PROGRAMACIÓN

IV

CARRERA: ANÁLISIS DE SISTEMAS

SEMESTRE: QUINTO

PROGRAMACIÓN IV

JAVA

Java es un lenguaje de programación y una plataforma informática, se utiliza para escribir programas, mientras que la plataforma Java proporciona un entorno en el que los programas Java pueden ser ejecutados. La plataforma Java incluye la Máquina Virtual de Java (JVM), que permite la portabilidad de los programas a través de distintos sistemas operativos, así como un conjunto de bibliotecas

INTRODUCCIÓN A LA ASIGNATURA

En esta materia, exploraremos a fondo cada aspecto fundamental del lenguaje de programación Java, ofreciéndote la oportunidad de expandir y consolidar tus habilidades en el apasionante mundo de la tecnología, nos sumergiremos en el origen y las características del lenguaje Java, abordando detalladamente temas como variables, constantes, operadores y expresiones. Además, utilizaremos la plataforma Eclipse, una poderosa herramienta que será nuestra aliada en el desarrollo de proyectos en Java.

A lo largo del curso, aprenderás a valorar la importancia de seleccionar estructuras de programación adecuadas en Java, utilizando ejemplos y casos prácticos como guías para aplicar estos conocimientos de manera efectiva con la ayuda de Eclipse.

Nos adentraremos en las funciones de control y operadores en Java, explorando el control de flujo, funciones como procedimientos y recursividad, brindándote las herramientas necesarias para simplificar y evaluar su aplicación, exploraremos en detalle el mundo de los objetos y clases en Java, examinando tipos de clases, variables, métodos, herencia, clases abstractas e interfaces, entre otros conceptos, y aplicándolos en la práctica donde expandirás tus habilidades en Java y explorarás nuevas dimensiones en el apasionante mundo de la programación

Rosa Virginia Ortega
Ing. de Computación
Docente Universitario

Una publicación

iutepi
Instituto
Universitario
de Tecnología
para la Informática

INDICE

Contenido

PROGRAMACIÓN 1
UNIDAD I 5
INTRODUCCIÓN A JAVA 5
INTRODUCCIÓN A JAVA 6
¿QUÉ ES EL LENGUAJE DE PROGRAMACIÓN JAVA? 8
HISTORIA DEL LENGUAJE JAVA 9
CARACTERISTICAS DE JAVA 11
VENTAJAS SOBRE OTROS LENGUAJES 13
DESVENTAJAS SOBRE OTROS LENGUAJES 14
ECOSISTEMA JAVA 15
APLICACIONES ACTUALES DE JAVA 16
TENDENCIAS Y FUTURO DE JAVA 18
PROGRAMACION ORIENTADA A OBJETOO POO 19
UNIDAD 2 21
VALORAR LA IMPORTANCIA DE LA SELECCIÓN DE ESTRUCTURAS DE PROGRAMACIÓN EN JAVA 21
VARIABLES 22
¿QUÉ SON LAS VARIABLES EN JAVA? 24
REGLAS PARA NOMBRAR VAIABLES EN JAVA 25
FUNCIONES DE CONTROL 26
EJERCICIO CÓDIGO CON OPERADORES 28
EXPRESIONES: 29
ARREGLOS O ARRAY 29
ACCESO A LOS ELEMENTOS DE UN ARREGLO 31
¿QUÉ ES UNA CLASE? 34
CLASE 36
¿QUÉ SON LOS OBJETOS EN JAVA? 37
¿QUÉ SON LOS MENSAJES EN JAVA? 37
EJEMPLO DE UNA CLASE EN JAVA 39
COMENTARIOS EN JAVA DE UNA SOLA LINEA 40
COMENTARIOS EN JAVA DE MULTIPLES LINEAS 41

TIPO DE DATOS PRIMITIVOS EN JAVA....................44
BLOQUE DE EJERCICIOS PARA PRACTICAR....................46
CUESTIONARIO DE EVALUACIÓN - Unidad 2: 49
Bloque de Ejercicios para Practicar en Java....................50
UNIDAD 3 51
FUNCIONES DE CONTROL Y OPERADORES JAVA 51
CONSTRUCTOR 52
¿QUÉ ES HERENCIA EN JAVA ? 53
SOBREESCRITURA DE MÉTODOS....................55
CLASES ABSTRACTAS 56
MÉTODOS Y CONSTRUCTORES 58
UNIDAD 4 62
OBJETOS Y CLASES EN JAVA 62
POLIMORFISMO 63
2. Polimorfismo en Tiempo de Ejecución (Sobreescritura de Métodos) 64
HERENCIA 65
Beneficios de la Herencia....................65
Ejemplo de Herencia en Java 65
SOBREESCRITURA DE MÉTODOS....................66
CLASES ABSTRACTAS 66
Métodos Abstractos....................66
Métodos Concretos....................67
CLASES ANIDADAS....................68
Tipos de Clases Anidadas....................68
Ejemplo en Eclipse 70
Creación y Ejecución en Eclipse....................71
ESTUDIO DE OBJETOS Y CLASES EN JAVA, ÁMBITO DE VARIABLES 72
Ámbito de Variables....................73
REFERENCIAS BIBLIOGRÁFICAS 67

UNIDAD I

INTRODUCCIÓN A JAVA

- Origen del Lenguaje Java
- ¿Qué es el Lenguaje de Programación Java?
- Historia del Lenguaje Java
- Características de Java
- Ventajas y Desventajas sobre otros lenguajes
- Ecosistema de Java y Aplicaciones Actuales de Java
- Tendencias y Futuro de Java
- Cuestionario Evaluativo

INTRODUCCIÓN A JAVA

Las redes sociales se han convertido en pilares fundamentales de la comunicación y la interacción en la era digital. Estas plataformas en línea facilitan la conexión entre personas, permitiéndoles comunicarse y compartir experiencias, pensamientos y emociones a través de Internet.

Al ofrecer un espacio virtual donde individuos de todo el mundo pueden conectarse instantáneamente, las redes sociales han transformado radicalmente la forma en que nos relacionamos. Desde mantener contacto con amigos y familiares hasta establecer nuevas conexiones profesionales, estas plataformas han creado una red global de relaciones personales y profesionales, enriqueciendo así la vida de las personas.

Además, las redes sociales han propiciado la creación de comunidades virtuales donde los usuarios comparten intereses y pasiones comunes, permitiendo el intercambio de conocimientos, la colaboración en proyectos y el apoyo mutuo. Estas comunidades

han abierto un mundo de posibilidades para la creación de contenido, el activismo social y la difusión de información relevante en tiempo real.

Las redes sociales no solo han redefinido la forma en que nos conectamos y comunicamos, sino que también han fomentado la creación de redes de apoyo, comunidades de aprendizaje y espacios de expresión individual y colectiva en la era digital. Se han convertido en poderosas herramientas que trascienden fronteras geográficas y culturales, creando un puente entre personas de diferentes partes del mundo y enriqueciendo nuestras vidas de múltiples maneras.

¿QUÉ ES EL LENGUAJE DE PROGRAMACIÓN JAVA?

Java es un lenguaje de programación de propósito general, orientado a objetos y tipado estáticamente que abarca una amplia gama de aplicaciones, desde programas básicos hasta complejas soluciones empresariales y aplicaciones móviles. Desde su concepción, Java ha destacado por su capacidad de ser multiplataforma y multidispositivo, fundamentado en el paradigma "Write Once Run Anywhere" (WORA).

Cuando un programa Java se compila, no se genera código máquina específico, sino que produce bytecodes. Estos bytecodes son interpretados por la Java Virtual Machine (JVM), una máquina virtual escrita para cada sistema operativo. De esta manera, un programa Java puede ejecutarse en diversos sistemas operativos como Windows, macOS y UNIX, así como en diferentes dispositivos.

HISTORIA DEL LENGUAJE JAVA

El renombrado lenguaje Java tuvo sus humildes comienzos en 1991 de la mano de James Gosling, quien en un principio lo denominó Oak o Green. Finalmente, la primera versión oficial de Java vio la luz en 1995, bajo el nombre JDK 1.0.2, y para el año 1996 se oficializó su nombre definitivo como Java.

La historia del lenguaje Java es un fascinante viaje que se remonta a la década de 1990. El desarrollo de Java comenzó en 1991 liderado por James Gosling y su equipo en Sun Microsystems, inicialmente concebido para dispositivos electrónicos, y posteriormente adaptado para el desarrollo de software para la World Wide Web. El 23 de mayo de 1995, Sun Microsystems lanzó oficialmente Java 1.0, marcando su nacimiento como lenguaje de programación. La promoción de Java se basó en su lema "Write Once, Run Anywhere" (Escribe una vez, ejecuta en cualquier lugar), destacando su portabilidad y capacidad de ejecución en diferentes plataformas y sistemas operativos.

A lo largo de su historia, Java se ha popularizado en el desarrollo de applets y ha competido con otros lenguajes, como JavaScript, en la web. En 2010, Oracle Corporation adquirió Sun Microsystems, marcando un nuevo capítulo en la historia de Java bajo su administración. No obstante, Java ha continuado evolucionando con nuevas versiones que presentan mejoras en rendimiento, seguridad y funcionalidad.

Java se ha convertido en el lenguaje principal para el desarrollo de aplicaciones en la plataforma Android, lo que ha impulsado su demanda. A pesar de no ser tan común como otros lenguajes, como C++.

CARACTERISTICAS DE JAVA

- **Independiente de Plataforma:** Los bytecodes generados por Java son interpretados por la JVM, permitiendo que un mismo código fuente pueda ejecutarse en múltiples plataformas sin modificación.
- **Orientado a Objetos:** En Java, todo se considera un objeto, lo cual favorece la encapsulación y modularidad del código, facilitando su reutilización.
- **Sencillez:** Java se destaca por ser un lenguaje amigable y fácil de aprender, ideado para simplificar el proceso de programación orientada a objetos.
- **Seguridad:** Los programas en Java se ejecutan en un
- entorno aislado

  (JVM) que garantiza la integridad y seguridad del sistema.

- **Portabilidad:** La característica multiplataforma de Java permite que las aplicaciones escritas en este lenguaje sean altamente portables y funcionales en distintas plataformas.

- ✓ **Robustez:** Java se enfoca en evitar errores, con procesos de compilación que detectan problemas de antemano y un eficiente manejo de memoria a través del "garbage collector".
- ✓ **Programación Multi-hilo:** Java es capaz de gestionar múltiples hilos de ejecución simultáneamente, permitiendo una mejor optimización de procesos.
- ✓ **Interpretado:** Los bytecodes de Java son interpretados en tiempo real, lo que garantiza su ejecución en cualquier dispositivo con una JVM compatible.
- ✓ **Alto Rendimiento:** Gracias al empleo de compiladores Just-In-Time (JIT), Java logra una eficiente traducción en tiempo real que mejora su rendimiento.

VENTAJAS SOBRE OTROS LENGUAJES

Python:

Ventaja de Java: Java brinda una robustez en la seguridad y un enfoque orientado a objetos riguroso, que lo distingue de Python, un lenguaje interpretado.

Comparación de portabilidad: Si bien Python es más rápido, carece de la portabilidad y el nivel de seguridad integral que ofrece Java.

C++:

Ventaja de Java: Java se destaca por su alto nivel de orientación a objetos y robustez en seguridad, frente a la complejidad y la dificultad de aprendizaje de **C++.**

Comparación de rendimiento: Aunque C++ es más rápido, Java sobresale

en términos de seguridad integrada.

JavaScript:

Ventaja de Java: Java, siendo un lenguaje clásico orientado a objetos,

contrasta con el modelo de prototipos de JavaScript.

Comparación de aplicaciones: Aunque JavaScript es estándar en navegadores, Java puede sobrepasarlo en aplicaciones autónomas.

DESVENTAJAS SOBRE OTROS LENGUAJES

Rendimiento en aplicaciones intensivas en CPU: Java puede verse superado por **C++** en aplicaciones exigentes con una gran carga de CPU.

Consumo de recursos del sistema: En entornos con recursos limitados, Java

puede consumir más recursos en comparación con lenguajes más livianos.

Complejidad sintáctica: La complejidad de la sintaxis y estructura de Java

puede dificultar la escritura de código rápido y limpio en comparación con

lenguajes de scripting.

ECOSISTEMA JAVA

JDK: Conjunto esencial de herramientas para desarrolladores Java, incluyendo

el compilador Java y la JVM.

Biblioteca Estándar de Java: Ofrece estructuras de datos, I/O, manipulación de archivos, concurrencia y más para un desarrollo acelerado.

Frameworks y bibliotecas: Abundantes recursos como Spring, Hibernate y Apache Struts agilizan el desarrollo de una variedad de aplicaciones.

IDEs: Entornos de desarrollo como Eclipse, IntelliJ IDEA y NetBeans ofrecen herramientas ricas en funcionalidades.

Herramientas de construcción (Build tools): Maven y Gradle facilitan la gestión de dependencias y la automatización de la construcción de proyectos.

OpenJDK y versiones de Java: La implementación de código abierto OpenJDK respaldada por la comunidad, junto con las actualizaciones regulares del lenguaje, aseguran un entorno dinámico y en evolución.

APLICACIONES ACTUALES DE JAVA

Desarrollo Web: A través de Servlets, JSP y frameworks como Spring. **Aplicaciones Móviles:** Java es el lenguaje principal en el desarrollo de aplicaciones Android.

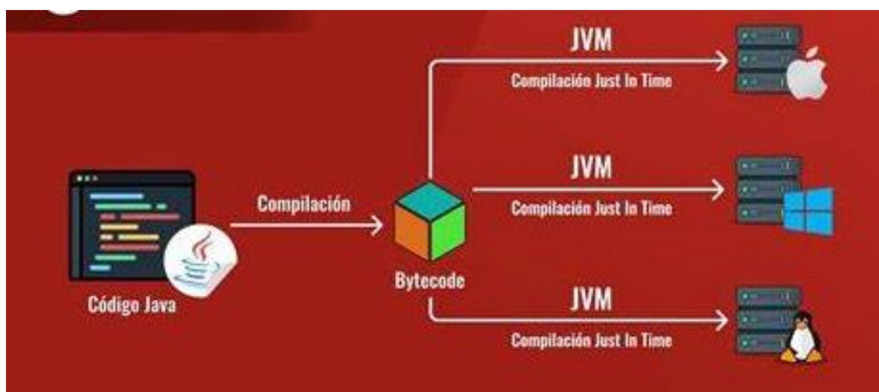
Juegos y Análisis de Datos: Utilizado en juegos móviles, análisis de datos

con herramientas como Apache Spark.

Dispositivos IoT y Automatización del Hogar: Presente en sistemas IoT y domótica.

Gestión de Bases de Datos: Utilizado en sistemas de gestión de bases de datos como Apache Cassandra y Neo

TENDENCIAS Y FUTURO DE JAVA



Aplicaciones en la Nube y Servicios Web: Java es ampliamente empleado en estos entornos, junto con tecnologías como Docker y Kubernetes.

Crecimiento de Kotlin: Como opción popular para aplicaciones Android, Kotlin se ha vuelto relevante en la interacción con Java.

Aplicaciones en Big Data y Machine Learning: Java es fundamento para frameworks como Apache Hadoop y Apache Spark, impulsando su presencia en est

PROGRAMACION ORIENTADA A OBJETOO POO

Java es un lenguaje orientado a objetos en el que prácticamente todo se considera un objeto. Este enfoque favorece la encapsulación y el modularidad del código, lo que a su vez facilita la reutilización del código y el desarrollo de sistemas más estructurados y mantenibles.

La programación orientada a objetos se basa en cuatro principales conceptos: encapsulación, herencia, polimorfismo y abstracción. Java incorpora estos conceptos de forma integral, lo que permite a los desarrolladores crear sistemas complejos mediante la combinación y manipulación de objetos.

La encapsulación permite restringir el acceso a ciertas componentes internas de los objetos, protegiendo así el estado de un objeto y garantizando que solamente métodos autorizados puedan manipularlo. Esto promueve la seguridad y el manejo controlado de los datos.

La herencia permite la creación de jerarquías de clases, lo que facilita la reutilización del código y la implementación de comportamientos comunes en múltiples clases.

CUESTIONARIO DE EVALUACIÓN –
Unidad 1: Origen del Lenguaje Java

1. ¿Quiénes fueron los creadores originales del lenguaje Java?

A) James Gosling
B) Mike Sheridan
C) Patrick Naughton
D) Todas las anteriores

2. ¿Por qué surgió la necesidad de desarrollar un nuevo lenguaje de programación como Java?

A) Para crear un lenguaje específico para dispositivos móviles.
B) Para abordar las limitaciones de otros lenguajes existentes.
C) Para competir con otros lenguajes de programación populares.
D) Todas las anteriores

3. Contexto tecnológico en el que se originó Java.

A) Se originó en el contexto de un boom en el desarrollo de software empresarial.
B) Surgió en un momento en el que el internet estaba comenzando a expandirse.
C) Se creó como respuesta a la demanda de lenguajes de programación más seguros y portátiles.
D) Todas las anteriores

4. ¿Cuál fue el objetivo principal al crear el lenguaje Java?

A) Facilitar la creación de videojuegos.
B) Desarrollar un lenguaje seguro y portátil.
C) Competir con otros lenguajes como C++.
D) Aumentar la complejidad de la programación

UNIDAD 2

VALORAR LA IMPORTANCIA DE LA SELECCIÓN DE ESTRUCTURAS DE PROGRAMACIÓN EN JAVA

- ¿Qué son las Variables en Java?
- Variables, constantes, operadores
- Expresiones en Java
- Arreglos O Array
- ¿Qué es una Clase?
- Clase
- ¿Qué son los Objetos en Java?
- Tipos de Clases
- ¿Qué son los mensajes en Java?
- Encapsulación de Datos
- Tipo de Datos Primitivos
- Métodos y Constructores
- Bloque de Ejercicios para Practicar en Java

VARIABLES

En Java, se pueden utilizar diferentes tipos de datos para declarar variables,

como:

- **Enteros (int):** para números enteros como 5, 300, -45, etc.
- **Flotantes (float, double):** para números con decimales como 3.14, -

  15.7, etc.
- **Caracteres (char):** para un solo carácter como 'a', '&', etc.
- **Booleanos (boolean):** para valores lógicos verdadero/falso.
- **Cadenas (String):** para texto entre comillas como "Hola mundo".

El alcance o ámbito de una variable en Java se refiere a las partes del código donde esa variable es accesible. Se pueden distinguir variables de alcance local, parámetros y atributos, cada uno con su propio ámbito de acceso.

En Java, se pueden declarar variables con palabras reservadas que permiten

una declaración más concisa, como:

- **final para constantes. Ejemplo:** final int MAX = 100;

- **static para variables de clase.**

  Ejemplo: static double PI = 3.14;

Es fundamental recordar no utilizar nombres de variables que sean palabras

reservadas en Java para evitar conflictos.

Las constantes en Java son contenedores de datos cuyo valor no puede cambiar una vez asignado. Se definen con la palabra reservada final para asegurar que el valor permanezca constante a lo largo de la ejecución del programa**.**

Variables primitivas

Nombre	Tipo	Valor mínimo	Valor máximo
byte	Entero	-127	128
short	Entero	-32768	32767
int	Entero	-2147483648	2147483647
long	Entero	-9.223372^{18}	9.223372^{18}
float	Decimal Simple	1.4^{-45}	$3.4028235 \cdot 10^{38}$
double	Decimal Entero	4.9^{-324}	1.7976931348623157^{308}
char	Caracter Simple	_	_
boolean	True/False	_	_

¿QUÉ SON LAS VARIABLES EN JAVA?

En Java, una variable es un contenedor en la memoria que almacena valores. Estos valores pueden ser de diferentes tipos, como números, texto o booleanos. Las variables nos permiten referenciar estos valores utilizando un nombre descriptivo.

TIPOS DE VARIABLES

En Java, existen diferentes tipos de variables que podemos utilizar:

Variables primitivas: almacenan valores simples, como números enteros, decimales o caracteres.

Variables de referencia: almacenan referencias a objetos en memoria.

DECLARACIÓN DE VARIABLES EN JAVA

tipoDeVatiable nombreDeVariable;

Para declarar una variable en Java, utilizamos la siguiente sintaxis

Por ejemplo, si queremos declarar una variable de tipo entero llamada "edad", podemos hacerlo de la siguiente manera:

Int edad;

REGLAS PARA NOMBRAR VAIABLES EN JAVA

Al nombrar variables en Java, es importante seguir algunas reglas:

- El nombre de la variable debe comenzar con una letra o un guion bajo.
- No se pueden utilizar espacios en el nombre de la variable.
- El nombre de la variable distingue entre mayúsculas y minúsculas.

Ámbito de las variables en Java

El ámbito de una variable determina dónde puede ser utilizada. En Java, existen tres niveles de ámbito:

Variables locales: declaradas dentro de un método o bloque de código y solo pueden ser accedidas dentro de ese contexto.

Variables de instancia: declaradas dentro de una clase y pueden ser accedidas por cualquier método de esa clase.

Variables estáticas: declaradas con la palabra clave "static" y se comparten entre todas las instancias de una clase.

Inicialización de variables en Java

Una variable puede ser inicializada con un valor en el momento de su declaración. Por ejemplo:

Int edad = 25;

FUNCIONES DE CONTROL

constantes, operadores y expresiones

Funciones de control

Las funciones de control son estructuras en un lenguaje de programación que permiten controlar el flujo de ejecución de un programa. Esto incluye estructuras como decisiones condicionales (if, else, else if), bucles (for, while, do-while), y otras formas de controlar el flujo de código a través de un programa.

Constantes

Las constantes son valores que no cambian a lo largo del programa. En Java, se definen utilizando la palabra clave final. Una vez que se asigna

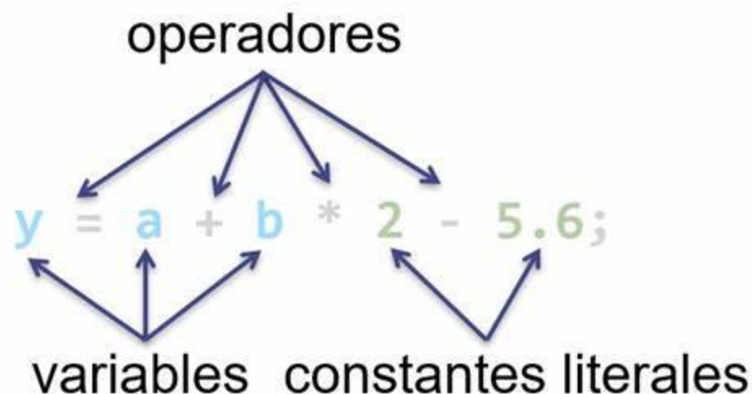
un valor a una constante, este no puede ser modificado durante la ejecución del programa.

Operadores

Los operadores son símbolos especiales que realizan operaciones en una o más variables. En Java, los operadores incluyen los operadores aritméticos (+, -, *, /, %), operadores de comparación (==, !=, <, >, <=, >=), operadores lógicos (&&, ||, !), operadores de asignación (=, +=, -=, *=, /=, %=), entre otros.

Expresiones

En programación, una expresión es una combinación de variables, operadores y valores que da como resultado un valor. Por ejemplo, en Java, una expresión puede ser a + b, donde a y b son variables y + es el operador. Las expresiones pueden ser tan simples como una sola variable o tan complejas como una combinación de operaciones.



EJERCICIO CÓDIGO CON OPERADORES

```
public class CalculoDescuento {
    public static void main(String[] args) {
        double precioProducto = 100.0;  // Precio del producto sin descuento
        int cantidad = 3;  // Cantidad de productos comprados
        double descuento = 0.1;  // Descuento del 10%

        // Cálculo del monto total sin descuento
        double montoTotalSinDescuento = precioProducto * cantidad;

        // Cálculo del monto total con descuento
        double montoTotalConDescuento = montoTotalSinDescuento - (montoTotalSinDescuento * descuento);

        // Determinar si el descuento supera cierto umbral
        boolean descuentoSignificativo = descuento > 0.05;

        // Mostrar los resultados
        System.out.println("Monto total sin descuento: " + montoTotalSinDescuento);
        System.out.println("Monto total con descuento: " + montoTotalConDescuento);
        System.out.println("¿El descuento es significativo?: " + descuentoSignificativo);
    }
}
```

En este ejercicio, utilizamos los operadores aritméticos para realizar cálculos de precios y descuentos, y el operador relacional para determinar si el descuento aplicado es significativo.

Este ejercicio te permite practicar el uso de operadores aritméticos y relacionales en Java, así como para comprender cómo utilizarlos en situaciones del mundo real.

EXPRESIONES:

En Java, una expresión es una combinación de variables, operadores y valores que da como resultado un valor. Por ejemplo:

```
int resultado = (5 * 3) + (8 / 2);
// Expresión aritmética
boolean esMayor = (edad > 18);
// Expresión de comparación
```

funciones de control, constantes, operadores y expresiones son elementos fundamentales en el desarrollo de programas en Java.

```
public class CalculadoraIMC {
    public static void main(String[] args) {
        double peso = 68.5;  // Peso en kilogramos
        double altura = 1.75;  // Altura en metros
        // Cálculo del índice de masa corporal (IMC)
        double imc = peso / (altura * altura);
        System.out.println("Tu IMC es: " + imc);
    }
}
```

ARREGLOS O ARRAY

Es un conjunto de elementos ordenados de un mismo tipo y se declara de la siguiente manera

Tipo_datos[] nombre_array;

los array también pueden contener objetos. Se puede acceder a cada elemento del array por medio de un índice que comienza por 0.

ejemplo1: char[]nombres;

ejemplo3: Vehiculos[]array_de_transporte;

En el contexto de la programación, un array (o arreglo) en Java es una estructura de datos que permite almacenar múltiples elementos del mismo tipo bajo un mismo nombre. Los arrays son útiles para organizar y manipular conjuntos de datos de manera eficiente. Aquí tienes información clave sobre los arrays en Java:

Los arreglos son fundamentales en la programación ya que nos permiten trabajar de manera eficiente con conjuntos de datos relacionados.:

Declaración de un Arreglo

Para declarar un arreglo en Java, se utiliza la siguiente sintaxis:

```
int[] numeros = new int[5]; // Crear un arreglo de enteros con tamaño 5
//Forma abreviada de inicialización
int[] numeros = {1, 2, 3, 4, 5}; // Inicializar el arreglo con valores
```

ACCESO A LOS ELEMENTOS DE UN ARREGLO

Para acceder a los elementos de un arreglo, utilizamos el índice del elemento dentro de corchetes. El índice comienza en 0 y va hasta la longitud del arreglo menos uno.

```
int[] numeros = {10, 20, 30, 40, 50};

System.out.println(numeros[0]); // Imprime el primer elemento del arreglo (10)

System.out.println(numeros[2]); // Imprime el tercer elemento del arreglo (30)
```

Recorrido de un Arreglo

Para recorrer todos los elementos de un arreglo, podemos utilizar un bucle for:

```
int[] numeros = {10, 20, 30, 40, 50};

for (int i = 0; i < numeros.length; i++) {
   System.out.println(numeros[i]);
}
```

Arreglos Multidimensionales

Java también permite la creación de arreglos multidimensionales. Por ejemplo, un arreglo bidimensional se declara e inicializa de la siguiente manera:

```
int [][] matriz = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
System.out.println(matriz[0][1]); // Acceder al elemento en la primera fila y segunda columna (2)
```

Los arreglos en Java son una herramienta poderosa para trabajar con colecciones de datos, y comprender su uso y manejo es esencial para desarrollar aplicaciones eficientes y robustas.

Creación de un Array:

Después de declarar un array, se puede crear e inicializar utilizando la palabra clave new. Por ejemplo:

```
numeros = new int[5];
// Creación de un array de enteros con capacidad para 5 elementos
nombres = new String[10];
 // Creación de un array de cadenas de texto con capacidad para 10 elementos
```

Inicialización de un Array:

También es posible inicializar un array al momento de su creación, especificando los valores iniciales entre llaves {}. Por ejemplo:

```
int[] numeros = {10, 20, 30, 40, 50};
// Inicialización de un array de enteros con valores iniciales

String [] diasSemana = {"Lunes", "Martes", "Miércoles", "Jueves", "Viernes",

"Sábado", "Domingo"}; // Inicialización de un array de cadenas de texto
```

Acceso a Elementos de un Array:

Los elementos de un array se acceden mediante su índice, comenzando desde 0. Por ejemplo:

```
int primerNumero = numeros[0];
 // Acceso al primer elemento del array 'numeros'
String tercerDia = diasSemana[2];
// Acceso al tercer elemento del array 'diasSemana'
```

Longitud de un Array:

La longitud de un array (es decir, la cantidad de elementos que puede almacenar) se obtiene con la propiedad length. Por ejemplo:

```
int cantidadElementos = numeros.length;
// Obtiene la longitud del array 'numeros'
int cantidadDias = diasSemana.length;
// Obtiene la longitud del array 'diasSemana'
```

Los arrays son fundamentales en Java y proporcionan una forma eficiente de trabajar con conjuntos de datos.

Los arrays son fundamentales en Java y proporcionan una forma eficiente de trabajar con conjuntos de datos

```
//Declarar e inicializar un arreglo de enteros
int[] numeros = {10, 20, 30, 40, 50};

//Acceder a elementos del arreglo
System.out.println("Primer elemento: " + numeros[0]);
System.out.println("Tercer elemento: " + numeros[2]);
```

Los arreglos son una herramienta fundamental en la programación que te permite almacenar y manipular colecciones de datos de manera eficiente.

- **Definición**: Colección de elementos del mismo tipo, accesibles por un índice.
- **Declaración y Creación**: Se pueden declarar y crear con tamaños específicos.
- **Acceso y Modificación**: Los elementos se acceden y modifican usando su índice.
- **Recorrido**: Se pueden recorrer utilizando bucles, como for o for-each.
- **Multidimensionales**: Soportan múltiples dimensiones, como matrices bidimensionales.

¿QUÉ ES UNA CLASE?

En el contexto de la programación en Java, una clase representa un modelo o prototipo que define tanto las variables (atributos que distinguen a cada objeto) como los métodos (comportamientos que pueden ejecutarse) que son comunes a un determinado tipo de objetos.

Tomando como ejemplo una fábrica de galletas, podríamos asimilar una clase a uno de los moldes utilizados para fabricarlas, que incluye los métodos para decorarlas y la receta para elaborarlas. En este sentido, en un programa Java, la clase nos proporciona una estructura para definir las características de cada galleta teniendo en cuenta que existen diferentes moldes disponibles.

En Java, las clases funcionan como plantillas a partir de las cuales se pueden crear múltiples instancias u objetos del mismo tipo. La clase establece las variables y los métodos que son comunes a todas las instancias de ese tipo (es decir, a los objetos creados a partir de la clase), pero cada instancia posee sus propios valores únicos (su molde específico, color y receta), aun compartiendo las mismas funcionalidades.

En Java, es necesario definir una clase antes de poder crear instancias o ejemplares de la misma. Esta analogía se asemeja a la necesidad de contar con los moldes y demás elementos necesarios para elaborar las galletas antes de producirlas en la fábrica.

CLASE

Las clases representan los prototipos de los objetos que tenemos en el mundo real. Es decir, es una generalización de un conjunto de objetos. A su vez los objetos serán instancias de una determinada clase.

Si volvemos al ejemplo del televisor, existen múltiples tipos de televisores y cada uno con sus características. Si bien existe un esquema o prototipo que define el televisor. Este prototipo es lo que conocemos la clase.

En la clase es dónde realmente definimos las propiedades y métodos que podrán contener cada una de las instancias de los objetos.

Por ejemplo, para nuestro caso de las figuras geométricas podríamos definir un triángulo de la siguiente forma:

```
class Triangulo {
    private long base;
    private long altura;

    public Triangulo(long base, long altura) {
        this.base = base;
        this.altura = altura;
    }

    public long area() {
        return (base*altura)/2;
    }
}

Triangulo t1 = new Triangulo(2.0,3.0);
Triangulo t2 = new Triangulo(4.0,7.0);

t1.area(); // Área 3.0
t2.area(); // Área 14.0
```

¿QUÉ SON LOS OBJETOS EN JAVA?

En el ámbito del lenguaje de programación Java, un objeto es básicamente una instancia de una clase. Para nuestro ejemplo de la fábrica de galletas, los objetos vendrían siendo cada una de las diferentes galletas obtenidas de los moldes definidos (clases).

Lo más importante de los objetos en Java es que permiten tener un control total sobre "quién" o "qué" puede acceder a sus miembros, es decir, los objetos pueden tener miembros públicos a los que podrán acceder otros objetos o miembros privados a los que sólo puede acceder él. Estos miembros pueden ser tanto variables como funciones.

El gran beneficio de todo esto es la encapsulación. En java el código fuente de un objeto puede escribirse y mantenerse de forma independiente a los otros objetos contenidos en la aplicación

¿QUÉ SON LOS MENSAJES EN JAVA?

Para poder crear una aplicación Java que sea un tanto robusta, es muy probable que necesitemos más de un objeto y probablemente

algunos de estos objetos no deben estar aislados unos de otros, pues bien, para comunicarse, esos objetos se envían mensajes entre sí.

Los mensajes son simples llamadas a las funciones o métodos del objeto en particular con el cual se quiere comunicar para solicitarle que ejecute alguna "accion" según sus métodos y/o atributos.

```
/Definición de la clase Persona
public class Persona {
 // Método para saludar a otra persona
 public void saludar(String nombre) {
    System.out.println("¡Hola, " + nombre + "! ¡Mucho gusto!");
 }
}

//Clase principal que contiene el método main
public class Main {
 public static void main(String[] args) {
    // Crear un objeto de la clase Persona
    Persona persona = new Persona();

    // Enviar un mensaje al objeto persona para que salude
    persona.saludar("Juan");
 }
}
```

EJEMPLO DE UNA CLASE EN JAVA

```
//Definición de la clase Producto
class Producto {
 // Atributos de la clase Producto
 private String nombre;
 private double precio;

 // Constructor de la clase Producto
 public Producto(String nombre, double precio) {
     this.nombre = nombre;
     this.precio = precio;
 }

 // Método para obtener el nombre del producto
 public String obtenerNombre() {
     return nombre;
 }

 // Método para obtener el precio del producto
 public double obtenerPrecio() {
     return precio;
 }

 // Método para establecer un nuevo precio al producto
 public void establecerPrecio(double nuevoPrecio) {
     this.precio = nuevoPrecio;
 }
}

//Clase principal que contiene el método main
public class Main {
 public static void main(String[] args) {
     // Crear un objeto de la clase Producto
     Producto producto1 = new Producto("Camisa", 25.99);

     // Mostrar información del producto
     System.out.println("Producto: " + producto1.obtenerNombre());
     System.out.println("Precio: $" + producto1.obtenerPrecio());

     // Cambiar el precio del producto
     producto1.establecerPrecio(29.99);
     System.out.println("Precio actualizado: $" + producto1.obtenerPrecio());
 }
```

}

En este ejemplo, se crea una clase Estudiante con atributos para el nombre, edad y número de estudiante. Se define un constructor para inicializar estos atributos, métodos getter y setter para acceder y modificar los atributos, y un método mostrarInformacion() para imprimir los detalles del estudiante.

COMENTARIOS EN JAVA DE UNA SOLA LINEA

Pueden ser colocados en cualquier parte de nuestro código en Java y comienzan por un doble slash "//", al colocar el doble slash en cualquier línea de código, todo lo que haya de ahí en adelante en dicha línea será tomado como comentario, ten en cuenta que el doble slash solo convierte en comentario al texto que haya justo después de éstos y que pertenezca a su misma línea, las líneas de abajo de este, no se verán afectadas, tal como es de esperarse, el doble slash "//", solo afecta una línea desde el lugar donde se colocan.

```
public class Ejemplo {
    public static void main(String[] args) {
        // Inicio del método principal
        System.out.println("¡Hola, mundo!"); // Imprime un saludo en la consola

        // Declaración e inicialización de una variable
        int numero = 10; // Asigna el valor 10 a la variable numero

        // Llamada a una función
        mostrarMensaje(); // Llama a la función mostrarMensaje
    }

    public static void mostrarMensaje() {
        System.out.println("Este es otro mensaje."); // Imprime otro mensaje
    }
}
```

COMENTARIOS EN JAVA DE MULTIPLES LINEAS

. Los comentarios multilínea en Java tal como el nombre lo indica nos permiten comentar varias líneas de nuestro código Java de manera mucho más sencilla en vez de esta añadiendo doble slash "//" a cada línea. Estos comentarios van cerrados entre "/*" y "*/", es decir comienzan donde se ponga "/*" y terminan donde esté el "*/". Estos comentarios funcionan de manera similar a los comentarios de una sola línea, pero deben tener un comienzo y un final

```
public class Ejemplo {
    public static void main(String[] args) {
        /*
         * Este es un comentario de múltiples líneas.
         * Se utiliza para explicar secciones más grandes del código
         * o proporcionar descripciones detalladas.
         */
        System.out.println("¡Hola, mundo!"); // Imprime un saludo en la consola

        /*
         * Declaración e inicialización de una variable.
         * En este caso, estamos asignando el valor 10 a la variable numero.
         */
        int numero = 10;

        /*
         * Llamada a una función.
         * Aquí llamamos a la función mostrarMensaje que imprime otro mensaje en la
consola.
         */
        mostrarMensaje();
    }

    public static void mostrarMensaje() {
        System.out.println("Este es otro mensaje."); // Imprime otro mensaje
    }
}
```

ENCAPSULACIÓN DE DATOS

Las interacciones con los objetos se hacen mediante los métodos. Es decir, si queremos conocer información del estado del objeto deberemos de llamar a uno de sus métodos y no directamente a las propiedades.

Esta encapsulación nos permitiría el cambiar las propiedades del objeto sin que los consumidores se vean afectados siempre y cuando les sigamos retornando el mismo resultado.

Si bien hay objetos que tienen propiedades públicas, por lo cual podremos acceder directamente a dichas propiedades sin necesidad de utilizar un método.

El uso de objetos nos proporciona los siguientes beneficios:

- ***Modularidad***, el objeto y sus propiedades puede ser pasado por diferentes estructuras del código fuente, pero el objeto es el mismo.

- ***Encapsular Datos***, ocultamos la implementación de propiedades del objeto ya que accederemos a través de los métodos del objeto.
- ***Reutilización de Código,*** podemos tener diferentes instancias de un objeto de tal manera que esas diferentes instancias están compartiendo el mismo código.

- ***Reemplazo***, podemos reemplazar un objeto por otro siempre y cuando estos objetos tengan el mismo comportamiento.

```
public class Persona {
    private String nombre;
    private int edad;

    // Método getter para obtener el nombre
    public String getNombre() {
        return nombre;
    }

    // Método setter para establecer el nombre
    public void setNombre(String nuevoNombre) {
        this.nombre = nuevoNombre;
    }

    // Método getter para obtener la edad
    public int getEdad() {
        return edad;
    }

    // Método setter para establecer la edad
    public void setEdad(int nuevaEdad) {
        if (nuevaEdad > 0 && nuevaEdad < 120) {
            this.edad = nuevaEdad;
        } else {
            System.out.println("Edad inválida");
        }
    }
}
```

En este ejemplo, la clase Persona contiene dos campos nombre y edad que están marcados como private, lo que significa que no pueden ser accedidos directamente desde fuera de la clase. Los métodos getNombre, setNombre, getEdad y setEdad proporcionan acceso controlado a estos campos. Los métodos get permiten obtener el valor de los campos, mientras que los métodos set permiten establecer nuevos valores, y en este caso, se incluye una validación para la edad.

```
public class Main {
    public static void main(String[] args) {
        Persona persona = new Persona();
        persona.setNombre("Juan");
        persona.setEdad(25);

        System.out.println("Nombre: " + persona.getNombre());
        System.out.println("Edad: " + persona.getEdad());
    }
}
```

La encapsulación de datos en Java se logra mediante el uso de modificadores de acceso y métodos getter y setter, lo que ayuda a mantener la integridad de los datos y a controlar su manipulación.

TIPO DE DATOS PRIMITIVOS EN JAVA

Como ya hemos comentado Java es un lenguaje de tipado estático. Es decir, se define el tipo de dato de la variable a la hora de definir esta. Es por ello que todas las variables tendrán un tipo de dato asignado.

byte

Representa un tipo de dato de 8 bits con signo. De tal manera que puede almacenar los valores numéricos de -128 a 127 (ambos inclusive).

short

Representa un tipo de dato de 16 bits con signo. De esta manera almacena valores numéricos de -32.768 a 32.767.

int

Es un tipo de dato de 32 bits con signo para almacenar valores numéricos. Cuyo valor mínimo es -231 y el valor máximo 231-1.

long

Es un tipo de dato de 64 bits con signo que almacena valores numéricos entre - 263 a 263-1

float

Es un tipo dato para almacenar números en coma flotante con precisión simple de 32 bits.

double

Es un tipo de dato para almacenar números en coma flotante con doble precisión de 64 bits.

boolean

Sirve para definir tipos de datos booleanos. Es decir, aquellos que tienen un

valor de true o false. Ocupa 1 bit de información.

char

Es un tipo de datos que representa a un carácter Unicode sencillo de 16 bit

```
import java.util.Scanner;

public class CalculoAreaCirculo {
    public static void main(String[] args) {
        final double PI = 3.14159;
// Definir el valor de PI como constante

        Scanner scanner = new Scanner(System.in);

        System.out.print("Ingresa el radio del círculo: ");
        double radio = scanner.nextDouble();
// Leer el radio del círculo desde la entrada del usuario

        // Calcular el área del círculo
        double area = PI * radio * radio;

        // Mostrar el resultado
        System.out.println("El área del círculo con radio " + radio + " es: " + area);
    }
}
```

BLOQUE DE EJERCICIOS PARA PRACTICAR

Ejercicio 1: Ciclo While

En este ejercicio, se utiliza un ciclo while para imprimir los primeros 10 números pares.

```
public class EjercicioWhile1 {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 10) {
            System.out.println(i * 2);
            i++;
        }
    }
```

Ejercicio 2: Ciclo Do-While

En este ejercicio, se utiliza un ciclo do-while para pedir al usuario que adivine un número aleatorio entre 1 y 10. El ciclo se repetirá hasta que el usuario adivine el número

```
import java.util.Scanner;
import java.util.Random;

public class EjercicioDoWhile {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();
        int numeroAleatorio = random.nextInt(10) + 1;
        int intento;
        do {
            System.out.print("Adivina el número (entre 1 y 10): ");
            intento = scanner.nextInt();
            if (intento != numeroAleatorio) {
                System.out.println("¡Inténtalo de nuevo!");
            }
        } while (intento != numeroAleatorio);
}
```

Programa para verificar si un número es par o impar usando "if-else":

```
public class ParImpar {
    public static void main(String[] args) {
        int numero = 7;
        if(numero % 2 == 0) {
            System.out.println(numero + " es un número par.");
        } else {
            System.out.println(numero + " es un número impar.");
        }
    }
}
```

Utilizando un bucle "while" para imprimir los primeros 10 números naturales en orden descendente:

```
public class NumerosNaturales {
    public static void main(String[] args) {
        int i = 10;
        while(i >= 1) {
            System.out.println(i);
            i--;
        }
    }
}
```

Programa que permite al usuario adivinar un número utilizando un bucle "do-while":

```
import java.util.Scanner;
import java.util.Random;

public class AdivinarNumero {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();
        int numeroAleatorio = random.nextInt(100) + 1;
        int intento;
        do {
            System.out.print("Adivina el número (entre 1 y 100): ");
            intento = scanner.nextInt();
            if(intento > numeroAleatorio) {
                System.out.println("El número a adivinar es menor.");
            } else if(intento < numeroAleatorio) {
                System.out.println("El número a adivinar es mayor.");
            }
        } while(intento != numeroAleatorio);
        System.out.println("¡Felicidades! Has adivinado el número.");
    }
}
```

bucle do-while para sumar los dígitos de un número entero positivo ingresado por el usuario:

```
import java.util.Scanner;

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

        int numero;
        do {
            System.out.print("Ingresa un número entero positivo (o 0 para salir): ");
            numero = scanner.nextInt();

            if (numero > 0) {
                int sumaDigitos = 0;
                int num = numero;
                while (num > 0) {
                    sumaDigitos += num % 10;
                    num /= 10;
                }
                System.out.println("La suma de los dígitos de " + numero + " es: " + sumaDigitos);
            } else if (numero < 0) {
                System.out.println("El número ingresado no es positivo. Por favor, ingresa un número
positivo.");
            }
        } while (numero != 0);

        System.out.println("Fin del programa");
    }
}
```

CUESTIONARIO DE EVALUACIÓN - Unidad 2:

1. **¿Qué es una variable en Java?**
   - A) Un contenedor que almacena múltiples valores.
   - B) Un contenedor que almacena un valor único.
   - C) Una función que realiza cálculos matemáticos.
   - D) Un tipo de dato constante.
2. **¿Cuál de las siguientes opciones NO es un tipo de dato primitivo en Java?**
   - A) int
   - B) float
   - C) String
   - D) boolean
3. **¿Qué es una clase en Java?**
   - A) Un conjunto de constantes.
   - B) Un contenedor para objetos y métodos.
   - C) Un tipo de variable.
   - D) Un tipo de operador.
4. **¿Qué es un objeto en Java?**
   - A) Un operador que realiza cálculos.
   - B) Una instancia de una clase.
   - C) Una variable constante.
   - D) Un método estático.
5. **¿Cuál es el propósito de los métodos y constructores en Java?**
   - A) Definir variables.
   - B) Inicializar objetos.
   - C) Ejecutar loops.
   - D) Almacenar datos.
6. **¿Qué es la encapsulación de datos?**
   - A) Un método para almacenar datos.
   - B) Un proceso para crear variables.
   - C) Un principio para restringir el acceso a los datos.
   - D) Una clase que contiene múltiples métodos.
7. **¿Cuál de las siguientes NO es una expresión en Java?**
   - A) a + b
   - B) x * y
   - C) 5 < 3
   - D) new Object()

8. **¿Qué son los arreglos o arrays en Java?**
   - A) Una colección de métodos.
   - B) Una colección de objetos.
   - C) Una colección de clases.
   - D) Una colección de elementos del mismo tipo.
9. **¿Qué describe mejor un mensaje en Java?**
   - A) Una línea de comentario.
   - B) Un método estático.
   - C) La comunicación entre objetos a través de métodos.
   - D) Un valor constante.
10. **¿Qué representan las constantes en Java?**
    - A) Valores que no cambian.
    - B) Variables que cambian de valor.
    - C) Operadores matemáticos.
    - D) Clases y objetos.
11. **¿Cuál de las siguientes afirmaciones es correcta sobre tipos de clases en Java?**
    - A) Todos los tipos de clases deben ser públicas.
    - B) Solo existen clases privadas y públicas.
    - C) Puede haber clases abstractas y finales.
    - D) Las clases no pueden contener métodos.
12. **¿Cuál de las siguientes NO es una práctica común en Java?**
    - A) Declarar variables.
    - B) Usar constantes.
    - C) Definir métodos sin cuerpo.
    - D) Utilizar operadores.

Bloque de Ejercicios para Practicar en Java

1. Declara una variable int y asígnale el valor 10.
2. Crea una clase llamada Persona con atributos nombre y edad.
3. Declara un arreglo de 5 elementos y asígnales valores.
4. Escribe un método que devuelva la suma de dos números.
5. Crea un constructor para la clase Persona.

UNIDAD 3

FUNCIONES DE CONTROL Y OPERADORES JAVA

- Constructor
- ¿Qué es la herencia en Java?
- Herencia, subclases
- Sobreescritura de Métodos
- Clases abstractas, interfaces
- Clases anidadas
- métodos y constructores

CONSTRUCTOR

Un constructor en Java es un tipo especial de método que se utiliza para inicializar objetos de una clase. Se llama "constructor" porque se utiliza para construir (es decir, inicializar) objetos. Los constructores tienen el mismo nombre que la clase a la que pertenecen y no tienen tipo de retorno, ni siquiera void.

En resumen, un constructor realiza las siguientes tareas:

1. Inicializa los campos de datos de un objeto.
2. Realiza cualquier inicialización adicional requerida por el objeto antes de que pueda ser utilizado.

Aquí tienes un ejemplo de un constructor en Java:

```
public class Persona {
    private String nombre;
    private int edad;

    // Constructor de la clase Persona
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Otros métodos de la clase Persona
    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }
```

¿QUÉ ES HERENCIA EN JAVA ?

¿Qué significa esto de la herencia?, ¿quién hereda qué?, bueno calma, no entres en pánico, esto sólo significa que en Java puedes crear una clase partiendo de otra que ya exista.
Es decir, puedes crear una clase a través de una clase existente, y esta clase tendrá todas las variables y los métodos de su "superclase", y además se le podrán añadir otras variables y métodos propios.

Se llama "Superclase" a aquella clase de la cual desciende una clase, más adelante veremos un poco más de esto al detalle.
Vamos a ver un pequeño ejemplo basado en la fábrica de galletas que hemos estado tratando a lo largo de este artículo. Vamos a crear una pequeña clase llamada "Galleta" que contendrá los diferentes atributos.

La herencia es un concepto fundamental en la programación orientada a objetos que permite a una clase heredar atributos y métodos de otra clase. En este contexto, la "subclase" hace referencia a la clase que hereda de otra clase, denominada superclase o clase base.

Este es un ejemplo simple para ilustrar la herencia y las subclases en Java:

EJEMPLO DE HERENCIA

```
//Definición de la subclase Coche que hereda de Vehiculo
class Coche extends Vehiculo {
 private int numeroPuertas;

 public Coche(String marca, String modelo, int numeroPuertas) {
    super(marca, modelo); // Llamada al constructor de la superclase
    this.numeroPuertas = numeroPuertas;
 }

 public void mostrarInformacionCoche() {
    mostrarInformacion(); // Llamada al método de la superclase
    System.out.println("Número de puertas: " + numeroPuertas);
 }
}

//Clase principal que contiene el método main
public class Main {
 public static void main(St}
ring[] args) {
    // Crear un objeto de la subclase Coche
    Coche miCoche = new Coche("Toyota", "Corolla", 4);
    miCoche.mostrarInformacionCoche();
 }
}
```

En este ejemplo, la clase Vehículo es la superclase que tiene atributos como marca y modelo, así como un método mostrarInformación para mostrar la información del vehículo.

La clase Coche es una subclase de Vehículo que agrega un atributo numeroPuertas y un método adicional mostrarInformacionCoche que muestra la información específica de un coche.

Al crear un objeto de la clase Coche, se puede acceder a los métodos y atributos tanto de la superclase Vehiculo como de la subclase Coche.

SOBREESCRITURA DE MÉTODOS

La sobreescritura de métodos es un concepto importante en la programación orientada a objetos que permite a una subclase proporcionar una implementación específica para un método que ya está definido en su superclase.

Cuando una subclase redefine un método de su superclase, se dice que está sobrescribiendo el método.

```
//Definición de la superclase Animal
class Animal {
 public void hacerSonido() {
     System.out.println("El animal hace un sonido.");
 }
}

//Definición de la subclase Perro que hereda de Animal
class Perro extends Animal {
 @Override
 public void hacerSonido() {
     System.out.println("El perro ladra.");
 }
}

//Clase principal que contiene el método main
public class Main {
 public static void main(String[] args) {
     Animal miAnimal = new Animal();
     miAnimal.hacerSonido();  // Salida: El animal hace un sonido.

     Animal miPerro = new Perro();  // Polimorfismo: un objeto de tipo Perro asignado a
una referencia de tipo Animal
     miPerro.hacerSonido();  // Salida: El perro ladra. (a pesar de que la referencia es de
tipo Animal, se usa la implementacion de Perro)
  }
}
```

CLASES ABSTRACTAS

Las clases abstractas y las interfaces son conceptos importantes en la programación orientada a objetos, que permiten la definición de estructuras y comportamientos que pueden ser heredados por otras clases. Ambos conceptos tienen diferencias clave en su uso y aplicación.

1. **Clases Abstractas:**

- Una clase abstracta es una clase que no puede ser instanciada directamente, es decir, no se pueden crear objetos de una clase abstracta. Se utiliza principalmente como una superclase que proporciona una estructura base para sus subclases.
- Puede contener métodos concretos (es decir, métodos con implementación) y métodos abstractos (es decir, métodos sin implementación).
- Las subclases de una clase abstracta deben implementar todos los métodos abstractos definidos en la superclase o también deben ser declaradas como abstractas.

Ejemplo de clase abstracta en Java:

```
abstract class Figura {
   abstract void dibujar();
}

class Circulo extends Figura {
   void dibujar() {
      System.out.println("Dibujar un círculo");
   }
}
```

las clases abstractas se utilizan para proporcionar una estructura base y métodos comunes para sus subclases, mientras que las interfaces se utilizan para definir comportamientos comunes que pueden ser implementados por diferentes clases.

Las clases anidadas en Java son clases definidas dentro de otra clase, lo que significa que una clase puede estar dentro de otra clase. Las clases anidadas se utilizan para organizar y estructurar el código de manera más coherente, y proporcionan un nivel adicional de encapsulación.

Hay dos tipos principales de clases anidadas en Java: clases estáticas anidadas (clases estáticas internas) y clases internas no estáticas (clases internas).

Clase estática anidada:

```
public class ClaseExterna {
    private static int x = 10;
    static class ClaseAnidada {
        public void imprimir() {
            System.out.println("El valor de x es: " + x);
        }
    }
    public static void main(String[] args) {
        ClaseExterna.ClaseAnidada anidada = new ClaseExterna.ClaseAnidada();
        anidada.imprimir();
    }
}
```

Clase interna no estática:

```
public class ClaseExterna {
    private int y = 20;

    class ClaseAnidada {
        public void imprimir() {
            System.out.println("El valor de y es: " + y);
        }
    }

    public static void main(String[] args) {
        ClaseExterna externa = new ClaseExterna();
        ClaseExterna.ClaseAnidada anidada = externa.new ClaseAnidada();
        anidada.imprimir();
    }
}
```

ClaseAnidada es una clase interna no estática de la clase ClaseExterna. Para acceder a la clase interna, se utiliza externa.new ClaseAnidada(). La clase interna puede acceder a miembros no estáticos de la clase externa.

MÉTODOS Y CONSTRUCTORES

Los métodos y los constructores son elementos fundamentales en la programación orientada a objetos, y desempeñan roles importantes en la definición y manipulación de clases y objetos en Java.

Métodos:

Los métodos son bloques de código que realizan tareas específicas y pueden devolver un valor o simplemente ejecutar una secuencia de operaciones.

En Java, los métodos se definen dentro de una clase y pueden ser utilizados para encapsular la lógica, proporcionar funcionalidad específica y modularizar el código.

Un método puede tener parámetros (información que se pasa al método), un tipo de retorno (el tipo de dato que el método devuelve) y un cuerpo que contiene las operaciones que realiza el método.

Ejemplo de un método simple en Java:

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

Constructores:

- Un constructor es un tipo especial de método que se utiliza para inicializar un objeto. En Java, el constructor se llama automáticamente cuando se crea un objeto de una clase.
- Los constructores tienen el mismo nombre que la clase a la que pertenecen y no tienen un tipo de retorno explícito.
- Se pueden sobrecargar los constructores proporcionando diferentes conjuntos de parámetros, lo que permite la creación de objetos de formas diferentes.

Ejemplo de constructor en Java:

```
public class Coche {
    private String marca;
    private String modelo;

    public Coche(String marca, String modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }
}
```

EVALUACIÓN: Unidad 3 Funciones de Control y Operadores en Java

1. Constructor

¿Cuál es la función principal de un constructor en Java?

a) Iniciar objetos de una clase
b) Sobrescribir métodos
c) Definir interfaces
d) Crear subclases

2. ¿Qué es la herencia en Java?

La herencia en Java permite:

a) Crear objetos sin definir clases
b) Compartir comportamiento entre clases
c) Definir métodos abstractos
d) Anidar clases

3. Herencia, subclases

Las subclases en Java:

a) No pueden tener constructores
b) Solo pueden sobrescribir métodos estáticos
c) Extienden la funcionalidad de las superclases
d) No pueden tener interfaces

4. Sobreescritura de Métodos

La sobrescritura de métodos en Java se utiliza para:

a) Definir nuevas clases
b) Modificar el comportamiento de métodos heredados
c) Crear múltiples constructores
d) Establecer clases finales

5. Clases abstractas, interfaces

Las interfaces en Java:

a) Pueden tener métodos concretos
b) No pueden ser instanciadas
c) Deben ser siempre abstractas
d) No permiten herencia múltiple

6. Clases anidadas

¿Qué tipo de clase anidada puede acceder a miembros de la clase externa

incluso si son privados?

a) Clases estáticas
b) Clases abstractas
c) Clases internas
d) Clases finales

7. métodos y constructores

¿Qué sucede si un constructor no se define en una clase?

a) La clase no puede ser instanciada
b) Se utiliza un constructor predeterminado
c) La clase se convierte en abstracta
d) La clase no puede heredar

UNIDAD 4

OBJETOS Y CLASES EN JAVA

- Polimorfismos
- Exploración de herencia, sobreescritura de métodos, clases abstractas
- Análisis de clases anidadas y su aplicación con Eclipse
- Estudio de objetos y clases en Java, ámbito de variables,

POLIMORFISMO

el **polimorfismo** es uno de los pilares fundamentales de la Programación Orientada a Objetos (POO), junto con la herencia y el encapsulamiento. El polimorfismo permite que un objeto de una clase se comporte de diferentes maneras dependiendo del contexto. Aquí tienes una explicación más detallada:

Tipos de Polimorfismo en Java

1. **Polimorfismo en Tiempo de Compilación (Sobrecarga de Métodos)**:

- Ocurre cuando dos o más métodos en la misma clase tienen el mismo nombre, pero diferentes parámetros (tipo o número de parámetros).

Ejemplo:

```
class Calculadora {
    // Método para sumar dos enteros
    int sumar(int a, int b) {
        return a + b;
    }

    // Sobrecarga del método sumar para sumar tres enteros
    int sumar(int a, int b, int c) {
        return a + b + c;
    }

    // Sobrecarga del método sumar para sumar dos números de punto flotante
    double sumar(double a, double b) {
        return a + b;
    }
}

public class TestPolimorfismo {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();

        // Llamadas a los diferentes métodos sobrecargados
        System.out.println("Suma de dos enteros: " + calc.sumar(2, 3)); // 5
        System.out.println("Suma de tres enteros: " + calc.sumar(1, 2, 3)); // 6
        System.out.println("Suma de dos flotantes: " + calc.sumar(2.5, 3.5)); // 6.0
    }
}
```

En este ejemplo, la clase Calculadora tiene varios métodos sumar con diferentes parámetros, y el compilador selecciona el método adecuado basado en los argumentos pasados.

2. Polimorfismo en Tiempo de Ejecución (Sobreescritura de Métodos)

Descripción: Ocurre cuando una subclase redefine un método de su superclase con la misma firma (nombre y parámetros).

Ejemplo:

```
class Animal {
   void sonido() {
      System.out.println("El animal hace un sonido");
   }
}

class Perro extends Animal {
   @Override
   void sonido() {
      System.out.println("El perro ladra");
   }
}

public class TestPolimorfismo {
   public static void main(String[] args) {
      Animal miAnimal = new Perro();
      miAnimal.sonido(); // Output: "El perro ladra"
   }
}
```

En este ejemplo, el método sonido de la clase Perro sobrescribe el método sonido de la clase Animal. Cuando se llama al método sonido en una instancia de Perro, se ejecuta la versión sobrescrita

HERENCIA

La **herencia** es un concepto fundamental en la Programación Orientada a Objetos (POO). Permite crear nuevas clases basadas en clases existentes, promoviendo la reutilización del código y la creación de relaciones jerárquicas entre las clases.

Beneficios de la Herencia

1. **Reutilización del Código**: Las subclases heredan métodos y atributos de sus superclases, evitando duplicar código.
2. **Extensibilidad**: Puedes añadir nuevas características a una clase existente creando una subclase.
3. **Mantenimiento**: Las modificaciones en la superclase se reflejan automáticamente en las subclases, facilitando el mantenimiento.

Ejemplo de Herencia en Java

```java
// Superclase
class Animal {
    void comer() {
        System.out.println("El animal está comiendo");
    }
}

// Subclase
class Perro extends Animal {
    void ladrar() {
        System.out.println("El perro está ladrando");
    }
}

public class TestHerencia {
    public static void main(String[] args) {
        Perro miPerro = new Perro();
        miPerro.comer(); // Método heredado de Animal
        miPerro.ladrar(); // Método de la clase Perro
    }
}
```

En este ejemplo, la clase Perro hereda el método comer de la clase Animal y añade su propio método ladrar.

SOBREESCRITURA DE MÉTODOS

La **sobreescritura de métodos** (override) permite a una subclase proporcionar una implementación específica de un método que ya está definido en su superclase.

Ejemplo:

```
class Animal {
    void sonido() {
        System.out.println("El animal hace un sonido");
    }
}

class Gato extends Animal {
    @Override
    void sonido() {
        System.out.println("El gato maúlla");
    }
}

public class TestSobreescritura {
    public static void main(String[] args) {
        Animal miAnimal = new Gato();
        miAnimal.sonido(); // Output: "El gato maúlla"
    }
}
```

CLASES ABSTRACTAS

Las **clases abstractas** son clases que no pueden ser instanciadas y están destinadas a ser subclasificadas. Pueden contener métodos abstractos (sin implementación) y métodos concretos.

Métodos Abstractos

- **Definición**: Un método abstracto es un método que se declara, pero no se implementa en la clase abstracta. La implementación del método debe ser proporcionada por las subclases que heredan de la clase abstracta.
- **Sintaxis**: Un método abstracto se declara con la palabra clave abstract y no tiene un cuerpo de método.

Ejemplo:

```
abstract class Animal {
   // Método abstracto
   abstract void sonido();
}
```

Métodos Concretos

- **Definición**: Un método concreto es un método que tiene una implementación completa en la clase donde se define. Todas las clases pueden tener métodos concretos.
- **Sintaxis**: Un método concreto se declara con su firma y su cuerpo de método.

Ejemplo:

```
class Animal {
   // Método concreto
   void comer() {
      System.out.println("El animal está comiendo");
   }
}
```

Ejemplo Combinado:

```
abstract class Animal {
    // Método abstracto
    abstract void sonido();

    // Método concreto
    void comer() {
        System.out.println("El animal está comiendo");
    }
}

class Perro extends Animal {
    // Implementación del método abstracto
    @Override
    void sonido() {
        System.out.println("El perro ladra");
    }
}

public class TestMetodos {
    public static void main(String[] args) {
        Animal miPerro = new Perro();
        miPerro.sonido(); // Output: "El perro ladra"
        miPerro.comer();  // Output: "El animal está comiendo"
    }
}
```

En este ejemplo, Animal es una clase abstracta que tiene un método abstracto sonido y un método concreto comer. La clase Perro proporciona una implementación para el método abstracto sonido, pero también puede usar el método concreto comer tal como está.

CLASES ANIDADAS

Las **clases anidadas** en Java son clases definidas dentro de otra clase. Estas clases pueden ser útiles para agrupar clases que solo son utilizadas en un contexto específico, mejorando la legibilidad y mantenibilidad del código. Existen diferentes tipos de clases anidadas:

Tipos de Clases Anidadas

1. **Clases Internas**
    - Definidas dentro de otra clase y asociadas con una instancia de la clase externa.
    - Pueden acceder a los miembros de la clase externa, incluso si son privados.

```
class Externa {
    private String mensaje = "Hola";

    class Interna {
        void mostrarMensaje() {
            System.out.println(mensaje);
        }
    }
}
```

2. **Clases Internas Estáticas**
    - Definidas con la palabra clave static dentro de otra clase.
    - No pueden acceder a miembros de la clase externa directamente sin una instancia de la clase externa.

```
class Externa {
    static class InternaEstatica {
        void mostrarMensaje() {
            System.out.println("Mensaje desde la clase interna estática");
        }
    }
}
```

3. **Clases Anónimas**
    - No tienen un nombre específico y se declaran e instancian en una única expresión.
    - Útiles para crear instancias de clases con ligeras modificaciones.

```
class Externa {
    void metodoConClaseAnonima() {
        Animal miAnimal = new Animal() {
```

```
            void sonido() {
                System.out.println("Sonido del animal anónimo");
            }
        };
        miAnimal.sonido();
    }

}
```

4. **Clases Locales**
   - Definidas dentro de un bloque de código, como un método.
   - Solo accesibles dentro del bloque en el que se definen.

```
class Externa {
    void metodoConClaseLocal() {
        class InternaLocal {
            void mostrarMensaje() {
                System.out.println("Mensaje desde la clase local");
            }
        }
        InternaLocal internaLocal = new InternaLocal();
        internaLocal.mostrarMensaje();
    }

}
```

Ejemplo en Eclipse

Veamos un ejemplo práctico utilizando Eclipse:

```
// Clase externa
public class Externa {
    private String mensaje = "Hola desde la clase externa";

    // Clase interna
    class Interna {
        void mostrarMensaje() {
            System.out.println(mensaje); // Accediendo a miembro privado de la clase externa
        }
    }

    // Clase interna estática
    static class InternaEstatica {
```

```
        void mostrarMensajeEstatico() {
            System.out.println("Mensaje desde la clase interna estática");
        }
    }

    // Método para crear una instancia de la clase interna
    void usarClaseInterna() {
        Interna interna = new Interna();
        interna.mostrarMensaje();
    }
}

// Clase principal para probar las clases anidadas
public class TestClasesAnidadas {
    public static void main(String[] args) {
        Externa externa = new Externa();
        externa.usarClaseInterna(); // Uso de clase interna

        Externa.InternaEstatica internaEstatica = new Externa.InternaEstatica();
        internaEstatica.mostrarMensajeEstatico(); // Uso de clase interna estática
    }
}
```

Creación y Ejecución en Eclipse

1. **Crear un Proyecto**: Abre Eclipse, selecciona **File > New > Java Project** y dale un nombre a tu proyecto.
2. **Agregar Clases**: Crea dos clases, Externa y TestClasesAnidadas.
    - En Externa, define las clases anidadas y los métodos como se muestra en el ejemplo.
    - En TestClasesAnidadas, escribe el método main para probar las clases anidadas.
3. **Ejecutar el Código**: Haz clic derecho en TestClasesAnidadas.java y selecciona **Run As > Java Application** para ejecutar el programa y ver los resultados en la consola.
4.

ESTUDIO DE OBJETOS Y CLASES EN JAVA, ÁMBITO DE VARIABLES

Clases

Una clase en Java es una plantilla o modelo a partir del cual se crean objetos. Define atributos (variables) y métodos que los objetos creados a partir de la clase pueden usar

```
// Definición de una clase llamada 'Coche'
class Coche {
    // Atributos (variables)
    String marca;
    String modelo;
    int año;

    // Métodos
    void arrancar() {
        System.out.println("El coche está arrancando");
    }

    void detener() {
        System.out.println("El coche está deteniéndose");
    }
}
```

Objetos

Un objeto es una instancia de una clase. Cuando se crea un objeto, se asigna memoria para almacenar sus atributos y puede usar los métodos definidos en la clase.

Ejemplo:

```
public class TestCoche {
    public static void main(String[] args) {
        // Crear un objeto de la clase 'Coche'
        Coche miCoche = new Coche();

        // Asignar valores a los atributos
```

```
        miCoche.marca = "Toyota";
        miCoche.modelo = "Corolla";
        miCoche.año = 2021;

        // Llamar a los métodos
        miCoche.arrancar();
        miCoche.detener();
    }
}
```

Ámbito de Variables

1. Variables de Instancia

- **Definición**: Variables que se declaran dentro de una clase, pero fuera de cualquier método, constructor o bloque.
- **Ámbito**: Pertenecen a una instancia específica de la clase y se accede a ellas a través de los objetos.
- **Ejemplo**:

```
class Coche {
    String marca; // Variable de instancia
}
```

2. Variables de Clase (Estáticas)

- **Definición:** Variables que se declaran con la palabra clave static.
- **Ámbito:** Pertenecen a la clase y no a una instancia específica. Se pueden acceder directamente a través de la clase.
- Ejemplo:

```
class Coche {
    static int numeroDeCoches; // Variable de clase
}
```

3. Variables Locales

- **Definición**: Variables que se declaran dentro de un método, constructor o bloque.
- **Ámbito**: Solo son accesibles dentro del método, constructor o bloque en el que se declaran.
- **Ejemplo**:

```
class Coche {
    void acelerar(int velocidad) { // Parámetro 'velocidad'
        System.out.println("El coche está acelerando a " + velocidad + " km/h");
    }
}
```

4. Parámetros

- **Definición**: Variables que se declaran en la firma de un método.
- **Ámbito**: Solo son accesibles dentro del método donde se declaran.
- **Ejemplo**:

```
class Coche {
    void acelerar(int velocidad) { // Parámetro 'velocidad'
        System.out.println("El coche está acelerando a " + velocidad + " km/h");
    }
}
```

Estos conceptos son fundamentales para entender cómo se estructuran y funcionan los programas en Java

EVALUACIÓN UNIDAD 4:
Objetos y Clases en Java

1. Polimorfismos

¿Cuál de las siguientes opciones describe mejor el polimorfismo en tiempo de ejecución?

a) Sobrecarga de métodos en la misma clase.
b) Uso de interfaces en diferentes clases.
c) Sobreescritura de métodos en una subclase.
d) Uso de métodos estáticos en clases abstractas.

2. Exploración de herencia, sobreescritura de métodos, clases abstractas

¿Cuál de las siguientes afirmaciones es verdadera acerca de las clases abstractas en Java?

a) Las clases abstractas no pueden tener métodos concretos.
b) Las clases abstractas pueden ser instanciadas directamente.
c) Las clases abstractas deben ser declaradas con la palabra clave `abstract`.
d) Las clases abstractas no pueden tener variables de instancia.

3. Análisis de clases anidadas y su aplicación con Eclipse

¿Cuál es una característica de las clases internas en Java?

a) Solo pueden ser accedidas por métodos estáticos.
b) Pueden acceder a los miembros de la clase externa, incluso si son privados.
c) No pueden tener constructores propios.
d) Solo pueden ser definidas dentro de interfaces.

4**. Estudio de objetos y clases en Java, ámbito de variables**

¿Qué tipo de variable pertenece a una instancia específica de una clase?

a) Variable estática.
b) Variable de clase.
c) Variable local.
d) Variable de instancia.

5. **Polimorfismos**

¿Cuál es un ejemplo de polimorfismo en tiempo de compilación?

a) Sobreescritura de métodos.
b) Implementación de interfaces.
c) Sobrecarga de métodos.
d) Uso de clases abstractas.

6**. Exploración de herencia, sobreescritura de métodos, clases abstractas**

¿Qué permite la herencia en Java?

a) Crear clases sin definir sus métodos.
b) Definir métodos privados en subclases.
c) Compartir y extender la funcionalidad de una clase existente.
d) Eliminar la necesidad de constructores en clases derivadas.

7**. Análisis de clases anidadas y su aplicación con Eclipse**
¿Cuál de las siguientes afirmaciones es verdadera acerca de las clases internas estáticas?

a) Pueden acceder directamente a los miembros no estáticos de la clase externa.
b) No pueden tener métodos propios.
c) Pueden ser instanciadas sin una instancia de la clase externa.
d) Deben ser declaradas dentro de métodos.

8**. Estudio de objetos y clases en Java, ámbito de variables**
¿Cuál es el ámbito de una variable declarada dentro de un método?

a) Variable de instancia.
b) Variable estática.
c) Variable local.
d) Variable global.

REFERENCIAS BIBLIOGRÁFICAS

Bloch, J. (2018). Effective Java. Addison-Wesley Professional.

Eckel, B. (2006). Thinking in Java. Prentice Hall.

Flanagan, D. (2005). Java in a nutshell. O'Reilly Media.

Fowler, M. (1999). Refactoring: Improving the design of existing code. Addison-Wesley Professional.

Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2014). The Java language specification. Addison-Wesley Professional.

Herlihy, M., & Shavit, N. (2012). The art of multiprocessor programming. Morgan Kaufmann.

Horstmann, C. S., & Cornell, G. (2019). Core Java volume I: Fundamentals. Prentice Hall.

Horstmann, C. S., & Cornell, G. (2019). Core Java volume II: Advanced features. Prentice Hall.

Oracle. (n.d.). Java Platform, standard edition 8 documentation. Retrieved from https://docs.oracle.com/javase/8/docs/

Sierra, K., & Bates, B. (2005). Head first Java. O'Reilly Media