FAMILIA PROFESIONAL:	**Informática y Comunicaciones**
CICLOS FORMATIVOS:	**Desarrollo de Aplicaciones Multiplataforma, Desarrollo de Aplicaciones Web**
MÓDULO:	**Programación**

UNIDAD 1: PROGRAMACIÓN ESTRUCTURADA

CONTENIDOS

AUTORES: **Fernando Rodríguez Alonso**
Sonia Pasamar Franco

ÍNDICE DE CONTENIDOS

1. TIPOS DE DATOS SIMPLES O ESCALARES

Un **dato** representa una información (puede ser cuantitativa o cualitativa) de una entidad del mundo real a la que se le asigna un valor que está compuesto por una secuencia de símbolos, números o letras y que se interpreta para tener un significado.

Un **tipo de datos** permite reservar espacios de la memoria del ordenador con ciertas restricciones impuestas sobre los datos, como qué valores pueden tomar y qué operaciones se pueden realizar:

- Define un conjunto de valores válidos.
- Define un conjunto específico de operaciones que se pueden realizar sobre esos valores.

Un **tipo de datos simple** o **escalar** (o básico) almacena un dato atómico e indivisible (que no se puede descomponer en datos más pequeños) y, por eso, se conoce de este tipo:

- La representación interna del tipo en memoria.
- El tamaño exacto que ocupa el tipo en memoria.
- El conjunto de operaciones disponible para los datos del tipo.

1.1. TIPOS DE DATOS PRIMITIVOS DEL LENGUAJE DE PROGRAMACIÓN

Los **tipos de datos primitivos del lenguaje de programación** son aquellos que proporciona el propio lenguaje de programación.

Cualquier programa puede utilizar un tipo de datos primitivo directamente sin la necesidad previa de importar o referenciar una librería o un paquete.

Los tipos de datos primitivos más comunes en los lenguajes de programación son el booleano, el número entero (con varias precisiones), el número real (representado en coma fija o en coma flotante), el carácter (codificado con ASCII o Unicode) y la referencia (también llamada puntero o descriptor).

1.1.1. Booleano o Lógico

El tipo **booleano** o **lógico** tiene únicamente dos valores distintos: falso y verdadero. Se utiliza para evaluar expresiones o condiciones lógicas, es decir, para distinguir la ocurrencia de un suceso entre dos posibles.

Java tiene un tipo booleano o lógico denominado **`boolean`**, con dos valores posibles (`false` y `true`) y con un tamaño de 1 byte en memoria.

1.1.2. Numéricos Enteros

El tipo **entero** representa un número entero (sin parte decimal o fraccionaria) tanto positivo como negativo.

La mayoría de los lenguajes de programación definen varios tipos enteros (al menos un tipo entero y un tipo entero largo), que se distinguen por dos características de la representación del número:

- **El tamaño (en bytes) que ocupa en memoria.** Cuantos más bytes se utilicen para representar, el rango de números enteros representable será mayor.

- **Con signo o sin signo.** Una representación con signo permite representar números positivos y negativos (repartiendo el rango de valores entre ellos), mientras que una representación sin signo únicamente permite representar números positivos (dedicando todo el rango de valores para los positivos).

Java tiene cuatro tipos enteros denominados **`byte`**, **`short`**, **`int`** y **`long`**, representados con signo (en complemento a dos) y con las siguientes características:

	`byte`	**`short`**	**`int`**	**`long`**
TAMAÑO	1 byte	2 bytes	4 bytes	8 bytes
TOTAL DE NÚMEROS DISTINTOS	$2^8 = 256$	$2^{16} = 65.536$	$2^{32} =$ 4.294.967.296	$2^{64} =$ 18.446.744.073.709.551.616
NÚMERO MÍNIMO	-128	-32.768	-2.147.483.648	-9.223.372.036.854.775.808
NÚMERO MÁXIMO	+127	+32.767	+2.147.483.647	+9.223.372.036.854.775.807

1.1.3. Numéricos Reales

El tipo **real** representa un número real (con parte decimal o fraccionaria) tanto positivo como negativo.

La mayoría de los lenguajes de programación definen más de un tipo real, que se distinguen por el método de representación del número:

- **Aritmética en Coma Flotante.** Consiste en almacenar el número en un formato equivalente a la notación científica. Es el método más utilizado y los procesadores de los ordenadores actuales suelen incorporar una Unidad de Coma Flotante (FPU) dedicada.
- **Aritmética en Coma Fija.** Consiste en almacenar el número en forma de entero escalado, fijando un número de dígitos (o de bits) para la parte fraccionaria. El rango de valores suele ser más limitado que en el método anterior, por lo que el programador debe ser cuidadoso para evitar desbordamientos.

Dado que ambos métodos definen un compromiso entre rango de valores y precisión, la precisión resultante es limitada, y por tanto, solo un subconjunto de los números reales tiene una representación exacta y el resto de números reales se representa de forma aproximada.

Cuantos más bytes se utilicen para representar, el número real será más grande en valor absoluto (tanto positivo como negativo) y tendrá mayor precisión (con más decimales exactos). Cuando no haya limitaciones serias de la memoria disponible, se recomienda usar siempre el tipo de mayor precisión, ya que así los errores cometidos al realizar sucesivos cálculos serán menores.

Java tiene dos tipos reales denominados **`float`** y **`double`**, representados según el estándar IEEE 754 para aritmética en coma flotante y con las siguientes características:

	`float`	**`double`**
TAMAÑO	4 bytes	8 bytes
TOTAL DE NÚMEROS DISTINTOS	$2^{32} =$ 4.294.967.296	$2^{64} =$ 18.446.744.073.709.551.616
PRECISIÓN	6-7 dígitos decimales	15 dígitos decimales
NÚMERO MÍNIMO	-3,4028234663852886E+38	-1,7976931348623157E+308

NÚMERO MÁXIMO	+3,4028234663852886E+38	+1,7976931348623157E+308

Para indicar desbordamientos y errores, hay tres valores especiales en coma flotante: infinito positivo, infinito negativo y NaN (no es un número).

1.1.4. Carácter

El tipo **carácter** representa un símbolo o grafema de un alfabeto o silabario utilizado para la forma escrita de un lenguaje natural. Existen diferentes tipos de caracteres: letras mayúsculas (A-Z), letras minúsculas (a-z), números arábigos (0-9), signos de puntuación, signos matemáticos, signos monetarios, caracteres de control y caracteres gráficos.

Inicialmente, los caracteres utilizados en informática fueron los determinados por el idioma inglés. Poco a poco, se introdujeron otros caracteres pertenecientes a otros idiomas (francés, alemán, castellano, etc.). Por último, en un esfuerzo global, se definió el estándar Unicode, que incluye 144.762 caracteres distribuidos en 159 conjuntos de símbolos denominados scripts (cada uno puede ser usado en uno o más sistemas de escritura).

Las **codificaciones de caracteres** más utilizadas son:

1) **ASCII.** Fue publicado inicialmente en 1967 y está basado en el alfabeto del idioma inglés. Utiliza 7 bits para representar cada carácter. Por tanto, codifica 128 caracteres diferentes.
2) **ISO/IEC 8859 (ASCII extendido).** Fue publicado inicialmente en 1987 con la intención de extender la codificación ASCII añadiendo más caracteres procedentes de otros alfabetos latinos. Utiliza 8 bits para representar cada carácter. Por tanto, codifica 256 caracteres diferentes. Aún asi, es bastante limitado para representar todos los posibles símbolos de todos los idiomas y alfabetos existentes.
3) **Unicode.** Fue publicado inicialmente entre 1991 y 1992 con el objetivo de facilitar el tratamiento informático y la transmisión y visualización de textos de numerosos idiomas y disciplinas técnicas. Define cada símbolo o carácter mediante un nombre o punto de código. Organiza los símbolos en diferentes planos y conjuntos. Contiene tres formas de codificación:
   - **UTF-8.** Codifica con un byte los caracteres definidos con ASCII y codifica con 2, 3 o 4 bytes el resto.
   - **UTF-16.** Codifica cada carácter con 2 bytes, permitiendo representar 65.536 símbolos, los cuales pertenecen al plano multilingual básico (BMP). Es compatible con ASCII.
   - **UTF-32.** Codifica cada punto de control con 4 bytes. Está pensado para los ideogramas de Asia oriental.

Java tiene un tipo carácter denominado **`char`**, que utiliza la codificación UTF-16 de Unicode y que también es considerado por el compilador como un tipo entero, ya que internamente cada carácter se almacena o codifica con un número o código.

Las operaciones disponibles para este tipo son: obtener el código Unicode de un carácter, obtener el carácter correspondiente a un código Unicode y las operaciones aritméticas definidas para números.

1.2. TIPOS DE DATOS DEFINIDOS POR EL PROGRAMADOR

Los **tipos de datos definidos por el programador** son aquellos que pueden ser definidos o declarados por el programador a partir de los tipos de datos primitivos del lenguaje de programación.

Para poder utilizar un tipo de dato definido por el programador en un programa, se necesita importar o referenciar la librería, el paquete o la clase que contiene su definición.

Los tipos de datos definidos por el programador más comunes en los lenguajes de programación son el enumerado y el subrango.

1.2.1. Enumerado

El tipo **enumerado** representa un valor constante que pertenece a un conjunto restringido y ordenado de valores de un tipo de datos primitivo. Cada valor constante tiene asociado un nombre, debe ser único (no repetido) dentro del conjunto de valores y se almacena internamente como un número entero (empezando en cero).

Varios ejemplos de tipos enumerados definidos por el programador son:

ENUMERADO	CONJUNTO DE VALORES
PuntoCardinal	{Norte, Sur, Este, Oeste}
PaloBarajaEspañola	{Oros, Copas, Espadas, Bastos}
DíaSemana	{Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo}

Java permite definir tipos enumerados mediante **`enum`**.

1.2.2. Subrango

El tipo **subrango** representa un valor variable de un tipo primitivo del lenguaje de programación (entero, real o carácter) que pertenece a un subconjunto de valores definido desde un límite inferior hasta un límite superior (ambos incluidos). Ambos límites deben definirse con el mismo tipo de datos y deben cumplir la condición de que el límite inferior sea menor que el límite superior.

Varios ejemplos de tipos subrangos definidos por el programador son:

SUBRANGO	CONJUNTO DE VALORES
Calificación	0 .. 10
LetraMayúscula	'A' .. 'Z' (sin incluir 'Ñ')
TemperaturaCorporalNormal	36.1 .. 37.2

Java no permite definir tipos subrangos.

2. ELEMENTOS DE UN PROGRAMA INFORMÁTICO

Un **programa informático** es una secuencia de instrucciones o sentencias escritas en un lenguaje de programación concreto que resuelven un problema dado en un computador. El desarrollo de un programa informático consta de los siguientes pasos:

1) Diseñar un algoritmo (normalmente con pseudocódigo) para resolver un problema planteado.
2) Codificar el algoritmo en un programa de un lenguaje de programación concreto. El programa se almacenará como un fichero de texto plano legible para el programador (código fuente).
3) Depurar el programa, es decir, compilar el programa, analizar los posibles errores sintácticos y semánticos del código fuente y editar el código fuente para corregirlos. Este paso deberá repetirse hasta que el programa no tenga errores. Como resultado de este proceso, se obtendrá un código máquina que podrá ser ejecutado en el ordenador.
4) Probar el programa en el ordenador ejecutándolo en el ordenador para todos los datos de entrada posibles del problema y verificando que los resultados obtenidos son correctos. Si el programa no funciona según todos los requerimientos del problema, se deberá volver al paso 1 o al paso 2.

2.1. SECCIONES DE UN PROGRAMA

En general, todo programa tiene varias secciones o bloques:

- **Cabecera.** Incluye la información necesaria para definir el programa y poder utilizarlo después, como el nombre del programa, los datos de entrada que necesita y los datos de salida que devuelve. Esta sección es obligatoria.
- **Declaraciones.** Es un bloque que contiene únicamente las declaraciones de todos los datos que va a usar el programa en sus instrucciones. Esta sección depende del lenguaje de programación. Algunos lenguajes (como Ada) incluyen explícitamente un bloque de declaraciones en el que se deben definir todos los datos del programa. Otros lenguajes (como Java) no tienen un bloque de declaraciones, sino que permiten definir datos dentro del cuerpo del programa, con la restricción de declararlos antes de utilizarlos.
- **Cuerpo.** Incluye la secuencia de instrucciones o sentencias, organizadas mediante diversas estructuras de control de flujo, que pueden ejecutarse en un computador para resolver un problema planteado.

2.2. PALABRAS RESERVADAS

Las **palabras reservadas** son un conjunto de palabras en minúsculas del idioma inglés que tienen un significado o función propia en el lenguaje de programación y en sus programas. Cada palabra reservada solo puede ser utilizada para lo que el lenguaje establece, y por tanto, no puede ser usada como identificador.

Las **palabras reservadas de Java** son:

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	final	finally	float
for	goto	if	implements	import

instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

Dos consideraciones:

- Las palabras `const` y `goto` no se utilizan.
- A este listado hay que añadir `true`, `false` y `null`, las cuales son palabras reservadas para ciertos literales.

2.3. IDENTIFICADORES

Un **identificador** es una secuencia de uno o más caracteres elegida por el programador para un elemento de un programa (constante, variable, atributo, método, clase o paquete). Todo identificador debe cumplir las siguientes reglas:

- El identificador debe contener un carácter como mínimo.
- Los caracteres válidos son letras mayúsculas (A-Z), letras minúsculas (a-z), dígitos numéricos (0-9), guión bajo (_) y dólar ($).
- El primer carácter no puede ser un dígito numérico (0-9).
- No se puede utilizar ninguna palabra reservada del lenguaje.

Los identificadores deben ser lo más descriptivos posibles, es decir, el nombre que se elija para un elemento de un programa debería describir el dato o la operación que representa. Es mejor usar palabras completas con significado en lugar de abreviaturas crípticas. De esta manera, el código fuente será más fácil de leer y comprender.

La **convención de nombres de Java** se resume a continuación:

ELEMENTO	CONVENCIÓN	EJEMPLOS

Variable	Comienza por letra minúscula. Si tiene más de una palabra, se colocan juntas, poniendo en cada palabra la primera letra en mayúsculas y el resto en minúsculas.	`suma`
		`numeroEmpleados`
		`mediaSalarioMensual`

Constante	Se colocan todas las palabras en mayúsculas y separadas con el guión bajo.	`PI` `TAMAGNO_MAXIMO`

Método	Comienza por letra minúscula. Si tiene más de una palabra, se colocan juntas, poniendo en cada palabra la primera letra en mayúsculas y el resto en minúsculas.	`calcular`
		`modificarSaldo`
		`obtenerSaldo`

Clase e Interfaz	Comienza por letra mayúscula. Si tiene más de una palabra, se colocan juntas, poniendo en cada palabra la primera letra en mayúsculas y el resto en minúsculas.	`Fecha`
		`ProgramadorJunior`
		`ProgramadorSenior`

Paquete	Se colocan todas las palabras en minúsculas y sin espacios entre las palabras.	`graficos` `entradasalida`

Java distingue entre mayúsculas y minúsculas. Por ejemplo, los identificadores `suma`, `Suma` y `SUMA` son distintos. Además, la longitud de un identificador es prácticamente ilimitada.

2.4. VARIABLES Y CONSTANTES

Tanto una **variable** como una **constante** son espacios de la memoria del ordenador a las que se asigna un nombre o identificador (a modo de etiqueta) y a las que se asocia un tipo de datos, de forma que sólo se podrán almacenar en esa zona o posición de memoria valores correspondientes a ese tipo de datos.

La **diferencia entre una variable y una constante** es:

- El valor de una variable se puede modificar tantas veces como se desee a lo largo del programa. Se permite almacenar diferentes valores en una variable durante la vida del programa.
- El valor de una constante sólo se puede modificar una vez (normalmente, al principio del programa) y después de asignarlo no se podrá volver a modificarlo durante el programa y el valor permanecerá constante.

Además, aparecen dos conceptos nuevos:

- **Declarar** una variable o constante de un tipo de datos específico consiste en reservar espacio en la memoria del ordenador para almacenar un valor correspondiente a dicho tipo de datos.
- **Inicializar** una variable o constante consiste en asignar por primera vez un valor válido (según el tipo de datos con el que se ha definido la variable o constante) en la zona de memoria reservada para dicha variable o constante.

Los siguientes ejemplos muestran declaraciones e inicializaciones de variables y constantes en Java:

DECLARACIÓN DE VARIABLES	**INICIALIZACIÓN DE VARIABLES**

`boolean encontrado;`
`int contador;`
`int i, j, k;`
`double precio;`
`char opcion;`
	`boolean encontrado = false;`
	`int contador = 0;`
	`int i = 1, j = 2, k = 3;`
	`double precio = 25.95;`
	`char opcion = 'a';`

DECLARACIÓN DE CONSTANTES	**INICIALIZACIÓN DE CONSTANTES**
`final int TAMAGNO_MAXIMO;` `final double PI;`	`final int TAMAGNO_MAXIMO = 30;` `final double PI = 3.14159265359;`

2.5. LITERALES

Un **literal** es un valor de un tipo de datos primitivo del lenguaje de programación.

Los **literales de Java** se corresponden con los tipos booleano (`boolean`), entero (`int` y `long`), real (`float` y `double`) y carácter (`char`) y se incluyen también literales de cadenas de caracteres y un literal específico para referencias nulas (`null`).

2.5.1. Literales Booleanos

Los **literales de tipo booleano de Java** son `false` (falso) y `true` (verdadero).

2.5.2. Literales Enteros

Los **literales de tipo entero de Java** se definen por defecto con el tipo `int`, excepto si su valor absoluto es mayor que el valor máximo de un `int` o si el valor termina con el sufijo `L` o `l`, en cuyo caso será del tipo `long`.

Al principio del literal, el signo `+` es opcional si el número es positivo y el signo `-` es obligatorio si el número es negativo.

Los literales de tipo entero de Java se pueden expresar en **cuatro bases**:

- **Decimal (base 10).** Está formado por uno o más digitos decimales (0-9). El primer dígito debe ser distinto de cero.
- **Binario (base 2).** Empieza siempre con `0b` o `0B`, seguido de uno o más dígitos binarios (0-1).
- **Octal (base 8).** Empieza siempre con `0`, seguido de uno o más dígitos octales (0-7).
- **Hexadecimal (base 16).** Empieza siempre con `0x` o `0X`, seguido de uno o más dígitos hexadecimales (0-9 y A-F o a-f).

Los literales de tipo entero de cualquier base pueden contener el carácter _ para facilitar la lectura del número. Este separador sólo puede aparecer entre dígitos.

Los siguientes ejemplos muestran literales de tipo entero de Java:

`int` (decimal)	`-25`
	`+100`
	`1500`
		`long`	`1234L`
			`10000001`
			`888800000000`

`int` (binario)	`0b001001` `0B000111001101`		
`int` (octal)	`025` `01234`		

`int` (hexadecimal)	`0x1A2C`
	`0x430`
	`0X00be`		

2.5.3. Literales Reales

Los **literales de tipo real de Java** se definen por defecto con el tipo `double`. Si el valor termina con el sufijo `F` o `f`, será del tipo `float`. Si el valor termina con el sufijo `D` o `d`, será del tipo `double`.

Al principio del literal, el signo `+` es opcional si el número es positivo y el signo `-` es obligatorio si el número es negativo.

Los literales de tipo real de Java se pueden expresar en **dos notaciones**:

- **Coma Decimal.** El número se expresa en base 10 (decimal) y está compuesto por una parte entera, un punto decimal y una parte fraccionaria. Si la parte entera es cero, se puede omitir.
- **Notación Científica.** El número se exprese en base 10 (decimal) y está compuesto por una mantisa, la letra `E` o `e` y el exponente (positivo o negativo y con dígitos 0-9).

Los siguientes ejemplos muestran literales de tipo real de Java:

float	6.42F 6.42f -1.39e-2F 58.07E12f	**double**	5.79 5.79D 5.79d -2.15E2

2.5.4. Literales de Caracteres

Los **literales de caracteres de Java** se definen con el tipo `char` y un único carácter encerrado entre comillas simples.

Los literales se pueden expresar de dos maneras:

- **Carácter / Secuencia de Escape.** La mayoría de caracteres latinos se escriben directamente. Por ejemplo: `'A'`, `'B'`, `'C'`, `'é'`, `'í'`, `'ó'`, `'x'`, `'y'`, `'z'`, `'Ñ'`, `'ñ'`, `'ü'`, `'0'`, `'1'`, `'2'`, `'.'`, `','`, `'?'`, `'!'`, `'+'`, `'-'`, `'*'`, `'/'`, `'$'`, etc. Sin embargo, existen unos caracteres especiales que se deben indicar con secuencias de escape:

CARÁCTER	SECUENCIA DE ESCAPE	CARÁCTER	SECUENCIA DE ESCAPE
Retorno de Carro	`'\r'`	Salto de Página	`'\f'`
Salto de Línea	`'\n'`	Comillas Simples	`'\''`
Tabulador	`'\t'`	Comillas Dobles	`'\"'`
Retroceso	`'\b'`	Contrabarra	`'\\'`

- **Notación Unicode.** Todos los caracteres se pueden escribir en notación UTF-16 de Unicode, desde `'\u0000'` hasta `'\uFFFF'`, con 4 dígitos hexadecimales (0-9 y A-F o a-f). El prefijo `\u` indica un valor Unicode y los 4 dígitos hexadecimales identifican el carácter o punto de código correspondiente.

2.5.5. Literales de Cadenas de Caracteres

Los **literales de cadenas de caracteres de Java** son objetos de la clase `String` y están formados por cero o más caracteres encerrados entre comillas dobles. Una cadena de caracteres también puede incluir secuencias de escape y caracteres expresados en notación Unicode.

Los siguientes ejemplos muestran literales de cadenas de caracteres de Java:

`""`
`"El partido de fútbol se ha suspendido."`
`"Juan dijo: \"Hoy hace un día fantastico.\""`
`"Pulsa \'C\' para continuar."`
`"¿Desea salir del programa (S/N)?"`

2.6. OPERADORES Y EXPRESIONES

Los **operadores** realizan operaciones sobre un conjunto de datos u operandos, representados por literales y/o identificadores de diversos tipos de datos, y devuelven un resultado del mismo u otro tipo de

datos. Los operadores se clasifican en unarios, binarios o ternarios, según el número de operandos que tengan (uno, dos o tres, respectivamente).

Una **expresión** es una combinación de operadores, literales e identificadores que se evalúan para calcular o producir un único resultado de un tipo de datos determinado. El resultado de una expresión se puede utilizar como parte de otra expresión o en una instrucción o sentencia.

Una expresión se puede definir también de forma recursiva:

1) Todo literal de un determinado tipo de datos es también una expresión de ese mismo tipo de datos.
2) Todo identificador de un determinado tipo de datos (variable o constante) es también una expresión de ese mismo tipo de datos.
3) Dado un operador *n*-ario que requiere *n* operandos (expresiones) de los tipos t_1, t_2, …, t_n y dados n operandos de los tipos de datos correspondientes, el resultado es una expresión del tipo de datos generado por el operador.

2.6.1. Operadores Booleanos o Lógicos

Las principales **operaciones booleanas** son:

- **Negación (not).** Tiene un operando. Devuelve el valor contrario al operando.
- **Conjunción (and).** Tiene dos operandos. Devuelve verdadero sólo cuando los dos operandos son verdaderos. En otros casos, devuelve falso.
- **Disyunción (or).** Tiene dos operandos. Devuelve verdadero si alguno de los dos operandos es verdadero. En otro caso, devuelve falso.

Las tablas de verdad de estas operaciones son:

A	not A
verdadero	falso
falso	verdadero

A	B	A and B
verdadero	verdadero	verdadero
verdadero	falso	falso
falso	verdadero	falso
falso	falso	falso

A	B	A or B
verdadero	verdadero	verdadero
verdadero	falso	verdadero
falso	verdadero	verdadero
falso	falso	falso

Los operadores booleanos o lógicos realizan operaciones sobre valores booleanos o resultados de expresiones relacionales y generan como resultado un valor booleano.

Los **operadores booleanos de Java** son:

- **Negación (!).** Evalúa el operando y devuelve la negación del operando.
- **Conjunción (&).** Evalúa los dos operandos y devuelve la conjunción de los operandos.
- **Conjunción con Evaluación Cortocircuitada o Condicional (&&).** Evalúa el primer operando (izquierda). Si es `false`, no evalúa el segundo y devuelve `false`. Si es `true`, evalúa el segundo operando (derecha) y devuelve el resultado del segundo operando.
- **Disyunción (|).** Evalúa los dos operandos y devuelve la disyunción de los operandos.
- **Disyunción con Evaluación Cortocircuitada o Condicional (||).** Evalúa el primer operando (izquierda). Si es `true`, no evalúa el segundo y devuelve `true`. Si es `false`, evalúa el segundo operando (derecha) y devuelve el resultado del segundo operando.

2.6.2. Operadores Aritméticos

Los operadores aritméticos realizan operaciones sobre valores numéricos (enteros o reales) o resultados de expresiones aritméticas y generan como resultado un valor numérico (entero o real).

Los **operadores aritméticos de Java** son:

OPERADOR	NOMBRE	TIPOS DE DATOS	EJEMPLO
`- +`	Cambio de Signo	enteros y reales	`-10`
`+`	Suma o Adición	enteros y reales	`1.2 + 9.3`
`-`	Resta o Sustracción	enteros y reales	`312.5 – 12.3`
`*`	Producto o Multiplicación	enteros y reales	`1.7 * 3.8`
`/`	División Real	reales	`6.4 / 2.1`
`/`	División Entera	enteros	`10 / 4`
`%`	Resto/Módulo de la División Entera	enteros	`10 % 4`

Cuando un operador aritmético opera sobre dos operandos de diferentes tipos, estos operandos se convierten a un mismo tipo común antes de realizar la operación. Java realiza esta conversión automática según las siguientes **reglas de conversión entre tipos numéricos**:

- Si alguno de los operandos es de tipo `double`, el otro operando será convertido a tipo `double`.
- En caso contrario, si alguno de los operandos es de tipo `float`, el otro operando será convertido a tipo `float`.
- En caso contrario, si alguno de los operandos es de tipo `long`, el otro operando será convertido a tipo `long`.
- En caso contrario, los dos operandos serán convertidos a tipo `int`.

Los **operadores de incremento y decremento de Java** son:

OPERADOR	NOMBRE	EJEMPLO	COMENTARIOS
`++`	Incremento en Notación Prefija	`x = 3;` `y = ++x;`	x vale 4 y vale 4
	Incremento en Notación Postfija	`x = 3;` `y = x++;`	x vale 4 y vale 3
`--`	Decremento en Notación Prefija	`x = 3;` `y = --x;`	x vale 2 y vale 2
	Decremento en Notación Postfija	`x = 3;` `y = x--;`	x vale 2 y vale 3

2.6.3. Operadores Relacionales

Los operadores relacionales se utilizan para comparar datos de tipos primitivos (booleano, entero, real y carácter) y generan como resultado un valor booleano.

Los **operadores relacionales de Java** son:

OPERADOR	NOMBRE	EJEMPLO
`==`	Igual a	`a == b`
`!=`	Distinto de	`a != b`
`<`	Menor que	`a < b`

<=	Menor o Igual que	a <= b
>	Mayor que	a > b
>=	Mayor o Igual que	a >= b

2.6.4. Operadores de Asignación

Los operadores de asignación se utilizan para asignar el resultado de una expresión (que puede ser un literal, una variable o constante, o una composición de expresiones más simples) a una variable.

El tipo de datos del resultado de la expresión debe ser compatible con el tipo de datos de la variable a la que se asigna dicho resultado.

Los **operadores de asignación de Java** son:

OPERADOR	NOMBRE	EJEMPLO	EXPRESIÓN EQUIVALENTE
=	Asignación	a = b;	
+=	Suma y Asignación	a += b;	a = a + b;
-=	Resta y Asignación	a -= b;	a = a - b;
*=	Multiplicación y Asignación	a *= b;	a = a * b;
/=	División y Asignación	a /= b;	a = a / b;
%=	Resto/Módulo y Asignación	a %= b;	a = a % b;

2.6.5. Operador de Conversión Explícita de Tipos

El **operador de conversión explicita de tipos de Java** se denomina **`cast`** y se debe utilizar para realizar una conversión de un tipo representado con más bytes a otro tipo representado con menos bytes. Es posible que haya una pérdida de información durante la conversión dependiendo del valor.

Dos ejemplos de conversión explícita de tipos de Java son:

CONVERSIÓN EXPLÍCITA DE TIPOS	COMENTARIOS
`int a = 1000;` `short b = (short) a;`	Se realiza una conversión de `int` a `short`. El valor entero `1000` está incluido en el rango de valores representables por un dato de tipo `short` (desde -32.768 hasta +32.767). No hay pérdida de información.
`short a = 500;` `byte b = (byte) a;`	Se realiza una conversión de `short` a `byte`. El valor entero corto `500` no está incluido en el rango de valores representables por un dato de tipo `byte` (desde -127 hasta +128). Hay pérdida de información.

No se pueden realizar conversiones entre un valor booleano y cualquier otro tipo numérico.

Las conversiones de datos de tipo real a tipo entero siempre requieren el uso de este operador `cast`. En estos casos, la posible pérdida de información puede implicar una pérdida de precisión.

2.6.6. Precedencia y Asociatividad de Operadores

La **precedencia** de los operadores determina el orden o la secuencia en que deben realizarse las operaciones cuando en una expresión intervienen operadores de distintos tipos.

En general, para las expresiones aritméticas o matemáticas, los lenguajes de programación siguen las reglas del álgebra convencional:

- La multiplicación, la división y el resto/módulo se evalúan primero. Si hay varias operaciones de multiplicación, división o resto/módulo dentro de una expresión, se evalúan de izquierda a derecha.
- La suma y la resta se evalúan después de las anteriores. Si hay varias operaciones de suma y resta dentro de una expresión, se evalúan de izquierda a derecha.

La **asociatividad** de los operadores indica qué operador se evalúa antes en condiciones de igualdad de precedencia:

- Los operadores de asignación, los operadores aritméticos de incremento y decremento y la conversión explícita de tipos numéricos (cast) son asociativos por la **derecha**. Se evalúan de derecha a izquierda.
- El resto de operadores (booleanos, aritméticos y relacionales) son asociativos por la **izquierda**. Se evalúan de izquierda a derecha.

La **precedencia y asociatividad de los operadores de Java** se resumen a continuación:

OPERADORES	TIPO (CATEGORÍA Y OPERANDOS)		ASOCIATIVIDAD
`++ --`	Incremento/Decremento Postfijos	Unario	de Derecha a Izquierda

`!`
`++ --`
`+ -`
`(cast)`
	Booleano
	Incremento/Decremento Prefijos
	Aritméticos
	Conversión Explícita de Tipos
		Unario	de Derecha a Izquierda

`* / %`	Aritméticos	Binario	de Izquierda a Derecha
`+ -`	Aritméticos	Binario	de Izquierda a Derecha
`< <= > >=`	Relacionales	Binario	de Izquierda a Derecha
`== !=`	Relacionales	Binario	de Izquierda a Derecha
`&`	Booleano	Binario	de Izquierda a Derecha
`|`	Booleano	Binario	de Izquierda a Derecha
`&&`	Booleano	Binario	de Izquierda a Derecha
`||`	Booleano	Binario	de Izquierda a Derecha
`= += -= *= /= %=`	Asignación	Binario	de Derecha a Izquierda

2.7. INSTRUCCIONES

Una **instrucción** o **sentencia** de un lenguaje de programación representa una acción completa de una programa y que el ordenador es capaz de entender y ejecutar. Todo programa está compuesto por un un conjunto de instrucciones o sentencias que están secuenciadas con una estructura.

Existen tres tipos de instrucciones o sentencias:

1) **Instrucción de Declaración.** Se utiliza para declarar o definir una variable o constante (también un objeto) en el programa, asociando un tipo de datos al identificador de la variable o constante, y opcionalmente un valor. Esta instrucción debe finalizar con un símbolo de terminación (`;`).

TIPO DE INSTRUCCIÓN DE DECLARACIÓN	EJEMPLOS

Declaración de una Variable	`boolean b;`
	`int i;`
	`double d;`

Declaración e Inicialización de una Variable	`boolean b = false;`
	`int i = 0;`
	`double d = 0.0;`

Declaración de varias Variables del mismo tipo de datos	`int a, b, c;` `double x, y, z;`
Declaración e Inicialización de varias Variables del mismo tipo de datos	`int a = 1, b = 2, c = 3;` `double x = 24, y = 25, z = 26;`

2) **Instrucción de Expresión.** Está formada por una expresión que, al ser evaluada, produce un resultado o efecto en el programa. Esta instrucción debe finalizar con un símbolo de terminación (;).

TIPO DE INSTRUCCIÓN DE EXPRESIÓN	EJEMPLOS

Asignación de una Variable	`a = 3;`
	`b = (x + y) * 2;`
	`c += 5.0;`

Incremento o Decremento de una Variable	`a++;`
	`b--;`
	`c = ++y;`

Invocación o Llamada a un Subprograma (Función o Procedimiento)	`potencia = elevar(2, 10);` `escribirTablaMultiplicar(5);`
Creación de un Objeto de una Clase	`s = new Scanner(System.in);` `p = new Persona(1, "Juan Sánchez");`

3) **Instrucción o Estructura de Control de Flujo.** Es una estructura, bloque o contenedor que incluye instrucciones del programa. Se utiliza para establecer el orden en el que se van a ejecutar esas instrucciones y bajo qué condiciones. Esta instrucción o estructura debe empezar con un símbolo de inicio de bloque ({) y debe terminar con un símbolo de fin de bloque (}).

2.8. ESTRUCTURAS DE CONTROL DE FLUJO

Un programa informático se define como una secuencia ordenada de instrucciones, dedicadas a ejecutar una tarea en un ordenador. Debido a esto aparece el concepto de **flujo de ejecución de un programa**, que indica el orden que siguen las sentencias durante la ejecución del mismo.

El flujo de ejecución de un programa viene determinado por una serie de patrones o estructuras de programación. Cada **estructura de programación** se comporta exteriormente como una instrucción o sentencia única, de forma que se puede concatenar después o incluir dentro de otra estructura y así componer el flujo de ejecución de un programa completo. Estas estructuras de programación son independientes del lenguaje de programación que se esté usando, se pueden aplicar a cualquiera de los que existen en la actualidad y son las siguientes **sentencias de control de flujo**:

1) **Secuencial.** Está compuesta por 0, 1 o N instrucciones, sentencias o estructuras que se ejecutan en el orden en que han sido escritas.

   Es la estructura más sencilla y sobre la que se construirán el resto de estructuras.

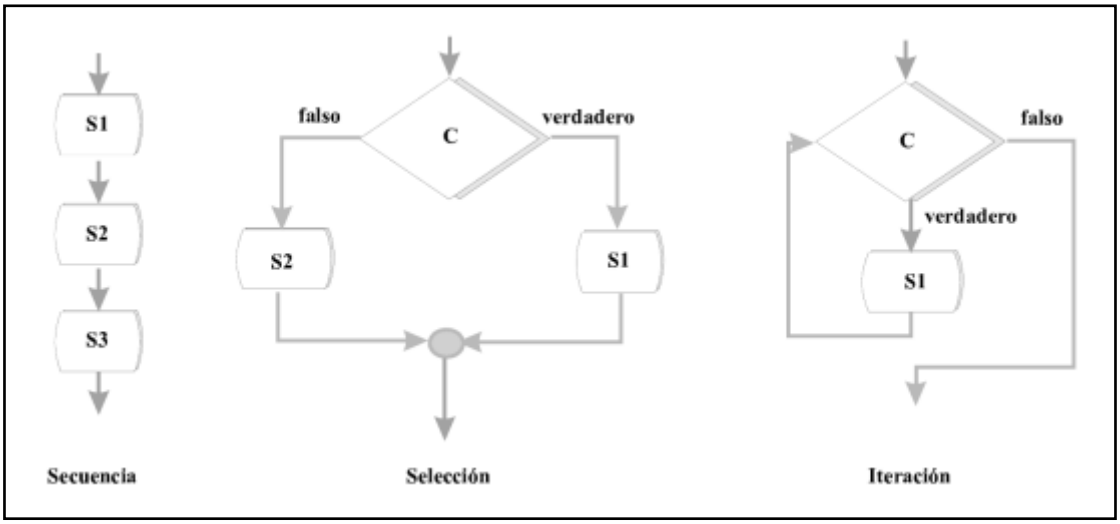
2) **Condicional, Selectiva o Alternativa.** Consta de una expresión especial de decisión y de un conjunto de secuencias de instrucciones. Se evalúa la expresión de decisión, que generará un resultado (que puede ser de tipo booleano o entero) y en función de éste, se ejecutará una secuencia de instrucciones u otra.

   Las estructuras de selección pueden ser simples, dobles y múltiples.

3) **Repetitiva, Iterativa o Cíclica.** Consta de una expresión especial de decisión y de una secuencia de instrucciones. La expresión de decisión generará un resultado de tipo booleano (verdadero o falso). Si la expresión de decisión genera un resultado verdadero, se ejecutará la secuencia de instrucciones y después se volverá a evaluar la expresión de decisión. Si la expresión de decisión genera un resultado falso, finalizará la ejecución de la estructura de repetición.

   Las estructuras iterativas pueden evaluar la expresión de decisión antes o después de ejecutar la secuencia de instrucciones y, en algunos casos, especificar el número de iteraciones a realizar.

A continuación, se muestran tres ejemplos: estructura secuencial, estructura selectiva doble y estructura iterativa con expresión antes de la secuencia de instrucciones.



2.8.1. Estructuras Condicionales, Selectivas o Alternativas

Java tiene las sentencias **`if`** y **`switch`** para establecer estructuras de selección.

Para establecer estructuras de selección sencillas, se utiliza la estructura `if` o `if-else`.

SELECCIÓN SIMPLE (if)	SELECCIÓN DOBLE (if-else)

`if (expresión_booleana) {`
`  instrucciones_si_expresión_verdadera`
`}`
	`if (expresión_booleana) {`
	`  instrucciones_si_expresión_verdadera`
	`}`
	`else {`
	`  instrucciones_si_expresión_falsa`
	`}`

Para establecer estructuras de selección complejas, hay dos opciones:

SELECCIÓN MÚLTIPLE (if)	SELECCIÓN MÚLTIPLE (switch)

`if (expresión_booleana_1) {`
`  instrucciones_si_expresión1_verdadera`
`}`
`else if (expresión_booleana_2) {`
`  instrucciones_si_expresión2_verdadera`
`}`
`else if (expresión_booleana_3) {`
`  instrucciones_si_expresión3_verdadera`
`}`
`...`
`else {`
`  instrucciones_si_expresiones_falsas`
`}`
	`switch (expresión_entera) {`
	`  case valor1:`
	`  instrucciones_si_expresión_igual_valor1`
	`  break;`
	`  case valor2:`
	`  instrucciones_si_expresión_igual_valor2`
	`  break;`
	`  case valor3:`
	`  instrucciones_si_expresión_igual_valor3`
	`  break;`
	`  ...`
	`  default:`
	`  instrucciones_si_expresion_distinta_valores`
	`}`

2.8.2. Estructuras Repetitivas, Iterativas o Cíclicas

Java tiene las sentencias **`while`** y **`do-while`** para establecer estructuras repetitivas en las que la ejecución de un bloque de instrucciones dependa del resultado de una expresión booleana.

REPETICIÓN CON EXPRESIÓN ANTES DE INSTRUCCIONES (while)	REPETICIÓN CON EXPRESIÓN DESPUÉS DE INSTRUCCIONES (do-while)

`while (expresión_booleana) {`
`  instrucciones_a_repetir`
`}`
	`do {`
	`  instrucciones_a_repetir`
	`}`
	`while (expresión_booleana);`

Java tiene la sentencia **`for`** para establecer estructuras repetitivas cuyo número de iteraciones se conoce o se puede calcular.

REPETICIÓN CON NÚMERO DE ITERACIONES CONOCIDO (for)	`for (inicialización ; expresión_booleana ; actualización) {`
	`  instrucciones_a_repetir`
	`}`

EJEMPLO	`for (numero = 1 ; numero <= 10 ; numero++ ) {`
	`  instrucciones_a_repetir`
	`}`

2.9. COMENTARIOS

Los **comentarios** son muy importantes a la hora de describir qué hace un determinado programa, pues facilitan la lectura y el seguimiento del código fuente. Todos los lenguajes de programación disponen de alguna forma de introducir comentarios en el código. En el caso de Java, existen los siguientes tipos de comentarios:

1) **Comentario de una Única Línea.** Comienza con el delimitador `//` .

```
// Esto es un comentario de una única línea.
```

2) **Comentario de Múltiples Líneas.** Comienza con el delimitador `/*` al principio del párrafo en la primera línea y termina con el delimitador `*/` al final del mismo en la última línea.

```
/* Esto es un comentario de múltiples líneas.
   Por ejemplo, un párrafo descriptivo de varias líneas. */
```

3) **Comentario Javadoc.** Utiliza los delimitadores `/**` y `*/`, al principio y al final del párrafo, respectivamente. Este comentario se emplea para generar la documentación automática del programa. A través del programa javadoc, incluido en JavaSE, se recogen todos estos comentarios y se llevan a un documento en formato HTML.

```
/** Esto es un comentario de documentación.
    Javadoc extrae los comentarios del código fuente y
    genera un archivo HTML a partir de estos comentarios.
*/
```

3. CASOS PRÁCTICOS

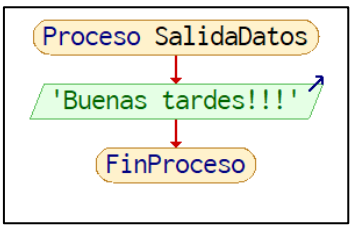
3.1. INSTRUCCIONES DE SALIDA DE DATOS POR CONSOLA

El siguiente ejemplo muestra la salida de un mensaje de texto con salto de línea al final.

Pseudocódigo en PSeInt:

```
Proceso SalidaDatos
    Escribir "Buenas tardes!!!";
FinProceso
```

Ordinograma en PSeInt:



Java:

```
public class SalidaDatos {

    public static void main(String[] args) {
          System.out.println("Buenas tardes!!!");
    }

}
```

En Java, al utilizar **System.out.println** se imprime un salto de línea a continuación del texto. Para escribir un texto sin que pase a la siguiente línea se utiliza en su lugar **System.out.print**.

Además, es posible construir una salida de información indicando su formato con **System.out.printf** creando máscaras de la siguiente manera:

%[ancho][.precision]tipo de dato

% se indica siempre

ancho sirve para indicar la cantidad de espacios que se ocupan. Es opcional.

precisión sirve para indicar la cantidad de números decimales. Es opcional.

tipo de dato se toma de los que aparecen a continuación, según el tipo de dato a representar:

- **b** – booleano
- **s** – cadena
- **c** – carácter Unicode
- **d** – entero en formato decimal
- **f** – real en formato decimal
- **e** – real con notación científica
- **t** – fecha u hora

Ejemplo de uso:

```
int numeroEnteroA, numeroEnteroB;
numeroEnteroA = 5;
numeroEnteroB = 7;
System.out.println("NÚMEROS ENTEROS");
System.out.printf("%d\n", numeroEnteroA);
//dos números enteros separados por un espacio:
System.out.printf("%d %d\n", numeroEnteroA, numeroEnteroB);
//3 posiciones de ancho:
System.out.printf("%3d\n", numeroEnteroA);
//3 posiciones de ancho y completado con ceros:
System.out.printf("%03d\n", numeroEnteroA);

double numeroReal = 6.27;
System.out.println("\nNÚMEROS REALES");
//sin decir la cantidad de decimales:
System.out.printf("%f\n", numeroReal);
//2 decimales:
System.out.printf("%.2f\n", numeroReal);
//5 posiciones de ancho y dos decimales:
System.out.printf("%5.2f\n", numeroReal);
//5 posiciones de ancho y dos decimales y completado con ceros:
System.out.printf("%05.2f\n", numeroReal);
```

Salida:

```
NÚMEROS ENTEROS
5
5 7
  5
005

NÚMEROS REALES
6,270000
6,27
 6,27
06,27
```

3.2. INSTRUCCIONES DE ENTRADA DE DATOS A TECLADO

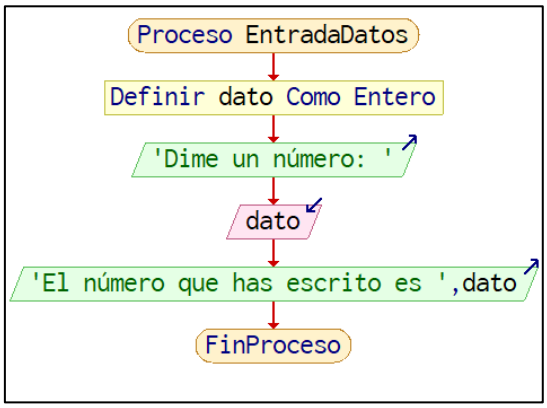
El siguiente ejemplo muestra la solicitud de un dato por teclado.

En primer lugar, se muestra un mensaje informando al usuario de lo que tiene que escribir. A continuación, se recupera el dato que el usuario introduzca. Finalmente, se muestra un mensaje por consola que contiene el dato que indicó el usuario.

Pseudocódigo en PSeInt:

```
Proceso EntradaDatos
    Definir dato como entero;
    Escribir "Dime un número: ";
    Leer dato;
    Escribir "El número que has escrito es ", dato;
FinProceso
```

Ordinograma en PSeInt:



Java:

```
import java.util.Scanner;

public class EntradaDatos {

    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        int dato;
        System.out.println("Dime un número: ");
        dato = teclado.nextInt();
        System.out.println("El número que has escrito es: " + dato);
    }

}
```

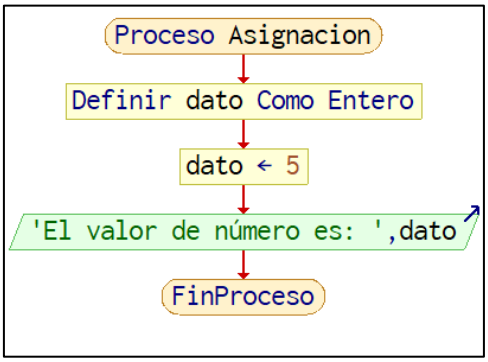
3.3. INSTRUCCIÓN DE ASIGNACIÓN

El siguiente ejemplo muestra la asignación de un valor a una variable. Tras declarar la variable y una vez realizada la asignación, se muestra por consola un mensaje que contiene el dato almacenado en la variable.

Pseudocódigo en PSeInt:

```
Proceso Asignacion
    Definir dato Como Entero;
    dato ← 5;
    Escribir "El valor de número es: ", dato;
FinProceso
```

Ordinograma en PSeInt:



Java:

```
public class Asignacion {

    public static void main(String[] args) {
        int dato;
        dato = 5;
        System.out.println("El valor del número es: " + dato);

    }

}
```

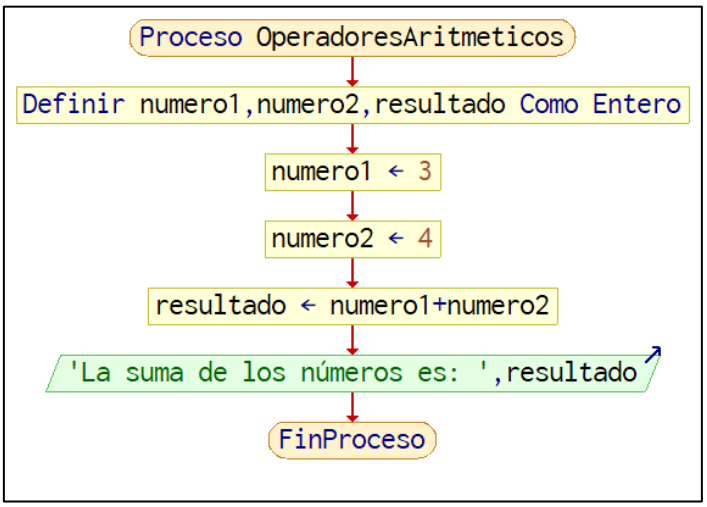
3.4. OPERADORES ARITMÉTICOS

El siguiente ejemplo muestra la operación aritmética suma. Tras declarar variables para almacenar los datos necesarios y asignar valor a dos de ellas, se realiza la operación cuyo resultado se almacenará en otra variable. Finalmente, se muestra por consola un mensaje que contiene el dato almacenado en la variable con el resultado de la operación.

Pseudocódigo en PSeInt:

```
Proceso OperadoresAritmeticos
    Definir numero1, numero2, resultado Como Entero;
    numero1 ← 3;
    numero2 ← 4;
    resultado ← numero1 + numero2;
    Escribir "La suma de los números es: ", resultado;
FinProceso
```

Ordinograma en PSeInt:



Java:

```java
public class OperadoresAritmeticos {

    public static void main(String[] args) {
        int numero1, numero2, resultado;
        numero1 = 3;
        numero2 = 4;
        resultado = numero1 + numero2;
        System.out.println("La suma de los número es " + resultado);
    }

}
```

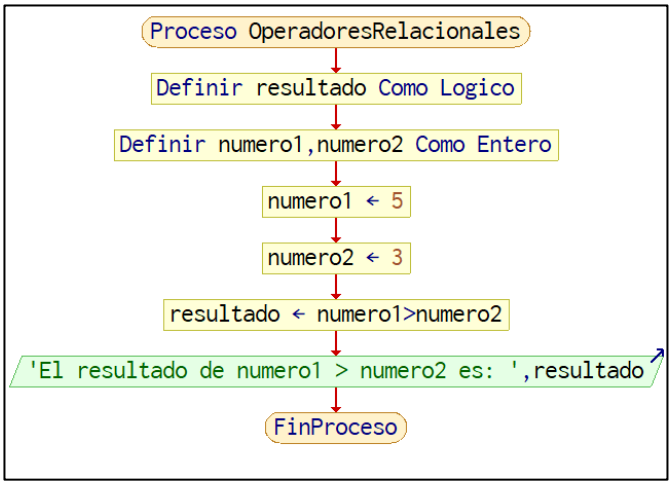
3.5. OPERADORES RELACIONALES

El siguiente ejemplo muestra el uso del operador relacional ***mayor que***. Tras declarar variables para almacenar los datos necesarios y asignar valor a dos de ellas, se realiza la operación cuyo resultado se almacenará en otra variable. Finalmente, se muestra por consola un mensaje que contiene el dato almacenado en la variable con el resultado de la operación.

Pseudocódigo en PSeInt:

```
Proceso OperadoresRelacionales
    Definir resultado Como Logico;
    Definir numero1, numero2 Como Entero;
    numero1 ← 5;
    numero2 ← 3;
    resultado ← numero1 > numero2;
    Escribir "El resultado de numero1 > numero2 es: ", resultado;
FinProceso
```

Ordinograma en PSeInt:



Java:

```java
public class OperadoresRelacionales {

    public static void main(String[] args) {
        boolean resultado;
        int numero1, numero2;
        numero1 = 5;
        numero2 = 3;
        resultado = numero1 > numero2;
        System.out.println("El resultado de numero1 > numero2 es: " + resultado);
    }

}
```

3.6. OPERADORES LÓGICOS O BOOLEANOS

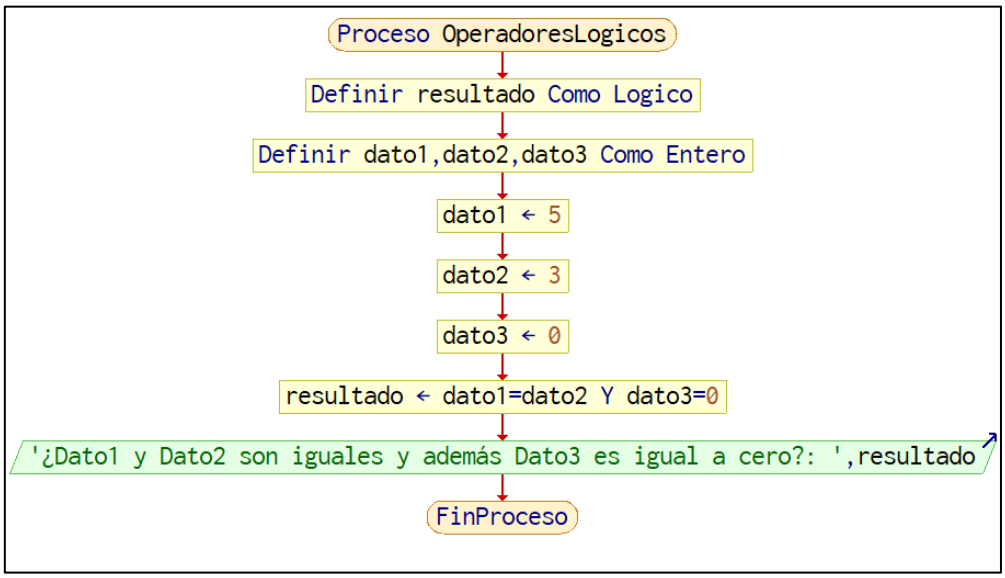
El siguiente ejemplo muestra el uso del operador booleano ***and***. Tras declarar variables para almacenar los datos necesarios y asignar valor a tres de ellas, se realiza la operación cuyo resultado se almacenará en otra variable. Esta operación consiste en evaluar dos condiciones, cuyo resultado es un booleano y aplicar el operador and al resultado de estas dos condiciones.

Finalmente, se muestra por consola un mensaje que contiene el dato almacenado en la variable con el resultado de la operación.

Pseudocódigo en PSeInt:

```
Proceso OperadoresLogicos
    Definir resultado Como Logico;
    Definir dato1, dato2, dato3 Como Entero;
    dato1 ← 5;
    dato2 ← 3;
    dato3 ← 0;
    resultado ← dato1 = dato2 y dato3 = 0;
    Escribir "¿Dato1 y Dato2 son iguales y además Dato3 es igual a cero?: ", resultado;
FinProceso
```

Ordinograma en PSeInt:



Java:

```java
public class OperadoresLogicos {

    public static void main(String[] args) {
        boolean resultado;
        int dato1, dato2, dato3;
        dato1 = 5;
        dato2 = 3;
        dato3 = 0;
        resultado = (dato1 == dato2) && (dato3 == 0);
        System.out.println("¿Dato1 y Dato2 son iguales y además Dato3 es igual a cero?: " + resultado);
    }

}
```

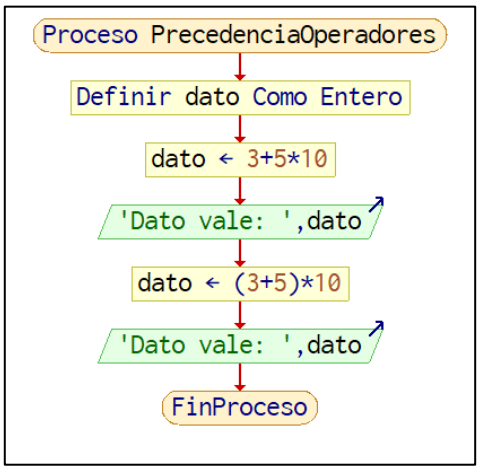
3.7. PRECEDENCIA DE OPERADORES

El siguiente ejemplo muestra los diferentes resultados que se pueden obtener según cómo se indiquen los operadores.

Pseudocódigo en PSeInt:

```
Proceso PrecedenciaOperadores
    Definir dato Como Entero;
    dato ← 3 + 5 * 10;
    Escribir "Dato vale: ", dato; //53
    dato ← (3 + 5) * 10;
    Escribir "Dato vale: ", dato; //80
FinProceso
```

Ordinograma en PSeInt:



Java:

```java
public class PrecedenciaOperadores {

    public static void main(String[] args) {
        int dato;
        dato = 3 + 5 * 10;
        System.out.println("Dato vale " + dato); //53
        dato = (3 + 5) * 10;
        System.out.println("Dato vale " + dato); //80
    }

}
```

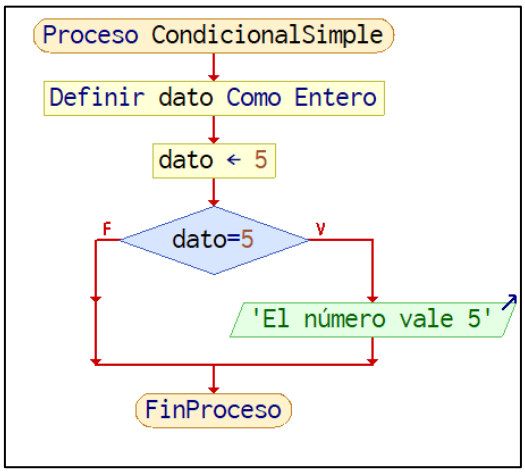
3.8. ESTRUCTURA CONDICIONAL SIMPLE

El siguiente ejemplo muestra el uso de una condicional simple. Tras declarar una variable y asignarle un valor, se verifica si cumple una condición (que sea igual a 5), y en caso afirmativo se muestra un mensaje por consola. En caso de que no se cumpla la condición, no se realizará ninguna acción alternativa.

Pseudocódigo en PSeInt:

```
Proceso CondicionalSimple
    Definir dato Como Entero;
    dato ← 5;
    Si dato = 5 Entonces
        Escribir "El número vale 5";
    FinSi
FinProceso
```

Ordinograma en PSeInt:



Java:

```java
public class CondicionalSimple {

    public static void main(String[] args) {
        int dato;
        dato = 5;
        if(dato == 5) {
            System.out.println("El número vale 5");
        }
    }

}
```

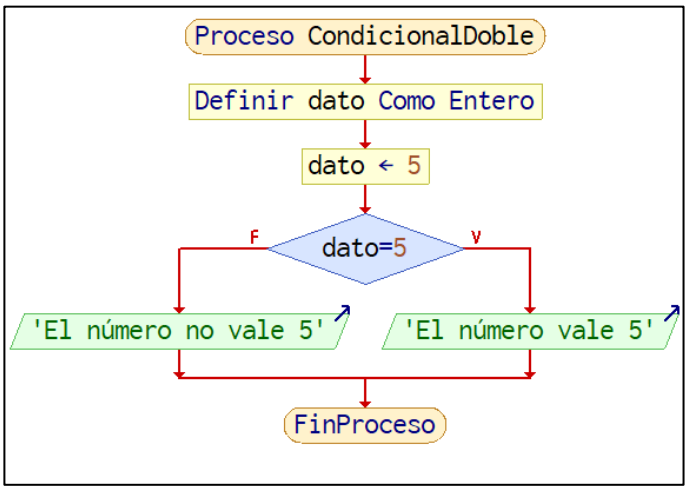
3.9. ESTRUCTURA CONDICIONAL DOBLE

El siguiente ejemplo muestra el uso de una condicional doble. Tras declarar una variable y asignarle un valor, se verifica si cumple una condición (que sea igual a 5). En caso afirmativo, se muestra un mensaje por consola y en caso contrario, se muestra otro mensaje distinto.

Pseudocódigo en PSeInt:

```
Proceso CondicionalDoble
    Definir dato Como Entero;
    dato ← 5;
    Si dato = 5 Entonces
        Escribir 'El número vale 5';
    SiNo
        Escribir 'El número no vale 5';
    FinSi
FinProceso
```

Ordinograma en PSeInt:



Java:

```
public class CondicionalDoble {

    public static void main(String[] args) {
          int dato;
          dato = 5;
          if(dato == 5) {
              System.out.println("El número vale 5");
          }
          else {
              System.out.println("El número no vale 5");
          }
    }

}
```

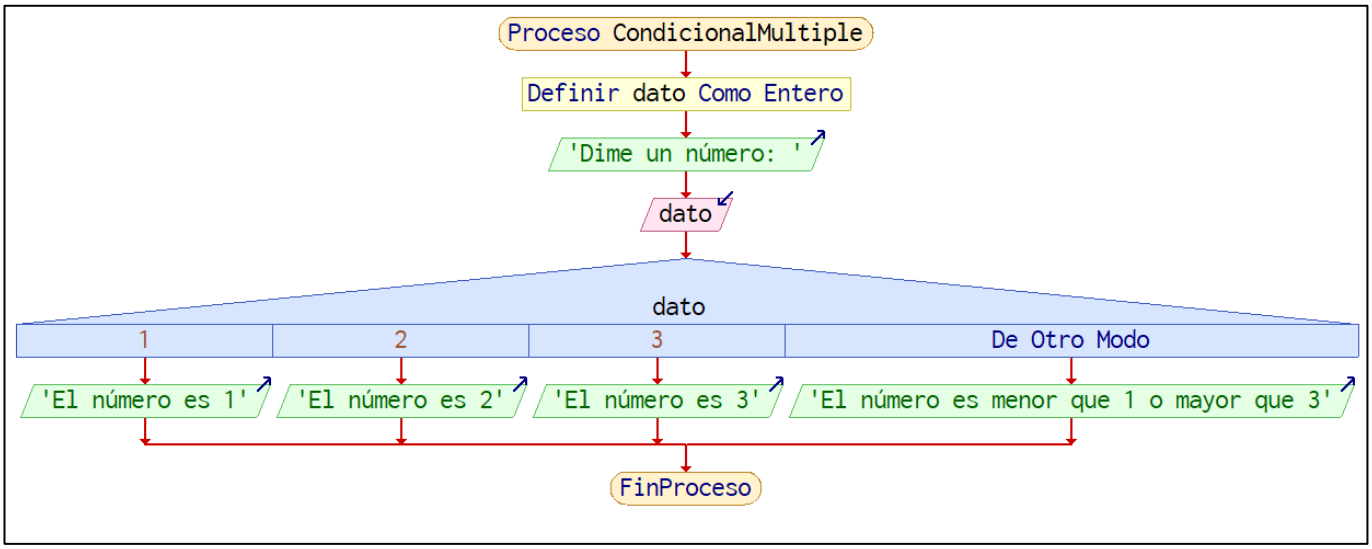
3.10. ESTRUCTURA CONDICIONAL CON MÚLTIPLES CASOS

El siguiente ejemplo muestra el uso de una estructura condicional con múltiples opciones. Tras declarar una variable y solicitar al usuario que indique su contenido, se evalúa el valor de dicha variable. Dependiendo del valor que tome, se realizará una acción u otra. También se puede indicar la acción a realizar en caso de que el valor no esté contemplado en los casos posibles.

Pseudocódigo en PSeInt:

```
Proceso CondicionalMultiple
    Definir dato Como Entero;
    Escribir 'Dime un número: ';
    Leer dato;
    Segun dato  Hacer
        1:
            Escribir 'El número es 1';
        2:
            Escribir 'El número es 2';
        3:
            Escribir 'El número es 3';
        De Otro Modo:
            Escribir 'El número es menor que 1 o mayor que 3';
    FinSegun
FinProceso
```

Ordinograma en PSeInt:



Java:

```java
import java.util.Scanner;

public class CondicionalMultiple {

    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        int dato;
        System.out.println("Dime un número: ");
        dato = teclado.nextInt();
        switch(dato) {
            case 1:
                System.out.println("El número es 1");
                break;
            case 2:
                System.out.println("El número es 2");
                break;
            case 3:
                System.out.println("El número es 3");
                break;
            default:
                System.out.println("El número es menor que 1 o mayor que 3");
        }
    }

}
```

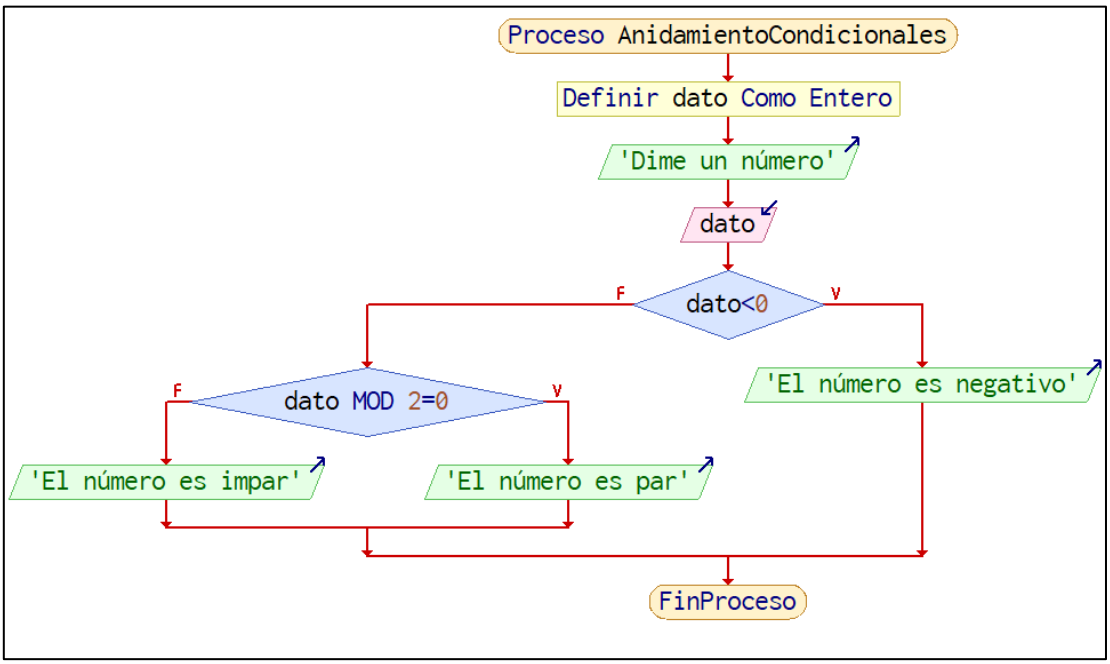
3.11. ANIDAMIENTO DE ESTRUCTURAS CONDICIONALES

El siguiente ejemplo muestra el uso de condicionales anidadas, es decir, una dentro de otra. Tras declarar una variable y solicitar al usuario que indique su valor, se comprueba si es menor que cero, en caso afirmativo se muestra un mensaje informativo y en caso contrario se evalúa otra condición. Esta segunda condición verifica si el número es divisible para dos, en caso afirmativo muestra un mensaje y en caso contrario, muestra otro distinto.

Pseudocódigo en PSeInt:

```
Proceso AnidamientoCondicionales
    Definir dato Como Entero;
    Escribir "Dime un número";
    Leer dato;
    Si dato < 0 Entonces
        Escribir "El número es negativo";
    SiNo
        Si dato mod 2 = 0 Entonces
            Escribir "El número es par";
        SiNo
            Escribir "El número es impar";
        FinSi
    FinSi
FinProceso
```

Ordinograma en PSeInt:



Java:

```
import java.util.Scanner;

public class AnidamientoCondicionales {

    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        int dato;
        System.out.println("Dime un número: ");
        dato = teclado.nextInt();
        if (dato < 0) {
            System.out.println("El número es negativo");
        }
        else {
            if(dato % 2 == 0) {
                System.out.println("El número es par");
            }
            else {
                System.out.println("El número es impar");
            }
        }
    }

}
```

En Java, la anidación de condicionales se puede realizar también mediante la estructura **if – else if - else**, como se muestra en el siguiente ejemplo, que es equivalente al anterior.

En esta estructura es posible indicar tantos **else if** como sean necesarios, pero solamente un **else**. No obstante, cuando se pueden dar múltiples condiciones se recomienda el uso de la estructura **switch**.

```java
import java.util.Scanner;

public class AnidamientoCondicionales {

    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        int dato;
        System.out.println("Dime un número: ");
        dato = teclado.nextInt();
        if (dato < 0) {
            System.out.println("El número es negativo");
        }
        else if(dato % 2 == 0) {
                System.out.println("El número es par");
        }
        else {
            System.out.println("El número es impar");
        }
    }

}
```

3.12. ESTRUCTURA REPETITIVA CON EXPRESIÓN ANTES DE CADA ITERACIÓN

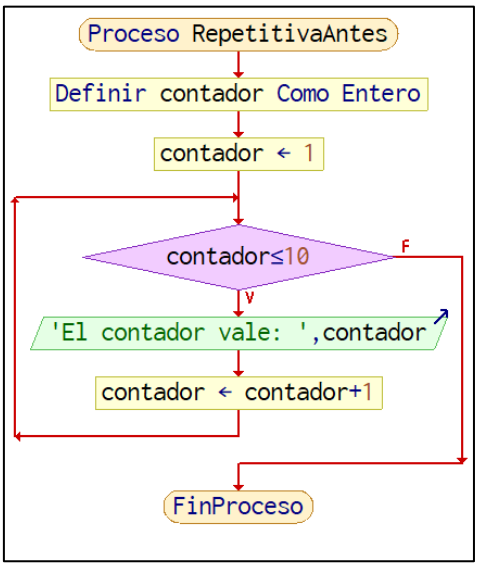
El siguiente ejemplo muestra el uso de una estructura repetitiva donde la condición se evalúa antes de entrar en el bucle.

Tras declarar una variable e inicializar su valor a uno, se comprueba si su valor es menor o igual que diez, en caso afirmativo, se muestra un mensaje informativo que contiene el valor de dicha variable y se incrementa en uno su valor; y en caso negativo, se continúa con la ejecución del programa, que en este caso no tiene más instrucciones. A continuación, se vuelve a evaluar si su valor es menor o igual que diez, en caso afirmativo se realizan las mismas acciones que se ha indicado antes y en caso negativo, se continúa con la ejecución del programa.

Pseudocódigo en PSeInt:

```
Proceso RepetitivaAntes
    Definir contador Como Entero;
    contador ← 1;
    Mientras contador≤10 Hacer
        Escribir 'El contador vale: ',contador;
        contador ← contador+1;
    FinMientras
FinProceso
```

Ordinograma en PSeInt:



Java:

```java
public class RepetitivaAntes {

    public static void main(String[] args) {
        int contador;
        contador = 1;
        while(contador <= 10) {
            System.out.println("El contador vale " + contador);
            contador++; //contador = contador + 1
        }
    }

}
```

3.13. ESTRUCTURA REPETITIVA CON EXPRESIÓN DESPUÉS DE CADA ITERACIÓN

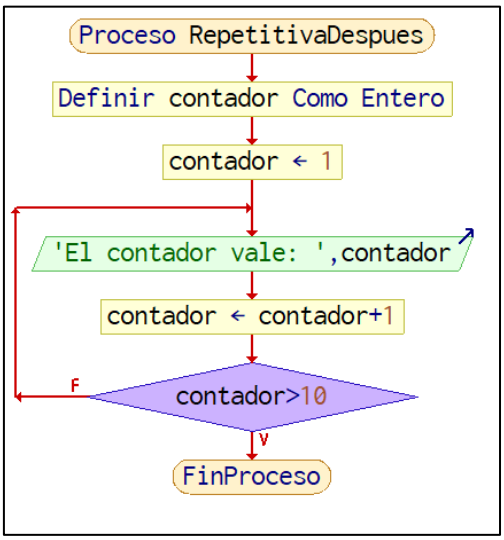
El siguiente ejemplo muestra el uso de una estructura repetitiva donde la condición se evalúa después de entrar en el bucle.

Tras declarar una variable e inicializar su valor a uno, se muestra un mensaje informativo que contiene el valor de dicha variable y se incrementa en uno su valor. A continuación, se comprueba si su valor es menor o igual que diez, en caso afirmativo, se realizan las mismas acciones que se ha indicado antes y se vuelve a evaluar la condición; y en caso negativo, se continúa con la ejecución del programa, que en este caso no tiene más instrucciones.

Pseudocódigo en PSeInt:

```
Proceso RepetitivaDespues
    Definir contador Como Entero;
    contador ← 1;
    Repetir
        Escribir "El contador vale: ", contador;
        contador ← contador + 1;
    Hasta Que contador > 10 //contador = 11
FinProceso
```

Ordinograma en PSeInt:



Java:

```java
public class RepetitivaDespues {

    public static void main(String[] args) {
        int contador;
        contador = 1;
        do{
            System.out.println("El contador vale " + contador);
            contador++; //contador = contador + 1
        }while(contador <= 10);

    }

}
```

3.14. ESTRUCTURA REPETITIVA CON NÚMERO DE ITERACIONES CONOCIDO

El siguiente ejemplo muestra el uso de una estructura repetitiva en la que se conoce el número de iteraciones a realizar.

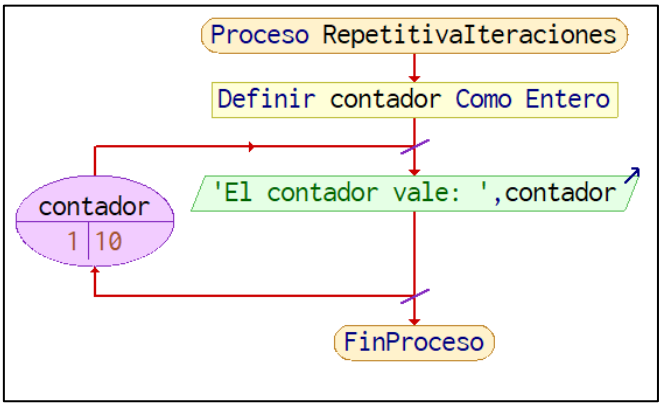
Tras declarar la variable que hará de contador, en la primera vuelta del bucle se le asigna un valor inicial. A continuación, se evalúa la condición de entrada y en caso de que se cumpla, se realizan las acciones que se indiquen dentro del bucle. Después de realizar dichas acciones, se modifica en contador según la expresión indicada (en este caso se incrementa en uno), y se evalúa de nuevo la condición, si se cumple, se realizan las acciones indicadas y en caso contrario, se finaliza el bucle. Se repetirá esta mecánica hasta que deje de cumplirse la condición de entrada.

Es importarte observar que la representación en Pseudocódigo de la condición de salida es distinta a la representación que se hace en Java. En Pseudocódigo se indica “hasta 10”, y en Java la condición se representa con “contador<=10”, en ambos casos tiene el mismo significado: que itere hasta que contador valga 10, o dicho de otro modo, que lo haga mientras el valor de contador sea menor o igual que 10.

Pseudocódigo en PSeInt:

```
Proceso RepetitivaIteraciones
    Definir contador Como Entero;
    Para contador ← 1 Hasta 10 Con Paso 1 Hacer
        Escribir "El contador vale: ", contador;
    FinPara
FinProceso
```

Ordinograma en PSeInt:



Java:

```
public class RepetitivaIteraciones {

    public static void main(String[] args) {
        int contador;
        for(contador = 1; contador <=10; contador++) {
            System.out.println("El contador vale " + contador);
        }
    }

}
```

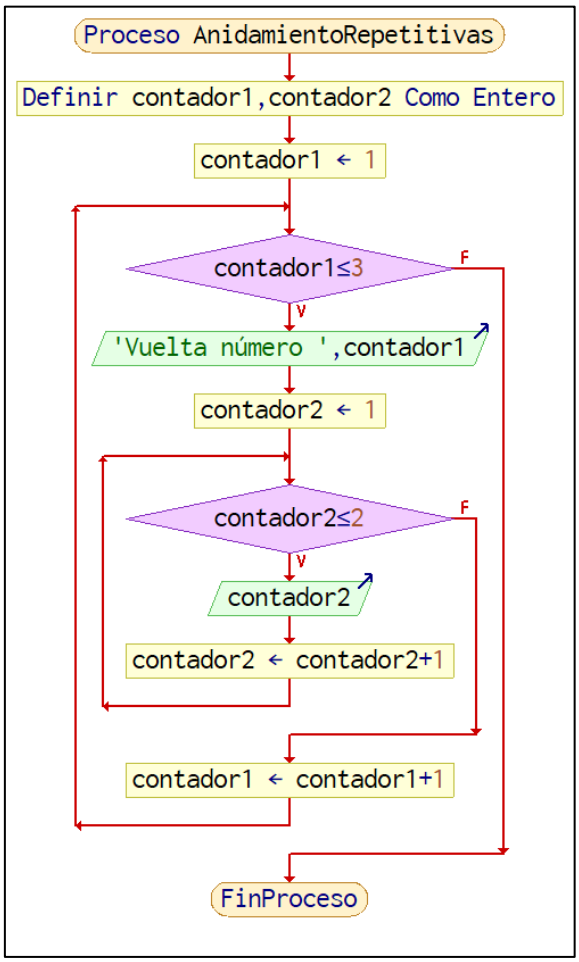
3.15. ANIDAMIENTO DE ESTRUCTURAS REPETITIVAS

El siguiente ejemplo muestra el uso de estructuras repetitivas anidadas, es decir, una dentro de otra.

Pseudocódigo en PSeInt:

```
Proceso AnidamientoRepetitivas
    Definir contador1, contador2 Como Entero;
    contador1 ← 1;
    Mientras contador1 <= 3 Hacer
        Escribir 'Vuelta número ',contador1;
        contador2 ← 1;
        Mientras contador2 <= 2 Hacer
            Escribir contador2;
            contador2 ← contador2 + 1;
        FinMientras
        contador1 ← contador1 + 1;
    FinMientras
FinProceso
```

Ordinograma en PSeInt:



Java:

```
public class AnidamientoRepetitivas {

    public static void main(String[] args) {
        int contador1, contador2;
        contador1 = 1;
        while(contador1 <= 3) {
            System.out.println("Vuelta número " + contador1);
            contador2 = 1;
            while(contador2 <=2) {
                System.out.println(contador2);
                contador2++;
            }
            contador1++;
        }

    }

}
```