**Plan de trabajo autónomo lenguaje de**

**programación Java**

Plan de trabajo:

1. Tipos de datos y variables en java

*Tipos de datos*

* **byte**: Tamaño: 8 bits, Rango: -128 a 127, Uso: Útil para ahorrar memoria en grandes arreglos.
* **short**: Tamaño: 16 bits, Rango: -32,768 a 32,767, Uso: Similar al byte pero con un rango más amplio.
* **int**: Tamaño: 32 bits, Rango: -2^31 a 2^31-1 (aproximadamente -2.1 mil millones a 2.1 mil millones), Uso: Tipo de dato entero más común.
* **long**: Tamaño: 64 bits, Rango: -2^63 a 2^63-1 (aproximadamente -9.2 quintillones a 9.2 quintillones), Uso: Para valores enteros más grandes.
* **float**: Tamaño: 32 bits, Rango: Aproximadamente ±3.40282347E+38F (6-7 dígitos decimales de precisión), Uso: Para números en punto flotante con precisión simple.
* **double**: Tamaño: 64 bits, Rango: Aproximadamente ±1.79769313486231570E+308 (15-16 dígitos decimales de precisión), Uso: Para números en punto flotante con precisión doble.
* **char:** Tamaño: 16 bits, Rango: 0 a 65,535 (representa un único carácter Unicode), Uso: Para representar un solo carácter.
* **boolean:** Tamaño**:** No está claramente especificado (generalmente 1 bit en la práctica), Valores**:** true o falso, Uso**:** Para valores lógicos.

Aunque actualmente solo se utilizan int, double, char y boolean debido al espacio con el que vienen actualmente los dispositivos.

*Variables*

Son un espacio en el memoria el cual pude almacenar un dato y mediante el desarrollo del código se puede ver un cambio.

**int** edad = 30;

**double** altura = 1.75;

**char** inicial = 'A';

**boolean** estudiante = **true**;

1. Constantes en Java

En Java, las constantes se definen utilizando la palabra clave “final”. Una constante es una variable que se declara como “final”, lo que significa que su valor no puede ser modificado después de su inicialización.

Sintaxis:

“final TipoDeDato VARIABLE = valor;”

EJEMPLO:

**public** **class** ejercicio2 {

// Definición de constantes

**public** **static** **final** **int** ***EDAD\_MINIMA*** = 18;

**public** **static** **final** **double** ***PI*** = 3.14159;

**public** **static** **final** **char** ***INICIAL*** = 'A';

**public** **static** **final** **boolean** ***ES\_ACTIVO*** = **true**;

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

// imprimir constante

System.***out***.println("Edad mínima: " + ***EDAD\_MINIMA***);

System.***out***.println("Valor de PI: " + ***PI***);

System.***out***.println("Inicial: " + ***INICIAL***);

System.***out***.println("Es activo: " + ***ES\_ACTIVO***);

}

}

1. Estructuras de control en condicionales simples y dobles

Principalmente su función es ejecutar un bloque de código siempre y cuando la condición dicha se cumpla

Sintaxis:

“**if** (condition) {}”

EJEMPLO:

**public** **class** CondicionalesSimples {

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

**int** number = 10;

**if** (number > 0) {

System.***out***.println("el numero "+number+" es positivo");

}

}

}

**public** **class** CondicionalesElse{

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

**int** number = 10;

**if** (number > 0) {

System.***out***.println("el numero "+number+" es positivo");

} **else** {

System.***out***.println("el numero "+number+" es negativo");

}

}

}

**public** **class** CondicionalesDobles{

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

**int** number = 0;

**if** (number == 0) {

System.***out***.println("el numero "+number+" es '0'");

} **else** **if** (number > 0) {

System.***out***.println("el numero "+number+" es positivo");

} **else** {

System.***out***.println("el numero "+number+" es negativo");

}

}

}

1. Estructuras de control múltiples

las estructuras de control múltiples permiten manejar varias condiciones y casos, facilitando la toma de decisiones y el flujo del programa basado en diferentes criterios.

Sintaxis:

**“switch** () {

**case** 1:

**break**;

**case** n:

**break**;

**default**:

**break**;

}”

EJEMPLO:

**public** **class** EjemploSwitch {

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

**int** dia = 4;

**switch** (dia) {

**case** 1:

System.***out***.println("Lunes");

**break**;

**case** 2:

System.***out***.println("Martes");

**break**;

**case** 3:

System.***out***.println("Miércoles");

**break**;

**case** 4:

System.***out***.println("Jueves");

**break**;

**case** 5:

System.***out***.println("Viernes");

**break**;

**case** 6:

System.***out***.println("Sábado");

**break**;

**case** 7:

System.***out***.println("Domingo");

**break**;

**default**:

System.***out***.println("Número de día inválido");

**break**;

}

}

}

Una forma más corta de hacerlo sería la siguiente:

**public** **class** EjemploSwitchDia {

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

**int** dia = 3;

String nombreDia = **switch** (dia) {

**case** 1 -> "Lunes";

**case** 2 -> "Martes";

**case** 3 -> "Miércoles";

**case** 4 -> "Jueves";

**case** 5 -> "Viernes";

**case** 6 -> "Sábado";

**case** 7 -> "Domingo";

**default** -> "Número de día inválido";

}

System.***out***.println(nombreDia);

}

}

**public** **class** EjemploSwitchColores {

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

**int** color = 3;

String nombreDelColor = **switch** (color) {

**case** 1 -> "azul";

**case** 2 -> "amarillo";

**case** 3 -> "verde";

**case** 4 -> "rosado";

**case** 5 -> "gris";

**case** 6 -> "negro";

**case** 7 -> "blanco";

**default** -> "Número de color inválido";

}

System.***out***.println(nombreDelColor);

}

}

1. Estructuras iterativas While

La estructura de control “while” en Java es un bucle que ejecuta un bloque de código mientras se cumple una condición específica.

Sintaxis:

“**while** (condition) {}”

EJEMPLO:

**public** **class** EjemploWhile {

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

**int** i = 1;

**while** (i <= 5) {

System.***out***.println(i);

i++;

}

}

}

**public** **class** SumaAcumulativa {

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

**int** suma = 0;

**int** numero = 1;

**while** (suma <= 100) {

suma += numero;

numero++;

}

System.***out***.println("La suma acumulativa es: " + suma);

}

}

**public** **class** ControlWhileTrueAndFalse {

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

**int** i = 0;

**while** (**true**) {

i++;

**if** (i == 5) {

**continue**;

}

**if** (i > 10) {

**break**;

}

System.***out***.println(i);

}

}

}

1. Estructuras iterativas do while

la estructura iterativa “do-while” es un tipo de bucle que garantiza que el bloque de código dentro del bucle se ejecute al menos una vez antes de verificar la condición para continuar iterando.

Sintaxis:

“**do** {

} **while** (condición);”

EJEMPLO:

**public** **class** doWhile{

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

**int** i = 1;

**do** {

System.***out***.println(i);

i++;

} **while** (i <= 5);

}

}

**import** javax.swing.JOptionPane;

**public** **class** doWhileContrasena{

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

String contraseña;

String contraseñaCorrecta = "secreta";

**do** {

contraseña = JOptionPane.*showInputDialog*("Introduce la contraseña: ");

**if** (!contraseña.equals(contraseñaCorrecta)) {

System.***out***.println("Contraseña incorrecta. Inténtalo de nuevo.");

}

} **while** (!contraseña.equals(contraseñaCorrecta));

System.***out***.println("Contraseña correcta. Acceso concedido.");

}

}

**import** java.util.Random;

**import** javax.swing.JOptionPane;

**public** **class** juegoDoWhile {

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

Random random = **new** Random();

**int** user = 0;

**int** cpu = 0;

**do** {

System.***out***.println("Menu : \n 1. Piedra \n 2. Papel \n 3. Tijeras");

**int** eleccion = Integer.*parseInt*(JOptionPane.*showInputDialog*("Elige una opción (1, 2, 3):"));

**if** (eleccion < 1 || eleccion > 3) {

System.***out***.println("Opción no válida. Elige 1, 2 o 3.");

**continue**;

}

**int** randomvalor = random.nextInt(3) + 1;

System.***out***.println("La CPU eligió: " + randomvalor);

**if** ((eleccion == 1 && randomvalor == 3) ||

(eleccion == 2 && randomvalor == 1) ||

(eleccion == 3 && randomvalor == 2)) {

user++;

System.***out***.println("Gana el usuario");

} **else** **if** (eleccion == randomvalor) {

System.***out***.println("Empate");

} **else** {

cpu++;

System.***out***.println("Gana la CPU");

}

System.***out***.println("Usuario: " + user + " - CPU: " + cpu);

**if** (user == 3 || cpu == 3) {

**break**;

}

} **while** (**true**);

**if** (user == 3) {

System.***out***.println("¡El usuario gana!");

} **else** **if** (cpu == 3) {

System.***out***.println("¡La CPU gana!");

}

}

}

1. Estructuras iterativas for

las estructuras iterativas for en Java son una herramienta fundamental para realizar tareas repetitivas de manera eficiente. El bucle for es muy útil cuando conoces de antemano el número de iteraciones que necesitas realizar

Sintaxis:

“for (inicialización; condición; actualización) { }”

EJEMPLOS:

**public** **class** EjemploFor {

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

**int** numero = 5; // Número para la tabla de multiplicar

**for** (**int** i = 1; i <= 10; i++) {

System.***out***.println(numero + " x " + i + " = " + (numero \* i));

}

}

}

**public** **class** EjemploForIf{

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

**for** (**int** i = 1; i <= 10; i++) {

**if** (i == 5) {

**continue**;

}

**if** (i == 8) {

**break**;

}

System.***out***.println(i);

}

}

}

**public** **class** EjemploForAnidados {

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

**for** (**int** i = 1; i <= 3; i++) {

**for** (**int** j = 1; j <= 3; j++) {

System.***out***.println("i = " + i + ", j = " + j);

}

}

}

}

1. Concepto Estructuras de Datos

Una estructura de datos es una forma de organizar y almacenar datos en una computadora para que se pueda acceder y modificar de manera eficiente. Las estructuras de datos son cruciales para el desarrollo de algoritmos eficientes y la gestión efectiva de la memoria.

En Java, las estructuras de datos se implementan a menudo utilizando clases e interfaces que forman parte de la biblioteca estándar. La elección de una estructura de datos adecuada puede mejorar significativamente el rendimiento y la legibilidad del código.

1. Estructuras de datos – Arreglos

*Tamaño Fijo:* Una vez que se crea un arreglo, su tamaño no puede cambiar. Esto significa que el número de elementos en el arreglo es fijo y debe ser especificado al momento de la creación.

*Acceso Rápido:* Los arreglos permiten acceso rápido a los elementos mediante índices, ya que el índice se utiliza para calcular la ubicación de un elemento en la memoria.

*Homogeneidad:* Todos los elementos de un arreglo deben ser del mismo tipo de datos. Si se requiere almacenar diferentes tipos de datos, se puede utilizar un arreglo de objetos.

Sintaxis:

“**i**nt[] numeros = **new** **int**[5];”

EJEMPLO:

**int**[ ] numeros = {1, 2, 3, 4, 5};

String[] colores = {"azul", "amarillo", "verde", "rojo"};

1. Ejercicio Suma Arreglos en Java

**package** ejercicio10;

**public** **class** pricipal {

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

**int**[ ] arreglo1 = {1, 2, 3, 4, 5, 1};

**int**[ ] arreglo2 = {10, 20, 30, 40, 50, 1};

**int**[ ] arregloSuma = **new** **int**[arreglo1.length];

**for** (**int** i = 0; i < arreglo1.length; i++) {

arregloSuma[ i ] = arreglo1[ i ] + arreglo2[i];

}

System.***out***.println("Arreglo 1:");

*imprimirArreglo*(arreglo1);

System.***out***.println("Arreglo 2:");

*imprimirArreglo*(arreglo2);

System.***out***.println("Arreglo de la suma:");

*imprimirArreglo*(arregloSuma);

}

**public** **static** **void** imprimirArreglo(**int**[] arreglo) {

**for** (**int** elemento : arreglo) {

System.***out***.print(elemento + " ");

}

System.***out***.println();

}

}

1. Estructuras de datos – ArrayList

El “ArrayList” es una de las colecciones más utilizadas en Java, perteneciente al marco de colecciones de Java (java.util). Ofrece una implementación dinámica de listas que permite un crecimiento automático, a diferencia de los arreglos tradicionales, cuyo tamaño es fijo una vez creado.

Redimensionamiento Dinámico: A diferencia de los arreglos, ArrayList puede crecer o reducirse automáticamente según las necesidades.

Acceso Rápido por Índice: Permite acceder a los elementos por índice de manera eficiente.

Admite Elementos Duplicados: Puedes almacenar múltiples instancias del mismo elemento en un ArrayList.

Orden de Inserción: Los elementos se mantienen en el orden en que se insertaron.

Sintaxis:

“ArrayList<String> listaDeCadenas = **new** ArrayList<>();

ArrayList<Integer> listaDeEnteros = **new** ArrayList<>();”

EJEMPLO:

**package** ejrcicio2;

**import** java.util.ArrayList;

**public** **class** pricipal {

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

ArrayList<String> listaDeCadenas = **new** ArrayList<>();

ArrayList<Integer> listaDeEnteros = **new** ArrayList<>();

listaDeCadenas.add("Hola");

listaDeCadenas.add("Mundo");

listaDeEnteros.add(1);

listaDeEnteros.add(2);

String saludo = listaDeCadenas.get(0);

**int** numero = listaDeEnteros.get(1);

System.***out***.println(saludo + numero);

}

}

Para cambiar un dato del array por otro en este caso “mundo” por “Java”:

***“listaDeCadenas.set(1, "Java");”***

para eliminar un dato del array:

***“listaDeCadenas.remove(0);”***

1. Estructuras de datos – HashTable

La Hashtable es una implementación de la interfaz Map en Java que proporciona una estructura de datos basada en una tabla hash. Se utiliza para almacenar pares clave-valor y permite acceder a los valores de manera eficiente utilizando la clave. A diferencia de HashMap, que es otra implementación de Map, Hashtable es sincronizada y, por lo tanto, es segura para el uso concurrente en entornos multihilo.

Sincronización: Hashtable es sincronizada, lo que significa que es segura para su uso en entornos multihilo sin necesidad de sincronización adicional.

No Permite Claves o Valores Nulos: No se permiten claves o valores nulos. Intentar insertar una clave o valor nulo resultará en una excepción NullPointerException.

Orden de Iteración: No garantiza el orden de los elementos, ya que la ordenación depende de la implementación de la función hash.

Acceso Rápido: Permite acceso rápido a los elementos utilizando las claves, gracias a su estructura basada en una tabla hash.

Sintaxis:

“Hashtable<String, Integer> tabla = **new** Hashtable<>();”

EJEMPLO:

**import** java.util.Hashtable;

**public** **class** principal {

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

**//para crear la tabla**

Hashtable<String, Integer> tabla = **new** Hashtable<>();

//agregar valores a la tabla

tabla.put("Uno", 1);

tabla.put("Dos", 2);

tabla.put("Tres", 3);

//para modificar un valor

**int** valor = tabla.get("Dos");

tabla.put("Dos", 22);

//para eliminar un valor

tabla.remove("Tres");

System.***out***.println(tabla);

System.***out***.println(valor);

}

}

1. Estructuras de datos – HashMap

El HashMap es una implementación de la interfaz Map en Java que proporciona una estructura de datos basada en una tabla hash. Se utiliza para almacenar pares clave-valor y permite acceder a los valores de manera eficiente utilizando la clave. A diferencia de Hashtable, HashMap no es sincronizado, lo que significa que no es seguro para el uso concurrente en entornos multihilo sin sincronización adicional. Sin embargo, ofrece mejor rendimiento en entornos de un solo hilo.

Sincronización: HashMap no es sincronizado, por lo que no es seguro para el uso concurrente en entornos multihilo sin sincronización externa. Para hacer que un HashMap sea seguro en un entorno multihilo, se puede usar Collections.synchronizedMap() o ConcurrentHashMap.

Permite Claves y Valores Nulos: A diferencia de Hashtable, HashMap permite una clave nula y múltiples valores nulos.

Orden de Iteración: HashMap no garantiza el orden de los elementos, ya que la ordenación depende de la implementación de la función hash.

Acceso Rápido: Permite acceso rápido a los elementos utilizando las claves, gracias a su estructura basada en una tabla hash.

Sintaxis:

“HashMap<String, Integer> mapa = new HashMap<>();”

EJEMPLO:

**import** java.util.HashMap;

**public** **class** Principal {

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

// Crear un HashMap

HashMap<String, Integer> mapa = **new** HashMap<>();

// para agregar valores al HashMap

mapa.put("Uno", 1);

mapa.put("Dos", 2);

mapa.put("Tres", 3);

//para modificar un valor asociado por la clave

**int** valor = mapa.get("Dos");

mapa.put("Dos", 22);

// para eliminar un valor

mapa.remove("Tres");

// para imprimir el HashMap y el valor modificado

System.***out***.println(mapa);

System.***out***.println("Valor modificado: " + valor);

}

}