flujo, ya que permiten representar de forma clara y concisa la lógica y el flujo de ejecución de un algoritmo o proceso. Al utilizar símbolos gráficos y conectarlos mediante

flechas, se crea una representación visual que facilita la visualización y el análisis de un algoritmo, lo que permite diseñar, analizar y documentar procesos de una manera efectiva y eficiente.

La condicional

Dentro de algoritmia podemos utilizar la condicional para tomar decisiones en nuestro código. La condicional es una estructura de control que nos permite evaluar una expresión y ejecutar un bloque de código si la expresión es verdadera.

Para hacer esto usamos la instrucción **Si**, que se escribe de la siguiente manera:

```
n. Si <expresión> Entonces  
    Inicio  
        <bloque de código>  
    Fin  
n+1. <><Otras instrucciones>>
```

A Nota: Las instrucciones **Inicio** y **Fin** son solo para indicar el inicio y fin del bloque de código que se ejecuta si la expresión es verdadera. No son instrucciones que se escriban en el código, y sirven para delimitar las instrucciones que se ejecutan si la expresión es verdadera.

Donde **<expresión>** es una expresión lógica que se evalúa a verdadero o falso, y **<bloque de código>** es un conjunto de instrucciones que se ejecutan si la expresión es verdadera.

Por ejemplo, si queremos imprimir un mensaje si un número es mayor que 5, podemos hacerlo de la siguiente manera:

```
1. Inicio  
2. Leer n  
3. Si n > 5 Entonces  
    Escribir "El número es mayor que 5"  
4. Fin
```

En este caso, si el número ingresado es mayor que 5, se imprimirá el mensaje "El número es mayor que 5".

Condicional doble

Además de la condicional simple, también podemos utilizar la condicional doble, que nos permite ejecutar un bloque de código si la expresión es verdadera, y otro bloque de código si la expresión es falsa.

La condicional doble se escribe de la siguiente manera:

```
n. Si <expresión> Entonces
    Inicio
        <bloque de código 1>
    Fin
    Otro caso
    Inicio
        <bloque de código 2>
    Fin
n+1. <>Otras instrucciones>>
```

Donde **<expresión>** es una expresión lógica que se evalúa a verdadero o falso, **<bloque de código 1>** es un conjunto de instrucciones que se ejecutan si la expresión es verdadera, y **<bloque de código 2>** es un conjunto de instrucciones que se ejecutan si la expresión es falsa.

Por ejemplo, si queremos imprimir un mensaje si un número es mayor que 5 y otro mensaje si no lo es, podemos hacerlo de la siguiente manera:

```
1. Inicio
2. Leer n
3. Si n > 5 Entonces
    Escribir "El número es mayor que 5"
    Otro caso
    Escribir "El número no es mayor que 5"
4. Fin
```

En este caso, si el número ingresado es mayor que 5, se imprimirá el mensaje "El número es mayor que 5", y si no lo es, se imprimirá el mensaje "El número no es mayor que 5".

Condicional múltiple

Además de la condicional simple y doble, también podemos utilizar la condicional múltiple, que nos permite evaluar varias expresiones y ejecutar un bloque de código dependiendo de cuál de ellas sea verdadera.

La condicional múltiple se escribe de la siguiente manera:

```
n. Si <expresión 1> Entonces
    Inicio
        <bloque de código 1>
    Fin
    Otro caso
        Si <expresión 2> Entonces
            Inicio
                <bloque de código 2>
            Fin
        Otro caso
            Si <expresión 3> Entonces
                Inicio
                    <bloque de código 3>
                Fin
            ...
        Otro caso
            Inicio
                <bloque de código n>
            Fin
n+1. <>Otras instrucciones>>
```

Donde <expresión 1>, <expresión 2>, <expresión 3>, ..., <expresión n> son expresiones lógicas que se evalúan a verdadero o falso, <bloque de código 1>, <bloque de código 2>, <bloque de código 3>, ..., <bloque de código n> son conjuntos de instrucciones que se ejecutan si la expresión correspondiente es verdadera.

Por ejemplo, si queremos imprimir un mensaje dependiendo de si un número es mayor que 5, igual a 5 o menor que 5, podemos hacerlo de la siguiente manera:

1. Inicio
2. Leer n
3. Si n > 5 Entonces

```

        Escribir "El número es mayor que 5"
    Otro caso
        Si n = 5 Entonces
            Escribir "El número es igual a 5"
        Otro caso
            Escribir "El número es menor que 5"
4. Fin

```

En este caso, si el número ingresado es mayor que 5, se imprimirá el mensaje "El número es mayor que 5", si es igual a 5, se imprimirá el mensaje "El número es igual a 5", y si es menor que 5, se imprimirá el mensaje "El número es menor que 5".

Condicional anidada

Además de la condicional simple, doble y múltiple, también podemos utilizar la condicional anidada, que nos permite anidar condicionales dentro de otros condicionales.

La condicional anidada se escribe de la siguiente manera:

```

n. Si <expresión 1> Entonces
    Inicio
        Si <expresión 2> Entonces
            Inicio
                <bloque de código 1>
            Fin
        Otro caso
            Inicio
                <bloque de código 2>
            Fin
        Fin
    Otro caso
        Inicio
            <bloque de código 3>
        Fin
    Fin
n+1. <>Otras instrucciones>>

```

Donde <expresión 1> y <expresión 2> son expresiones lógicas que se evalúan a verdadero o falso, <bloque de código 1>, <bloque de código 2>, <bloque de código 3>, ..., <bloque de código n> son conjuntos de instrucciones que se ejecutan si la expresión correspondiente es verdadera.

Por ejemplo, si queremos imprimir un mensaje dependiendo de si un número es mayor que 5 y par, mayor que 5 e impar, o menor que 5, podemos hacerlo de la siguiente manera:

```
1. Inicio
2. Leer n
3. Si n > 5 Entonces
    Si n % 2 = 0 Entonces
        Escribir "El número es mayor que 5 y par"
    Otro caso
        Escribir "El número es mayor que 5 e impar"
    Otro caso
        Escribir "El número es menor que 5"
4. Fin
```

En este caso, si el número ingresado es mayor que 5 y par, se imprimirá el mensaje "El número es mayor que 5 y par", si es mayor que 5 e impar, se imprimirá el mensaje "El número es mayor que 5 e impar", y si es menor que 5, se imprimirá el mensaje "El número es menor que 5".

Conclusiones

En resumen, la condicional es una estructura de control que nos permite tomar decisiones en nuestro código. Podemos utilizar la condicional simple, doble, múltiple y anidada para evaluar expresiones lógicas y ejecutar bloques de código dependiendo de si la expresión es verdadera o falsa. Esto nos permite crear programas más complejos y con un mayor grado de control sobre su ejecución.

Ejemplo de uso de Condicionales

El Problema

Fábricas “El cometa” produce artículos con claves (1, 2, 3, 4, 5 y 6). Se requiere un algoritmo para calcular los precios de venta, para esto hay que considerar lo siguiente:

- Costo de producción = materia prima + mano de obra + gastos de fabricación.
- Precio de venta = costo de producción + 45 % de costo de producción.

El costo de la mano de obra se obtiene de la siguiente forma: para los productos con clave 3 o 4 se carga 75 % del costo de la materia prima; para los que tienen clave 1 y 5 se carga 80 %, y para los que tienen clave 2 o 6, 85 %.

Para calcular el gasto de fabricación se considera que si el artículo que se va a producir tiene claves 2 o 5, este gasto representa 30 % sobre el costo de la materia prima; si las claves son 3 o 6, representa 35 %; si las claves son 1 o 4, representa 28 %. La materia prima tiene el mismo costo para cualquier clave.

Definición de Variables

Para efectos del problema planteado se tiene lo siguiente:

Variable	Descripción	Tipo de Dato
C	Clave del artículo a producir.	Entero
MP	Costo de la materia prima.	Real
MO	Costo de la mano de obra.	Real
GF	Costo de los gastos de fabricación.	Real
CP	Costo de producción.	Real
PV	Precio de venta.	Real

EL Algoritmo

Nombre del Algoritmo: CostosFabrica

Declaración de variables:

Entero: C

Real: MP, MO, GF, CP, PV

Algoritmo:

1. Inicio

2. Escribir "Ingrese el código del producto a fabricar"

3. Leer C

4. Escribir "Ingrese el costo de la materia prima"

5. Leer MP

6. Si C == 4 || C == 3 Entonces

Hacer MO= MP * 0.75

Otro caso

Si C == 1 || C == 5 Entonces

Hacer MO = MP * 0.80

Otro caso

Hacer MO = MP * 0.85

7. Si C == 2 || C == 5 Entonces

Hacer GF = MP * 0.30

Otro caso

Si C == 3 || C == 6 Entonces

Hacer GF = MP * 0.35

Otro caso

Hacer GF = MP * 0.28

8. Hacer CP = MP + MO + GF

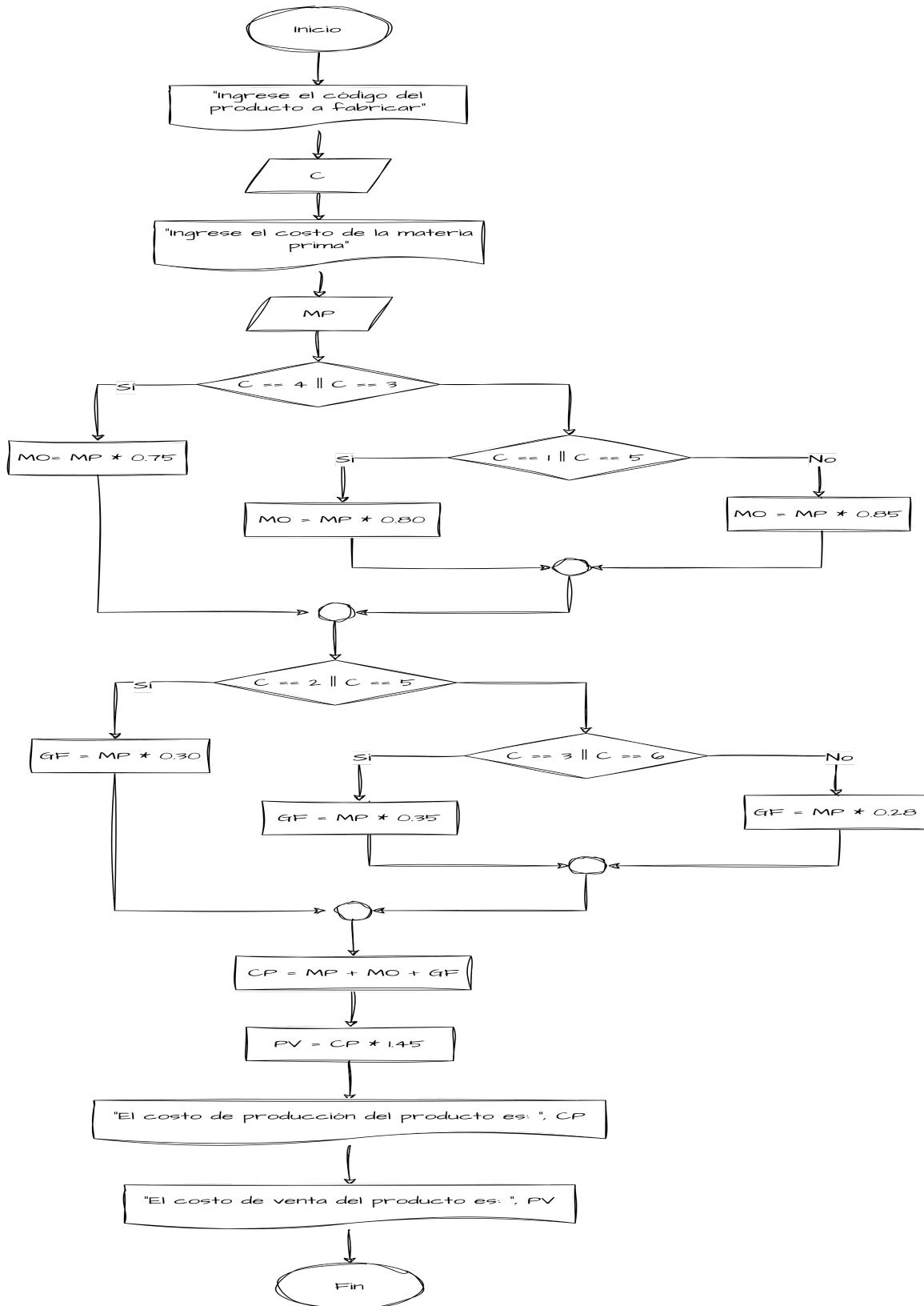
9. Hacer PV = CP * 1.45 => CP + CP * 0.45

10. Escribir "El costo de producción del producto es: ", CP

11. Escribir "El costo de venta del producto es: ", PV

12. Fin

El diagrama de flujo



condicional_1.png

Prueba del Algoritmo

Para probar el algoritmo se ingresan los siguientes datos:

- Clave del producto: 3
- Costo de la materia prima: 100

Al ejecutar el algoritmo se obtiene lo siguiente:

```
Ingrese el código del producto a fabricar
```

```
3
```

```
Ingrese el costo de la materia prima
```

```
100
```

```
El costo de producción del producto es: 210
```

```
El costo de venta del producto es: 304.5
```

Conclusiones

El uso de condicionales en un algoritmo permite tomar decisiones basadas en el valor de una expresión lógica. En este caso, se utilizó una condicional para determinar el costo de la mano de obra y los gastos de fabricación en función de la clave del producto a fabricar. Esto permitió calcular el costo de producción y el precio de venta del producto de forma automática, simplificando el proceso de cálculo y evitando errores humanos.

Referencias

- Estructuras de control en programación
(https://es.wikipedia.org/wiki/Estructura_de_control)
- Algoritmos y diagramas de flujo (https://es.wikipedia.org/wiki/Diagrama_de_flujo)
- Introducción a la programación (<https://es.wikipedia.org/wiki/Programación>)
- Lenguajes de programación
(https://es.wikipedia.org/wiki/Lenguaje_de_programación)

En caso de (switch)

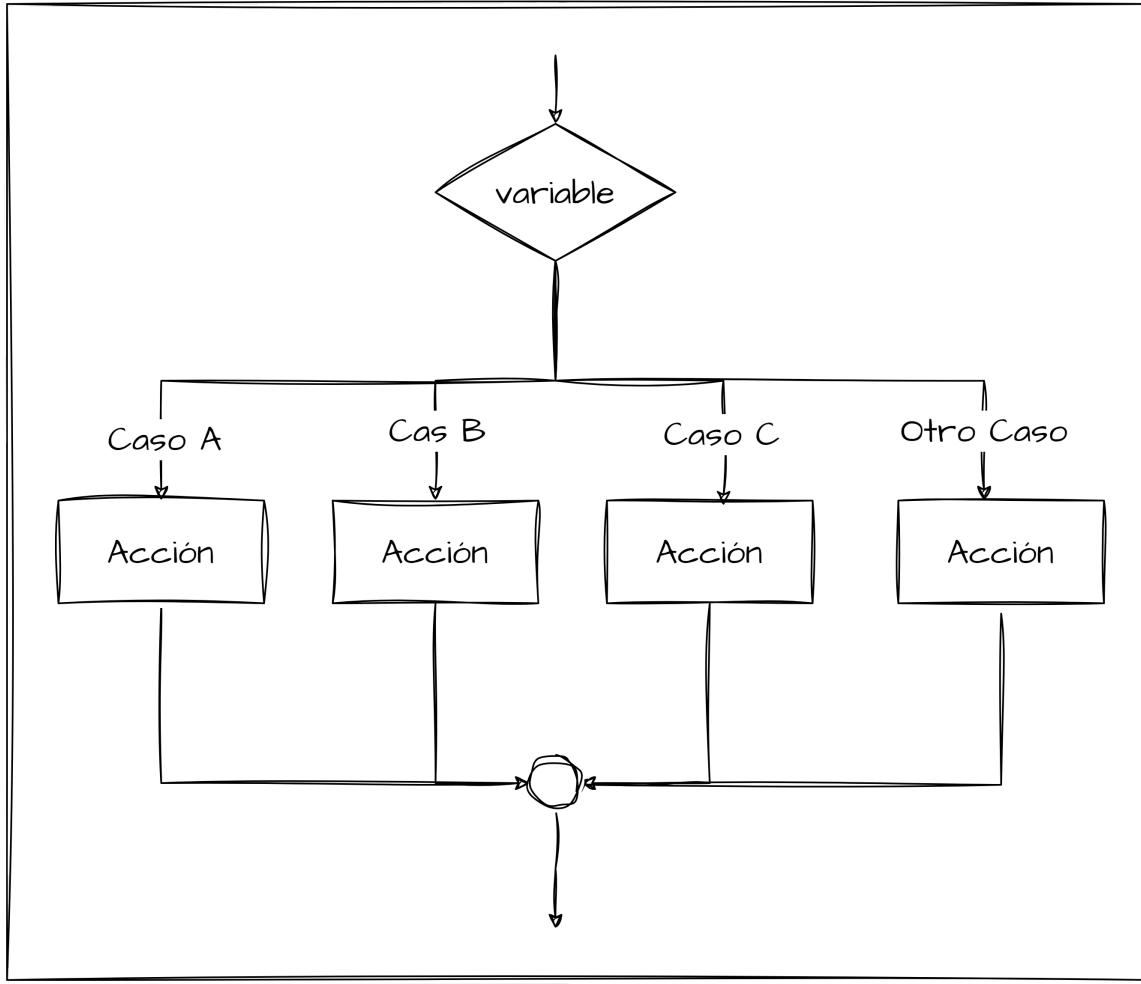
La estructura de decisión switch es una estructura de control que permite evaluar una expresión y ejecutar un bloque de código dependiendo del valor de la expresión. La estructura switch en pseudocódigo es la siguiente:

```
n.- En caso de [expresión]
    Caso [valor1]
        Inicio
        [Acción1]
        Fin
    Caso [valor2]
        Inicio
        [Acción2]
        Fin
    ...
    Caso [valorN]
        Inicio
        [AcciónN]
        Fin
    Otro Caso
        Inicio
        [Acción por defecto]
        Fin
```

Como se puede observar, la estructura switch consta de varios bloques de código, cada uno asociado a un valor específico de la expresión. Si el valor de la expresión coincide con alguno de los valores especificados, se ejecuta el bloque de código correspondiente. Si no se encuentra un valor coincidente, se ejecuta el bloque de código asociado a Otro Caso.

Es importante tener en cuenta que la estructura switch no permite evaluar expresiones complejas, como rangos de valores o expresiones booleanas. Solo se pueden evaluar valores concretos.

Diagrama de flujo



switch_flujo.png

En el diagrama de flujo, la estructura de decisión `switch` se representa con un rombo que contiene la variable o expresión a evaluar. Cada flecha que sale del rombo representa un caso posible, con una etiqueta que indica el valor asociado al caso. Si la expresión coincide con alguno de los valores especificados, se sigue la flecha correspondiente al caso. Si no se encuentra un valor coincidente, se sigue la flecha correspondiente a `Otro Caso`.

Ejemplo de en caso de (switch)

Caso 1

Una compañía de paquetería internacional tiene servicio en algunos países de América del Norte, América Central, América del Sur, Europa y Asia. El costo por el servicio de paquetería se basa en el peso del paquete y la zona a la que va dirigido. Lo anterior se muestra en la tabla siguiente:

Zona	Ubicación	Costo por Kg
1	América del Norte	24.00
2	América Central	20.00
3	América del Sur	21.00
4	Europa	10.00
5	Asia	18.00

Se requiere un algoritmo para determinar cuánto se debe cobrar por el servicio de paquetería, considerando que si el peso del paquete excede los 5 kg, se debe cobrar un costo adicional de 5.00 por cada kilo adicional.

Definición de Variables

Para efectos del problema planteado se tiene lo siguiente:

Variable	Descripción	Tipo de Dato
Z	Zona a la que va dirigido.	Entero
P	Peso del paquete.	Real
C	Costo del servicio.	Real

El Algoritmo

Nombre del Algoritmo: CostoPaqueteria

Declaración de variables:

Entero: Z

Real: P, C

Algoritmo:

1. Inicio

2. Escribir "Seleccione la zona a la que va dirigido el paquete:

- 1. América del Norte
- 2. América Central
- 3. América del Sur
- 4. Europa
- 5. Asia"

3. Leer Z

4. Escribir "Ingrese el peso del paquete"

5. Leer P

6. En caso de Z Hacer

Caso 1

Inicio

Hacer C = P * 24.00

Fin

Caso 2

Hacer C = P * 20.00

Caso 3

Inicio

Hacer C = P * 21.00

Fin

Caso 4

 Inicio

 Hacer C = P * 10.00

 Fin

Caso 5

 Inicio

 Hacer C = P * 18.00

 Fin

Otro Caso

 Inicio

 Hacer C = 0.00

 Fin

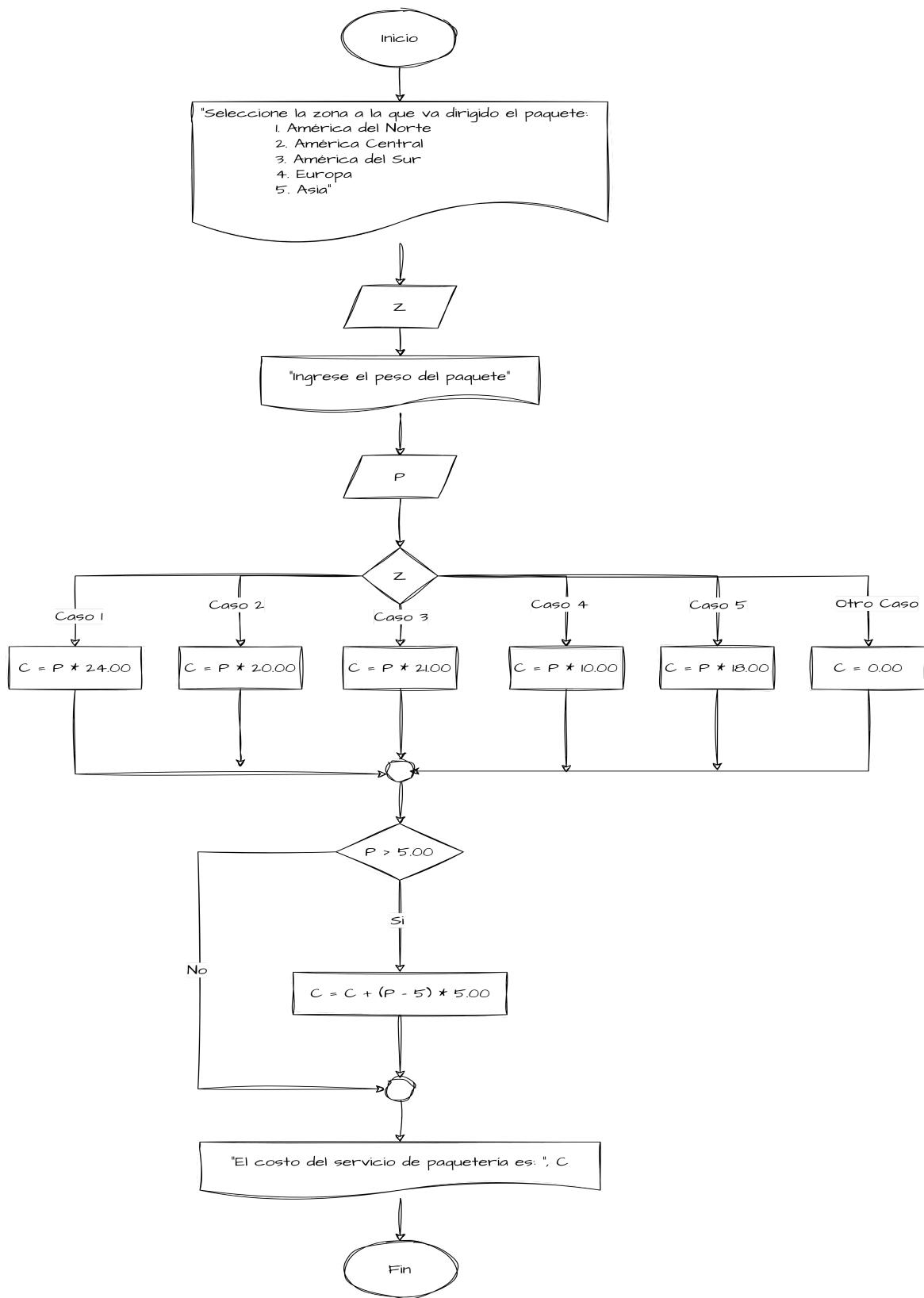
7. Si P > 5 Entonces

 Hacer C = C + (P - 5) * 5.00

8. Escribir "El costo del servicio de paquetería es: ", C

9. Fin

El diagrama de flujo



switch_1.png

Prueba del Algoritmo

Para probar el algoritmo se ingresan los siguientes datos:

- Zona: 3
- Peso del paquete: 7

Al ejecutar el algoritmo se obtiene lo siguiente:

Seleccione la zona a la que va dirigido el paquete:

1. América del Norte
2. América Central
3. América del Sur
4. Europa
5. Asia

3

Ingrese el peso del paquete

7

El costo del servicio de paquetería es: 147.00

Estructuras Repetitivas

Introducción

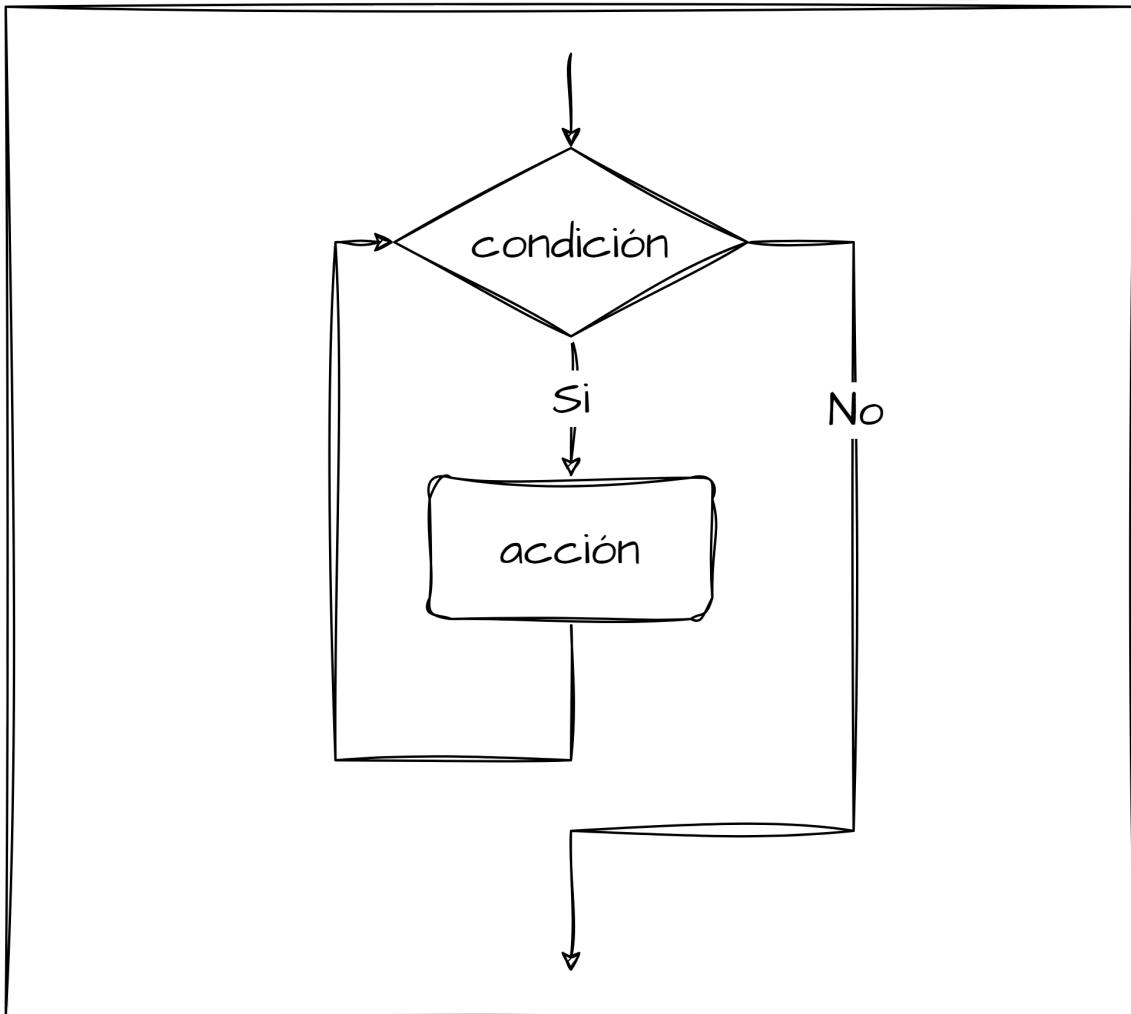
Las estructuras repetitivas son estructuras de control que permiten ejecutar un bloque de código múltiples veces. Estas estructuras son útiles cuando se necesita realizar una tarea repetitiva, como sumar una lista de números o imprimir una secuencia de caracteres.

Las estructuras repetitivas más comunes son el bucle `for`, el bucle `while` y el bucle `do while`. El bucle `for` se utiliza cuando se conoce el número de iteraciones que se deben realizar, mientras que el bucle `while` se utiliza cuando se desconoce el número de iteraciones.

Mientras que (while)

La estructura repetitiva `while` es una estructura de control que permite ejecutar un bloque de código mientras se cumpla una condición. La estructura `while` en pseudocódigo es la siguiente:

```
n.- Mientras que [condición]
    Inicio
        [Acción]
    Fin
```



while.png

Como se puede observar, la estructura `while` consta de un bloque de código que se ejecuta mientras la condición sea verdadera. Si la condición es falsa, el bloque de código no se ejecuta.

Es importante tener en cuenta que si la condición es siempre verdadera, el bucle `while` se ejecutará indefinidamente,

Hacer hasta que (do while)

La estructura repetitiva `do while` es una estructura de control que permite ejecutar un bloque de código al menos una vez y luego repetirlo mientras no se cumpla una condición. La estructura `do while` en pseudocódigo es la siguiente:

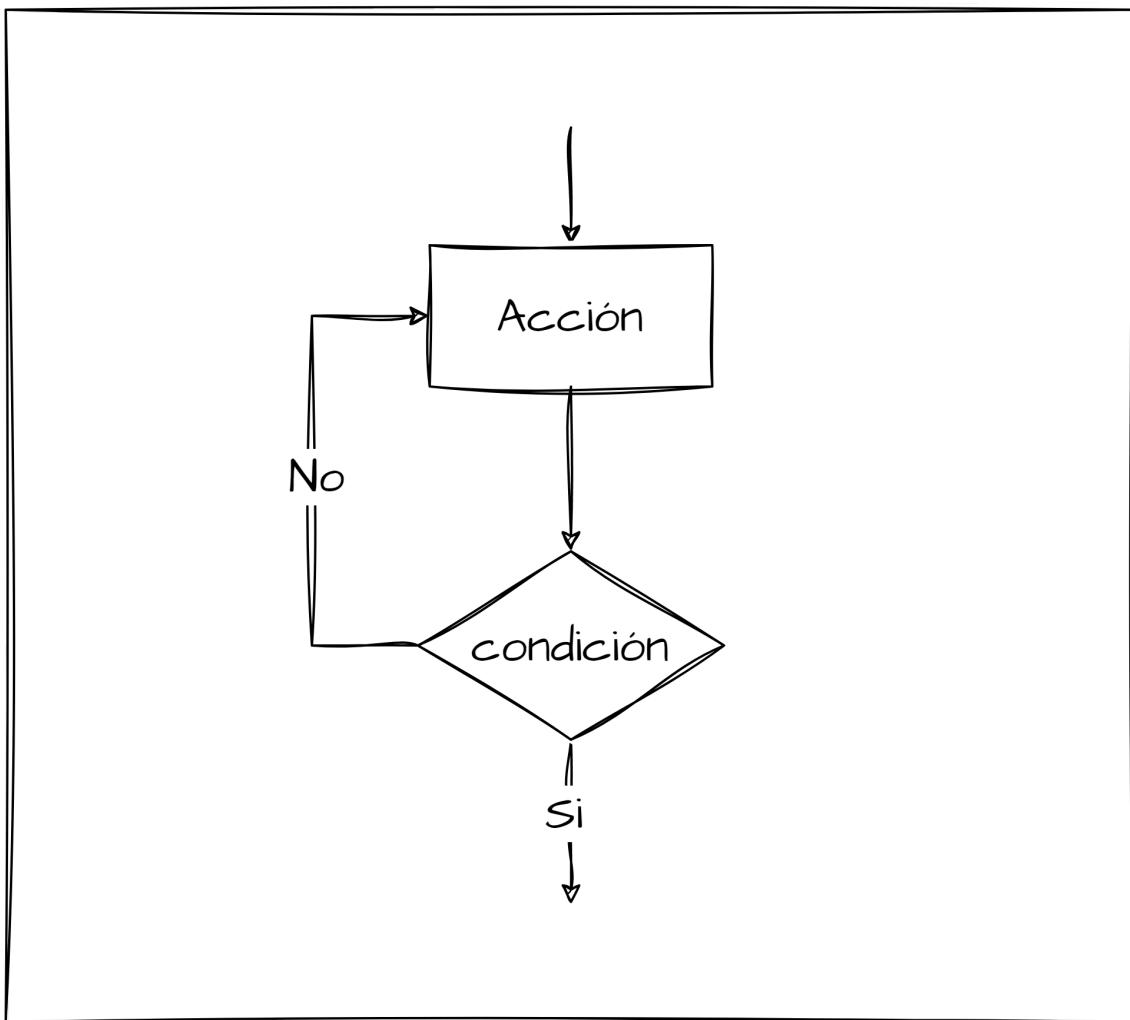
n.- Hacer

 Inicio

 [Acción]

 Fin

Hasta que [condición]



dowhile.png

Como se puede observar, la estructura `do while` consta de un bloque de código que se ejecuta al menos una vez y luego se repite mientras la condición sea falsa. Si la condición es verdadera, se continuará con el flujo del programa.

Al igual que con el bucle `while`, si la condición es siempre falsa, el bucle `do while` se ejecutará indefinidamente.

Para (for)

La estructura repetitiva `for` es una estructura de control que permite ejecutar un bloque de código un número fijo de veces. La estructura `for` en pseudocódigo es la siguiente:

```
n.- Para [variable] = [inicio] Hasta [fin] Con Paso [paso] Hacer  
    Inicio  
    [Acción]  
    Fin
```

Como se puede observar, la estructura `for` consta de una variable de control que se inicializa con un valor inicial y luego se incrementa o decrementa en cada iteración. El bucle se ejecuta mientras la variable de control esté dentro del rango especificado.

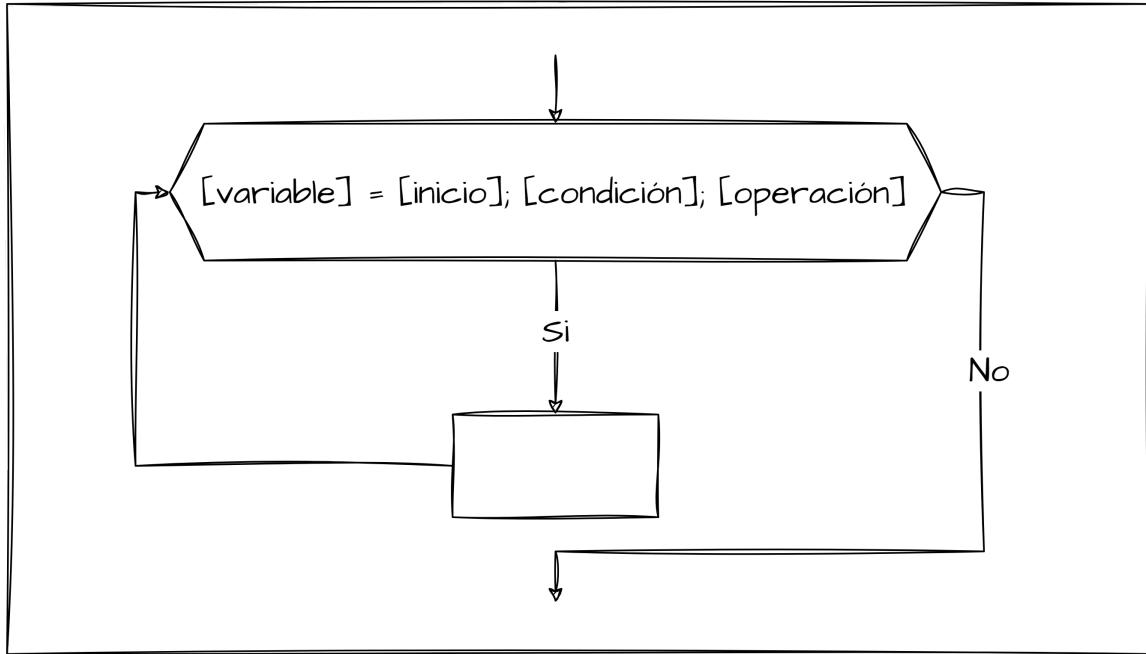
Sin embargo, la estructura `for` también puede no declarar la variable de control en la misma línea. En este caso, la variable de control se declara e inicializa fuera del bucle y se incrementa o decrementa dentro del bucle. La estructura `for` en pseudocódigo sin declarar la variable de control es la siguiente:

```
n.- [variable] = [inicio]  
m.- Para Hasta [condición] Con Paso [paso] Hacer  
    Inicio  
    [Acción]  
    Fin
```

Como se puede observar, la estructura `for` sin declarar la variable de control consta de una condición que se evalúa en cada iteración. El bucle se ejecuta mientras la condición sea verdadera.

Por último, el segmento `Con Paso [paso]` es opcional y se utiliza para especificar el incremento o decremento de la variable de control en cada iteración. Si no se especifica, la variable de control se incrementa o decrementa en una unidad por defecto. Además de esto, el segmento `Con Paso [paso]` también puede ser negativo, lo que permite decrementar la variable de control en cada iteración. Aunque también se puede emplear para realizar algún tipo de operación en la variable de control.

Diagrama de flujo



for.png

Ejemplos de Bucles

Ejercicio 1

Se requiere un algoritmo para obtener la suma de diez cantidades. Realice el diagrama de flujo y el pseudocódigo.

Con base en lo que se requiere determinar se puede establecer que las variables requeridas para la solución del problema son las mostradas en la tabla:

Variable	Tipo de Dato	Descripción
C	Entero	Contador
VA	Real	Valor por Sumar
SU	Real	Suma Acumulada

Resolviendo con el ciclo Mientras

Nombre del Algoritmo: Suma de 10 Cantidadades

Definición de variables

Entero: C

Real: VA, SU

Algoritmo:

1. Inicio
2. Hacer C=1
3. Hacer SU=0
4. Mientras C<=10

 Inicio

 Escribir "Ingrese el valor a sumar"

 Leer VA

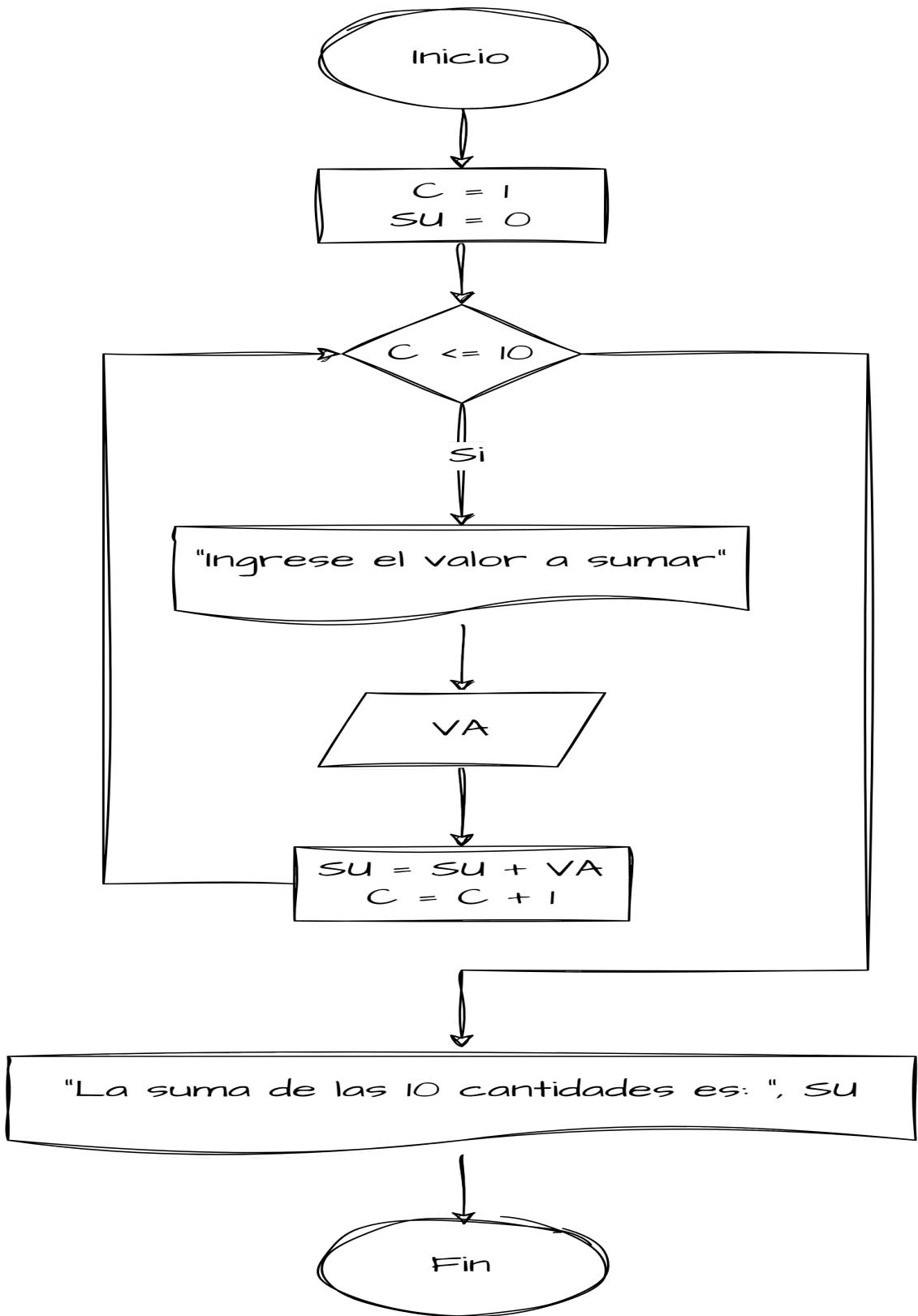
 Hacer SU=SU+VA

 Hacer C=C+1

 Fin

5. Escribir "La suma de las 10 cantidades es: ", SU

6. Fin



while_1.png

Resolviendo con el ciclo Hasta Que

Nombre del Algoritmo: Suma de 10 Cantidades

Definición de variables

Entero: C

Real: VA, SU

Algoritmo:

1. Inicio
2. Hacer $C=1$
3. Hacer $SU=0$
4. Hacer

 Inicio

 Escribir "Ingrese el valor a sumar"

 Leer VA

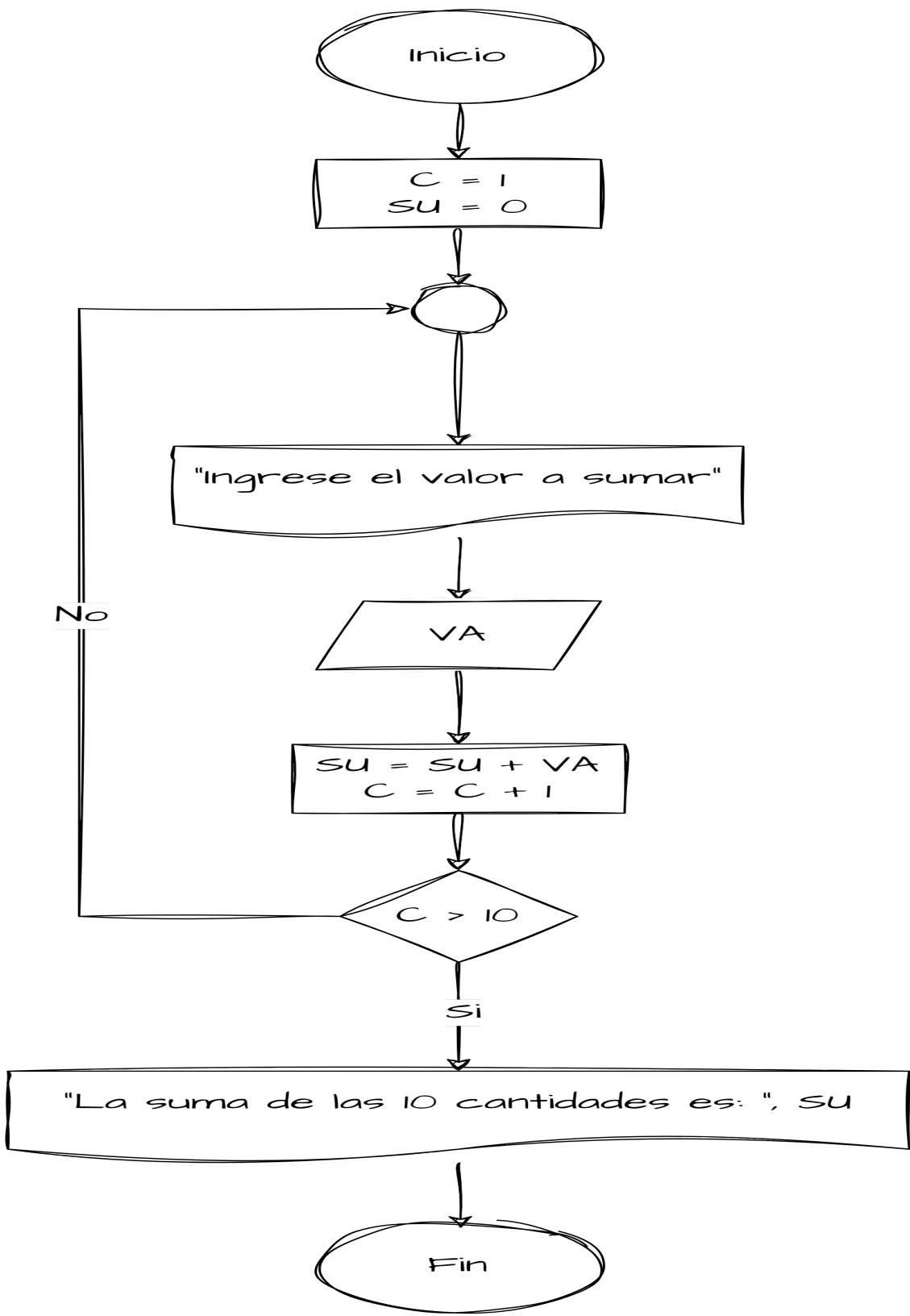
 Hacer $SU=SU+VA$

 Hacer $C=C+1$

 Fin

 Hasta Que $C>10$

5. Escribir "La suma de las 10 cantidades es: ", SU
6. Fin



do_while_1.png

Resolviendo con el ciclo Para

Nombre del Algoritmo: Suma de 10 Cantidadess

Definición de variables

Entero: C

Real: VA, SU

Algoritmo:

1. Inicio

2. Hacer SU=0

3. Para C=1 Hasta 10 // [Opcional] Con Paso 1

 Inicio

 Escribir "Ingrese el valor a sumar"

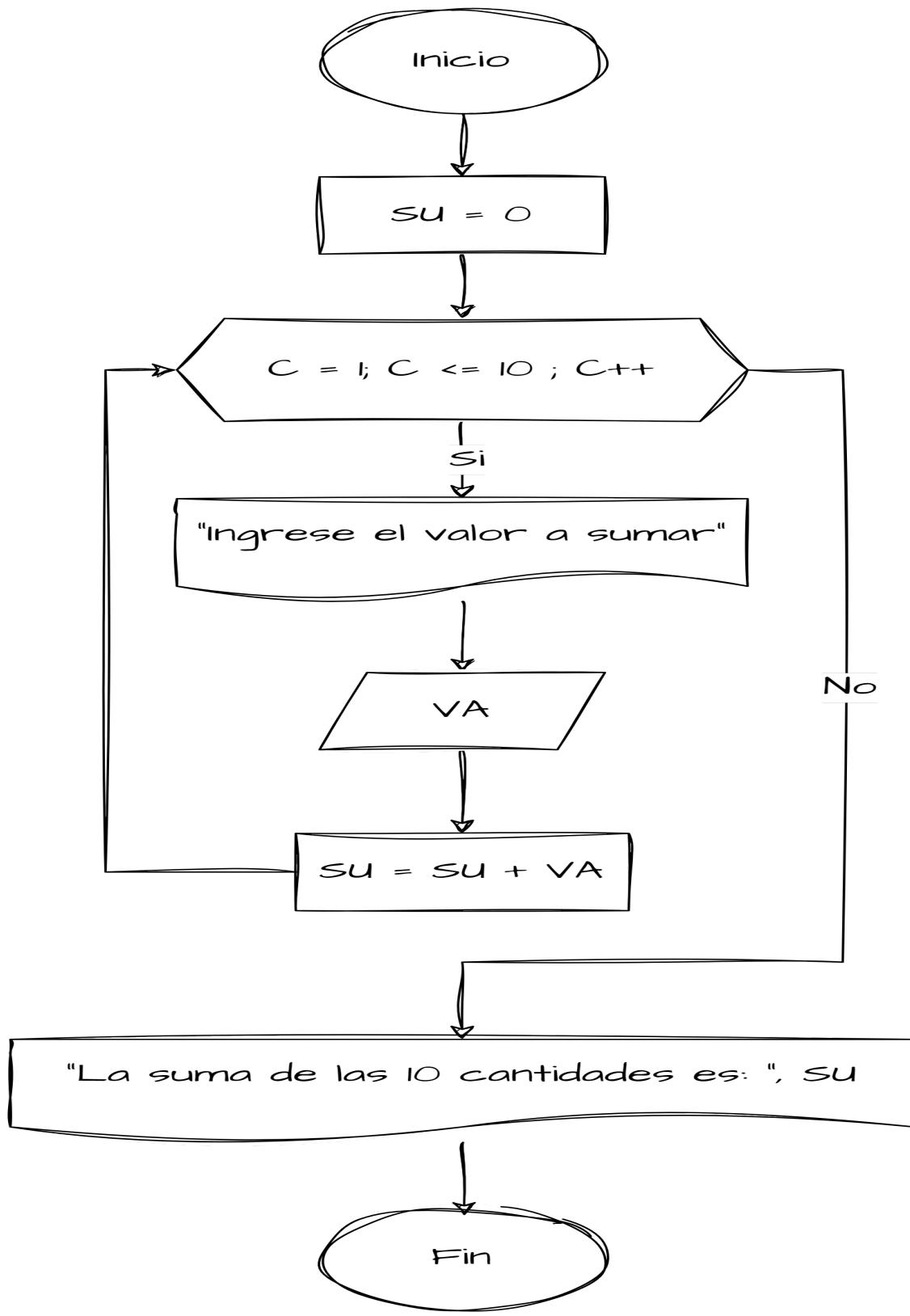
 Leer VA

 Hacer SU=SU+VA

 Fin

4. Escribir "La suma de las 10 cantidades es: ", SU

5. Fin



for_1.png

De estas maneras es como se puede resolver el problema planteado, utilizando los diferentes tipos de ciclos que se pueden encontrar en un lenguaje de programación.

Ejercicio 2

Una empresa les paga a sus empleados con base en las horas trabajadas en la semana. Realice un algoritmo para determinar el sueldo semanal de N trabajadores y, además, calcule cuánto pagó la empresa por los N empleados.

Con base en lo que se requiere determinar se puede establecer que las variables requeridas para la solución del problema son las mostradas en la tabla:

Variable	Tipo de Dato	Descripción
N	Entero	Número de Empleados
HT	Real	Horas Trabajadas
PH	Real	Pago por Hora
SS	Real	Sueldo Semanal
ST	Real	Sueldo Total
C	Entero	Contador

Resolviendo con el ciclo Mientras 2

Nombre del Algoritmo: Sueldo Semanal de N Trabajadores

Definición de variables

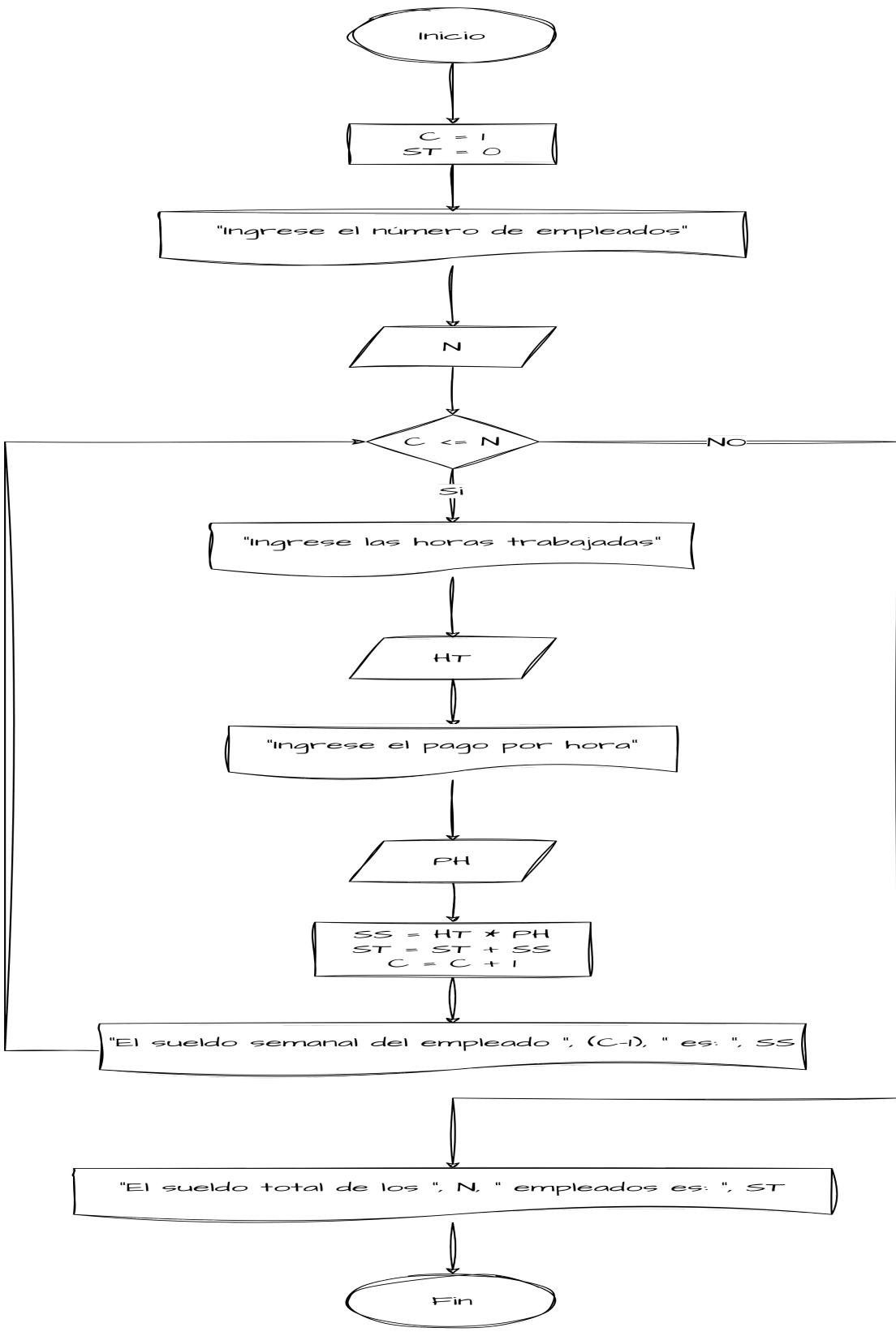
Entero: N, C

Real: HT, PH, SS, ST

Algoritmo:

1. Inicio
2. Hacer C=1
3. Hacer ST=0

```
4. Escribir "Ingrese el número de empleados"  
5. Leer N  
6. Mientras C<=N Entonces  
    Inicio  
        Escribir "Ingrese las horas trabajadas"  
        Leer HT  
        Escribir "Ingrese el pago por hora"  
        Leer PH  
        Hacer SS=HT*PH  
        Hacer ST=ST+SS  
        Escribir "El sueldo semanal del empleado ", C, " es: ", SS  
        Hacer C=C+1  
    Fin  
7. Escribir "El sueldo total de los ", N, " empleados es: ", ST  
8. Fin
```



while_2.png

Resolviendo con el ciclo Hasta Que 2

Nombre del Algoritmo: Sueldo Semanal de N Trabajadores

Definición de variables

Entero: N, C

Real: HT, PH, SS, ST

Algoritmo:

1. Inicio
2. Hacer C=1
3. Hacer ST=0
4. Escribir "Ingrese el número de empleados"
5. Leer N
6. Hacer
 Inicio
 Escribir "Ingrese las horas trabajadas"
 Leer HT
 Escribir "Ingrese el pago por hora"
 Leer PH
 Hacer SS=HT*PH
 Hacer ST=ST+SS
 Escribir "El sueldo semanal del empleado ", C, " es: ", SS
 Hacer C=C+1
 Fin
 Hasta Que C>N
7. Escribir "El sueldo total de los ", N, " empleados es: ", ST
8. Fin

Resolviendo con el ciclo Para 2

Nombre del Algoritmo: Sueldo Semanal de N Trabajadores

Definición de variables

Entero: N, C

Real: HT, PH, SS, ST

Algoritmo:

1. Inicio
2. Hacer ST=0
3. Escribir "Ingrese el número de empleados"

```
4. Leer N
5. Para C=1 Hasta N Hacer // [Opcional] Con Paso 1
    Inicio
        Escribir "Ingrese las horas trabajadas"
        Leer HT
        Escribir "Ingrese el pago por hora"
        Leer PH
        Hacer SS=HT*PH
        Hacer ST=ST+SS
        Escribir "El sueldo semanal del empleado ", C, " es: ", SS
    Fin
6. Escribir "El sueldo total de los ", N, " empleados es: ", ST
7. Fin
```

Los Subprocesos o Funciones

Los subprocesos o funciones son bloques de código que se pueden ejecutar de manera independiente al programa principal. Estos subprocesos pueden ser llamados desde cualquier parte del programa principal y pueden ser utilizados para realizar tareas específicas.

Características de los Subprocesos

- **Independencia:** Los subprocesos son independientes del programa principal, lo que significa que pueden ejecutarse de manera independiente y paralela al programa principal.
- **Reutilización:** Los subprocesos pueden ser reutilizados en diferentes partes del programa, lo que permite escribir código modular y reutilizable.
- **Paralelismo:** Los subprocesos pueden ejecutarse de manera paralela, lo que permite realizar tareas simultáneas y mejorar el rendimiento del programa.
- **Comunicación:** Los subprocesos pueden comunicarse entre sí a través de variables compartidas, lo que permite intercambiar información y coordinar tareas.
- **Concurrencia:** Los subprocesos pueden ejecutarse de manera concurrente, lo que permite realizar tareas simultáneas de manera independiente.
- **Escalabilidad:** Los subprocesos pueden escalarse para ejecutar tareas más complejas y demandantes, lo que permite mejorar el rendimiento y la eficiencia del programa.

En resumen, los subprocesos son bloques de código independientes que pueden ejecutarse de manera paralela al programa principal, lo que permite realizar tareas simultáneas y mejorar el rendimiento del programa.

Tipos de Subprocesos

Existen diferentes tipos de subprocesos, que se pueden clasificar en función de su comportamiento y su relación con el programa principal. Algunos de los tipos de subprocesos más comunes son:

- **Subprocesos Con Retorno:** Son subprocessos que devuelven un valor al programa principal, lo que permite obtener un resultado específico.
- **Subprocesos Sin Retorno:** Son subprocessos que no devuelven ningún valor al programa principal, lo que significa que solo realizan una tarea específica.

Además, los subprocessos pueden ser clasificados en función de su comportamiento y su relación con otros subprocessos. Algunos de los tipos de subprocessos más comunes son:

- **Subprocesos Síncronos:** Son subprocessos que se ejecutan de manera secuencial, lo que significa que el programa principal espera a que el subprocesso termine para continuar con la ejecución.
- **Subprocesos Asíncronos:** Son subprocessos que se ejecutan de manera paralela, lo que significa que el programa principal puede continuar con la ejecución sin esperar a que el subprocesso termine.
- **Subprocesos Anidados:** Son subprocessos que se pueden llamar desde otros subprocessos, lo que permite crear una jerarquía de subprocessos.
- **Subprocesos Independientes:** Son subprocessos que se ejecutan de manera independiente, lo que significa que pueden ejecutarse de manera paralela al programa principal.
- **Subprocesos Dependientes:** Son subprocessos que dependen de otros subprocessos, lo que significa que deben esperar a que los subprocessos anteriores terminen para poder ejecutarse.

En resumen, los subprocessos pueden ser clasificados en diferentes tipos en función de su comportamiento y su relación con el programa principal, lo que permite realizar tareas específicas de manera independiente y paralela.

Representación en Pseudo-Código de un Subproceso

La representación en pseudo-código de un subprocesso se realiza de la siguiente manera:

```
Subproceso NombreDelSubproceso(Parámetros)
    Instrucciones
```

```
Fin Subproceso
```

Donde:

- **NombreDelSubproceso**: Es el nombre del subprocesso.
- **Parámetros**: Son los parámetros que recibe el subprocesso.
- **Instrucciones**: Son las instrucciones que realiza el subprocesso.
- **Fin Subproceso**: Indica el fin del subprocesso.

Por ejemplo, el siguiente pseudo-código representa un subprocesso que suma dos números:

```
Subproceso Sumar(Entero A, Entero B)
    Entero Resultado
    Resultado = A + B
    Devolver Resultado
Fin Subproceso
```

En este caso, el subprocesso **Sumar** recibe dos parámetros de tipo entero **A** y **B**, realiza la operación de suma y devuelve el resultado.

Para hacer uso del mismo en un algoritmo principal, se puede hacer de la siguiente manera:

```
Entero Numero1 = 5
Entero Numero2 = 3
Entero Suma

Suma = Sumar(Numero1, Numero2)
Escribir "La suma de ", Numero1, " y ", Numero2, " es ", Suma
```

En este caso, se llama al subprocesso **Sumar** con los valores **Numero1** y **Numero2**, se almacena el resultado en la variable **Suma** y se muestra el resultado por pantalla.

En resumen, la representación en pseudo-código de un subprocesso permite definir bloques de código independientes que pueden ser reutilizados en diferentes partes del

programa principal para realizar tareas específicas.

Ejemplo de Subproceso

Nombre del Algoritmo: ProductoConIva

Definir Constantes:

Real: IVA = 0.16

Definición de Variables:

Real: Precio, Iva, IvaProducto, Total

Entero: CantidadProductos

Definición de Subprocesos:

1. Inicio Subproceso CalcularIva(Real Precio, Real Iva)
2. Definir Real: Total// Esto es opcional, dadó que se declara en el algoritmo principal
3. Hacer Total = (Precio * Iva)
4. Devolver Total
5. Fin Subproceso

1. Inicio Subproceso ProductoConIva()
2. Escribir "Ingrese el precio del producto:"
3. Leer Precio
4. Escribir "Ingrese el porcentaje de IVA:"
5. Leer Iva
- //6. Hacer IvaProducto = CalcularIva(Precio, IVA)
7. Escribir "El total a pagar es: ", (Precio + CalcularIva(Precio, Iva))
8. Fin Subproceso

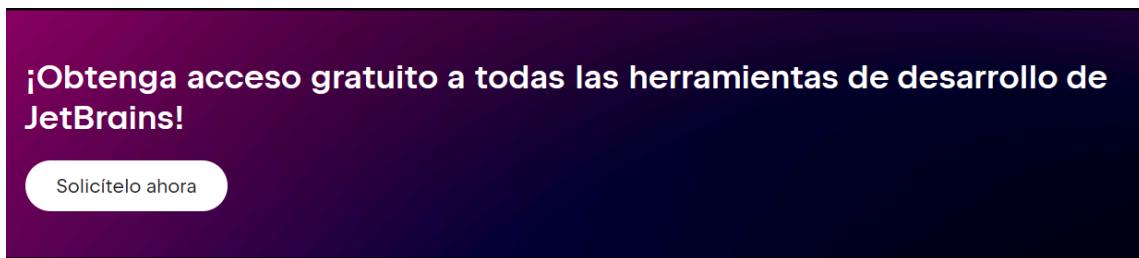
Algoritmo:

1. Inicio
2. Escribir "Ingrese la cantidad de productos a comprar:"
3. Leer CantidadProductos
4. Para i = 1 Hasta CantidadProductos Hacer
 ProductoConIva()
5. Fin

Solicitar Licencia Educativa

Para instalar CLion, primero debes solicitar una **Licencia Educativa** en el sitio web de JetBrains. Sigue estos pasos para obtener tu licencia:

1. Ve al sitio web de JetBrains: Licencias para Estudiantes (<https://www.jetbrains.com/es-es/community/education/#students/>).
2. Desplázate hacia abajo hasta encontrar la siguiente sección:



licencias.png

3. Haz clic en el botón **Solicitar Ahora**.
4. Completa el formulario con tu información personal y la información de tu institución educativa.

Solicitar con [Correo electrónico de la universidad](#) [Carnet ISIC/ITIC](#) [Documento oficial](#) [GitHub](#)

Estado: Soy estudiante Soy docente

País

Nivel de estudio

¿Su campo principal de estudios es la informática o la ingeniería?
 Sí No

Dirección de correo electrónico:

Confirmo que el correo electrónico de la universidad que indico arriba es válido y que me pertenece.

Nombre: Su nombre completo tal como aparece en su pasaporte, carnet de conducir, DNI u otro documento legal.

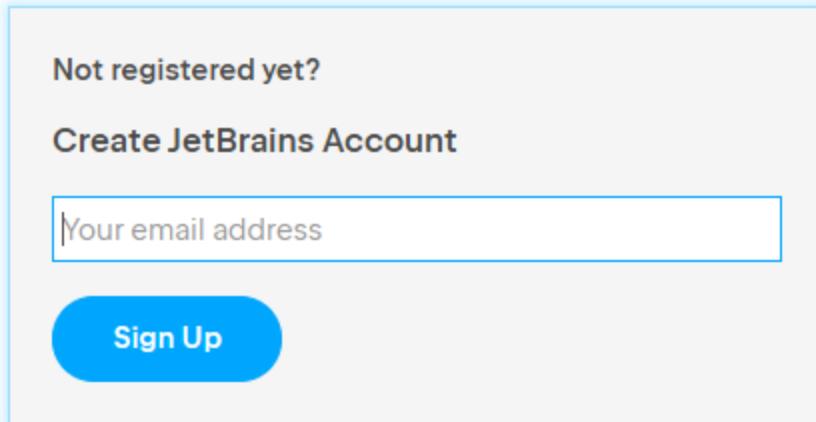
Tengo menos de 13 años
 He leído y acepto el [Acuerdo de cuenta de JetBrains](#)
 Me gustaría recibir información acerca de los productos JetBrains y JetBrains Academy, incluso invitaciones a investigaciones relacionadas ⓘ

formulario_licencia.png



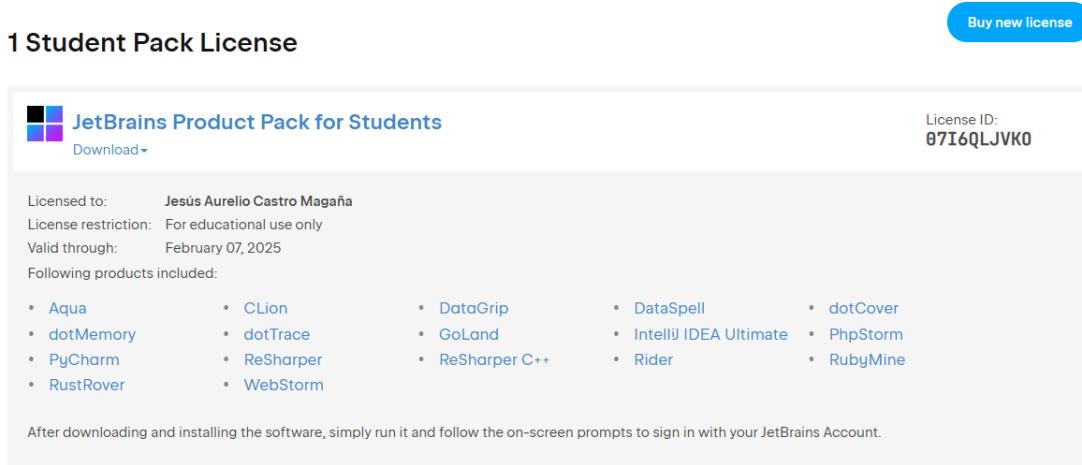
Nota: Asegúrate de utilizar tu correo electrónico institucional para solicitar la licencia. Es aquel con el dominio de tu institución educativa (por ejemplo, alumno@toluca.tecnm.mx).

5. Haz clic en el botón **Solicitar Productos Gratuitos**.
6. Verifica tu dirección de correo electrónico y sigue las instrucciones para activar tu licencia.
7. Una vez activada tu licencia, deberás crear una cuenta en JetBrains con el correo electrónico que utilizaste para solicitar la licencia. Haz clic en el enlace de activación que recibirás en tu correo electrónico y sigue las instrucciones para crear tu cuenta.



cuenta.png

8. Inicia sesión en tu cuenta de JetBrains y descarga CLion desde la sección de **JetBrains Product Pack for Students**.



descargas.png

El Editor IntelliJ IDEA

Instalación

Para instalar IntelliJ IDEA, sigue los siguientes pasos:

1. Descarga el instalador de IntelliJ IDEA desde la página oficial de JetBrains.
 - Descargar IntelliJ IDEA (<https://www.jetbrains.com/idea/download/>)
 - Selecciona la versión de IntelliJ IDEA Community Edition.
 - Selecciona tu sistema operativo.
 - Haz clic en el botón de descarga.
2. Ejecuta el instalador y sigue las instrucciones.
3. Abre IntelliJ IDEA y configura tu entorno de desarrollo.
4. ¡Listo! Ya puedes empezar a programar en Java con IntelliJ IDEA.

Configuración

Antes de comenzar a programar en IntelliJ IDEA, es recomendable configurar algunas opciones para que se adapten a tus preferencias. Sigue estos pasos:

1. Abre IntelliJ IDEA y selecciona **Configure** en la pantalla de bienvenida.
2. Haz clic en **Settings** para abrir la ventana de configuración.
3. En la sección **Plugins**, busca e instala los siguientes plugins:
 - **Diagrams.net Integration**: para crear diagramas de flujo.
 - **Indent Rainbow**: para resaltar la indentación del código.
 - **Rainbow Brackets**: para resaltar los corchetes del código.
 - **Inspection Lens**: para resaltar los problemas de código.

- **CodeGlance Pro:** para mostrar una vista previa del código.

Con estos plugins instalados, podrás programar de manera más eficiente y visual.

Atajos de teclado

IntelliJ IDEA cuenta con una serie de atajos de teclado que te permiten realizar tareas comunes de manera más rápida.

- **Ctrl + Alt + L :** Formatear el código.
- **Ctrl + Shift + F :** Buscar en todo el proyecto.
- **Ctrl + Shift + R :** Reemplazar en todo el proyecto.
- **Shift + F6 :** Renombrar una variable o método.
 - Selecciona el nombre de la variable o método que deseas cambiar y presiona **Shift + F6**.
 - Si seleccionas un archivo, podrás cambiar el nombre del archivo y todas las referencias al mismo.
- **Ctrl + D :** Duplicar una línea de código.

Estos son solo algunos de los atajos de teclado más útiles en IntelliJ IDEA. Puedes consultar la lista completa de atajos en la documentación oficial de IntelliJ IDEA.

Snippets

IntelliJ IDEA cuenta con una función llamada **Live Templates** que te permite insertar bloques de código predefinidos mediante atajos de teclado. Estos bloques de código se conocen como **snippets** y te permiten escribir código de forma rápida y eficiente.

Para utilizar un snippet en IntelliJ IDEA, sigue estos pasos:

1. Escribe el atajo de teclado correspondiente al snippet que deseas insertar.
2. Presiona la tecla **Tab** para insertar el snippet en el código.

3. Completa los campos requeridos del snippet, si es necesario.
4. Presiona la tecla `Enter` para finalizar la inserción del snippet.
5. ¡Listo! El snippet se ha insertado en tu código y puedes continuar escribiendo.

Algunos ejemplos de snippets útiles en IntelliJ IDEA son:

- **psvm**: Crea el método principal de una clase Java.
- **sout**: Imprime un mensaje en la consola.
- **fori**: Crea un bucle **for** con una variable de iteración.
- **ifn**: Crea una estructura condicional **if** con una condición nula.

Estos son solo algunos ejemplos de snippets disponibles en IntelliJ IDEA. Puedes crear tus propios snippets o personalizar los existentes según tus necesidades.

Crear un nuevo Proyecto en IntelliJ IDEA

Para crear un nuevo proyecto en IntelliJ IDEA, sigue estos pasos:

1. Abre IntelliJ IDEA y selecciona **Create New Project** en la pantalla de bienvenida.
2. Selecciona el tipo de proyecto que deseas crear. En nuestro caso, seleccionaremos **Java** y haremos clic en **Next**.
3. Configura las opciones del proyecto, como el nombre, la ubicación y el SDK de Java. Haz clic en **Finish** para crear el proyecto.
 - Si no tienes un SDK de Java configurado, puedes usar el propio IntelliJ IDEA para descargar uno automáticamente.
 - Para efecto del curso usaremos la versión 23 de Java.
4. IntelliJ IDEA creará la estructura base del proyecto y te mostrará la ventana principal del IDE.
5. ¡Listo! Ahora puedes comenzar a programar en Java con IntelliJ IDEA.

Estructura de un Proyecto en IntelliJ IDEA

Un proyecto en IntelliJ IDEA está compuesto por varios elementos que te permiten organizar y gestionar tu código de manera eficiente. Algunos de los elementos más importantes de un proyecto son:

- **src**: Directorio que contiene el código fuente de tu proyecto.
- **out**: Directorio que contiene los archivos compilados y generados por el proyecto.
- **.idea**: Directorio que contiene la configuración del proyecto para IntelliJ IDEA.
- **External Libraries**: Bibliotecas externas que se utilizan en el proyecto.

Estos elementos te permiten organizar tu código de manera estructurada y facilitan la gestión de dependencias y configuraciones del proyecto.

Crear un Nuevo Paquete

Para organizar tu código de manera más eficiente, puedes crear paquetes en IntelliJ IDEA. Sigue estos pasos para crear un nuevo paquete en tu proyecto:

1. Haz clic con el botón derecho en el directorio **src** de tu proyecto.
2. Selecciona **New > Package** en el menú contextual.
3. Ingresa el nombre del paquete y haz clic en **OK**.
4. IntelliJ IDEA creará el paquete y lo mostrará en la estructura del proyecto.
5. ¡Listo! Ahora puedes comenzar a agregar clases y archivos al paquete recién creado.



Nota: Los paquetes te permiten organizar y agrupar clases relacionadas en tu proyecto. Es una buena práctica utilizar paquetes para mantener una estructura de código clara y legible.

Crear una Nueva Clase

Para comenzar a escribir código en tu proyecto, necesitas crear una nueva clase en IntelliJ IDEA. Sigue estos pasos para crear una nueva clase en tu proyecto:

1. Haz clic con el botón derecho en el paquete donde deseas crear la clase.
2. Selecciona **New > Java Class** en el menú contextual.
3. Ingresa el nombre de la clase y haz clic en **OK**.
4. IntelliJ IDEA creará la clase y la abrirá en el editor de código.
5. ¡Listo! Ahora puedes comenzar a escribir el código de la clase en IntelliJ IDEA.

⚠ Nota: Las clases son los bloques fundamentales de un programa en Java. Cada clase contiene atributos y métodos que definen el comportamiento y la estructura de un objeto en el programa.

Ejecutar un Programa en IntelliJ IDEA

Una vez que hayas escrito el código de tu programa en IntelliJ IDEA, puedes ejecutarlo para ver los resultados. Sigue estos pasos para ejecutar un programa en IntelliJ IDEA:

1. Verifica la existencia del método `main` en la clase que deseas ejecutar.

- Si no existe, puedes crearlo manualmente o utilizar la plantilla de código `psvm` para generar el método automáticamente.
- El método se debe ver de la siguiente manera:

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

2. Haz clic en el icono de **Play** en la esquina superior derecha del editor de código. O bien, en el botón que sale en la parte lateral del código fuente.

3. IntelliJ IDEA compilará y ejecutará tu programa.

4. Observa la salida del programa en la consola de IntelliJ IDEA.

5. ¡Listo! Has ejecutado con éxito tu programa en IntelliJ IDEA.

⚠ Nota: El método `main` es el punto de entrada de un programa en Java. Debes asegurarte de que este método esté presente en la clase que deseas ejecutar.

Comprimiendo el Proyecto

Para comprimir el proyecto desde IntelliJ IDEA, se puede utilizar la opción de exportar el proyecto como un archivo ZIP. Esto es útil cuando se necesita compartir el proyecto con otros desarrolladores o para realizar una copia de seguridad del proyecto.

A continuación se muestra cómo comprimir un proyecto en IntelliJ IDEA:

1. Abre el proyecto que deseas comprimir en IntelliJ IDEA
2. Agrega el Plugin Zipper a IntelliJ IDEA si no lo tienes instalado.
3. Haz clic en menu superior Tools-> Pack the Whole Project. O bien, presiona Ctrl + Mayus + P.
4. Ingresa el nombre para tu archivo ZIP.
5. El mismo aparecerá dentro de la carpeta de tu proyecto.
6. Listo, tu proyecto ha sido comprimido.

Es importante tener en cuenta que al comprimir el proyecto, se incluirán todos los archivos y carpetas del proyecto en el archivo ZIP. Por lo tanto, asegúrate de que el proyecto esté organizado correctamente antes de comprimirlo.

De igual manera puedes agregar directorios a tu proyecto para capturas de pantalla, archivos de texto, etc. y estos también serán incluidos en el archivo ZIP.



Nota: Para crear directorios en el proyecto, puedes hacer clic derecho en el proyecto y seleccionar New-> Directory.

Recuerda que para saber en donde se guardó el archivo ZIP, puedes hacer clic derecho sobre el archivo en IntelliJ IDEA y seleccionar la opción Show in Explorer o Show in Finder para abrir la carpeta donde se encuentra el archivo.

Espero que esta guía te haya sido de ayuda para comprimir tu proyecto en IntelliJ IDEA. Si tienes alguna duda o sugerencia, no dudes en dejar un comentario.

Introducción a la algoritmia

La algoritmia es una disciplina que estudia los algoritmos, es decir, los procedimientos sistemáticos para resolver problemas. Un algoritmo es una secuencia finita de pasos que, partiendo de unos datos de entrada, produce unos datos de salida. Los algoritmos son la base de la informática y de la programación, y son fundamentales en la resolución de problemas en la vida cotidiana.

Características de los algoritmos

Los algoritmos tienen las siguientes características:

- **Preciso:** Los algoritmos deben ser precisos y no ambiguos, es decir, deben describir claramente los pasos a seguir para resolver un problema.
- **Finito:** Los algoritmos deben ser finitos, es decir, deben terminar en un número finito de pasos.
- **Definido:** Los algoritmos deben ser definidos, es decir, deben tener un número finito de pasos bien definidos.
- **Entrada:** Los algoritmos deben tener unos datos de entrada, que son los datos con los que se inicia el proceso.
- **Salida:** Los algoritmos deben tener unos datos de salida, que son los resultados obtenidos al final del proceso.

Elementos de un algoritmo

Los elementos básicos de un algoritmo son los siguientes:

- **Entrada de datos:** Los datos de entrada son los valores que se utilizan para iniciar el proceso. Por ejemplo, si queremos calcular el área de un círculo, el radio sería un dato de entrada.
- **Proceso:** El proceso es la secuencia de pasos que se deben seguir para resolver el problema. Por ejemplo, para calcular el área de un círculo, se debe multiplicar el radio por sí mismo y por el valor de pi.

- **Salida de datos:** Los datos de salida son los resultados obtenidos al final del proceso. Por ejemplo, el área calculada del círculo sería un dato de salida.

Ejemplo de algoritmo

A continuación, se muestra un ejemplo de un algoritmo que calcula el área de un círculo a partir de su radio:

Nombre del algoritmo: Calcular área de un círculo

Definición de constantes:

Real: PI = 3.1416

Definición de variables:

Real: radio, area

Algoritmo:

1. Inicio
2. Escribir "Ingrese el radio del círculo:"
3. Leer radio
4. Hacer area = PI * radio * radio
5. Escribir "El área del círculo es:", area
6. Fin

En este ejemplo, PI es una constante que representa el valor de pi, radio es la variable que almacena el radio del círculo y area es la variable que almacena el área calculada. Las instrucciones Escribir y Leer se utilizan para mostrar mensajes en pantalla y leer valores del usuario, respectivamente.

¿Qué es el lenguaje de programación Java?

Java es un lenguaje de programación de propósito general, orientado a objetos y diseñado para ser portable, es decir, que pueda ejecutarse en cualquier plataforma sin necesidad de recompilar el código fuente. Fue creado por Sun Microsystems en 1995 y actualmente es propiedad de Oracle Corporation.

Java se caracteriza por su sintaxis sencilla y fácil de aprender, así como por su robustez y seguridad. Estas características lo han convertido en uno de los lenguajes de programación más populares y ampliamente utilizados en la industria del software.

Características principales de Java

Algunas de las características más importantes de Java son las siguientes:

- **Orientación a objetos:** Java es un lenguaje de programación orientado a objetos, lo que significa que todo en Java es un objeto. Esto facilita la reutilización de código y la creación de programas modulares y escalables.
- **Portabilidad:** Java es un lenguaje portable, lo que significa que el código fuente escrito en Java puede ejecutarse en cualquier plataforma que tenga una máquina virtual Java (JVM) instalada.
- **Robustez:** Java está diseñado para ser robusto y resistente a errores. Su sistema de gestión de memoria automático evita los problemas de corrupción de memoria y fugas de memoria.
- **Seguridad:** Java cuenta con un modelo de seguridad sólido que protege al usuario de posibles amenazas como virus y malware.
- **Multi-hilo:** Java es un lenguaje multi-hilo, lo que significa que puede ejecutar múltiples hilos de forma concurrente. Esto facilita la creación de programas que realizan múltiples tareas simultáneamente.

¿Por qué aprender Java?

Java es uno de los lenguajes de programación más populares y demandados en la industria del software. Aprender Java te permitirá acceder a un amplio abanico de oportunidades laborales y desarrollar aplicaciones de todo tipo, desde aplicaciones web y móviles hasta sistemas embebidos y aplicaciones empresariales.

Además, Java es un lenguaje versátil y escalable que se adapta a las necesidades de cualquier proyecto, desde pequeñas aplicaciones personales hasta grandes sistemas distribuidos. Su amplia comunidad de desarrolladores y su extensa biblioteca de clases y frameworks hacen de Java una excelente elección para cualquier programador.

En resumen, aprender Java te abrirá las puertas a un mundo de posibilidades en el campo de la programación y te permitirá desarrollar habilidades y competencias que serán valiosas a lo largo de tu carrera profesional.

¿Cómo funciona Java?

Java es un lenguaje de programación compilado e interpretado. Esto significa que el código fuente escrito en Java se compila a un código intermedio llamado *bytecode*, que es ejecutado por la máquina virtual Java (JVM). La JVM es un componente fundamental de la plataforma Java y se encarga de interpretar el bytecode y ejecutar el programa en tiempo de ejecución.

El proceso de ejecución de un programa Java consta de los siguientes pasos:

1. El código fuente escrito en Java se compila a bytecode mediante el compilador de Java (`javac`). El bytecode resultante se almacena en archivos con extensión `.class`.
 - El compilador de Java verifica la sintaxis y la semántica del código fuente y genera un archivo `.class` por cada clase definida en el programa.
 - El bytecode es independiente de la plataforma y puede ejecutarse en cualquier máquina virtual Java (JVM).
2. El bytecode es interpretado por la JVM, que se encarga de ejecutar el programa en tiempo de ejecución. La JVM es responsable de cargar las clases, gestionar la memoria y ejecutar las instrucciones del programa.
3. El programa Java se ejecuta en la JVM y produce la salida esperada, ya sea en forma de texto, gráficos, sonido, etc.

4. El proceso de ejecución termina y la JVM libera los recursos utilizados por el programa.
5. El ciclo se repite cada vez que se ejecuta el programa, lo que permite que el código fuente escrito en Java sea portable y pueda ejecutarse en cualquier plataforma que tenga una JVM instalada.

En resumen, Java es un lenguaje de programación versátil y portable que se ejecuta en una máquina virtual Java (JVM) y permite desarrollar aplicaciones robustas y seguras para una amplia variedad de plataformas y dispositivos.

¿Cómo se compila y se ejecuta un programa Java?

Para compilar y ejecutar un programa Java, sigue estos pasos:

1. Crea un archivo con extensión `.java` que contenga el código fuente de tu programa.
2. Abre una terminal o línea de comandos y navega hasta la ubicación del archivo `.java`.
3. Compila el programa Java con el comando `javac` seguido del nombre del archivo `.java`.

Por ejemplo:

```
javac MiPrograma.java
```

- El compilador de Java generará un archivo `.class` por cada clase definida en el programa.
- Si hay errores de compilación, el compilador mostrará mensajes de error que deberás corregir antes de continuar.
- Si la compilación es exitosa, no se mostrarán mensajes de error y se generarán los archivos `.class`.

4. Ejecuta el programa Java con el comando `java` seguido del nombre de la clase principal (la que contiene el método `main`). Por ejemplo:

```
java MiPrograma
```

- La JVM cargará la clase principal y ejecutará el método `main` del programa.

- Si el programa requiere argumentos de línea de comandos, puedes pasarlo despu s del nombre de la clase principal.
 - La salida del programa se mostrar  en la terminal o l nea de comandos.
5. ¡Listo! Has compilado y ejecutado tu primer programa Java. Ahora puedes modificar el c digo fuente y repetir los pasos anteriores para probar tus cambios.



Nota: Si est s utilizando un IDE como IntelliJ IDEA, Eclipse o NetBeans, puedes compilar y ejecutar programas Java directamente desde el entorno de desarrollo. En estos casos, el IDE se encargará de compilar y ejecutar el programa automáticamente, sin necesidad de utilizar la l nea de comandos.

Tipos de Datos en Java

Los tipos de datos en Java son una parte fundamental del lenguaje de programación, ya que permiten almacenar y manipular información de diferentes tipos. Java es un lenguaje de programación fuertemente tipado, lo que significa que cada variable debe tener un tipo de dato específico y no puede cambiar de tipo una vez que ha sido declarada.

En Java, los tipos de datos se dividen en dos categorías principales: tipos primitivos y tipos de referencia. Los tipos primitivos son los tipos de datos básicos que están integrados en el lenguaje, mientras que los tipos de referencia son objetos que se definen mediante clases y se almacenan en el *heap*. A continuación, veremos los tipos de datos más comunes en Java y cómo se utilizan en la programación.

Tipos de Datos Primitivos

Los tipos de datos primitivos en Java son los siguientes:

Tipo de Dato	Descripción	Tamaño	Rango
byte	Entero de 8 bits con signo	8 bits	-128 a 127
short	Entero de 16 bits con signo	16 bits	-32,768 a 32,767
int	Entero de 32 bits con signo	32 bits	-2,147,483,648 a 2,147,483,647
long	Entero de 64 bits con signo	64 bits	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
float	Número de punto flotante de 32 bits	32 bits	1.4e-45 a 3.4e+38
double	Número de punto flotante de 64 bits	64 bits	4.9e-324 a 1.8e+308
boolean	Valor booleano (true o false)	1 bit	true o false
char	Carácter Unicode de 16 bits	16 bits	'\u0000' a '\uffff'

Los tipos de datos primitivos en Java se utilizan para almacenar valores simples como números enteros, números de punto flotante, caracteres y valores booleanos. Cada tipo de dato primitivo tiene un tamaño específico en memoria y un rango de valores que puede almacenar.

Tipos de Datos de Referencia

Los tipos de datos de referencia en Java son objetos que se definen mediante clases y se almacenan en el *heap*. Algunos de los tipos de datos de referencia más comunes en Java son:

- **String**: Cadena de caracteres.
- **Array**: Arreglo de elementos.
- **Clase**: Objeto de una clase definida por el programador.
- **Interfaz**: Interfaz que define un conjunto de métodos.
- **Enum**: Enumeración de constantes.

Los tipos de datos de referencia en Java se utilizan para almacenar información más compleja que los tipos primitivos, como cadenas de caracteres, arreglos de elementos y objetos definidos por el programador. Estos tipos de datos se almacenan en el *heap* y se acceden mediante referencias.

Conversión de Tipos de Datos Casting

En Java, la conversión de tipos de datos se realiza mediante un proceso llamado *casting*. El *casting* permite convertir un valor de un tipo de dato a otro tipo compatible. Existen dos tipos de *casting* en Java: *casting implícito* y *casting explícito*.

- *Casting implícito*: Se realiza de forma automática cuando se asigna un valor de un tipo de dato a otro tipo compatible. Por ejemplo, asignar un entero a un número de punto flotante.
- *Casting explícito*: Se realiza de forma manual mediante la especificación del tipo de dato entre paréntesis. Por ejemplo, convertir un número de punto flotante a un entero.

```
int entero = 10;
double flotante = entero; // Casting implícito de int a double

double flotante = 10.5;
int entero = (int) flotante; // Casting
```

Es importante tener en cuenta que el *casting* puede provocar la pérdida de precisión o la truncación de valores, por lo que es necesario realizarlo con precaución para evitar errores en el programa.

Conclusiones

Los tipos de datos en Java son una parte esencial del lenguaje de programación y permiten almacenar y manipular información de diferentes tipos de forma eficiente. Tanto los tipos de datos primitivos como los tipos de datos de referencia son fundamentales para el desarrollo de aplicaciones en Java y deben utilizarse de forma adecuada para garantizar el correcto funcionamiento del programa.

Literales, Constantes y Variables

Literales

Los literales son valores constantes que se utilizan en un programa. Por ejemplo, los números 1, 2, 3, etc., son literales numéricos, mientras que las cadenas de texto Hola, mundo son literales de cadena.

En Java, los literales se pueden clasificar en los siguientes tipos:

- **Literales numéricos:** representan valores numéricos enteros o de punto flotante.
- **Literales de cadena:** representan valores de texto.
- **Literales booleanos:** representan valores booleanos true o false.
- **Literales de caracteres:** representan valores de un solo carácter.
- **Literales nulos:** representan la ausencia de un valor.
- **Literales de arreglos:** representan arreglos de valores.
- **Literales de punto flotante:** representan valores de punto flotante.
- **Literales de punto flotante en notación científica:** representan valores de punto flotante en notación científica.
- **Literales numéricos con formato hexadecimal:** representan valores numéricos en formato hexadecimal.
- **Literales numéricos con formato binario:** representan valores numéricos en formato binario.

A continuación, se muestran algunos ejemplos de literales en Java:

- **Literales numéricos:**
 - 123: literal entero.
 - 3.14: literal de punto flotante.

- 0b1010: literal binario.
- 0xFF: literal hexadecimal.
- 1.23e-4: literal de punto flotante en notación científica.
- 123L: literal entero largo.
- 3.14f: literal de punto flotante simple.

- **Literales de cadena:**

- "Hola, mundo": literal de cadena.
- "Java es un lenguaje de programación": literal de cadena.
- "123": literal de cadena que representa un número.

- **Literales booleanos:**

- true: literal booleano verdadero.
- false: literal booleano falso.

- **Literales de caracteres:**

- 'a': literal de carácter.
- 'b': literal de carácter.
- '1': literal de carácter numérico.

- **Literales nulos:**

- null: literal nulo.

- **Literales de arreglos:**

- {1, 2, 3}: literal de arreglo de enteros.
- {"Hola", "Mundo"}: literal de arreglo de cadenas.

Constantes

Las **constantes** son variables cuyo valor no cambia durante la ejecución de un programa. En Java, se pueden declarar constantes utilizando la palabra clave `final`. Por convención, los nombres de las constantes se escriben en mayúsculas y separados por guiones bajos (`_`).

Por ejemplo, para declarar una constante que represente el valor de PI, se puede hacer lo siguiente:

```
public class Constantes {  
    public static final double PI = 3.14159;  
}
```

En este caso, la constante `PI` se declara como `public static final double PI = 3.14159;`. Esto significa que `PI` es una constante pública (`public`), estática (`static`), final (`final`) y de tipo `double`. Una vez que se asigna un valor a la constante `PI`, este no puede cambiar durante la ejecución del programa.

⚠ Nota: Las constantes en Java se pueden declarar en cualquier parte del código, pero es común declararlas como variables de clase (`static`) para que estén disponibles en todo el programa. También es común declararlas como `public` para que puedan ser accedidas desde otras clases.

⚠ Nota: La palabra reservada `final` se utiliza para declarar constantes en Java. Una vez que se asigna un valor a una constante o variable `final`, no se puede cambiar su valor. Las constantes se suelen declarar con letras mayúsculas y guiones bajos para separar las palabras, por ejemplo: `PI`, `MAX_VALUE`, `DEFAULT_COLOR`, etc.

Las constantes son útiles para representar valores fijos que se utilizan en múltiples partes de un programa y que no deben cambiar. Al declarar una constante, se garantiza que su valor permanecerá constante y que no se podrá modificar accidentalmente.

Las constantes se utilizan comúnmente para representar valores como:

- Valores matemáticos (PI, Euler, etc.).
- Valores de configuración (tamaños de ventana, colores predeterminados, etc.).

- Valores de códigos de error o mensajes de error.
- Valores de configuración de aplicaciones (URLs, rutas de archivos, etc.).
- Valores de configuración de bases de datos (nombres de tablas, nombres de columnas, etc.).
- Valores de configuración de API (claves de acceso, URLs de endpoints, etc.).
- Valores de configuración de seguridad (claves de cifrado, tokens de autenticación, etc.).
- Valores de configuración de servicios externos (URLs de servicios, claves de acceso, etc.).

Al utilizar constantes en un programa, se mejora la legibilidad del código y se facilita la modificación de los valores fijos en un solo lugar, en caso de que sea necesario realizar cambios en el futuro.

Variables

Las **variables** son espacios de memoria que se utilizan para almacenar valores en un programa. En Java, las variables se declaran especificando su tipo y nombre, y opcionalmente se les asigna un valor inicial.

Por ejemplo, para declarar una variable entera llamada `edad` con un valor inicial de `25`, se puede hacer lo siguiente:

```
int edad = 25;
```

En este caso, la variable `edad` se declara como `int edad = 25;`. Esto significa que `edad` es una variable de tipo entero (`int`) y se inicializa con el valor `25`. Una vez que se asigna un valor a la variable `edad`, este puede cambiar durante la ejecución del programa.

⚠ Nota: Las variables en Java se pueden declarar en cualquier parte del código y su alcance depende del contexto en el que se declaren. Es importante tener en cuenta que las variables locales solo son visibles dentro del bloque de código en el que se declaran, mientras que las variables de instancia y de clase son

visibles en todo el programa. Además, las variables locales deben inicializarse antes de ser utilizadas, de lo contrario, se producirá un error de compilación.

⚠ Nota: Las variables en Java se pueden declarar con diferentes tipos de datos, como enteros (int), flotantes (float), dobles (double), caracteres (char), booleanos (boolean), cadenas (String), etc. Cada tipo de dato tiene un tamaño y un rango de valores específico, por lo que es importante elegir el tipo de dato adecuado para cada variable.

Las variables se utilizan para almacenar valores temporales en un programa y se pueden utilizar para realizar cálculos, almacenar resultados intermedios, interactuar con el usuario, etc. Al utilizar variables en un programa, se pueden realizar operaciones dinámicas y manipular datos de forma eficiente.

Variabes de Instancia o Atributos de Clase

Los **atributos de clase** son variables que se declaran dentro de una clase y representan las propiedades o características de dicha clase. Los atributos de clase se utilizan para almacenar información sobre los objetos instanciados a partir de esa clase.

Por ejemplo, en una clase `Persona`, se pueden declarar atributos de clase como `nombre`, `edad` y `genero` para representar las propiedades de una persona:

```
public class Persona {  
    String nombre;  
    int edad;  
    char genero;  
}
```

En este caso, los atributos de clase `nombre`, `edad` y `genero` se declaran dentro de la clase `Persona`. Estos atributos se utilizan para almacenar información sobre una persona y son visibles en todos los métodos de la clase.

⚠ Nota: Los atributos de clase se declaran dentro de una clase y representan las propiedades o características de dicha clase. Estos atributos se utilizan para

almacenar información sobre los objetos instanciados a partir de esa clase y son visibles en todos los métodos de la clase.

Variables Locales

Las **variables locales** son variables que se declaran dentro de un método o bloque de código y que solo son visibles dentro de ese ámbito. Las variables locales se utilizan para almacenar valores temporales y realizar cálculos específicos en un método.

Por ejemplo, en un método `calcularPromedio`, se pueden declarar variables locales como `suma`, `contador` y `promedio` para calcular el promedio de una lista de números:

```
public double calcularPromedio(int[] numeros) {  
    int suma = 0;  
    int contador = 0;  
  
    for (int numero : numeros) {  
        suma += numero;  
        contador++;  
    }  
  
    double promedio = (double) suma / contador;  
    return promedio;  
}
```

En este caso, las variables `suma`, `contador` y `promedio` se declaran como variables locales dentro del método `calcularPromedio`. Estas variables se utilizan para realizar el cálculo del promedio de los números en la lista `numeros` y solo son visibles dentro de este método.

⚠ Nota: Las variables locales se declaran dentro de un método o bloque de código y su alcance está limitado al contexto en el que se declaran. Estas variables deben inicializarse antes de ser utilizadas y se destruyen al salir del ámbito en el que se declaran.

⚠ Nota: Las variables locales se utilizan para almacenar valores temporales y realizar cálculos específicos en un método. Estas variables son efímeras y solo

existen mientras el método está en ejecución, por lo que no se pueden acceder desde otros métodos o clases.

Las variables locales son útiles para almacenar valores temporales y realizar cálculos específicos en un método. Al utilizar variables locales en un método, se pueden realizar operaciones dinámicas y manipular datos de forma eficiente sin afectar el estado de otros métodos o clases.

Variables de Clase o Variables Estáticas

Las **variables de clase** son variables que se declaran con la palabra clave `static` dentro de una clase y que son compartidas por todos los objetos instanciados a partir de esa clase. Las variables de clase se utilizan para almacenar información común a todos los objetos de la clase.

Por ejemplo, una variable de clase `contador` se puede utilizar para llevar la cuenta de la cantidad de objetos instanciados a partir de una clase `Persona`:

```
public class Persona {  
    private static int contador = 0;  
  
    public Persona() {  
        contador++;  
    }  
}
```

En este caso, la variable de clase `contador` se declara como `static int contador = 0;` dentro de la clase `Persona`.

⚠ Nota: Las variables de clase se declaran con la palabra clave `static` y se inicializan con un valor predeterminado cuando se carga la clase en memoria. Estas variables son compartidas por todos los objetos de la clase y se pueden acceder utilizando el nombre de la clase.

⚠ Nota: Las variables de clase se utilizan para almacenar información común a todos los objetos de la clase y se pueden utilizar para realizar operaciones

globales en la clase. Estas variables son útiles para mantener el estado de la clase y compartir información entre los objetos de la clase.

Las variables de clase son útiles para almacenar información común a todos los objetos de la clase y para realizar operaciones globales en la clase. Al utilizar variables de clase en una clase, se pueden mantener el estado de la clase y compartir información entre los objetos de la clase de forma eficiente.

Resumen

En este tema, se ha cubierto el concepto de literales, constantes y variables en Java. A continuación, se resumen los puntos clave:

- Los **literales** son valores constantes que se utilizan en un programa para representar valores numéricos, de cadena, booleanos, de caracteres, nulos, de arreglos, de punto flotante, de punto flotante en notación científica, numéricos con formato hexadecimal y numéricos con formato binario.
- Las **constantes** son variables cuyo valor no cambia durante la ejecución de un programa. Se declaran con la palabra clave `final` y se utilizan para representar valores fijos que no deben cambiar.
- Las **variables** son espacios de memoria que se utilizan para almacenar valores en un programa. Se declaran con un tipo y un nombre, y se pueden inicializar con un valor opcional.
- Los **atributos de clases** son variables que se declaran dentro de una clase y representan las propiedades o características de dicha clase. Se utilizan para almacenar información sobre los objetos instanciados a partir de esa clase.
- Las **variables locales** son variables que se declaran dentro de un método o bloque de código y solo son visibles dentro de ese ámbito. Se utilizan para almacenar valores temporales y realizar cálculos específicos en un método.

En resumen, los literales, constantes y variables son elementos fundamentales en la programación en Java y se utilizan para representar valores constantes y variables en un

programa. Al comprender estos conceptos, se puede escribir código más claro, eficiente y mantenable en Java.

Alcance de las variables en Java

En Java, el alcance de una variable se refiere a la región del código en la que la variable es visible y puede ser utilizada. El alcance de una variable está determinado por el bloque de código en el que se declara y puede variar según el tipo de variable y su contexto.

Alcance de las variables locales

Las variables locales son aquellas que se declaran dentro de un método, constructor o bloque de código y solo son visibles dentro de ese ámbito. El alcance de una variable local se extiende desde el punto de declaración hasta el final del bloque en el que se encuentra.

Por ejemplo, en el siguiente código, la variable `numero` es local al método `sumar` y solo es visible dentro de ese método:

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

En este caso, la variable `numero` solo es accesible dentro del método `sumar` y no puede ser utilizada en otros métodos de la clase `Calculadora` o fuera de ella.

Alcance de las variables de instancia

Las variables de instancia son aquellas que se declaran a nivel de clase y son visibles en todos los métodos de la clase. El alcance de una variable de instancia se extiende a lo largo de toda la clase y su valor es persistente mientras la instancia de la clase exista.

Por ejemplo, en el siguiente código, la variable `resultado` es una variable de instancia de la clase `Calculadora` y puede ser utilizada en cualquier método de la clase:

```
public class Calculadora {  
    private int resultado;
```

```
public void sumar(int a, int b) {  
    resultado = a + b;  
}  
  
public void restar(int a, int b) {  
    resultado = a - b;  
}  
}
```

En este caso, la variable `resultado` es accesible en los métodos `sumar` y `restar` y su valor se mantiene a lo largo de la vida de la instancia de la clase `Calculadora`.

Alcance de las variables de clase

Las variables de clase son aquellas que se declaran con el modificador `static` y son compartidas por todas las instancias de la clase. El alcance de una variable de clase se extiende a lo largo de toda la clase y su valor es compartido por todas las instancias de la clase.

Por ejemplo, en el siguiente código, la variable `contador` es una variable de clase de la clase `Calculadora` y puede ser utilizada por todas las instancias de la clase:

```
public class Calculadora {  
    private static int contador;  
  
    public Calculadora() {  
        contador++;  
    }  
  
    public static int getContador() {  
        return contador;  
    }  
}
```

En este caso, la variable `contador` es compartida por todas las instancias de la clase `Calculadora` y su valor se incrementa cada vez que se crea una nueva instancia de la clase.

Alcance de las variables de parámetro

Las variables de parámetro son aquellas que se utilizan para pasar valores a un método y son locales al método en el que se declaran. El alcance de una variable de parámetro se extiende desde el punto de declaración hasta el final del método en el que se encuentra.

Por ejemplo, en el siguiente código, las variables `a` y `b` son parámetros del método `sumar` y solo son visibles dentro de ese método:

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

En este caso, las variables `a` y `b` solo son accesibles dentro del método `sumar` y no pueden ser utilizadas en otros métodos de la clase `Calculadora` o fuera de ella.

Variables de bloque

Las variables de bloque son aquellas que se declaran dentro de un bloque de código y solo son visibles dentro de ese bloque. El alcance de una variable de bloque se extiende desde el punto de declaración hasta el final del bloque en el que se encuentra.

Por ejemplo, en el siguiente código, la variable `i` es una variable de bloque que solo es visible dentro del bucle `for`:

```
public class Ejemplo {  
    public void imprimirNumeros() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

En este caso, la variable `i` solo es accesible dentro del bucle `for` y no puede ser utilizada fuera de él.

Conclusión

El alcance de las variables en Java determina la visibilidad y accesibilidad de las variables en diferentes partes de un programa. Comprender el alcance de las variables es fundamental para escribir código claro y evitar errores de referencia o visibilidad. Al seguir las reglas de alcance de las variables, puedes garantizar que tus programas sean coherentes y fáciles de mantener.

Los operadores en Java

Los operadores son símbolos que se utilizan para realizar operaciones sobre variables y valores en un programa. En Java, los operadores se dividen en varias categorías, como aritméticos, de asignación, de comparación, lógicos, de incremento y decremento, entre otros. En este artículo, veremos los operadores más comunes en Java y cómo se utilizan en la programación.

Operadores aritméticos

Los operadores aritméticos se utilizan para realizar operaciones matemáticas básicas, como la suma, resta, multiplicación y división. A continuación, se muestran los operadores aritméticos en Java:

Operador	Descripción	Ejemplo
+	Suma	a + b
-	Resta	a - b
*	Multiplicación	a * b
/	División	a / b
%	Módulo	a % b

```
int a = 10;
int b = 5;

int suma = a + b; // 15
int resta = a - b; // 5
int multiplicacion = a * b; // 50
int division = a / b; // 2
int modulo = a % b; // 0
```

Operadores de asignación

Los operadores de asignación se utilizan para asignar un valor a una variable. En Java, el operador de asignación más común es `=`. A continuación, se muestran algunos ejemplos de operadores de asignación en Java:

Operador	Descripción	Ejemplo
<code>=</code>	Asignación	<code>a = 10</code>
<code>+=</code>	Suma y asignación	<code>a += 5</code>
<code>-=</code>	Resta y asignación	<code>a -= 5</code>
<code>*=</code>	Multiplicación y asignación	<code>a *= 5</code>
<code>/=</code>	División y asignación	<code>a /= 5</code>
<code>%=</code>	Módulo y asignación	<code>a %= 5</code>

```
int a = 10;

a += 5; // a = a + 5 = 15
a -= 5; // a = a - 5 = 10
a *= 5; // a = a * 5 = 50
a /= 5; // a = a / 5 = 2
a %= 5; // a = a % 5 = 0
```

Operadores de comparación

Los operadores de comparación se utilizan para comparar dos valores y devolver un resultado booleano (`true` o `false`). A continuación, se muestran los operadores de comparación en Java:

Operador	Descripción	Ejemplo
<code>==</code>	Igual a	<code>a == b</code>
<code>!=</code>	Diferente de	<code>a != b</code>
<code>></code>	Mayor que	<code>a > b</code>
<code><</code>	Menor que	<code>a < b</code>
<code>>=</code>	Mayor o igual que	<code>a >= b</code>
<code><=</code>	Menor o igual que	<code>a <= b</code>

```
int a = 10;

boolean igual = a == 10; // true
boolean diferente = a != 5; // true
boolean mayor = a > 5; // true
boolean menor = a < 5; // false
boolean mayorIgual = a >= 10; // true
boolean menorIgual = a <= 10; // true
```

Operadores lógicos

Los operadores lógicos se utilizan para combinar expresiones booleanas y devolver un resultado booleano. En Java, los operadores lógicos más comunes son `&&` (AND), `||` (OR) y `!` (NOT). A continuación, se muestran los operadores lógicos en Java:

Operador	Descripción	Ejemplo
&&	AND	a && b
	OR	a b
!	NOT	!a

```
boolean a = true;
boolean b = false;

boolean and = a && b; // false
boolean or = a || b; // true
boolean not = !a; // false
```

Operadores de incremento y decremento

Los operadores de incremento y decremento se utilizan para aumentar o disminuir el valor de una variable en una unidad.

Operador	Descripción	Ejemplo
++	Incremento	a++
--	Decremento	a--

```
int a = 10;

a++; // a = a + 1 = 11
a--; // a = a - 1 = 10
```



Nota: Los operadores de incremento y decremento pueden utilizarse tanto en la forma de prefijo (++a, --a) como en la forma de sufijo (a++, a--). La diferencia

radica en el momento en que se incrementa o decrementa el valor de la variable. En el caso del prefijo, el incremento o decremento se realiza antes de evaluar la expresión, mientras que en el caso del sufijo, el incremento o decremento se realiza después de evaluar la expresión.

Operadores de concatenación

El operador de concatenación `+` se utiliza para unir cadenas de texto en Java. A continuación, se muestra un ejemplo de cómo utilizar el operador de concatenación en Java:

```
String nombre = "Juan";
String apellido = "Pérez";

String nombreCompleto = nombre + " " + apellido; // "Juan Pérez"
```

Las Clases y Objetos en Java

Introducción

En Java, una clase es una plantilla que define el comportamiento y las propiedades de un objeto. Un objeto, por otro lado, es una instancia de una clase que puede contener datos y métodos. En este tutorial, aprenderás cómo definir clases y objetos en Java y cómo interactuar con ellos.

Definición de una clase

Para definir una clase en Java, utiliza la palabra clave `class` seguida del nombre de la clase y las llaves `{ } para encerrar el cuerpo de la clase. Por ejemplo:`

```
public class Persona {  
    // Propiedades  
    String nombre;  
    int edad;  
  
    // Métodos  
    public void saludar() {  
        System.out.println("Hola, soy " + nombre + " y tengo " + edad +  
" años.");  
    }  
}
```

En este ejemplo, se define una clase `Persona` con dos propiedades (`nombre` y `edad`) y un método `saludar` que imprime un mensaje por consola.

Creación de un objeto

Para crear un objeto en Java, utiliza la palabra clave `new` seguida del nombre de la clase y los paréntesis `()`. Por ejemplo:

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

```
// Crear un objeto de la clase Persona  
Persona persona = new Persona();  
  
// Inicializar las propiedades del objeto  
persona.nombre = "Juan";  
persona.edad = 30;  
  
// Llamar al método saludar  
persona.saludar();  
}  
}
```

En este ejemplo, se crea un objeto `persona` de la clase `Persona` y se inicializan sus propiedades `nombre` y `edad`.

Conclusiones

En resumen, las clases y objetos son elementos fundamentales en Java que te permiten modelar el comportamiento y las propiedades de tus aplicaciones. Al definir clases y crear objetos, puedes organizar y reutilizar tu código de manera eficiente. ¡Práctica la creación de clases y objetos en Java para mejorar tus habilidades de programación!

La función `main`

En Java, la función `main` es el punto de entrada de un programa. Es el método que se ejecuta cuando se inicia la aplicación y es el lugar donde comienza la ejecución del código. La función `main` tiene la siguiente firma:

```
public static void main(String[] args) {  
    // Código a ejecutar  
}
```

⚠ Nota: La función `main` es un método estático (`static`) de la clase principal del programa. En este caso, la clase principal puede tener cualquier nombre, pero por convención se suele llamar `Main`. Además, el parámetro `args` es un array de cadenas que se utiliza para pasar argumentos a la aplicación desde la línea de comandos. Toma en cuenta que pueden existir otros métodos `main` en diferentes clases, pero solo uno de ellos será el punto de entrada del programa.

A continuación, se muestra un ejemplo de la función `main` en Java:

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

En este ejemplo, se define una clase `Main` con un método `main` que imprime el mensaje "Hola, mundo!" por consola.

Salidas y Entradas de datos en Java

Salidas de texto en Java

En Java, puedes mostrar salidas de texto en la consola utilizando el método `System.out.println()`. Este método imprime una cadena de texto seguida de un salto de línea. A continuación, se muestra un ejemplo de cómo utilizar el método `println()` en Java:

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

En este ejemplo, se imprime el mensaje "Hola, mundo!" por consola. Puedes utilizar el método `println()` para mostrar mensajes, resultados de operaciones y cualquier otro tipo de información en la consola.

Además del método `println()`, Java también proporciona el método `System.out.print()`, que imprime una cadena de texto sin salto de línea. A continuación, se muestra un ejemplo de cómo utilizar el método `print()` en Java:

```
public class Main {  
    public static void main(String[] args) {  
        System.out.print("Hola, ");  
        System.out.print("mundo!");  
    }  
}
```

En este ejemplo, se imprime el mensaje "Hola, mundo!" sin salto de línea. Puedes utilizar el método `print()` cuando quieras imprimir varias cadenas de texto en la misma línea.

En resumen, los métodos `println()` y `print()` te permiten mostrar salidas de texto en la consola en Java. Utiliza estos métodos para comunicar información a los usuarios y depurar tus programas de manera efectiva.

Existen otras variantes de estos métodos, como `printf()` que permite formatear la salida de texto con especificadores de formato. Puedes explorar estas variantes para adaptar la

presentación de tus salidas de texto a tus necesidades.

Además de ellos existen clases visuales que permiten mostrar salidas de texto en ventanas, como `JOptionPane` que veremos más adelante en el curso.

Entradas de texto en Java

En Java, puedes leer entradas de texto desde la consola utilizando la clase `Scanner` del paquete `java.util`. La clase `Scanner` proporciona métodos para leer diferentes tipos de datos, incluidos enteros, decimales y cadenas de texto.

A continuación, se muestra un ejemplo de cómo utilizar la clase `Scanner` para leer una cadena de texto desde la consola:

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        // Crear un objeto Scanner para leer datos desde la consola
        Scanner scanner = new Scanner(System.in);
        // Mostrar un mensaje para solicitar al usuario que ingrese su
        nombre
        System.out.print("Ingrese su nombre: ");
        // Leer una cadena de texto ingresada por el usuario
        String nombre = scanner.nextLine();
        // Mostrar un saludo personalizado
        System.out.println("Hola, " + nombre + "!");
        // Cerrar el Scanner al finalizar y libera recursos
        scanner.close();
    }
}
```

En este ejemplo, se crea un objeto `Scanner` para leer datos desde la consola. Luego, se utiliza el método `nextLine()` para leer una cadena de texto ingresada por el usuario. Finalmente, se muestra un saludo personalizado con el nombre ingresado.

Además del método `nextLine()`, la clase `Scanner` proporciona otros métodos para leer diferentes tipos de datos:

Método	Descripción	Ejemplo
nextInt()	Lee un entero	int edad = scanner.nextInt();
nextDouble()	Lee un decimal	double precio = scanner.nextDouble();
nextBoolean()	Lee un booleano	boolean activo = scanner.nextBoolean();

Puedes utilizar estos métodos para leer diferentes tipos de datos desde la consola y procesar la información ingresada por el usuario en tus programas.

En resumen, la clase `Scanner` te permite leer entradas de texto desde la consola en Java. Utiliza esta clase para interactuar con los usuarios y crear programas interactivos que respondan a la entrada del usuario.

Entendiendo mejor JOptionPane

Otra forma de leer datos es utilizando la clase `JOptionPane` que se encuentra en el paquete `javax.swing`. Para poder utilizar esta clase se debe importar al inicio del archivo.

```
import javax.swing.JOptionPane;
```

La clase nos ofrece varios métodos para mostrar mensajes y leer datos de los usuarios de una forma más amigable.

Mostrar un mensaje

Para mostrar un mensaje en una ventana de diálogo, se puede utilizar el método `showMessageDialog()` de la clase `JOptionPane`:

```
JOptionPane.showMessageDialog(null, "Hola, mundo!");
```

En este ejemplo, se muestra un mensaje con el texto `Hola, mundo!` en una ventana de diálogo. El primer argumento del método `showMessageDialog()` es el componente padre de la ventana de diálogo, que en este caso es `null`.

Aunque tenemos otras alternativas para `showMessageDialog`, como:

```
JOptionPane.showMessageDialog(null, "Hola, mundo!", "Título",
JOptionPane.INFORMATION_MESSAGE);
```

En este caso, se muestra un mensaje con el texto `Hola, mundo!`, un título `Título` y un ícono de información en una ventana de diálogo.

Siendo los iconos de los mensajes los siguientes:

Icono	Constante
Información	JOptionPane.INFORMATION_MESSAGE
Advertencia	JOptionPane.WARNING_MESSAGE
Error	JOptionPane.ERROR_MESSAGE
Pregunta	JOptionPane.QUESTION_MESSAGE
Sin icono	JOptionPane.PLAIN_MESSAGE

Leer datos

Para leer datos utilizando JOptionPane, se puede utilizar el método showInputDialog() de la clase JOptionPane:

```
String nombre = JOptionPane.showInputDialog("Ingrese su nombre:");
```

En este ejemplo, se muestra un mensaje con el texto Ingrese su nombre: en una ventana de diálogo y se guarda el valor ingresado por el usuario en la variable nombre.

Si se desea leer un número entero, se puede utilizar el método parseInt() de la clase Integer para convertir el valor devuelto por showInputDialog() a un entero:

```
int edad = Integer.parseInt(JOptionPane.showInputDialog("Ingrese su edad:"));
```



Nota: Si el usuario ingresa un valor que no puede ser convertido a un número entero, se lanzará una excepción de tipo NumberFormatException.

Existen otras alternativas para showInputDialog, como:

```
String[] opciones = { "Opción 1", "Opción 2", "Opción 3" };
String opcion = (String) JOptionPane.showInputDialog(null, "Seleccione
```

```
una opción:", "Título",
JOptionPane.QUESTION_MESSAGE, null, opciones, opciones[0]);
```

En este caso, se muestra un cuadro de diálogo con un mensaje, un título, un ícono de pregunta y una lista de opciones para que el usuario seleccione una de ellas.

Ejemplo completo

A continuación se muestra un ejemplo completo que lee el nombre y la edad de una persona utilizando `JOptionPane` y muestra un mensaje con estos datos:

```
import javax.swing.JOptionPane;

class Main {
    public static void main(String[] args) {
        String nombre = JOptionPane.showInputDialog("Ingrese su
nombre:");
        int edad = Integer.parseInt(JOptionPane.showInputDialog("Ingrese
su edad"));

        JOptionPane.showMessageDialog(null, "Hola " + nombre + ", tienes
" + edad + " años.");
    }
}
```

En este ejemplo, se lee el nombre y la edad de una persona utilizando `JOptionPane` y se muestra un mensaje con estos datos en una ventana de diálogo.

Es importante tener en cuenta que `JOptionPane` es una forma sencilla y práctica de interactuar con el usuario en aplicaciones de escritorio en Java, ya que proporciona una interfaz gráfica amigable para mostrar mensajes y leer datos.

Ventana de Confirmación

Otra funcionalidad que nos ofrece `JOptionPane` es la posibilidad de mostrar una ventana de confirmación al usuario. Esta ventana permite al usuario confirmar una acción antes de que se realice.

Para mostrar una ventana de confirmación, se puede utilizar el método `showConfirmDialog()` de la clase `JOptionPane`:

```
int respuesta = JOptionPane.showConfirmDialog(null, "¿Está seguro de que  
desea continuar?", "Confirmación",  
JOptionPane.YES_NO_OPTION);
```

En este ejemplo, se muestra una ventana de confirmación con un mensaje, un título y dos botones de opción (`Sí` y `No`). El método `showConfirmDialog()` devuelve un valor entero que representa la opción seleccionada por el usuario:

- `JOptionPane.YES_OPTION` si el usuario selecciona `Sí`.
- `JOptionPane.NO_OPTION` si el usuario selecciona `No`.
- `JOptionPane.CLOSED_OPTION` si el usuario cierra la ventana sin seleccionar ninguna opción.
- `JOptionPane.CANCEL_OPTION` si el usuario cancela la operación.
- `JOptionPane.OK_OPTION` si el usuario selecciona `Aceptar`.
- `JOptionPane.CANCEL_OPTION` si el usuario selecciona `Cancelar`.

Este valor puede ser utilizado para realizar diferentes acciones en función de la respuesta del usuario.

Las alternativas de `showConfirmDialog` son:

Constante	Descripción
<code>YES_NO_OPTION</code>	Muestra los botones <code>Sí</code> y <code>No</code> .
<code>YES_NO_CANCEL_OPTION</code>	Muestra los botones <code>Sí</code> , <code>No</code> y <code>Cancelar</code> .
<code>OK_CANCEL_OPTION</code>	Muestra los botones <code>Aceptar</code> y <code>Cancelar</code> .

Ejemplo completo de ventana de confirmación

A continuación se muestra un ejemplo completo que muestra una ventana de confirmación al usuario y realiza una acción en función de la respuesta:

```
import javax.swing.JOptionPane;

class Main {
    public static void main(String[] args) {
        int respuesta = JOptionPane.showConfirmDialog(null, "¿Está
seguro de que desea continuar?", "Confirmación",
JOptionPane.YES_NO_OPTION);

        if (respuesta == JOptionPane.YES_OPTION) {
            JOptionPane.showMessageDialog(null, "La operación se realizó
con éxito.");
        } else {
            JOptionPane.showMessageDialog(null, "La operación fue
cancelada.");
        }
    }
}
```

En este ejemplo, se muestra una ventana de confirmación al usuario y se muestra un mensaje diferente en función de la respuesta seleccionada por el usuario.

La ventana de confirmación es una herramienta útil para solicitar la confirmación del usuario antes de realizar una acción importante en una aplicación de escritorio en Java.

Conclusión

JOptionPane es una clase que nos permite interactuar con el usuario de una forma sencilla y amigable en aplicaciones de escritorio en Java. Con JOptionPane, podemos mostrar mensajes, leer datos y solicitar confirmaciones de forma rápida y eficiente. Esta clase es especialmente útil para aplicaciones que requieren una interfaz gráfica simple y directa para interactuar con el usuario.

Estructuras Secuenciales

Dentro de Java, las estructuras secuenciales son aquellas que se ejecutan de manera secuencial, es decir, una tras otra. En este tipo de estructuras, el flujo de ejecución de un programa sigue un orden específico, en el que cada instrucción se ejecuta una vez que la anterior ha finalizado.

En general este tipo de estructuras se utilizan para realizar operaciones sencillas, como la lectura de datos, la realización de cálculos matemáticos, la impresión de resultados, entre otros.

Podemos agrupar las estructuras secuenciales en tres tipos:

- 1. Entrada de datos:** En esta estructura se realiza la lectura de datos desde el teclado o desde un archivo.
- 2. Procesamiento de datos:** En esta estructura se realizan operaciones matemáticas o lógicas con los datos leídos.
- 3. Salida de datos:** En esta estructura se imprime el resultado de las operaciones realizadas.

A continuación, se presentan algunos ejemplos de estructuras secuenciales en Java.

Ejemplo 1: Entrada de datos

En este ejemplo se muestra cómo leer un número entero desde el teclado y almacenarlo en una variable.

```
import java.util.Scanner;

public class EntradaDatos {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int numero;

        System.out.print("Ingrese un número entero: ");
        numero = sc.nextInt();
```

```
        System.out.println("El número ingresado es: " + numero);
    }
}
```

Ejemplo 2: Procesamiento de datos

En este ejemplo se muestra cómo realizar una operación matemática con dos números enteros.

```
public class ProcesamientoDatos {
    public static void main(String[] args) {
        int numero1 = 5;
        int numero2 = 3;
        int suma, resta, multiplicacion, division;

        suma = numero1 + numero2;
        resta = numero1 - numero2;
        multiplicacion = numero1 * numero2;
        division = numero1 / numero2;

        System.out.println("Suma: " + suma);
        System.out.println("Resta: " + resta);
        System.out.println("Multiplicación: " + multiplicacion);
        System.out.println("División: " + division);
    }
}
```

Ejemplo 3: Salida de datos

En este ejemplo se muestra cómo imprimir un mensaje en la consola.

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

```
 }  
 }
```

Estos son solo algunos ejemplos de estructuras secuenciales en Java. Es importante tener en cuenta que en la programación se pueden combinar diferentes estructuras para lograr un resultado más complejo.

Para más información sobre las estructuras secuenciales en Java, se recomienda consultar la documentación oficial de Java en el siguiente enlace: Java Documentation (<https://docs.oracle.com/en/java/>).

Conclusiones

Las estructuras secuenciales son fundamentales en la programación, ya que permiten realizar operaciones de manera ordenada y secuencial. En Java, estas estructuras se utilizan para leer datos, procesarlos y mostrar los resultados.

Es importante tener en cuenta que las estructuras secuenciales son solo una parte de la programación, y que existen otras estructuras más complejas que permiten realizar tareas más avanzadas.

En resumen, las estructuras secuenciales son la base de la programación y son fundamentales para comprender y dominar cualquier lenguaje de programación.

La estructura de decisión en Java

La estructura de decisión en Java permite ejecutar un bloque de código si se cumple una condición determinada. En caso de que la condición no se cumpla, se puede ejecutar otro bloque de código. Existen tres tipos de estructuras de decisión en Java: `if`, `if-else` y `switch`.

Estructura if

La estructura `if` permite ejecutar un bloque de código si se cumple una condición determinada. La sintaxis de la estructura `if` es la siguiente:

```
if (condicion) {  
    // Bloque de código a ejecutar si la condición es verdadera  
}
```

A continuación, se presenta un ejemplo de la estructura `if` en Java:

```
public class EstructuraIf {  
    public static void main(String[] args) {  
        int numero = 10;  
  
        if (numero > 0) {  
            System.out.println("El número es positivo");  
        }  
    }  
}
```

En este ejemplo, se verifica si el número es mayor que cero y se imprime un mensaje en la consola si la condición es verdadera.

Estructura if-else

La estructura `if-else` permite ejecutar un bloque de código si se cumple una condición determinada y otro bloque de código si la condición no se cumple. La sintaxis de la estructura `if-else` es la siguiente:

```
if (condicion) {  
    // Bloque de código a ejecutar si la condición es verdadera  
} else {  
    // Bloque de código a ejecutar si la condición es falsa  
}
```

A continuación, se presenta un ejemplo de la estructura if-else en Java:

```
public class EstructuraIfElse {  
    public static void main(String[] args) {  
        int numero = -5;  
  
        if (numero > 0) {  
            System.out.println("El número es positivo");  
        } else {  
            System.out.println("El número es negativo");  
        }  
    }  
}
```

En este ejemplo, se verifica si el número es mayor que cero y se imprime un mensaje en la consola dependiendo de si la condición es verdadera o falsa.

Estructura if-else if

La estructura if-else if permite evaluar múltiples condiciones en secuencia y ejecutar un bloque de código si alguna de las condiciones se cumple. La sintaxis de la estructura if-else if es la siguiente:

```
if (condicion1) {  
    // Bloque de código a ejecutar si la condición 1 es verdadera  
} else if (condicion2) {  
    // Bloque de código a ejecutar si la condición 2 es verdadera  
} else {  
    // Bloque de código a ejecutar si ninguna de las condiciones
```

```
anteriores es verdadera  
}
```

A continuación, se presenta un ejemplo de la estructura `if-else if` en Java:

```
public class EstructuraIfElseIf {  
    public static void main(String[] args) {  
        int numero = 0;  
  
        if (numero > 0) {  
            System.out.println("El número es positivo");  
        } else if (numero < 0) {  
            System.out.println("El número es negativo");  
        } else {  
            System.out.println("El número es cero");  
        }  
    }  
}
```

En este ejemplo, se verifica si el número es mayor que cero, menor que cero o igual a cero y se imprime un mensaje en la consola dependiendo de la condición que se cumpla.

Toma en cuenta que la estructura `if-else if` puede tener tantas condiciones intermedias como sea necesario.

Por ejemplo, si se desea evaluar si un número es positivo, negativo o cero, se pueden agregar más condiciones intermedias:

```
if (numero > 0) {  
    System.out.println("El número es positivo");  
} else if (numero < 0) {  
    System.out.println("El número es negativo");  
} else if (numero == 0) {  
    System.out.println("El número es cero");  
} else {
```

```
        System.out.println("El número no es un entero");
    }
```

Estructura switch

La estructura `switch` permite evaluar una expresión y ejecutar un bloque de código dependiendo del valor de la expresión. La sintaxis de la estructura `switch` es la siguiente:

```
switch (expresion) {
    case valor1:
        // Bloque de código a ejecutar si la expresión es igual a valor1
        break;
    case valor2:
        // Bloque de código a ejecutar si la expresión es igual a valor2
        break;
    default:
        // Bloque de código a ejecutar si la expresión no coincide con
        ningún caso anterior
}
```

⚠ Nota: La palabra clave `break` se utiliza para salir del bloque `switch` una vez se ha ejecutado el código correspondiente a un caso.

A continuación, se presenta un ejemplo de la estructura `switch` en Java:

```
public class EstructuraSwitch {
    public static void main(String[] args) {
        int diaSemana = 1;

        switch (diaSemana) {
            case 1:
                System.out.println("Lunes");
                break;
            case 2:
                System.out.println("Martes");
```

```

        break;
    case 3:
        System.out.println("Miércoles");
        break;
    case 4:
        System.out.println("Jueves");
        break;
    case 5:
        System.out.println("Viernes");
        break;
    case 6:
        System.out.println("Sábado");
        break;
    case 7:
        System.out.println("Domingo");
        break;
    default:
        System.out.println("Día no válido");
    }
}
}

```

En este ejemplo, se evalúa el valor de la variable `diaSemana` y se imprime el nombre del día correspondiente en la consola.

Sin embargo, en las nuevas versiones de Java, la estructura `switch` ha sido mejorada para permitir el uso de expresiones más complejas y la eliminación de la palabra clave `break`. A partir de Java 12, se puede utilizar la estructura `switch` de la siguiente manera:

```

public class EstructuraSwitchJava12 {
    public static void main(String[] args) {
        int diaSemana = 1;

        switch (diaSemana) {
            case 1 -> System.out.println("Lunes");
            case 2 -> System.out.println("Martes");
            case 3 -> System.out.println("Miércoles");
        }
    }
}

```

```

        case 4 -> System.out.println("Jueves");
        case 5 -> System.out.println("Viernes");
        case 6 -> System.out.println("Sábado");
        case 7 -> System.out.println("Domingo");
        default -> System.out.println("Día no válido");
    }
}
}

```

En este ejemplo, se utiliza la nueva sintaxis de la estructura `switch` introducida en Java 12, que permite simplificar la escritura del código y eliminar la necesidad de la palabra clave `break`.

Esta versión mejorada de la estructura `switch` es más concisa y legible, lo que facilita la escritura y mantenimiento del código. Además de usarse en situaciones en las cuales se necesita devolver un valor respecto a una expresión, como por ejemplo:

```

public class EstructuraSwitchJava12 {

    public static String obtenerNombreDia(int diaSemana) {
        return switch (diaSemana) {
            case 1 -> "Lunes";
            case 2 -> "Martes";
            case 3 -> "Miércoles";
            case 4 -> "Jueves";
            case 5 -> "Viernes";
            case 6 -> "Sábado";
            case 7 -> "Domingo";
            default -> "Día no válido";
        };
    }

    public static void main(String[] args) {
        int diaSemana = 1;
        System.out.println(obtenerNombreDia(diaSemana));
    }
}

```

```
 }  
 }
```

Conclusión

Las estructuras de decisión en Java son fundamentales para controlar el flujo de ejecución de un programa y permiten realizar acciones condicionales en función de los valores de las variables. Es importante comprender cómo funcionan las estructuras `if`, `if-else`, `if-else if` y `switch` para poder diseñar programas más complejos y eficientes.

Es recomendable practicar con ejemplos y casos de uso reales para familiarizarse con el uso de las estructuras de decisión en Java y mejorar las habilidades de programación.

Para más información sobre las estructuras de decisión en Java, se recomienda consultar la documentación oficial de Java en el siguiente enlace: Java Documentation (<https://docs.oracle.com/en/java/>).

El Operador Ternario en Java

El operador ternario en Java es una forma abreviada de escribir una estructura `if-else` en una sola línea. Este operador se utiliza para evaluar una expresión y devolver un valor dependiendo de si la expresión es verdadera o falsa. La sintaxis del operador ternario es la siguiente:

```
variable = (condición) ? valor_si_verdadero : valor_si_falso;
```

Donde:

- `condición` es la expresión que se evalúa.
- `valor_si_verdadero` es el valor que se asigna a la variable si la condición es verdadera.
- `valor_si_falso` es el valor que se asigna a la variable si la condición es falsa.
- `variable` es la variable a la que se asigna el valor.
- Los dos puntos `:` separan el valor si verdadero del valor si falso.
- Los paréntesis `()` son opcionales, pero se utilizan para mejorar la legibilidad del código.
- El operador ternario devuelve un valor, por lo que se puede utilizar en cualquier lugar donde se espere un valor.

A continuación, se presenta un ejemplo del operador ternario en Java:

```
public class OperadorTernario {  
    public static void main(String[] args) {  
        int numero = 10;  
        String mensaje = (numero > 0) ? "El número es positivo" : "El  
número es negativo";  
  
        System.out.println(mensaje);  
    }  
}
```

```
 }  
 }
```

En este ejemplo, se verifica si el número es mayor que cero y se asigna un mensaje dependiendo de si la condición es verdadera o falsa. El mensaje se imprime en la consola al final.

El operador ternario es una forma concisa de escribir estructuras `if-else` simples y puede mejorar la legibilidad del código en ciertos casos. Sin embargo, se debe tener cuidado al utilizarlo para no complicar demasiado la lógica del programa.

Conclusión

El operador ternario en Java es una forma abreviada de escribir estructuras `if-else` en una sola línea. Permite evaluar una expresión y devolver un valor dependiendo de si la expresión es verdadera o falsa. El operador ternario puede mejorar la legibilidad del código en ciertos casos, pero se debe utilizar con moderación para no complicar demasiado la lógica del programa.

El Ciclo `while` en Java

El ciclo `while` en Java es una estructura de control que se utiliza para repetir un bloque de código mientras una condición sea verdadera. La sintaxis del ciclo `while` es la siguiente:

```
while (condición) {  
    // Código a ejecutar mientras la condición sea verdadera  
}
```

Donde:

- `condición` es una expresión booleana que se evalúa antes de cada iteración del ciclo.
- El bloque de código entre llaves `{ }` se ejecuta mientras la condición sea verdadera.
- Si la condición es falsa desde el principio, el bloque de código no se ejecuta.

A continuación, se presenta un ejemplo del ciclo `while` en Java:

```
public class CicloWhile {  
    public static void main(String[] args) {  
        int contador = 0;  
  
        while (contador < 5) {  
            System.out.println("Contador: " + contador);  
            contador++;  
        }  
    }  
}
```

En este ejemplo, se inicializa un contador en cero y se imprime su valor mientras sea menor que 5. Después de cada iteración, se incrementa el contador en uno. El resultado de este programa sería:

```
Contador: 0  
Contador: 1  
Contador: 2
```

```
Contador: 3
```

```
Contador: 4
```

El ciclo `while` es útil cuando se necesita repetir un bloque de código un número indeterminado de veces, siempre y cuando se cumpla una condición. Es importante tener cuidado con la condición del ciclo para evitar caer en un bucle infinito, donde el código se ejecuta continuamente sin terminar.

Las Palabras Clave `break` y `continue`

Dentro de un ciclo `while`, se pueden utilizar las palabras clave `break` y `continue` para controlar el flujo de la ejecución. Estas palabras clave tienen los siguientes efectos:

- `break`: Termina inmediatamente la ejecución del ciclo y continúa con la siguiente instrucción después del ciclo.
- `continue`: Salta a la siguiente iteración del ciclo, omitiendo el resto del bloque de código actual.

A continuación, se presenta un ejemplo del uso de `break` y `continue` en un ciclo `while`:

```
public class BreakContinue {  
    public static void main(String[] args) {  
        int contador = 0;  
  
        while (contador < 5) {  
            contador++;  
  
            if (contador == 2) {  
                continue;  
            }  
  
            if (contador == 4) {  
                break;  
            }  
  
            System.out.println("Contador: " + contador);  
        }  
    }  
}
```

```
    }  
}  
}
```

En este ejemplo, se utiliza `continue` para saltar la iteración cuando el contador es igual a 2, y `break` para salir del ciclo cuando el contador es igual a 4. El resultado de este programa sería:

```
Contador: 1  
Contador: 3
```

El uso de `break` y `continue` puede ayudar a controlar el flujo de un ciclo `while` y a evitar bucles infinitos o repetición innecesaria de código.

Conclusión

El ciclo `while` en Java es una estructura de control que se utiliza para repetir un bloque de código mientras una condición sea verdadera. Se compone de una condición que se evalúa antes de cada iteración y un bloque de código que se ejecuta mientras la condición sea verdadera. Es importante tener cuidado con la condición para evitar bucles infinitos. Además, se pueden utilizar las palabras clave `break` y `continue` para controlar el flujo de la ejecución dentro del ciclo. El ciclo `while` es una herramienta poderosa para implementar la repetición en un programa Java de forma controlada y eficiente.

El Ciclo `do-while` en Java

El ciclo do-while en Java es una estructura de control similar al ciclo while, pero con una diferencia importante: el bloque de código se ejecuta al menos una vez, y luego se repite mientras una condición sea verdadera. La sintaxis del ciclo do-while es la siguiente:

```
do {  
    // Código a ejecutar al menos una vez  
} while (condición);
```

Donde:

- El bloque de código entre llaves {} se ejecuta al menos una vez, independientemente de la condición.
- condición es una expresión booleana que se evalúa después de cada iteración del ciclo.
- Si la condición es verdadera, el bloque de código se repite; de lo contrario, el ciclo termina.

A continuación, se presenta un ejemplo del ciclo do-while en Java:

```
public class CicloDoWhile {  
    public static void main(String[] args) {  
        int contador = 0;  
  
        do {  
            System.out.println("Contador: " + contador);  
            contador++;  
        } while (contador < 5);  
    }  
}
```

En este ejemplo, se inicializa un contador en cero y se imprime su valor al menos una vez. Después de cada iteración, se incrementa el contador en uno. El resultado de este programa sería:

```
Contador: 0  
Contador: 1  
Contador: 2  
Contador: 3  
Contador: 4
```

El ciclo `do-while` es útil cuando se necesita ejecutar un bloque de código al menos una vez, y luego repetirlo mientras se cumpla una condición. Al igual que con el ciclo `while`, es importante tener cuidado con la condición para evitar caer en un bucle infinito.

De `do-while` a `while` y Viceversa

En algunos casos, se puede transformar un ciclo `do-while` en un ciclo `while` y viceversa, manteniendo el mismo comportamiento. Por ejemplo, el ciclo `do-while` del ejemplo anterior se puede reescribir como un ciclo `while` de la siguiente manera:

```
public class CicloWhile {  
    public static void main(String[] args) {  
        int contador = 0;  
  
        while (true) {  
            System.out.println("Contador: " + contador);  
            contador++;  
  
            if (contador >= 5) {  
                break;  
            }  
        }  
    }  
}
```

En este caso, se utiliza un ciclo `while` con una condición `true` y se agrega una instrucción `break` para salir del ciclo cuando el contador llega a 5. De manera similar, un ciclo `while` se puede transformar en un ciclo `do-while` de la siguiente manera:

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

```
int contador = 0;

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

En este caso, se inicializa el contador fuera del ciclo y se ejecuta el bloque de código al menos una vez antes de verificar la condición.

Ambas formas de escribir los ciclos son válidas y pueden utilizarse según la preferencia del programador y la legibilidad del código. Es importante recordar que la elección entre un ciclo `do-while` y un ciclo `while` depende de si se necesita ejecutar el bloque de código al menos una vez o no.

Ahora veamos la comparación de un mismo código con `do-while` y `while`:

Conclusión

El ciclo `do-while` en Java es una estructura de control que se utiliza para ejecutar un bloque de código al menos una vez, y luego repetirlo mientras una condición sea verdadera. El ciclo `do-while` es útil cuando se necesita garantizar que el bloque de código se ejecute al menos una vez, independientemente de la condición. Al igual que con otros ciclos, es importante tener cuidado con la condición para evitar bucles infinitos y asegurar que el ciclo se comporte como se espera.

El Ciclo `for` en Java y su versión mejorada `for-each`

El ciclo `for` en Java es una estructura de control que se utiliza para repetir un bloque de código un número determinado de veces. La sintaxis del ciclo `for` es la siguiente:

```
for (inicialización; condición; actualización) {  
    // Código a ejecutar mientras la condición sea verdadera  
}
```

Donde:

- `inicialización` es una expresión que se ejecuta una sola vez al inicio del ciclo.
- `condición` es una expresión booleana que se evalúa antes de cada iteración del ciclo.
- `actualización` es una expresión que se ejecuta al final de cada iteración del ciclo.
- El bloque de código entre llaves `{ }` se ejecuta mientras la condición sea verdadera.
- Si la condición es falsa desde el principio, el bloque de código no se ejecuta.



Nota: La `inicialización`, `condición` y `actualización` son opcionales, pero los puntos y coma `;` deben mantenerse en su lugar. Así que por ejemplo, si no se necesita una `inicialización`, se puede dejar en blanco `,` pero el punto y coma `;` debe permanecer.



Nota: El segundo punto y coma `;` es obligatorio, incluso si no se necesita una `actualización`.



Nota: La `actualización` puede ser cualquier expresión válida en Java, como un incremento `i++` o un decremento `i--`. E incluso puede ser una expresión más compleja, como `i += 2` o `i *= 3`.

A continuación, se presenta un ejemplo del ciclo `for` en Java:

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

En este ejemplo, se inicializa un contador en cero y se imprime su valor mientras sea menor que 5. Después de cada iteración, se incrementa el contador en uno. El resultado de este programa sería:

```
Contador: 0  
Contador: 1  
Contador: 2  
Contador: 3  
Contador: 4
```

El ciclo `for` es útil cuando se necesita repetir un bloque de código un número determinado de veces. Es una forma concisa de escribir ciclos y es especialmente útil cuando se conoce de antemano cuántas veces se debe repetir el código.

La versión mejorada `for-each`

Además del ciclo `for` tradicional, Java también ofrece una versión mejorada conocida como `for-each` o `for-each loop`. Esta versión simplifica la iteración sobre arreglos y colecciones, eliminando la necesidad de manejar índices y longitudes. La sintaxis del ciclo `for-each` es la siguiente:

```
for (tipo variable : arreglo) {  
    // Código a ejecutar para cada elemento del arreglo  
}
```

Donde:

- **tipo** es el tipo de dato de los elementos del arreglo.
- **variable** es la variable que representa cada elemento del arreglo.
- **arreglo** es el arreglo sobre el cual se está iterando.
- El bloque de código entre llaves **{ }** se ejecuta para cada elemento del arreglo.
- La variable **variable** toma el valor de cada elemento del arreglo en cada iteración.
- El ciclo **for-each** itera sobre todos los elementos del arreglo, en orden, sin necesidad de manejar índices.

A continuación, se presenta un ejemplo del ciclo **for-each** en Java:

```
public class CicloForEach {  
    public static void main(String[] args) {  
        int[] numeros = {1, 2, 3, 4, 5};  
  
        for (int numero : numeros) {  
            System.out.println("Número: " + numero);  
        }  
    }  
}
```

En este ejemplo, se crea un arreglo de números y se itera sobre cada elemento utilizando el ciclo **for-each**. El resultado de este programa sería:

```
Número: 1  
Número: 2  
Número: 3  
Número: 4  
Número: 5
```

El ciclo **for-each** es una forma más simple y segura de iterar sobre arreglos y colecciones en Java. Se recomienda utilizar esta versión mejorada siempre que sea posible, ya que simplifica el código y reduce la posibilidad de errores relacionados con los índices y las longitudes de los arreglos.

Anidación de ciclos en Java

En Java, es posible anidar ciclos, es decir, colocar un ciclo dentro de otro ciclo. Esto permite realizar operaciones más complejas y estructuras de control más avanzadas. En este tutorial, aprenderás cómo anidar ciclos en Java y cómo utilizarlos en tus programas.

Sintaxis de la anidación de ciclos

La anidación de ciclos en Java implica colocar un ciclo dentro de otro ciclo. La sintaxis general es la siguiente:

```
for (inicialización; condición; actualización) {  
    // Código del ciclo externo  
  
    for (inicialización; condición; actualización) {  
        // Código del ciclo interno  
    }  
}
```

En este ejemplo, se coloca un ciclo `for` dentro de otro ciclo `for`. El ciclo externo se ejecuta primero y, dentro de él, se ejecuta el ciclo interno. Esto permite realizar operaciones más complejas que requieren múltiples iteraciones.

Ejemplos de anidación de ciclos en Java

A continuación, se presentan algunos ejemplos de cómo anidar ciclos en Java y cómo utilizarlos en diferentes situaciones.

Ejemplo 1: Tabla de multiplicar

En este ejemplo, se muestra cómo utilizar la anidación de ciclos para generar una tabla de multiplicar del 1 al 10.

```
public class TablaMultiplicar {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 10; i++) {  
            System.out.println("Tabla de multiplicar del " + i + ":");  
        }  
    }  
}
```

```
        for (int j = 1; j <= 10; j++) {
            System.out.println(i + " x " + j + " = " + (i * j));
        }
    }
}
```

En este ejemplo, se utiliza un ciclo externo para recorrer los números del 1 al 10 y un ciclo interno para generar la tabla de multiplicar de cada número.

Anidando distintos tipos de ciclos

Es importante tener en cuenta que se pueden anidar distintos tipos de ciclos en Java, como ciclos `for`, `while` y `do-while`. La elección del tipo de ciclo dependerá de la situación y de la lógica del programa.

Ejemplo 2: Patrón de asteriscos con ciclos distintos

En este ejemplo, se muestra cómo utilizar la anidación de ciclos con distintos tipos para generar un patrón de asteriscos.

```
public class PatronAsteriscos {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            int j = 1;
            while (j <= i) {
                System.out.print("* ");
                j++;
            }
            System.out.println();
        }
    }
}
```

En este ejemplo, se utiliza un ciclo `for` externo y un ciclo `while` interno para generar un patrón de asteriscos creciente.

Conclusiones

La anidación de ciclos en Java es una técnica poderosa que permite realizar operaciones más complejas y estructuras de control más avanzadas. Al comprender cómo anidar ciclos y cómo utilizarlos en diferentes situaciones, podrás escribir programas más eficientes y elegantes en Java.

¿Qué son las clases predefinidas en Java?

Las clases predefinidas en Java son un conjunto de clases que forman parte del lenguaje y que proporcionan funcionalidades básicas y comunes para el desarrollo de aplicaciones. Estas clases están disponibles en la biblioteca estándar de Java y se pueden utilizar directamente en los programas sin necesidad de importar paquetes adicionales.

Las clases predefinidas en Java abarcan una amplia gama de funcionalidades, desde el manejo de cadenas de texto y números, hasta la interacción con el sistema operativo y la entrada y salida de datos. Algunas de las clases predefinidas más comunes en Java son:

Clase	Descripción
String	Clase para el manejo de cadenas de texto.
Integer	Clase envoltorio para números enteros.
Double	Clase envoltorio para números decimales.
Boolean	Clase envoltorio para valores booleanos.
Math	Clase con funciones matemáticas estáticas.
System	Clase para interactuar con el sistema operativo.
Scanner	Clase para leer datos de la entrada estándar.
PrintStream	Clase para escribir datos en la salida estándar.
File	Clase para representar archivos y directorios.
DateFormat	Clase para formatear fechas y horas.
Random	Clase para generar números aleatorios.
ArrayList	Clase para almacenar colecciones de objetos de manera dinámica.
HashMap	Clase para almacenar pares clave-valor de manera eficiente.
Thread	Clase para crear y controlar hilos de ejecución.
Exception	Clase base para excepciones en Java.

Object	Clase base de la jerarquía de clases en Java.
StringBuider	Clase para construir cadenas de texto de manera eficiente.

Estas clases predefinidas proporcionan una base sólida para el desarrollo de aplicaciones en Java y cubren la mayoría de las necesidades comunes de los programadores. Al utilizar estas clases, los desarrolladores pueden aprovechar las funcionalidades existentes y centrarse en la lógica específica de sus aplicaciones, en lugar de tener que implementar funcionalidades básicas desde cero.

Los Wrappers

Los **wrappers** son clases que permiten representar tipos de datos primitivos como objetos en Java. Estas clases se utilizan para envolver los tipos primitivos y proporcionar funcionalidades adicionales, como métodos y constantes útiles para el manejo de estos tipos de datos.

En Java, los **wrappers** están disponibles para los siguientes tipos de datos primitivos:

Tipo Primitivo	Wrapper
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character

Funciones comunes de los Wrappers

Los **wrappers** proporcionan una serie de funciones comunes para trabajar con los tipos de datos primitivos. Algunas de las funciones más utilizadas son las siguientes:

Función	Descripción
<code>parseTipo(String)</code>	Convierte una cadena en el tipo de dato correspondiente.
<code>valueOf(tipo)</code>	Convierte un tipo de dato en el wrapper correspondiente.
<code>tipoValue()</code>	Devuelve el valor del tipo de dato envuelto por el wrapper .
<code>compareTo(tipo)</code>	Compara el valor del wrapper con otro valor del mismo tipo.
<code>equals(Object)</code>	Compara el wrapper con otro objeto para determinar si son iguales.
<code>toString()</code>	Convierte el valor del wrapper en una cadena de texto.
<code>hashCode()</code>	Devuelve el código hash del wrapper .
<code>TYPE</code>	Constante que representa el tipo de dato primitivo envuelto por el wrapper .

Ejemplos de Uso

A continuación, se presentan algunos ejemplos de cómo utilizar los **wrappers** en Java:

```
public class Wrappers {
    public static void main(String[] args) {
        // Crear un Integer a partir de un int
        Integer entero = Integer.valueOf(42);

        // Convertir un String en un Double
        Double decimal = Double.parseDouble("3.14159");

        // Comparar dos Longs
    }
}
```

```

        Long numero1 = Long.valueOf(1000);
        Long numero2 = Long.valueOf(2000);
        int comparacion = numero1.compareTo(numero2);

        // Imprimir el valor de un Character
        Character letra = Character.valueOf('A');
        System.out.println("Letra: " + letra.toString());
    }
}

```

En estos ejemplos, se utilizan los **wrappers** para convertir tipos de datos primitivos en objetos, realizar comparaciones entre valores y obtener representaciones en cadena de los valores envueltos.

Los **wrappers** son útiles cuando se necesita trabajar con tipos de datos primitivos como objetos, ya que proporcionan métodos y funcionalidades adicionales que no están disponibles para los tipos primitivos directamente.



Nota: A partir de Java 5, se introdujo el concepto de **autoboxing** y **unboxing**, que permiten la conversión automática entre tipos primitivos y sus **wrappers**. Esto simplifica el código y hace que sea más fácil trabajar con ambos tipos de datos de manera intercambiable.

Como dato interesante la función `compareTo` de los **wrappers** devuelve un valor entero que indica si el valor del **wrapper** es menor, igual o mayor que el valor del otro objeto.

Este valor es negativo si el **wrapper** es menor, cero si son iguales y positivo si es mayor.

Por otro lado, la función `equals` compara el **wrapper** con otro objeto y devuelve `true` si son iguales y `false` en caso contrario. Es importante tener en cuenta que esta función compara los valores de los **wrappers**, no las referencias a los objetos.

La clase 'Math'

La clase `Math` en Java es una clase integrada que proporciona métodos estáticos para realizar operaciones matemáticas comunes. Estos métodos pueden ser utilizados para realizar cálculos matemáticos en un programa Java de manera sencilla y eficiente.

Métodos de la clase Math

A continuación, se presentan algunos de los métodos más comunes de la clase `Math` en Java:

Método	Descripción
<code>abs(x)</code>	Devuelve el valor absoluto de un número <code>x</code> .
<code>ceil(x)</code>	Devuelve el entero más pequeño que es mayor o igual a <code>x</code> .
<code>floor(x)</code>	Devuelve el entero más grande que es menor o igual a <code>x</code> .
<code>round(x)</code>	Devuelve el valor redondeado de <code>x</code> al entero más cercano.
<code>max(x, y)</code>	Devuelve el valor más grande entre <code>x</code> e <code>y</code> .
<code>min(x, y)</code>	Devuelve el valor más pequeño entre <code>x</code> e <code>y</code> .
<code>pow(x, y)</code>	Devuelve <code>x</code> elevado a la potencia <code>y</code> .
<code>sqrt(x)</code>	Devuelve la raíz cuadrada de <code>x</code> .
<code>random()</code>	Devuelve un número aleatorio entre 0.0 y 1.0.
<code>sin(x)</code>	Devuelve el seno de un ángulo <code>x</code> en radianes.
<code>cos(x)</code>	Devuelve el coseno de un ángulo <code>x</code> en radianes.
<code>tan(x)</code>	Devuelve la tangente de un ángulo <code>x</code> en radianes.
<code>toDegrees(x)</code>	Convierte un ángulo <code>x</code> de radianes a grados.

Ejemplos de uso de la clase Math

A continuación, se presentan algunos ejemplos de uso de los métodos de la clase `Math` en Java:

Ejemplo 1: Calcular el valor absoluto de un número

```
public class ValorAbsoluto {  
    public static void main(String[] args) {  
        int numero = -5;  
        int valorAbsoluto = Math.abs(numero);  
  
        System.out.println("El valor absoluto de " + numero + " es " +  
                           valorAbsoluto);  
    }  
}
```

En este ejemplo, se calcula el valor absoluto de un número utilizando el método `abs()` de la clase `Math`.

Ejemplo 2: Calcular la raíz cuadrada de un número

```
public class RaizCuadrada {  
    public static void main(String[] args) {  
        double numero = 16;  
        double raizCuadrada = Math.sqrt(numero);  
  
        System.out.println("La raíz cuadrada de " + numero + " es " +  
                           raizCuadrada);  
    }  
}
```

En este ejemplo, se calcula la raíz cuadrada de un número utilizando el método `sqrt()` de la clase `Math`.

Ejemplo 3: Generar un número aleatorio

```
public class NumeroAleatorio {  
    public static void main(String[] args) {  
        double numeroAleatorio = Math.random();  
  
        System.out.println("Número aleatorio: " + numeroAleatorio);  
    }  
}
```

```
    }  
}
```

En este ejemplo, se genera un número aleatorio entre 0.0 y 1.0 utilizando el método `random()` de la clase `Math`.

Ejemplo 4: Generar un número aleatorio de un rango específico

```
public class NumeroAleatorioRango {  
    public static void main(String[] args) {  
        int min = 1;  
        int max = 10;  
        int numeroAleatorio = (int) (Math.random() * (max - min + 1)) +  
min;  
  
        System.out.println("Número aleatorio entre " + min + " y " + max  
+ ": " + numeroAleatorio);  
    }  
}
```

En este ejemplo, se genera un número aleatorio entre un rango específico utilizando el método `random()` de la clase `Math`.

Conclusión

La clase `Math` en Java proporciona una amplia gama de métodos para realizar operaciones matemáticas comunes de manera sencilla y eficiente. Estos métodos son útiles para realizar cálculos matemáticos en un programa Java y pueden ser utilizados en una variedad de situaciones. Al conocer los métodos disponibles en la clase `Math`, los programadores pueden aprovechar al máximo las capacidades matemáticas de Java y desarrollar aplicaciones más sofisticadas y funcionales.

StringBuilder

La clase `StringBuilder` en Java es una alternativa eficiente para la concatenación de cadenas de texto. A diferencia de la clase `String`, que es inmutable, `StringBuilder` permite modificar el contenido de la cadena sin crear un nuevo objeto en cada operación.

Creación de un `StringBuilder`

Para crear un objeto `StringBuilder`, se puede utilizar uno de los siguientes métodos:

- **Constructor sin argumentos:** Crea un objeto vacío con una capacidad inicial de 16 caracteres.
 - `StringBuilder sb = new StringBuilder();`
- **Constructor con capacidad inicial:** Crea un objeto vacío con la capacidad inicial especificada.
 - `StringBuilder sb = new StringBuilder(32);`
- **Constructor con cadena inicial:** Crea un objeto con la cadena especificada y una capacidad adicional de 16 caracteres.
 - `StringBuilder sb = new StringBuilder("Hola");`

Métodos de `StringBuilder`

Algunos de los métodos más comunes de la clase `StringBuilder` son los siguientes:

Método	Descripción
append(valor)	Agrega el valor especificado al final de la cadena.
insert(pos, valor)	Inserta el valor especificado en la posición indicada.
delete(inicio, fin)	Elimina los caracteres en el rango especificado.
replace(inicio, fin, nuevo)	Reemplaza los caracteres en el rango especificado con la nueva cadena.
reverse()	Invierte el contenido de la cadena.
length()	Devuelve la longitud de la cadena.
capacity()	Devuelve la capacidad actual de la cadena.
toString()	Convierte el <code>StringBuilder</code> en una cadena de texto (<code>String</code>).

Ejemplo de Uso

A continuación, se presenta un ejemplo de cómo utilizar la clase `StringBuilder` en Java:

```
public class EjemploStringBuilder {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Hola, ");
        sb.append("mundo!");
        sb.insert(5, "Java ");
        sb.replace(0, 5, "¡Adiós,");
        sb.delete(5, 10);
        sb.reverse();
        System.out.println(sb.toString());
    }
}
```

```
    }  
}
```

La salida de este programa sería:

```
, sóidA!
```

En este ejemplo, se crea un objeto `StringBuilder` con la cadena "Hola," y se realizan varias operaciones de modificación sobre ella. Al final, se imprime el contenido del `StringBuilder` convertido a una cadena de texto.

El uso de `StringBuilder` es más eficiente que la concatenación de cadenas con el operador `+`, especialmente cuando se realizan múltiples operaciones de modificación sobre la cadena. Al utilizar `StringBuilder`, se evita la creación repetida de objetos `String` y se mejora el rendimiento de la aplicación.

Validación de entradas de datos

Introducción

La validación de datos es un proceso que se realiza para asegurar que los datos introducidos en un sistema son correctos y útiles. La validación de datos es importante para garantizar la integridad y la calidad de los datos, así como para prevenir errores y problemas en el sistema.

En Java, la validación de datos se puede realizar de varias maneras. En este artículo, veremos cómo validar datos en Java utilizando las clases Wrapper y la clase JOptionPane.

Validación de datos con clases Wrapper

Las clases Wrapper en Java son clases que envuelven tipos de datos primitivos y proporcionan métodos para trabajar con ellos. Las clases Wrapper también proporcionan métodos para validar datos.

Por ejemplo, para validar un número entero en Java, podemos utilizar la clase `Integer` y su método `parseInt()`. El método `parseInt()` convierte una cadena en un número entero y lanza una excepción si la cadena no es un número válido.

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Introduce un número entero: ");
        String input = scanner.nextLine();

        try {
            int number = Integer.parseInt(input);
            System.out.println("El número introducido es: " + number);
        } catch (NumberFormatException e) {
            System.out.println("Error: Introduce un número entero
válido.");
        }
    }
}
```

```
    }  
}  
}
```

Analicemos el código anterior:

- Se crea un objeto `Scanner` para leer la entrada del usuario.
- Se solicita al usuario que introduzca un número entero.
- Se lee la entrada del usuario como una cadena.
- Se intenta convertir la cadena en un número entero utilizando el método `parseInt()`.
- Si la conversión tiene éxito, se imprime el número introducido.
- Si la conversión falla (por ejemplo, si el usuario introduce una cadena que no es un número válido), se captura la excepción `NumberFormatException` y se imprime un mensaje de error.
- La validación de datos se realiza mediante el uso de excepciones para manejar los casos en los que los datos no son válidos.
- Este enfoque es útil para validar datos simples como números enteros, pero puede ser complicado para validar datos más complejos.
- En tales casos, se pueden utilizar otras técnicas de validación, como la validación de datos con la clase `JOptionPane`.
- La clase `JOptionPane` proporciona métodos para mostrar cuadros de diálogo que permiten al usuario introducir datos y validarlos.
- A continuación, veremos cómo validar datos con la clase `JOptionPane`.

Validación de datos con la clase `JOptionPane`

La clase `JOptionPane` en Java es una clase que proporciona métodos para mostrar cuadros de diálogo y obtener entrada del usuario. La clase `JOptionPane` también proporciona métodos para validar datos introducidos por el usuario.

Por ejemplo, para validar un número entero en Java utilizando la clase `JOptionPane`, podemos utilizar el método `showInputDialog()` para mostrar un cuadro de diálogo que solicite al usuario que introduzca un número entero. Luego, podemos utilizar el método `parseInt()` de la clase `Integer` para convertir la entrada del usuario en un número entero y validarla.

```
import javax.swing.JOptionPane;

public class Main {
    public static void main(String[] args) {
        String input = JOptionPane.showInputDialog("Introduce un número
entero:");

        try {
            int number = Integer.valueOf(input);
            JOptionPane.showMessageDialog(null, "El número introducido
es: " + number);
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(null, "Error: Introduce un
número entero válido.");
        }
    }
}
```

Analicemos el código anterior:

- Se utiliza el método `showInputDialog()` de la clase `JOptionPane` para mostrar un cuadro de diálogo que solicita al usuario que introduzca un número entero.
- Se lee la entrada del usuario como una cadena.
- Se intenta convertir la cadena en un número entero utilizando el método `valueOf()` de la clase `Integer`.
- Si la conversión tiene éxito, se muestra un cuadro de diálogo con el número introducido.
- Si la conversión falla (por ejemplo, si el usuario introduce una cadena que no es un número válido), se captura la excepción `NumberFormatException` y se muestra un

cuadro de diálogo con un mensaje de error.

- La validación de datos se realiza utilizando excepciones para manejar los casos en los que los datos no son válidos.

Cadenas con Formato en Java y Bloque de texto

Las cadenas con formato en Java son una forma de crear cadenas de texto con valores dinámicos. Esto permite combinar texto fijo con valores variables, como números, fechas o cualquier otro tipo de dato. Las cadenas con formato son útiles para generar mensajes personalizados, formatear la salida de datos o construir textos dinámicos.

La Clase función String.format()

En Java, la clase `String` proporciona un método estático llamado `format()` que permite crear cadenas con formato. La sintaxis de este método es la siguiente:

```
String resultado = String.format(formato, argumentos);
```

Donde:

- `formato` es una cadena que define el formato de la salida.
- `argumentos` son los valores que se insertarán en el formato.
 - Los argumentos pueden ser de cualquier tipo de dato y se insertan en el orden en que aparecen en la cadena de formato.
 - El formato de los argumentos se controla mediante especificadores de formato, que se indican con `%` seguido de un carácter que define el tipo de dato y el formato de presentación.
 - Los parámetros deben coincidir con los especificadores de formato en cantidad y tipo. De lo contrario, se producirá una excepción en tiempo de ejecución.
- `resultado` es la cadena con formato resultante.

Especificadores de Formato

Los especificadores de formato se utilizan para controlar la presentación de los valores en la cadena de salida. Algunos de los especificadores de formato más comunes son los

siguientes:

Especificador	Tipo de Dato	Descripción
%s	String	Cadena de texto
%d	int	Número entero
%f	float	Número decimal
%b	boolean	Valor booleano
%c	char	Carácter
%t	Date	Fecha y hora
%n	-	Salto de línea
%%	-	Carácter %

Modificadores de Formato

Además de los especificadores de formato, se pueden utilizar modificadores para controlar la presentación de los valores. Algunos de los modificadores más comunes son los siguientes:

Modificador	Descripción
%.nf	Número de decimales
%[n]s	Longitud mínima
%-[n]s	Alineación a la izquierda
%[0]nd	Enteros con ceros
%,[.nf]f	Separador de miles

Ejemplos de Cadenas con Formato

A continuación, se presentan algunos ejemplos de cadenas con formato en Java:

```
public class CadenasFormato {
    public static void main(String[] args) {
        String nombre = "Juan";
        int edad = 30;
        float altura = 1.75f;

        String mensaje = String.format("Hola, %s. Tienes %d años y mides
%.2f metros de altura.", nombre, edad, altura);
        System.out.println(mensaje);

        int numero = 12345;
        String numeroFormateado = String.format("El número es: %,d",
numero);
        System.out.println(numeroFormateado);

        float precio = 1234.56f;
        String precioFormateado = String.format("El precio es: $%,.2f",
precio);
        System.out.println(precioFormateado);
```

```
    }  
}
```

En este ejemplo, se utilizan diferentes especificadores y modificadores de formato para crear cadenas con valores dinámicos. El resultado de este programa sería:

```
Hola, Juan. Tienes 30 años y mides 1.75 metros de altura.  
El número es: 12,345  
El precio es: $1,234.56
```

Las cadenas con formato en Java son una herramienta poderosa para crear mensajes personalizados y formatear la salida de datos de manera clara y legible. Al utilizar especificadores y modificadores de formato, es posible controlar la presentación de los valores de forma precisa y detallada.

Los bloques de texto en Java

Los bloques de texto en Java son una característica introducida en Java 13 que permite definir cadenas de texto multilínea de forma más legible y concisa. Los bloques de texto se delimitan con tres comillas dobles ("") y pueden contener saltos de línea y comillas simples y dobles sin necesidad de escaparlas.

A continuación, se presenta un ejemplo de un bloque de texto en Java:

```
public class BloqueTexto {  
    public static void main(String[] args) {  
        String texto = """  
            Este es un bloque de texto  
            que contiene múltiples líneas  
            y comillas simples ' y dobles ".  
            """;  
        System.out.println(texto);  
    }  
}
```

En este ejemplo, se define un bloque de texto con tres líneas y se imprime en la consola. Los bloques de texto son útiles para escribir cadenas de texto largas y legibles, sin la necesidad de concatenar múltiples líneas o escapar caracteres especiales.

Los bloques de texto en Java son una característica conveniente que facilita la escritura y lectura de cadenas de texto multilínea. Al utilizar bloques de texto, es posible definir cadenas largas y complejas de forma más clara y concisa, mejorando la legibilidad y mantenibilidad del código.

Bloques de texto y cadenas con formato

Los bloques de texto en Java también se pueden combinar con cadenas con formato para crear mensajes personalizados y legibles. Al utilizar bloques de texto para definir la estructura del mensaje y cadenas con formato para insertar valores dinámicos, es posible generar mensajes complejos de forma clara y concisa.

A continuación, se presenta un ejemplo de un mensaje personalizado utilizando un bloque de texto y una cadena con formato en Java:

```
public class MensajePersonalizado {
    public static void main(String[] args) {
        String nombre = "María";
        int edad = 25;
        float altura = 1.65f;

        String mensaje = """
                        Hola, %s.
                        Te damos la bienvenida a nuestro sitio.
                        Esperamos que disfrutes tu estancia.

                        Detalles del perfil:
                        - Edad: %d años
                        - Altura: %.2f metros
                        """;

        String mensajePersonalizado = String.format(mensaje, nombre,
        edad, altura);
        System.out.println(mensajePersonalizado);
    }
}
```

En este ejemplo, se define un bloque de texto con la estructura del mensaje y se utiliza una cadena con formato para insertar los valores dinámicos. Al combinar bloques de texto y cadenas con formato, es posible crear mensajes personalizados de forma clara y legible, mejorando la experiencia del usuario y la mantenibilidad del código.

- ⚠** Los bloques de texto y las cadenas con formato son herramientas poderosas para generar mensajes personalizados y formatear la salida de datos en Java. Al utilizar estas características de forma combinada, es posible crear mensajes complejos y detallados de manera clara y concisa, mejorando la legibilidad y mantenibilidad del código.

- ⚠** Los bloques de texto con formato en Java empezaron a ser soportados a partir de la versión 13 de Java. Si estás utilizando una versión anterior, es posible que esta característica no esté disponible. Asegúrate de utilizar una versión compatible de Java para aprovechar al máximo las funcionalidades más recientes.

Conclusión

Las cadenas con formato en Java son una forma poderosa de crear mensajes personalizados y formatear la salida de datos de manera clara y legible. Al utilizar especificadores y modificadores de formato, es posible controlar la presentación de los valores de forma precisa y detallada, mejorando la experiencia del usuario y la mantenibilidad del código.

Menús de Interacción con JOptionPane

En Java, la clase `JOptionPane` proporciona un formato de su función `showInputDialog()` que permite mostrar un menú de opciones al usuario. Este método recibe 7 parámetros, de los cuales 5 son obligatorios y 2 son opcionales. A continuación, se muestra la sintaxis de este método:

```
String[] opciones = { "Opción 1", "Opción 2", "Opción 3" };
String opcionSeleccionada = (String) JOptionPane.showInputDialog(
    null,
    "Selecciona una opción:",
    "Menú de Opciones",
    JOptionPane.QUESTION_MESSAGE,
    null,
    opciones,
    opciones[0]
);
```

Donde:

- `null`: Es el componente padre del diálogo. Si se pasa `null`, el diálogo se centrará en la pantalla.
- `"Selecciona una opción:"`: Es el mensaje que se muestra en el diálogo.
- `"Menú de Opciones"`: Es el título del diálogo.
- `JOptionPane.QUESTION_MESSAGE`: Es el tipo de mensaje que se muestra en el diálogo.
- `null`: Es el ícono que se muestra en el diálogo. Si se pasa `null`, se muestra el ícono predeterminado.
- `opciones`: Es un arreglo de cadenas que contiene las opciones del menú.
- `opciones[0]`: Es la opción predeterminada seleccionada en el menú.

En este ejemplo, se muestra un menú de opciones con las opciones "Opción 1", "Opción 2" y "Opción 3". El usuario puede seleccionar una de las opciones y el valor seleccionado se guarda en la variable opcionSeleccionada.

Es importante tener en cuenta que el valor devuelto por showInputDialog() es de tipo Object, por lo que es necesario convertirlo al tipo de dato deseado. En este caso, se convierte a String ya que las opciones del menú son cadenas.

A continuación, se presenta un ejemplo completo de cómo mostrar un menú de opciones utilizando JOptionPane:

```
import javax.swing.JOptionPane;

public class MenuOpciones {
    public static void main(String[] args) {
        String[] opciones = { "Opción 1", "Opción 2", "Opción 3" };
        String opcionSeleccionada = (String)
JOptionPane.showInputDialog(
            null,
            "Selecciona una opción
            :",
            "Menú de Opciones",
            JOptionPane.QUESTION_MESSAGE,
            null,
            opciones,
            opciones[0]
        );

        if (opcionSeleccionada != null) {
            JOptionPane.showMessageDialog(null, "Opción seleccionada: "
+ opcionSeleccionada);
        } else {
            JOptionPane.showMessageDialog(null, "No se seleccionó
ninguna opción.");
        }
    }
}
```

En este ejemplo, se muestra un menú de opciones con las opciones "Opción 1", "Opción 2" y "Opción 3". Si el usuario selecciona una opción, se muestra un mensaje con la opción seleccionada; de lo contrario, se muestra un mensaje indicando que no se seleccionó ninguna opción.

Los menús de opciones son útiles para presentar al usuario una lista de opciones para elegir, lo que facilita la interacción con la aplicación y mejora la experiencia del usuario.

Ventana de Confirmación

Además de los menús de opciones, JOptionPane también proporciona un método para mostrar una ventana de confirmación al usuario. Este método recibe 4 parámetros, de los cuales 3 son obligatorios y 1 es opcional. A continuación, se muestra la sintaxis de este método:

```
int confirmacion = JOptionPane.showConfirmDialog(  
    null,  
    "¿Estás seguro de continuar?",  
    "Confirmación",  
    JOptionPane.YES_NO_OPTION  
);
```

Donde:

- `null`: Es el componente padre del diálogo. Si se pasa `null`, el diálogo se centrará en la pantalla.
- `"¿Estás seguro de continuar?"`: Es el mensaje que se muestra en la ventana de confirmación.
- `"Confirmación"`: Es el título de la ventana de confirmación.
- `JOptionPane.YES_NO_OPTION`: Es el tipo de opciones que se muestran en la ventana.
- `confirmacion`: Es el valor entero que representa la opción seleccionada por el usuario.

En este ejemplo, se muestra una ventana de confirmación con el mensaje `"¿Estás seguro de continuar?"` y dos opciones: `Sí` y `No`. El valor devuelto por `showConfirmDialog()` representa la opción seleccionada por el usuario:

- `JOptionPane.YES_OPTION` si el usuario selecciona Sí.
- `JOptionPane.NO_OPTION` si el usuario selecciona No.
- `JOptionPane.CLOSED_OPTION` si el usuario cierra la ventana sin seleccionar ninguna opción.
- `JOptionPane.CANCEL_OPTION` si el usuario cancela la operación.
- `JOptionPane.OK_OPTION` si el usuario selecciona Aceptar.
- `JOptionPane.CANCEL_OPTION` si el usuario selecciona Cancelar.

Este valor puede ser utilizado para realizar diferentes acciones en función de la respuesta del usuario.

Las alternativas de `showConfirmDialog` son:

Constante	Descripción
<code>YES_NO_OPTION</code>	Muestra los botones Sí y No.
<code>YES_NO_CANCEL_OPTION</code>	Muestra los botones Sí, No y Cancelar.
<code>OK_CANCEL_OPTION</code>	Muestra los botones Aceptar y Cancelar.

A continuación se muestra un ejemplo completo que muestra una ventana de confirmación al usuario y realiza una acción en función de la respuesta:

```
import javax.swing.JOptionPane;

public class VentanaConfirmacion {
    public static void main(String[] args) {
        int confirmacion = JOptionPane.showConfirmDialog(
            null,
            "¿Estás seguro de continuar?",
            "Confirmación",
            JOptionPane.YES_NO_OPTION
        );
    }
}
```

```
        if (confirmacion == JOptionPane.YES_OPTION) {
            JOptionPane.showMessageDialog(null, "Operación
confirmada.");
        } else {
            JOptionPane.showMessageDialog(null, "Operación cancelada.");
        }
    }
}
```

En este ejemplo, se muestra una ventana de confirmación al usuario y se muestra un mensaje diferente en función de la respuesta seleccionada por el usuario.

La ventana de confirmación es una herramienta útil para solicitar la confirmación del usuario antes de realizar una acción importante en una aplicación de escritorio en Java.

Tipos Enumerados

Un tipo enumerado o `enum` es un tipo de dato especial que permite definir un conjunto fijo de constantes con nombre. Cada constante en un tipo enumerado representa un valor específico y se puede utilizar en lugar de valores numéricos o cadenas.

En Java, los tipos enumerados se definen utilizando la palabra clave `enum` seguida de una lista de constantes separadas por comas. A continuación, se presenta un ejemplo de un tipo enumerado en Java:

```
public enum DiaSemana {  
    LUNES,  
    MARTES,  
    MIÉRCOLES,  
    JUEVES,  
    VIERNES,  
    SÁBADO,  
    DOMINGO  
}
```

En este ejemplo, se define un tipo enumerado `DiaSemana` que representa los días de la semana. Cada constante en el tipo enumerado es un día de la semana y se puede utilizar para representar valores específicos en el código.

Los tipos enumerados en Java son útiles para representar conjuntos de constantes relacionadas, como los días de la semana, los meses del año, los colores, etc. Al utilizar tipos enumerados, se mejora la legibilidad y mantenibilidad del código, ya que se evita el uso de valores mágicos y se proporciona un conjunto fijo de opciones válidas.

Tipos Enumerados con atributos

Además de las constantes, los tipos enumerados en Java también pueden tener atributos asociados a cada constante. Estos atributos pueden ser de cualquier tipo de dato y se pueden utilizar para almacenar información adicional sobre cada constante.

A continuación, se presenta un ejemplo de un tipo enumerado con atributos en Java:

```

public enum Mes {
    ENERO(1),
    FEBRERO(2),
    MARZO(3),
    ABRIL(4),
    MAYO(5),
    JUNIO(6),
    JULIO(7),
    AGOSTO(8),
    SEPTIEMBRE(9),
    OCTUBRE(10),
    NOVIEMBRE(11),
    DICIEMBRE(12);

    private final int numero;

    Mes(int numero) {
        this.numero = numero;
    }

    public int getNumero() {
        return numero;
    }
}

```

En este ejemplo, se define un tipo enumerado `Mes` que representa los meses del año. Cada constante en el tipo enumerado tiene un atributo `numero` que almacena el número correspondiente al mes. Además, se define un constructor y un método `getNumero()` para acceder al valor del atributo.

Los tipos enumerados con atributos son útiles cuando se necesita asociar información adicional a cada constante, como números, cadenas, valores predeterminados, etc. Estos atributos pueden mejorar la flexibilidad y funcionalidad de los tipos enumerados, permitiendo un mayor control sobre las constantes definidas.

La función `toString()` en tipos enumerados

En Java, los tipos enumerados tienen una implementación predeterminada del método `toString()` que devuelve el nombre de la constante en forma de cadena. Este método se puede utilizar para obtener una representación legible de una constante enumerada.

A continuación, se presenta un ejemplo de cómo utilizar el método `toString()` en un tipo enumerado en Java:

```
public class EjemploEnum {  
    public static void main(String[] args) {  
        DiaSemana dia = DiaSemana.LUNES;  
        System.out.println("Día de la semana: " + dia.toString());  
    }  
}
```

En este ejemplo, se crea una variable `dia` de tipo `DiaSemana` con el valor `LUNES` y se imprime el nombre del día de la semana utilizando el método `toString()`. La salida de este programa sería:

```
Día de la semana: LUNES
```

El método `toString()` en tipos enumerados es útil para obtener una representación legible de las constantes definidas en el tipo enumerado. Al utilizar este método, se mejora la legibilidad y claridad del código, ya que se evita la necesidad de acceder directamente al nombre de la constante.

Por consiguiente, podemos redefinir el método `toString()` en un tipo enumerado para proporcionar una representación personalizada de las constantes. A continuación, se presenta un ejemplo de cómo redefinir el método `toString()` en un tipo enumerado en Java:

```
public enum Mes {  
    ENERO(1),  
    FEBRERO(2),  
    MARZO(3),  
    ABRIL(4),  
    MAYO(5),  
    JUNIO(6),  
    JULIO(7),
```

```

AGOSTO(8),
SEPTIEMBRE(9),
OCTUBRE(10),
NOVIEMBRE(11),
DICIEMBRE(12);

private final int numero;

Mes(int numero) {
    this.numero = numero;
}

public int getNumero() {
    return numero;
}

@Override
public String toString() {
    return switch (this) {
        case ENERO -> "Ene";
        case FEBRERO -> "Feb";
        case MARZO -> "Mar";
        case ABRIL -> "Abr";
        case MAYO -> "May";
        case JUNIO -> "Jun";
        case JULIO -> "Jul";
        case AGOSTO -> "Ago";
        case SEPTIEMBRE -> "Sep";
        case OCTUBRE -> "Oct";
        case NOVIEMBRE -> "Nov";
        case DICIEMBRE -> "Dic";
    };
}
}

```

En este ejemplo, se redefine el método `toString()` en el tipo enumerado `Mes` para devolver una representación abreviada de los nombres de los meses. Al utilizar una

expresión `switch` en el método `toString()`, se puede personalizar la representación de cada constante en el tipo enumerado.

Al redefinir el método `toString()` en un tipo enumerado, se puede proporcionar una representación personalizada de las constantes, lo que mejora la legibilidad y claridad del código. Esta técnica es útil cuando se necesita mostrar las constantes enumeradas de una manera específica en la interfaz de usuario o en la salida del programa.

Conclusión

Los tipos enumerados en Java son una forma conveniente de definir un conjunto fijo de constantes con nombre. Al utilizar tipos enumerados, se mejora la legibilidad y mantenibilidad del código, ya que se evita el uso de valores mágicos y se proporciona un conjunto fijo de opciones válidas. Además, los tipos enumerados con atributos y métodos personalizados pueden mejorar la funcionalidad y flexibilidad de las constantes definidas.

Usando `enum` con ` JOptionPane`

En Java, la clase `JOptionPane` proporciona métodos estáticos para mostrar ventanas de diálogo con mensajes, opciones y entradas de usuario. Una forma de mejorar la legibilidad y mantenibilidad de un programa que utiliza `JOptionPane` es mediante el uso de enumeraciones (`enum`) para definir las opciones disponibles en los diálogos.

En este tutorial, se mostrará cómo utilizar un tipo enumerado (`enum`) en combinación con la clase `JOptionPane` para definir y mostrar un menú de opciones al usuario.

Definición de un tipo enumerado

Primero, se define un tipo enumerado llamado `OpcionMenu` que contiene las opciones

```
public enum OpcionMenu {  
    OPCION_1("Opción 1"),  
    OPCION_2("Opción 2"),  
    OPCION_3("Opción 3");  
  
    private final String descripcion;  
  
    OpcionMenu(String descripcion) {  
        this.descripcion = descripcion;  
    }  
  
    public String getDescripcion() {  
        return descripcion;  
    }  
  
    @Override  
    public String toString() {  
        return String.format("%d. %s", ordinal() + 1, descripcion);  
    }  
}
```

En este ejemplo, el tipo enumerado `OpcionMenu` define tres constantes con sus respectivas descripciones. Cada constante tiene un atributo `descripcion` que almacena

el texto asociado a la opción. Además, se redefine el método `toString()` para que devuelva un formato numérico de la opción y su descripción.

En la función `toString()`, se utiliza el método `ordinal()` para obtener el índice de la constante en la enumeración y se suma 1 para mostrar un índice basado en 1 en lugar de 0.

Mostrar un menú de opciones

Una vez definido el tipo enumerado `OpcionMenu`, se puede utilizar en conjunto con la clase `JOptionPane` para mostrar un menú de opciones al usuario. A continuación, se presenta un ejemplo de cómo mostrar un menú de opciones utilizando `JOptionPane` y el tipo enumerado `OpcionMenu`:

```
import javax.swing.JOptionPane;

public class MenuOpciones {
    public static void main(String[] args) {
        OpcionMenu[] opciones = OpcionMenu.values();
        OpcionMenu opcionSeleccionada = (OpcionMenu)
JOptionPane.showInputDialog(
            null,
            "Selecciona una opción:",
            "Menú de Opciones",
            JOptionPane.QUESTION_MESSAGE,
            null,
            opciones,
            opciones[0]
        );

        if (opcionSeleccionada != null) {
            JOptionPane.showMessageDialog(null, "Opción seleccionada: "
+ opcionSeleccionada.getDescripcion());
        } else {
            JOptionPane.showMessageDialog(null, "No se seleccionó
ninguna opción.");
        }
    }
}
```

```
    }  
}
```

En este ejemplo, se obtienen todas las constantes del tipo enumerado `OpcionMenu` mediante el método `values()`. Luego, se muestra un cuadro de diálogo con un menú de opciones que permite al usuario seleccionar una de las opciones. El valor seleccionado se guarda en la variable `opcionSeleccionada`, la cual se utiliza para mostrar la descripción de la opción seleccionada en un mensaje de diálogo.

Al utilizar un tipo enumerado para definir las opciones del menú, se mejora la legibilidad y mantenibilidad del código, ya que se centraliza la definición de las opciones en un solo lugar y se evita el uso de cadenas literales dispersas en el código.

Ahora bien, podemos usar un `switch` dentro del `if` para realizar acciones específicas según la opción seleccionada:

```
import javax.swing.JOptionPane;  
  
public class MenuOpciones {  
    public static void main(String[] args) {  
        OpcionMenu[] opciones = OpcionMenu.values();  
        OpcionMenu opcionSeleccionada = (OpcionMenu)  
JOptionPane.showInputDialog(  
            null,  
            "Selecciona una opción:",  
            "Menú de Opciones",  
            JOptionPane.QUESTION_MESSAGE,  
            null,  
            opciones,  
            opciones[0]  
        );  
  
        if (opcionSeleccionada != null) {  
            switch (opcionSeleccionada) {  
                case OPCION_1:  
                    JOptionPane.showMessageDialog(null, "Has  
seleccionado la Opción 1.");  
                    break;  
            }  
        }  
    }  
}
```

```
        case OPCION_2:  
            JOptionPane.showMessageDialog(null, "Has  
seleccionado la Opción 2.");  
            break;  
        case OPCION_3:  
            JOptionPane.showMessageDialog(null, "Has  
seleccionado la Opción 3.");  
            break;  
        default:  
            JOptionPane.showMessageDialog(null, "Opción no  
válida.");  
            break;  
    }  
} else {  
    JOptionPane.showMessageDialog(null, "No se seleccionó  
ninguna opción.");  
}  
}  
}
```

En este caso, se utiliza un `switch` para realizar acciones específicas según la opción seleccionada por el usuario. Cada `case` corresponde a una de las constantes del tipo enumerado `OpcionMenu`, y se muestra un mensaje diferente según la opción seleccionada. En el `default`, se maneja el caso en que se seleccione una opción no válida. Al utilizar un `switch` con un tipo enumerado, se mejora la claridad y mantenibilidad del código, ya que se evita el uso de múltiples comparaciones con cadenas literales y se centraliza la lógica de selección de opciones en un solo lugar.

El Modificador de Acceso Static

El modificador de acceso static se utiliza para declarar miembros de una clase que pertenecen a la clase en sí, y no a las instancias de la clase. Esto significa que los miembros estáticos son compartidos por todas las instancias de la clase.

Declaración de miembros estáticos

Para declarar un miembro estático, se utiliza la palabra clave static seguida del tipo de dato y el nombre del miembro.

```
public class MiClase {  
    public static int miVariable = 10;  
    public static void miMetodo() {  
        System.out.println("Este es un método estático");  
    }  
}
```

En el ejemplo anterior, miVariable y miMetodo son miembros estáticos de la clase MiClase. Esto significa que miVariable y miMetodo son compartidos por todas las instancias de MiClase.

Acceso a miembros estáticos

Los miembros estáticos pueden ser accedidos directamente a través de la clase, sin necesidad de crear una instancia de la clase.

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println(MiClase.miVariable);  
        MiClase.miMetodo();  
    }  
}
```

En el ejemplo anterior, se accede a miVariable y miMetodo directamente a través de la clase MiClase.

Ventajas de los miembros estáticos

- **Compartir información:** Los miembros estáticos son compartidos por todas las instancias de la clase, lo que permite compartir información entre las instancias.
- **Acceso directo:** Los miembros estáticos pueden ser accedidos directamente a través de la clase, sin necesidad de crear una instancia de la clase.
- **Eficiencia:** Los miembros estáticos se almacenan en un solo lugar en la memoria, lo que puede mejorar la eficiencia de la aplicación.
- **Constantes:** Los miembros estáticos se pueden utilizar para declarar constantes en una clase.
- **Métodos de utilidad:** Los métodos estáticos se pueden utilizar para implementar funciones de utilidad que no dependen del estado de un objeto.
- **Clases de utilidad:** Las clases de utilidad, que contienen solo miembros estáticos, se pueden utilizar para agrupar funciones relacionadas.
- **Patrones de diseño:** Los miembros estáticos se pueden utilizar para implementar patrones de diseño como el patrón Singleton.

Desventajas de los miembros estáticos

- **Acoplamiento:** El uso excesivo de miembros estáticos puede aumentar el acoplamiento entre las clases, lo que puede dificultar la modificación y reutilización del código.
- **Pruebas unitarias:** Los miembros estáticos pueden dificultar las pruebas unitarias, ya que pueden tener efectos secundarios inesperados en otras partes del código.
- **Estado compartido:** Los miembros estáticos pueden introducir problemas de concurrencia si se comparte estado entre múltiples hilos de ejecución.
- **Dificultad para rastrear dependencias:** Los miembros estáticos pueden dificultar el rastreo de dependencias entre las clases, lo que puede dificultar la comprensión y el mantenimiento del código.

- **Dificultad para reemplazar la implementación:** Los miembros estáticos pueden dificultar la reemplazo de la implementación de un miembro con una implementación diferente, ya que todos los clientes de la clase dependen de la implementación estática.
- **Dificultad para probar el código:** Los miembros estáticos pueden dificultar la prueba del código, ya que pueden tener efectos secundarios inesperados en otras partes del código.

Conclusión

El modificador de acceso `static` se utiliza para declarar miembros de una clase que pertenecen a la clase en sí, y no a las instancias de la clase. Los miembros estáticos son compartidos por todas las instancias de la clase, lo que permite compartir información entre las instancias. Sin embargo, el uso excesivo de miembros estáticos puede aumentar el acoplamiento entre las clases y dificultar la modificación y reutilización del código. Por lo tanto, es importante utilizar los miembros estáticos con moderación y tener en cuenta las ventajas y desventajas de su uso.

El modificador de acceso final

El modificador de acceso `final` se utiliza para declarar que un miembro de una clase no puede ser modificado una vez que ha sido inicializado. Esto significa que una vez que un miembro ha sido asignado un valor, no se puede cambiar. El modificador `final` se puede aplicar a variables, métodos y clases.

Variables finales

Para declarar una variable final, se utiliza la palabra clave `final` seguida del tipo de dato y el nombre de la variable.

```
public class MiClase {  
    public final int miVariable = 10;  
}
```

En el ejemplo anterior, `miVariable` es una variable final de la clase `MiClase`. Esto significa que una vez que `miVariable` ha sido asignada un valor, no se puede cambiar. Si se intenta modificar `miVariable`, se producirá un error en tiempo de compilación.

Así mismo, si deseamos que una variable sea final en términos de la clase, se debe declarar como `static` y `final`.

```
public class MiClase {  
    public static final int MI_CONSTANTE = 100;  
}
```

En el ejemplo anterior, `MI_CONSTANTE` es una constante de la clase `MiClase`. Esto significa que `MI_CONSTANTE` es compartida por todas las instancias de `MiClase` y no puede ser modificada.

Por otro lado, si deseamos que una variable sea final en términos de un método, se debe declarar como `final`.

```
public class MiClase {  
    public void miMetodo() {  
        final int miVariable = 10;
```

```
    }  
}
```

En el ejemplo anterior, `miVariable` es una variable final del método `miMetodo`. Esto significa que una vez que `miVariable` ha sido asignada un valor, no se puede cambiar.

Métodos finales

Para declarar un método final, se utiliza la palabra clave `final` antes de la palabra clave `void` en la declaración del método.

```
public class MiClase {  
    public final void miMetodo() {  
        System.out.println("Este es un método final");  
    }  
}
```

En el ejemplo anterior, `miMetodo` es un método final de la clase `MiClase`. Esto significa que `miMetodo` no puede ser sobrescrito por las subclases de `MiClase`.

Clases finales

Para declarar una clase final, se utiliza la palabra clave `final` antes de la palabra clave `class` en la declaración de la clase.

```
public final class MiClase {  
    // Código de la clase  
}
```

En el ejemplo anterior, `MiClase` es una clase final. Esto significa que `MiClase` no puede ser extendida por otras clases.

Ventajas de los miembros finales

- **Inmutabilidad:** Los miembros finales son inmutables, lo que puede ayudar a prevenir errores y mejorar la legibilidad del código.

- **Seguridad:** Los miembros finales pueden ayudar a prevenir la modificación accidental de valores críticos en la aplicación.
- **Optimización:** Los miembros finales pueden ser optimizados por el compilador, lo que puede mejorar el rendimiento de la aplicación.
- **Claridad:** El uso de miembros finales puede hacer que el código sea más claro y fácil de entender, ya que indica que un valor no cambiará.
- **Seguridad en subclases:** Al declarar un método o clase como final, se evita que las subclases lo modifiquen o extiendan, lo que puede ayudar a mantener la integridad del diseño de la clase.
- **Constantes:** Los miembros finales se pueden utilizar para declarar constantes en una clase.

Desventajas de los miembros finales

- **Flexibilidad:** Al declarar un miembro como final, se pierde la flexibilidad de modificar su valor en tiempo de ejecución, lo que puede ser necesario en ciertos casos.
- **Dificultad de prueba:** Los miembros finales pueden dificultar las pruebas unitarias, ya que pueden tener efectos secundarios inesperados en otras partes del código.
- **Complejidad:** El uso excesivo de miembros finales puede aumentar la complejidad del código, lo que puede dificultar su mantenimiento y comprensión.
- **Rendimiento:** En algunos casos, el uso de miembros finales puede afectar el rendimiento de la aplicación, ya que el compilador puede tener que realizar optimizaciones adicionales.
- **Acoplamiento:** El uso excesivo de miembros finales puede aumentar el acoplamiento entre las clases, lo que puede dificultar la modificación y reutilización del código.
- **Dificultad para rastrear dependencias:** Los miembros finales pueden dificultar el rastreo de dependencias entre las clases, lo que puede dificultar la comprensión y el mantenimiento del código.

Conclusión

El modificador de acceso `final` es una herramienta poderosa que puede ayudar a mejorar la seguridad, la claridad y el rendimiento del código. Sin embargo, su uso debe ser cuidadoso y equilibrado, ya que puede tener efectos secundarios indeseados si se aplica de manera indiscriminada. Al utilizar el modificador `final`, es importante considerar las ventajas y desventajas de su uso en cada caso particular y tomar decisiones informadas sobre su aplicación.

¿Qué es una función en Java?

Una función en Java es un bloque de código que realiza una tarea específica. Las funciones se utilizan para dividir un programa en partes más pequeñas y manejables, lo que facilita la lectura, la depuración y la reutilización del código.

Declaración de funciones

Para declarar una función en Java, se utiliza la palabra clave `public` seguida del tipo de dato de retorno de la función, el nombre de la función y los parámetros de la función entre paréntesis.

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

En el ejemplo anterior, `sumar` es una función de la clase `MiClase` que recibe dos parámetros `a` y `b` de tipo entero y devuelve la suma de los dos parámetros.

Llamada a funciones

Para llamar a una función en Java, se utiliza el nombre de la función seguido de los parámetros de la función entre paréntesis.

```
public class Main {  
    public static void main(String[] args) {  
        MiClase miObjeto = new MiClase();  
        int resultado = miObjeto.sumar(5, 3);  
        System.out.println(resultado);  
    }  
}
```

En el ejemplo anterior, se crea una instancia de la clase `MiClase` llamada `miObjeto` y se llama a la función `sumar` con los parámetros `5` y `3`. El resultado de la función se

almacena en la variable resultado y se imprime en la consola.

Funciones estáticas

Las funciones estáticas en Java se declaran con la palabra clave static antes del tipo de dato de retorno de la función. Las funciones estáticas pertenecen a la clase en sí y no a las instancias de la clase.

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

En el ejemplo anterior, multiplicar es una función estática de la clase MiClase que recibe dos parámetros a y b de tipo entero y devuelve el producto de los dos parámetros.

Llamada a funciones estáticas

Para llamar a una función estática en Java, se utiliza el nombre de la clase seguido del operador de punto (.) y el nombre de la función.

```
public class Main {  
    public static void main(String[] args) {  
        int resultado = MiClase.multiplicar(5, 3);  
        System.out.println(resultado);  
    }  
}
```

En el ejemplo anterior, se llama a la función estática multiplicar de la clase MiClase con los parámetros 5 y 3. El resultado de la función se almacena en la variable resultado y se imprime en la consola.

Ventajas de las funciones

- **Reutilización de código:** Las funciones permiten reutilizar el código en diferentes

partes de un programa.

- **Abstracción:** Las funciones permiten abstraer la lógica de un programa en bloques reutilizables.
- **Facilidad de mantenimiento:** Las funciones facilitan la lectura, la depuración y el mantenimiento del código.
- **División de tareas:** Las funciones permiten dividir un programa en tareas más pequeñas y manejables.
- **Claridad:** Las funciones mejoran la claridad y la organización del código.
- **Modularidad:** Las funciones promueven la modularidad del código al dividirlo en partes independientes y reutilizables.
- **Facilidad de pruebas:** Las funciones facilitan las pruebas unitarias al permitir probar partes específicas del código.

Desventajas de las funciones

- **Acoplamiento:** El uso excesivo de funciones puede aumentar el acoplamiento entre las clases, lo que puede dificultar la modificación y reutilización del código.
- **Complejidad:** El uso excesivo de funciones puede aumentar la complejidad del código, lo que puede dificultar su mantenimiento y comprensión.
- **Rendimiento:** En algunos casos, el uso excesivo de funciones puede afectar el rendimiento de la aplicación, ya que el compilador puede tener que realizar optimizaciones adicionales.
- **Seguridad:** Las funciones pueden introducir vulnerabilidades de seguridad si no se implementan correctamente.

Conclusión

Las funciones son una parte fundamental de la programación en Java y son una herramienta poderosa para dividir, organizar y reutilizar el código. Al utilizar funciones de

manera efectiva, los programadores pueden mejorar la legibilidad, la mantenibilidad y la eficiencia de sus programas.

La Recursividad en la programación

La recursividad es un concepto fundamental en la programación que se refiere a la capacidad de una función de llamarse a sí misma. En este artículo, exploraremos qué es la recursividad, cómo funciona y cómo se implementa en Java.

¿Qué es la Recursividad?

La recursividad es un concepto en programación en el que una función se llama a sí misma para resolver un problema. En otras palabras, una función recursiva es aquella que se define en términos de sí misma. La recursividad se basa en la idea de dividir un problema en subproblemas más pequeños y resolverlos de forma recursiva.

Características de la Recursividad

La recursividad tiene las siguientes características:

- **Llamada a sí misma:** Una función recursiva se llama a sí misma dentro de su definición.
- **Caso Base:** Una función recursiva debe tener un caso base que detenga la recursión.
- **División del Problema:** La recursividad se basa en la idea de dividir un problema en subproblemas más pequeños y resolverlos de forma recursiva.
- **Pila de Llamadas:** Cada llamada recursiva se agrega a la pila de llamadas, lo que puede llevar a un desbordamiento de pila si no se maneja correctamente.
- **Eficiencia:** La recursividad puede ser menos eficiente que las soluciones iterativas en algunos casos, ya que implica múltiples llamadas a la función.
- **Legibilidad:** La recursividad puede hacer que el código sea más legible y conciso en comparación con las soluciones iterativas.
- **Problemas Recursivos:** Algunos problemas son naturalmente recursivos y se pueden resolver de forma más sencilla con la recursividad.

- **Problemas Iterativos:** No todos los problemas son adecuados para la recursividad, y en algunos casos, las soluciones iterativas pueden ser más apropiadas.
- **Memoria:** La recursividad puede consumir más memoria que las soluciones iterativas, ya que cada llamada recursiva agrega una nueva entrada en la pila de llamadas.
- **Profundidad de Recursión:** La profundidad de recursión es el número máximo de llamadas recursivas que se pueden realizar antes de que se produzca un desbordamiento de pila.

¿Cómo saber si un problema es recursivo?, o, ¿Cuándo usar la recursividad?

La recursividad es una técnica poderosa que se puede utilizar para resolver una amplia variedad de problemas. Algunas señales de que un problema puede ser resuelto de forma recursiva son:

- El problema se puede dividir en subproblemas más pequeños que son similares al problema original.
- El problema se puede resolver de forma más sencilla si se resuelve un caso base y luego se aplica la recursión a los subproblemas restantes.
- El problema debe de tener un caso limitante que detenga la recursión ya que de lo contrario se producirá un desbordamiento de pila.

En general, la recursividad es útil cuando un problema se puede dividir en subproblemas más pequeños y similares al problema original. Sin embargo, no todos los problemas son adecuados para la recursividad, y en algunos casos, las soluciones iterativas pueden ser más apropiadas.

Ejemplo de Recursividad en Java

A continuación, se muestra un ejemplo de una función recursiva en Java que calcula la potencia de un número:

```
public class Recursividad {
```

```

public static void main(String[] args) {

    int base = 2;
    int exponente = 3;
    int resultado = potencia(base, exponente);
    System.out.println(base + " elevado a la " + exponente + " es
igual a " + resultado);
}

public static int potencia(int base, int exponente) {

    if (exponente == 0) {

        return 1;
    } else {
        return base * potencia(base, exponente - 1);
    }
}

```

Analicemos el código anterior:

- La función `potencia` es una función recursiva que calcula la potencia de un número `base` elevado a un número `exponente`.
- La función tiene un caso base que detiene la recursión cuando el exponente es igual a cero.
- Si el exponente es mayor que cero, la función se llama a sí misma con un exponente menor y multiplica el resultado por la base. Y este proceso se repite hasta que el exponente llega a cero.
- Finalmente, el resultado se devuelve como el resultado de la potencia.
- En el método `main`, se llama a la función `potencia` con una base de 2 y un exponente de 3, y se imprime el resultado en la consola.
- La salida del programa será 2 elevado a la 3 es igual a 8.

Este es un ejemplo sencillo de cómo se puede utilizar la recursividad para resolver un problema en Java. La recursividad es una técnica poderosa que se puede utilizar para resolver una amplia variedad de problemas de forma elegante y concisa.

Conclusión

La recursividad es un concepto fundamental en la programación que se basa en la idea de que una función puede llamarse a sí misma para resolver un problema. La recursividad se utiliza para dividir un problema en subproblemas más pequeños y resolverlos de forma recursiva. Si se utiliza correctamente, la recursividad puede hacer que el código sea más legible y conciso, y puede ser una herramienta poderosa para resolver una amplia variedad de problemas.

Los Registros en Java

Los registros o `record` son una nueva característica introducida en Java 14 que permite definir clases de datos de forma concisa y eficiente. Los `record` son una forma de definir una clase que encapsula un conjunto de campos y proporciona métodos de acceso a esos campos de forma automática.

Declaración de un Registro

Para declarar un registro, se utiliza la palabra clave `record` seguida del nombre del registro y una lista de campos separados por comas.

```
public record NombreDelRegistro(TipoDeDatos nombreDelCampo1, TipoDeDatos  
nombreDelCampo2, ...) {  
    // Cuerpo del registro  
}
```

Dónde:

- `NombreDelRegistro`: Es el nombre del registro.
- `TipoDeDatos`: Es el tipo de dato de los campos del registro.
- `nombreDelCampo1`, `nombreDelCampo2`, etc.: Son los nombres de los campos del registro.
- `Cuerpo del registro`: Es opcional y puede contener métodos y constructores adicionales.

Características de los Registros

Los registros tienen las siguientes características:

- **Campos Inmutables**: Los campos de un registro son inmutables, lo que significa que no se pueden modificar una vez que se han inicializado.
 - Los campos de un registro son finales, lo que significa que no se pueden reasignar después de la inicialización.

- Los campos de un registro son privados, lo que significa que no se pueden acceder directamente desde fuera del registro.
 - Los campos de un registro se inicializan automáticamente a través del constructor del registro.
 - Los campos de un registro se pueden acceder a través de métodos de acceso generados automáticamente.
- **Métodos de Acceso:** Los registros proporcionan métodos de acceso para cada campo, lo que permite acceder a los campos de forma segura.
- Los métodos de acceso siguen la convención de nombres `getNombreDelCampo`. A excepción de los campos `boolean`, que siguen la convención de nombres `isNombreDelCampo`.
 - Los métodos de acceso son generados automáticamente por el compilador.
- **Método `toString`:** Los registros proporcionan un método `toString` que genera una representación de cadena del registro.
- Esta función es útil para imprimir el registro en la consola o en un archivo.
 - La representación de cadena generada incluye el nombre de la clase y los valores de los campos del registro.
 - Un ejemplo de representación de cadena generada por un registro es `NombreDelRegistro[nombreDelCampo1=valor1, nombreDelCampo2=valor2, ...]`.
 - Puedes sobre escribir el método `toString` para personalizar la representación de cadena del registro.
- **Método `equals`:** Los registros proporcionan un método `equals` que compara dos registros por igualdad estructural.
- Dos registros son iguales si son del mismo tipo y tienen los mismos valores de campo.
- **Método `hashCode`:** Los registros proporcionan un método `hashCode` que genera un código hash basado en los campos del registro.

- **Constructor Implícito:** Los registros proporcionan un constructor implícito que inicializa los campos del registro.
 - Puedes proporcionar constructores adicionales para inicializar los campos del registro de forma personalizada.

Ejemplos de Registros

Punto en un Plano Cartesiano

A continuación se muestra un ejemplo de un registro que representa un punto en un plano cartesiano:

```
public record Punto(int x, int y) {  
    public Punto {  
        if (x < 0 || y < 0) {  
            throw new IllegalArgumentException("Los valores de x e y  
deben ser mayores o iguales a cero");  
        }  
    }  
}
```

En el ejemplo anterior, se declara un registro `Punto` con dos campos `x` e `y`. Se proporciona un constructor adicional que valida que los valores de `x` e `y` sean mayores o iguales a cero.

Persona

A continuación se muestra un ejemplo de un registro que representa una persona:

```
public record Persona(String nombre, int edad) {  
}
```

En el ejemplo anterior, se declara un registro `Persona` con dos campos `nombre` y `edad`.

Ventajas de los Registros

Los registros tienen las siguientes ventajas:

- **Concisión:** Los registros permiten definir clases de datos de forma concisa y eficiente.
- **Inmutabilidad:** Los campos de un registro son inmutables, lo que evita errores de modificación accidental.
- **Métodos de Acceso:** Los registros proporcionan métodos de acceso generados automáticamente para acceder a los campos de forma segura.
- **Métodos `toString`, `equals` y `hashCode`:** Los registros proporcionan métodos `toString`, `equals` y `hashCode` generados automáticamente para facilitar la impresión, comparación y uso en colecciones.

Desventajas de los Registros

Los registros tienen las siguientes desventajas:

- **Limitaciones:** Los registros tienen limitaciones en comparación con las clases normales, como la falta de constructores con parámetros y la imposibilidad de extender otras clases.
- **Compatibilidad:** Los registros son una característica relativamente nueva en Java, por lo que pueden no ser compatibles con versiones anteriores de Java.
- **Complejidad:** Los registros pueden ser más complejos de entender para los programadores que no están familiarizados con esta característica.
- **Uso Limitado:** Los registros son más adecuados para clases de datos simples y no para clases con lógica de negocio compleja.
- **Personalización Limitada:** Los registros tienen una personalización limitada en comparación con las clases normales, ya que los métodos y constructores son generados automáticamente.

Conclusión

Los registros son una característica poderosa de Java que permite definir clases de datos de forma concisa y eficiente. Los registros proporcionan campos inmutables, métodos de acceso generados automáticamente y métodos `toString`, `equals` y `hashCode` para facilitar el uso de las clases de datos. Aunque los registros tienen algunas limitaciones,

como la falta de constructores con parámetros y la imposibilidad de extender otras clases, son una herramienta útil para definir clases de datos simples y eficientes en Java.

Los niveles de acceso público y privado en Java

En Java, los niveles de acceso `public` y `private` se utilizan para controlar la visibilidad de los miembros de una clase. Estos niveles de acceso determinan si un miembro de una clase es accesible desde otras clases o solo desde la propia clase.

Nivel de acceso `public`

El nivel de acceso `public` se utiliza para declarar miembros de una clase que son accesibles desde cualquier clase en el mismo paquete o en otros paquetes. Los miembros declarados con el nivel de acceso `public` son visibles para todas las clases, independientemente de su ubicación en el proyecto.

```
public class MiClase {  
    public int miVariable;  
    public void miMetodo() {  
        System.out.println("Este es un método público");  
    }  
}
```

En el ejemplo anterior, `miVariable` y `miMetodo` son miembros públicos de la clase `MiClase`. Esto significa que pueden ser accedidos desde cualquier clase en el mismo paquete o en otros paquetes.

Por ejemplo, si tenemos otra clase en un paquete diferente que desea acceder a `miVariable` y `miMetodo`, puede hacerlo sin restricciones:

```
public class OtraClase {  
    public static void main(String[] args) {  
        MiClase instancia = new MiClase();  
        instancia.miVariable = 10;  
        instancia.miMetodo();  
    }  
}
```

Nivel de acceso private

El nivel de acceso `private` se utiliza para declarar miembros de una clase que solo son accesibles desde la propia clase. Los miembros declarados con el nivel de acceso `private` no son visibles para otras clases, incluso si están en el mismo paquete.

```
public class MiClase {  
    private int miVariable;  
    private void miMetodo() {  
        System.out.println("Este es un método privado");  
    }  
}
```

En el ejemplo anterior, `miVariable` y `miMetodo` son miembros privados de la clase `MiClase`. Esto significa que solo pueden ser accedidos desde la propia clase `MiClase`.

Si intentamos acceder a `miVariable` y `miMetodo` desde otra clase, obtendremos un error de compilación:

```
public class OtraClase {  
    public static void main(String[] args) {  
        MiClase instancia = new MiClase();  
        instancia.miVariable = 10; // Error: miVariable has private  
access in MiClase  
        instancia.miMetodo(); // Error: miMetodo() has private access in  
MiClase  
    }  
}
```

Resumen

- El nivel de acceso `public` se utiliza para declarar miembros de una clase que son accesibles desde cualquier clase en el mismo paquete o en otros paquetes.
- El nivel de acceso `private` se utiliza para declarar miembros de una clase que solo son accesibles desde la propia clase.

- Los niveles de acceso `public` y `private` son fundamentales para controlar la visibilidad y el encapsulamiento de los miembros de una clase en Java.
- Es una buena práctica utilizar el nivel de acceso `private` para los miembros de una clase que no deben ser accesibles desde fuera de la clase, y utilizar el nivel de acceso `public` para los miembros que deben ser accesibles desde otras clases.

¿Qué son los arreglos?

Un arreglo es una estructura de datos que nos permite almacenar una colección de elementos del mismo tipo. Los arreglos son una herramienta muy útil en programación, ya que nos permiten almacenar y manipular grandes cantidades de datos de manera eficiente.

En Java, los arreglos son objetos que se definen mediante la palabra clave `new` seguida del tipo de datos que queremos almacenar y el tamaño del arreglo. Por ejemplo, para crear un arreglo de enteros con 5 elementos, podemos hacer lo siguiente:

```
int[] numeros = new int[5];
```

En este caso, `numeros` es un arreglo de enteros con 5 elementos, cuyos índices van desde 0 hasta 4. Sin embargo, existe una forma más sencilla de inicializar arreglos en Java, que consiste en declarar el arreglo y asignarle los valores directamente:

```
int[] numeros = {1, 2, 3, 4, 5};
```

En este caso, `numeros` es un arreglo de enteros con 5 elementos, cuyos valores son 1, 2, 3, 4 y 5. Esta forma de inicializar arreglos es más cómoda y legible que la anterior, por lo que se recomienda utilizarla siempre que sea posible.

Los arreglos en Java son de tamaño fijo, es decir, una vez que se crea un arreglo con un tamaño determinado, no se puede modificar su tamaño. Sin embargo, es posible cambiar los valores de los elementos del arreglo en cualquier momento. Por ejemplo, para cambiar el valor del segundo elemento del arreglo `numeros`, podemos hacer lo siguiente:

```
numeros[1] = 10;
```

En este caso, el valor del segundo elemento del arreglo `numeros` pasa a ser 10 en lugar de 2. Es importante tener en cuenta que los índices de los arreglos en Java comienzan en 0, por lo que el primer elemento del arreglo tiene índice 0, el segundo elemento tiene índice 1, y así sucesivamente.

En resumen, los arreglos en Java son una estructura de datos que nos permite almacenar y manipular colecciones de elementos del mismo tipo. Los arreglos en Java son de tamaño fijo, pero es posible cambiar los valores de los elementos en cualquier momento.

Además, es posible inicializar arreglos de forma sencilla y legible utilizando la notación de corchetes {}.

¿Cómo acceder a los elementos de un arreglo?

Para acceder a los elementos de un arreglo en Java, utilizamos el nombre del arreglo seguido de un índice entre corchetes.

Por ejemplo, para acceder al primer elemento del arreglo numeros, podemos hacer lo siguiente:

```
int primerElemento = numeros[0];
```

En este caso, primerElemento es igual al primer elemento del arreglo numeros, cuyo índice es 0. De manera similar, para acceder al segundo elemento del arreglo numeros, podemos hacer lo siguiente:

```
int segundoElemento = numeros[1];
```

En este caso, segundoElemento es igual al segundo elemento del arreglo numeros, cuyo índice es 1. Es importante tener en cuenta que los índices de los arreglos en Java comienzan en 0, por lo que el primer elemento del arreglo tiene índice 0, el segundo elemento tiene índice 1, y así sucesivamente.

Si intentamos acceder a un elemento de un arreglo utilizando un índice que está fuera del rango de índices válidos del arreglo, se producirá un error en tiempo de ejecución. Por ejemplo, si intentamos acceder al sexto elemento del arreglo numeros, que tiene solo 5 elementos, se producirá un error en tiempo de ejecución.

En resumen, para acceder a los elementos de un arreglo en Java, utilizamos el nombre del arreglo seguido de un índice entre corchetes. Los índices de los arreglos en Java comienzan en 0, por lo que el primer elemento del arreglo tiene índice 0, el segundo elemento tiene índice 1, y así sucesivamente. Si intentamos acceder a un elemento de un arreglo utilizando un índice que está fuera del rango de índices válidos del arreglo, se producirá un error en tiempo de ejecución.

¿Cómo recorrer un arreglo en Java?

Para recorrer un arreglo en Java, podemos utilizar un bucle `for` que vaya desde 0 hasta el tamaño del arreglo menos 1. Por ejemplo, para recorrer el arreglo `numeros` que contiene 5 elementos, podemos hacer lo siguiente:

```
for (int i = 0; i < numeros.length; i++) {  
    System.out.println(numeros[i]);  
}
```

En este caso, el bucle `for` recorre el arreglo `numeros` desde el primer elemento hasta el último, imprimiendo en pantalla cada uno de los elementos del arreglo. Es importante tener en cuenta que los índices de los arreglos en Java comienzan en 0, por lo que el primer elemento del arreglo tiene índice 0, el segundo elemento tiene índice 1, y así sucesivamente.

También es posible recorrer un arreglo en Java utilizando un bucle `for-each`, que nos permite recorrer todos los elementos del arreglo en orden sin necesidad de utilizar un índice. Por ejemplo, para recorrer el arreglo `numeros` que contiene 5 elementos, podemos hacer lo siguiente:

```
for (int numero : numeros) {  
    System.out.println(numero);  
}
```

En este caso, el bucle `for-each` recorre el arreglo `numeros` desde el primer elemento hasta el último, imprimiendo en pantalla cada uno de los elementos del arreglo. Es importante tener en cuenta que el bucle `for-each` solo nos permite recorrer los elementos del arreglo en orden, pero no nos proporciona acceso a los índices de los elementos.

En resumen, para recorrer un arreglo en Java, podemos utilizar un bucle `for` que vaya desde 0 hasta el tamaño del arreglo menos 1, o un bucle `for-each` que nos permite recorrer todos los elementos del arreglo en orden sin necesidad de utilizar un índice. Los índices de los arreglos en Java comienzan en 0, por lo que el primer elemento del arreglo tiene índice 0, el segundo elemento tiene índice 1, y así sucesivamente.

¿Cómo inicializar un arreglo en Java?

En Java, los arreglos se pueden inicializar de varias formas. La forma más sencilla de inicializar un arreglo en Java es utilizando la notación de corchetes {} para asignarle los valores directamente. Por ejemplo, para inicializar un arreglo de enteros con los valores 1, 2, 3, 4 y 5, podemos hacer lo siguiente:

```
int[] numeros = {1, 2, 3, 4, 5};
```

En este caso, `numeros` es un arreglo de enteros con 5 elementos, cuyos valores son 1, 2, 3, 4 y 5. Esta forma de inicializar arreglos es más cómoda y legible que la anterior, por lo que se recomienda utilizarla siempre que sea posible.

También es posible inicializar un arreglo en Java utilizando un bucle `for`. Por ejemplo, para inicializar un arreglo de enteros con los valores del 1 al 5, podemos hacer lo siguiente:

```
int[] numeros = new int[5];
for (int i = 0; i < numeros.length; i++) {
    numeros[i] = i + 1;
}
```

En este caso, el bucle `for` inicializa el arreglo `numeros` con los valores del 1 al 5, asignando a cada elemento del arreglo el valor de `i + 1`. Esta forma de inicializar arreglos es útil cuando queremos inicializar un arreglo con valores que siguen un patrón o una secuencia.

En resumen, en Java los arreglos se pueden inicializar de varias formas. La forma más sencilla de inicializar un arreglo en Java es utilizando la notación de corchetes {} para asignarle los valores directamente. También es posible inicializar un arreglo en Java utilizando un bucle `for` para asignarle valores que siguen un patrón o una secuencia.

¿Cómo copiar un arreglo en Java?

En Java, para copiar un arreglo a otro arreglo, podemos utilizar el método `arraycopy` de la clase `System`. Por ejemplo, para copiar el arreglo `numeros` a un nuevo arreglo `copia`, podemos hacer lo siguiente:

```
int[] copia = new int[numeros.length];
System.arraycopy(numeros, 0, copia, 0, numeros.length);
```

En este caso, el método `arraycopy` copia los elementos del arreglo `numeros` al arreglo `copia`, comenzando desde el índice 0 en ambos arreglos y copiando todos los elementos del arreglo `numeros`. Al finalizar este código, el arreglo `copia` contendrá los mismos elementos que el arreglo `numeros`.

También es posible copiar un arreglo en Java utilizando un bucle `for`. Por ejemplo, para copiar el arreglo `numeros` a un nuevo arreglo `copia`, podemos hacer lo siguiente:

```
int[] copia = new int[numeros.length];
for (int i = 0; i < numeros.length; i++) {
    copia[i] = numeros[i];
}
```

En este caso, el bucle `for` copia los elementos del arreglo `numeros` al arreglo `copia`, asignando a cada elemento del arreglo `copia` el valor del elemento correspondiente del arreglo `numeros`. Al finalizar este código, el arreglo `copia` contendrá los mismos elementos que el arreglo `numeros`.

En resumen, en Java para copiar un arreglo a otro arreglo, podemos utilizar el método `arraycopy` de la clase `System` o un bucle `for` para copiar los elementos uno por uno. Ambos métodos son válidos y nos permiten copiar un arreglo a otro arreglo de manera eficiente.

¿Cómo ordenar un arreglo en Java?

En Java, para ordenar un arreglo en orden ascendente, podemos utilizar el método `sort` de la clase `Arrays`. Por ejemplo, para ordenar el arreglo `numeros` en orden ascendente, podemos hacer lo siguiente:

```
Arrays.sort(numeros);
```

En este caso, el método `sort` ordena los elementos del arreglo `numeros` en orden ascendente, de menor a mayor. Al finalizar este código, el arreglo `numeros` estará ordenado en orden ascendente.

También es posible ordenar un arreglo en orden descendente utilizando el método `sort` de la clase `Arrays` en combinación con un comparador. Por ejemplo, para ordenar el arreglo `numeros` en orden descendente, podemos hacer lo siguiente:

```
Arrays.sort(numeros, Collections.reverseOrder());
```

En este caso, el método `sort` ordena los elementos del arreglo `numeros` en orden descendente, de mayor a menor. Al finalizar este código, el arreglo `numeros` estará ordenado en orden descendente.

En resumen, en Java para ordenar un arreglo en orden ascendente, podemos utilizar el método `sort` de la clase `Arrays`. También es posible ordenar un arreglo en orden descendente utilizando el método `sort` de la clase `Arrays` en combinación con un comparador. Ambos métodos nos permiten ordenar un arreglo en orden ascendente o descendente de manera eficiente.

¿Cómo buscar un elemento en un arreglo en Java?

En Java, para buscar un elemento en un arreglo, podemos utilizar el método `binarySearch` de la clase `Arrays`. Por ejemplo, para buscar el elemento `3` en el arreglo `numeros`, podemos hacer lo siguiente:

```
int indice = Arrays.binarySearch(numeros, 3);
```

En este caso, el método `binarySearch` busca el elemento `3` en el arreglo `numeros` y devuelve su índice si se encuentra, o un valor negativo si no se encuentra. Si el elemento `3` se encuentra en el arreglo `numeros`, el valor de `indice` será el índice del elemento `3` en el arreglo `numeros`.

Es importante tener en cuenta que el método `binarySearch` requiere que el arreglo esté ordenado en orden ascendente antes de realizar la búsqueda. Si el arreglo no está ordenado, el resultado de la búsqueda puede ser incorrecto.

En resumen, en Java para buscar un elemento en un arreglo, podemos utilizar el método `binarySearch` de la clase `Arrays`. Este método nos permite buscar un elemento en un arreglo ordenado en orden ascendente y nos devuelve el índice del elemento si se encuentra, o un valor negativo si no se encuentra.

¿Cómo eliminar un elemento de un arreglo en Java?

En Java, los arreglos son de tamaño fijo, es decir, una vez que se crea un arreglo con un tamaño determinado, no se puede modificar su tamaño. Sin embargo, es posible

"eliminar" un elemento de un arreglo asignándole un valor especial como 0 o null.

Por ejemplo, para "eliminar" el segundo elemento del arreglo numeros, podemos hacer lo siguiente:

```
numeros[1] = 0;
```

En este caso, el valor del segundo elemento del arreglo numeros se cambia a 0, lo que simula la eliminación del elemento. Es importante tener en cuenta que esta técnica no elimina realmente el elemento del arreglo, sino que lo marca como "eliminado" asignándole un valor especial.

Si necesitamos eliminar un elemento de un arreglo de manera más eficiente, podemos utilizar una lista en lugar de un arreglo. Las listas en Java, como ArrayList, nos permiten agregar, eliminar y modificar elementos de manera dinámica, lo que las hace más flexibles que los arreglos.

En resumen, en Java para "eliminar" un elemento de un arreglo, podemos asignarle un valor especial como 0 o null para marcarlo como "eliminado". Sin embargo, esta técnica no elimina realmente el elemento del arreglo, sino que lo marca como "eliminado". Si necesitamos eliminar elementos de manera más eficiente, es recomendable utilizar una lista en lugar de un arreglo.

¿Cómo invertir un arreglo en Java?

En Java, para invertir un arreglo, podemos utilizar un bucle for que recorra la mitad del arreglo e intercambie los elementos de las posiciones opuestas. Por ejemplo, para invertir el arreglo numeros, podemos hacer lo siguiente:

```
for (int i = 0; i < numeros.length / 2; i++) {  
    int temp = numeros[i];  
    numeros[i] = numeros[numeros.length - i - 1];  
    numeros[numeros.length - i - 1] = temp;  
}
```

En este caso, el bucle for recorre la mitad del arreglo numeros intercambiando los elementos de las posiciones opuestas. Al finalizar este código, el arreglo numeros estará invertido, es decir, los elementos estarán en orden inverso al original.

También es posible invertir un arreglo en Java utilizando el método `reverse` de la clase `Collections`. Por ejemplo, para invertir el arreglo `numeros`, podemos hacer lo siguiente:

```
List<Integer> lista = Arrays.asList(numeros);
Collections.reverse(lista);
```

En este caso, el método `reverse` de la clase `Collections` invierte los elementos de la lista `lista`, que contiene los elementos del arreglo `numeros`. Al finalizar este código, el arreglo `numeros` estará invertido, es decir, los elementos estarán en orden inverso al original.

En resumen, en Java para invertir un arreglo, podemos utilizar un bucle `for` que recorra la mitad del arreglo e intercambie los elementos de las posiciones opuestas. También es posible invertir un arreglo en Java utilizando el método `reverse` de la clase `Collections`. Ambos métodos nos permiten invertir un arreglo de manera eficiente.

¿Cómo encontrar el máximo y el mínimo de un arreglo en Java?

En Java, para encontrar el máximo y el mínimo de un arreglo, podemos utilizar los métodos `max` y `min` de la clase `Collections`. Por ejemplo, para encontrar el máximo y el mínimo del arreglo `numeros`, podemos hacer lo siguiente:

```
int maximo = Collections.max(Arrays.asList(numeros));
int minimo = Collections.min(Arrays.asList(numeros));
```

En este caso, los métodos `max` y `min` de la clase `Collections` nos permiten encontrar el máximo y el mínimo de la lista `lista`, que contiene los elementos del arreglo `numeros`. Al finalizar este código, las variables `maximo` y `minimo` contendrán el máximo y el mínimo del arreglo `numeros`, respectivamente.

También es posible encontrar el máximo y el mínimo de un arreglo en Java utilizando un bucle `for`. Por ejemplo, para encontrar el máximo y el mínimo del arreglo `numeros`, podemos hacer lo siguiente:

```
int maximo = numeros[0];
int minimo = numeros[0];

for (int i = 1; i < numeros.length; i++) {
```

```
if (numeros[i] > maximo) {  
    maximo = numeros[i];  
}  
if (numeros[i] < minimo) {  
    minimo = numeros[i];  
}  
}
```

En este caso, el bucle `for` recorre el arreglo `numeros` buscando el máximo y el mínimo de los elementos. Al finalizar este código, las variables `maximo` y `minimo` contendrán el máximo y el mínimo del arreglo `numeros`, respectivamente.

En resumen, en Java para encontrar el máximo y el mínimo de un arreglo, podemos utilizar los métodos `max` y `min` de la clase `Collections` o un bucle `for` que recorra el arreglo buscando el máximo y el mínimo de los elementos. Ambos métodos nos permiten encontrar el máximo y el mínimo de un arreglo de manera eficiente.

¿Cómo calcular la suma y el promedio de un arreglo en Java?

En Java, para calcular la suma y el promedio de un arreglo, podemos utilizar un bucle `for` que recorra el arreglo acumulando la suma de los elementos. Por ejemplo, para calcular la suma y el promedio del arreglo `numeros`, podemos hacer lo siguiente:

```
int suma = 0;  
for (int numero : numeros) {  
    suma += numero;  
}  
  
double promedio = (double) suma / numeros.length;
```

En este caso, el bucle `for` recorre el arreglo `numeros` acumulando la suma de los elementos en la variable `suma`. Al finalizar este código, la variable `suma` contendrá la suma de los elementos del arreglo `numeros`, y la variable `promedio` contendrá el promedio de los elementos del arreglo `numeros`.

También es posible calcular la suma y el promedio de un arreglo en Java utilizando los métodos `sum` y `average` de la clase `Arrays`. Por ejemplo, para calcular la suma y el promedio del arreglo `numeros`, podemos hacer lo siguiente:

```
int suma = Arrays.stream(numeros).sum();
double promedio = Arrays.stream(numeros).average().orElse(0);
```

En este caso, los métodos `sum` y `average` de la clase `Arrays` nos permiten calcular la suma y el promedio de los elementos del arreglo `numeros`. Al finalizar este código, la variable `suma` contendrá la suma de los elementos del arreglo `numeros`, y la variable `promedio` contendrá el promedio de los elementos del arreglo `numeros`.

En resumen, en Java para calcular la suma y el promedio de un arreglo, podemos utilizar un bucle `for` que recorra el arreglo acumulando la suma de los elementos, o los métodos `sum` y `average` de la clase `Arrays` que nos permiten calcular la suma y el promedio de los elementos del arreglo de manera eficiente.

¿Cómo encontrar un elemento repetido en un arreglo en Java?

En Java, para encontrar un elemento repetido en un arreglo, podemos utilizar un bucle `for` anidado que compare cada elemento con los demás elementos del arreglo. Por ejemplo, para encontrar un elemento repetido en el arreglo `numeros`, podemos hacer lo siguiente:

```
for (int i = 0; i < numeros.length; i++) {
    for (int j = i + 1; j < numeros.length; j++) {
        if (numeros[i] == numeros[j]) {
            System.out.println("El elemento " + numeros[i] + " está
repetido.");
        }
    }
}
```

En este caso, el bucle `for` anidado compara cada elemento del arreglo `numeros` con los demás elementos del arreglo buscando elementos repetidos. Si se encuentra un

elemento repetido, se imprime en pantalla un mensaje indicando que el elemento está repetido.

También es posible encontrar un elemento repetido en un arreglo en Java utilizando un conjunto (Set) para almacenar los elementos únicos del arreglo. Por ejemplo, para encontrar un elemento repetido en el arreglo `numeros`, podemos hacer lo siguiente:

```
Set<Integer> conjunto = new HashSet<>();
for (int numero : numeros) {
    if (!conjunto.add(numero)) {
        System.out.println("El elemento " + numero + " está repetido.");
    }
}
```

En este caso, el conjunto `conjunto` almacena los elementos únicos del arreglo `numeros`, y se utiliza el método `add` para agregar un elemento al conjunto. Si el método `add` devuelve `false`, significa que el elemento ya estaba en el conjunto, por lo que se imprime en pantalla un mensaje indicando que el elemento está repetido.

En resumen, en Java para encontrar un elemento repetido en un arreglo, podemos utilizar un bucle `for` anidado que compare cada elemento con los demás elementos del arreglo, o un conjunto (Set) para almacenar los elementos únicos del arreglo y buscar elementos repetidos. Ambos métodos nos permiten encontrar elementos repetidos en un arreglo de manera eficiente.

¿Cómo encontrar los elementos únicos de un arreglo en Java?

En Java, para encontrar los elementos únicos de un arreglo, podemos utilizar un conjunto (Set) para almacenar los elementos únicos del arreglo. Por ejemplo, para encontrar los elementos únicos del arreglo `numeros`, podemos hacer lo siguiente:

```
Set<Integer> conjunto = new HashSet<>();
for (int numero : numeros) {
    conjunto.add(numero);
}

for (int numero : conjunto) {
```

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

En este caso, el conjunto `conjunto` almacena los elementos únicos del arreglo `numeros`, y se utiliza el método `add` para agregar un elemento al conjunto. Al finalizar este código, el conjunto `conjunto` contendrá los elementos únicos del arreglo `numeros`, y se imprime en pantalla cada uno de los elementos únicos.

También es posible encontrar los elementos únicos de un arreglo en Java utilizando un bucle `for` anidado que compare cada elemento con los demás elementos del arreglo. Por ejemplo, para encontrar los elementos únicos del arreglo `numeros`, podemos hacer lo siguiente:

```
for (int i = 0; i < numeros.length; i++) {
    boolean unico = true;
    for (int j = 0; j < numeros.length; j++) {
        if (i != j && numeros[i] == numeros[j]) {
            unico = false;
            break;
        }
    }
    if (unico) {
        System.out.println(numeros[i]);
    }
}
```

En este caso, el bucle `for` anidado compara cada elemento del arreglo `numeros` con los demás elementos del arreglo buscando elementos únicos. Si se encuentra un elemento único, se imprime en pantalla el elemento único.

En resumen, en Java para encontrar los elementos únicos de un arreglo, podemos utilizar un conjunto (`Set`) para almacenar los elementos únicos del arreglo, o un bucle `for` anidado que compare cada elemento con los demás elementos del arreglo. Ambos métodos nos permiten encontrar los elementos únicos de un arreglo de manera eficiente.

¿Cómo encontrar los elementos comunes de dos arreglos en Java?

En Java, para encontrar los elementos comunes de dos arreglos, podemos utilizar dos conjuntos (`Set`) para almacenar los elementos de cada arreglo y luego comparar los conjuntos para encontrar los elementos comunes. Por ejemplo, para encontrar los elementos comunes de los arreglos `numeros1` y `numeros2`, podemos hacer lo siguiente:

```
Set<Integer> conjunto1 = new HashSet<>(Arrays.asList(numeros1));
Set<Integer> conjunto2 = new HashSet<>(Arrays.asList(numeros2));

conjunto1.retainAll(conjunto2);

for (int numero : conjunto1) {
    System.out.println(numero);
}
```

En este caso, los conjuntos `conjunto1` y `conjunto2` almacenan los elementos de los arreglos `numeros1` y `numeros2`, respectivamente. Luego, se utiliza el método `retainAll` para retener solo los elementos comunes entre los conjuntos `conjunto1` y `conjunto2`. Al finalizar este código, el conjunto `conjunto1` contendrá los elementos comunes de los arreglos `numeros1` y `numeros2`, y se imprime en pantalla cada uno de los elementos comunes.

También es posible encontrar los elementos comunes de dos arreglos en Java utilizando dos bucles `for` anidados que comparan cada elemento de un arreglo con los elementos del otro arreglo. Por ejemplo, para encontrar los elementos comunes de los arreglos `numeros1` y `numeros2`, podemos hacer lo siguiente:

```
for (int numero1 : numeros1) {
    for (int numero2 : numeros2) {
        if (numero1 == numero2) {
            System.out.println(numero1);
        }
    }
}
```

En este caso, los bucles `for` anidados comparan cada elemento del arreglo `numeros1` con los elementos del arreglo `numeros2` buscando elementos comunes. Si se encuentra un elemento común, se imprime en pantalla el elemento común.

En resumen, en Java para encontrar los elementos comunes de dos arreglos, podemos utilizar dos conjuntos (`Set`) para almacenar los elementos de cada arreglo y luego comparar los conjuntos para encontrar los elementos comunes, o dos bucles `for` anidados que comparen cada elemento de un arreglo con los elementos del otro arreglo. Ambos métodos nos permiten encontrar los elementos comunes de dos arreglos de manera eficiente.

¿Qué son los arreglos bidimensionales?

Un arreglo bidimensional es una estructura de datos que permite almacenar datos en una tabla de filas y columnas. En Java, los arreglos bidimensionales son arreglos de arreglos, es decir, cada elemento de un arreglo bidimensional es un arreglo unidimensional.

Declaración de arreglos bidimensionales

Para declarar un arreglo bidimensional en Java, se debe especificar el tipo de datos que se almacenará en el arreglo, seguido de dos corchetes `[][]` y el nombre del arreglo. Por ejemplo, para declarar un arreglo bidimensional de enteros llamado `matriz`, se puede hacer de la siguiente manera:

```
int[][] matriz;
```

Inicialización de arreglos bidimensionales

Para inicializar un arreglo bidimensional en Java, se debe especificar el tamaño de las filas y columnas del arreglo. Por ejemplo, para inicializar un arreglo bidimensional de 3 filas y 4 columnas llamado `matriz`, se puede hacer de la siguiente manera:

```
int[][] matriz = new int[3][4];
```

También se puede inicializar un arreglo bidimensional con valores específicos. Por ejemplo, para inicializar un arreglo bidimensional con los valores 1, 2, 3, 4, 5, 6, 7, 8, 9, se puede hacer de la siguiente manera:

```
int[][] matriz = {  
    {1, 2, 3},  
    {4, 5, 6},  
    {7, 8, 9}  
};
```

En este caso, el arreglo bidimensional `matriz` tiene 3 filas y 3 columnas, y los valores se inicializan en orden de izquierda a derecha y de arriba a abajo.

Acceso a elementos de arreglos bidimensionales

Para acceder a un elemento de un arreglo bidimensional en Java, se debe especificar el índice de la fila y columna del elemento. Por ejemplo, para acceder al elemento en la fila 1 y columna 2 de la matriz `matriz`, se puede hacer de la siguiente manera:

```
int elemento = matriz[1][2];
```

Ejemplo de uso de arreglos bidimensionales

A continuación se muestra un ejemplo de uso de arreglos bidimensionales en Java para almacenar una matriz de enteros y calcular la suma de los elementos de la matriz:

```
public class Main {
    public static void main(String[] args) {
        int[][] matriz = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        int suma = 0;

        for (int i = 0; i < matriz.length; i++) {
            for (int j = 0; j < matriz[i].length; j++) {
                suma += matriz[i][j];
            }
        }

        System.out.println("La suma de los elementos de la matriz es: " +
+ suma);
    }
}
```

En este ejemplo, se declara e inicializa un arreglo bidimensional `matriz` con valores específicos, se calcula la suma de los elementos de la matriz utilizando un bucle `for` anidado, y se imprime el resultado en la consola.

Los arreglos bidimensionales son una herramienta poderosa para trabajar con datos tabulares en Java y se utilizan comúnmente en aplicaciones que requieren el almacenamiento y procesamiento de datos en forma de tablas.

Conclusiones

Los arreglos bidimensionales son una estructura de datos que permite almacenar datos en forma de tablas de filas y columnas en Java. Se pueden declarar, inicializar y acceder a los elementos de un arreglo bidimensional de manera similar a un arreglo unidimensional, pero con la diferencia de que se utilizan dos índices para acceder a los elementos de las filas y columnas del arreglo. Los arreglos bidimensionales son útiles para representar y manipular datos tabulares en aplicaciones Java y son una herramienta fundamental en el desarrollo de software.

¿Existen los arreglos de más de dos dimensiones en Java?

Sí, en Java es posible declarar arreglos de más de dos dimensiones. Por ejemplo, para declarar un arreglo tridimensional de enteros llamado `cubo`, se puede hacer de la siguiente manera:

```
int[][][] cubo;
```

Para inicializar un arreglo tridimensional en Java, se debe especificar el tamaño de las filas, columnas y profundidad del arreglo. Por ejemplo, para inicializar un arreglo tridimensional de 2 filas, 3 columnas y 4 profundidades llamado `cubo`, se puede hacer de la siguiente manera:

```
int[][][] cubo = new int[2][3][4];
```

También se puede inicializar un arreglo tridimensional con valores específicos. Por ejemplo, para inicializar un arreglo tridimensional con los valores 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, se puede hacer de la siguiente manera:

```
int[][][] cubo = {  
    {  
        {1, 2, 3, 4},  
        {5, 6, 7, 8},  
        {9, 10, 11, 12}  
    },  
    {  
        {13, 14, 15, 16},  
        {17, 18, 19, 20},  
        {21, 22, 23, 24}  
    }  
};
```

En este caso, el arreglo tridimensional `cubo` tiene 2 filas, 3 columnas y 4 profundidades, y los valores se inicializan en orden de izquierda a derecha, de arriba a abajo y de adelante hacia atrás.

Para acceder a un elemento de un arreglo tridimensional en Java, se debe especificar el índice de la fila, columna y profundidad del elemento. Por ejemplo, para acceder al elemento en la fila 1, columna 2 y profundidad 3 del cubo `cubo`, se puede hacer de la siguiente manera:

```
int elemento = cubo[1][2][3];
```

En resumen, en Java es posible declarar, inicializar y acceder a arreglos de más de dos dimensiones, como arreglos tridimensionales, para almacenar y manipular datos de forma estructurada y eficiente.

¿Existen arreglos de más dimensiones en Java?

Sí, en Java es posible declarar arreglos de más de tres dimensiones. Por ejemplo, para declarar un arreglo de cuatro dimensiones, se puede hacer de la siguiente manera:

```
int[][][][] arreglo4D;
```

De manera similar, se pueden declarar arreglos de cinco dimensiones, seis dimensiones, y así sucesivamente, según las necesidades del programa. Sin embargo, es importante tener en cuenta que el uso de arreglos de más dimensiones puede complicar la estructura y la legibilidad del código, por lo que se recomienda utilizar arreglos de dimensiones más bajas o estructuras de datos alternativas en la mayoría de los casos.

Práctica 1: Nuestra Primera Clase en Java

Introducción

En Java, una clase es una plantilla que define el comportamiento y las propiedades de un objeto. Un objeto, por otro lado, es una instancia de una clase que puede contener datos y métodos. En este tutorial, aprenderás cómo definir clases y objetos en Java y cómo interactuar con ellos.

Definición de una clase

Para definir una clase en Java, utiliza la palabra clave `class` seguida del nombre de la clase y las llaves `{}` para encerrar el cuerpo de la clase. Por ejemplo:

```
public class Persona {  
    // Propiedades  
    String nombre;  
    int edad;  
  
    // Métodos  
    public void saludar() {  
        System.out.println("Hola, soy " + nombre + " y tengo " + edad +  
" años.");  
    }  
}
```

En este ejemplo, se define una clase `Persona` con dos propiedades (`nombre` y `edad`) y un método `saludar` que imprime un mensaje por consola.

Creación de un objeto

Para crear un objeto en Java, utiliza la palabra clave `new` seguida del nombre de la clase y los paréntesis `()`. Por ejemplo:

```
public class Main {  
    public static void main(String[] args) {  
        // Crear un objeto de la clase Persona  
        Persona persona = new Persona();  
  
        // Inicializar las propiedades del objeto  
        persona.nombre = "Juan";  
        persona.edad = 30;  
  
        // Llamar al método saludar  
        persona.saludar();  
    }  
}
```

En este ejemplo, se crea un objeto `persona` de la clase `Persona` y se inicializan sus propiedades `nombre` y `edad`.

⚠ Nota: En Java, los métodos y propiedades de una clase pueden tener diferentes niveles de acceso, como `public`, `private` y `protected`. Estos modificadores de acceso determinan quién puede acceder a los miembros de la clase. Por ejemplo, si un método es `public`, puede ser accedido desde cualquier parte del programa, mientras que si es `private`, solo puede ser accedido desde dentro de la clase. Pero en este curso solo usaremos `public`.

Conclusiones

En resumen, las clases y objetos son elementos fundamentales en Java que te permiten modelar el comportamiento y las propiedades de tus aplicaciones. Al definir clases y crear objetos, puedes organizar y reutilizar tu código de manera eficiente. ¡Práctica la creación de clases y objetos en Java para mejorar tus habilidades de programación!

Práctica 2: Sistema de Cálculo de Descuentos

Descripción

Crea una clase llamada CalculadoraDescuentos que permita calcular el precio final de un producto después de aplicar un descuento. El programa debe permitir:

- Calcular el precio final de un producto con un descuento fijo.
- Calcular el precio final de un producto con un descuento porcentual.
- Mostrar un mensaje de bienvenida y despedida.

Requisitos

- Usa un atributo final para definir el descuento fijo (por ejemplo, 10 unidades monetarias).
- Usa un atributo estático para definir el descuento porcentual (por ejemplo, 15%).
- Implementa métodos estáticos para calcular el precio final con ambos tipos de descuentos.
- Usa JOptionPane para mostrar el menú y leer los datos necesarios.

Resolvamos el problema

Primero que nada vamos a crear la clase CalculadoraDescuentos. En esta clase vamos a definir los atributos finales de Descuentos tanto fijos como porcentuales. A continuación, se muestra el código de la clase CalculadoraDescuentos:

```
import javax.swing.JOptionPane;  
  
public class CalculadoraDescuentos {
```

```
    private static final double DESCUENTO_FIJO = 10;
    private static final double DESCUENTO_PORCENTUAL = 0.15;
}
```

Ahora vamos a implementar los métodos estáticos para calcular el precio final con ambos tipos de descuentos. A continuación, se muestra el código de las funciones:

```
// Método estático para calcular el precio con descuento fijo
public static double calcularPrecioConDescuentoFijo(double precio){

    return precio - DESCUENTO_FIJO;
}

// Método estático para calcular el precio con descuento porcentual
public static double calcularPrecioConDescuentoPorcentual(double precio)
{

    return precio * (1 - DESCUENTO_PORCENTUAL);
}
```

Por último, vamos a implementar el método main para mostrar el menú y leer los datos necesarios. A continuación, se muestra el código del método main:

```
public static void menu(){

    String mensaje = "Bienvenido a la Calculadora de Descuentos\n\n";
    mensaje += "1. Calcular precio con descuento fijo\n";
    mensaje += "2. Calcular precio con descuento porcentual\n";
    mensaje += "3. Salir\n\n";
    mensaje += "Seleccione una opción:";

    int opcion = 0;
    double precio = 0;

    while(opcion != 3){

        opcion = Integer.parseInt(JOptionPane.showInputDialog(mensaje));
```

```

switch(opcion){
    case 1->{
        precio =
Double.parseDouble(JOptionPane.showInputDialog("Ingrese el precio del
producto:"));

        JOptionPane.showMessageDialog(null, "El precio final con
descuento fijo es: " + calcularPrecioConDescuentoFijo(precio));
    }
    case 2->{
        precio =
Double.parseDouble(JOptionPane.showInputDialog("Ingrese el precio del
producto:"));

        JOptionPane.showMessageDialog(null, "El precio final con
descuento porcentual es: " +
calcularPrecioConDescuentoPorcentual(precio));
    }
    case 3->{
        JOptionPane.showMessageDialog(null, "Gracias por usar la
Calculadora de Descuentos");
    }
}
}

```

Para finalizar, vamos a llamar al método menu en el método main de la clase CalculadoraDescuentos. A continuación, se muestra el código completo de la clase CalculadoraDescuentos:

```

import javax.swing.JOptionPane;

public class CalculadoraDescuentos {

    private static final double DESCUENTO_FIJO = 10;
    private static final double DESCUENTO_PORCENTUAL = 0.15;

    // Método estático para calcular el precio con descuento fijo
}

```

```
public static double calcularPrecioConDescuentoFijo(double precio){

    return precio - DESCUENTO_FIJO;
}

// Método estático para calcular el precio con descuento porcentual
public static double calcularPrecioConDescuentoPorcentual(double
precio){

    return precio * (1 - DESCUENTO_PORCENTUAL);
}

public static void menu(){

    String mensaje = "Bienvenido a la Calculadora de
Descuentos\n\n";
    mensaje += "1. Calcular precio con descuento fijo\n";
    mensaje += "2. Calcular precio con descuento porcentual\n";
    mensaje += "3. Salir\n\n";
    mensaje += "Seleccione una opción:";

    int opcion = 0;
    double precio = 0;

    while(opcion != 3){

        opcion =
Integer.parseInt(JOptionPane.showInputDialog(mensaje));

        switch(opcion){
            case 1->{
                precio =
Double.parseDouble(JOptionPane.showInputDialog("Ingrese el precio del
producto:"));

                JOptionPane.showMessageDialog(null, "El precio final
con descuento fijo es: " + calcularPrecioConDescuentoFijo(precio));
            }
            case 2->{

```

```

        precio =
Double.parseDouble(JOptionPane.showInputDialog("Ingrese el precio del
producto:"));
                JOptionPane.showMessageDialog(null, "El precio final
con descuento porcentual es: " +
calcularPrecioConDescuentoPorcentual(precio));
            }
        case 3->{
                JOptionPane.showMessageDialog(null, "Gracias por
usar la Calculadora de Descuentos");
            }
        }
    }

public static void main(String[] args) {

    menu();
}
}

```

Con esto hemos terminado de implementar la clase CalculadoraDescuentos. Ahora puedes ejecutar el programa y probar su funcionamiento. ¡Buena suerte!

Práctica 3: Calculadora Básica

Descripción

Vamos a implementar una pequeña calculadora que realice operaciones aritméticas simples (suma, resta, multiplicación y división). Tendremos dos tipos de funciones estáticas:

- Funciones sin retorno (void): Para mostrar resultados directamente en pantalla.
- Funciones con retorno (int o double): Para realizar el cálculo y devolver el resultado.

Código

```
import javax.swing.JOptionPane;

public class Calculadora {

    // Función estática sin retorno para mostrar un mensaje
    public static void mostrarMensaje(String mensaje) {
        JOptionPane.showMessageDialog(null, mensaje);
    }

    // Función estática con retorno para sumar dos números
    public static int sumar(int a, int b) {
        return a + b;
    }

    // Función estática con retorno para restar dos números
    public static int restar(int a, int b) {
        return a - b;
    }

    // Función estática con retorno para multiplicar dos números
    public static int multiplicar(int a, int b) {
        return a * b;
    }
}
```

```

}

// Función estática con retorno para dividir dos números
public static double dividir(int a, int b) {
    if (b != 0) {
        return (double) a / b;
    } else {
        mostrarMensaje("Error: División por cero.");
        return 0; // Valor por defecto en caso de error
    }
}

// Función principal (main)
public static void main(String[] args) {
    int num1 = Integer.parseInt(JOptionPane.showInputDialog("Ingrese el primer número:"));
    int num2 = Integer.parseInt(JOptionPane.showInputDialog("Ingrese el segundo número:"));

    int suma = sumar(num1, num2);
    int resta = restar(num1, num2);
    int producto = multiplicar(num1, num2);
    double cociente = dividir(num1, num2);

    // Mostrar los resultados usando la función sin retorno
    mostrarMensaje("Resultados:\n" +
                    "Suma: " + suma + "\n" +
                    "Resta: " + resta + "\n" +
                    "Multiplicación: " + producto + "\n" +
                    "División: " + cociente);
}
}

```

Explicación

En el código anterior, hemos creado una clase `Calculadora` que implementa una calculadora básica en Java. La clase `Calculadora` contiene las siguientes funciones

estáticas:

- `mostrarMensaje`: Función estática sin retorno que muestra un mensaje en una ventana emergente.
- `sumar`: Función estática con retorno que suma dos números enteros.
- `restar`: Función estática con retorno que resta dos números enteros.
- `multiplicar`: Función estática con retorno que multiplica dos números enteros.
- `dividir`: Función estática con retorno que divide dos números enteros y maneja el caso de división por cero.
- `main`: Función principal que solicita al usuario dos números enteros, realiza las operaciones aritméticas y muestra los resultados en una ventana emergente.
- Las funciones `sumar`, `restar`, `multiplicar` y `dividir` realizan las operaciones aritméticas y devuelven el resultado como un valor entero o de punto flotante.
- La función `dividir` comprueba si el segundo número es cero para evitar una división por cero y muestra un mensaje de error en caso de que ocurra.
- La función `main` solicita al usuario dos números enteros, realiza las operaciones aritméticas y muestra los resultados en una ventana emergente utilizando la función `mostrarMensaje`.

Puntos clave

- Las funciones estáticas en Java se declaran con la palabra clave `static` y pertenecen a la clase en sí, no a las instancias de la clase.
- Las funciones estáticas pueden tener o no un valor de retorno y pueden recibir parámetros o no.
- El uso de `JOptionPane` permite mostrar mensajes y solicitar entrada del usuario en una ventana emergente.

- La combinación de funciones con y sin retorno ilustra cómo pueden trabajar juntas para resolver un problema.

Practica 4: Creación de anagramas

Objetivo

El objetivo de esta práctica es crear un programa que genere anagramas a partir de una palabra dada. Un anagrama es una palabra o frase que resulta de la transposición de letras de otra palabra o frase. Por ejemplo, "roma" es un anagrama de "amor".

Descripción

El programa debe recibir una palabra y generar todos los anagramas posibles de la misma. Para ello, se debe implementar una función recursiva que genere los anagramas. La función debe recibir la palabra original y una lista de letras que representan las letras que faltan por colocar en el anagrama. La función debe generar todos los anagramas posibles y mostrarlos en pantalla.

Solución

```
public class Anagrama {  
  
    public static void main(String[] args) {  
  
        anagramas("abc");  
    }  
  
    /**  
     * Función que imprime los anagramas de una palabra  
     *  
     * @param palabra Palabra de la que se quieren obtener los anagramas  
     */  
    public static void anagramas(String palabra) {  
  
        // Convertimos la palabra en un array de caracteres  
        char[] letras = palabra.toCharArray();  
        // Llamamos a la función recursiva que se encarga de obtener los  
        anagramas
```

```

        anagramas("", letras);
    }

    /**
     * Función recursiva que se encarga de obtener los anagramas de una
     * palabra
     *
     * @param anagrama Anagrama que se va formando
     * @param letras Letras que quedan por formar el anagrama
     */
    public static void anagramas(String anagrama, char[] letras) {
        // Recorremos las letras que quedan por formar el anagrama
        for (int i = 0; i < letras.length; i++) {
            // Creamos un nuevo array de longitud una unidad menor que
            el array de letras
            char[] nuevasLetras = new char[letras.length - 1];
            // Copiamos las letras que no estamos utilizando al nuevo
            array
            System.arraycopy(letras, 0, nuevasLetras, 0, i);
            // Verificamos que no nos salgamos de los límites del array
            if (letras.length - (i + 1) >= 0)
                // Copiamos las letras que no estamos utilizando al
                nuevo array
                System.arraycopy(letras, i + 1, nuevasLetras, i,
                letras.length - (i + 1));
            // Llamamos recursivamente a la función para que siga
            formando el anagrama
            anagramas(anagrama + letras[i], nuevasLetras);
        }
        // Si no quedan letras por formar el anagrama, lo imprimimos
        if (letras.length == 0) {
            // Imprimimos el anagrama
            System.out.println(anagrama);
        }
    }
}

```

Analicemos el código:

1. En el método `main` se llama a la función `anagramas` con la palabra "abc" como argumento.
2. La función `anagramas` recibe la palabra y la convierte en un array de caracteres. Luego, llama a la función `anagramas` que recibe el anagrama que se va formando y las letras que quedan por formar el anagrama.
3. La función `anagramas` recibe el anagrama que se va formando y las letras que quedan por formar el anagrama. Luego, recorre las letras que quedan por formar el anagrama y, por cada letra, crea un nuevo array de letras que no contienen la letra que se está utilizando. Luego, llama recursivamente a la función `anagramas` con el anagrama formado y las letras que quedan por formar el anagrama.
4. Si no quedan letras por formar el anagrama, se imprime el anagrama.
5. Al ejecutar el programa, se obtiene la siguiente salida:

```
abc  
acb  
bac  
bca  
cab  
cba
```

En la salida se muestran todos los anagramas posibles de la palabra "abc".

Conclusiones

El uso de funciones recursivas es muy útil para resolver problemas en los que se requiere realizar una tarea de forma repetitiva. En este caso, se utilizó una función recursiva para generar los anagramas de una palabra. La función se encarga de formar el anagrama letra por letra y, cuando no quedan letras por formar el anagrama, lo imprime en pantalla.

Actividad 1: Calculadora de áreas y perímetros de figuras geométricas

Descripción

En esta actividad, se debe implementar una calculadora de áreas y perímetros de figuras geométricas. La calculadora debe ser capaz de calcular el área y perímetro de las siguientes figuras geométricas:

- Cuadrado
- Rectángulo
- Triángulo
- Círculo
- Trapecio
- Rombo
- Romboide
- Polígono regular
- Elipse
- Paralelogramo

La calculadora debe ser capaz de calcular el área y perímetro de cada una de las figuras geométricas mencionadas. Para ello, deberemos generar un tipo enumerado que contenga los tipos de figuras geométricas que se pueden calcular. Además, deberemos usar un menú desplegable con las opciones de figuras geométricas que se pueden calcular. Al seleccionar una figura geométrica, se deberán solicitar los datos necesarios para calcular el área y perímetro de la figura geométrica seleccionada. Finalmente, se deberá mostrar el resultado del cálculo del área y perímetro de la figura geométrica seleccionada.

En caso de que se seleccione una opción no válida, se deberá mostrar un mensaje de error indicando que la opción no es válida, y se deberá solicitar nuevamente la selección de la figura geométrica.

En caso de que algún dato de la figura geométrica seleccionada no sea válido, se deberá mostrar un mensaje de error indicando que el dato no es válido, y se deberá solicitar nuevamente el dato de la figura geométrica.

Requerimientos

1. Crear un tipo enumerado con los tipos de figuras geométricas que se pueden calcular.
2. Crear la funcionalidad de la calculadora de áreas y perímetros de figuras geométricas dentro de una función main de una clase de Java.
3. Probar la funcionalidad de la calculadora de áreas y perímetros de figuras geométricas.
Y tomar capturas de pantalla de la ejecución de la calculadora de áreas y perímetros de figuras geométricas.
4. Crear una portada con el nombre de la actividad y los integrantes del equipo.
5. Adjuntar las evidencias y la portada en el proyecto de Java.
6. Comprimir el proyecto de Java en un archivo .zip y subir el archivo en la plataforma Moodle.

Criterios de evaluación

Criterio	Descripción	Puntaje
Portada	Contiene los datos de identificación de los integrantes del equipo (nombres completos, grupo, número de control).	5%
Menú	El menú de opciones se despliega y muestra adecuadamente.	5%
Figuras	Se permite seleccionar una figura geométrica y calcular el área y perímetro de la figura geométrica.	30%
Validación	Se valida que los datos ingresados sean correctos.	10%
Áreas y perímetros	Se calculan correctamente las áreas y perímetros de las figuras geométricas.	30%
Excepciones	Se manejan correctamente las excepciones de números negativos.	20%
Total		100%

Fecha de entrega

La fecha de entrega de la actividad es el miércoles 12 de marzo de 2025, a las 11:59 PM (media noche).

A Actividades entregadas fuera de la fecha y hora límite no serán consideradas para la evaluación.

Miembro del equipo que no se encuentre en portada no recibirá calificación.

Actividades en otro lenguaje ajeno a Java, no serán consideradas para la evaluación.

Actividad 2: Calculadora extendida

Descripción

En esta actividad, deberemos de extender las funcionalidades de la calculadora básica, agregando las siguientes funcionalidades:

- **Factorial:** Calcular el factorial de un número entero.
- **Permutación:** Calcular la permutación de dos números enteros.
- **Combinación:** Calcular la combinación de dos números enteros.

De tal razón que la calculadora deberá de tener un menú desplegable con las opciones de cálculo de las funcionalidades siguientes:

1. Suma
2. Resta
3. Multiplicación
4. División
5. Factorial
6. Permutación
7. Combinación
8. Salir

Al seleccionar una opción, se deberán solicitar los datos necesarios para realizar el cálculo de la operación seleccionada. Mismos que deberán ser validados para evitar errores en la ejecución de la calculadora.

Finalmente, se deberá mostrar el resultado del cálculo de la operación seleccionada y luego regresar al menú principal. En caso de que se seleccione una opción no válida, se deberá mostrar un mensaje de error indicando que la opción no es válida, y se deberá solicitar nuevamente la selección de la operación. En caso de que algún dato de la

operación seleccionada no sea válido, se deberá mostrar un mensaje de error indicando que el dato no es válido, y se deberá solicitar nuevamente el dato de la operación.

Requerimientos

1. Crear la funcionalidad de la calculadora extendida dentro de una función main de una clase de Java.
2. Probar la funcionalidad de la calculadora extendida. Y tomar capturas de pantalla de la ejecución de la calculadora extendida.
3. Crear una portada con el nombre de la actividad y los integrantes del equipo.
4. Adjuntar las evidencias y la portada en el proyecto de Java.
5. Comprimir el proyecto de Java en un archivo .zip y subir el archivo en la plataforma Moodle.

Criterios de evaluación

Criterio	Descripción	Puntaje
Portada	Contiene los datos de identificación de los integrantes del equipo.	5%
Menú	El menú de opciones se despliega y muestra adecuadamente.	5%
Operaciones	Se permite seleccionar una operación y calcular el resultado de la operación.	30%
Validación	Se valida que los datos ingresados sean correctos.	10%
Resultados	Se calculan correctamente los resultados de las operaciones.	30%
Excepciones	Se manejan correctamente las excepciones de números negativos.	20%
Total		100%

Fecha de entrega

La fecha de entrega de la actividad es el miércoles 19 de marzo de 2025, a las 11:59 PM (media noche).

A Actividades entregadas fuera de la fecha y hora límite no serán consideradas para la evaluación.

Miembro del equipo que no se encuentre en portada no recibirá calificación.

Actividades en otro lenguaje ajeno a Java, no serán consideradas para la evaluación.