**Unidad 4 – Arreglos, Cadenas y Estructuras dinámicas** Manejo de Estructuras Secuenciales de Datos. Arreglos unidimensionales y multidimensionales. Operaciones con Arreglos: Recorrido, búsqueda, inserción. Cadena de Caracteres. Operaciones con cadenas. Concepto de Estructuras. Procedimientos de Ordenación y Búsqueda Binaria. Estructuras Dinámicas: Listas. Pilas y Colas. Árboles binarios. Operaciones con estructuras dinámicas: Recorrido, búsqueda, inserción. Árboles binarios. Archivos.

title:   
style: nestedList # TOC style (nestedList|inlineFirstLevel)  
minLevel: 0 # Include headings from the specified level  
maxLevel: 0 # Include headings up to the specified level  
includeLinks: true # Make headings clickable  
debugInConsole: false # Print debug info in Obsidian console

# Teoría

## Estructuras de datos

¿Por que es necesario? - Para representar objetos que no son atómicos - Para facilitar la manipulación de los mismos - Para almacenar colecciones de objetos

### Registros

Los registros son estructuras de datos que permiten combinar diferentes tipos de datos relacionados bajo un solo nombre. Cada componente de un registro se conoce como campo o atributo, y puede ser de cualquier tipo de datos, incluso otro registro. Los registros son útiles para modelar entidades complejas en un programa, como usuarios, productos, etc.

// Definición de una clase para representar un registro de estudiante  
class Estudiante {  
 // Campos o atributos del registro  
 String nombre;  
 int edad;  
 String carrera;  
  
 // Constructor  
 public Estudiante(String nombre, int edad, String carrera) {  
 this.nombre = nombre;  
 this.edad = edad;  
 this.carrera = carrera;  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 // Crear una instancia de Estudiante (registro)  
 Estudiante estudiante1 = new Estudiante("Juan", 20, "Ingeniería Informática");  
  
 // Acceder a los campos del registro  
 System.out.println("Nombre: " + estudiante1.nombre);  
 System.out.println("Edad: " + estudiante1.edad);  
 System.out.println("Carrera: " + estudiante1.carrera);  
 }  
}

### Archivos

Una colección de registros se denomina archivo

### Arreglos

Son colecciones de elementos del mismo tipo, almacenados de manera contigua en la memoria. Los arreglos ofrecen acceso aleatorio a sus elementos a través de un índice entero. Sin embargo, su tamaño suele ser fijo y puede ser ineficiente en términos de inserción y eliminación de elementos en posiciones intermedias.

**Problemas**: - Tienen una capacidad de almacenamiento determinada y no modificable - Una solución sería utilizar arreglos “lo suficientemente grandes”, desperdiciando espacio de memoria - La inserción de un objeto al principio del arreglo, sin sobrescribir el primer espacio, se torna mas complicada, pues debemos correr en un espacio a la derecha a todo el resto de los datos - Es decir, los arreglos no manejan los datos de forma dinámica.

#### Búsqueda binaria

La búsqueda binaria es un algoritmo eficiente para buscar un elemento en una lista ordenada. Funciona dividiendo repetidamente a la mitad la porción de la lista donde podría encontrarse el elemento deseado y luego descartando la mitad en la que el elemento no puede estar, hasta que se encuentra el elemento buscado o se determina que no está presente en la lista.

**Principio de Funcionamiento:** 1. **Ordenamiento:** Antes de aplicar la búsqueda binaria, la lista debe estar ordenada en orden ascendente o descendente para garantizar que el algoritmo funcione correctamente. 2. **División:** Se compara el elemento buscado con el elemento central de la lista. Si son iguales, se ha encontrado el elemento. Si el elemento buscado es menor que el elemento central, la búsqueda continúa en la mitad inferior de la lista; de lo contrario, en la mitad superior. 3. **Recursión o Iteración:** El proceso se repite en la mitad seleccionada de la lista hasta que se encuentre el elemento buscado o hasta que la porción de la lista seleccionada sea vacía, lo que indica que el elemento no está presente en la lista.

**Ventajas:** - Eficiencia: La búsqueda binaria tiene una complejidad temporal de O(log n), lo que significa que su tiempo de ejecución aumenta de manera muy lenta con el tamaño de la lista. - Eficacia: Es especialmente útil para buscar en grandes conjuntos de datos donde el tiempo de búsqueda es crítico.

**Desventajas:** - Requisito de Ordenamiento: Antes de aplicar la búsqueda binaria, la lista debe estar ordenada, lo que puede ser costoso en términos de tiempo y recursos si la lista cambia frecuentemente. - Limitación a Listas Ordenadas: Solo se puede aplicar a listas que estén previamente ordenadas.

**Implementación (Java):**

public static int binarySearch(int[] arr, int key) {  
 int left = 0;  
 int right = arr.length - 1;  
   
 while (left <= right) {  
 int mid = left + (right - left) / 2;  
   
 // Si el elemento buscado está en la mitad  
 if (arr[mid] == key) return mid;  
   
 // Si el elemento buscado es menor, buscar en la mitad inferior  
 if (arr[mid] < key) left = mid + 1;  
   
 // Si el elemento buscado es mayor, buscar en la mitad superior  
 else right = mid - 1;  
 }  
   
 // Si el elemento no está presente en la lista  
 return -1;  
}

#### Técnicas de Ordenamiento

##### Bubble Sort

**Principio de Funcionamiento:** 1. **Comparación de Elementos:** El algoritmo compara cada par de elementos adyacentes en la lista y los intercambia si están en el orden incorrecto. 2. **Repetición de Pasos:** Este proceso se repite varias veces hasta que la lista esté completamente ordenada y no se realicen intercambios en un ciclo completo de pasadas.

**Ventajas:** - Simple implementación: Es fácil de entender e implementar, lo que lo hace útil para fines educativos y pequeñas listas. - Espacio de Memoria: Requiere un espacio de memoria constante, ya que no utiliza estructuras de datos adicionales.

**Desventajas:** - Ineficiencia: Tiene una complejidad temporal de O(n^2), lo que lo hace ineficiente para listas grandes. - Rendimiento: No es recomendado para conjuntos de datos grandes debido a su baja eficiencia en comparación con otros algoritmos de ordenamiento.

procedure bubbleSort(arr: array)  
 n = length(arr)  
 for i from 0 to n-1 do  
 swapped = false // Indica si se realizó algún intercambio en esta pasada  
 for j from 0 to n-1-i do  
 if arr[j] > arr[j+1] then  
 // Intercambia los elementos si están en el orden incorrecto  
 swap(arr[j], arr[j+1])  
 swapped = true  
 end if  
 end for  
 // Si no se realizó ningún intercambio, la lista está ordenada  
 if not swapped then  
 break  
 end if  
 end for  
end procedure

##### Merge Sort

**Principio de Funcionamiento:** 1. **Divide y Conquista:** Divide la lista en mitades recursivamente hasta que cada mitad tenga un solo elemento. 2. **Ordenación y Fusión:** Fusiona las mitades de manera ordenada, combinando gradualmente sub-listas más grandes hasta que la lista original esté ordenada por completo.

**Ventajas:** - Eficiencia: Tiene una complejidad temporal de O(n log n), lo que lo hace eficiente incluso para listas muy grandes. - Estabilidad: Es un algoritmo estable, lo que significa que no cambia el orden relativo de elementos con valores iguales.

**Desventajas:** - Espacio de Memoria: Requiere espacio adicional para almacenar las sublistas durante la fusión, lo que puede ser costoso en términos de memoria para conjuntos de datos grandes.

procedure mergeSort(arr: array)  
 n = length(arr)  
 if n > 1 then  
 mid = n / 2  
 // Dividir la lista en dos mitades  
 left = arr[0:mid-1]  
 right = arr[mid:n-1]  
 // Llamadas recursivas a mergeSort para ordenar las dos mitades  
 mergeSort(left)  
 mergeSort(right)  
 // Fusionar las mitades ordenadas  
 merge(left, right, arr)  
 end if  
end procedure  
  
procedure merge(left: array, right: array, arr: array)  
 i = j = k = 0  
 // Comparar elementos de las dos mitades y fusionar en orden en la lista original  
 while i < length(left) and j < length(right) do  
 if left[i] <= right[j] then  
 arr[k] = left[i]  
 i++  
 else  
 arr[k] = right[j]  
 j++  
 end if  
 k++  
 end while  
 // Agregar los elementos restantes de left, si los hay  
 while i < length(left) do  
 arr[k] = left[i]  
 i++  
 k++  
 end while  
 // Agregar los elementos restantes de right, si los hay  
 while j < length(right) do  
 arr[k] = right[j]  
 j++  
 k++  
 end while  
end procedure

##### Quick Sort

**Principio de Funcionamiento:** 1. **Pivote y Partición:** Selecciona un elemento como pivote y particiona la lista en dos sub-listas, una con elementos menores que el pivote y otra con elementos mayores. 2. **Ordenación Recursiva:** Aplica el mismo proceso de manera recursiva a las sub-listas generadas hasta que toda la lista esté ordenada.

**Ventajas:** - Eficiencia: Tiene una complejidad temporal promedio de O(n log n), lo que lo convierte en uno de los algoritmos de ordenamiento más rápidos en la práctica. - In-Situ: No requiere espacio adicional para almacenar sub-listas, lo que lo hace eficiente en términos de memoria.

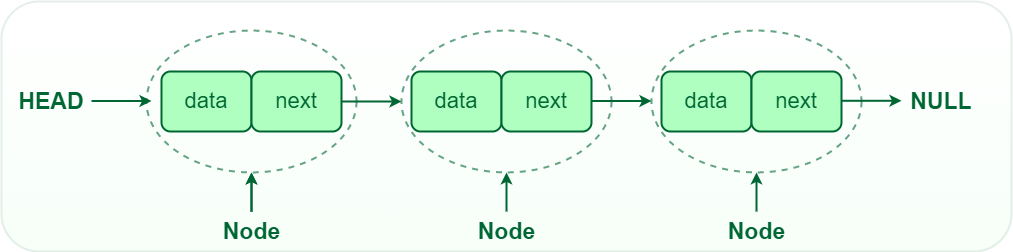
**Desventajas:** - Peor Caso: Puede tener una complejidad temporal de O(n^2) en el peor caso si la selección del pivote no está bien optimizada, aunque esto es raro en la práctica. - No Estable: No garantiza la estabilidad del orden de los elementos con valores iguales.

procedure quickSort(arr: array, low: int, high: int)  
 if low < high then  
 // Particionar el arreglo y obtener el índice del pivote  
 pivotIndex = partition(arr, low, high)  
 // Llamadas recursivas a quickSort para las sub-listas izquierda y derecha del pivote  
 quickSort(arr, low, pivotIndex - 1)  
 quickSort(arr, pivotIndex + 1, high)  
 end if  
end procedure  
  
function partition(arr: array, low: int, high: int) -> int  
 pivot = arr[high] // Seleccionar el último elemento como pivote  
 i = low - 1 // Índice del elemento más pequeño  
 for j from low to high-1 do  
 if arr[j] <= pivot then  
 // Intercambiar arr[i] y arr[j]  
 swap(arr[i+1], arr[j])  
 i++  
 end if  
 end for  
 // Intercambiar arr[i+1] y arr[high] (el pivote)  
 swap(arr[i+1], arr[high])  
 return i + 1 // Retorna el índice del pivote  
end function

### Listas Enlazadas

*Solución al problema de los arreglos* Las listas enlazadas son estructuras de datos dinámicas que consisten en nodos donde cada nodo contiene un valor y una referencia al siguiente nodo en la secuencia. Esto permite una inserción y eliminación eficiente de elementos en cualquier posición de la lista. Sin embargo, el acceso aleatorio a los elementos puede ser más lento en comparación con los arreglos.

* **puntero/apuntado:** es un espacio en la memoria que almacena una referencia o dirección de memoria. El valor de un puntero puede ser null, lo que significa actualmente no se refiere a ningún apuntado.
* **referencia:** la operación de referencia en un puntero sirve para tener acceso a su apuntado.
* **Asignación de puntero:** una operación de asignación entre dos punteros, como p=q, hace que los dos punteros se refieran (o apunten) al mismo apuntado. No se copia dos veces en la memoria al apuntado, sino que los dos punteros almacenan la dirección de memoria del apuntado.

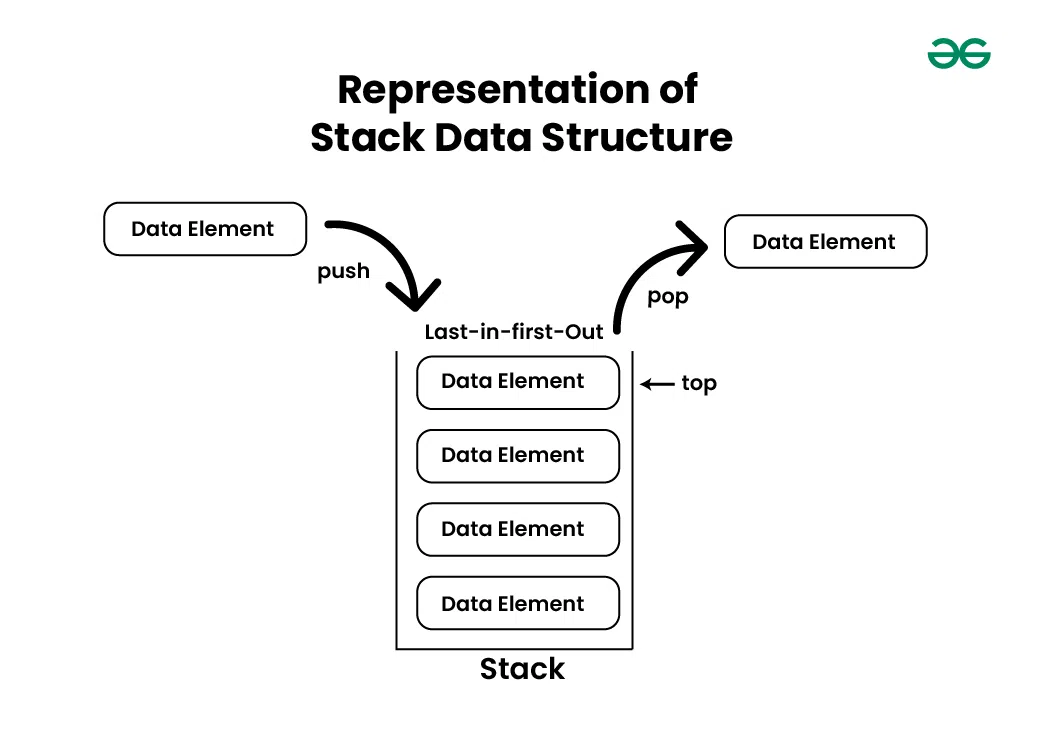


Pasted image 20240508191225.png

// Definición de la clase Nodo para representar los nodos de la lista enlazada  
class Nodo {  
 int dato;  
 Nodo siguiente;  
  
 // Constructor  
 public Nodo(int dato) {  
 this.dato = dato;  
 this.siguiente = null;  
 }  
}  
  
// Definición de la clase ListaEnlazada  
public class ListaEnlazada {  
 Nodo cabeza;  
  
 // Constructor  
 public ListaEnlazada() {  
 this.cabeza = null;  
 }  
  
 // Método para insertar un nuevo nodo al principio de la lista  
 public void insertarAlInicio(int dato) {  
 Nodo nuevoNodo = new Nodo(dato);  
 nuevoNodo.siguiente = cabeza;  
 cabeza = nuevoNodo;  
 }  
  
 // Método para imprimir los elementos de la lista  
 public void imprimirLista() {  
 Nodo temp = cabeza;  
 System.out.print("Lista enlazada: ");  
 while (temp != null) {  
 System.out.print(temp.dato + " ");  
 temp = temp.siguiente;  
 }  
 System.out.println();  
 }  
  
 public static void main(String[] args) {  
 // Crear una instancia de ListaEnlazada  
 ListaEnlazada lista = new ListaEnlazada();  
  
 // Insertar elementos en la lista  
 lista.insertarAlInicio(3);  
 lista.insertarAlInicio(7);  
 lista.insertarAlInicio(9);  
 lista.insertarAlInicio(5);  
  
 // Imprimir la lista  
 lista.imprimirLista();  
 }  
}

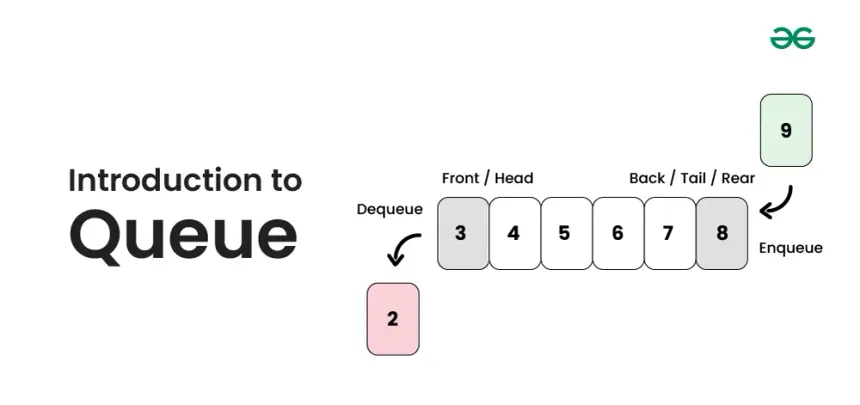
### Pilas (Stacks)

Una pila es una estructura de datos que sigue el principio de “último en entrar, primero en salir” (LIFO). Los elementos se agregan y eliminan solo desde un extremo, llamado la cima (top) de la pila. Esto hace que las operaciones de inserción y eliminación sean eficientes, pero el acceso a elementos en posiciones intermedias no está permitido.



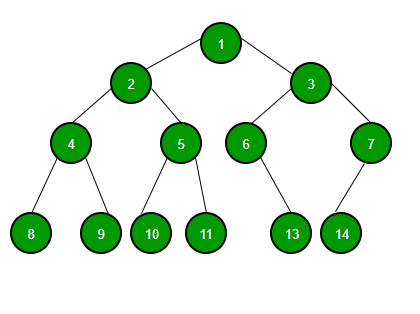
Pasted image 20240508191312.png

### Colas (Queues)

Una cola es una estructura de datos que sigue el principio de “primero en entrar, primero en salir” (FIFO). Los elementos se agregan al final de la cola y se eliminan desde el principio. Esto es útil en situaciones donde se necesita procesar elementos en el mismo orden en que llegaron. 

### Árbol Binario

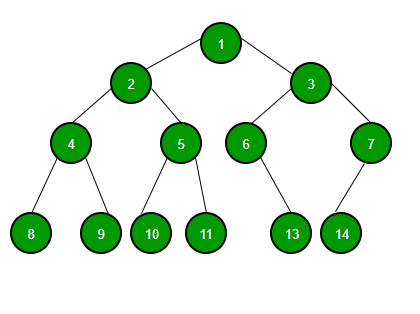
Es una estructura de datos jerárquica donde cada nodo tiene como máximo dos hijos: uno izquierdo y uno derecho. Esta estructura permite una búsqueda eficiente, inserción y eliminación de datos. Los árboles binarios son la base de muchas otras estructuras de datos más complejas, como los árboles binarios de búsqueda.



Pasted image 20240508192840.png

// Definición de la clase Nodo para representar los nodos del árbol binario  
class Nodo {  
 int dato;  
 Nodo izquierdo, derecho;  
  
 // Constructor  
 public Nodo(int dato) {  
 this.dato = dato;  
 this.izquierdo = null;  
 this.derecho = null;  
 }  
}  
  
// Definición de la clase ArbolBinario  
public class ArbolBinario {  
 Nodo raiz;  
  
 // Constructor  
 public ArbolBinario() {  
 raiz = null;  
 }  
  
 // Método para insertar un nuevo nodo en el árbol binario  
 public void insertar(int dato) {  
 raiz = insertarRec(raiz, dato);  
 }  
  
 // Método auxiliar recursivo para insertar un nuevo nodo en el árbol  
 private Nodo insertarRec(Nodo raiz, int dato) {  
 // Si el árbol está vacío, crea un nuevo nodo y lo devuelve  
 if (raiz == null) {  
 raiz = new Nodo(dato);  
 return raiz;  
 }  
  
 // Si el dato es menor que el valor del nodo actual, insertarlo en el subárbol izquierdo  
 if (dato < raiz.dato)  
 raiz.izquierdo = insertarRec(raiz.izquierdo, dato);  
 // Si el dato es mayor que el valor del nodo actual, insertarlo en el subárbol derecho  
 else if (dato > raiz.dato)  
 raiz.derecho = insertarRec(raiz.derecho, dato);  
  
 // Devolver el puntero de la raíz del árbol  
 return raiz;  
 }  
  
 // Método para realizar un recorrido inorden (izquierda, raíz, derecha)  
 public void inorden() {  
 inordenRec(raiz);  
 }  
  
 // Método auxiliar recursivo para recorrido inorden  
 private void inordenRec(Nodo raiz) {  
 if (raiz != null) {  
 inordenRec(raiz.izquierdo);  
 System.out.print(raiz.dato + " ");  
 inordenRec(raiz.derecho);  
 }  
 }  
  
 public static void main(String[] args) {  
 // Crear una instancia de ArbolBinario  
 ArbolBinario arbol = new ArbolBinario();  
  
 // Insertar nodos en el árbol  
 arbol.insertar(50);  
 arbol.insertar(30);  
 arbol.insertar(70);  
 arbol.insertar(20);  
 arbol.insertar(40);  
 arbol.insertar(60);  
 arbol.insertar(80);  
  
 // Imprimir el árbol binario en orden  
 System.out.println("Recorrido Inorden del Árbol Binario:");  
 arbol.inorden();  
 }  
}

### Árbol Binario de Búsqueda

Es un tipo especial de árbol binario donde los elementos están organizados de manera que los elementos menores que el nodo raíz se encuentran en el subárbol izquierdo, y los elementos mayores se encuentran en el subárbol derecho. Esto permite realizar búsquedas, inserciones y eliminaciones de manera eficiente, con un tiempo de complejidad logarítmico en el peor de los casos. 

// Definición de la clase Nodo para representar los nodos del árbol binario de búsqueda (ABB)  
class Nodo {  
 int dato;  
 Nodo izquierdo, derecho;  
  
 // Constructor  
 public Nodo(int dato) {  
 this.dato = dato;  
 this.izquierdo = null;  
 this.derecho = null;  
 }  
}  
  
// Definición de la clase ArbolBinarioBusqueda  
public class ArbolBinarioBusqueda {  
 Nodo raiz;  
  
 // Constructor  
 public ArbolBinarioBusqueda() {  
 raiz = null;  
 }  
  
 // Método para insertar un nuevo nodo en el árbol binario de búsqueda  
 public void insertar(int dato) {  
 raiz = insertarRec(raiz, dato);  
 }  
  
 // Método auxiliar recursivo para insertar un nuevo nodo en el árbol  
 private Nodo insertarRec(Nodo raiz, int dato) {  
 // Si la raíz es nula, crea un nuevo nodo y lo devuelve  
 if (raiz == null) {  
 raiz = new Nodo(dato);  
 return raiz;  
 }  
  
 // Si el dato es menor que el valor del nodo actual, insertarlo en el subárbol izquierdo  
 if (dato < raiz.dato)  
 raiz.izquierdo = insertarRec(raiz.izquierdo, dato);  
 // Si el dato es mayor que el valor del nodo actual, insertarlo en el subárbol derecho  
 else if (dato > raiz.dato)  
 raiz.derecho = insertarRec(raiz.derecho, dato);  
  
 // Devolver el puntero de la raíz del árbol  
 return raiz;  
 }  
  
 // Método para realizar un recorrido inorden (izquierda, raíz, derecha)  
 public void inorden() {  
 inordenRec(raiz);  
 }  
  
 // Método auxiliar recursivo para recorrido inorden  
 private void inordenRec(Nodo raiz) {  
 if (raiz != null) {  
 inordenRec(raiz.izquierdo);  
 System.out.print(raiz.dato + " ");  
 inordenRec(raiz.derecho);  
 }  
 }  
  
 public static void main(String[] args) {  
 // Crear una instancia de ArbolBinarioBusqueda  
 ArbolBinarioBusqueda abb = new ArbolBinarioBusqueda();  
  
 // Insertar nodos en el árbol binario de búsqueda  
 abb.insertar(50);  
 abb.insertar(30);  
 abb.insertar(70);  
 abb.insertar(20);  
 abb.insertar(40);  
 abb.insertar(60);  
 abb.insertar(80);  
  
 // Imprimir el árbol binario de búsqueda en orden  
 System.out.println("Recorrido Inorden del Árbol Binario de Búsqueda:");  
 abb.inorden();  
 }  
}

# Práctica

## Métodos

### ¿Qué son los métodos?

Los métodos son bloques de código que realizan una tarea específica y pueden ser reutilizados en diferentes partes de un programa

#### Métodos en Java

##### Definición de un método:

Un método en Java se define dentro de una clase y sigue la siguiente sintaxis:

modificador\_de\_acceso tipo\_de\_retorno nombre\_del\_método (lista\_de\_parámetros) {  
 // Cuerpo del método  
}

* **Modificador de acceso**: Define la accesibilidad del método (por ejemplo: public, private, protected).
* **Tipo de retorno**: Especifica el tipo de dato que el método devolverá después de su ejecución. Si el método no devuelve nada, se utiliza el tipo void.
* **Nombre del método**: Es el identificador único del método que se utilizará para llamarlo desde otras partes del programa.
* **Lista de parámetros**: Son variables que se pasan al método para su procesamiento. Pueden ser de cualquier tipo y pueden ser opcionales.

##### Llamada a un método:

Para llamar a un método desde otra parte del programa, se utiliza el nombre del método seguido de paréntesis. Si el método tiene parámetros, se deben proporcionar los valores adecuados en el mismo orden que se definieron en la lista de parámetros.

Ejemplo de llamada a un método:

tipo\_de\_retorno resultado = nombre\_del\_método(argumento1, argumento2, ...);

#### Parámetros y argumentos:

* **Parámetros**: Son variables que se declaran en la definición del método y se utilizan para recibir valores desde fuera del método.
* **Argumentos**: Son los valores reales que se pasan al método al llamarlo.

Ejemplo de método con parámetros:

public void saludar(String nombre) {  
 System.out.println("¡Hola, " + nombre + "!");  
}

En este ejemplo, nombre es un parámetro del método saludar.

Ejemplo de llamada al método con argumento:

saludar("Juan");

En este caso, “Juan” es el argumento que se pasa al método saludar.

##### Retorno de valores:

Los métodos pueden o no devolver un valor. Si un método tiene un tipo de retorno que no sea void, debe utilizar la palabra clave return seguida del valor que desea devolver.

Ejemplo de método con retorno:

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

## Arreglos

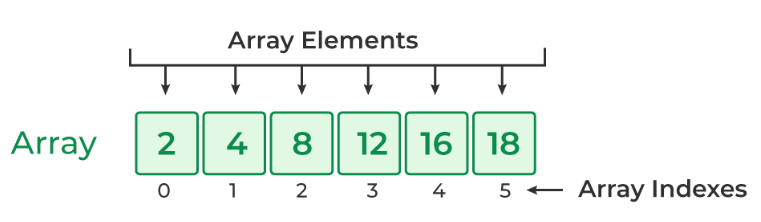
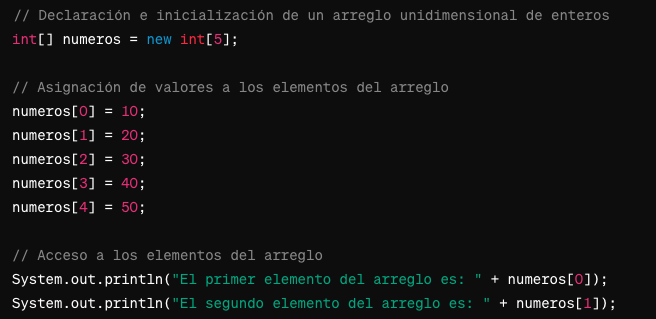
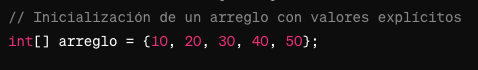
### ¿Qué son los arreglos?

Los arrays son una forma de almacenar múltiples valores del mismo tipo en una sola variable. Es como si tuvieras una caja con muchos compartimentos, donde cada compartimento puede contener un valor diferente. Para acceder a un valor específico en el array, puedes hacerlo utilizando un índice numérico.

### Características principales de los arrays en Java

* **Son estructuras de datos estáticas:** Una vez que se define el tamaño del array, éste no puede ser modificado durante la ejecución del programa.
* **Contienen elementos del mismo tipo de datos:** Todos los elementos del array deben ser del mismo tipo de datos, ya sea numérico, boolean, character o String.
* **Son objetos:** En Java, los Arrays son objetos y tienen un identificador de objeto que se puede utilizar para acceder a sus elementos.
* **Acceso a los elementos:** Los elementos de un array se acceden mediante un índice numérico, empezando por el índice cero (0).
* **Capacidad limitada**: La capacidad de almacenamiento del array está limitada por la cantidad de memoria disponible en el sistema.
* **Los elementos del array están ordenados**: Los elementos de un array están ordenados secuencialmente, lo que permite realizar operaciones de búsqueda y ordenación.
* **Se pueden inicializar en la declaración o mediante un bucle:** Los elementos de un array se pueden inicializar al declarar el array o mediante un bucle.

### Arreglos unidimensionales

En este ejemplo, tenemos un arreglo unidimensional. Cada elemento tiene un valor numérico y está almacenado en un índice específico del arreglo. Para acceder a un valor específico en el arreglo, puedes utilizar su índice correspondiente.   

### Operaciones sobre arreglos

#### 1. Declaración e inicialización de arreglos:

// Declaración e inicialización estática  
tipo[] nombre\_arreglo = {valor1, valor2, valor3};  
  
// Declaración e inicialización dinámica  
tipo[] nombre\_arreglo = new tipo[tamaño];

#### 2. Acceso a elementos del arreglo:

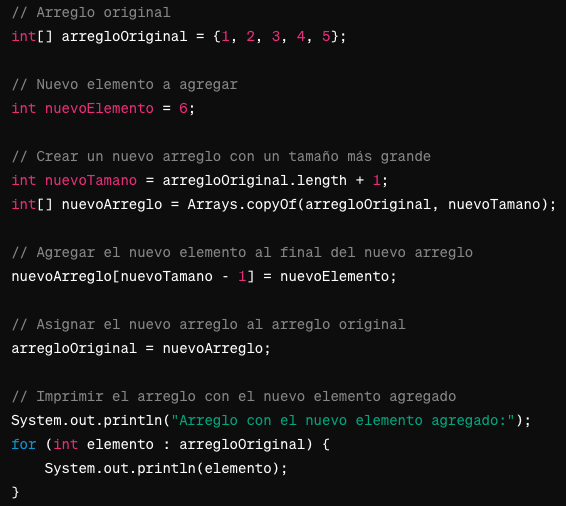
tipo valor = nombre\_arreglo[posición];

#### 3. Modificación de elementos del arreglo:

nombre\_arreglo[posición] = nuevo\_valor;

#### 4. Recorrido de un arreglo:

// Recorrido con bucle for  
for (int i = 0; i < nombre\_arreglo.length; i++) {  
 // Operaciones con nombre\_arreglo[i]  
}  
  
// Recorrido con bucle mejorado (foreach)  
for (tipo elemento : nombre\_arreglo) {  
 // Operaciones con elemento  
}



Pasted image 20240506180610.png

#### 5. Copia de arreglos:

tipo[] copia\_arreglo = nombre\_arreglo.clone();

Pasted image 20240506180739.png #### 6. Ordenamiento de arreglos:

Arrays.sort(nombre\_arreglo); // Para arreglos de tipos primitivos  
Arrays.sort(nombre\_arreglo, Comparator); // Para arreglos de objetos con un comparador personalizado

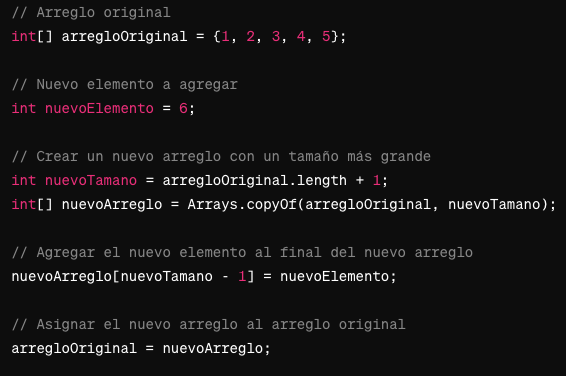
#### 7. Búsqueda en arreglos:

int posición = Arrays.binarySearch(nombre\_arreglo, valor\_a\_buscar); // Solo para arreglos ordenados

#### 8. Comparación de arreglos:

boolean iguales = Arrays.equals(arreglo1, arreglo2);

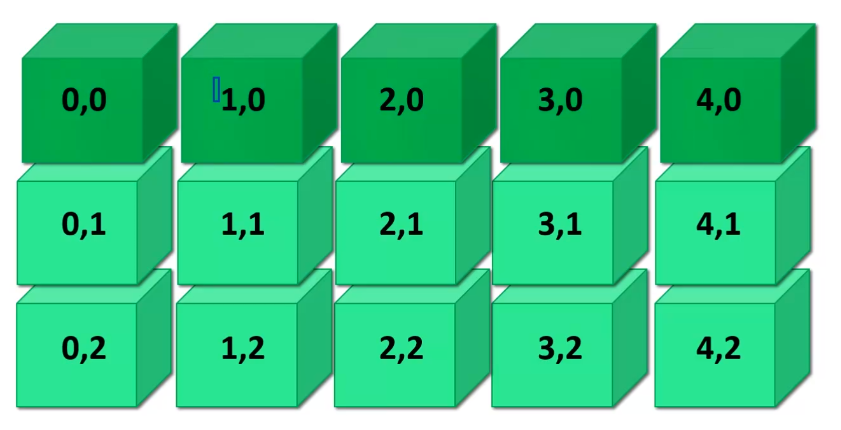
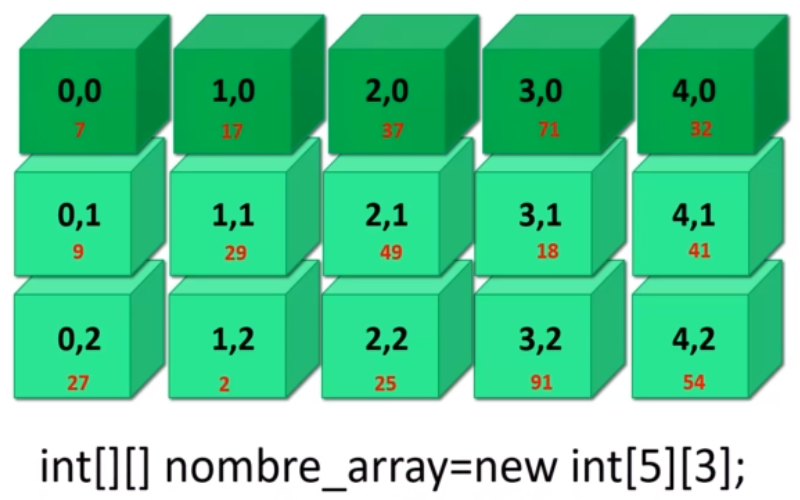
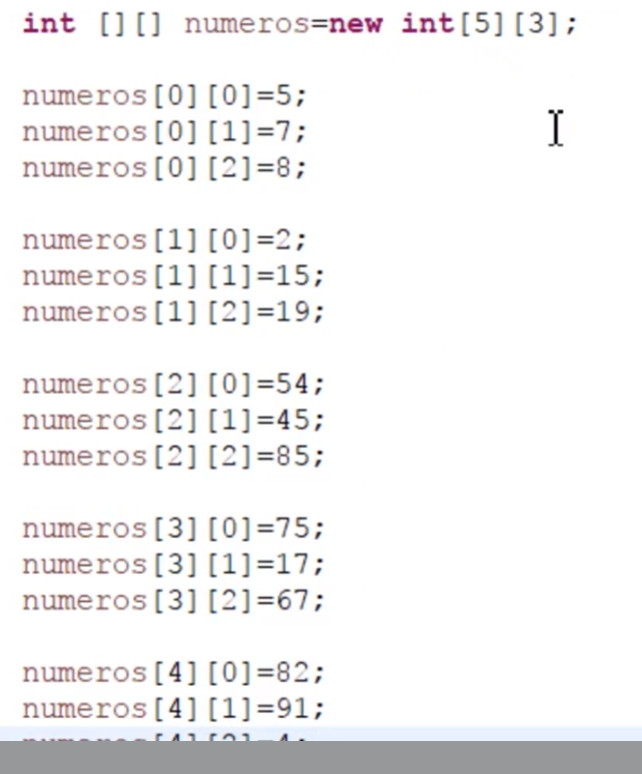
#### 9. Agregar un elemento



Pasted image 20240506180642.png

### Arreglos bidimensionales

Un array bidimensional es un array que dentro de un elemento contienen otro array. Las matrices multidimensionales emplean más de un índice para acceder a los elementos de la matriz.

Para hacer referencia a un elemento de nuestro array de 2 dimensiones utilizamos dos valores. Uno que referencia a la fila y otro que haga referencia a la columna.   

#### 1. Declaración e inicialización de arreglos bidimensionales:

// Declaración e inicialización estática  
tipo[][] nombre\_arreglo = {{val11, val12}, {val21, val22}};  
  
// Declaración e inicialización dinámica  
tipo[][] nombre\_arreglo = new tipo[filas][columnas];

#### 2. Acceso a elementos del arreglo bidimensional:

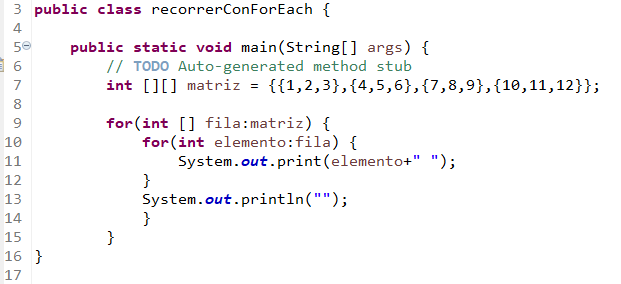
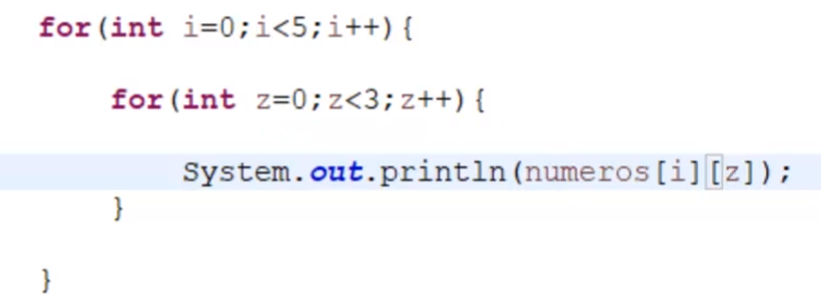
tipo valor = nombre\_arreglo[fila][columna];

#### 3. Modificación de elementos del arreglo bidimensional:

nombre\_arreglo[fila][columna] = nuevo\_valor;

#### 4. Recorrido de un arreglo bidimensional:

for (int i = 0; i < nombre\_arreglo.length; i++) {  
 for (int j = 0; j < nombre\_arreglo[i].length; j++) {  
 // Operaciones con nombre\_arreglo[i][j]  
 }  
}

#### 5. Tamaño del arreglo bidimensional:

int filas = nombre\_arreglo.length;  
int columnas = nombre\_arreglo[0].length; // Asumiendo que todas las filas tienen la misma longitud

#### 6. Arreglo bidimensional irregular:

En Java, los arreglos bidimensionales también pueden ser irregulares, lo que significa que cada fila puede tener una longitud diferente. Por ejemplo:

tipo[][] nombre\_arreglo = new tipo[3][];  
nombre\_arreglo[0] = new tipo[2];  
nombre\_arreglo[1] = new tipo[3];  
nombre\_arreglo[2] = new tipo[4];

#### 7. Operaciones comunes:

* Copia de arreglos bidimensionales.
* Comparación de arreglos bidimensionales.
* Búsqueda en arreglos bidimensionales.
* Ordenamiento de arreglos bidimensionales.

## Ordenamiento

### Bubblesort

BubbleSort es un algoritmo de ordenamiento simple que compara pares adyacentes de elementos en el arreglo y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que el arreglo esté completamente ordenado y no se realicen intercambios en un paso completo.

**Eficiencia:** BubbleSort es menos eficiente en comparación con QuickSort y MergeSort, ya que tiene un tiempo de ejecución promedio de O(n^2) en el peor y mejor caso. Es adecuado para arreglos pequeños o casi ordenados, pero no es práctico para arreglos grandes debido a su ineficiencia.

**Proceso:** 1. Compara el primer par de elementos adyacentes en la lista. 2. Si están en el orden incorrecto (por ejemplo, el primer elemento es mayor que el segundo), los intercambia. 3. Avanza un paso en la lista y repite los pasos 1 y 2 para el siguiente par de elementos adyacentes. 4. Continúa este proceso hasta el final de la lista. 5. Una vez que llegues al final de la lista, habrás puesto el elemento más grande en su posición correcta en el extremo derecho de la lista. 6. Repite los pasos 1-5 para el resto de la lista, excluyendo el elemento ya ordenado en el extremo derecho (ya que ya está en su posición correcta). 7. Repite este proceso hasta que no se realicen intercambios en un paso completo a través de la lista, lo que indica que la lista está ordenada.

public class BubbleSort {  
 // Método para ordenar una lista de enteros utilizando Bubble Sort  
 public static void bubbleSort(int[] arr) {  
 int n = arr.length;  
 // Iterar sobre la lista n veces  
 for (int i = 0; i < n - 1; i++) {  
 // En cada iteración, comparar pares de elementos adyacentes  
 // y hacer los intercambios necesarios para mover el elemento más grande al final  
 for (int j = 0; j < n - i - 1; j++) {  
 // Comparar el elemento actual con el siguiente elemento  
 if (arr[j] > arr[j + 1]) {  
 // Si el elemento actual es mayor que el siguiente, intercambiarlos  
 int temp = arr[j];  
 arr[j] = arr[j + 1];  
 arr[j + 1] = temp;  
 }  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 int[] arr = {64, 34, 25, 12, 22, 11, 90};  
  
 System.out.println("Lista antes de ordenar:");  
 printArray(arr);  
  
 // Llamar al método bubbleSort para ordenar la lista  
 bubbleSort(arr);  
  
 System.out.println("Lista después de ordenar:");  
 printArray(arr);  
 }  
  
 // Método auxiliar para imprimir una lista de enteros  
 public static void printArray(int[] arr) {  
 for (int i = 0; i < arr.length; i++) {  
 System.out.print(arr[i] + " ");  
 }  
 System.out.println();  
 }  
}

**Explicación del funcionamiento:** 1. La función bubbleSort toma una matriz de enteros como argumento. 2. El bucle externo for itera sobre la lista n veces, donde n es la longitud de la lista. 3. Dentro del bucle externo, el bucle interno for itera sobre la lista hasta el elemento n - i - 1. Esto se debe a que después de cada iteración del bucle externo, el elemento más grande se coloca en su posición correcta en el extremo derecho de la lista, por lo que no es necesario volver a compararlo. 4. Dentro del bucle interno, se comparan los elementos adyacentes y se intercambian si están en el orden incorrecto. 5. Después de n iteraciones del bucle externo, la lista estará ordenada.

### QuickSort

QuickSort es un algoritmo de ordenamiento eficiente que utiliza la estrategia de dividir y conquistar. El algoritmo elige un elemento pivote y particiona el arreglo en dos sub-arreglos alrededor del pivote, de manera que los elementos más pequeños que el pivote estén a su izquierda y los elementos más grandes estén a su derecha. Luego, aplica recursivamente el mismo proceso a los sub-arreglos. **Eficiencia:** QuickSort es generalmente rápido y eficiente, con un tiempo de ejecución promedio de O(n log n). Sin embargo, en el peor caso puede degradarse a O(n^2) si el pivote seleccionado no divide el arreglo de manera equilibrada.

El algoritmo Quicksort es un algoritmo de ordenamiento eficiente que utiliza el método de “dividir para conquistar” para ordenar una lista de elementos. Funciona seleccionando un elemento como pivote y dividiendo la lista en dos subgrupos: uno con elementos menores que el pivote y otro con elementos mayores. Luego, Quicksort se aplica recursivamente a cada subgrupo.

1. **Selección del pivote**: Selecciona un elemento de la lista como pivote. Hay varias estrategias para elegir el pivote, pero una opción común es seleccionar el elemento medio.
2. **Partición**: Reordena la lista de manera que todos los elementos menores que el pivote estén a su izquierda y todos los elementos mayores estén a su derecha. Esto se logra moviendo los elementos alrededor del pivote de manera que los elementos menores estén en un lado y los mayores en el otro.
3. **Recursión**: Aplica Quicksort recursivamente a las dos sub-listas resultantes, una que contiene elementos menores que el pivote y otra que contiene elementos mayores. Esto se hace hasta que las sub-listas sean lo suficientemente pequeñas como para estar ordenadas trivialmente (por ejemplo, sub-listas de tamaño 0 o 1).
4. **Combinación**: No se requiere ninguna combinación explícita en Quicksort, ya que los elementos se reorganizan en su lugar durante el proceso de partición.

public class QuickSort {  
 public static void quickSort(int[] arr, int low, int high) {  
 if (low < high) {  
 // Encuentra la posición del pivote después de particionar la lista  
 int pivotIndex = partition(arr, low, high);  
  
 // Aplica Quicksort recursivamente a las sub-listas izquierda y derecha del pivote  
 quickSort(arr, low, pivotIndex - 1);  
 quickSort(arr, pivotIndex + 1, high);  
 }  
 }  
  
 public static int partition(int[] arr, int low, int high) {  
 // Selecciona el pivote (en este caso, el último elemento)  
 int pivot = arr[high];  
 int i = low - 1;  
  
 // Recorre la lista desde el principio hasta el penúltimo elemento  
 for (int j = low; j < high; j++) {  
 // Si el elemento actual es menor o igual que el pivote, lo coloca en la posición correcta  
 if (arr[j] <= pivot) {  
 i++;  
 // Intercambia el elemento actual con el elemento en la posición correcta  
 int temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
 }  
 }  
  
 // Intercambia el pivote con el elemento en la posición correcta  
 int temp = arr[i + 1];  
 arr[i + 1] = arr[high];  
 arr[high] = temp;  
  
 // Retorna la posición del pivote  
 return i + 1;  
 }  
  
 public static void main(String[] args) {  
 int[] arr = {64, 34, 25, 12, 22, 11, 90};  
  
 System.out.println("Lista antes de ordenar:");  
 printArray(arr);  
  
 // Llama al método quickSort para ordenar la lista  
 quickSort(arr, 0, arr.length - 1);  
  
 System.out.println("Lista después de ordenar:");  
 printArray(arr);  
 }  
  
 // Método auxiliar para imprimir una lista de enteros  
 public static void printArray(int[] arr) {  
 for (int i = 0; i < arr.length; i++) {  
 System.out.print(arr[i] + " ");  
 }  
 System.out.println();  
 }  
}

Esta implementación de Quicksort tiene una complejidad de tiempo promedio de O(n log n), lo que lo hace muy eficiente para ordenar grandes conjuntos de datos. Sin embargo, en el peor caso, puede degenerar a O(n^2), por ejemplo, si la lista está completamente ordenada y se elige el primer o último elemento como pivote.

### MergeSort

MergeSort es un algoritmo de ordenamiento estable que también utiliza la estrategia de dividir y conquistar. Divide el arreglo en dos mitades, ordena recursivamente cada mitad y luego combina las mitades ordenadas en un solo arreglo ordenado.

**Eficiencia:** MergeSort es eficiente y siempre tiene un tiempo de ejecución garantizado de O(n log n), independientemente de la distribución de los elementos en el arreglo. Sin embargo, requiere espacio adicional para almacenar el arreglo auxiliar durante el proceso de fusión.

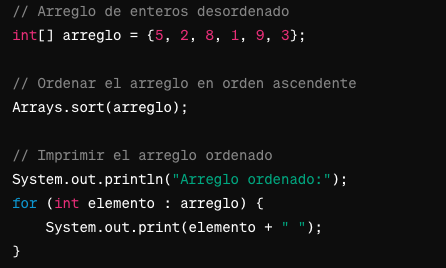
Merge Sort es otro algoritmo de ordenamiento eficiente que también utiliza el enfoque de “dividir para conquistar”. En Merge Sort, la lista se divide en sub-listas más pequeñas hasta que cada sublista tenga un solo elemento, luego las sub-lista se combinan en orden hasta que la lista esté completamente ordenada.

1. **División**: Dividir la lista en dos mitades iguales.
2. **Recursión**: Aplicar Merge Sort recursivamente a cada mitad de la lista hasta que cada mitad tenga un solo elemento.
3. **Combinación (Merge)**: Combina las sub-lista ordenadas en una sola lista ordenada. Esto se hace comparando los elementos de las sub-listas y colocando el elemento más pequeño en la lista resultante. Este proceso se repite hasta que todas las sub-listas se hayan combinado en una lista ordenada.

public class MergeSort {  
 public static void mergeSort(int[] arr, int left, int right) {  
 if (left < right) {  
 // Encuentra el punto medio de la lista  
 int middle = (left + right) / 2;  
  
 // Aplica Merge Sort recursivamente a la primera mitad  
 mergeSort(arr, left, middle);  
 // Aplica Merge Sort recursivamente a la segunda mitad  
 mergeSort(arr, middle + 1, right);  
  
 // Combina las dos mitades ordenadas  
 merge(arr, left, middle, right);  
 }  
 }  
  
 public static void merge(int[] arr, int left, int middle, int right) {  
 // Tamaño de las dos sub-listas  
 int n1 = middle - left + 1;  
 int n2 = right - middle;  
  
 // Crea dos arreglos temporales para almacenar las sub-listas  
 int[] L = new int[n1];  
 int[] R = new int[n2];  
  
 // Copia los elementos a los arreglos temporales  
 for (int i = 0; i < n1; ++i) {  
 L[i] = arr[left + i];  
 }  
 for (int j = 0; j < n2; ++j) {  
 R[j] = arr[middle + 1 + j];  
 }  
  
 // Índices iniciales de las dos sub-listas y el índice de la lista principal  
 int i = 0, j = 0, k = left;  
  
 // Combina las dos sub-listas en la lista principal  
 while (i < n1 && j < n2) {  
 if (L[i] <= R[j]) {  
 arr[k] = L[i];  
 i++;  
 } else {  
 arr[k] = R[j];  
 j++;  
 }  
 k++;  
 }  
  
 // Copia los elementos restantes de L[] si los hay  
 while (i < n1) {  
 arr[k] = L[i];  
 i++;  
 k++;  
 }  
  
 // Copia los elementos restantes de R[] si los hay  
 while (j < n2) {  
 arr[k] = R[j];  
 j++;  
 k++;  
 }  
 }  
  
 public static void main(String[] args) {  
 int[] arr = {64, 34, 25, 12, 22, 11, 90};  
  
 System.out.println("Lista antes de ordenar:");  
 printArray(arr);  
  
 // Llama al método mergeSort para ordenar la lista  
 mergeSort(arr, 0, arr.length - 1);  
  
 System.out.println("Lista después de ordenar:");  
 printArray(arr);  
 }  
  
 // Método auxiliar para imprimir una lista de enteros  
 public static void printArray(int[] arr) {  
 for (int i = 0; i < arr.length; i++) {  
 System.out.print(arr[i] + " ");  
 }  
 System.out.println();  
 }  
}

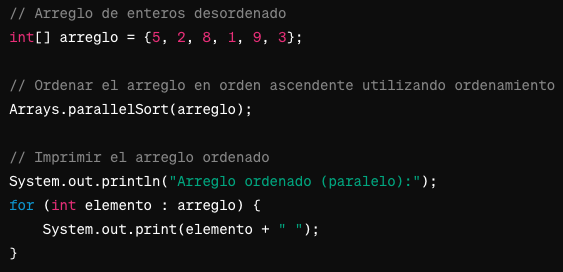
Esta implementación de Merge Sort tiene una complejidad de tiempo de O(n log n) en todos los casos, lo que la hace muy eficiente para ordenar grandes conjuntos de datos. Además, Merge Sort es estable y no tiene un peor caso particular como Quicksort.

### Ordenamiento Secuencial



Pasted image 20240506185818.png

### Ordenamiento paralelo



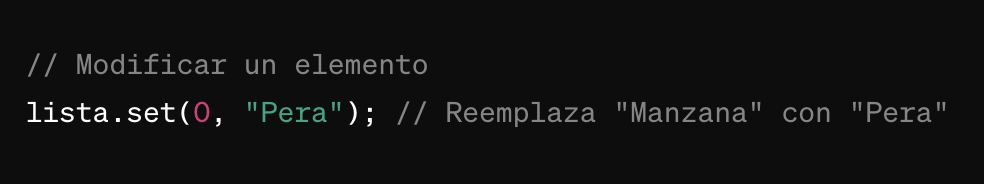
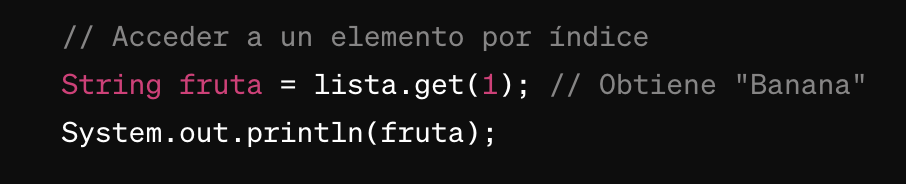
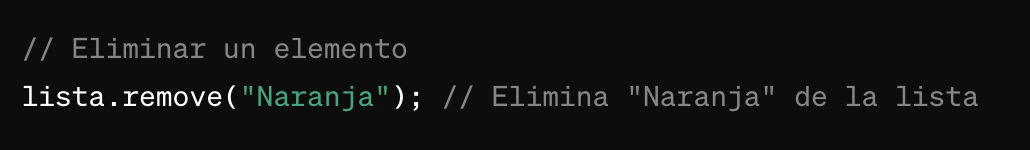
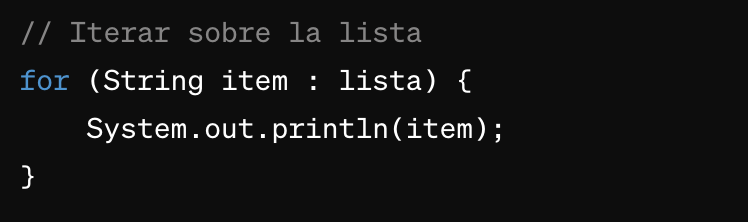
Pasted image 20240506185831.png

## Estructuras dinámicas

### Listas

Una lista es una estructura de datos que almacena una colección ordenada de elementos. En Java, una de las implementaciones más comunes es ArrayList, que es una lista redimensionable que implementa la interfaz List.

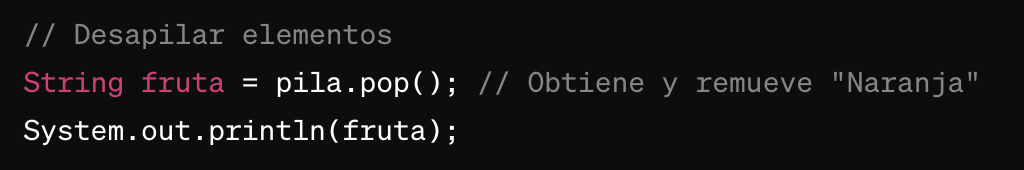
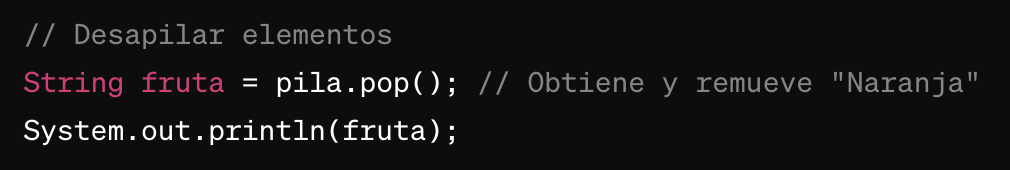
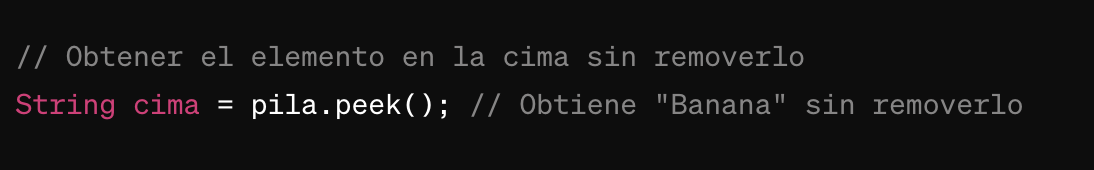
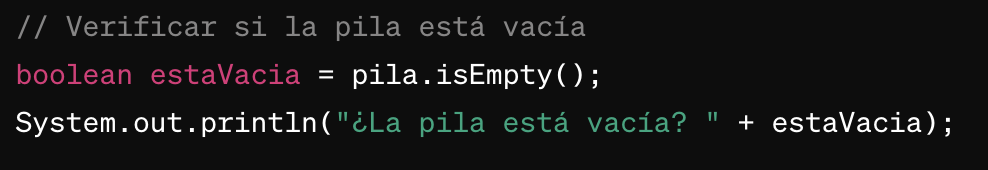
**Características:** - **Tamaño Dinámico:** Puedes agregar o eliminar elementos en cualquier momento sin tener que preocuparte por el tamaño inicial del array. - **Implementa la Interfaz List:** ArrayList implementa la interfaz List, lo que significa que hereda todas las operaciones definidas en esa interfaz, como agregar, eliminar, buscar elementos, entre otras. - **Acceso Aleatorio Eficiente:** Puedes acceder a cualquier elemento en la lista directamente mediante su índice. - **Permite Duplicados y Elementos Nulos:** Permiten elementos duplicados y pueden contener elementos nulos (null). - **Iteración Eficiente:** Los ArrayList proporcionan métodos eficientes para iterar sobre los elementos de la lista, como el uso de bucles for-each o la iteración a través de índices.

### Pilas

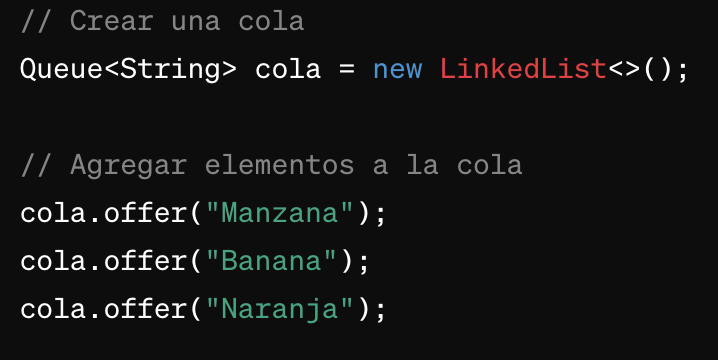
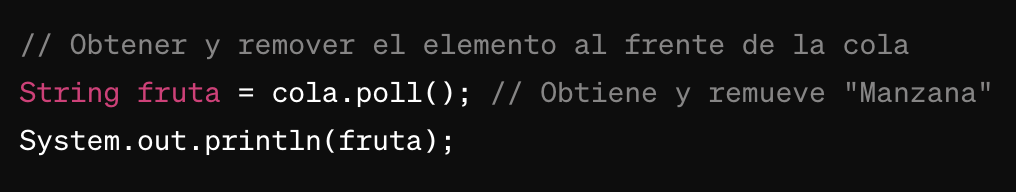
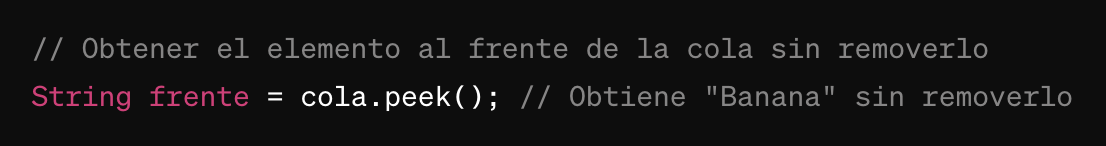
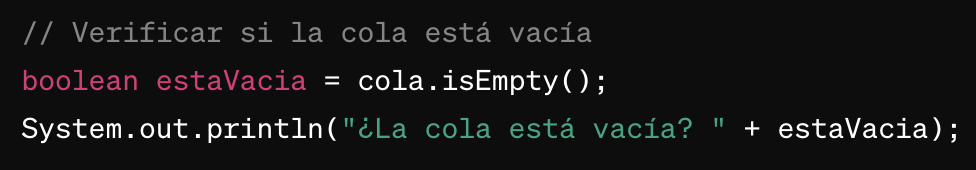
Una pila es una estructura de datos que sigue el principio LIFO (Last In, First Out), lo que significa que el último elemento en entrar es el primero en salir. En Java, puedes implementar una pila utilizando Stack de la clase java.util.

**Características:** - **LIFO:** El último elemento agregado es el primero en ser removido. Esto lo hace útil en situaciones donde se necesita un comportamiento de apilamiento. - **Operaciones Básicas:** Un Stack proporciona las operaciones básicas de una pila. - push() para agregar un elemento en la cima de la pila - pop() para remover y devolver el elemento en la cima de la pila - peek() para obtener el elemento en la cima sin removerlo. - **Métodos Adicionales:** Además de las operaciones básicas, también proporciona otros métodos. - isEmpty() para verificar si la pila está vacía - search() para buscar un elemento y obtener su posición en la pila - size() para obtener el tamaño actual de la pila

### Colas

Una cola es una estructura de datos que sigue el principio FIFO (First In, First Out), lo que significa que el primer elemento en entrar es el primero en salir. En Java, puedes implementar una cola utilizando LinkedList de la clase java.util.

* **Estructura de Datos de Lista Doblemente Enlazada:** Cada elemento está representado por un nodo que contiene referencias tanto al anterior como al siguiente en la lista. Esto permite un acceso eficiente tanto hacia adelante como hacia atrás en la lista.
* **Almacenamiento Dinámico:** Puede crecer o reducirse automáticamente según sea necesario.
* **Operaciones de Inserción y Eliminación Eficientes:** Debido a su estructura de lista doblemente enlazada, LinkedList es particularmente eficiente para operaciones de inserción y eliminación en cualquier posición de la lista.
* **Acceso Secuencial Eficiente:** Las operaciones como get() pueden ser menos eficientes que en un ArrayList. Uso de Memoria Adicional: Debido a la necesidad de almacenar referencias a los nodos previos y siguientes, LinkedList puede consumir más memoria que ArrayList, especialmente para listas grandes.
* **Iteración Eficiente:** LinkedList proporciona métodos eficientes para iterar sobre los elementos de la lista, ya sea mediante el uso de bucles for-each o iteradores.    

## Archivos

### Archivos y Directorios

Trabajar con directorios y archivos es una tarea común en la programación, y Java proporciona una amplia gama de clases y métodos en el paquete java.io y java.nio.file para manejar archivos y directorios.

La clase File en Java se utiliza para representar archivos y directorios en el sistema de archivos.

### Creación de un Directorio

Para crear un directorio en Java, puedes usar el método mkdir() o mkdirs() de la clase File.

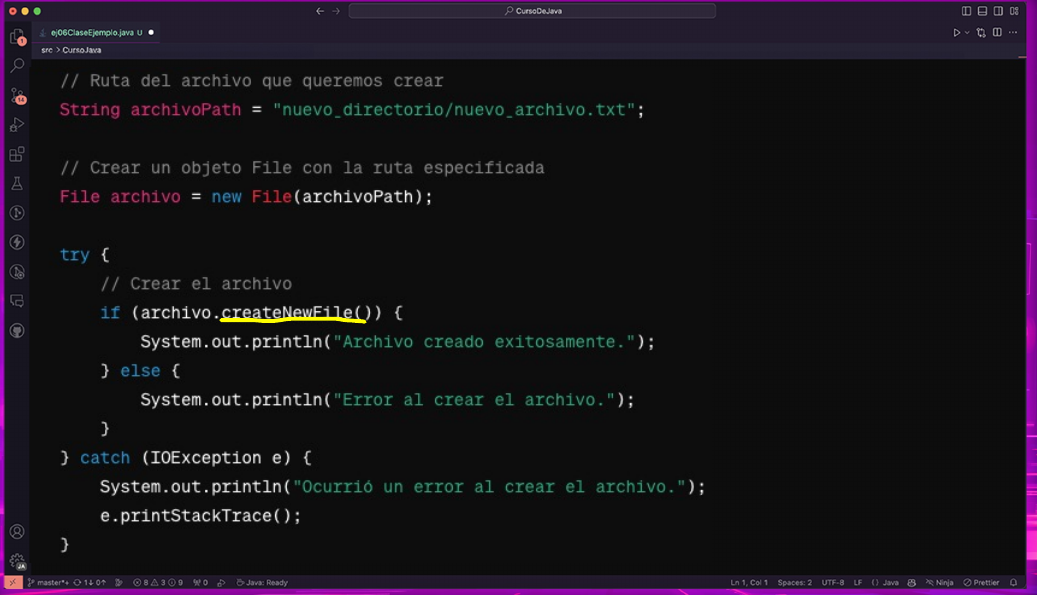
import java.io.File;  
  
public class CrearDirectorio {  
 public static void main(String[] args) {  
 File directorio = new File("nuevoDirectorio");  
 if (directorio.mkdir()) {  
 System.out.println("Directorio creado exitosamente.");  
 } else {  
 System.out.println("No se pudo crear el directorio.");  
 }  
 }  
}

* **mkdir()**: Crea un directorio si no existe. Devuelve true si el directorio fue creado con éxito, y false en caso contrario (por ejemplo, si ya existe).
* **mkdirs()**: Crea el directorio especificado junto con cualquier directorio intermedio necesario. Devuelve true si se crearon los directorios, y false si ya existían.

### Creación de un Archivo

Para crear un archivo en Java, puedes usar el método createNewFile() de la clase File.

import java.io.File;  
import java.io.IOException;  
  
public class CrearArchivo {  
 public static void main(String[] args) {  
 File archivo = new File("nuevoArchivo.txt");  
 try {  
 if (archivo.createNewFile()) {  
 System.out.println("Archivo creado exitosamente.");  
 } else {  
 System.out.println("El archivo ya existe.");  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}



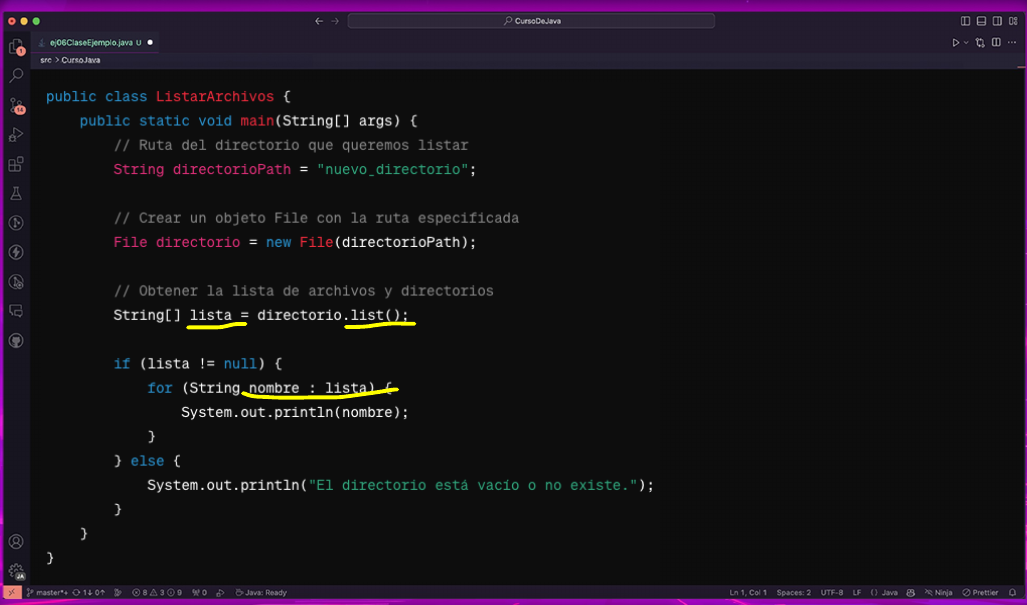
Pasted image 20240526221321.png

**createNewFile()**: Crea un nuevo archivo vacío si no existe. Devuelve true si el archivo fue creado con éxito, y false si el archivo ya existe. Puede lanzar una excepción IOException si ocurre un error de entrada/salida.

### Listar Archivos y Directorios

Para listar archivos y directorios dentro de un directorio, puedes usar el método listFiles() de la clase File.

import java.io.File;  
  
public class ListarArchivos {  
 public static void main(String[] args) {  
 File directorio = new File("rutaDelDirectorio");  
 File[] listaDeArchivos = directorio.listFiles();  
 if (listaDeArchivos != null) {  
 for (File archivo : listaDeArchivos) {  
 if (archivo.isDirectory()) {  
 System.out.println("Directorio: " + archivo.getName());  
 } else {  
 System.out.println("Archivo: " + archivo.getName());  
 }  
 }  
 } else {  
 System.out.println("El directorio no existe o no es un directorio.");  
 }  
 }  
}



Pasted image 20240526221829.png

* **listFiles()**: Devuelve un array de objetos File que representan los archivos y directorios en el directorio especificado. Devuelve null si el objeto File no representa un directorio.
* list(): Devuelve un array de String con los nombres de los archivos y directorios.
* listFiles(): Devuelve un array de File con los objetos File que representan los archivos y directorios.

### Eliminar Archivos

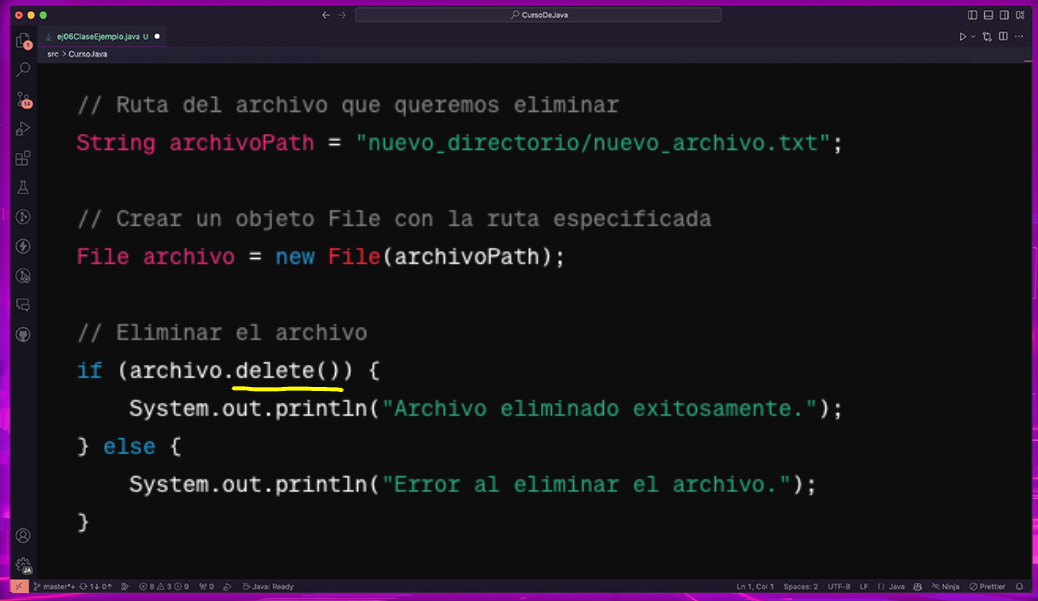
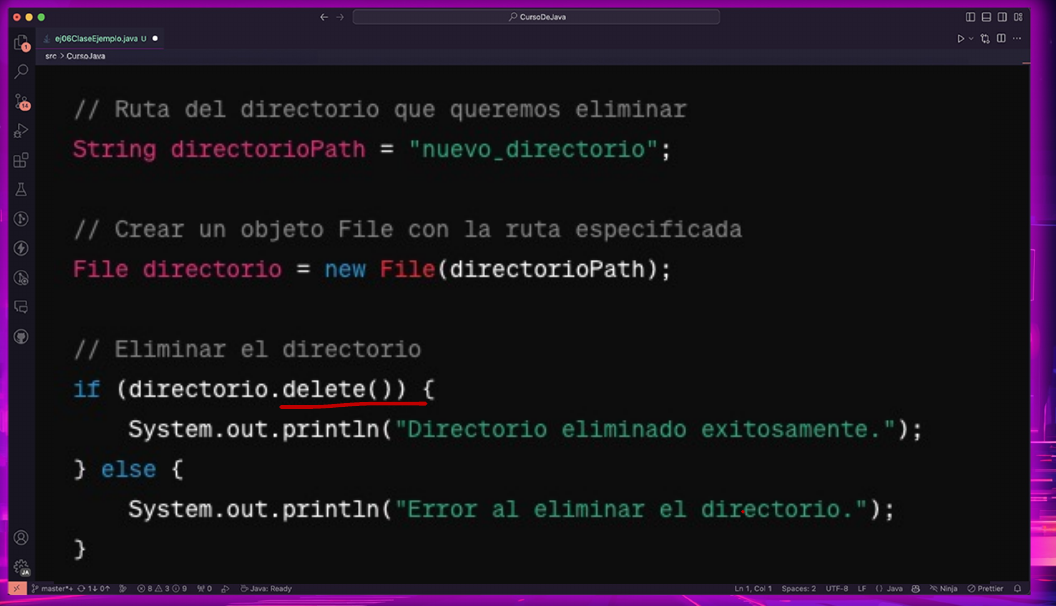
Para eliminar un archivo, puedes usar el método delete() de la clase File.

import java.io.File;  
  
public class EliminarArchivo {  
 public static void main(String[] args) {  
 File archivo = new File("archivoAEliminar.txt");  
 if (archivo.delete()) {  
 System.out.println("Archivo eliminado exitosamente.");  
 } else {  
 System.out.println("No se pudo eliminar el archivo.");  
 }  
 }  
}

### Eliminar Directorios

Para eliminar un directorio, puedes usar el método delete() de la clase File. El directorio debe estar vacío para que pueda ser eliminado.

import java.io.File;  
  
public class EliminarDirectorio {  
 public static void main(String[] args) {  
 File directorio = new File("directorioAEliminar");  
 if (directorio.delete()) {  
 System.out.println("Directorio eliminado exitosamente.");  
 } else {  
 System.out.println("No se pudo eliminar el directorio. Asegúrate de que esté vacío.");  
 }  
 }  
}

  1. **Método de Creación**: - **Archivos**: Se utiliza createNewFile(). - **Directorios**: Se utilizan mkdir() y mkdirs(). 2. **Condiciones para la Eliminación**: - **Archivos**: Pueden eliminarse directamente si existen. - **Directorios**: Deben estar vacíos para poder eliminarse. 3. **Excepciones**: - **Archivos**: createNewFile() puede lanzar IOException. - **Directorios**: mkdir() y mkdirs() no lanzan excepciones, pero devuelven false si no pueden crear el directorio. 4. **Retorno de Métodos**: - **Archivos**: createNewFile() devuelve true si se crea el archivo, false si ya existe. - **Directorios**: mkdir() y mkdirs() devuelven true si se crea el directorio, false en caso contrario. 5. **Existencia y Estado Previo**: - **Archivos**: createNewFile() no hace nada si el archivo ya existe. - **Directorios**: mkdir() crea solo el último directorio especificado, mientras que mkdirs() crea todos los directorios necesarios en la ruta especificada.

### Lectura de archivos de texto

Para leer un archivo de texto en Java, puedes usar FileReader junto con BufferedReader.

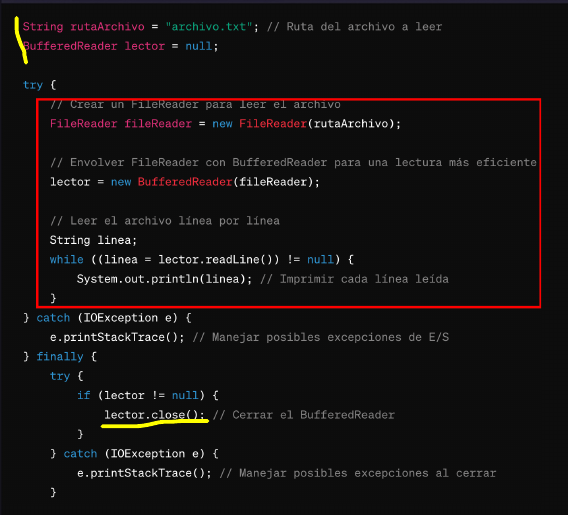
1. Asegúrate de que el archivo exista en el directorio de trabajo.
2. Ejecuta el programa LeerArchivo.
3. El programa leerá e imprimirá el contenido del archivo línea por línea.

**Leer el archivo línea por línea** - FileReader se encarga de la lectura básica del archivo. - BufferedReader se envuelve alrededor de FileReader para proporcionar una lectura más eficiente y métodos adicionales como readLine().

Se usa un bucle while para leer el archivo línea por línea hasta que readLine() devuelva null, lo que indica el final del archivo.

El bloque finally asegura que el BufferedReader se cierre, incluso si ocurre una excepción.

import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
  
public class LeerArchivo {  
 public static void main(String[] args) {  
 BufferedReader reader = null;  
 try {  
 reader = new BufferedReader(new FileReader("archivo.txt"));  
 String linea;  
 while ((linea = reader.readLine()) != null) {  
 System.out.println(linea);  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (reader != null) {  
 reader.close();  
 }  
 } catch (IOException ex) {  
 ex.printStackTrace();  
 }  
 }  
 }  
}

 ### Escribir en un archivo de texto

Para escribir en un archivo de texto en Java, puedes usar FileWriter junto con BufferedWriter.

1. El programa escribirá las líneas de texto en archivo.
2. Verifica el contenido de archivo para asegurarte de que las líneas se hayan escrito correctamente.

**Escribir en un archivo de texto**

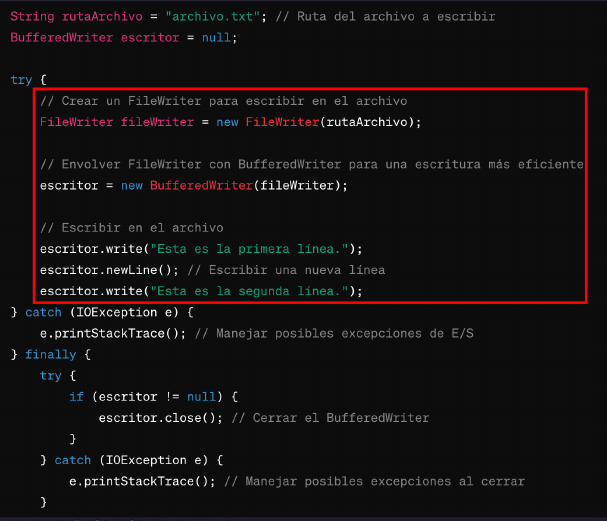
FileWriter y BufferedWriter se utilizan para escribir en archivos de texto de manera eficiente.

* FileWriter se encarga de la escritura básica del archivo.
* BufferedWriter se envuelve alrededor de FileWriter para proporcionar una escritura más eficiente y métodos adicionales como newLine().

Se escriben líneas de texto en el archivo utilizando el método write() y newLine() para insertar saltos de línea.

El bloque finally asegura que el BufferedWriter se cierre, incluso si ocurre una excepción.

import java.io.BufferedWriter;  
import java.io.FileWriter;  
import java.io.IOException;  
  
public class EscribirArchivo {  
 public static void main(String[] args) {  
 BufferedWriter writer = null;  
 try {  
 writer = new BufferedWriter(new FileWriter("archivo.txt"));  
 writer.write("Primera línea de texto");  
 writer.newLine();  
 writer.write("Segunda línea de texto");  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (writer != null) {  
 writer.close();  
 }  
 } catch (IOException ex) {  
 ex.printStackTrace();  
 }  
 }  
 }  
}



Pasted image 20240526223511.png

### Escribir al final del archivo

Para añadir texto al final de un archivo existente, se puede usar el constructor de FileWriter que acepta un segundo parámetro true para el modo de adición.

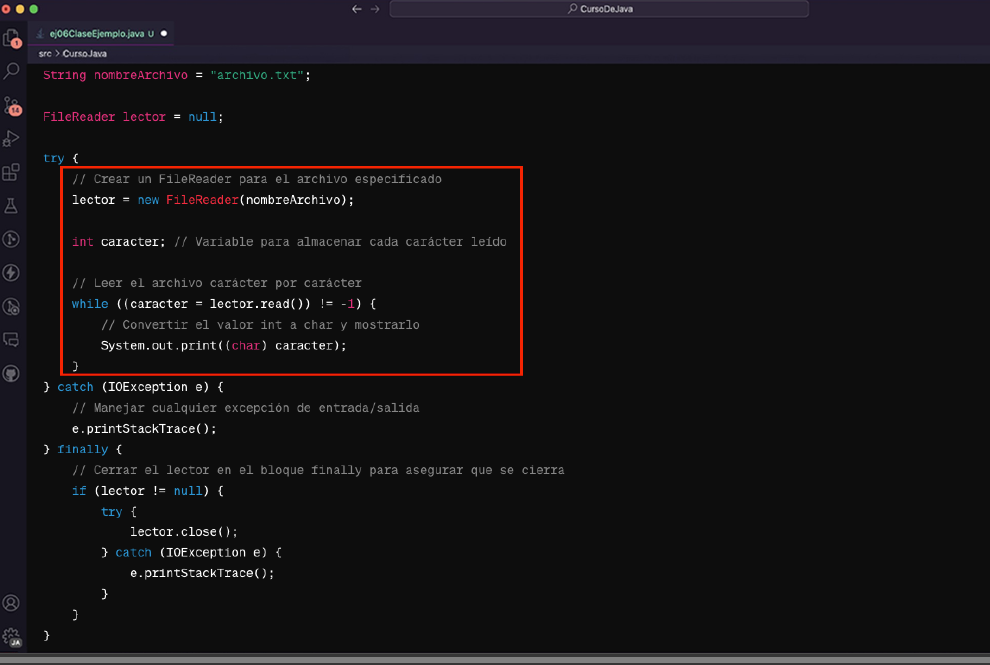
import java.io.BufferedWriter;  
import java.io.FileWriter;  
import java.io.IOException;  
  
public class EscribirAlFinal {  
 public static void main(String[] args) {  
 BufferedWriter writer = null;  
 try {  
 writer = new BufferedWriter(new FileWriter("archivo.txt", true));  
 writer.write("Línea adicional de texto");  
 writer.newLine();  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (writer != null) {  
 writer.close();  
 }  
 } catch (IOException ex) {  
 ex.printStackTrace();  
 }  
 }  
 }  
}

### Leer archivo carácter por carácter

Para recorrer y leer un archivo de texto carácter por carácter en Java, puedes utilizar la clase FileReader que es adecuada para leer flujos de datos de caracteres.

**Leer el archivo carácter por carácter**

import java.io.FileReader;  
import java.io.IOException;  
  
public class LeerCaracterPorCaracter {  
 public static void main(String[] args) {  
 FileReader reader = null;  
 try {  
 reader = new FileReader("archivo.txt");  
 int caracter;  
 while ((caracter = reader.read()) != -1) {  
 System.out.print((char) caracter);  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (reader != null) {  
 reader.close();  
 }  
 } catch (IOException ex) {  
 ex.printStackTrace();  
 }  
 }  
 }  
}



Pasted image 20240526223743.png

**Cerrar el FileReader**

Cerrar el FileReader es esencial para liberar los recursos del sistema y evitar fugas de memoria. Siempre se debe cerrar en un bloque finally para asegurar que se cierre incluso si ocurre una excepción.

## try-catch

En Java, el bloque try-catch se utiliza para manejar excepciones. Una excepción es un evento que interrumpe el flujo normal de la ejecución de un programa. Las excepciones pueden ocurrir por diversas razones, como errores de entrada/salida, operaciones aritméticas inválidas, acceso a índices fuera de los límites de un array, entre otros.

El manejo de excepciones es crucial para escribir programas robustos y que puedan gestionar errores de manera adecuada.

### ¿Por qué se usa try-catch?

1. **Robustez:** Mejora la robustez del programa al permitir que el programa maneje errores de manera controlada en lugar de terminar abruptamente.
2. **Mantenimiento:** Facilita el mantenimiento del código al centralizar la lógica de manejo de errores.
3. **Seguridad:** Permite limpiar recursos (como archivos o conexiones de base de datos) de manera segura incluso cuando ocurre un error.
4. **Experiencia del Usuario:** Proporciona una mejor experiencia de usuario al manejar los errores de manera adecuada, mostrando mensajes de error amigables o intentando recuperarse del error.

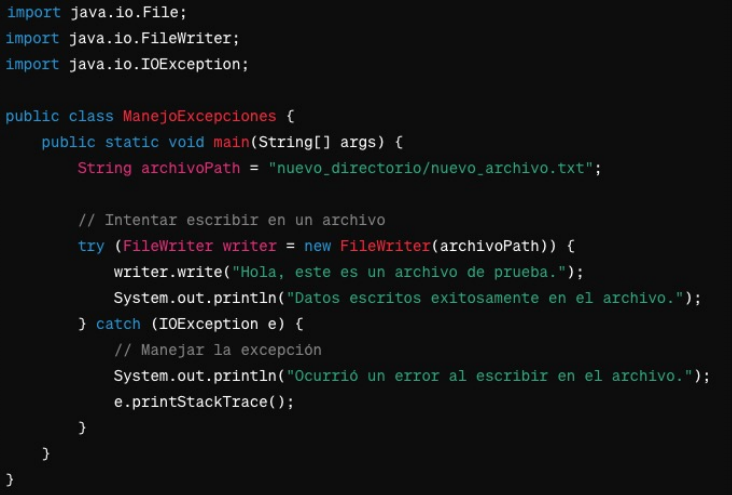
### Estructura de try-catch

try {  
 // Código que puede lanzar una excepción  
} catch (TipoDeExcepcion nombreVariable) {  
 // Código para manejar la excepción  
}

### Manejo de Excepciones en Entrada/Salida

* **try:** Intenta escribir datos en un archivo. FileWriter puede lanzar una excepción IOException si ocurre un error durante la escritura.
* **catch (IOException e):** Captura la excepción IOException y permite al programa manejarla, en este caso, imprimiendo un mensaje de error y la traza de la excepción.

import java.io.FileWriter;  
import java.io.IOException;  
  
public class ManejoDeExcepcionesIO {  
 public static void main(String[] args) {  
 try {  
 FileWriter escritor = new FileWriter("archivo.txt");  
 escritor.write("Hola, Mundo!");  
 escritor.close();  
 } catch (IOException e) {  
 System.out.println("Ocurrió un error de entrada/salida: " + e.getMessage());  
 e.printStackTrace();  
 }  
 }  
}



Pasted image 20240526224610.png

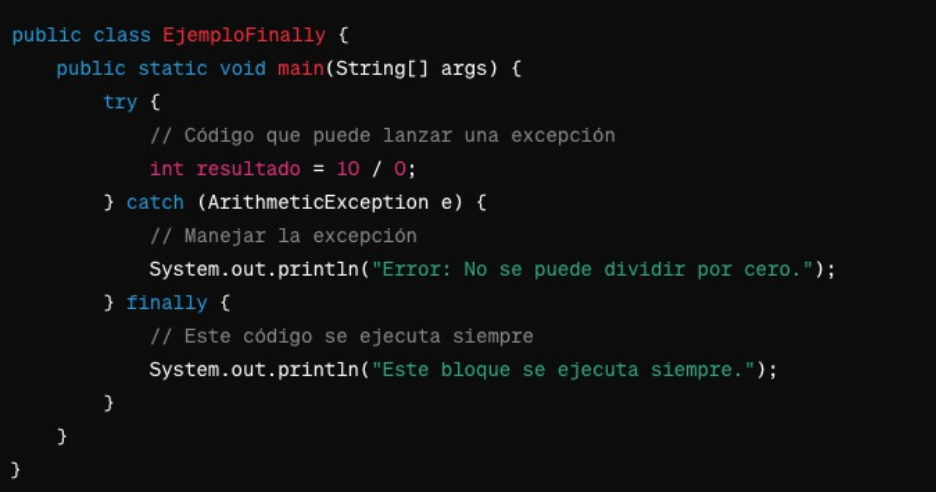
### Excepciones en Operaciones Aritméticas

* **try:** Intenta realizar una división. Dividir por cero lanzará una excepción ArithmeticException.
* **catch (ArithmeticException e):** Captura la excepción ArithmeticException y maneja el error, en este caso, imprimiendo un mensaje de error específico.

public class ManejoDeExcepcionesAritmeticas {  
 public static void main(String[] args) {  
 try {  
 int resultado = 10 / 0;  
 } catch (ArithmeticException e) {  
 System.out.println("Error: División por cero.");  
 e.printStackTrace();  
 }  
 }  
}

### Bloques finally

A veces, además de try y catch, se usa un bloque finally que se ejecuta siempre, independientemente de si se lanzó una excepción o no. Es útil para liberar recursos.



Pasted image 20240526224844.png

### try-with-resources

Esta declaración simplifica el manejo de recursos como archivos, conexiones a bases de datos, etc., asegurando que se cierren automáticamente al final del bloque try.

El bloque *try-with-resources* se usa para manejar automáticamente la liberación de recursos que implementan la interfaz AutoCloseable. Aquí hay un ejemplo práctico:

import java.io.FileWriter;  
import java.io.IOException;  
  
public class TryWithResourcesExample {  
 public static void main(String[] args) {  
 try (FileWriter escritor = new FileWriter("archivo.txt")) {  
 escritor.write("Hola, Mundo!");  
 } catch (IOException e) {  
 System.out.println("Ocurrió un error de entrada/salida: " + e.getMessage());  
 e.printStackTrace();  
 }  
 }  
}

### Explicación del código

* **try-with-resources**: En este bloque try, los recursos que se abren (como FileWriter) se declaran entre paréntesis después de try. Estos recursos se cerrarán automáticamente cuando se complete el bloque try, ya sea que ocurra una excepción o no.
* **catch (IOException e)**: Captura cualquier IOException que ocurra durante la operación de escritura.

### Beneficios de try-with-resources

1. **Simplificación del Código**: No es necesario escribir código adicional en un bloque finally para cerrar recursos. Esto reduce el riesgo de errores y hace que el código sea más limpio y fácil de mantener.
2. **Seguridad y Confiabilidad**: Asegura que los recursos se cierren adecuadamente incluso si ocurre una excepción, evitando posibles fugas de recursos.

### Con y sin Try with resources

try-with-resources and traditional try-catch-finally blocks are both mechanisms in Java for handling resources, but they differ in terms of syntax, readability, and resource management.

**1. Syntax:** - In traditional try-catch-finally blocks, you manually open and close resources within the try block and handle exceptions and resource closure in the catch and finally blocks respectively.

InputStream inputStream = null;  
try {  
 inputStream = new FileInputStream("file.txt");  
 // Read from inputStream  
} catch (IOException e) {  
 // Handle exception  
} finally {  
 if (inputStream != null) {  
 try {  
 inputStream.close();  
 } catch (IOException e) {  
 // Handle exception  
 }  
 }  
}

* With try-with-resources, you declare the resource inside the parentheses of the try statement. The resource is automatically closed at the end of the block, eliminating the need for a finally block.

try (InputStream inputStream = new FileInputStream("file.txt")) {  
 // Read from inputStream  
} catch (IOException e) {  
 // Handle exception  
}