UD11 – GESTIÓN DINÁMICA DE MEMORIA

Contenido

1 Colecciones

En muchas aplicaciones, es necesario crear y administrar grupos de objetos relacionados. Hasta ahora, hemos visto cómo utilizar arrays para almacenar múltiples objetos de un mismo tipo. Sin embargo, los arrays tienen una limitación importante: su tamaño debe definirse de antemano, lo que los hace poco flexibles si necesitamos agregar, eliminar o modificar elementos dinámicamente.

1.1 Alternativa a los Arrays: Colecciones en Java

Para superar estas limitaciones, Java proporciona la API de colecciones en el paquete java.util, que permite administrar grupos de objetos de manera dinámica. Las colecciones ajustan automáticamente su tamaño a medida que se añaden o eliminan elementos, lo que las hace más flexibles que los arrays.

Una colección es una clase, por lo que debemos instanciarla antes de poder agregar elementos. En Java, todas las colecciones se encuentran en el paquete java.util y están organizadas bajo la interfaz Collection<E>, que define los métodos básicos para manejar grupos de objetos.

1.2 Principales tipos de colecciones en Java

1.2.1 List<E> (Lista)

Las listas son colecciones ordenadas que permiten elementos duplicados. Se basan en la interfaz List<E> y tienen una capacidad variable, lo que significa que podemos agregar o eliminar elementos sin preocuparnos por el tamaño. Algunas implementaciones comunes incluyen:

- ArrayList<E>: Basada en un array dinámico, ofrece acceso rápido a elementos, pero las inserciones y eliminaciones pueden ser costosas.
- LinkedList<E>: Basada en una lista doblemente enlazada, es eficiente para inserciones y eliminaciones, pero más lenta en acceso aleatorio.

1.2.2 Queue<E> (Cola - FIFO)

Las colas son colecciones en las que los elementos se insertan en un extremo y se recuperan en el orden en que fueron añadidos (FIFO: First In, First Out). Ejemplo de uso: una cola de tareas.

- LinkedList<E> implementa Queue<E>, permitiendo gestionar elementos en orden de llegada.
- PriorityQueue<E> ordena los elementos según un criterio de prioridad.

1.2.3 Stack<E> (Pila - LIFO)

Las pilas funcionan con la lógica Last In, First Out (LIFO), lo que significa que el último elemento en entrar es el primero en salir. Se usa, por ejemplo, para gestionar la pila de ejecución de un programa.

- Desde Java 1.0 existe la clase Stack<E>, aunque actualmente se recomienda usar Deque<E> con ArrayDeque<E> para una mejor eficiencia.

1.2.4 Map<K, V> (Diccionarios o Tablas Hash)

Un Map<K, V> almacena pares clave-valor, permitiendo acceder rápidamente a los valores mediante su clave. Ejemplo de uso: una agenda de contactos donde los nombres son las claves y los números de teléfono los valores.

- HashMap<K, V>: Implementación basada en tablas hash, proporciona acceso rápido a los elementos.
- TreeMap<K, V>: Mantiene los elementos ordenados según su clave.
- LinkedHashMap<K, V>: Mantiene el orden de inserción de los elementos.

1.2.5 Colecciones Genéricas en Java

En Java, las colecciones utilizan tipos genéricos, lo que permite asegurar que todos los elementos de una colección sean del mismo tipo. Esto evita conversiones innecesarias y errores en tiempo de ejecución. Por ejemplo:

```
List<String> nombres = new ArrayList<>();

nombres.add("Ana");

nombres.add("Carlos");

// nombres.add(10); // Error: Solo se permiten Strings
```

Las colecciones en Java ofrecen una manera flexible y eficiente de administrar grupos de objetos, superando las limitaciones de los arrays tradicionales.

1.3 Uso de Colecciones Genéricas en Java

Se recomienda utilizar colecciones genéricas en Java, ya que proporcionan seguridad de tipos sin necesidad de derivar de una clase base específica ni de implementar métodos adicionales. Al definir una colección con un tipo genérico, se evita la conversión de tipos en tiempo de ejecución y se mejora la legibilidad del código.

Además, las colecciones genéricas suelen ofrecer mejor rendimiento en comparación con las colecciones no genéricas cuando los elementos almacenados son tipos primitivos envueltos en clases wrapper (como Integer en lugar de int). Esto se debe a la optimización del compilador mediante autoboxing y unboxing.

Por ejemplo, en lugar de usar una lista sin tipo:

```
List lista = new ArrayList();  // Lista sin tipo, no recomendado

lista.add("Texto");

lista.add(10);  // Error en tiempo de ejecución si no se controla
```

Se debe usar una colección genérica con tipo definido:

```
List<String> lista = new ArrayList<>();

lista.add("Texto");

// lista.add(10);  // Error en tiempo de compilación
```

1.4 Colecciones Concurrentes en Java

En aplicaciones que requieren acceso simultáneo a estructuras de datos desde múltiples hilos (threads), Java proporciona el paquete java.util.concurrent, que contiene clases optimizadas para concurrencia.

Las colecciones concurrentes deben usarse en lugar de las estructuras de datos estándar (ArrayList, HashMap, etc.) cuando varios hilos acceden a la misma colección. Estas clases están diseñadas para garantizar seguridad en el acceso a los datos sin necesidad de bloquear manualmente.

Algunas colecciones concurrentes importantes en Java incluyen:

- ConcurrentHashMap<K, V> → Similar a HashMap, pero optimizado para múltiples hilos.
- ConcurrentLinkedQueue<E> → Cola sin bloqueos para acceso concurrente.
- ConcurrentLinkedDeque<E> → Implementación de una doble cola concurrente.
- CopyOnWriteArrayList<E> → Versión segura para hilos de ArrayList, útil cuando hay más lecturas que escrituras.
- BlockingQueue<E> → Interfaz para colas bloqueantes, con implementaciones como ArrayBlockingQueue<E> y LinkedBlockingQueue<E>.

La programación concurrente es un tema avanzado que se estudia en niveles posteriores, por lo que en este módulo nos centraremos en el uso de colecciones genéricas estándar para gestionar datos de manera eficiente y segura.

2 Pilas

La clase Stack<E> en Java representa una colección de tipo pila, es decir, el último elemento en entrar será el primero en salir (LIFO - Last In, First Out). Se trata de una estructura de datos de tamaño variable en la que todos los elementos son del mismo tipo (E representa el tipo de los elementos en la pila).

Métodos principales de Stack<E>

- push(E item) → Inserta un elemento en la pila.
- pop() → Extrae y devuelve el último elemento agregado a la pila.
- peek() → Devuelve el último elemento sin extraerlo.
- size() → Devuelve el número de elementos en la pila.
- isEmpty() → Indica si la pila está vacía.

Ejemplo de uso de Stack<E> en Java

Podemos usar una pila para implementar funcionalidades como "deshacer" (undo) y "rehacer" (redo), donde el último cambio realizado será el primero en deshacerse.

```
import java.util.Stack;


public class EjemploPila {
   public static void main(String[] args) {
```

```
        Stack<String> pila = new Stack<>();

        // Agregar elementos a la pila
        pila.push("Primer elemento");
        pila.push("Segundo elemento");
        pila.push("Tercer elemento");

        // Consultar el último elemento sin eliminarlo
        System.out.println("Último elemento (peek): " + pila.peek());

        // Extraer elementos de la pila
        System.out.println("Elemento eliminado (pop): " + pila.pop());

        // Consultar el tamaño de la pila
        System.out.println("Tamaño de la pila: " + pila.size());

        // Verificar si la pila está vacía
        System.out.println("¿Está vacía? " + pila.isEmpty());
    }
}
```

Salida esperada:

```
Último elemento (peek): Tercer elemento
Elemento eliminado (pop): Tercer elemento
Tamaño de la pila: 2
¿Está vacía? false
```

En este ejemplo, primero insertamos elementos en la pila con push(), luego usamos peek() para ver el último elemento sin eliminarlo, y pop() para sacarlo de la pila. Finalmente, verificamos el tamaño y si la pila está vacía.

En Java, aunque Stack<E> es ampliamente utilizada, en muchos casos se recomienda usar Deque<E> con ArrayDeque<E> en lugar de Stack<E>, ya que ofrece un mejor rendimiento:

```
import java.util.Deque;

import java.util.ArrayDeque;

Deque<String> pila = new ArrayDeque<>();
```

EJEMPLO

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Stack;

public class FrmStack extends JFrame {

    private Stack<String> stack;
    private JTextField txtIn, txtOut, txtPeek;
    private JLabel lblCount;
    private JButton btnPush, btnPop;

    public FrmStack() {
        setTitle("Stack Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new GridLayout(5, 2, 5, 5));

        // Inicializar componentes
        stack = new Stack<>();

        add(new JLabel("Enter Text:"));
        txtIn = new JTextField();
        add(txtIn);

        btnPush = new JButton("Push");
        add(btnPush);

        btnPop = new JButton("Pop");
        add(btnPop);

        add(new JLabel("Top of Stack (Peek):"));
        txtPeek = new JTextField();
        txtPeek.setEditable(false);
        add(txtPeek);
```

```
        add(new JLabel("Popped Element:"));
        txtOut = new JTextField();
        txtOut.setEditable(false);
        add(txtOut);
        add(new JLabel("Stack Count:"));
        lblCount = new JLabel("0");
        add(lblCount);

        // Agregar eventos a los botones
        btnPush.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                pushElement();
            }
        });

        btnPop.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                popElement();
            }
        });
        ShowStackData();
        setVisible(true);
    }

    private void ShowStackData() {
        lblCount.setText(String.valueOf(stack.size()));

        if (!stack.isEmpty()) {
            txtPeek.setText(stack.peek());
            btnPop.setEnabled(true);
        } else {
            txtPeek.setText("");
            btnPop.setEnabled(false);
        }
    }
```

```
    private void pushElement() {
        String s = txtIn.getText().trim();
        if (!s.isEmpty()) {
            stack.push(s);
            ShowStackData();
            txtIn.setText("");
        } else {
            JOptionPane.showMessageDialog(this, "Text cannot be empty");
        }
        txtIn.requestFocus();
    }

    private void popElement() {
        if (!stack.isEmpty()) {
            txtOut.setText(stack.pop());
            ShowStackData();
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new FrmStack());
    }
}
```

3 Colas

La clase Queue<E> en Java representa una colección de tipo cola, donde el primer elemento en entrar será el primero en salir (FIFO - First In, First Out). Se trata de una estructura de datos de tamaño variable en la que todos los elementos son del mismo tipo (E representa el tipo de los elementos en la cola).

Métodos principales de Queue<E>

- offer(E e) → Agrega un elemento al final de la cola (equivalente a Enqueue en C#).
- poll() → Extrae y devuelve el primer elemento de la cola (equivalente a Dequeue en C#).
- peek() → Devuelve el primer elemento sin extraerlo.

- size() → Devuelve el número de elementos en la cola.
- isEmpty() → Indica si la cola está vacía.

```
import java.util.LinkedList;
import java.util.Queue;

public class EjemploCola {
   public static void main(String[] args) {
      Queue<String> cola = new LinkedList<>();

      // Agregar elementos a la cola
      cola.offer("Tarea 1");
      cola.offer("Tarea 2");
      cola.offer("Tarea 3");

      // Consultar el primer elemento sin eliminarlo
      System.out.println("Primer elemento (peek): " + cola.peek());

      // Extraer elementos de la cola
      System.out.println("Elemento procesado (poll): " + cola.poll());

      // Consultar el tamaño de la cola
      System.out.println("Tamaño de la cola: " + cola.size());

      // Verificar si la cola está vacía
      System.out.println("¿Está vacía? " + cola.isEmpty());
   }
}
```

Salida Esperada:

```
Primer elemento (peek): Tarea 1

Elemento procesado (poll): Tarea 1

Tamaño de la cola: 2

¿Está vacía? false
```

Ejemplo de pila

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.LinkedList;
import java.util.Queue;

public class FrmQueue extends JFrame {

    private Queue<String> queue;
    private JTextField txtIn, txtOut, txtPeek;
    private JLabel lblCount;
    private JButton btnEnqueue, btnDequeue;

    public FrmQueue() {
        setTitle("Queue Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new GridLayout(5, 2, 5, 5));

        // Inicializar cola
        queue = new LinkedList<>();

        add(new JLabel("Enter Text:"));
        txtIn = new JTextField();
        add(txtIn);

        btnEnqueue = new JButton("Enqueue");
        add(btnEnqueue);

        btnDequeue = new JButton("Dequeue");
        add(btnDequeue);

        add(new JLabel("Front of Queue (Peek):"));
        txtPeek = new JTextField();
        txtPeek.setEditable(false);
        add(txtPeek);

        add(new JLabel("Dequeued Element:"));
        txtOut = new JTextField();
        txtOut.setEditable(false);
        add(txtOut);
```

```
        add(new JLabel("Queue Count:"));
        lblCount = new JLabel("0");
        add(lblCount);

        // Agregar eventos a los botones
        btnEnqueue.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                enqueueElement();
            }
        });

        btnDequeue.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                dequeueElement();
            }
        });

        ShowQueueData();
        setVisible(true);
    }

    private void ShowQueueData() {
        lblCount.setText(String.valueOf(queue.size()));

        if (!queue.isEmpty()) {
            txtPeek.setText(queue.peek());
            btnDequeue.setEnabled(true);
        } else {
            txtPeek.setText("");
            btnDequeue.setEnabled(false);
        }
    }

    private void enqueueElement() {
        String s = txtIn.getText().trim();
        if (!s.isEmpty()) {
            queue.offer(s); // Enqueue en Java
            ShowQueueData();
            txtIn.setText("");
        } else {
            JOptionPane.showMessageDialog(this, "Text cannot be empty");
        }
        txtIn.requestFocus();
    }

    private void dequeueElement() {
        if (!queue.isEmpty()) {
            txtOut.setText(queue.poll()); // Dequeue en Java
            ShowQueueData();
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new FrmQueue());
    }
}
```

4 Listas

En Java, la clase List<E> representa una lista de objetos fuertemente tipados a la que se puede acceder mediante un índice. Proporciona métodos para buscar, ordenar y manipular listas, similar a List<T> en C#.

Características de List<E> en Java:

- Acceso por índice: Podemos obtener, modificar o eliminar elementos por su posición en la lista.
- Capacidad dinámica: A diferencia de los arrays, las listas pueden crecer o reducirse automáticamente.
- Métodos útiles: Buscar (contains()), ordenar (sort()), eliminar (remove()), recorrer (forEach()).

```
import java.util.ArrayList;
import java.util.List;

public class ListaEjemplo {
    public static void main(String[] args) {
        // Crear lista de nombres
        List<String> lista = new ArrayList<>();

        // Añadir elementos (equivalente a Add en C#)
        lista.add("Juan");
        lista.add("María");
        lista.add("Carlos");
        lista.add("Ana");
        System.out.println("Lista inicial: " + lista);

        // Insertar elemento en un índice específico (equivalente a Insert en C#)
        lista.add(2, "Luis");
        System.out.println("Después de insertar 'Luis' en índice 2: " + lista);
```

```
        // Añadir un rango de elementos (equivalente a AddRange en C#)
        List<String> nuevosNombres = List.of("Pedro", "Sofía", "Elena");
        lista.addAll(nuevosNombres);
        System.out.println("Después de añadir un rango de nombres: " + lista);

        // Eliminar elemento por índice (equivalente a RemoveAt en C#)
        lista.remove(1);
        System.out.println("Después de eliminar el índice 1: " + lista);

        // Eliminar por valor (equivalente a Remove en C#)
        lista.remove("Luis");
        System.out.println("Después de eliminar 'Luis': " + lista);

        // Eliminar todos los elementos que cumplan una condición (equivalente a
RemoveAll en C#)
        lista.removeIf(nombre -> nombre.startsWith("P"));
        System.out.println("Después de eliminar nombres que empiezan con 'P': " +
lista);

        // Limpiar la lista (equivalente a Clear en C#)
        lista.clear();
        System.out.println("Lista después de Clear(): " + lista);
    }
}
```

4.1 Buscar y ordenar elementos

Buscar y ordenar elementos en Java

En Java, la clase ArrayList<E> proporciona varios métodos para buscar y ordenar elementos de una lista.

Buscar elementos

1. contains(Object o)

Verifica si un elemento está presente en la lista.

```
lista.contains("Juan");
```

2. indexOf(Object o)

Devuelve el índice de la primera aparición del elemento.

```
lista.indexOf("Carlos");
```

3. lastIndexOf(Object o)

Devuelve el índice de la última aparición del elemento.

```
lista.lastIndexOf("Carlos");
```

4. removeIf(Predicate<? super E> filter)

Elimina todos los elementos que coincidan con un predicado dado (similar a FindAll en C#).

```
lista.removeIf(nombre -> nombre.startsWith("P"));
```

5. stream().filter()

Filtra los elementos de la lista de acuerdo con una condición, similar a Where en C#.

```
lista.stream().filter(nombre -> nombre.startsWith("J")).forEach(System.out::println);
```

6. find()

Aunque Java no tiene un equivalente exacto a Find() en C#, puedes usar el método stream().filter().findFirst() para encontrar el primer elemento que coincida con el predicado.

```
lista.stream().filter(nombre ->
nombre.startsWith("A")).findFirst().ifPresent(System.out::println);
```

Ordenar elementos

1. sort(Comparator<? super E> c)

Ordena la lista usando un comparador.

```
lista.sort(Comparator.naturalOrder());  // Orden ascendente

lista.sort(Comparator.reverseOrder());  // Orden descendente
```

2. stream().sorted()

Puedes ordenar los elementos usando un Stream.

```
lista.stream().sorted().forEach(System.out::println);  // Orden ascendente
```

3. stream().sorted(Comparator.reverseOrder())

Ordenar en orden descendente usando Comparator.

```
lista.stream().sorted(Comparator.reverseOrder()).forEach(System.out::println);  // Orden descendente
```

Ejemplo en Java: Buscar y ordenar elementos

```
import java.util.ArrayList;
import java.util.List;
import java.util.Comparator;

public class BuscarOrdenarLista {
   public static void main(String[] args) {
      // Crear lista de nombres
      List<String> lista = new ArrayList<>();
      lista.add("Juan");
      lista.add("María");
      lista.add("Carlos");
      lista.add("Ana");
      lista.add("Pedro");

      // Buscar si un elemento existe
      boolean existe = lista.contains("Carlos");
      System.out.println("¿Existe Carlos? " + existe);

      // Buscar el índice de un elemento
```

```
        int indice = lista.indexOf("María");
        System.out.println("Índice de María: " + indice);

        // Buscar el último índice de un elemento
        int ultimoIndice = lista.lastIndexOf("Carlos");
        System.out.println("Último índice de Carlos: " + ultimoIndice);

        // Eliminar elementos que comienzan con 'P'
        lista.removeIf(nombre -> nombre.startsWith("P"));
        System.out.println("Lista después de eliminar nombres que empiezan con 'P': " + lista);

        // Ordenar la lista de manera ascendente
        lista.sort(Comparator.naturalOrder());
        System.out.println("Lista ordenada ascendentemente: " + lista);

        // Ordenar la lista de manera descendente
        lista.sort(Comparator.reverseOrder());
        System.out.println("Lista ordenada descendentemente: " + lista);

        // Filtrar y mostrar nombres que empiezan con 'A'
        lista.stream().filter(nombre -> nombre.startsWith("A")).forEach(System.out::println);
    }
}
```

Salida:

```
¿Existe Carlos? true
Índice de María: 1
Último índice de Carlos: 2
Lista después de eliminar nombres que empiezan con 'P': [Juan, María, Carlos, Ana]
Lista ordenada ascendentemente: [Ana, Carlos, Juan, María]
```

```
Lista ordenada descendentemente: [María, Juan, Carlos, Ana]
Ana
Carlos
```

4.2 Funciones de agregación

En Java, las funciones de agregación como Max, Min, y Sum son bastante útiles para trabajar con listas y realizar cálculos o análisis de datos. En Java, estas funciones se encuentran principalmente a través de los Streams. A continuación, te muestro cómo puedes utilizar estas funciones de agregación con una lista de datos.

Funciones de agregación en Java

1. Max: Devuelve el valor máximo de una secuencia de elementos.

```
int max = lista.stream().max(Integer::compare).orElseThrow();
System.out.println("Máximo valor: " + max);
```

2. Min: Devuelve el valor mínimo de una secuencia de elementos.

```
int min = lista.stream().min(Integer::compare).orElseThrow();
System.out.println("Mínimo valor: " + min);
```

3. Sum: Suma los elementos de una secuencia.

```
int sum = lista.stream().mapToInt(Integer::intValue).sum();
System.out.println("Suma de los elementos: " + sum);
```

4. Average: Calcula el promedio de los elementos.

```
double average = lista.stream().mapToInt(Integer::intValue).average().orElse(0);
System.out.println("Promedio: " + average);
```

5. Count: Cuenta el número total de elementos en una secuencia.

```
long count = lista.stream().count();
System.out.println("Total de elementos: " + count);
```

Ejemplo de uso con números enteros

```
java
Copiar
Editar
import java.util.List;
import java.util.Arrays;

public class FuncionesAgregacion {
   public static void main(String[] args) {
      // Crear lista de números
      List<Integer> lista = Arrays.asList(10, 20, 30, 40, 50);

      // Máximo
      int max = lista.stream().max(Integer::compare).orElseThrow();
      System.out.println("Máximo valor: " + max);

      // Mínimo
      int min = lista.stream().min(Integer::compare).orElseThrow();
      System.out.println("Mínimo valor: " + min);

      // Suma
      int sum = lista.stream().mapToInt(Integer::intValue).sum();
      System.out.println("Suma de los elementos: " + sum);

      // Promedio
      double average = lista.stream().mapToInt(Integer::intValue).average().orElse(0);
      System.out.println("Promedio: " + average);

      // Contar elementos
```

```
        long count = lista.stream().count();
        System.out.println("Total de elementos: " + count);
    }
}
```

Salida esperada:

```
Máximo valor: 50
Mínimo valor: 10
Suma de los elementos: 150
Promedio: 30.0
Total de elementos: 5
```

4.3 Comprobar la igualdad

Cómo funciona la comparación de igualdad en Java:

1. Interfaz equals(): En Java, la comparación de igualdad generalmente se maneja mediante el método equals() de la clase Object. Este método debe ser sobrescrito en las clases de objetos cuando necesitamos que la comparación sea lógica (en lugar de comparar las referencias de memoria).

2. Método hashCode(): Además de equals(), es recomendable sobrescribir el método hashCode() cuando se sobrescribe equals(). Esto es especialmente importante cuando se usan colecciones como HashSet o HashMap, que dependen de hashCode() para almacenar y recuperar elementos eficientemente.

Directrices de comparación en Java:

1. Si un tipo T (el tipo de los objetos que estamos almacenando en la colección) sobrescribe el método equals(), el comparador de igualdad utilizará esa implementación personalizada.
2. Si el tipo T no sobrescribe equals(), Java usará la implementación predeterminada de equals() que compara referencias de memoria (es decir, verifica si ambos objetos son exactamente el mismo objeto en memoria).

Ejemplo: Sobrescribiendo equals() en una clase personalizada

Supongamos que tienes una clase Persona y quieres usarla en una colección como una lista.

```
import java.util.ArrayList;
import java.util.List;


class Persona {
    private String nombre;
    private int edad;


    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }


    // Sobrescribir el método equals() para comparar objetos de tipo Persona
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true; // Verifica si son el mismo objeto
        if (obj == null || getClass() != obj.getClass()) return false; // Verifica que sean de la misma clase
        Persona persona = (Persona) obj; // Convierte el objeto a Persona
        return edad == persona.edad && nombre.equals(persona.nombre); // Compara los campos de la clase
    }


    @Override
    public int hashCode() {
        // Es recomendable también sobrescribir hashCode() cuando sobrescribes equals()
        return 31 * nombre.hashCode() + edad;
```

```
    }

    // Getters y setters
    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }
}

public class Ejemplo {
    public static void main(String[] args) {
        List<Persona> lista = new ArrayList<>();

        // Crear algunos objetos de tipo Persona
        Persona p1 = new Persona("Juan", 25);
        Persona p2 = new Persona("Ana", 30);
        Persona p3 = new Persona("Juan", 25); // Misma información que p1

        // Agregar a la lista
        lista.add(p1);
        lista.add(p2);

        // Comprobar si contiene un objeto específico
        System.out.println("Contiene a Juan (25): " + lista.contains(p3)); // true, porque p1 y p3 son iguales según equals()

        // Buscar el índice de un elemento
```

```
        int index = lista.indexOf(p3);
        System.out.println("Índice de Juan (25): " + index); // 0, porque p3 es igual a p1


        // Eliminar un objeto
        lista.remove(p2);
        System.out.println("Lista después de eliminar a Ana: " + lista.size()); // 1, porque p2 fue removido
    }
}
```

5 Tablas Hash

En Java, las tablas hash también son estructuras de datos muy comunes para almacenar pares clave-valor, y su equivalente más directo es la clase HashMap. Al igual que en .NET, la clase HashMap en Java permite asociar claves con valores, y ofrece una gran eficiencia en las operaciones de búsqueda, inserción y eliminación, ya que implementa una función hash interna para acceder a los elementos de manera rápida.

En Java, el HashMap<K, V> ofrece una estructura eficiente para almacenar pares clave-valor y proporciona métodos para agregar, acceder, eliminar y verificar elementos. A diferencia de C#, HashMap permite una mayor flexibilidad porque acepta null como valor y como clave (aunque esto depende de la implementación).

Operaciones principales en HashMap<K, V>:

- Agregar elementos: put(K key, V value)
- Acceder a valores: Usando get(K key)
- Verificar si una clave existe: containsKey(Object key)
- Recuperar valores de manera segura: getOrDefault(K key, V defaultValue)
- Eliminar elementos: remove(Object key)
- Enumerar claves y valores: keySet() y values()

Ejemplo en Java de un HashMap<K, V>:

```
import java.util.HashMap;
import java.util.Map;


public class Main {
    public static void main(String[] args) {
```

```
// Crear un HashMap de claves y valores
Map<String, String> map = new HashMap<>();

// Agregar elementos al HashMap
map.put("John", "123-4567");
map.put("Mary", "234-5678");
map.put("Paul", "345-6789");

// Intentar agregar una clave duplicada (reemplazará el valor)
map.put("John", "555-5555");

// Acceder a un valor usando una clave
System.out.println("John's phone number: " + map.get("John"));

// Usar getOrDefault para evitar null si la clave no existe
String value = map.getOrDefault("Mary", "Not found");
System.out.println("Mary's phone number: " + value);

// Comprobar si una clave existe
if (map.containsKey("Paul")) {
   System.out.println("Paul is in the map");
}

// Eliminar un elemento
map.remove("Paul");

// Enumerar las claves y valores
for (Map.Entry<String, String> entry : map.entrySet()) {
   System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
```

```
    }
  }
}
```

4.1 Listas ordenadas

La clase SortedList<K, V> representa una colección de pares clave-valor que se ordenan por las claves.

De esta forma podremos acceder a los elementos de la lista tanto por clave como por índice, estando los elementos ordenados por la clave.

Propiedades:

- getCount(): Obtiene el número total de elementos en la colección.
- getItem(K key): Obtiene el elemento con la clave especificada.
- setItem(K key, V value): Establece el elemento con la clave especificada.
- getKeys(): Obtiene la lista de claves de la colección.
- getValues(): Obtiene la lista de valores de la colección.

Métodos:

- add(K key, V value): Añade un par clave-valor a la colección.
- remove(K key): Borra el elemento con la clave especificada.
- removeAt(int index): Borra el elemento en el índice especificado.
- containsKey(K key): Chequea si existe la clave especificada.
- containsValue(V value): Chequea si existe el valor especificado.
- clear(): Borra todos los elementos de la colección.
- indexOfKey(K key): Devuelve el índice correspondiente a la clave especificada.
- indexOfValue(V value): Devuelve el índice correspondiente al valor especificado.
- tryGetValue(K key, V[] outValue): Devuelve true y asigna el valor correspondiente a la clave especificada. Si la clave no existe, devuelve false.

En el ejemplo utilizamos una SortedList para guardar las apariciones de las letras que contiene una frase, las letras guardadas aparecerán en orden alfabético. Observamos cómo podemos acceder a claves y valores a través del índice, lo cual no es posible en los diccionarios.

```
import java.util.*;
public class SortedList<K, V> {
   private final TreeMap<K, V> map;
   public SortedList() {
      this.map = new TreeMap<>();
   }

   public int getCount() {
      return map.size();
   }

   public V getItem(K key) {
      return map.get(key);
   }

   public void setItem(K key, V value) {
      map.put(key, value);
   }

   public Set<K> getKeys() {
      return map.keySet();
   }

   public Collection<V> getValues() {
      return map.values();
   }
public void add(K key, V value) {
      map.put(key, value);
   }

   public void remove(K key) {
      map.remove(key);
   }
```

```
public void removeAt(int index) {
    if (index < 0 || index >= map.size()) {
        throw new IndexOutOfBoundsException("Index out of range");
    }
    K key = new ArrayList<>(map.keySet()).get(index);
    map.remove(key);
}

public boolean containsKey(K key) {
    return map.containsKey(key);
}

public boolean containsValue(V value) {
    return map.containsValue(value);
}

public void clear() {
    map.clear();
}

public int indexOfKey(K key) {
    List<K> keys = new ArrayList<>(map.keySet());
    return keys.indexOf(key);
}
public int indexOfValue(V value) {
    List<V> values = new ArrayList<>(map.values());
    return values.indexOf(value);
}
public boolean tryGetValue(K key, V[] outValue) {
    if (map.containsKey(key)) {
        outValue[0] = map.get(key);
        return true;
    }
    return false;
}
```

```
    public static void main(String[] args) {
        SortedList<Character, Integer> letterCount = new SortedList<>();
        String phrase = "hello world";
        for (char c : phrase.toCharArray()) {
            if (Character.isLetter(c)) {
                letterCount.setItem(c, letterCount.getItem(c) != null ? letterCount.getItem(c) + 1 : 1);
            }
        }
        System.out.println("Keys: " + letterCount.getKeys());
        System.out.println("Values: " + letterCount.getValues());
    }
}
```

6 Algoritmos de Ordenación

Cuando estudiamos los arrays en el primer trimestre vimos el algoritmo de ordenación de la ***burbuja*** que es uno de los menos eficientes. En computación resulta clave ordenar los datos que tenemos y vemos que las clases disponibles en las APIs de los lenguajes implementan métodos de ordenación que nos facilitan esta operación enormemente.

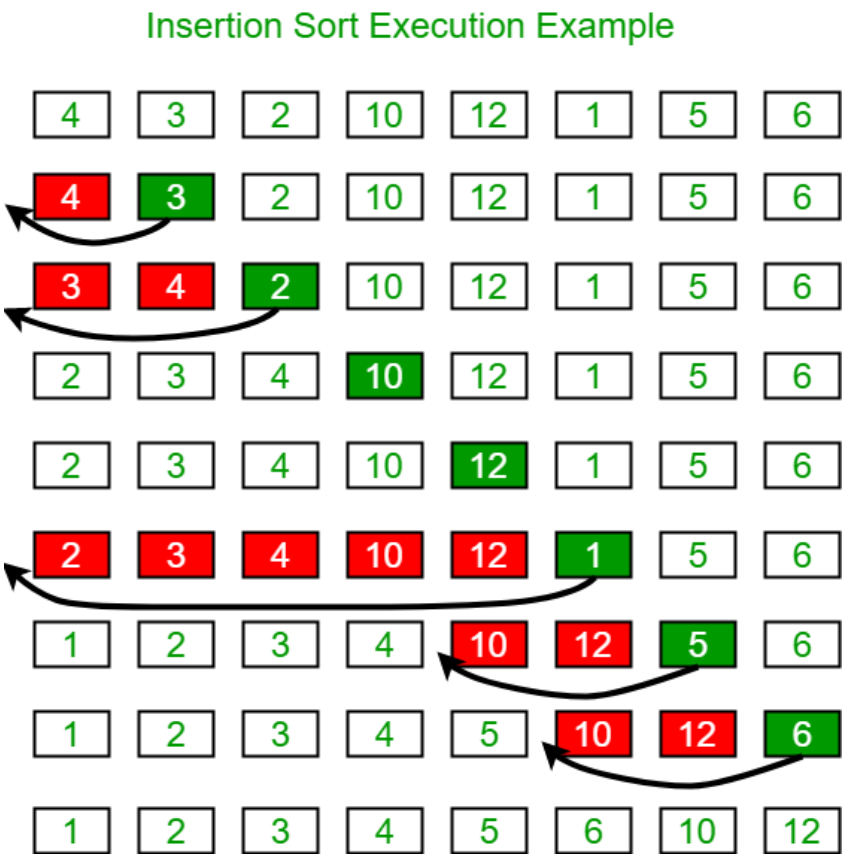
Dada la relevancia de los algoritmos de ordenación en la computación vamos a estudiar dos algoritmos más; se trata del algoritmo de inserción directa o ***Insertion Sort*** y del algoritmo ***Quick Sort.***

4.2 Inserción directa

Se trata de un algoritmo simple de ordenación que funciona de manera similar a como ordenaríamos una baraja de cartas.

Pseudocódigo

```
// Ordenar un array por inserción directa
i ← 1
mientras i < longitud(A)
    j ← i
    mientras j > 0 y A[j-1] > A[j]
        intercambiar A[j] y A[j-1]
        j ← j - 1
    fin mientras
    i ← i + 1
fin mientras
```



4.3 Quick Sort

El algoritmo *Quick Sort* es un algoritmo de tipo *Divide y Vencerás*. Se elige un elemento como pivote y se divide el array dado por pivote seleccionado. Se mueven los elementos menores al pivote a la izquierda del pivote y los elementos mayores a la derecha del pivote. Se repite el proceso para los intervalos resultantes a izquierda y derecha del pivote de manera recursiva.

Hay muchas versiones diferentes de *QuickSort* que seleccionan pivote de diferentes maneras.

- Elegir el punto medio del intervalo como pivote.
- Elegir el primer elemento como pivote.
- Elegir el último elemento como pivote.
- Elegir aleatoriamente un elemento del intervalo como pivote.

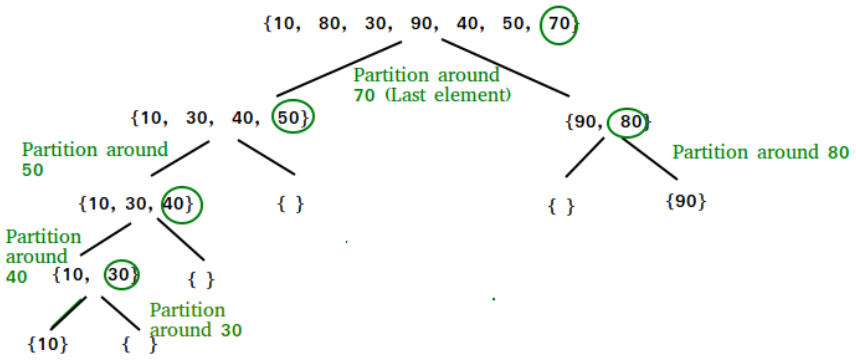
El proceso clave en el algoritmo es la partición. El objetivo de las particiones es, dado un array y un elemento x del array como pivote, colocar x en su posición correcta en una matriz ordenada y colocar todos los elementos más pequeños que el pivote antes de éste y poner todos los elementos mayores después del pivote.

Pseudocódigo para la función recursiva *QuickSort*:

```
/ * bajo -> Índice inicial, alto -> Índice final * /
QuickSort (arr [], bajo, alto)
{
    si (bajo <alto)
```

```
    {
        / * pi es el índice de partición, arr [pivote] ahora
           en el lugar correcto * /
        pivote = Particion (arr, bajo, alto);

        QuickSort (arr, bajo, pivote - 1); // Antes de pivote
        QuickSort (arr, pivote + 1, alto); // Después de pivote
    }
}
```



Algoritmo de partición:

Puede haber muchas formas de hacer una partición. La lógica es simple, comenzamos desde el elemento más a la izquierda y hacemos un seguimiento del índice de elementos más pequeños (o iguales a) como i. Al recorrer, si encontramos un elemento más pequeño, intercambiamos el elemento actual con arr [i]. De lo contrario, ignoramos el elemento actual.

```
/ * Esta función toma el último elemento como pivote, colocar
   el elemento pivote en su posición correcta en orden
   poniendo todos los elementos menores que el pivote su izquierda
   y todos los elementos mayores a la derecha de pivote * /
Particion (arr [], bajo, alto)
{
    // pivote (Elemento que se colocará en la posición correcta)
    pivote = arr[alto];

    i = (bajo - 1) // Índice de elemento más pequeño

    para (j = bajo; j <= alto - 1; j ++)
    {
        // Si el elemento actual es más pequeño que el pivote
        si (arr [j] <pivote)
        {
            i ++; // índice de incremento del elemento más pequeño
            intercambiar arr [i] y arr [j]
        }
    }
    intercambiar arr [i + 1] y arr [alto])
    devolver (i + 1) //pivote
}
```

7 Stream Api Collections Api y JPA Criteria API

Java ofrece varias herramientas para manipular datos, dependiendo del origen de la información. Vamos a revisar cómo trabajar con colecciones, bases de datos y XML de manera eficiente.

Consultas sobre Colecciones en Java → Stream API

Cuando trabajamos con listas, arreglos u otras estructuras de datos en memoria, Streams API nos permite realizar filtros, transformaciones y agrupaciones de forma declarativa.

Ejemplo: Filtrar una lista de nombres que comiencen con "A"

```
 import java.util.*;
import java.util.stream.Collectors;


public class StreamExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Ana", "Juan", "Carlos", "Beatriz", "Pedro", "Ana");


        // Filtrar nombres que comienzan con 'A'
        List<String> filteredNames = names.stream()
                                .filter(name -> name.startsWith("A"))
                                .collect(Collectors.toList());


        System.out.println("Nombres que comienzan con 'A': " + filteredNames);
    }
}
```

Ejemplo de agrupación de datos: Contar la cantidad de veces que aparece cada nombre en una lista

```
import java.util.*;
import java.util.stream.Collectors;
public class CountNames {
   public static void main(String[] args) {
      List<String> names = Arrays.asList("Ana", "Juan", "Carlos", "Beatriz", "Pedro", "Ana");
      // Contar la cantidad de veces que aparece cada nombre
      Map<String, Long> nameCounts = names.stream()
                                 .collect(Collectors.groupingBy(name -> name, Collectors.counting()));

      System.out.println("Frecuencia de nombres: " + nameCounts);
   }
}
```

Consultas sobre Bases de Datos → JDBC y JPA

Para consultar bases de datos en Java, podemos utilizar JDBC o JPA (Java Persistence API).

Ejemplo de consulta SQL con JDBC

```
import java.sql.*;
public class DatabaseQuery {
   public static void main(String[] args) {
      String url = "jdbc:mysql://localhost:3306/mi_base";
      String user = "root";
      String password = "password";
      try (Connection conn = DriverManager.getConnection(url, user, password);
          Statement stmt = conn.createStatement();
          ResultSet rs = stmt.executeQuery("SELECT nombre FROM usuarios WHERE edad > 18")) {
         while (rs.next()) {
            System.out.println("Nombre: " + rs.getString("nombre"));
         }
      } catch (SQLException e) {
         e.printStackTrace();
}}}
```

Ejemplo con JPA (Hibernate) → Consulta de usuarios mayores de 18 años

```
import javax.persistence.*;
import java.util.List;

@Entity
class Usuario {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String nombre;
   private int edad;
}

public class JpaExample {
   public static void main(String[] args) {
      EntityManagerFactory emf = Persistence.createEntityManagerFactory("mi_persistencia");
      EntityManager em = emf.createEntityManager();

      List<Usuario> usuarios = em.createQuery("SELECT u FROM Usuario u WHERE u.edad > 18", Usuario.class)
                          .getResultList();

      usuarios.forEach(u -> System.out.println("Nombre: " + u.getNombre()));

      em.close();
      emf.close();
   }
}
```

Consultas sobre XML → XPath y JAXB

Para trabajar con XML en Java, podemos usar XPath (para consultas) y JAXB (para convertir XML a objetos Java).

Ejemplo con XPath: Buscar nodos en un XML

Supongamos que tenemos este archivo usuarios.xml:

```
<usuarios>
  <usuario>
    <nombre>Ana</nombre>
    <edad>25</edad>
  </usuario>
  <usuario>
    <nombre>Juan</nombre>
    <edad>30</edad>
  </usuario>
</usuarios>
```

Podemos extraer todos los nombres con XPath:

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.*;
import javax.xml.xpath.*;
public class XPathExample {
  public static void main(String[] args) throws Exception {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document doc = builder.parse("usuarios.xml");

    XPathFactory xPathFactory = XPathFactory.newInstance();
    XPath xpath = xPathFactory.newXPath();
    NodeList nodes = (NodeList) xpath.evaluate("/usuarios/usuario/nombre", doc, XPathConstants.NODESET);

    for (int i = 0; i < nodes.getLength(); i++) {
      System.out.println("Nombre: " + nodes.item(i).getTextContent());
    }
  }
}
```

Ejemplo con JAXB: Convertir XML a objetos Java

Si queremos convertir el XML a objetos Java automáticamente, podemos usar **JAXB**:

```
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;
import java.io.File;
import java.util.List;
@XmlRootElement(name = "usuarios")
class Usuarios {
   private List<Usuario> usuario;
   @XmlElement(name = "usuario")
   public List<Usuario> getUsuario() {
      return usuario;
}}
@XmlRootElement
class Usuario {
   private String nombre;
   private int edad;
   @XmlElement
   public String getNombre() {
      return nombre;
   }
   @XmlElement
   public int getEdad() {
      return edad;
}}
public class JAXBExample {
   public static void main(String[] args) throws Exception {
      JAXBContext context = JAXBContext.newInstance(Usuarios.class);
      Unmarshaller unmarshaller = context.createUnmarshaller();
      Usuarios usuarios = (Usuarios) unmarshaller.unmarshal(new File("usuarios.xml"));
      usuarios.getUsuario().forEach(u -> System.out.println("Nombre: " + u.getNombre()));
   }}
```