Lógica de programación y laboratorio

Contenido

UNIDAD 1 CONCEPTOS BASICOS 6

1.1 Definición de problema 7

1.2 Tipos de solución a un problema 7

1.2.1 Solución al azar 7

1.2.2 Solución Algorítmica 8

1.2.3 Solución heurística 9

1.3 Pensamiento Computacional 10

1.4. Ventajas 11

1.5. Pilares 11

1.5.1 Descomposición 12

1.5.2 Generalización y Abstracción 12

1.5.3 Reconocimiento de patrones 13

1.5.4 Lógica y Algoritmia 13

1.6 Análisis- especificación de requerimientos y regla de negocio 15

1.6.1 Tipos de requisitos y reglas de negocio 15

1.7 Cuestionario 19

UNIDAD 2 LENGUAJES EN INFORMÁTICA Y REGLAS DE COMUNICACIÓN 20

2.1. Introducción 21

2.2. Definición de lenguaje 21

2. 3. Tipos de lenguaje 21

2.4 Definición de lenguaje Binario 22

2.5 ¿Cómo funciona el lenguaje binario? 22

2.6. Definición de lenguaje de programación 24

2.7 Tipos de lenguajes de programación 25

2.8 ¿Cómo funciona un lenguaje de programación? 26

2.9. Definición de programa 26

2.10. Definición Producto Software 27

2.11 Tipos de Software 27

2.12 Ciclo de vida del software 28
2.13 Metodología de trabajo 32
2.14 Cuestionario 33
UNIDAD 3 ALGORITMOS 34
3.1. Introducción 35
3.2. Definición de Algoritmo 35
3.3. Características de los algoritmos 36
3.4 Estructura de un Algoritmo 37
3.5. Lenguajes Algorítmicos 39
3.5.1 Diagrama de flujo 39
3.5.2 Diagramas rectangulares o diagrama de Chapin 42
3.5.3 Pseudocódigo 43
3.6. Ventajas de utilizar seudocódigo a diagrama de flujo 44
3.7 Cuestionario 45
UNIDAD 4 PROGRAMACION ORIENTADA A OBJETOS 46
4.1 Introducción 47
4.2. Los paradigmas de programación y sus diferencias 47
4.2.1 Secuencial 47
4.2.2 Estructurado 48
4.2.3 Programación orientada a objetos (POO) 49
4.3. Definición de programación orientado a objeto (POO) 50
4.4. Ventajas de la programación orientado a objeto 51
4.5. Elementos básicos de la programación orientado a objetos 52
4.5.1. Clase 52
4.5.1.1 La estructura general de una clase 52
4.5.1.2 Elementos de una clase 53
4.5.1.2.1 Atributos 53
4.5.1.2.2 Métodos 54
4.5.1.2.2.1 Estructura general de un método 54
4.5.2. Objetos e instancias 56
4.6 Standares para dar nombre a las clases, métodos y variables 58
4.7 UML 59
4.7.1 Diagramas UML 59

4.7.1.1 Diagramas Estructurales 60
4.7.1.2 Diagramas de Comportamiento 61
4.7.2 Diagrama de clase 61
4.8 Cuestionario 63
64
UNIDAD 5 ENTIDADES PRIMITIVAS PARA EL DESARROLLO DE ALGORITMOS 64
5.1 Introducción 65
5.2 Identificadores como localidades de memoria (variables) 65
5.3 Reglas para formar un Identificador 66
5.3.1 Palabras reservadas 68
5.4 Declarar y definir variables 69
5.4.1 Tipos de datos 70
5.4.2 Clasificación de las variables según su ámbito 71
5.5 Expresiones 72
5.6 Operadores Aritméticos 73
5.6.1 Jerarquía de los operadores Aritméticos 74
5.7 Convertir una expresión algebraica a expresión algorítmica 75
5.8 Ejercicios Resueltos 76
5.9 Ejercicios Propuestos 78
5.10 Cuestionario 79
UNIDAD 6 ESTRUCTURAS ALGORITMICAS 80
6.1 Introducción 83
6.2 Estructuras Secuenciales 83
6.2.1 Asignación directa y lectura de datos 84
6.2.2 Escritura 86
6.2.2.1 Formatear un numero 87
6.2.3 Problemas resueltos 88
6.2.4 Prueba de escritorio 92
6.3. Trabajando con 2 o más métodos 93
6.3.1 paso de parámetro por valor 94
6.3.2 No Envía Parámetros, No Recibe Parámetros 95
6.3.3 Envía parámetros, No recibe Parámetros 96
6.3.4 No Envía Parámetros, Si Recibe parámetro 97

6.3.5 Envía Parámetros, Recibe parámetro 98
6.4 Estructuras Condicionales 100
6.4.1 Operadores relacionales 101
6.4.2 Operadores lógicos 101
6.4.3 Estructura Condicional Simple 103
6.4.4 Estructuras Condicionales Compuestas 105
6.4.5 Múltiples (según) 107
6.4.6. Operador Ternario 111
6.4.7 Problemas resueltos 113
6.4.8 Problemas propuestos 116
6.5 Estructuras Repetitivas o Cíclicas 117
6.5.1 Contadores 117
6.5.2 Acumuladores 117
6.5.3 Ciclo para 118
6.5.3.1 variables tipo bloque 119
6.5.4 Ciclo mientras 120
6.5.5 Ciclo hacer mientrasque 121
6.5.6 Centinela 126
6.5.7 Swiches o banderas 127
6.5.8 Rompimiento de ciclos 128
6.5.9 Trabando con 2 o más Clases 130
6.5.9.1 instancia 130
6.5.9.2 Método constructor 134
6.5.9.3 sobrecarga de métodos **¡Error! Marcador no definido.**
6.5.10 Problemas resueltos **¡Error! Marcador no definido.**
6.5.11 Problemas propuestos 137
6..6 Cuestionario **¡Error! Marcador no definido.**
UNIDAD 7 ARREGLOS 138
7.1 Introducción 140
7.2 Definición de arreglos 140
7.3 Características de los arreglos 140
7.4 Tipos de Arreglos 140
7.4.1 Unidimensionales (Vectores) 140

7.4.1.1 creación 140
7.4.1.2 Asignación 140
7.4.1.3 Paso de parámetro por referencia 140
7.4.1.4 Métodos de ordenamiento 140
7.4.1.4.1 Burbuja 140
7.4.1.4.2 Selección 140
7.4.1.4.3 Inserción 140
7.4.1.4.4 Shell 140
7.4.1.5 Búsqueda 140
7.4.1.5.1 Secuencial 140
7.4.1.5.2 Binaria. 140
7.4.1.6 Manipulación de Cadenas 140
7.4.2 N dimensiones (matrices) 140
7.4.2.1 Creación 140
7.4.2.2 Recorridos 140
7.5 Arreglos Objetuales 140
7.6 Ejercicios resueltos 141
7.7 Ejercicios propuestos 141
7.8 Cuestionario 141
UNIDAD 8 CARACTERISTICAS DEL ENFOQUE ORIENTADO A OBJETO 142
8.1 Introducción 143
8.2 Abstracción 143
8.3 Encapsulación 143
8.4 Modularidad 143
8.5 Herencia 143
8.6 Jerarquía de clase 143
8.7 Polimorfismo 143
8.8 Problemas propuestos 143
8.9 Problemas Resueltos 143
8.10 Cuestionario 143

1

UNIDAD 1 CONCEPTOS BASICOS

CONTENIDO

1.1. Definición de problema
1.2 Tipos de solución a un problema
1.2.1 Solución al azar
1.2.2 Solución Algorítmica
1.2.3 Solución heurística
1.3. Pensamiento Computacional
1.4. Ventajas
1.5. Pilares
1.51. Descomposición
1.5.2. Abstracción
1.5.3. Reconocimiento de patrones
1.5.4. Lógica y algoritmos
1.6. Análisis- especificación de requerimientos y regla de negocio
1.6.1. Tipos de requisitos y reglas de negocio
1.7 Cuestionario

 Resultado de Aprendizaje:

El estudiante

Demostrará su capacidad para aplicar un enfoque sistemático y lógico en la solución de problemas, utilizando herramientas y métodos del pensamiento computacional y análisis de requisitos

1.1 Definición de problema

Un problema es cualquier situación que requiere una solución. Puede ser simple o complejo, abstracto o concreto, y puede surgir en cualquier ámbito de la vida.

Ejemplo:

- **Problema simple:** ¿Cuál es la capital de Francia?
- **Problema complejo:** ¿Cómo curar una enfermedad desconocida?
- **Problema abstracto:** ¿Cuál es el sentido de la vida?

1.2 Tipos de solución a un problema

1.2.1 Solución al azar

La solución por azar implica encontrar una respuesta mediante la prueba y error, sin seguir un método específico. Es como lanzar una moneda al aire para tomar una decisión.

Ejemplo:

- **Buscar las llaves perdidas:** Revisar lugares al azar hasta encontrarlas.
- **Jugar a la lotería:** Elegir números aleatoriamente esperando ganar.

Imagen:

Figura 1 Solución al Azar

1.2.2 Solución Algorítmica

Una solución algorítmica sigue una secuencia de pasos bien definida para llegar a una respuesta. Es como una receta de cocina, donde cada paso debe seguirse en un orden específico.

Ejemplo:

- **Resolver una ecuación matemática:** Aplicar las reglas del álgebra para encontrar el valor de la incógnita.
- **Ordenar una lista de números:** Utilizar un algoritmo de ordenamiento como el de burbuja o quicksort.

Imagen:

Figura 2 solución algorítmica

1.2.3 Solución heurística

Una solución heurística busca una buena aproximación a la solución óptima, utilizando reglas empíricas o atajos mentales. No garantiza encontrar la mejor solución, pero es útil cuando la búsqueda exhaustiva es demasiado costosa o compleja.

Ejemplo:

- **Resolver un sudoku:** Aplicar reglas generales como "en cada fila, columna y región solo puede haber un número del 1 al 9" para reducir las posibilidades.
- **Encontrar el camino más corto en un mapa:** Utilizar una heurística como la distancia euclidiana para estimar la distancia al destino y guiar la búsqueda.

Imagen:

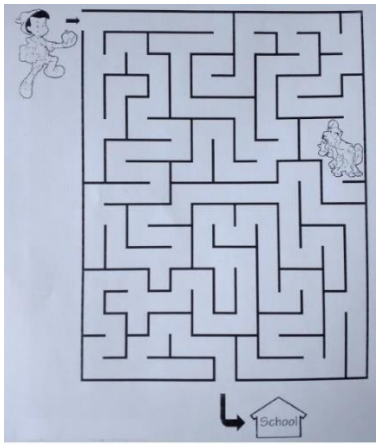


Figura 3 Solución heurística

Comparación entre los Tipos de Solución

Tipo de Solución	Características	Ventajas	Desventajas
Azar	Basada en la suerte	Fácil de implementar	No garantiza una solución, puede ser ineficiente.
Algorítmica	Sigue una secuencia de pasos	Garantiza una solución (si existe), puede ser eficiente.	Puede ser compleja de diseñar para problemas grandes.
Heurística	Busca una buena aproximación	Rápida y eficiente para problemas complejos.	No garantiza la solución óptima, puede quedar atrapada en óptimos locales.

Tabla 1 Comparación entre los Tipos de Solución

Teniendo en cuenta lo anterior podemos concluir que:

- **El azar** es útil para problemas simples o cuando no hay información suficiente.
- **La algoritmia** es ideal para problemas bien definidos y que requieren una solución exacta.
- **La heurística** es adecuada para problemas complejos donde la búsqueda exhaustiva no es práctica.

1.3 Pensamiento Computacional

El pensamiento computacional es una habilidad fundamental en el mundo actual, similar a aprender a leer o escribir. Consiste en enfrentar y resolver problemas de manera estructurada y lógica, utilizando conceptos y herramientas de la informática.

En esencia, es como enseñar a una computadora a realizar una tarea, pero aplicado a cualquier situación de la vida.

1.4. Ventajas

- **Desarrolla habilidades críticas:** Fomenta el pensamiento crítico, la resolución de problemas, la creatividad y la colaboración.
- **Prepara para el futuro:** El pensamiento computacional es esencial en el mundo laboral, donde la tecnología desempeña un papel cada vez más importante.
- **Facilita la comprensión del mundo:** Nos ayuda a entender cómo funcionan los sistemas y procesos que nos rodean.

1.5. Pilares

El pensamiento computacional se basa en cuatro pilares fundamentales:

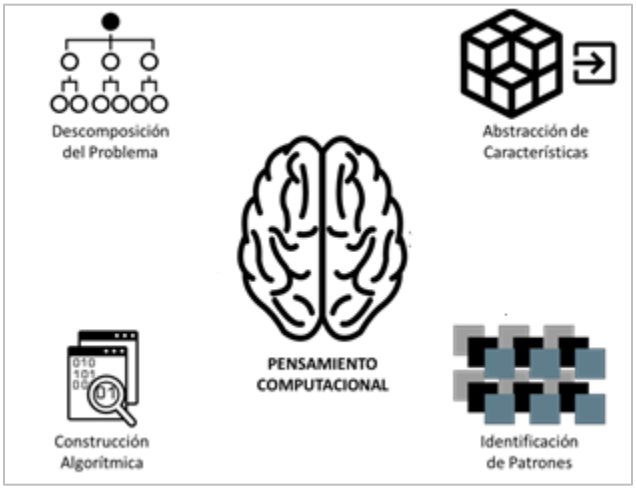


Figura 4 Pensamiento Computacional

1.5.1 Descomposición

La descomposición es una habilidad esencial en el pensamiento computacional. Al descomponer un problema en partes más pequeñas, podemos abordarlo de manera más efectiva y encontrar soluciones creativas.

Ejemplo: Para hacer una torta, primero se separan los ingredientes, luego se mezclan y finalmente se hornea.

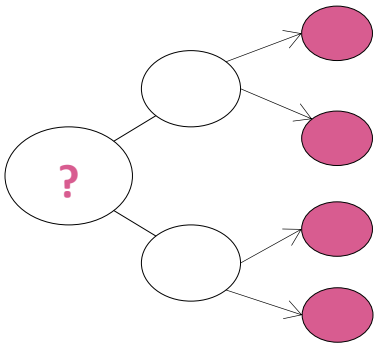


Figura 5 Descomposición

1.5.2 Generalización y Abstracción

Se centra en la información importante y deja de lado los detalles irrelevantes

Ejemplo: Al diseñar un mapa, se representan las calles y edificios, pero se omiten los detalles como los árboles o las personas.

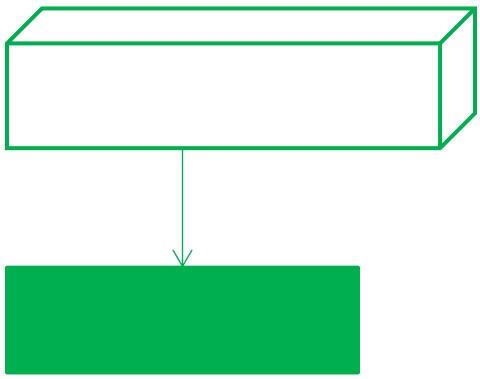


Figura 6 abstracción

1.5.3 Reconocimiento de patrones

Consiste en: Identificar similitudes y regularidades en los datos.

Ejemplo: En una secuencia numérica como 3, 6, 9, 12, se reconoce el patrón de sumar 3 a cada número

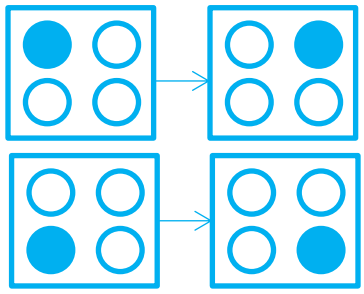


Figura 7 Reconocimiento de patrones

1.5.4 Lógica y Algoritmia

Consiste en crear una secuencia de pasos ordenados para resolver un problema.

Ejemplo: Una receta de cocina es un algoritmo que describe los pasos para preparar un plato.

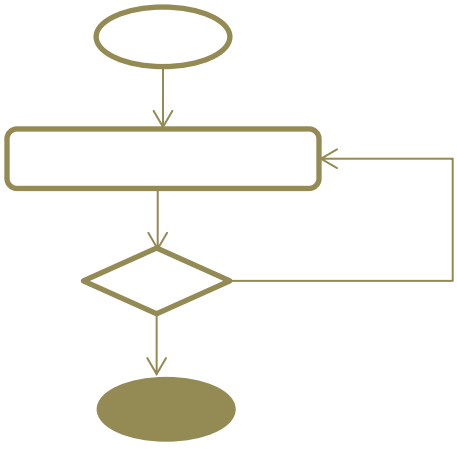


Figura 8 Lógica y Algoritmia

Ejemplos de Pensamiento Computacional en la Vida Cotidiana

- **Planificar un viaje:** Descomponer el viaje en etapas, buscar la ruta más eficiente (algoritmo), identificar patrones en el tráfico (reconocimiento de patrones), y abstraer la información relevante de un mapa (abstracción).
- **Resolver un rompecabezas:** Dividir el rompecabezas en secciones, identificar piezas con formas similares (reconocimiento de patrones), y encajar las piezas de acuerdo con un plan (algoritmo).
- **Crear una presentación:** Organizar las ideas principales (descomposición), diseñar diapositivas con un estilo visual coherente (abstracción), y establecer una secuencia lógica para la presentación (algoritmo).

¿Cómo desarrollar el pensamiento computacional?

- **Programar:** La programación es una excelente forma de poner en práctica los conceptos del pensamiento computacional.
- **Resolver puzzles y juegos lógicos:** Estos juegos ayudan a desarrollar habilidades de resolución de problemas y pensamiento crítico.
- **Participar en proyectos de robótica:** La construcción y programación de robots fomenta la creatividad y la resolución de problemas.
- **Pensar de manera algorítmica en la vida cotidiana:** Identificar patrones y secuencias en las tareas diarias.

El pensamiento computacional es una herramienta poderosa que nos permite abordar problemas de manera más efectiva y eficiente. Al desarrollar estas habilidades, estamos mejor preparados para enfrentar los desafíos del mundo actual y futuro.

Estos pilares nos servirán para poder identificar o comprender los requisitos o necesidades de un cliente al momento de desarrollar un software

1.6 Análisis- especificación de requerimientos y regla de negocio

¿Qué es un requisito? Un requisito en el desarrollo de software es como una regla o condición que el software debe cumplir para satisfacer las necesidades del usuario. Es como una receta que detalla los ingredientes y pasos necesarios para crear un plato específico.

los requisitos son la base de cualquier proyecto de software. Al definir claramente los requisitos, se asegura que el software final cumpla con las necesidades del cliente y sea un éxito.

Proceso de Recopilación de Requisitos:

1. **Identificar a los stakeholders:** Son todas las personas involucradas en el proyecto (cliente, usuarios finales, equipo de desarrollo, etc.).
2. **Reunir información:** Se utilizan diversas técnicas como entrevistas, encuestas, talleres y análisis de documentos.
3. **Documentar los requisitos:** Se crea un documento de requisitos que detalla todas las características del software.

Figura 9 Proceso de Recopilación de Requisitos

1.6.1 Tipos de requisitos y reglas de negocio

- **Funcionales:** Definen lo que el software debe hacer. Por ejemplo, "El sistema debe permitir al usuario registrarse".

- **No funcionales:** Describen las características del software, como la performance, la seguridad o la usabilidad. Por ejemplo, "El sistema debe responder en menos de 2 segundos".
- Los **Reglas de negocio** restringen o limitan como deben funcionar los requisitos funcionales

Requerimientos Funcionales	**Requerimientos No Funcionales**
Describe lo que el sistema debe hacer, es decir, una función o tarea específica	Describe cómo debe funcionar el sistema, es decir, los atributos o la calidad del sistema
Se centra en el comportamiento y las características del sistema	Se centra en el rendimiento, la facilidad de uso y otros atributos de calidad
Define las acciones y operaciones del sistema	define las restricciones o condiciones bajo las cuales debe funcionar el sistema
Entrada/salida de datos de autenticación de usuario, procesamiento de transacciones	Escalabilidad seguridad, tiempo de respuesta, confiabilidad, mantenibilidad

Tabla 2 tipos de requisitos

Practiquemos

Se requiere obtener la distancia entre dos puntos en el plano cartesiano, tal y como se muestra en la *figura 1.* indique los pasos para obtener la distancia entre esos puntos, e impleméntelo en lenguaje de programación java

REQUISITOS FUNCIONALES:
Se requiere obtener la distancia entre dos puntos en el plano

REGLAS DE NEGOCIO:
Se requieren las coordenadas de dos puntos del plano P1(x_1, y_1), P2(x_2, y_2)
Las coordenadas son números reales
La distancia=$\sqrt{(x_1 - x_2)^2+(y_1 - y_2)^2}$

REQUISITOS NO FUNCIONALES:
Implementarlo en lenguaje de programación java

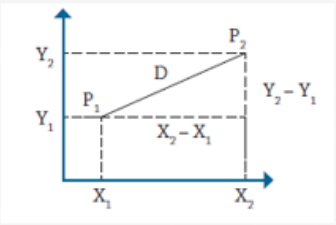


Figura 1.

Figura 10 Ejercicio 1 Requisitos

Ahora identifica los requisitos funcionales, no funcionales y regla de negocio para el siguiente ejemplo

Una modista, para realizar sus prendas de vestir, encarga las telas al extranjero. Para cada pedido, tiene que proporcionar las medidas de la tela en pulgadas, pero ella generalmente las tiene en metros. Realice un algoritmo para ayudar a resolver el problema, determinando cuántas pulgadas debe pedir con base en los metros que requiere. Al implementar el proyecto, debe poder utilizarse desde un dispositivo móvil.

REQUISITOS FUNCIONALES:

REGLAS DE NEGOCIO:

REQUISITOS NO FUNCIONALES:

Figura 11 Ejercicio 2 Requisitos

La solución sería:

Cliente	La modista
Usuario	La modista
Requerimientos Funcionales	El sistema debe permitir: ▪ Convertir de metros a pulgadas
Reglas de Negocio	▪ La medida de entrada en metros es un número real ▪ pulgadas =39.37 × *metros*
Requerimientos NO Funcionales	▪ La aplicación debe ser desarrollada para un dispositivo móvil

Figura 12 solución Ejercicio 2

Y este último ejemplo

"El náufrago satisfecho" ofrece hamburguesas sencillas, dobles y triples, las cuales tienen un costo de $8.000, $10.000 y $12.000 respectivamente. La empresa acepta efectivo y si es con tarjetas de crédito hay un cargo de 5 % sobre la compra. Suponiendo que los clientes adquieren sólo un tipo de hamburguesa, realice un algoritmo para determinar cuánto debe pagar una persona por N hamburguesas

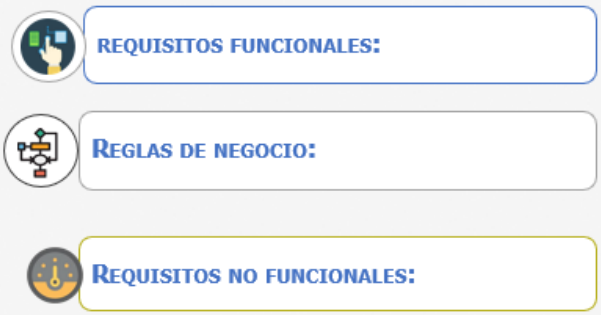


Figura 13 Ejercicio 3 Requisitos

La solución seria

CLIENTE	El "náufrago satisfecho"
USUARIO	El cajero del puesto de ventas
REQUERIMIENTOS FUNCIONALES	El sistema debe permitir: ▪ Calcular el costo a pagar en efectivo de N hamburguesas de un tipo específico ▪ Calcular el costo a pagar con tarjeta de crédito de N hamburguesas de un tipo específico
REGLAS DE NEGOCIO	▪ El número N de hamburguesas a comprar que ingresa e usuario es un valor entero mayor a cero ▪ El $valorEfectivo = N \times valoHamburguesa$ donde valorHamburguesa depende de la hamburguesa seleccionada: 8000, 10000 o 12000 si es sencilla, doble o triple, respectivamente. ▪ El $valorTarjeta = valorFectivo + (valorEfectivo \times 0.05)$
REQUERIMIENTOS NO FUNCIONALES	

Figura 14 solución Ejercicio 3

En este caso no hay requerimientos no funcionales.

1.7 Cuestionario

2

UNIDAD 2 LENGUAJES EN INFORMÁTICA Y REGLAS DE COMUNICACIÓN

CONTENIDO

2.1. Introducción
2.2. Definición de lenguaje
2.3 Tipos de lenguaje
2.4 Definición de lenguaje Binario
2.5 Como funciona el lenguaje binario?
2.6. Definición de lenguaje de programación
2.7 Tipos de lenguaje de programación
2.8 Como funciona un lenguaje de programación?
2.9. Definición de programa
2.10 Definición producto Software
2.11 Tipos de software
2.12 Ciclo de vida del software
2.13 Metodología de Trabajo
2.14 Cuestionario

 Resultado de Aprendizaje:

***El estudiante*:**

- Comprende los conceptos fundamentales del lenguaje y sus tipos Explicar el funcionamiento del lenguaje binario
- Identifica y comparar los diferentes tipos de lenguajes de programación
- Define y diferencia los conceptos clave en el desarrollo de software
- Describe y analizar el ciclo de vida del software

2.1. Introducción

Los lenguajes en informática son herramientas fundamentales que permiten a los desarrolladores y a las máquinas comunicarse de manera efectiva. Además, las reglas de comunicación dentro de estos lenguajes son esenciales para garantizar que el código sea comprensible y ejecutable. Estas reglas abarcan desde la sintaxis, que define la estructura correcta de las instrucciones, hasta la semántica, que se refiere al significado de dichas instrucciones.

2.2. Definición de lenguaje

El **lenguaje** es un sistema de signos que utilizamos los seres humanos para comunicarnos, expresar ideas, sentimientos y pensamientos. Estos signos pueden ser sonidos (habla), gestos, imágenes o símbolos escritos.

Figura 15 Lenguaje

2. 3. Tipos de lenguaje

Existen diversas formas de clasificar el lenguaje. A continuación, te presento algunas de las más comunes:

- **Lenguaje natural:** Es el que se desarrolla de forma espontánea en una comunidad y se transmite de generación en generación. Ejemplos: español, inglés, francés.
- **Lenguaje verbal:** Utiliza palabras para transmitir información. Puede ser oral (hablado) o escrito.

- **Lenguaje no verbal:** Se expresa a través de gestos, expresiones faciales, postura corporal, etc.
- **Lenguaje artificial:** Es creado por el ser humano para fines específicos, como la programación o la comunicación entre máquinas. Ejemplos: lenguaje de programación (Python, Java), lenguaje matemático, lenguaje binario entre otros

2.4 Definición de lenguaje Binario

El **lenguaje binario** es un sistema de numeración que utiliza únicamente dos dígitos: el 0 y el 1. Estos dígitos representan estados opuestos, como encendido y apagado, verdadero y falso, o presencia y ausencia de una señal eléctrica. Las computadoras utilizan el lenguaje binario para procesar y almacenar toda la información, desde texto y números hasta imágenes y videos.

¿Por qué usamos el sistema binario en las computadoras?

- **Simplicidad:** Los circuitos electrónicos son más fáciles de diseñar y construir cuando solo tienen dos estados posibles.
- **Fiabilidad:** Es menos probable que ocurran errores en un sistema binario, ya que solo hay dos opciones.
- **Flexibilidad:** El sistema binario puede representar cualquier tipo de información, desde números hasta letras y símbolos especiales.

2.5 ¿Cómo funciona el lenguaje binario?

Cada dígito binario se llama **bit**. **El bit (Binary Digit)** es la unidad más pequeña de información en informática. Imagina un interruptor de la luz: cuando está encendido representa un 1, y cuando está apagado, un 0.

Figura 16 bit

El byte es un grupo de 8 bits. Puedes imaginarlo como una pequeña caja que contiene 8 interruptores.

Figura 17 byte

Con 8 bits, podemos representar muchas más combinaciones que con solo uno, lo que nos permite representar letras, números, y otros símbolos.

Ejemplo

Imagina que queremos representar la letra "A" en binario. Según la tabla ASCII, la letra "A" se representa como 01000001. Aquí tenemos un byte (8 bits) que representa una sola letra.

A continuación, te mostraremos los diferentes sistemas de representación

Decimal - Binary - Octal - Hex – ASCII
Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Figura 18 tabla de conversión de caracteres

2.6. Definición de lenguaje de programación

Un lenguaje de programación es como un idioma que utilizamos para comunicarnos con las computadoras. En lugar de palabras y frases, usamos un conjunto específico de instrucciones y reglas que las máquinas pueden entender y ejecutar.

Imagina que una computadora es como un chef muy obediente, pero que solo entiende recetas muy específicas. Un lenguaje de programación es esa receta, un conjunto de instrucciones detalladas que le decimos a la computadora para que realice una tarea determinada, ya sea desde realizar un cálculo sencillo hasta crear un videojuego complejo.

¿Por qué necesitamos lenguajes de programación?

- **Automatización:** Realizar tareas repetitivas de forma rápida y precisa.

- **Creación de software:** Desarrollar aplicaciones, juegos, sistemas operativos y más.
- **Control de hardware:** Interactuar con dispositivos físicos como robots, drones o sensores.
- **Análisis de datos:** Procesar grandes cantidades de información para obtener conclusiones.

2.7 Tipos de lenguajes de programación

Existen muchos lenguajes de programación, cada uno con sus propias características y diseñado para diferentes tareas. Algunos de los más populares son:

Lenguaje	Descripción	Ejemplo de uso
Python	Fácil de aprender, versátil, utilizado en ciencia de datos, aprendizaje automático y desarrollo web.	Creación de modelos de machine learning, análisis de datos, desarrollo de aplicaciones web.
JavaScript	Esencial para el desarrollo web interactivo, se ejecuta en el navegador.	Crear animaciones, juegos en línea, aplicaciones web dinámicas.
Java	Robusto y seguro, utilizado para desarrollar aplicaciones empresariales a gran escala.	Desarrollo de aplicaciones Android, sistemas bancarios, software empresarial.
C++	Lenguaje de bajo nivel, ofrece un gran control sobre el hardware, utilizado en videojuegos y sistemas operativos.	Desarrollo de videojuegos, software de sistemas, aplicaciones de alto rendimiento.

Tabla 3 Tipos de Lenguajes de Programación

2.8 ¿Cómo funciona un lenguaje de programación?

1. **Escribir el código:** El programador escribe las instrucciones en un editor de texto, siguiendo las reglas y sintaxis del lenguaje de programación seleccionado, este dará como resultado **el programa fuente**.
2. **Compilación o interpretación:** El código se traduce a un lenguaje que la computadora puede entender directamente (código máquina), dará como resultado **el programa objeto**
3. **Ejecución:** La computadora sigue las instrucciones del código, realizando las tareas especificadas.

Figura 19 Funcionamiento de un lenguaje de programación

2.9. Definición de programa

Un programa es un conjunto de instrucciones detalladas, escritas en un lenguaje que la computadora entiende, que le indica cómo realizar una tarea específica. Estas instrucciones son ejecutadas secuencialmente por la computadora para producir un resultado deseado.

Características de un programa:

- **Precisión:** Cada instrucción debe ser clara y concisa.
- **Secuencia:** Las instrucciones se ejecutan en un orden específico.
- **Lenguaje:** Los programas se escriben en lenguajes de programación como Python, Java, C++, etc.

¿Para qué sirven los programas?

Los programas nos permiten:

- **Automatizar tareas:** Realizar tareas repetitivas de forma rápida y eficiente.
- **Crear aplicaciones:** Desarrollar software para diversas funciones, como procesar textos, realizar cálculos, jugar videojuegos, etc.
- **Controlar dispositivos:** Interactuar con dispositivos físicos como robots, drones, sensores, etc.
- **Analizar datos:** Procesar grandes cantidades de información para obtener conclusiones.

2.10. Definición Producto Software

El **software** es la colección de todos los programas, datos y documentación asociados que permiten a una computadora funcionar. Es el componente intangible de un sistema informático, en contraste con el hardware que es la parte física.

2.11 Tipos de Software

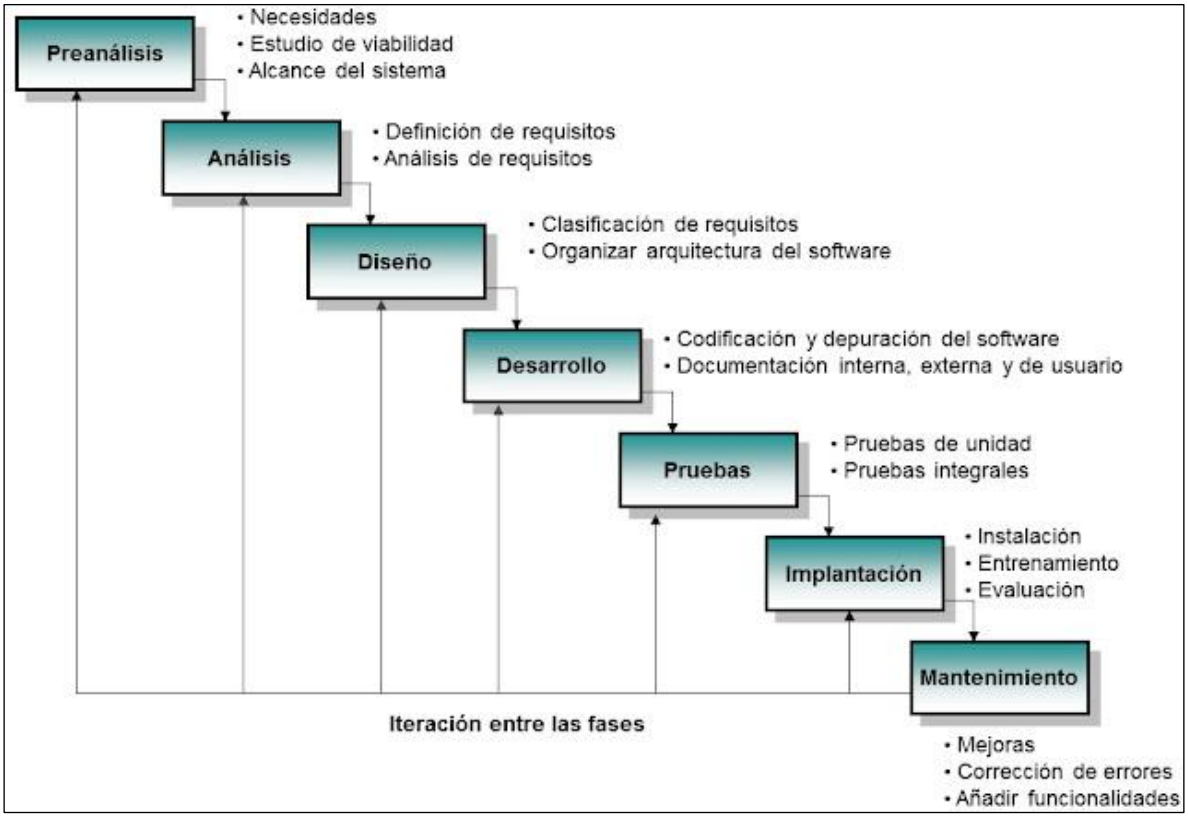
El software se clasifica en diferentes tipos según su función y propósito dentro de un sistema informático. Los tres tipos principales son:

- **Software de sistema:** Gestiona los recursos de hardware y proporciona una plataforma para ejecutar otros programas. Ejemplos: sistemas operativos (Windows, macOS, Linux), controladores de dispositivos.
- **Software de aplicación:** Realiza tareas específicas para el usuario. Ejemplos: procesadores de texto (Word), hojas de cálculo (Excel), navegadores web (Chrome), juegos.

- **Software de programación:** Herramientas para crear nuevos programas. Ejemplos: compiladores, intérpretes, entornos de desarrollo integrados (IDE).

2.12 Ciclo de vida del software

El ciclo de vida del software (SDLC, por sus siglas en inglés) es un proceso que describe las etapas involucradas en el desarrollo de un software, desde que comienza hasta su finalización y mantenimiento



El desarrollo de software sigue generalmente un ciclo que incluye las siguientes etapas:

1. **Preanálisis:** Se definen los objetivos y viabilidad del software
    - **Definición de objetivos:** Se establece qué se quiere lograr con el software.

- **Análisis de viabilidad:** Se evalúa si el proyecto es factible desde el punto de vista técnico y económico.
- **Estimación de recursos:** Se determinan los recursos necesarios (tiempo, presupuesto, personal).

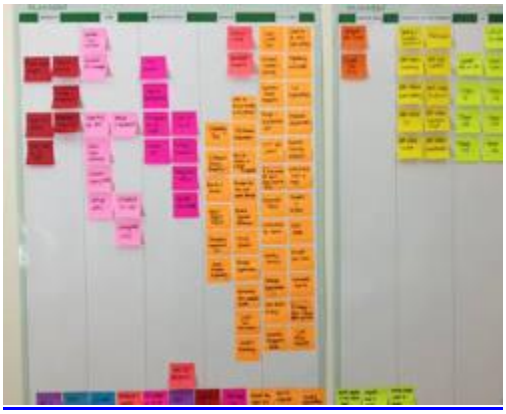


Figura 20 Planificación. ciclo de vida del software

2. **Análisis de requisitos:** Se define qué debe hacer el software
    - **Recopilación de requisitos:** Se identifican las necesidades del usuario y del sistema.
    - **Documentación de requisitos:** Se crean documentos detallados de los requisitos.

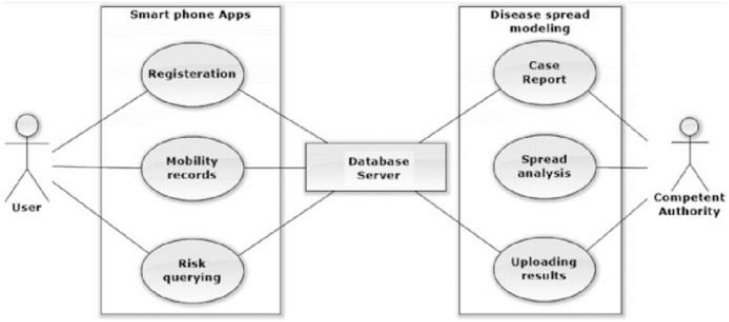


Figura 21 Análisis. ciclo de vida del software

3. **Diseño:** Se crea un plan detallado de cómo se construirá el software.
    - **Diseño arquitectónico:** Se define la estructura general del software.
    - **Diseño de interfaces:** Se diseñan las interfaces de usuario.
    - **Diseño de bases de datos:** Se diseña la estructura de almacenamiento de datos.

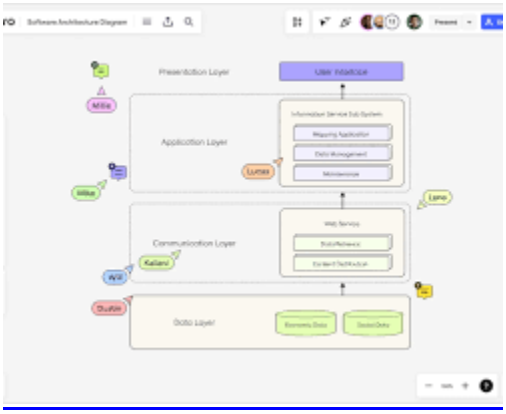


Figura 22 Diseño. ciclo de vida del software

4. **Desarrollo:** Se escriben las instrucciones en un lenguaje de programación.
    - **Codificación:** Se escribe el código fuente del software.
    - **Integración:** Se combinan los diferentes componentes del software.

Figura 23 Desarrollo. ciclo de vida del software

5. **Pruebas:** Se verifica que el software funcione correctamente
    - **Pruebas unitarias:** Se prueban las unidades individuales de código.
    - **Pruebas de integración:** Se prueban las interacciones entre los componentes.
    - **Pruebas de sistema:** Se prueba el software completo.

Figura 24 Pruebas. ciclo de vida del software

6. **Implementación:** Se instala el software en el sistema.
    - **Despliegue:** Se instala el software en el entorno de producción.
    - **Migración de datos:** Se transfieren los datos existentes al nuevo sistema.

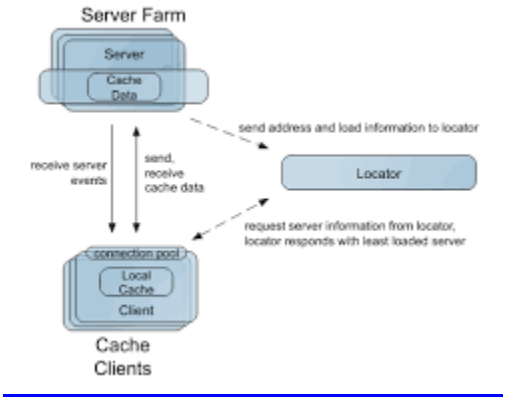


Figura 25 Implementación. ciclo de vida del software

7. **Mantenimiento:** Se realizan actualizaciones y correcciones de errores.
    - **Corrección de errores:** Se solucionan los problemas que surgen en el software.
    - **Mejoras:** Se realizan mejoras y actualizaciones al software.

Figura 26 Mantenimiento. ciclo de vida del software

2.13 Metodología de trabajo

Como has observado la realización de un software incluye muchas etapas, para este curso nos concentraremos en el análisis de la solución, diseño de la solución, validación y codificación

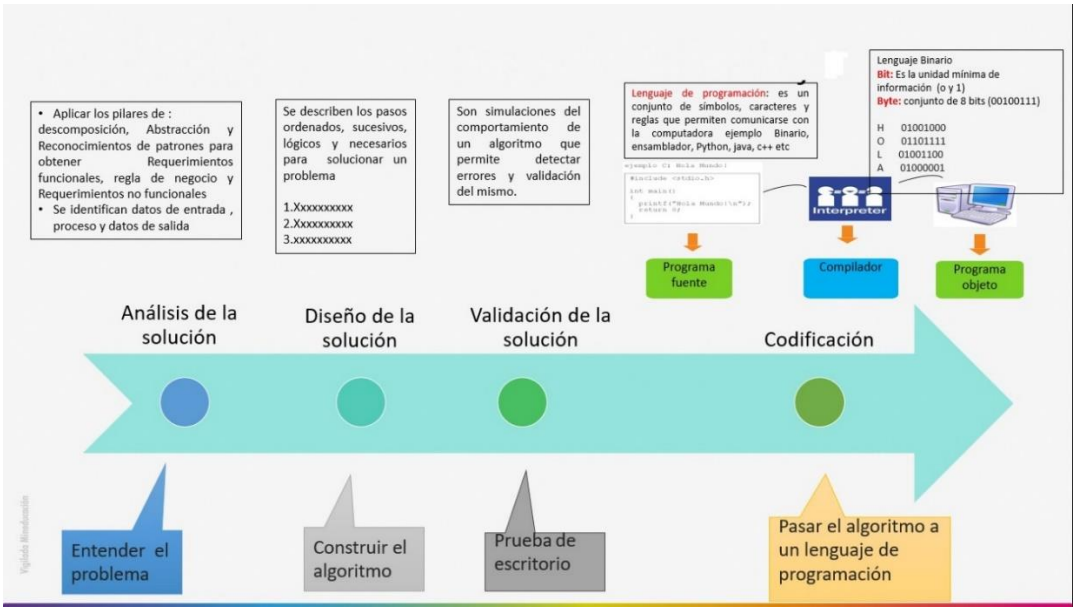


Figura 27 Metodología de trabajo

- **Análisis de la solución:** utilizando los pilares del pensamiento computación, identificamos los requisitos funcionales y reglas de negocio del problema a solucionar, Se identifican datos de entrada, proceso y datos de salida
- **Diseño de la solución:** Se describen los pasos ordenados, sucesivos, lógicos y necesarios para solucionar un problema
- **Validación de la solución o pruebas de escritorio**: Son simulaciones del comportamiento de un algoritmo que permite detectar errores y validación de este.
- **Codificación**: se escribe el algoritmo a un lenguaje de programación

2.14 Cuestionario

3

UNIDAD 3 ALGORITMOS

CONTENIDO

3.1. Introducción

3.2. Definición de Algoritmo

3.3. Características de los algoritmos

3.4 Estructura de un Algoritmo

3.5. Lenguajes Algorítmicos

3.5.1 Diagrama de flujo

3.5.2 Diagramas rectangulares o diagramas de Chapin

3.5.3 Pseudocodigo

3.6. Ventajas de utilizar seudocódigo a diagrama de flujo

3.7 Cuestionario

Resultado de Aprendizaje:

***El estudiante*:**

- Comprende qué es un algoritmo y sus características
- Distingue entre las diferentes representaciones de algoritmos
- Utiliza pseudocódigo para describir algoritmos
- Analiza las ventajas del pseudocódigo frente a los diagramas de flujo

3.1. Introducción

En el ámbito de la informática, los algoritmos son fundamentales, ya que proporcionan un marco estructurado para el procesamiento de datos y la toma de decisiones. Su importancia radica en que permiten a los desarrolladores diseñar soluciones eficientes y efectivas para una amplia variedad de problemas, desde cálculos matemáticos hasta la gestión de bases de datos y la inteligencia artificial.

Además, los algoritmos son esenciales para optimizar recursos, mejorar el rendimiento de sistemas y garantizar la precisión en la ejecución de tareas

3.2. Definición de Algoritmo

Un algoritmo es una receta paso a paso para resolver un problema o o llevar a cabo una tarea específica. Piensa en él como una serie de instrucciones precisas que una máquina (o una persona) puede seguir para obtener un resultado específico.

Ejemplos cotidianos:

- **Buscar una palabra en un diccionario:** Abrir el diccionario, encontrar la letra inicial, buscar la palabra en orden alfabético y leer su definición.
- **Hacer una llamada telefónica:** Desbloquear el teléfono, abrir la aplicación de contactos, buscar el contacto, presionar el botón de llamada y esperar a que conecte.
- **Atarse los zapatos:** Pasar un cordón por un ojal, cruzar los cordones, hacer un lazo y apretarlo.

Ejemplos en computación:

- **Control de procesos:** Los algoritmos controlan procesos industriales de manera automatizada.
- **Recomendar productos en una tienda en línea:** Los algoritmos de recomendación analizan el historial de compras y las preferencias del usuario para sugerir productos relevantes.
- **Motor de búsqueda de Google:** Utiliza algoritmos complejos para indexar y clasificar páginas web.

- **Recomendaciones de Netflix:** Emplea algoritmos para sugerir películas y series basadas en tus preferencias.
- **Rutas de GPS:** Utilizan algoritmos para encontrar la ruta más corta entre dos puntos.
- **Sistemas de reconocimiento facial:** Emplean algoritmos para identificar personas en imágenes.

Lo interesante de los algoritmos que no existe una única solución, se puede resolver el problema por medio de varias formas, siempre y cuando se logre el objetivo.

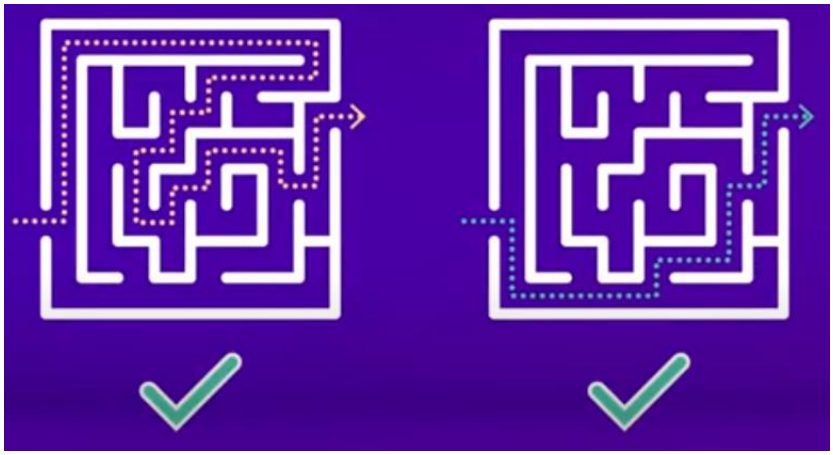


Figura 28 Formas de resolver un problema

3.3. Características de los algoritmos

Para realizar un algoritmo adecuado se debe cumplir con algunas características

Característica	**Descripción**	**Ejemplo (Calcular el promedio de tres números)**
Precisión	Cada instrucción debe ser clara y sin ambigüedades.	Solicitar al usuario que ingrese tres números.
Finito	El algoritmo debe terminar en un número finito de pasos.	Mostrar el resultado del promedio.

Definido	Cada paso debe tener un significado único.	Sumar los tres números ingresados y dividirlo por 3
Entrada	El algoritmo puede recibir datos para procesar.	Entrada: Tres números ingresados por el usuario.
Salida	El algoritmo produce un resultado.	Salida: El promedio de los tres números.
Efectividad	El algoritmo debe ser capaz de ser ejecutado.	Se puede ejecutar en una calculadora o en un programa de computadora.
Generalidad	El algoritmo debe poder aplicarse a una clase de problemas similares.	Este algoritmo puede utilizarse para calcular el promedio de cualquier conjunto de números.

Tabla 4 Características de los algoritmos

Otras características importantes de los algoritmos es que deben ser eficientes, flexibles, modulares

3.4 Estructura de un Algoritmo

Los algoritmos están compuestos en 3 partes fundamentales: Entrada, Proceso, y Salida que permiten dar la solución a un problema

Figura 29 4 Estructura de un Algoritmo

1.Inicio:

- Indica el inicio del algoritmo

2. Datos de Entrada:

- **Definición:** Son los datos que el algoritmo recibe para realizar sus cálculos o tareas. Estos datos pueden ser números, texto, imágenes, etc.
- **Ejemplo:** En un algoritmo que calcula el área de un círculo, la entrada sería el radio del círculo.

3. Proceso:

- **Definición:** Es el conjunto de instrucciones o pasos que se ejecutan sobre los datos de entrada para obtener un resultado.
- **Ejemplo:** En el algoritmo del área del círculo, el proceso sería la fórmula matemática para calcular el área ($Área = \pi * radio^2$).

4. Datos de Salida:

- **Definición:** Es el resultado final que produce el algoritmo a partir de los datos de entrada y el proceso realizado.
- **Ejemplo:** En el algoritmo del área del círculo, la salida sería el valor del área calculada.

5.Fin:

- Indica el fin del algoritmo

Tabla Resumen

Parte del Algoritmo	**Descripción**	**Ejemplo (Cálculo del área de un círculo)**

Entrada	Datos que se proporcionan al algoritmo	Radio del círculo
Proceso	Operaciones que se realizan sobre los datos	Fórmula: Área = π * radio²
Salida	Resultado final del algoritmo	Valor del área calculada

Tabla 5 Tabla resumen Estructura de un Algoritmo

3.5. Lenguajes Algorítmicos

Es una serie de símbolos y reglas que se utilizan para describir de manera explícita un proceso. Teniendo en cuenta la forma en que describen el proceso, existen dos tipos de lenguajes algorítmicos:

Gráficos: Es la representación gráfica de las operaciones que realiza un algoritmo (Diagrama de flujo y diagramas rectangulares)

No Gráficos: Representa en forma descriptiva las operaciones que debe realizar un algoritmo (pseudocodigo)

3.5.1 Diagrama de flujo

Un diagrama de flujo es una representación gráfica de un proceso o algoritmo, que utiliza símbolos y conectores para mostrar la secuencia de pasos a seguir. Es como un mapa que guía a través de las diferentes etapas de una tarea, desde el inicio hasta el final.

Un diagrama de flujo es una excelente herramienta visual para representar procesos y algoritmos de manera clara y concisa.

¿Por qué son útiles?

- **Facilitan la comprensión:** Al visualizar el proceso de manera gráfica, es más fácil entender cómo funciona.

- **Ayudan a identificar errores:** Los diagramas de flujo permiten detectar posibles fallos o ineficiencias en el proceso.
- **Promueven la comunicación:** Son una forma efectiva de comunicar un proceso a otras personas, incluso si no tienen conocimientos técnicos.
- **Documentan procesos:** Sirven como registro de cómo se realiza una tarea, lo cual es útil para futuras referencias o cambios.

Elementos de un diagrama de flujo:

- **Símbolos:** Cada símbolo representa una acción o decisión específica dentro del proceso. Los símbolos más comunes incluyen:
    - **Óvalo:** Inicio o fin del proceso.
    - **Rectángulo:** Acción o proceso.
    - **Rombo:** Decisión (sí o no).
    - **Flechas:** Indican la dirección del flujo.
- **Conectores:** Se utilizan para conectar los diferentes símbolos y mostrar la secuencia de pasos.

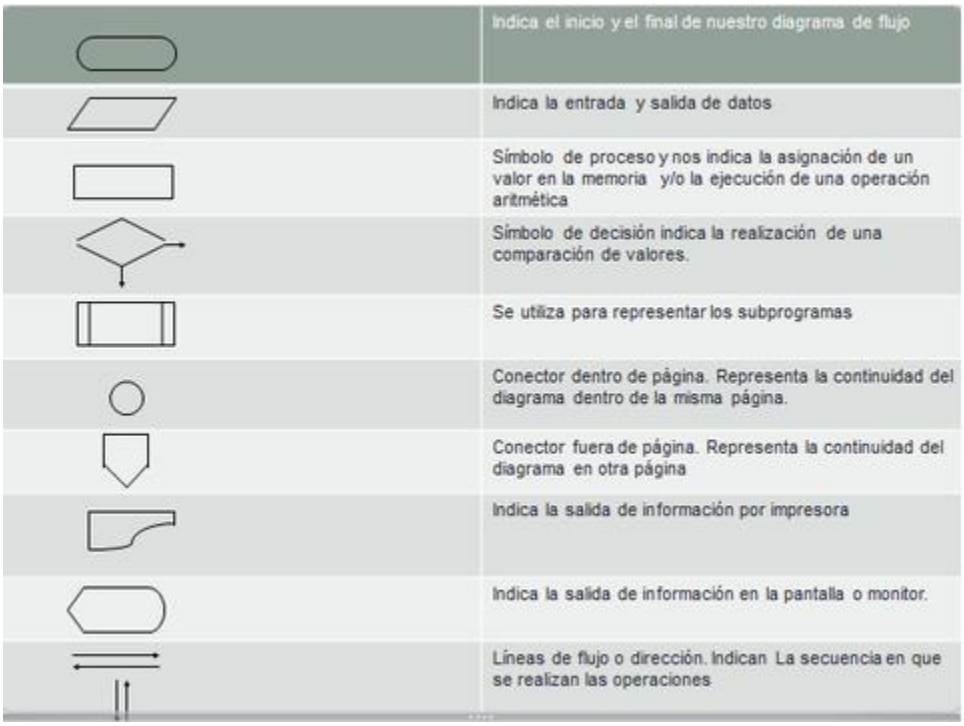


Figura 30 Símbolos diagrama de flujo

Ejemplo

Imagina que quieres crear un diagrama de flujo para indicamos si una persona puede votar o no en unas elecciones, dependiendo de su edad. Podría verse así:

Figura 31 Ejemplo Diagrama de flujo

3.5.2 Diagramas rectangulares o diagrama de Chapin

El diagrama rectangular, también conocido como diagrama de Nassi-Shneiderman o diagrama de Chapin, es una técnica de representación gráfica de algoritmos que combina la descripción textual (similar al pseudocódigo) con una estructura visual clara y concisa. A diferencia de los diagramas de flujo tradicionales, los diagramas rectangulares evitan el uso de flechas, lo que los hace más compactos y fáciles de leer.

Características Principales

- **Estructura Rectangular**: Todos los pasos del algoritmo se representan dentro de rectángulos, donde cada acción o decisión se coloca en un bloque contiguo.
- **Sin flechas:** La secuencia de ejecución se determina por la posición de los bloques, de arriba hacia abajo y de izquierda a derecha.

- **Concisión:** Ofrece una representación más compacta que los diagramas de flujo tradicionales.
- **Facilidad de lectura:** La estructura rectangular y la ausencia de flechas facilitan la comprensión del algoritmo.

Continuando con el ejercicio anterior en un diagrama rectangular quedaría de la siguiente forma.

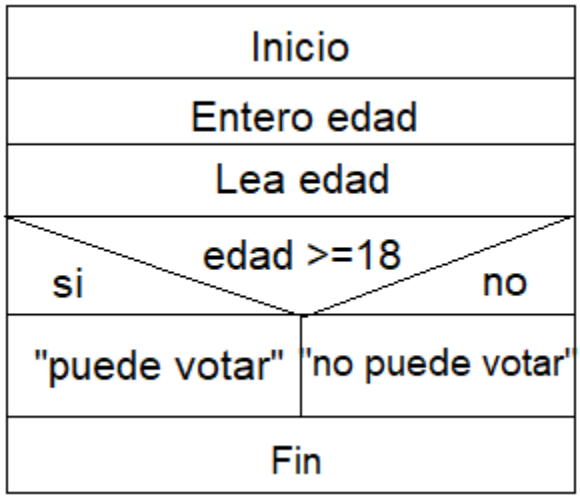


Figura 32 Ejemplo Diagramas rectangulares o diagrama de Chapin

3.5.3 Pseudocódigo

El **pseudocódigo** es una descripción de alto nivel de un algoritmo utilizando una combinación de lenguaje natural y elementos de programación.

Es la representación narrativa de los pasos que debe seguir un algoritmo para dar solución a un problema determinado. A diferencia de los lenguajes algorítmicos anteriores que usan gráficos, pseudocodigo utiliza palabras que indican el proceso a realizar.

Características principales:

- **Semiforma:** Utiliza palabras clave y estructuras de control similares a los lenguajes de programación, pero sin la rigidez de la sintaxis.

- **Independencia del lenguaje:** No está ligado a un lenguaje de programación específico, lo que facilita su comprensión por personas con diferentes conocimientos.
- **Facilidad de lectura:** Su similitud con el lenguaje natural lo hace más fácil de entender que un código fuente.

Continuando con el ejercicio anterior en un pseudocodigo quedaría de la siguiente forma

```
Inicio
entero edad
lea edad
si (edad > = 18)
    muestre (“puede votar”)
sino
    muestre (“No puede votar”)
finsi
fin
```

Figura 33 Ejemplo Pseudocódigo

3.6. Ventajas de utilizar seudocódigo a diagrama de flujo

Tanto los diagramas de flujo como el pseudocódigo son herramientas excelentes para comunicar la lógica de un algoritmo a otros programadores, analistas o incluso a personas no técnicas

Algunas ventajas de usar pseudocodigo a diagrama de flujo son:

- **Mayor detalle y precisión:** El pseudocódigo permite expresar de manera más detallada y precisa los pasos de un algoritmo. Puedes incluir expresiones matemáticas, llamadas a funciones y otras operaciones que son difíciles de representar gráficamente en un diagrama de flujo.
- **Flexibilidad:** El pseudocódigo no está limitado a una representación visual específica. Puedes utilizar cualquier estructura de control o sintaxis que te resulte más cómoda, adaptándolo a tu estilo de programación.
- **Fácil de escribir y modificar:** El pseudocódigo se escribe utilizando un lenguaje similar al natural, lo que facilita su creación y modificación. No requiere herramientas especializadas como los diagramas de flujo.
- **Integración con el código:** El pseudocódigo puede ser fácilmente traducido a un lenguaje de programación específico, ya que su sintaxis suele ser similar.
- **Mejor para algoritmos complejos:** Para algoritmos complejos con muchas condiciones y bucles anidados, el pseudocódigo puede ser más fácil de leer y comprender que un diagrama de flujo, que puede volverse muy complejo y difícil de seguir.

Pseudocódigo, es el lenguaje algorítmico que vamos a utilizar en este curso para la solución de los ejercicios propuestos. Sin embargo, utilizaremos los conceptos de la programación orientada a objetos (POO). Este concepto se verá en el próximo capitulo.

3.7 Cuestionario

4

UNIDAD 4 PROGRAMACION ORIENTADA A OBJETOS

CONTENIDO

4.1 Introducción

4.2 Los paradigmas de programación y sus diferencias

4.2.1 Secuencial

4.2.2 Estructurado

4.2.3 Programación Orientado a Objetos

4.3. Definición de Programación orientado a objeto (POO)

4.4. Ventajas de la programación orientado a objeto

4.5 Elementos Básicos de la POO

4.5.1 Clase

4.5.1.1 Estructura general de una clase

4.5.1.2 Elementos de una clase

4.5.1.2.1 Atributos

4.5.1.2.2 métodos

4.5.1.2.2 .1 Estructura general de un método

4.5.2 Objetos e instancia

4.6 Standares para dar nombre a las clases, métodos y atributos

4.7 UML

4.7.1 Diagramas UML

4.7.1.1 Diagramas Estructurales

4.7.1.2 Diagramas de Comportamiento

4.7.2Diagrama de clase

4.8 Cuestionario


Resultado de Aprendizaje:

El estudiante:

4.1 Introducción

La Programación orientada a objetos nos permite modelar el mundo real de una manera más intuitiva, creando objetos que representan cosas como personas, animales, vehículos o conceptos más abstractos. Esta forma de programar facilita la creación de software más modular, reutilizable y fácil de mantener, ya que permite descomponer problemas complejos en partes más pequeñas y manejables. La POO es fundamental en el desarrollo de software moderno y se utiliza en la mayoría de los lenguajes de programación populares, como Java, C++, Python y muchos otros.

4.2. Los paradigmas de programación y sus diferencias

Un paradigma de programación es una forma de trabajo, es como un conjunto de reglas y convenciones que nos dicen cómo pensar y organizar nuestro código. Es una forma de abordar la resolución de problemas a través de la programación, como si fuera un estilo arquitectónico para construir software.

Imagina que estás construyendo una casa. Puedes hacerlo siguiendo un estilo clásico, moderno o rústico. Cada estilo tiene sus propias características y formas de organizar los espacios. De la misma manera, en programación, cada paradigma ofrece una perspectiva diferente para estructurar el código.

Existen muchos paradigmas de programación, entre los cuales están:

4.2.1 Secuencial

En términos simples, la programación secuencial ejecuta instrucciones una tras otra, siguiendo un flujo lineal. Es como una cinta transportadora en una fábrica: cada producto pasa por cada estación de trabajo en un orden fijo.

Ejemplo

Se desea saber cuál es el área de un círculo que tiene un radio determinado

En este ejemplo se ingresa el radio, luego se realiza la operación correspondiente (fórmula matemática) y por último se muestra el resultado. Como puedes observar es de forma lineal

Inicio

real radio, area

Lea radio

area = 3.1416 * radio * radio

muestre "El área del círculo es:", area

Fin

4.2.2 Estructurado

En la programación estructurada, la **función** (o **subprograma**) es la unidad básica de código reutilizable. Imagina una función como un pequeño programa dentro de un programa más grande. Cada función se encarga de realizar una tarea específica, y puede ser llamada desde cualquier parte del programa principal o desde otras funciones.

Características de las funciones:

- **Modularidad:** Dividen el programa en partes más pequeñas y manejables.
- **Reutilización:** Una misma función puede ser llamada varias veces desde diferentes puntos del programa.
- **Abstracción:** Ocultan la complejidad interna de una tarea, exponiendo solo la interfaz necesaria para su uso.
- **Parametrización:** Permiten pasar datos (parámetros) a la función para personalizar su comportamiento.
- **Retorno de valores:** Muchas funciones devuelven un resultado al programa que las llamó.

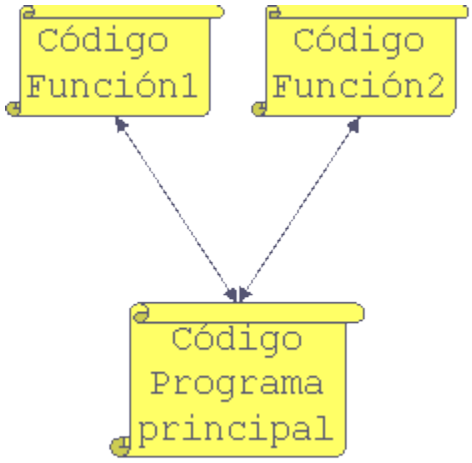


Figura 34 paradigma estructurado

4.2.3 Programación orientada a objetos (POO)

En la programación orientada a objetos (POO), el **objeto** es el elemento central y fundamental del paradigma. Un objeto es una instancia de una **clase**, que actúa como un molde o plantilla que define un conjunto de **atributos** (datos) y **métodos** (funciones o procesos). Esta combinación de datos y comportamientos en una única entidad permite una mejor organización y manipulación del código.

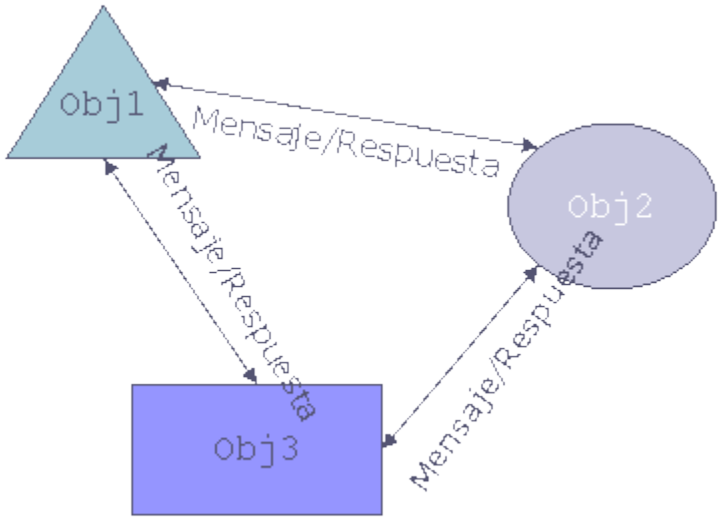


Figura 35Paradigma Orientado a Objetos

Este paradigma de programación es el usaremos en este curso

Vamos a profundizar más sobre este concepto

4.3. Definición de programación orientado a objeto (POO)

En el contexto de la programación orientada a objetos (POO), el objeto representa el componente fundamental del paradigma. Se entiende como una instancia específica de una clase, que funciona como un modelo o plantilla. Esta clase define un conjunto de atributos (que son los datos) y métodos (que son las funciones o procesos). La fusión de estos datos y comportamientos en una sola entidad, es decir, el objeto, permite una organización más eficaz y un manejo más sencillo del código.

Imagina que estás construyendo una casa. La clase sería el plano de la casa, definiendo cuántas habitaciones tiene, qué tamaño tienen, etc. El objeto sería una casa concreta construida a partir de ese plano, con sus propias características específicas (color, materiales, etc.).

La POO permite crear objetos reutilizables. Una vez definida una clase, puedes crear múltiples objetos a partir de ella, cada uno con sus propios valores para los atributos, pero compartiendo los mismos métodos.

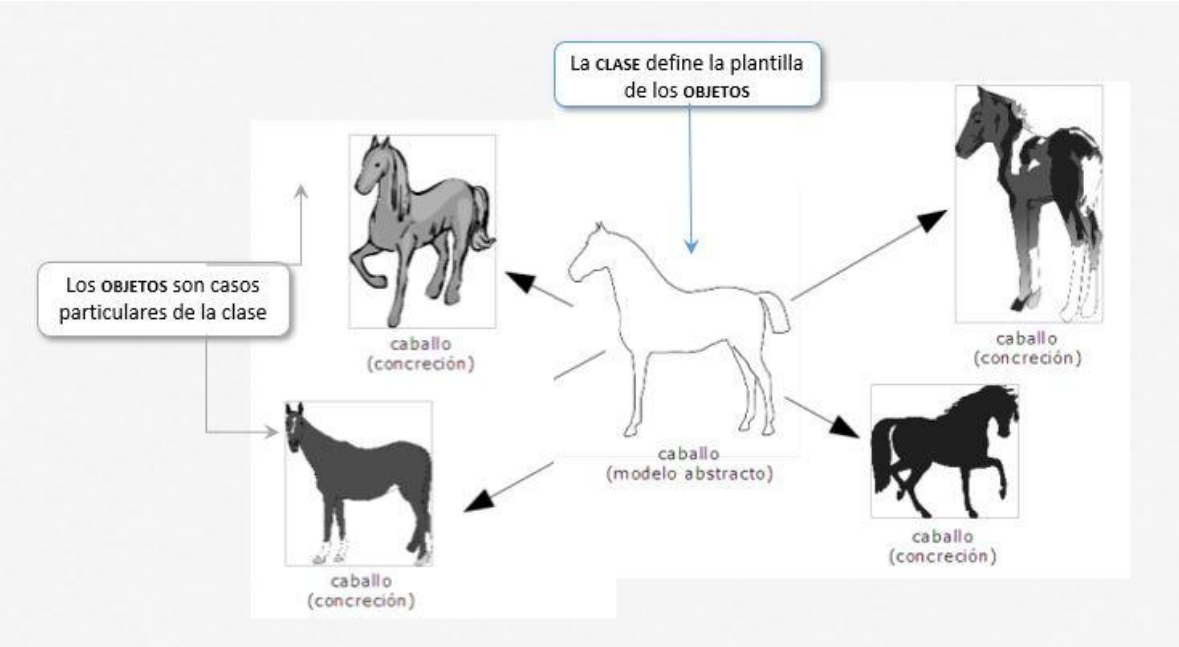


Figura 36 Ejemplo de Programación orientada a objetos

4.4. Ventajas de la programación orientado a objeto

La programación orientada a objetos (POO) ofrece múltiples ventajas que la han consolidado como un paradigma fundamental en el desarrollo de software. A continuación, se presentan las principales ventajas:

Reusabilidad: La POO permite crear clases que pueden ser reutilizadas en diferentes partes del mismo programa o incluso en otros proyectos. Esto evita la duplicación de código y facilita el mantenimiento, ya que los desarrolladores pueden usar y extender clases existentes sin necesidad de reescribir código

Facilita el trabajo en equipo: Al estructurar el código en clases y objetos, varios desarrolladores pueden trabajar simultáneamente en diferentes partes del mismo proyecto sin interferir entre sí. Esto mejora la colaboración y acelera el proceso de desarrollo

Modificabilidad: La POO facilita la adición, eliminación o modificación de objetos dentro del sistema. Esto significa que los desarrolladores pueden implementar

cambios de manera más sencilla y rápida, lo que es crucial en entornos donde los requisitos pueden cambiar con frecuencia

4.5. Elementos básicos de la programación orientado a objetos

4.5.1. Clase

Una clase puede considerarse como un modelo o plantilla que define un conjunto de atributos (variables de clase) y comportamientos (métodos) que caracterizan a los objetos o instancias que se generan a partir de ella. Las clases son esenciales en la programación orientada a objetos (POO) y se utilizan para representar entidades o conceptos del mundo real en el código.

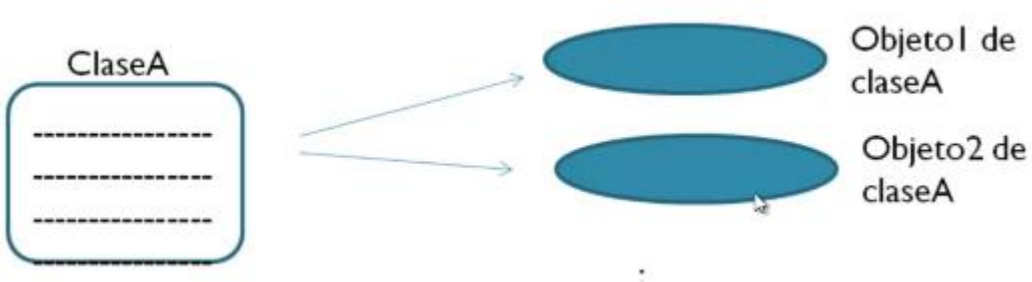


Figura 37Clase en programación orientado a objetos

A partir de una clase se pueden crear múltiples objetos

4.5.1.1 La estructura general de una clase

< Modificadores de acceso> **Clase** <*Nombre de la clase*>

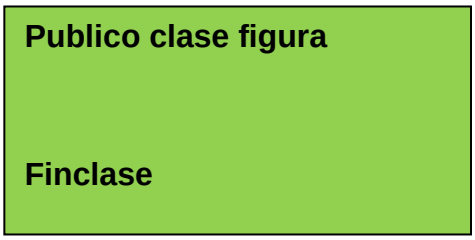
FinClase

Requiere el uso del modificador de acceso (público, privado o protegido) para especificar visibilidad, seguido la palabra reservada '**Clase'** y el nombre de la clase La clausura de la clase se indica con la palabra reservada 'FinClase'.

Los especificadores o modificadores de acceso determinan el tipo de acceso a la clase (público, privado o protegido)

- **Publico:** podemos acceder a las propiedades y métodos desde cualquier lugar, desde la clase actual, clases que heredan de la clase actual y desde otras clases.
- **Protegido**: se puede acceder al atributo o método desde la clase que lo define y desde cualquier otra que herede de esta clase.
- **Privado**: los atributos o métodos solo son accesibles desde la clase que los define.

De este modo la estructura de la clase quedaría como esta en el siguiente ejemplo



4.5.1.2 Elementos de una clase

Como se explicó anteriormente las clases están compuestas por atributos y métodos

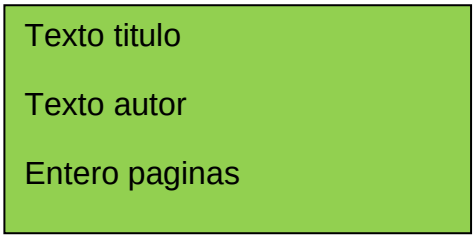
4.5.1.2.1 Atributos

Los atributos son las características o propiedades que definen a un objeto dentro de un programa orientado a objetos. Imagina que estás creando un programa para gestionar una librería. Un objeto podría ser un "Libro". Los atributos de este objeto podrían ser: título, autor, ISBN, número de páginas, género, etc.

En términos más técnicos, los atributos son variables que pertenecen a una clase y que almacenan los datos específicos de cada instancia (objeto) de esa clase.

Para crear los atributos hay que indicar el tipo de dato, (esto lo veremos con mayor detalle en la próxima sección) y el nombre del atributo

Ejemplo



4.5.1.2.2 Métodos

Un método, es un segmento de código que se encuentra dentro de una clase y que ejecuta una serie de instrucciones para llevar a cabo una tarea específica. Puede aceptar cero, uno o varios valores de entrada, conocidos como argumentos o parámetros, y devuelve un único resultado. Este resultado puede ser de diferentes tipos, como: entero, real, cadena, lógico o nulo.

Por ejemplo, un método podría ser 'sumar dos números' y recibiría dos números como datos de entrada y devolvería el resultado de la suma."

Al iniciar un programa, hay un método especial que se ejecuta primero. Este método, a menudo llamado **'Principal'**, puede estar en cualquier parte de la clase y marca el inicio de la ejecución del código.

4.5.1.2.2.1 Estructura general de un método

Al igual que las clases, los métodos requieren el uso del modificador de acceso (público, privado o protegido), la palabra estático (si no esta instanciada la clase) seguido el tipo de dato del retorno, la palabra reservada **método** y, el nombre del método y la colección de parámetros dentro de los paréntesis

La clausura del método se indica con la palabra reservada “Finmetodo”.

```
<Modificador de Acceso>  <estático>  <tipo de retorno> metodo  <nombre del metodo> (parametros)
. . .
FinMetodo
```

- **Especificadores**: determinan el tipo de acceso al método (publico privado o protegido)
- **tipoderetorno**: indica el tipo del valor que devuelve el método. Es imprescindible que, en la declaración de un método, se indique el tipo de dato que ha de devolver. El dato se devuelve mediante la instrucción retorne. Si el método no devuelve ningún valor este tipo será vacío
- **nombreMetodo**: es el nombre que se le da al método. Debe ser verbos en infinitivo, debe comenzar con minúscula y la primera letra de las palabras siguientes en mayúscula

Ejemplo: run(), runFast(), calcularSalario()

- Lista de parámetros (opcional): después del nombre del método y siempre entre paréntesis puede aparecer una lista de parámetros (también llamados argumentos) separados por comas. Estos parámetros son los datos de entrada que recibe el método para operar con ellos. Un método puede recibir cero o más argumentos. Se debe especificar para cada argumento su tipo. **Los paréntesis son obligatorios, aunque estén vacíos.**

Ejemplo:

```
Publico estático vacio metodo Principal ( )
. . .
Finmetodo
```

Otro ejemplo

Este ejemplo está un poco más elaborado, con elementos que se explicaran en la medida que vamos avanzando en el curso.

```
Publico estático real método calcular (real ancho, real alto)
. . .retorne ancho * alto
Finmetodo
```

4.5.2. Objetos e instancias

En la programación orientada a objetos (POO), un **objeto** es una instancia concreta de una **clase**, que actúa como una plantilla o modelo. Cada objeto combina datos y comportamientos, encapsulando atributos (propiedades) y métodos (funciones) que operan sobre esos datos.

A los objetos creados a partir de su clase se le conoce también como instancia de la clase. Para crear instancias utilizamos el operador ***nuevo (new).***

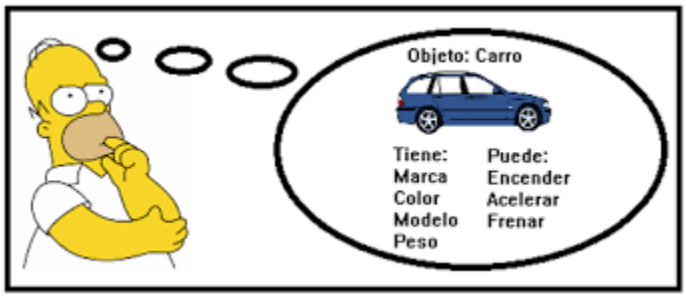


Figura 38 Ejemplo de objeto

Ejemplo de crear objetos representados en forma algorítmica

```
publico clase Persona
        texto nombre
         entero edad;
    publico vacio metodo saludar (texto nombre, entero edad)
          esteobjeto.nombre= nombre
         esteobjeto.edad=edad
            imprima ("Hola, soy " + nombre + " y tengo " + edad + " años.")
    finmetodo
finclase
```

cómo puedes observar la clase Persona contiene 2 atributos (nombre y edad) y un método llamado saludar el cual recibirá por parámetro 2 valores

ahora a partir de otra clase vamos a instanciar la clase Persona, es decir vamos a crear un objeto

```
publico clase  Main
    publico estatico vacio principal()
          Persona persona1 = nuevo Persona()
          persona1.saludar(“Maria”, 30)
         Persona persona2 = nuevo Persona()
```

persona2.saludar(“Juan”,25)

Finmetodo

finclase

En este ejemplo hemos creado 2 objetos (persona1 y persona2). Al ejecutar el programa el resultado sería:

Hola, soy María y tengo 30 años

Hola, soy Juan y tengo 25 años

Cada objeto guarda cierta información por eso se diferencia el uno del otro. En otro capítulo explicaremos lo que son los métodos constructores, sobrecarga de métodos y otros conceptos interesantes en la POO.

Además de estos elementos, la POO se basa en tres principios fundamentales: **Herencia, encapsulación y polimorfismo** se verán en el capítulo 8 de este curso

4.6 Standares para dar nombre a las clases, métodos y variables

Los **estándares de codificación** son un conjunto de reglas, directrices y mejores prácticas que guían a los desarrolladores en la escritura, formateo y documentación del código. Su objetivo principal es crear un código más limpio, legible y mantenible, lo que facilita la colaboración entre los miembros de un equipo y mejora la calidad general del software.

- **Nombres de clases**

  Los nombres de clases deben ser mezclas de mayúsculas y minúsculas, con la primera letra de cada palabra interna en mayúsculas (EjemploSecuenciales).

- **Métodos**

Los métodos deberán ser verbos (en infinitivo), en mayúsculas y minúsculas con la primera letra del nombre en minúsculas, y con la primera letra de cada palabra interna en mayúsculas (calcularSalario).

- **Variables**

  Los nombres de las variables reciben el mismo tratamiento que para los métodos, con la salvedad de que aquí sí importa más la relación entre la regla mnemónica (es decir dar un nombre coherente) y la longitud del nombre.

  Ejemplo:

  Correctos: *diaCalculo*, *fechaIncoporacion*

  Incorrectos: *dC*, *DCal*, *fI*, *FI¿*

4.7 UML

UML (Unified Modeling Language) es un lenguaje de modelado estandarizado que se utiliza para visualizar, especificar, construir y documentar los componentes de sistemas de software. UML es ampliamente utilizado en el desarrollo de software orientado a objetos y también se puede aplicar en otros dominios como el diseño de sistemas y procesos empresariales.

Su objetivo principal es proporcionar una forma común para que los desarrolladores, analistas, diseñadores y otras partes interesadas puedan comunicarse y entender la estructura y el comportamiento de un sistema.

4.7.1 Diagramas UML

UML incluye diferentes tipos de diagramas que se agrupan en dos categorías principales: diagramas estructurales, diagramas de comportamiento

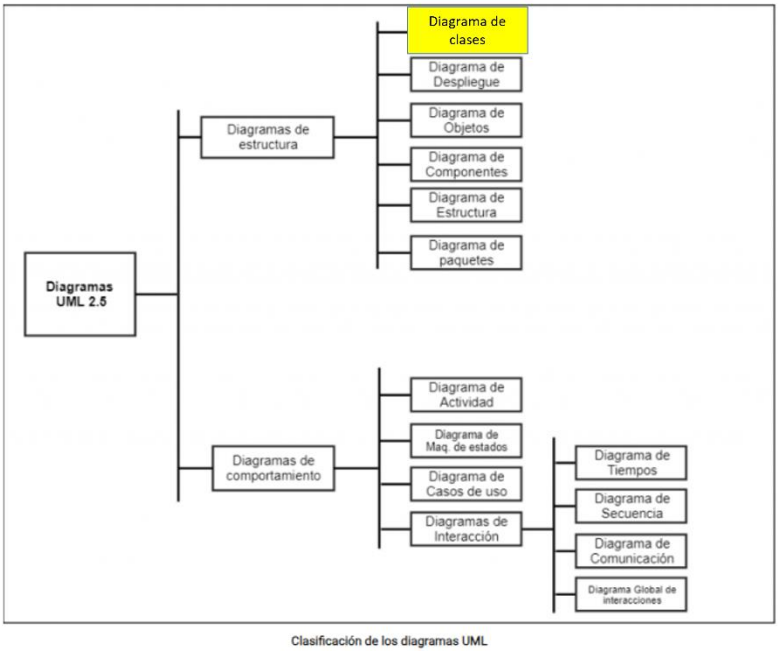


Figura 39 Tipos de diagramas UML

4.7.1.1 Diagramas Estructurales

Estos diagramas representan los componentes estáticos de un sistema y sus relaciones. Los tipos incluyen:

- **Diagrama de Clases**: Muestra las clases del sistema, sus atributos, métodos y relaciones entre ellas.
- **Diagrama de Componentes**: Representa la organización y dependencias entre los componentes del sistema.
- **Diagrama de Despliegue**: Ilustra cómo se distribuyen los componentes de software en el hardware.
- **Diagrama de Objetos**: Presenta instancias específicas de clases en un momento dado.
- **Diagrama de Paquetes**: Organiza las clases en grupos lógicos, mostrando sus dependencias.

- **Diagrama de Estructura Compuesta**: Detalla la estructura interna de una clase y su colaboración con otros componentes.
- **Diagrama de Perfiles**: Permite personalizar UML para necesidades específicas mediante estereotipos y extensiones

4.7.1.2 Diagramas de Comportamiento

Estos diagramas capturan los aspectos dinámicos del sistema, mostrando cómo interactúan los componentes. Los tipos incluyen:

- **Diagrama de Casos de Uso**: Representa las interacciones entre actores (usuarios) y el sistema, mostrando las funciones que ofrece.
- **Diagrama de Secuencia**: Muestra el orden en que ocurren las interacciones entre objetos a lo largo del tiempo.
- **Diagrama de Comunicación**: Enfocado en las interacciones entre objetos y el intercambio de mensajes.
- **Diagrama de Actividades**: Modela flujos de trabajo y procesos dentro del sistema.
- **Diagrama de Máquina de Estados**: Representa los estados posibles de un objeto y las transiciones entre ellos.
- **Diagrama de Tiempos**: Muestra el comportamiento de los objetos a lo largo del tiempo, con énfasis en restricciones temporales.
- **Diagrama General de Interacción**: Combina elementos de otros diagramas para representar escenarios complejos

Para efectos de este curso comenzaremos con el diagrama de clase

4.7.2 Diagrama de clase

El **diagrama de clases** es un tipo de diagrama estructural en UML que representa las clases de un sistema, sus atributos, métodos y las relaciones entre ellas. Es uno

de los diagramas más importantes en el modelado orientado a objetos, ya que muestra la estructura estática del sistema.

¿Para qué sirve el Diagrama de Clases?

- **Diseño del sistema:** Ayuda a visualizar la estructura del sistema antes de implementarlo.
- **Documentación:** Sirve como una guía para desarrolladores y otros interesados en entender el sistema.
- **Comunicación:** Facilita la discusión entre equipos sobre las características y relaciones del sistema.
- **Base para la codificación:** Se utiliza como referencia directa para implementar el código en lenguajes orientados a objetos.

Partes del Diagrama de Clases

Un diagrama de clases tiene varios componentes clave:

Una clase es el elemento principal. Se representa como un rectángulo dividido en tres secciones:

- **Nombre de la clase:** En la parte superior, el nombre de la clase (en negrita y centrado).
- **Atributos:** En la sección intermedia, lista de atributos (propiedades o variables de la clase).
- **Métodos:** En la parte inferior, lista de métodos (funciones o comportamientos de la clase).

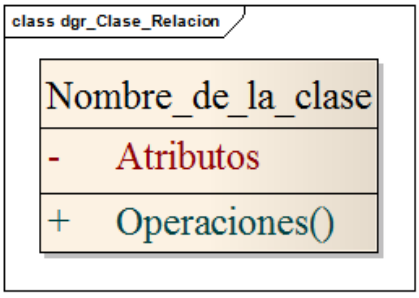


Figura 40 Estructura de una Clase

Ejemplo de representación de una clase:

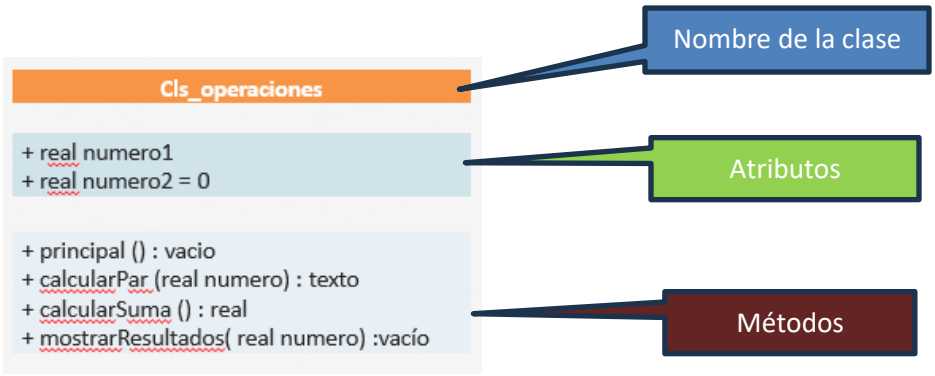


Figura 41 Ejemplo de una Clase

Las clases se pueden relacionar entre ellas, estas relaciones la veremos en el capítulo 7 de este curso

4.8 Cuestionario

5

UNIDAD 5 ENTIDADES PRIMITIVAS PARA EL DESARROLLO DE ALGORITMOS

CONTENIDO

5.1 Introducción
5.2 Identificadores como localidades de memoria (variables)
5.3 Reglas para formar un Identificador
5.3.1 palabras reservadas
5.4 Declarar y definir variables
5.4.1 Tipos de datos
5.4.2 Clasificación de las variables según su ámbito
5.5 Expresiones
5.6 Operadores Aritméticos
5.6.1 Jerarquía de los operadores Aritméticos
5.7 Convertir una expresión algebraica a expresión algorítmica
5.8 Ejercicios Resueltos
5.9 Ejercicios Propuestos
5.10 Cuestionario

 Resultado de Aprendizaje:

El estudiante:

5.1 Introducción

Las entidades primitivas para el desarrollo de algoritmos son los elementos básicos y fundamentales que se utilizan para construir y estructurar algoritmos en programación. Estos pueden incluir tipos de datos como números, texto y valores booleanos, así como variables y constantes que almacenan información. Comprender a fondo estas entidades es esencial para diseñar algoritmos eficientes y precisos.

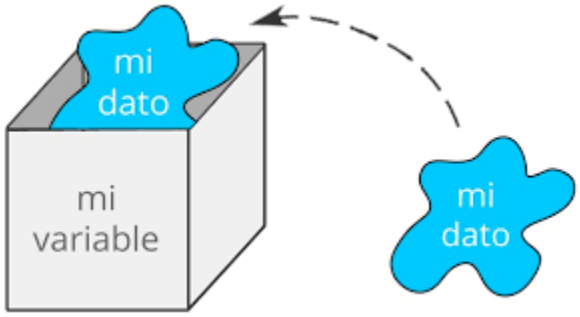
5.2 Identificadores como localidades de memoria (variables)

Todo dato que se utiliza en un programa debe ser almacenado en la memoria RAM del computador. Esa memoria está dividida en “compartimientos” del mismo tamaño en los que se almacena información. El tamaño de esos compartimentos está definido en bits, lo que indica cuál es el valor máximo que se puede almacenar en ellos. También, cada compartimiento tiene asociado una dirección única que permite a un programa (incluido el sistema operativo) acceder a la información almacenada bien sea para consultarla, modificarla o borrarla. Esas direcciones de memoria están representadas normalmente por un código hexadecimal, como se ejemplifica a continuación.

Contenido de la RAM	Dirección Física
10000010	0x00A01
00110011	0x00A02
01110111	0x00A03

Por la complejidad de manejar las direcciones de la memoria RAM, el programador usa en su lugar identificadores para almacenar los datos del programa en la memoria RAM sin preocuparse de las direcciones en hexadecimal. Así, cuando hablamos de una ***variable*** nos referimos a un identificador que nos permite almacenar un valor en la memoria RAM, el cual que puede cambiar durante la ejecución del algoritmo o programa. Por otro lado, cuando hablamos de una ***constante*** nos referimos a un identificador que nos permite almacenar en la RAM un valor que permanece constante durante la ejecución del algoritmo o programa.

Para poder reconocer una variable en la memoria de la computadora, es necesario darle un nombre con el cual podamos identificarla dentro de un algoritmo



5.3 Reglas para formar un Identificador

El identificador de una variable debe cumplir lo siguiente:

- **Inicio con Letra o Guion Bajo**: Un identificador debe comenzar con una letra (A-Z, a-z) o un carácter de subrayado (_). No puede comenzar con un número
- **Caracteres Permitidos**: Después del primer carácter, un identificador puede contener letras, números y guiones bajos. No se permiten espacios en blanco ni caracteres especiales como @, #, $, %, etc.
- **Distinción entre mayúsculas y minúsculas:** Los identificadores son sensibles a mayúsculas y minúsculas. *nombre* y *Nombre* son dos variables diferentes
- **Palabras reservadas:** No puedes usar palabras reservadas como identificadores. Por ejemplo, *clase, método, principal, vacío, para, mientras etc*
- **Nombres descriptivos:** Elige nombres que sean claros y descriptivos de la variable que representan. Por ejemplo, nombre_cliente es mejor que n
- **Evitar nombres genéricos:** No uses nombres como x, y, temp, a menos que su propósito sea muy obvio en un contexto pequeño.
- **Una variable** solo puede almacenar un único valor

- **Estilo de nomenclatura:** Sigue un estilo de nomenclatura consistente en todo tu código. Algunos estilos comunes son:

Camel case: Cada palabra, excepto la primera, comienza con una letra mayúscula, sin espacios ni guiones.

nombreCliente, calcularArea

Snake case: Todas las letras son minúsculas y las palabras están separadas por guiones bajos

nombre_cliente, calcular_area

Pascal case: Similar a Camel Case, pero la primera palabra también comienza con una letra mayúscula

NombreCliente, CalcularArea

Ejemplos

Válidos

- nombre
- _contador
- edad_cliente
- CostoArticulo
- var1

Inválidos

- 1nombre (comienza con un número)
- nombre cliente (contiene un espacio en blanco)
- total$ (contiene un carácter especial)
- if (palabra reservada)
- #alumnos (contiene un carácter especial)
- var!able (contiene un carácter especial)

5.3.1 Palabras reservadas

Las palabras reservadas en programación son términos que tienen un significado especial y predefinido en un lenguaje de programación específico. Estas palabras no pueden ser utilizadas como identificadores, como nombres de variables o funciones, ya que su uso está reservado para funciones específicas dentro del lenguaje, lo que garantiza la coherencia y el correcto funcionamiento del código

Ejemplo de palabras reservadas en español

Abstracta	**metodo**	**boleano**	**Salto**	**Byte**
Caso	**finmetodo**	**caracter**	**clase**	**Finclase**
continue	**En otro caso**	**finpara**	**principal**	**Sino**
finsi	**hereda**	**final**	**finmientras**	**Real**
para	**longitud**	**si**	**imprima**	**Import**
formatear	**entero**	**texto**	**Lea**	**Haga**
nuevo	**paquete**	**privado**	**protegido**	**Publico**
retorne	**potencia**	**estatico**	**mientras**	**Super**
segun	**finsegun**	**esteobjeto**	**establecer**	**Obtener**
verdadero	**nulo**	**vacio**	**Falso**	

Algunas palabras reservas en ingles 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**
var	**true**	**null**	**False**	

5.4 Declarar y definir variables

Antes de usar una variable o una constante debemos declararla, es decir, antes de usarlas debemos indicarle a la máquina que separe un espacio en memoria para almacenar un valor en estas. El espacio que se le asigna a una variable o constante depende de su tipo de dato.

Declarar es darle un nombre y definir es darle un tipo de dato

Veamos cuales son los tipos de datos comúnmente usados en los algoritmos

5.4.1 Tipos de datos

Tipos de datos numéricos: existen dos tipos de datos numéricos: enteros y reales.

- **Entero:** una variable declarada como Entero sólo puede almacenar números enteros, es decir, números que no tienen decimales. Este tipo de variables suele ocupar 32 bits de la memoria RAM.
- **Real:** una variable declarada como Real puede almacenar número con coma flotante, incluyendo números enteros. Este tipo de variables tiene un rango mayor que las variables de tipo Entero puesto que suelen ocupar 64 bits.
- ▪ **Tipo de dato Lógico o boleano :** las variables y constantes con este tipo de dato suelen ocupar 1 bit en la memoria RAM puesto que son variables que toman uno de dos valores: verdadero o falso.
- ▪ **Tipo de dato carácter**: las variables y constantes con este tipo de dato suelen ocupar 8 bits en la memoria RAM. Un carácter es cualquier número, letra o símbolo. Los valores de tipo Carácter se encierran en comillas simples, por ejemplo: ‘A’, ‘3’, ‘#’.
- ▪ **Tipo de dato cadena o texto**: las variables y constantes de tipo Cadena no tienen un tamaño fijo definido en la memoria RAM, esto porque tienen una cantidad indeterminada de caracteres. Los valores de tipo Cadena se encierran en comillas dobles, por ejemplo: “Esto es una cadena”, “Hola”, “El resultado es: ”.

Una vez identificados los tipos de datos, podemos definir una forma general para declarar variables y constantes en pseudocódigo:

Tipo_de_dato identificador= valor_inicial

Un ejemplo particular de esta forma general es la declaración de una variable entera a la que llamaremos x, además tiene valor inicial 7

5.4.2 Clasificación de las variables según su ámbito

Las variables se pueden clasificar según su ámbito en:

- Variables miembros de una clase o atributos de una clase
- Variables locales
- Variables de bloque

Variable miembro o atributos de una clase

Son las declaradas dentro de una clase y fuera de cualquier método.

Aunque suelen declararse al principio de la clase, se pueden declarar en cualquier lugar siempre que sea fuera de un método. Son accesibles en cualquier método de la clase.

Son referenciadas en el diagrama de Clases.

variables locales

Son las declaradas dentro de un método. Su ámbito comienza en el punto donde se declara la variable.

Están disponibles desde su declaración hasta el final del método donde se declaran. No son visibles desde otros métodos.

Distintos métodos de la clase pueden contener variables con el mismo nombre. Se trata de variables distintas. El nombre de una variable local debe ser único dentro de su ámbito.

Si se declara una variable local con el mismo nombre que una variable miembro de la clase, la variable local tiene preferencia.

Se crean en memoria cuando se declaran y se destruyen cuando acaba la ejecución del método.

Variables de bloque

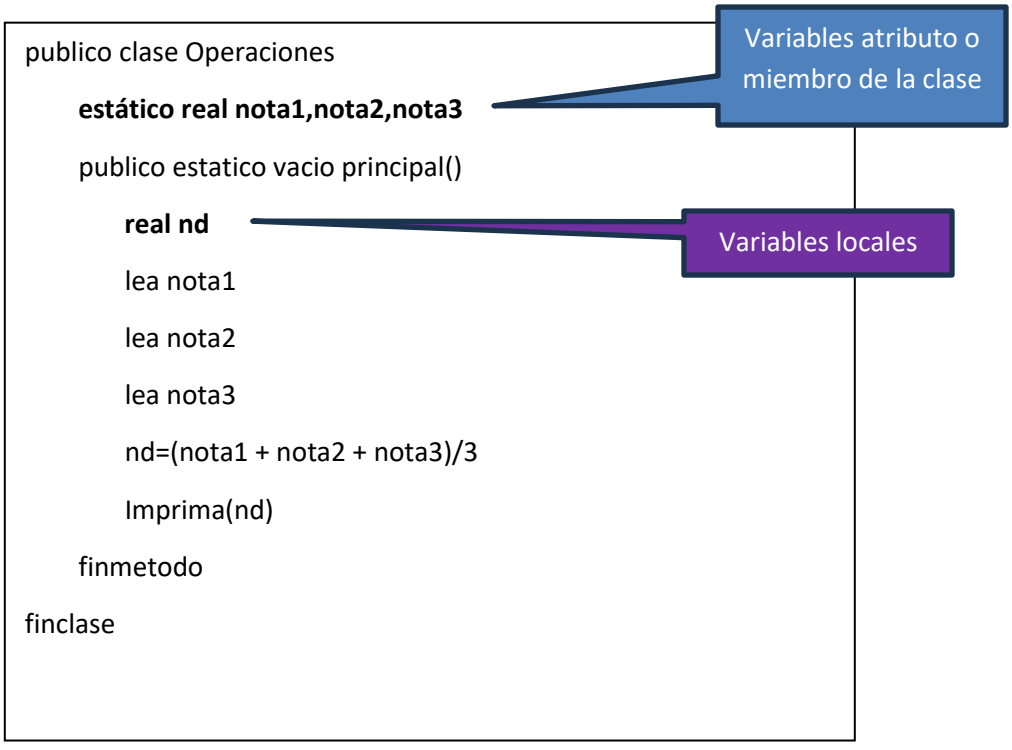
Son las declaradas dentro de un bloque de instrucciones. Su ámbito comienza en el punto donde se declara la variable.

Están disponibles desde su declaración hasta el final del bloque donde se declaran.

No son visibles desde otros bloques.

Distintos bloques pueden contener variables con el mismo nombre. Se trata de variables distintas. Se crean en memoria cuando se declaran y se destruyen cuando acaba la ejecución del bloque.

Estas variables las veremos con mas detalle en el capítulo 6



5.5 Expresiones

Las expresiones son combinaciones de constantes, variables, símbolos de operación, paréntesis y nombres de funciones especiales, cuyo objetivo es proporcionar un valor. Por ejemplo:

a + (b + 3) / c Aquí los paréntesis indican el orden del cálculo.

Una expresión consta de operadores y operandos. Según sea el tipo de datos que manipulan, se clasifican las expresiones en:

Expresiones aritméticas: Realizan operaciones matemáticas como suma, resta, multiplicación y división.

Expresiones relacionales: Comparan dos valores y devuelven un valor booleano (verdadero o falso).

Expresiones lógicas: Combinan valores booleanos utilizando operadores lógicos como "y", "o" y "no".

Expresiones de asignación: Asignan un valor a una variable

5.6 Operadores Aritméticos

Los operadores aritméticos son símbolos que se utilizan en programación para realizar operaciones matemáticas básicas en valores numéricos (números enteros o decimales). Estos operadores permiten llevar a cabo cálculos como sumas, restas, multiplicaciones, divisiones y otras operaciones relacionadas

Operador	Operación	Ejemplo	Resultado
+	suma	6+2	8
-	resta	9-3	6
*	multiplicación	8*4	32
/	división	9/3	3
mod	Modulo (residuo de una división	4 mod 2	0
potencia	potencia	potencia(5,2)	25
raíz	Raíz cuadrada	raíz(25)	5

5.6.1 Jerarquía de los operadores Aritméticos

En programación, al igual que en matemáticas, los operadores aritméticos tienen una jerarquía que determina el orden en que se realizan las operaciones dentro de una expresión.

La jerarquía de los operadores es la siguiente (de mayor a menor prioridad):

- **Paréntesis ( ):** Se utilizan para agrupar expresiones y forzar un orden de evaluación diferente al predeterminado.
- **Potenciación** : Eleva un número a una potencia.
- **Multiplicación, División y Módulo** : Se realizan de izquierda a derecha. El módulo, devuelve el resto de una división.
- **Suma y Resta** : Se realizan de izquierda a derecha.

Ejemplo 1:

*10 + 5 * 2*

- Primero se realiza la multiplicación: 5 * 2 = 10
- Luego se realiza la suma: 10 + 10 = 20
- Resultado = 20

Ejemplo 2:

*(10 + 5) * 2*

- Primero se realiza la suma dentro del paréntesis: 10 + 5 = 15
- Luego se realiza la multiplicación: 15 * 2 = 30
- Resultado = 30

Ejemplo 3:

20 - 10 / 2 + 3

- Primero se realiza la división: 10 / 2 = 5

- Luego se realizan la resta y la suma de izquierda a derecha: 20 - 5 = 15, y luego 15 + 3 = 18
- Resultado = 18

Ejemplo 4:

10 mod 3

- Se calcula el módulo (resto de la división): 10 dividido por 3 es igual a 3 con un resto de 1
- Resultado = 1

Ejemplo 5:

resultado = potencia(2,3) + 4

- Primero se realiza la potenciación: potencia(2,3) = 8 (2 elevado a la potencia 3)
- Luego se realiza la suma: 8 + 4 = 12
- Resultado = 12

Consejos adicionales

- Utiliza paréntesis para agrupar expresiones y asegurarte de que las operaciones se realicen en el orden deseado.
- Recuerda que la jerarquía de los operadores es fundamental para obtener los resultados correctos en tus cálculos.

5.7 Convertir una expresión algebraica a expresión algorítmica

Las expresiones algebraicas son combinaciones de números, variables y operadores que representan relaciones matemáticas. Estas expresiones se utilizan para modelar situaciones y resolver problemas en diversas áreas como matemáticas, física, ingeniería y ciencias de la computación.

Sin embargo, en un algoritmo, estas expresiones o fórmulas matemáticas no se pueden usar directamente como en los textos de matemáticas. Es necesario transformarlas a un formato que el algoritmo pueda entender y ejecutar como una instrucción. Para lograr esto, es importante considerar la jerarquía de los operadores, de modo que se obtenga el resultado esperado.

Ejemplo de convertir una expresión algebraica a una expresión algorítmica

Expresión algebraica	Expresión Algorítmica
1) $3x^2y-2x+4y$	3*potencia(x,2)*y-2*x+4*y
2) $-x^3y^3-2x+\frac{4}{y}$	-(potencia(x,3))*potencia(y,3)-2*x+4/y
3) $\frac{3x^2y-2x+4y}{\sqrt{3x-2}}$	(3*potencia(x,2)*y-2*x+4*y)/raíz(3*x-2)

5.8 Ejercicios Resueltos

Resolver las siguientes expresiones aritméticas

1 (((potencia(5,3)-5+(potencia((10/2-2),3))+12-(15*2 + 5+8 MOD 3)-(8*(2-10-5-2-6)))))

Respuesta

Paso 1: Resolver las potencias.

potencia(5,3) = 125

10 / 2 - 2 = 5 - 2 = 3

potencia(3,3) = 27

La expresión ahora es:
(((125 - 5 + 27 + 12 - (15 * 2 + 5 + 8 mod 3) - (8 * (2 - 10 - 5 - 2 - 6)))))

Paso 2: Resolver las multiplicaciones, divisiones y módulos.
15 * 2 = 30
8 mod 3 = 2
8 * (2 - 10 - 5 - 2 - 6) = 8 * (-21) = -168

La expresión queda:
(((125 - 5 + 27 + 12 - (30 + 5 + 2) - (-168))))

Paso 3: Resolver las operaciones dentro de los paréntesis.
30 + 5 + 2 = 37

La expresión ahora es:
(((125 - 5 + 27 + 12 - 37 - (-168))))

Paso 4: Resolver las sumas y restas.
125 - 5 = 120
120 + 27 = 147
147 + 12 = 159
159 - 37 = 122
122 - (-168) = 122 + 168 = 290

Respuesta final:
290

2 ((4+potencia(4,2)) + (20+5-25-potencia(4,2)-4)) / 4+ ((potencia(4,2) + 20+5)-25)-potencia(2,3)

Paso 1: Resolver las potencias.
potencia(4,2) = 16
potencia(2,3) = 8

La expresión queda:
((4 + 16) + (20 + 5 - 25 - 16 - 4)) / 4 + ((16 + 20 + 5) - 25) - 8

Paso 2: Resolver las operaciones dentro de los paréntesis.
Primero resolver los dos grupos de paréntesis principales:

En el primer grupo:
(4 + 16) = 20
(20 + 5 - 25 - 16 - 4) = -20
Entonces: (20 + -20) = 0

En el segundo grupo:
(16 + 20 + 5) = 41
(41 - 25) = 16

La expresión queda:
(0 / 4) + 16 - 8

Paso 3: Resolver las divisiones y las operaciones restantes.
0 / 4 = 0
0 + 16 = 16
16 - 8 = 8

Respuesta final 8

5.9 Ejercicios Propuestos

Ahora puedes practicar realizando los siguientes ejercicios

Resuelva las siguientes expresiones colocando al frente la respuesta	
((potencia(5,3)-5+10/2)-(potencia(2,3)+12-15*2)) + (5-(8*2-10-5-2-6))	_____
2+potencia(4,2)*3+2*6 mod 2-10+9 /2+ 4*2	_____
(2+30/2+(potencia(4,2)*3-5)) + (5*2-(20-15+potencia(3,2)))	_____
(potencia(2,4)*2+10+potencia(5,3)*2+20) + ((5*(3-potencia(5,2))-potencia(4,2)+5*2)-3)	_____
(2+ (30/2+potencia(4,2))*3-5) + ((20-15+potencia(3,2))+10*2) + (10 mod 3) - (5+ potencia(4,2))	_____

5.10 Cuestionario

6

UNIDAD 6 ESTRUCTURAS ALGORITMICAS

CONTENIDO

6.1 Introducción

6.2 Estructuras Secuenciales

6.2.1 Asignación directa y lectura de datos

6.2.2 Escritura

6.2.2.1 Formatear un numero

6.2.3 Problemas resueltos

6.2.4 Prueba de escritorio

6.3 Trabajando con 2 o más métodos

6.3.1 Paso de parámetro por valor

6.3.2 No enviar parámetros, No recibir parámetros

6.3.3 Enviar parámetros, No recibe parámetros

6.3.4 No enviar parámetros, Si recibe parámetros

6.3.5 Enviar parámetros, Recibir parámetros

6.4 Estructuras Condicionales

6.4.1 Operadores relacionales

6.4.2 Operadores lógicos

6.4.3 Estructura Condicional Simple

6.4.4 Estructura Condicional Compuesta

6.4.5 Múltiples (según)

6.4.6. Operador Ternario

6.4.7 Problemas resueltos

6.4.8 Problemas propuestos

6.5 Estructuras Repetitivas o Cíclicas

6.5.1 Contadores

6.5.2 Acumuladores

6.5.3 Ciclo para

6.5.3.1 variables tipo bloque

6.5.4 Ciclo mientras

6.5.5 Ciclo hacer mientrasque

6.5.6 Centinela

6.5.7 Suiches o banderas

6.5.8 Rompimiento de ciclos

6.5.9 Trabando con 2 o más Clases

6.5.9.1 instancia

6.5.9.2 Método constructor

6.5.9.3 sobrecarga de métodos

6.5.10 Problemas resueltos

6.5.11 Problemas propuestos

6.6 Cuestionario

 Resultado de Aprendizaje:

El estudiante

6.1 Introducción

Las estructuras algorítmicas son la base de la programación, son formas de organizar y manipular datos y operaciones dentro de un algoritmo, es decir, el orden en que se ejecutan las instrucciones. Estas estructuras permiten que los programas informáticos realicen tareas específicas de manera eficiente y efectiva, facilitando la resolución de problemas complejos. Se clasifican principalmente en tres tipos: secuenciales, condicionales y cíclicas.

6.2 Estructuras Secuenciales

Una estructura secuencial en algoritmos es aquella en la que las instrucciones de entrada de datos, proceso y datos de salida se ejecutan en el mismo orden en que han sido escritas, de manera lineal. Es el tipo de estructura más básica dentro de la programación y establece que cada acción debe completarse antes de pasar a la siguiente.

Ejemplo en Tabla

Paso	Instrucción	Acción Realizada
1	Leer número1	Usuario ingresa un valor
2	Leer número2	Usuario ingresa otro valor
3	Sumar número1 y número2	Se calcula la suma
4	Mostrar resultado	Se muestra el resultado en pantalla
5	Fin	Termina el algoritmo

La representación de la estructura secuencia en la programación orientada a objetos seria de la siguiente manera:

```
publico clase ejemplo
    publico estático vacio principal()
            Declarar y definir variables
            Datos de entrada
            Proceso
            Datos de salida
    finmetodo
finclase
```

6.2.1 Asignación directa y lectura de datos

Para ingresar valores a una variable se pueden hacer de dos formas posibles: asignación directa y lectura de datos

La asignación directa es un proceso en el cual se asigna un valor específico a una variable sin depender de su valor anterior. Este valor puede ser un dato literal (como un número o una cadena de texto) o el resultado de una expresión. Se realiza mediante el operador de asignación (=)

Sintaxis Común

```
Variable = valor
```

Ejemplo 1: Asignar un número a una variable

Entero edad = 20

nombre = “ana”

Real = 2.5

Carácter letra = ‘ a’

x= 10

Otro Ejemplo Asignar el resultado de una expresión a una variable

suma = 5 * 2 + 3 + x

La lectura de datos se refiere al proceso de obtener información desde una fuente externa (como un archivo, base de datos o entrada del usuario) y almacenarla en una variable para su posterior uso en el algoritmo. Este proceso es fundamental para que los algoritmos puedan operar con datos dinámicos. Se realiza mediante el operador (LEA)

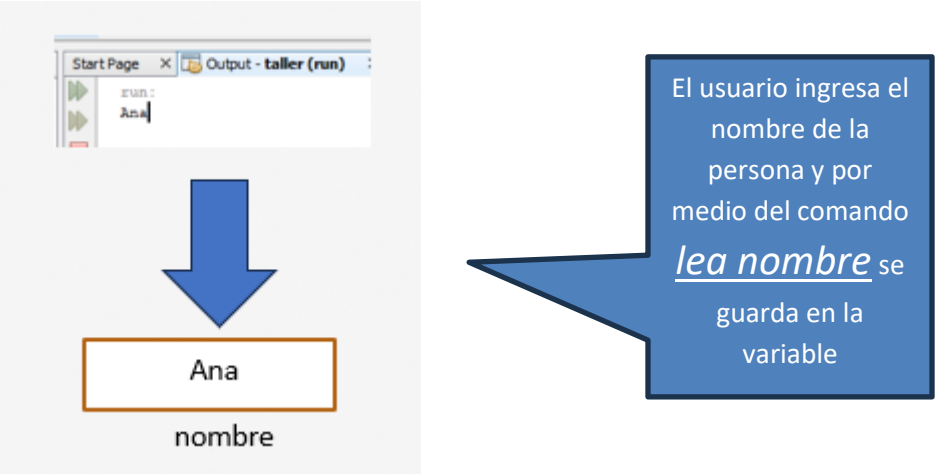
Sintaxis Común

```
Lea Variable
```

Ejemplo

```
Lea x
Lea z
suma = x + z
```

A diferencia del caso anterior, aquí el usuario ingresa los valores de x y z, y luego el programa los procesa para mostrar el resultado.



6.2.2 Escritura

El comando de escritura permite que un algoritmo presente información en la pantalla o en un archivo, facilitando la comunicación de resultados y la retroalimentación al usuario. Dependiendo del contexto, puede tener diferentes nombres como "imprimir", "mostrar" o "escribir". Para efectos de este curso utilizaremos el comando imprima

Ejemplos

Ejemplo 1: Mostrar un mensaje en la pantalla

Imprima("Hola, mundo")

Ejemplo 2: Mostrar el valor de una variable

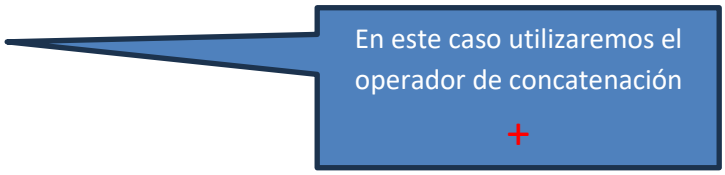
Imprima(edad)

Ejemplo 3: Mostrar el resultado de una operación

Imprima(5 * 2 + 3)

Ejemplo 3: Mostrar un mensaje y una variable)

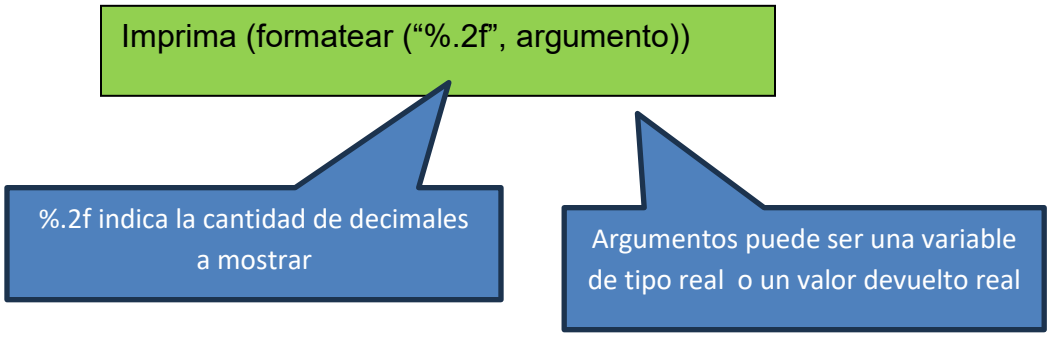
Imprima (“su edad es “ + edad)



6.2.2.1 Formatear un numero

El comando formatear se utiliza para mostrar un número con un formato específico en la consola. A continuación, se detalla su definición, funcionamiento y ejemplos ilustrativos.

Sintaxis general



Ejemplo 1: Mostrar una variable formateada con 2 decimales

Imprima (formatear (“%.2f”, salario))

Ejemplo 2: Mostrar dos variables una con 1 decimal, otra con 2 decimales

Imprima (formatear (“%.1f, valor1) + “ “ + formatear (“%.2f”, valor2))

Ejemplo 3: Otra manera de mostrar dos variables una con 1 decimal, otra con 2 decimales

Imprima (formatear (“%.1f, valor1) + “ “)

Imprima (formatear (“%.2f”, valor2))

Ejemplo 4: Mostrar texto con números formateados

Imprima (“su nombre es “+ nombre + “y su salario es “+ formatear (“%.1f”, salario))

6.2.3 Problemas resueltos

Los problemas planteados en este curso están compuestos de 6 secciones que se detallan a continuación:

El título: Es una frase breve y representativa del problema, da una idea general de lo que trata el desafío.

Descripción: Explica en detalle el contexto del problema y lo que se espera que el programa resuelva. Puede incluir una historia o narrativa breve para hacerlo más comprensible, aunque lo importante es definir claramente la **tarea principal** que debe realizar el algoritmo

Datos de entrada: Define el **formato exacto** en el que se recibirán los datos de entrada, así como las restricciones asociadas (rango de valores, número de elementos, etc.). Esta sección es crucial conocer la cantidad de valores, tipo y el **orden** que serán leídos

Datos de salida: Indica con precisión qué debe imprimir el programa después de procesar los datos de entrada. Se especifica el **formato exacto** en que debe mostrarse la respuesta, incluyendo si hay restricciones sobre los espacios, saltos de línea, numero de decimales o el uso de ciertos caracteres.

Ejemplo datos de entrada: Proporciona un ejemplo de cómo se ingresarán los datos en el programa, siguiendo el formato especificado en la sección **Datos de entrada**. Esto permite a los usuarios validar si están procesando correctamente la información

Ejemplo datos de salida: Muestra el resultado esperado cuando se ejecuta el programa con el ejemplo de entrada proporcionado. Esto sirve como referencia para verificar que la implementación del código es correcta.

Ejercicio 1

Viajando a estados unidos

Estados Unidos se ha consolidado como uno de los destinos turísticos más populares a nivel mundial, atrayendo a millones de visitantes cada año, incluidos un gran número de colombianos. Este país ofrece una amplia variedad de atracciones y experiencias que, en su mayoría, requieren pagos en la moneda local: el dólar estadounidense (USD).

Para los turistas colombianos, esto implica la necesidad de realizar un cambio de divisas antes o durante su viaje. Si una persona desea conocer cuántos dólares recibirá a partir de una cantidad determinada de pesos colombianos (COP), es necesario considerar el tipo de cambio vigente en el momento de la conversión

Calcular la cantidad de dólares que recibirá una persona por un monto de pesos colombianos que posee

Datos de Entrada

La entrada contendrá dos reales en cada línea. Estos dos reales denotan la cantidad de pesos a cambiar y valor del dólar en el momento

Datos de Salida

Para cada línea de entrada, imprima un único real en una línea que denote la cantidad de dólares que recibirá

Ejemplo Datos Entrada	**Ejemplo Datos Salida**
1500000 3000 850000 2950 2000000 3100	500.00 288.14 645.16

Solución

```
publico clase viaje
publico estatico vacio principal ()
      real pesos, dolares, unidad_cambiaria
        lea pesos
        lea unidad_cambiaria
        dolares = (pesos / unidad_cambiaria)
        imprima (formatear ("%.2f", dolares))
finprincipal
```

Ejercicio 2

Almacenando Datos

Hoy en día es muy usual que las personas utilicen cada vez más los recursos que les ofrece las nuevas tecnologías; se puede decir que lo que más le interesa a una gran parte de las personas es poder almacenar sus archivos de forma seguro y para eso hace uso en la gran mayoría de ocasiones de un disco duro. Pero por lo general las personas solo conocen la capacidad del disco duro en gigabytes. Una persona necesita conocer la capacidad del disco duro en megabyte, kilobytes y bytes; y así determinar si puede guardar sus archivos.
Considere que: 1 kilobyte = 1024 bytes, 1 megabyte = 1024 kilobyte, 1 gigabyte = 1024 megabytes

Datos de Entrada

Estará determinada por un valor real que denota la capacidad del disco duro en gigabytes

Datos de Salida

Se mostrará tres valores reales que será la equivalencia del disco duro en megabytes, kilobytes y bytes

Ejemplo Datos Entrada	**Ejemplo Datos Salida**
2 4 3	2048.0 2097152.0 2147483648.0 4096.0 4194304.0 4294967296.0 3072.0 3145728.0 3221225472.0

Solución

```
publico clase capacidad
   publico estatico vacio principal ()
      real gigas, megas, kilo, bytes
      lea gigas
      megas=1024*gigas
      kilo=1024*megas
      bytes=1024*kilo
      imprima (formatear ("%.1f", megas) +" "+formatear ("%.1f", kilo) + " "+formatear ("%.1f", bytes))
  finmetodo
finclase
```

6.2.4 Prueba de escritorio

Una prueba de escritorio es un proceso manual que se utiliza para verificar el funcionamiento de un algoritmo o programa, línea por línea, como si se estuviera ejecutando en una computadora. La prueba se realiza asignando valores a las variables del algoritmo y siguiendo paso a paso su flujo para comprobar si el resultado obtenido es el esperado. El objetivo principal es identificar posibles errores o fallas lógicas en el código antes de que se implemente o ejecute en un entorno real.

¿Cómo se realiza una prueba de escritorio?

1. **Selección del algoritmo o programa:** Se elige el código que se va a probar.
2. **Creación de una tabla:** Se crea una tabla con columnas que representen las variables del programa y las salidas esperadas.
3. **Ejecución manual:** Se simula la ejecución del programa paso a paso, siguiendo la lógica del código y registrando los valores de las variables en la tabla.
4. **Verificación de resultados:** Se comparan los valores obtenidos en la tabla con los resultados esperados. Si hay discrepancias, se identifican los errores en el código.

Ejemplo

Calcular el promedio de 3 números

```
Publico clase operaciones
    Publico estático vacio principal()
        Entero num1,num2,num3
        Lea num1
        Lea num2
        Lea num3
        Prom=(num1+num2+num3)/3
       Imprima(prom)
    Finmetodo
finclase
```

Prueba de escritorio

Paso	Variable	Valor
1	num1	5
2	num2	3
3	Num3	1
4	prom	3
5		Se escribe 3

En este ejemplo, se puede observar que el algoritmo funciona correctamente, ya que el promedio de 5, 3 y 1 es 3, que es el resultado que se muestra en la salida

6.3. Trabajando con 2 o más métodos

La programación orientada a objetos (POO) se basa en la idea de crear "objetos" que combinan datos (atributos) y acciones que operan en esos datos (métodos). Trabajar con dos o más métodos en POO es fundamental por varias razones:

- **Modularidad y Organización:**

División de responsabilidades: Cada método puede encargarse de una tarea específica dentro del objeto. Esto divide el código en unidades más pequeñas y manejables, lo que facilita su comprensión, mantenimiento y depuración.

Abstracción: Los métodos ocultan la complejidad interna de cómo se realizan las tareas. El usuario del objeto solo necesita saber qué hace cada método, no cómo lo hace.

Ejemplo:

Imagina un objeto "Coche". Podría tener métodos como arrancar(), acelerar(), frenar() y girar(). Cada método se encarga de una acción específica del coche, lo que hace que el código sea más organizado y fácil de entender.

- **Reutilización de Código:**

Métodos como bloques de construcción: Una vez que se ha definido un método, se puede llamar desde diferentes partes del programa o incluso desde otros objetos. Esto evita la repetición de código y facilita la creación de programas más complejos.

Ejemplo:

En el objeto "Coche", el método acelerar() podría ser utilizado tanto para aumentar la velocidad gradualmente como para realizar un adelantamiento rápido.

- **Interacción entre Objetos:**

Comunicación a través de métodos: Los objetos pueden interactuar entre sí llamando a los métodos de otros objetos. Esto permite crear sistemas complejos donde los objetos colaboran para lograr un objetivo común.

Ejemplo:

Un objeto "Semáforo" podría comunicarse con un objeto "Coche" a través de métodos como cambiarColor() y indicarEstado(). El coche, a su vez, podría reaccionar a esta información llamando a sus propios métodos frenar() o acelerar().

6.3.1 paso de parámetro por valor

Para trabajar con dos o más métodos debemos aplicar el concepto de parámetros, En programación orientada a objetos (POO), los parámetros de un método son variables que se utilizan para pasar información a un método desde otra parte del programa. Los parámetros permiten que los métodos operen con datos específicos o realicen acciones basadas en esa información.

- **parámetros de entrada**: Un método puede recibir o no parámetros, estos parámetros son los datos que el método no conoce y que necesita para hacer

su trabajo. Cuando un método recibe parámetros estos deben ser recibidos en variables del mismo tipo de dato.

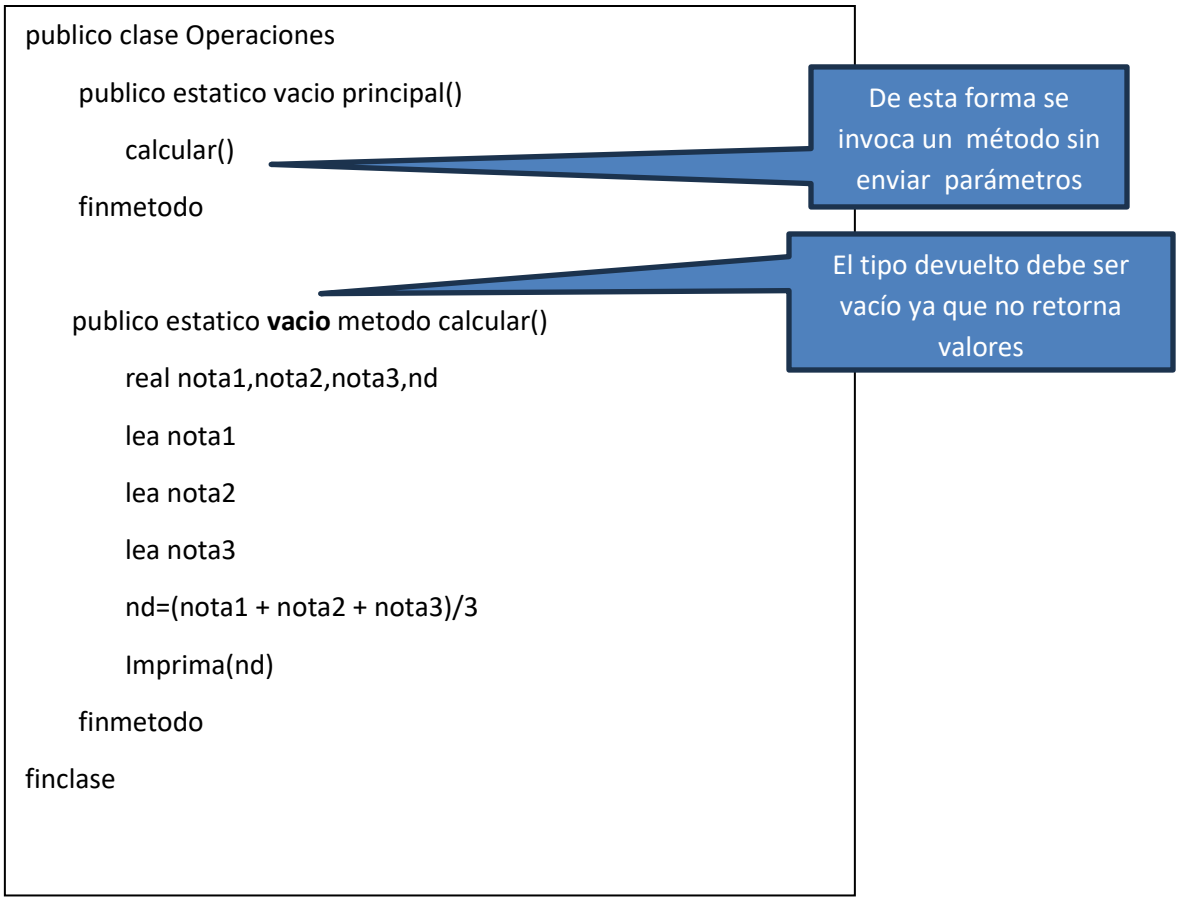
- **parámetros de salida**: Son parámetros cuyos valores se calcularán en el procedimiento y se deben devolver al programa principal o al método que lo invoco para su posterior proceso

Existen varias formas de trabajar con métodos

6.3.2 No Envía Parámetros, No Recibe Parámetros

son útiles para realizar acciones internas en un objeto sin necesidad de datos externos. Aunque tienen limitaciones, son valiosos para simplificar el código, encapsular lógica y promover la reutilización. Su uso dependerá del contexto y las necesidades específicas del programa.

Ejemplo



6.3.3 Envía parámetros, No recibe Parámetros

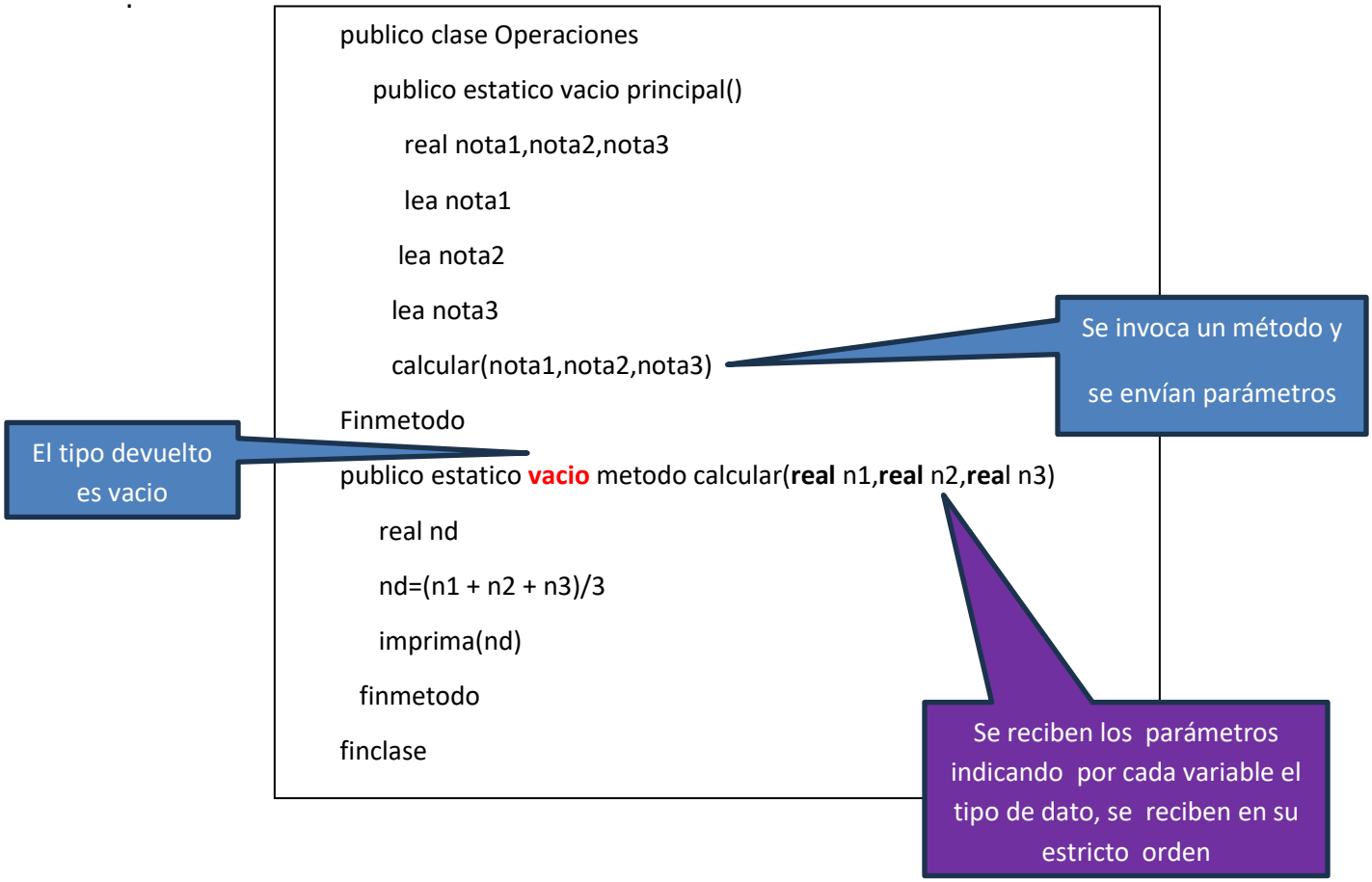
Se refiere a situaciones donde un método realiza una acción que requiere información externa (parámetros), pero no espera recibir ningún valor o resultado específico a cambio.

¿Qué significa esto?

Envía parámetros: El método necesita que se le proporcionen datos o valores específicos para poder realizar su tarea. Estos datos se envían como argumentos al llamar al método.

No recibe parámetros: El método no está diseñado para devolver ningún valor o resultado. Su propósito principal es realizar una acción o modificar el estado de un objeto, pero no necesita comunicar un resultado específico a quien lo llamó.

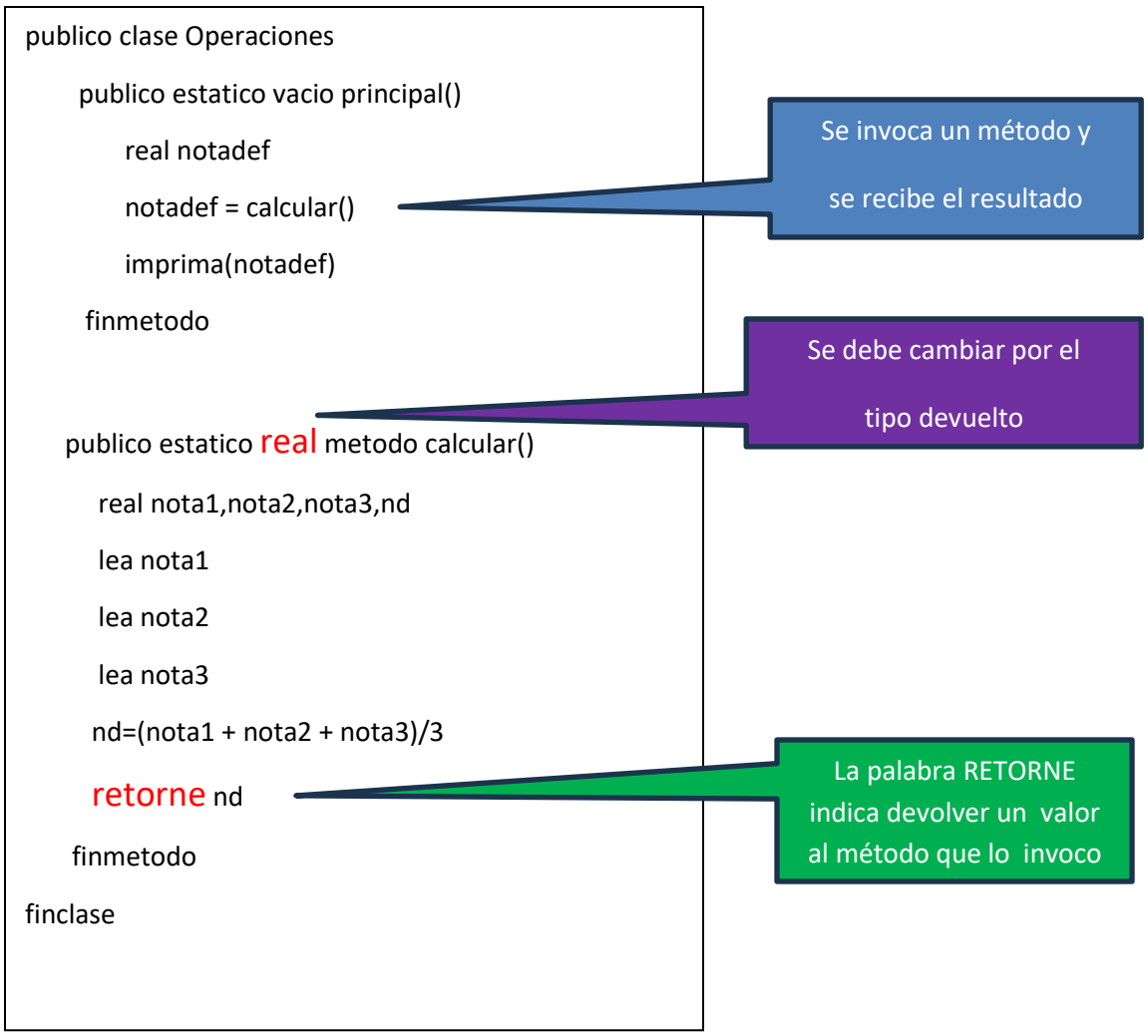
.



6.3.4 No Envía Parámetros, Si Recibe parámetro

Se refiere a un método que, si bien no recibe argumentos directamente durante su llamada, sí accede a información o datos que le son proporcionados de alguna otra forma, ya sea a través de variables de instancia (atributos) del objeto o por medio de algún otro mecanismo de "entrada" de datos.

Ejemplo



6.3.5 Envía Parámetros, Recibe parámetro

Es la forma más común y fundamental de trabajar con métodos y parámetros. Esto significa que un método acepta datos de entrada (parámetros) y también puede devolver un resultado o valor (parámetro de retorno) después de realizar su tarea.

¿Qué significa esto en detalle?

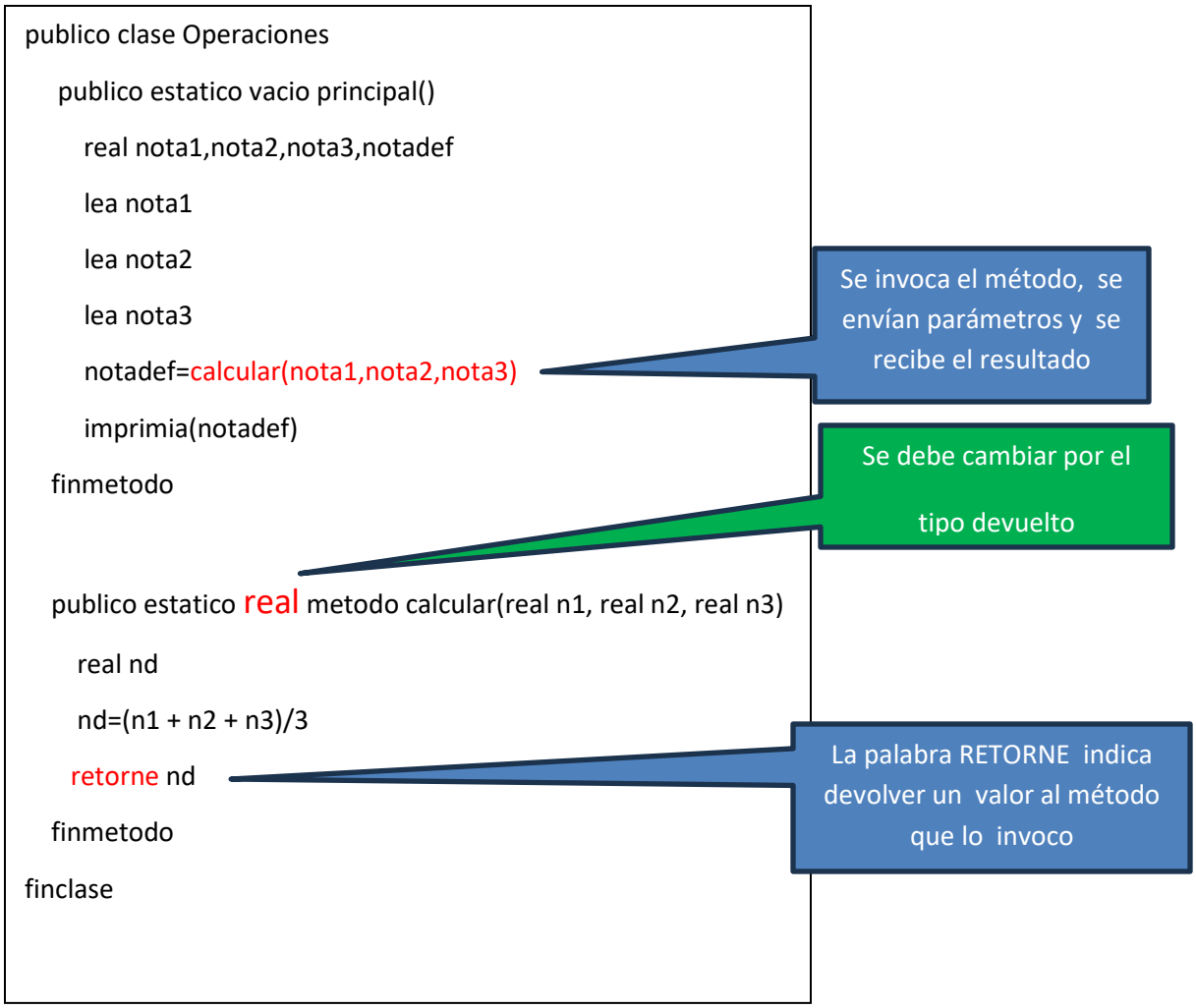
Envío de parámetros (Argumentos): Cuando se llama a un método, se le pueden proporcionar uno o más valores. Estos valores son los "argumentos" y se corresponden con los "parámetros" definidos en la declaración del método. Los

argumentos permiten que el método trabaje con datos específicos o realice acciones basadas en esa información.

Recepción de parámetros (Parámetros): Dentro del método, los parámetros actúan como variables locales que almacenan los valores de los argumentos enviados. El método utiliza estos parámetros para realizar sus operaciones.

Retorno de parámetros (Valor de retorno): Un método puede (o no) devolver un valor como resultado de su ejecución. Este valor se especifica mediante la palabra clave **retorne** y debe coincidir con el tipo de dato declarado como valor de retorno del método. El valor de retorno permite que el método comunique un resultado o información a quien lo llamó.

Ejemplo



6.4 Estructuras Condicionales

Las estructuras condicionales son un conjunto de instrucciones en un programa que permiten tomar decisiones basadas en una **condición lógica**. Estas estructuras evalúan una expresión booleana (**verdadero o falso**) y ejecutan diferentes bloques de código dependiendo del resultado.

En programación, las estructuras condicionales más comunes son:

Condicional simple: Ejecuta un bloque de código solo si la condición es verdadera.

Condicional compuesta: Ejecuta un bloque de código si la condición es verdadera y otro diferente si es falsa.

Multicondicional: Permite evaluar múltiples condiciones en secuencia.

Condicional ternario: Una forma simplificada del condicional en una sola línea.

Para que el programa pueda evaluar estas condiciones se requiere de los operadores relacionales y de los operadores lógicos

6.4.1 Operadores relacionales

Los operadores relacionales (también llamados operadores de comparación) son utilizados en programación para comparar valores y devolver un resultado booleano (verdadero o falso). Son fundamentales para la toma de decisiones en estructuras condicionales.

Operador	Descripción	Ejemplo	Resultado
==	Igual a	5 == 5	Verdadero
<>	Diferente de	5 <> 3	Verdadero
>	Mayor que	7 > 3	Verdadero
<	Menor que	4 < 2	Falso
>=	Mayor o igual que	6 >= 6	Verdadero
<=	Menor o igual que	3 <= 5	Verdadero

6.4.2 Operadores lógicos

Los operadores lógicos son símbolos que se utilizan para combinar o modificar expresiones booleanas (verdadero o falso). Se utilizan ampliamente en programación para controlar el flujo de ejecución de un programa y tomar decisiones basadas en múltiples condiciones.

Tipos de operadores lógicos

Los operadores lógicos más comunes son:

^ (y): Devuelve verdadero solo si ambas expresiones son verdaderas. En caso contrario, devuelve falso.

v (o): Devuelve verdadero si al menos una de las expresiones es verdadera. Solo devuelve falso si ambas expresiones son falsas.

Tablas de verdad

Las tablas de verdad muestran el resultado de una operación lógica para todas las combinaciones posibles de valores de entrada:

^ (y)

Expresión 1	Expresión 2	Resultado
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Falso
Falso	Verdadero	Falso
Falso	Falso	Falso

v (o)

Expresión 1	Expresión 2	Resultado
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Verdadero

Falso	Verdadero	Verdadero
Falso	Falso	Falso

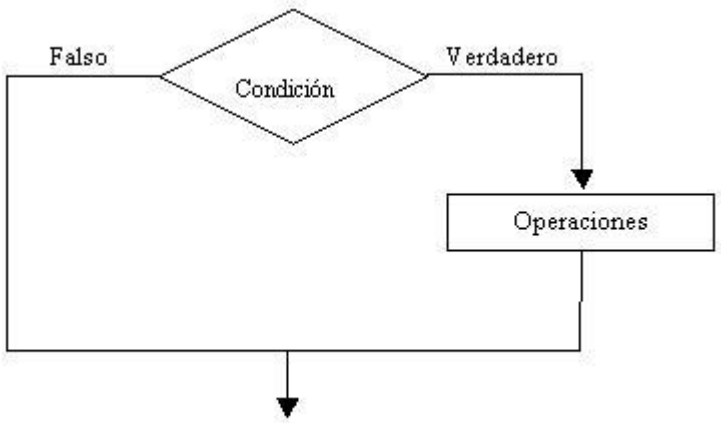
Ejemplo

correcto	**incorrecto**
si(5 <8) es verdadero	Si (a >b <c)
Edad= 15 Si(edad > 18) falso	Si(10 < a ^ > 20)
Numero= 20 Si(numero > 10 ^ numero <30) verdadero	Si(10 < a < 20)
Si (nombre == “ ana” ^ edad >= 18)	Si (a > b) ^ (a <c)
Si(ciudad ==“medellin” v ciudad==“bello”)	Si a<b faltan paréntesis
Si(5*3+2 <> 6 mod 3 +5) verdadero	

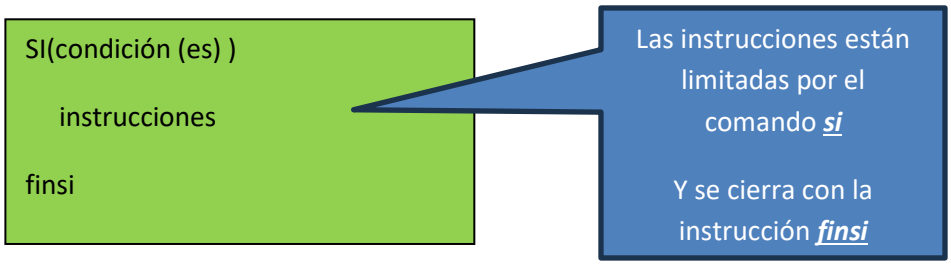
6.4.3 Estructura Condicional Simple

Una estructura condicional simple, es una estructura de control en programación que permite ejecutar un bloque de código solo si se cumple una determinada condición. La condición se evalúa como verdadera o falsa. Si la condición es verdadera, se ejecuta el bloque de código dentro de la estructura condicional. Si la

condición es falsa, se omite el bloque de código y se continúa con la siguiente instrucción del programa.



La sintaxis general de la estructura condicional simple en seudocódigo es la siguiente

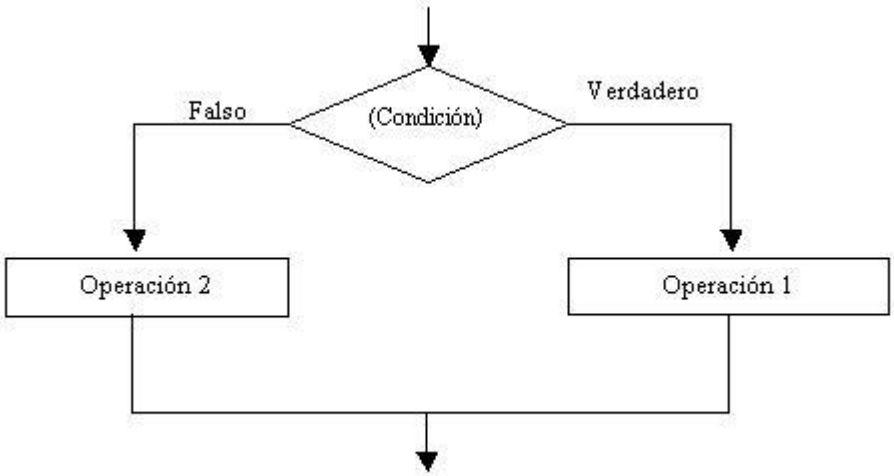


Ejemplo

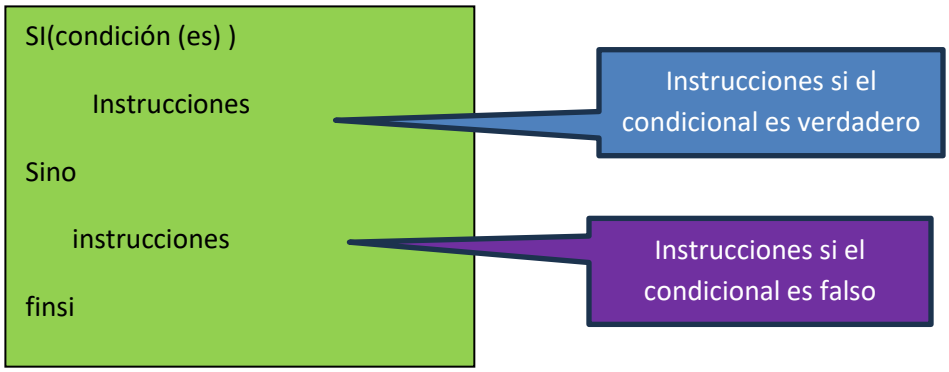
```
Lea dos números e imprima ambos números solos si son positivos
  publico clase Operaciones
        publico estatico vacio principal()
              real num1,num2
              lea num1
              lea num2
             mostrarNumeros(num1,num2)
    finmetodo
    publico estatico vacio metodo mostrarNumeros(real num1,real num2)
           si(num1> 0 ^ num2> 0)
                    imprima(num1+ " " +num2)
          finsi
     finmetodo
finclase
```

6.4.4 Estructuras Condicionales Compuestas

Una estructura condicional compuesta es aquella que evalúa una condición y ejecuta un bloque de código si la condición es verdadera, y otro bloque de código si la condición es falsa. Se utiliza cuando hay dos posibles caminos de ejecución en un programa.



La sintaxis general de la estructura condicional compuesta en seudocódigo es la siguiente



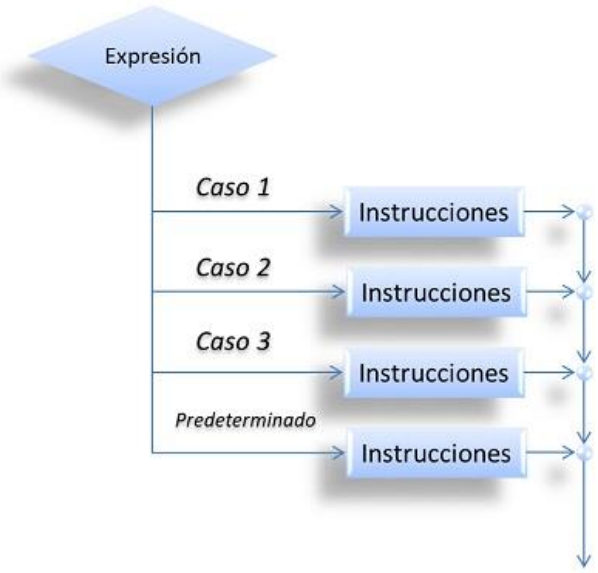
Ejemplo

capturar por pantalla 3 notas mostrar un mensaje si gano o perdió

```
Publico clase operaciones
      publico estatico vacio principal()
            real n1,n2,n3,notadef
           lea n1
           lea n2
           lea n3
          notadef=(n1+n2+n3)/3
         imprima(mostrarMensaje(notadef))
   finmetodo
   publico estatico texto metodo mostrarMensaje(real notadef)
          si(notadef >= 3)
               retorne “Gano”
          Sino
                Retorne “Perdio”
          finsi
finmetodo
finclase
```

6.4.5 Múltiples (según)

La sentencia según es una estructura de control que permite seleccionar un bloque de código para ejecutar entre múltiples opciones, basándose en el valor de una variable o expresión. Es útil cuando se tienen muchas condiciones que dependen de un solo valor, funcionando como una alternativa más ordenada y eficiente que múltiples condiciones anidades



La sintaxis general de la estructura multicondicional en seudocódigo es la siguiente

```
Segun(identificador)
    caso  valor1:
              ------
              salto
caso valor2:
              ----
              salto
caso  valor3:
              ----
              salto
En otro caso:
              -----
              Salto
FinSegun
```

Puede ser una variable o una expresión

Debe diferente a REAL

Explicación:

1. **segun(variable):** Se evalúa la expresión variable, que debe ser de tipo entero, carácter, o texto, No puede ser real
2. **caso valor1::** Se compara el valor de variable con valor1. Si coinciden, se ejecuta el bloque de código correspondiente.
3. **salto:** La sentencia **salto** es crucial. Indica el final del bloque de código para ese caso. Sin **salto**, la ejecución "caería" al siguiente caso, lo cual no suele ser deseado.
4. **En otro caso:** El caso **en otro caso** es opcional. Se ejecuta si variable no coincide con ningún valor de los casos anteriores.

Ejemplo

```
Publico clase operaciones
    publico estatico vacio principal()
        entero diaSemana
        lea diaSemana
        mostrarMensaje(diaSemana)
  finmetodo
  publico estatico vacio metodo mostrarMensaje(entero diaSemana)
     texto nombreDia
     segun(diaSemana)
    caso 1:
       nombreDia = "Lunes"
       salto
   case 2:
        nombreDia = "Martes"
        salto
  case 3:
        nombreDia = "Miércoles"
        salto
  case 4:
        nombreDia = "Jueves"
        salto
  case 5:
        nombreDia = "Viernes"
        salto
  case 6:
        nombreDia = "Sábado"
        salto
  case 7:
       nombreDia = "Domingo"
       salto
  en otro caso:
        nombreDia = "Día no válido"
  finsegun
imprima("El día de la semana es: " + nombreDia)

finmetodo
finclase
```

6.4.6. Operador Ternario

El **operador ternario** es una forma compacta de escribir una estructura condicional en una sola línea. Se representa con el símbolo **? :** y permite evaluar una condición y devolver uno de dos valores dependiendo de si la condición es verdadera o falsa.

Sintaxis general

```
resultado = (condición) ? valor_si_verdadero : valor_si_falso
```

Explicación:

1. **Condición** Es una expresión booleana que se evalúa como verdadera (true) o falsa (false).
2. **?** El signo de interrogación indica el inicio del operador ternario.
3. **valor_si_verdadero** Es el valor que se devuelve si la condición es verdadera.
4. **:** Los dos puntos separan los valores para verdadero y falso.
5. **valor_si_falso** Es el valor que se devuelve si la condición es falsa.

Ejemplo

```
Publico clase operaciones
      publico estatico vacio principal()
            entero edad
           texto mensaje
           lea edad
           mensaje= calcular(edad)
           imprima(mensaje)
     finmetodo
    publico estatico texto metodo calcular(entero edad)
       texto mensaje = (edad >= 18) ? "Eres mayor de edad" : "Eres menor de edad"
      retorne mensaje
    finmetodo
finclase
```

Ejemplo 2 Operador ternario anidado

```
real nota = 7
texto calificacion = nota < 5 ? "Insuficiente" :
 nota < 6 ? "Suficiente" :
nota < 8 ? "Bien" :
nota <= 9 ? "Notable" :
 "Sobresaliente “
imprima("He obtenido un“+ calificacion)
```

6.4.7 Problemas resueltos

Ejemplo 1 Consumo de gasolina

El consumo de gasolina de un automóvil depende de la distancia recorrida así: en los primeros 60 km el auto gasta un galón cada 30 km, mientras que después de esa distancia, el consumo es de 1/20 de galón por km. Desarrollar un algoritmo que permita calcular el consumo total de gasolina de un automóvil para un trayecto cuya distancia está dada por el usuario, asumiendo que el automóvil no se detiene.

Datos de Entrada

Se debe ingresar un valor entero que corresponden a la distancia recorrida

Datos de Salida

Para la salida se obtendrá la equivalencia de los galones consumidos por la distancia recorrida

Ejemplo Datos Entrada	Ejemplo Datos Salida
320	15.00
20	0.67
150	6.50
50	1.67

Solución

```
publico clase vehiculo
        publico estatico vacio principal()
              real consumo, distancia,gasolina
              lea distancia
             gasolina=calcular_consumo(distancia)
            imprima(formatear("%.2f",gasolina))
      finprincipal
      publico estatico real metodo calcular_consumo(real distancia)
              real consumo
             si(distancia < 60)
                 consumo = distancia*2/60
          sino
                  consumo = 2+(distancia - 60) * 1/20
          finsi
           retorne(consumo)
     finmetodo
 finclase
```

Ejemplo 2 De menor a mayor

A un estudiante de tercer grado le piden llevar tres números enteros dados por la profesora, organizados de menor a mayor. El niño aún no sabe distinguir muy bien el orden de números mayores y menores, así que le pide ayuda a su hermano para que le haga un algoritmo que le organice dichos números como debe organizarlos

Datos de Entrada

La entrada contendrá tres números enteros en cada línea, aquellos números que serán ordenados de menor a mayor.

Datos de Salida

Se mostrarán los 3 números ordenados de forma ascendente es decir de menor a mayor

Ejemplo Datos Entrada	**Ejemplo Datos Salida**
25 2 3 6 5 7 8 96 12	2 3 25 5 6 7 8 12 96

Solución

```
publico clase meno_a_mayor
   publico estatico vacio principal()
        entero num1,num2,num3
        lea num1
        lea num2
       lea num3
      calcular(num1,num2,num3)
finprincipal
   publico estático vacio método calcular(entero num1, entero num2, entero num3)

     si(num1<=num2 ^ num2 <= num3)
        imprima (num1+ " "+num2 + " "+num3)
     sino
       si(num1<=num3 ^ num3 <= num2)
            imprima (num1+ " "+num3+ " "+num2)
        sino
          si(num2<=num1 ^ num1 <= num3)
            imprima (num2+ " "+num1+ " "+num3)
          sino
             si(num2<=num3 ^ num3 <= num1)
              imprima (num2+ " "+num3+ " "+num1)
            sino
              si(num3<=num2 ^ num2 <= num1)
                imprima (num3+ " "+num2+ " "+num1)
             sino
                  imprima (num3+ " "+num1+ " "+num2)
             finsi
          finsi
       finsi
    finsi
finsi
  finmetodo
finclase
```

6.4.8 Problemas propuestos

Realizar los ejercicios propuestos en LOGICODE

6.5 Estructuras Repetitivas o Cíclicas

Las estructuras repetitivas, también conocidas como bucles, son fundamentales en programación, ya que permiten ejecutar un conjunto de instrucciones varias veces hasta que se cumpla una condición determinada.

6.5.1 Contadores

Definición: Un contador es una variable entera utilizada para llevar un registro numérico de cuántas veces ocurre un evento específico o cuántos elementos han sido procesados en un programa

Funcionamiento:

- **Inicialización:** Se inicializa a un valor inicial, generalmente 0.
- **Incremento:** Se incrementa cada vez que ocurre el evento que se está contando, este incremento debe ser una **constante**

Ejemplo:

```
cantidadpersonas= cantidadpersonas + 1
```

6.5.2 Acumuladores

Definición: Un acumulador es una variable numérica que se utiliza para mantener un valor acumulado a medida que se procesa un conjunto de datos

Funcionamiento:

- **Inicialización:** Se inicializa a un valor inicial, generalmente 0 para sumas, y 1 para productos.
- **Acumulación:** Se actualiza sumando o multiplicando el valor actual con el nuevo valor procesado.

Ejemplo:

```
Lea edad

Sumaedad=sumaedad + edad
```

6.5.3 Ciclo para

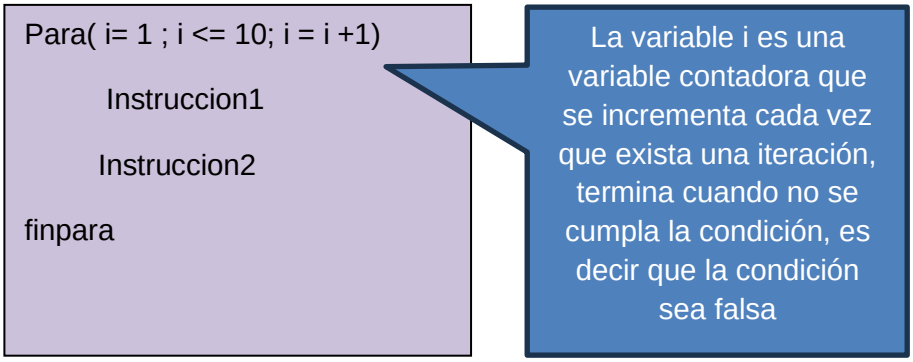
Definición: El ciclo para es una estructura repetitiva que se utiliza cuando se conoce de antemano el número de iteraciones que se deben realizar

sintaxis:

- **Inicialización:** Se define una variable que tomará valores desde un punto inicial hasta un punto final.
- **Condición:** El ciclo termina cuando la variable alcanza el valor final.
- **Incremento:** La variable se incrementa en cada iteración.

```
Para( variable_control= valor inicial ; condición ;  incremento o decremento)

        Instruccion1

        Instruccion2

finpara
```

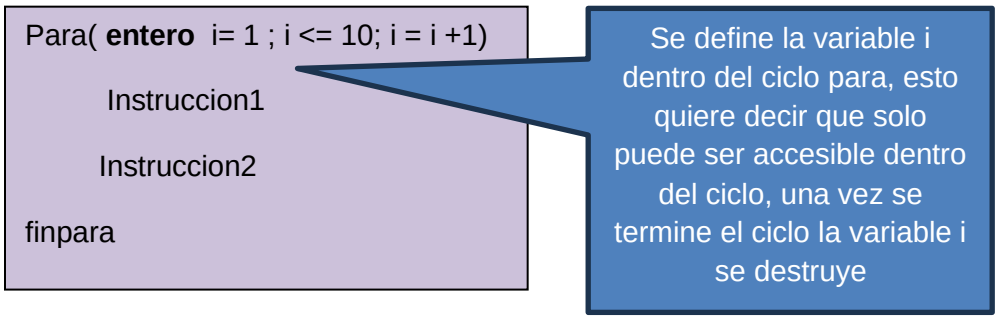
Ejemplo



6.5.3.1 variables tipo bloque

Las variables tipo bloque se refieren a las variables que se definen dentro de un bloque de código, como un ciclo para. Estas variables tienen un alcance limitado al bloque donde se declaran.

Ejemplo



Ejercicio con el ciclo para

Ejemplo Factorial

Realice un algoritmo que muestre la factorial de un número

Datos de Entrada

Contendrá un entero con un numero al cual se le calculará el factorial.

Datos de Salida

Se mostrará el factorial del número ingresado.

Ejemplo Datos Entrada	**Ejemplo Datos Salida**
3 5 7	6 120 5040

Solución

```
publico clase Factorial
    publico estatico vacio principal()
       entero x
       lea x
       imprima(factorial(x))
    finprincipal
     publico estatico entero metodo factorial(entero x)
         entero fact = 1, j
         para(j=1; j<= x; j=j+1)
              fact = fact *j
         finpara
         retorne(fact)
    finmetodo
finclase
```

6.5.4 Ciclo mientras

Un ciclo **mientras** repite un bloque de código mientras una condición sea verdadera. La condición se evalúa al principio de cada iteración.

Funcionamiento:

- **Inicialización de la variable(s) de control**: la variable(s) de control debe tener un valor inicial
- **Condición:** Se evalúa antes de cada iteración.
- **Bloque de código:** Se ejecuta si la condición es verdadera.
- **Modificación de la variable de control:** Es necesario actualizar la condición dentro del bloque para evitar un bucle infinito.

```
variable_control= valor inicial
mientras(condición(es))
        instruccion1
        instruccion2
        <Modificar la variable de control>
finmientras
```

Ejemplo:

```
I= 1
mientras( i <= 10)
        Instruccion1
        Instruccion2
        I=i+1
finmientras
```

La variable i es una variable controladora que se incrementa cada vez que exista una iteración, termina cuando no se cumpla la condición, es decir que la condición sea falsa

Ejercicios con el ciclo mientras

Realice un algoritmo que muestre la tabla de multiplicar de un numero entrado por el usuario

Datos de Entrada

Contendrá un entero que dirá el numero al cual se le hará la tabla de multiplicar.

Datos de Salida

mostrará los 5 primeros resultados de la tabla de multiplicar del número ingresado

Ejemplo Datos Entrada	Ejemplo Datos Salida
2 4 5	2*1=2 2*2=4 2*3=6 2*4=8 2*5=10 4*1=4 4*2=8 4*3=12 4*4=16 4*5=20 5*1=5 5*2=10 5*3=15 5*4=20 5*5=25

Solución

```
publico clase Tabla_Multiplicar
   publico estatico vacio principal()
      entero N, R, i
      lea N
      tabla(N)

   finmetodo

   publico estatico vacio metodo tabla(entero N)
   entero R,i
   i= 1
   mientras(i < 5)
         R = (i * N)
         imprimaln(N+"*"+i+"="+R)
         i= i+1
   finmientras
         R = (i * N)
         imprima(N+"*"+i+"="+R)
   finmetodo

finclase
```

El comando imprimaln permite imprimir y realizar un salto de línea

6.5.5 Ciclo haga mientrasque

El ciclo haga mientrasque es similar al ciclo mientras, pero garantiza que el bloque de código se ejecute al menos una vez, ya que la condición se evalúa al final del bucle.

Funcionamiento:

- **Bloque de código:** Se ejecuta al menos una vez.
- **Condición:** Se evalúa después de cada iteración.
- **Incremento:** Es necesario actualizar la condición dentro del bloque.

```
haga
    Instruccion1
   Instruccion2
<Modificar la variable de control>
Mientrasque(condición(es)
```

Ejemplo

```
i=0
haga
      Instruccion1
      Instruccion2
      i=i+1
mientrasque (i < 10)
```

Ejercicio con el ciclo haga-mientrasque

Elabore un algoritmo donde se ingresen por pantalla 2 números enteros diferentes. El usuario tiene la posibilidad de elegir una de 3 opciones posibles, se termina cuando se digite 4.

1. sumar los dos números
2. promedio de los números
3. Raíz cuadrada del segundo numero
4. Salir

```
publico clase cls_ejercicio
publico estatico vacio principal()
       menu()
finmetodo
publico estatico vacio metodo menu()
           entero num1, num2,opcion
            lea num1
            lea num2
            haga
                 imprima("1.suma, 2. Promedio,3. Raíz, 4.Salir")
                 lea opcion
                segun(opcion)
                  caso 1:
                      imprima(suma(num1,num2))
                      salto
                  caso 2:
                     imprima(promedio(num1,num2))
                     salto
                  caso 3:
                     imprima(raíz_num(num2))
                     salto
                  En otro caso:
                      imprima("error")
                     salto
               finsegun
           mientrasque(op  <> 4)
    finmetodo

publico estatico entero metodo suma(entero num1,entero num2)
        entero S
         s= num1 + num2
        retorne S
Finmetodo
publico estatico real metodo promedio(entero num1,entero num2)
        real prom
     prom= (num1 + num2)/2
     retorne prom
finmetodo
publico estatico real metodo raíz_num(entero num)
        real r
     r= raíz(num)
     retorne r
finmetodo


finclase
```

6.5.6 Centinela

Cuando no se conoce a priori el número de iteraciones que se van a realizar, el ciclo puede ser controlado por centinelas.

En un ciclo <Mientras> controlado por tarea, la condición de Mientras específica que el cuerpo del ciclo debe continuar ejecutándose mientras la tarea no haya sido completada.

En un ciclo controlado por centinela el usuario puede suspender la introducción de datos cuando lo desee, introduciendo una señal adecuada llamada centinela. Un ciclo controlado por centinela es cuando el usuario digita una letra para salir como por ejemplo S o N para indicar si desea continuar o no. El ciclo debe repetirse hasta que la respuesta del usuario sea "n" o "N".

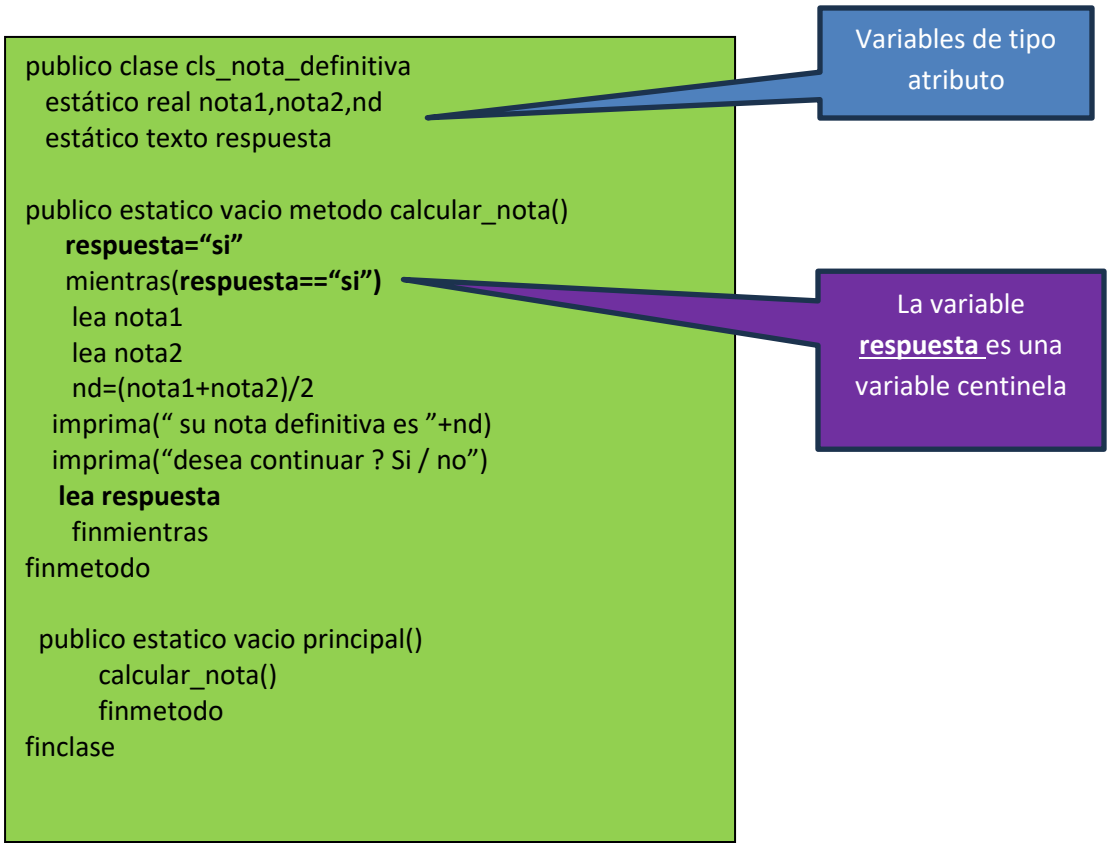
Cuando una decisión toma los valores de -1 o algún posible valor que no esté dentro del rango válido en un momento determinado, se le denomina centinela y su función primordial es detener el proceso de entrada de datos en una corrida de programa.

Por ejemplo, si se tienen las calificaciones de un test (comprendida entre 0 y 5); un valor centinela en esta lista puede ser -1, ya que nunca será una calificación válida y cuando aparezca este valor se terminará de ejecutar el ciclo.

Si la lista de datos son números positivos, un valor centinela puede ser un número negativo. Los centinelas solamente pueden usarse con las estructuras Mientras y Repetir, no con estructuras <Para>.

Ejemplo

Suponga que debemos obtener la nota definitiva de cada uno de los estudiantes de una universidad, pero no sabemos exactamente cuántos son, una solución seria



Ejercicio propuesto

Realizar un algoritmo que determine el valor que debe pagar un cliente en un supermercado de acuerdo con los productos que compra

Tenga en cuenta que no se sabe cuántos productos tiene.

6.5.7 Swiches o banderas

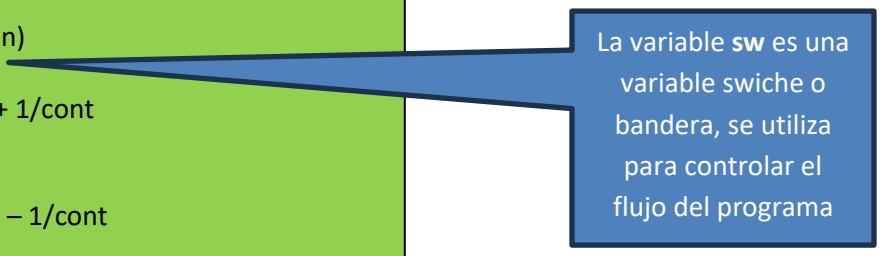
Conocidas también como interruptores, switches, flags, banderas o conmutadores, son variables que pueden tomar solamente dos valores durante la ejecución del programa, los cuales pueden ser 0 o 1, o bien los valores booleanos True o False. Se les suele llamar interruptores porque cuando toman los valores 0 o 1 están simulando un interruptor abierto/cerrado o encendido/apagado, se utilizan para indicar si un evento ha ocurrido o no. Se pueden usar para controlar el flujo de un programa.

Ejemplo

Leer un número entero N y calcular el resultado de la siguiente serie: 1 - 1/2+ 1/3 - 1/4 +.... 1/N.

En el ejercicio anterior se puede observar que se comienza sumando un valor y luego se resta el siguiente número, luego se suma y luego se resta. Esto se realiza N veces

```
publico clase cls_serie
   estático real serie=0, n, cont= 1, sw= 0
  publico estatico vacio principal()
         calcular_serie()
   finprincipal

   publico estatico vacio metodo calcular_serie()
      lea n
      mientras(cont <= n)
         si(sw==0)
              serie=serie + 1/cont
              sw=1
         sino
              serie= serie – 1/cont
              sw= 0
        finsi
        cont = cont+1
    finmientras
Imprima(serie)
finmetodo
finclase
```



6.5.8 Rompimiento de ciclos

En ocasiones, necesitamos salir de un bucle antes de que se complete su ejecución normal. Para ello, podemos utilizar las siguientes instrucciones:

- **salto:** Sale del bucle inmediatamente. En ingles **beak**
- **continue:** Salta a la siguiente iteración del bucle, omitiendo el resto del código en la iteración actual.

Ejemplo con la sentencia salto

Salto (break)	**continue**
para (entero i = 1; i <= 10; i=i+1) lea num si (num mod 2 == 0) **salto** sino imprima(num) finsi finpara	para (entero i = 1; i <= 10; i=i+1) lea num si (num mod 2 == 0) **continue** sino imprima(num) finsi finpara

Con la instrucción **salto**, se comienza a ingresar números (10 en total), si el numero ingresado es impar se imprime, pero en el primer momento que se ingrese un numero par se interrumpe el ciclo.

En cambio, con la instrucción **continue**, se comienza a ingresar números (10 en total), si el numero ingresado es impar se imprime, pero si el numero ingresado es par se salta a la siguiente interacción y se lee otro número.

En el ejercicio anterior

Si se ingresaran 3, 5, 21, 9, 12, 17, 20, 15, 22,13

Con la sentencia **salto** se imprimen 3, 5, 21, 9

Con la sentencia **continue** se imprimen 3, 5, 21, 9, 17, 15, 13

6.5.9 Trabando con 2 o más Clases

En las estructuras condicionales trabajamos con 2 o más métodos, en esta ocasión trabajaremos con 2 o más clases. El objetivo principal de trabajar con múltiples clases en POO es crear un código más organizado, modular, reutilizable y fácil de mantener. Esto se logra mediante la aplicación de los principios de la POO, como la herencia, el encapsulamiento y el polimorfismo

6.5.9.1 instancia

Para trabajar con múltiples clases debemos crear un objeto, este proceso es llamado "instanciación". Este proceso implica utilizar una clase como plantilla para construir un objeto específico. Aquí te explico los pasos y conceptos clave:

1. Definición de la Clase:

- Primero, necesitas tener una clase definida. La clase actúa como un plano o plantilla que describe las características (atributos) y los comportamientos (métodos) que tendrán los objetos creados a partir de ella.

2. Instanciación:

- La instanciación es el acto de crear un objeto a partir de una clase. Esto se logra utilizando una sintaxis específica que varía según el lenguaje de programación.
- Generalmente, se utiliza la palabra clave "nuevo" o “new” en ingles, seguida del nombre de la clase y paréntesis. Esto indica que se está creando una nueva instancia de esa clase.

3. Constructor:

- Cuando se instancia un objeto, se llama a un método especial llamado "constructor". El constructor es un método que se ejecuta automáticamente al crear el objeto.
- El constructor se utiliza para inicializar los atributos del objeto con valores iniciales. Puede recibir parámetros para personalizar la inicialización.

4. Almacenamiento del Objeto:

- El objeto creado se almacena en una variable. Esta variable actúa como una referencia al objeto, permitiendo acceder a sus atributos y métodos.
- Los métodos de la clase instanciada **no pueden ser métodos estáticos**

Ejemplo General:

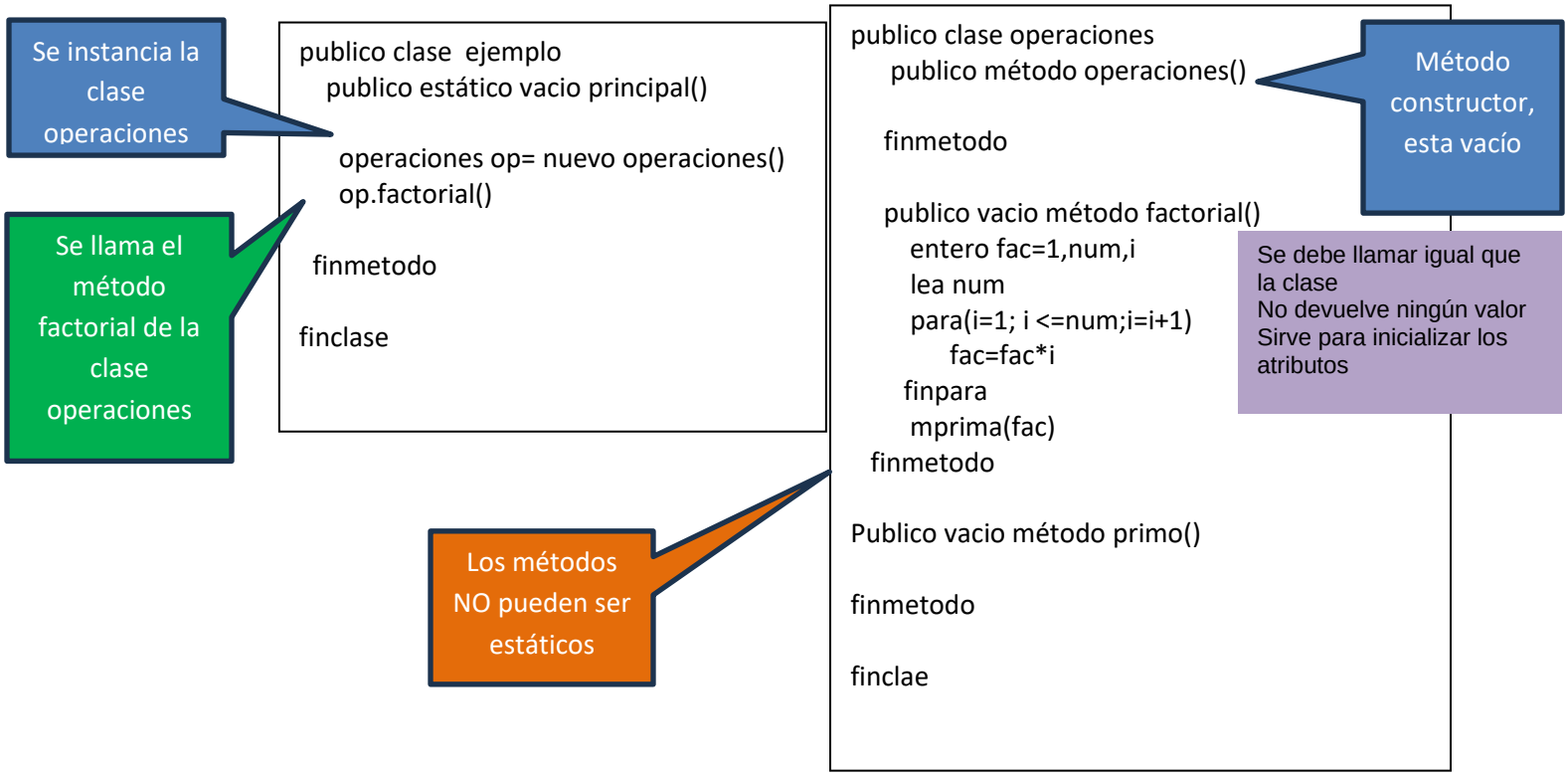
MiClase miObjeto = nuevo MiClase()

- "MiClase" es el nombre de la clase.
- "miObjeto" es el nombre de la variable que almacenará el objeto.
- "nuevo MiClase()" es la expresión que crea el objeto llamando al constructor de la clase.

Veamos las diferentes maneras de trabajar con múltiples clases enviando y retornando valores

En esta primera parte trabajaremos instanciando clases y con los métodos, el método constructor estará vacío, en la segunda parte se enviarán parámetros al método constructor

No Envía Parámetros, No Recibe Parámetros



Envía parámetros, No recibe Parámetros

```
publico clase  ejemplo
   publico estático vacio principal()
         entero num
         lea num

      operaciones op= nuevo operaciones()
      op.factorial(num)

   finmetodo

   finclase
```

Se instancia la clase operaciones

Se llama el método factorial y se envían parámetros

```
publico clase operaciones
     publico método operaciones()

    finmetodo

 publico vacio método factorial(entero num)
       entero fac=1,i
       para(i=1; i <=num;i=i+1)
            fac=fac*i
      finpara
       imprima(fac)
  finmetodo


Publico vacio método primo()

finmetodo

finclae
```

Método constructor, esta vacío

Se reciben los parámetros

No Envía parámetros, recibe Parámetros

Se instancia la clase operaciones

Se llama el método factorial

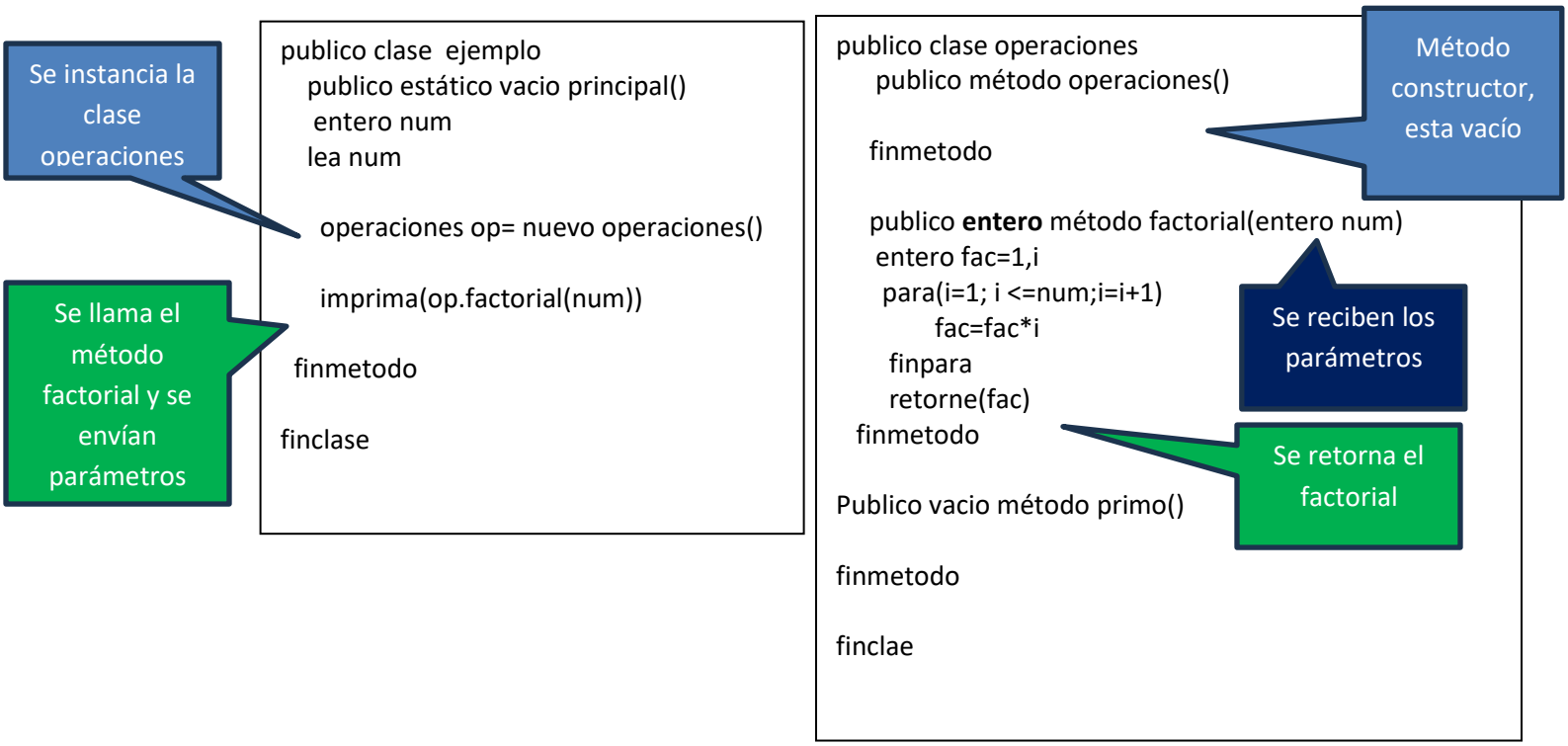
```
publico clase  ejemplo
   publico estático vacio principal()

     operaciones op= nuevo operaciones()
     imprima(op.factorial())

   finmetodo

finclase
```

```
publico clase operaciones
    publico método operaciones()

   finmetodo

   publico entero método factorial()
    entero num, i, fac=1
     lea num
     para(i=1; i <=num;i=i+1)
          fac=fac*i
     finpara
     retorne(fac)
  finmetodo

Publico vacio método primo()

finmetodo

finclae
```

Método constructor, esta vacío

Se retorna el factorial

Envía parámetros, recibe Parámetros



6.5.9.2 Método constructor

En la programación orientada a objetos (POO), el **método constructor** tiene como principal función **inicializar los objetos de una clase** al momento de su creación.

El método constructor se diferencia de los demás métodos

- El nombre del método debe ser igual al nombre de la clase
- No se coloca ningún tipo de valor devuelto, es decir no se coloca real, entero, texto, tampoco la palabra vacío.

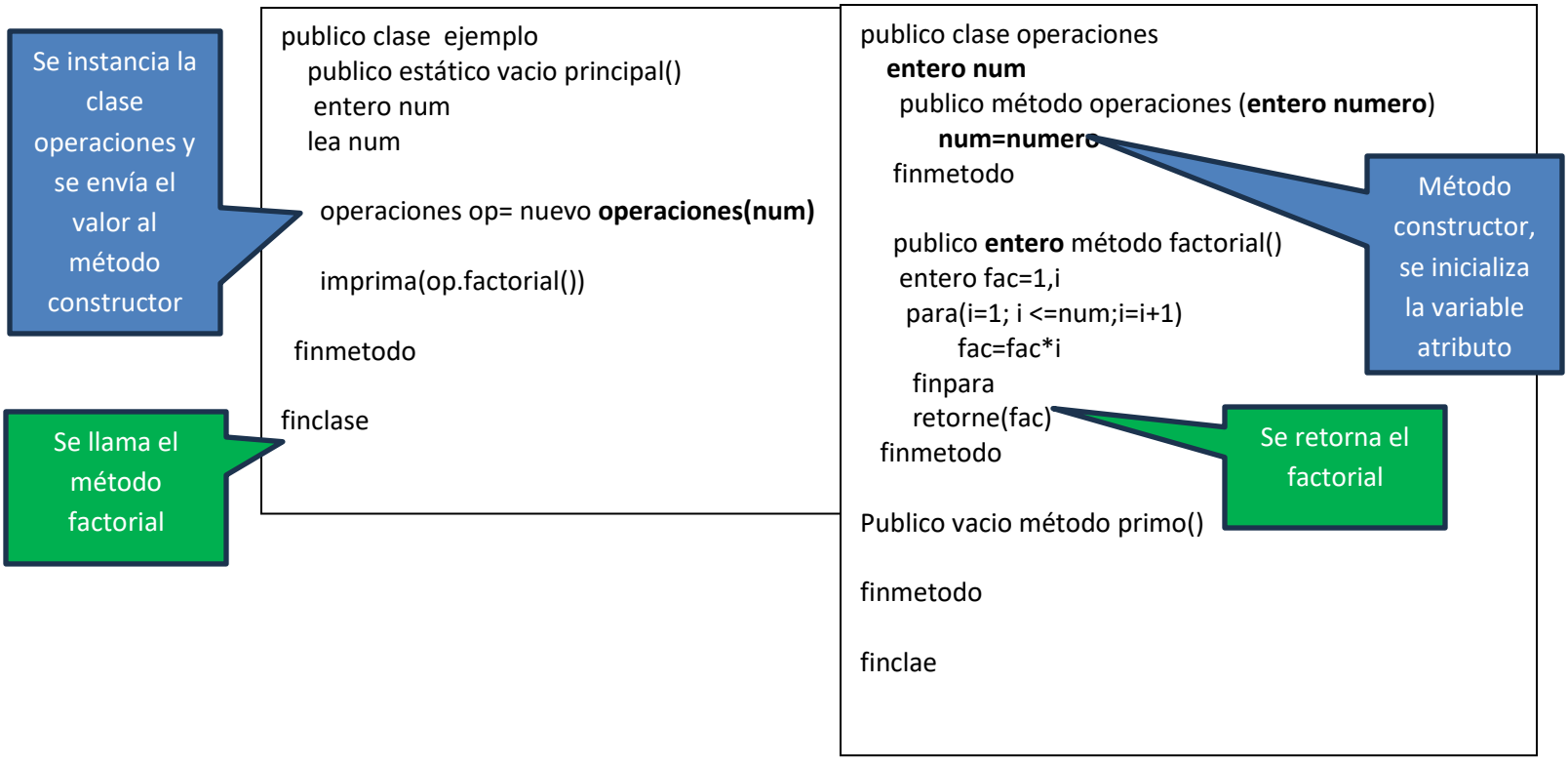
Su finalidad es la siguiente:

- **Inicialización de atributos:** El constructor se encarga de asignar valores iniciales a los atributos (variables miembro) del objeto. Esto asegura que cada objeto comience con un estado bien definido.
- **Realización de tareas de configuración:** Además de inicializar atributos, el constructor puede ejecutar otras operaciones necesarias para la correcta

creación del objeto. Esto podría incluir la asignación de recursos, la apertura de conexiones o la validación de datos.

- **Garantizar un estado válido:** Un constructor bien diseñado asegura que el objeto se cree en un estado consistente y válido, listo para ser utilizado.
- **Automatización:** El constructor se llama automáticamente cuando se crea una nueva instancia de la clase (utilizando la palabra clave **nuevo o new** en muchos lenguajes). Esto evita tener que llamar a un método de inicialización explícitamente después de crear el objeto, simplificando el código y reduciendo la posibilidad de errores.
- **Personalización de la creación de objetos:** Los constructores pueden recibir parámetros, lo que permite personalizar la forma en que se inicializa un objeto al momento de su creación. Esto hace que la clase sea más flexible y reutilizable.

El ejercicio anterior lo podemos realizar de la siguiente manera



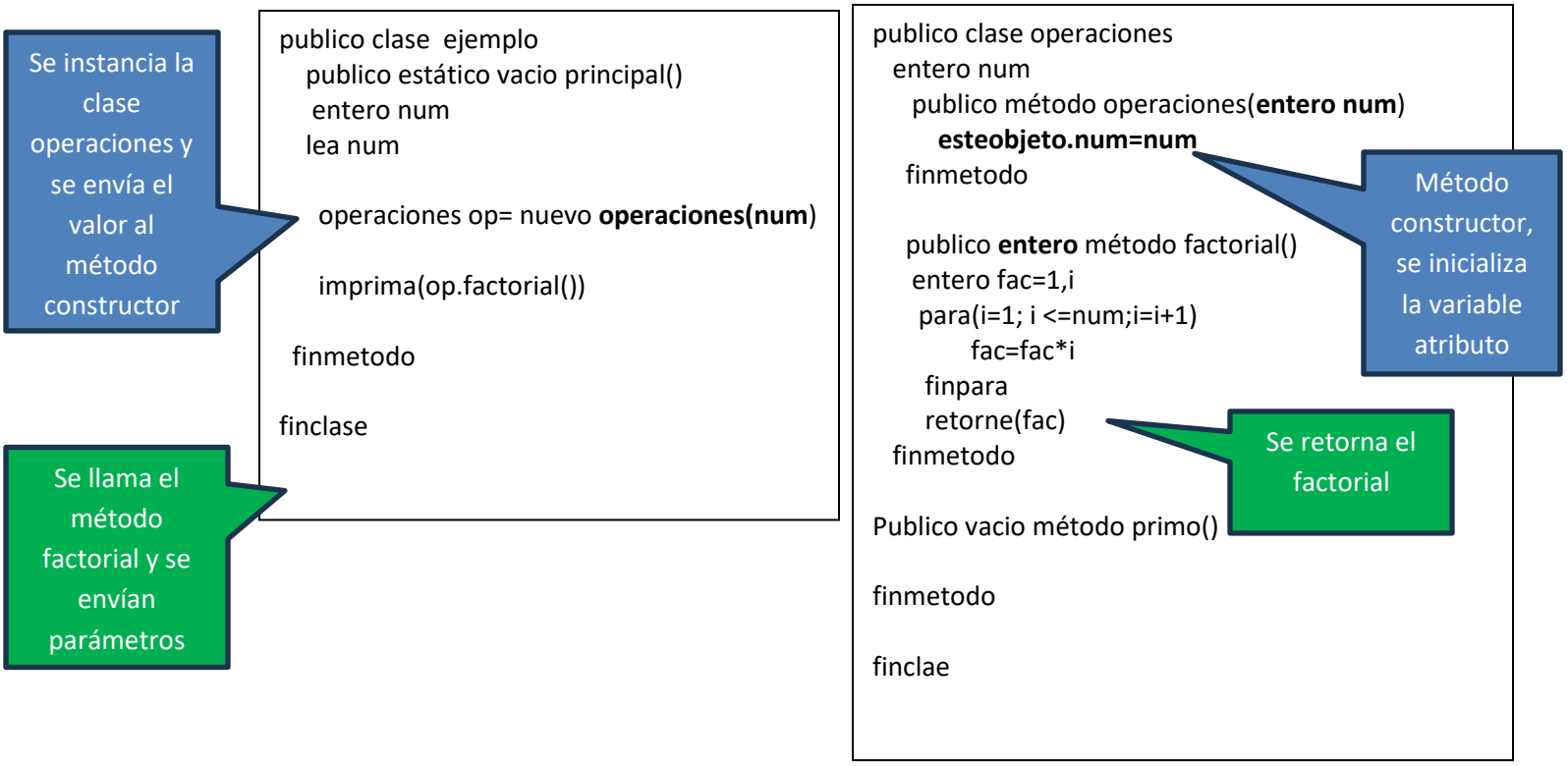
En el ejercicio anterior el método constructor llamado operaciones recibe un valor en una variable **(numero**), cuyo nombre es diferente a la variable atributo**(núm.),** por esta razón se puede realizar la asignación de esta manera:

```
num=numero
```

Sin embargo, si la variable del método constructor es igual a la variable atributo, se debe indicar cual es la variable atributo, para ello utilizaremos la instrucción **esteobjeto,** de la siguiente manera

Esteobjeto.num=num

El ejercicio quedaría de la siguiente manera



6.5.10 Problemas propuestos

Realizar los ejercicios propuestos en LOGICODE

7

UNIDAD 7 ARREGLOS

CONTENIDO

7.1 Introducción

7.2 Definición de arreglos

7.3 Características de los arreglos

7.4 Tipos de Arreglos

7.4.1 Unidimensionales (Vectores)

7.4.1.1 creación

7.4.1.2 Asignación

7.4.1.3 Paso de parámetro por referencia

7.4.1.4 Métodos de ordenamiento

7.4.1.4.1 Burbuja

7.4.1.4.2 Selección

7.4.1.4.3 Inserción

7.4.1.4.4 Shell

7.4.1.5 Búsqueda

7.4.1.5.1 Secuencial

7.4.1.5.2 Binaria.

7.4.1.6 Manipulación de Cadenas

7.4.2 N dimensiones (matrices)

7.4.2.1 Creación

7.4.2.2 Recorridos

7.5 Arreglos Objetuales

7.6 Ejercicios resueltos

7.7 Ejercicios propuestos

7.8 Cuestionario

 Resultado de Aprendizaje:

El estudiante:

7.1 Introducción

7.2 Definición de arreglos

7.3 Características de los arreglos

7.4 Tipos de Arreglos

7.4.1 Unidimensionales (Vectores)

7.4.1.1 creación

7.4.1.2 Asignación

7.4.1.3 Paso de parámetro por referencia

7.4.1.4 Métodos de ordenamiento

7.4.1.4.1 Burbuja

7.4.1.4.2 Selección

7.4.1.4.3 Inserción

7.4.1.4.4 Shell

7.4.1.5 Búsqueda

7.4.1.5.1 Secuencial

7.4.1.5.2 Binaria.

7.4.1.6 Manipulación de Cadenas

7.4.2 N dimensiones (matrices)

7.4.2.1 Creación

7.4.2.2 Recorridos

7.5 Arreglos Objetuales

7.6 Ejercicios resueltos

7.7 Ejercicios propuestos

7.8 Cuestionario

UNIDAD 8 CARACTERISTICAS DEL ENFOQUE ORIENTADO A OBJETO

CONTENIDO

8.1 Introducción

8.2 Abstracción

8.3 Encapsulación

8.4 Modularidad

8.5 Herencia

8.6 Jerarquía de clase

8.7 Polimorfismo

8.8 Problemas propuestos

8.9 Problemas Resueltos

8.10 Cuestionario

 Resultado de Aprendizaje:

El estudiante:

8.1 Introducción

8.2 Abstracción

8.3 Encapsulación

8.4 Modularidad

8.5 Herencia

8.6 Jerarquía de clase

8.7 Polimorfismo

8.8 Problemas propuestos

8.9 Problemas Resueltos

8.10 Cuestionario