Del 8 al 17 caracteristicas de Java que debe conocer

Java 8 marcó un cambio radical al introducir **programación funcional** y **expresiones Lambda**, permitiendo escribir código más **conciso**. Por ejemplo, en lugar de usar una clase anónima para ordenar una lista, se puede utilizar una expresión Lambda:

```
// Antes de Java 8
Collections.sort(nombres, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.length() - o2.length();
    }
});
```

```
// Con Java 8 (Lambda)
Collections.sort(nombres, (x, y) -> x.length() - y.length());
```

Otra característica clave de Java 8 es el uso de **referencias a métodos** y la clase **Optional** para evitar excepciones de tipo **`NullPointerException`**:

```
// Referencia a método en Java 8
names.sort(Utilities::compare);

// Uso de Optional para manejar valores nulos
Optional<String> aleron = Optional.empty();
String valorSeguro = aleron.orElse("sin aleron");
```

A partir de **Java 9**, se introdujo el **sistema de módulos** (**Project Jigsaw**) para modularizar el JDK. El sistema de módulos (**`module-info.java`**) mejora la encapsulación y el método **`stream()`** en **`Optional`** convierten colecciones de valores opcionales en flujos. También se introdujo el método **`ifPresentOrElse()`** para ejecutar lógica tanto si el valor existe como si está vacío

Java 10 añadió la inferencia de tipos con la palabra clave **`var`**. La palabra clave **`var`** para la inferencia de tipos locales, permite declarar variables sin especificar su tipo explícitamente si el compilador puede deducirlo, como en **`var nombres = List.of(...)`**.

```
// Java 10+
var lista = new ArrayList<String>();
```

Java 11, versión **LTS**, incluyó mejoras en la clase **`String`** con el método **`isBlank()`**, nuevas APIs para archivos (**`Files.readString()`**), y un nuevo Cliente HTTP asíncrono y síncrono que reemplaza a la antigua implementación.

Java 14-16: **Patrones y Registros**. Las versiones posteriores introdujeron las expresiones **`switch`** con flechas (**->**), los **`Text Blocks`** para cadenas multilínea, las clases **`Record`** para datos inmutables simples y el **`Pattern Matching`** preliminar para **`instanceof`**, simplificando la verificación de tipos.

En versiones más recientes como **Java 17**, se estandarizaron las **expresiones Switch** mejoradas y las **clases selladas** para controlar la herencia:

```
// Java 17+ Switch con patrón
static String formato(Object o) {
    return switch (o) {
        case Integer i -> String.format("int %d", i);
        case String s && s.length() > 3 -> "Cadena larga";
        default -> o.toString();
    };
}

// Java 17+ Clases selladas
sealed interface Figura permits Circulo, Rectangulo {}
```

Sentencias e instrucciones de Java 8 en adelante que hacen la diferencia

Las sentencias y características que marcan la diferencia en **Java 8** y versiones posteriores se centran en la **programación funcional**, **la concurrencia** y la **simplificación del código**.

Java 8 y 11: Funcionalidad y Concurrencia

- **Expresiones Lambda**: Permiten escribir código funcional anónimo, reduciendo la complejidad y facilitando el uso de la **Stream API** para filtrar, transformar y extraer datos sin modificar la fuente original.
- **Stream API**: Ofrece un procesamiento declarativo de colecciones que evalúa el código solo cuando es necesario.
- **Método `forEach()`**: Se define en la interfaz **`Iterable`** y permite iterar sobre colecciones combinado con lambdas.
- **Métodos predeterminados (`default`)**: Permiten agregar métodos no abstractos con cuerpo a las interfaces.
- **Cliente HTTP**: Introducido en Java 11, proporciona una API estándar moderna para consumir REST APIs con soporte para HTTP/2, WebSockets y peticiones asíncronas.

Java 10 y 17: Sintaxis y Legibilidad

- **Inferencia de tipo (`var`)**: En Java 10, la palabra clave *var* permite definir variables locales sin especificar el tipo explícito, manteniendo la seguridad de tipos estáticos.
- **Switch mejorado**: Java 17 estandarizó un estilo de **`switch`** más declarativo y legible que evita el uso repetido de **`if/else`** y el operador **`instanceof`**, permitiendo patrones más complejos y retorno directo de valores.

Resumen de diferencias clave

Versión	Características Distintivas
Java 8	Lambdas, Stream API, **`forEach`**, métodos **`default`**, Fecha/Hora API
Java 10	Inferencia de tipo con **`var`**
Java 11	Cliente HTTP estandarizado, ejecución desde archivo fuente único.
Java 17	**`switch`** mejorado (expresiones), patrones con **`instanceof`**, UTF-8 por defecto

Interfaces del paquete util de Java 8 y posteriores

En Java 8 y posteriores, el paquete **`java.util`** contiene interfaces fundamentales para la gestión de colecciones y programación funcional. Las interfaces de colecciones principales, heredadas de versiones anteriores pero esenciales, son **`Collection`**, **`List`**, **`Set`**, **`Map`** (aunque técnicamente una jerarquía separada, suele agruparse funcionalmente), **`SortedSet`**, **`SortedMap`**, **`Iterator`**, **`Comparator`** e **`Iterable`**.

Además, Java 8 introdujo el paquete **`java.util.function`** para soportar la programación funcional, definiendo interfaces clave como:

- **`Predicate<T>`**: Representa una función booleana de un argumento.
- **`Function<T, R>`**: Representa una función que acepta un argumento y produce un resultado.
- **`Consumer<T>`**: Representa una operación que acepta un argumento de entrada y no devuelve resultado.
- **`Supplier<T>`**: Provee un resultado de una fuente, sin argumentos.
- **`BiFunction<T, U, R>`**: Operación que acepta dos argumentos y produce un resultado.
- **`BiConsumer<T, U>`**: Operación que acepta dos argumentos de entrada.
- **`BiPredicate<T, U>`**: Predicado de dos argumentos.
- **`UnaryOperator<T>`**: Operación unaria que produce un resultado del mismo tipo que su entrada.
- **`BinaryOperator<T>`**: Operación binaria que produce un resultado del mismo tipo que sus argumentos.

Estas interfaces, anotadas con **`@FunctionalInterface`**, permiten el uso de expresiones Lambda y referencias a métodos, facilitando el procesamiento de datos y la escritura de código más conciso y legible.

El paquete `Java.util`

El paquete **`java.util`** es uno de los más utilizados en Java, ya que contiene clases esenciales para trabajar con **colecciones** (como **`ArrayList`**, **`HashMap`**, **`HashSet`**), **fechas y tiempo** (como **`Date`**, **`Calendar`** o las clases del paquete **`java.time`**), y **utilidades del sistema** (como **`Scanner`** para entrada de datos o **`Random`**).

Para usar sus componentes, es necesario importarlo en el código mediante la palabra clave **`import`**. Existen dos formas principales de hacerlo:

- **Importar todo el paquete**: Se utiliza **`import java.util.*;`** para cargar todas las clases disponibles en **`java.util`** automáticamente.
- **Importar clases específicas**: Pueden ser listadas de manera individual, por ejemplo, **`import java.util.ArrayList;`** y **`import java.util.Date;`**, lo cual es preferible en proyectos grandes para mejorar la legibilidad y evitar conflictos de nombres.

Este paquete forma parte de la biblioteca estándar de Java y facilita operaciones complejas sin necesidad de implementar la lógica desde cero.

Ejemplos de uso del paquete `java.util`

El paquete **`java.util`** es una biblioteca estándar de Java que proporciona clases para el manejo de estructuras de datos, fechas, tiempo, números aleatorios y utilidades de entrada/salida. Para usar sus clases, es necesario **importar** el paquete completo con **`import java.util.*;`** o importar clases específicas.

A continuación se presentan ejemplos prácticos de sus componentes más comunes:

1. Estructuras de Colecciones (Listas y Conjuntos)

La clase **`ArrayList`** permite crear listas dinámicas, mientras que **`HashSet`** almacena elementos únicos.

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

public class EjemploColecciones {
    public static void main(String[] args) {
        // Lista ordenada con duplicados permitidos
        List<String> dias = new ArrayList<>();
        dias.add("Lunes");
        dias.add("Martes");
        dias.add("Lunes"); // Duplicado permitido

        // Conjunto sin orden ni duplicados
        HashSet<String> idiomas = new HashSet<>();
        idiomas.add("Español");
        idiomas.add("Inglés");
        idiomas.add("Español"); // No se añade por ser duplicado

        System.out.println("Lista: " + dias);
        System.out.println("Conjunto: " + idiomas);
    }
}
```

2. Fechas y Calendars

La clase **`Date`** y **`Calendar`** se utilizan para obtener la fecha y hora actual.

```
import java.util.Date;
import java.util.Calendar;

public class EjemploFechas {
    public static void main(String[] args) {
        Date hoy = new Date();
        System.out.println("Fecha actual: " + hoy);

        Calendar calendario = Calendar.getInstance();
        System.out.println("Año: " + calendario.get(Calendar.YEAR));
        System.out.println("Mes: " + calendario.get(Calendar.MONTH)); // 0-11
    }
}
```

3. Entrada de Usuario (Scanner)

La clase **`Scanner`** facilita la lectura de datos desde la consola.

```
import java.util.Scanner;

public class EjemploScanner {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Ingrese su nombre: ");
        String nombre = scanner.nextLine();
        System.out.print("Ingrese su edad: ");
        int edad = scanner.nextInt();

        System.out.println("Hola " + nombre + ", tienes " + edad + " años.");
    }
}
```

4. Números Aleatorios

La clase **`Random`** genera números pseudoaleatorios.

```
import java.util.Random;

public class EjemploRandom {
    public static void main(String[] args) {
        Random random = new Random();

        // Número entero aleatorio entre 0 y 99
        int numero = random.nextInt(100);
        System.out.println("Número aleatorio: " + numero);

        // Booleano aleatorio
        boolean aleatorio = random.nextBoolean();
        System.out.println("Booleano aleatorio: " + aleatorio);
    }
}
```

Ejemplo de captura y ordenamiento de datos

Clasificador de Nombres

```
import java.util.Scanner;
import java.util.Set;
import java.util.TreeSet;

public class ClasificadorNombres {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // TreeSet garantiza el orden alfabético y la eliminación de duplicados
        Set<String> nombresUnicos = new TreeSet<>();

        int totalLeidos = 0;
        String centinela = "FINDATOS";

        System.out.println("=== Registro de Nombres ===");
        System.out.println("Introduce los nombres uno a uno. Para terminar, escribe: "
                           + centinela);
        System.out.println("------------------------------------------------------------
                           --------");

        while (true) {
            System.out.print("Ingrese un nombre: ");
            String entrada = scanner.nextLine();

            // Limpiamos espacios y convertimos a mayúsculas para estandarizar
            String nombreProcesado = entrada.trim().toUpperCase();

            // Verificamos si se ha introducido el centinela
            if (nombreProcesado.equals(centinela)) {
                break;
            }

            // Ignorar entradas vacías si el usuario presiona Enter por error
            if (nombreProcesado.isEmpty()) {
                continue;
            }

            // Contabilizamos cada nombre válido que se intentó ingresar
            totalLeidos++;

            // Se añade al Set (si ya existe, no se duplicará)
            nombresUnicos.add(nombreProcesado);
        }

        System.out.println("\n========================================");
        System.out.println("===          LISTA FILTRADA           ===");
        System.out.println("========================================");

        // Imprimir los nombres (ya ordenados y sin duplicados)
        // Usamos una referencia a método de Java 8 para un código más limpio
        nombresUnicos.forEach(System.out::println);

        System.out.println("----------------------------------------");
        System.out.println("Total de nombres leídos (excluyendo centinela): "
                           + totalLeidos);
        System.out.println("Total de nombres impresos (sin duplicados)  : "
                           + nombresUnicos.size());
        System.out.println("========================================");

        scanner.close();
    }
}
```

Tenga en cuenta:

- **Tratamiento preventivo**: Al aplicar **`.trim().toUpperCase()`** antes de la validación, nos aseguramos de que si el usuario escribe "**findatos** ", "**FinDatos**" o "**FINDATOS**", el programa lo reconozca inmediatamente como el cierre.
- **Eficiencia de `TreeSet`**: La inserción y ordenación en un **TreeSet** tiene un costo de tiempo de $O(\log n)$, lo cual es ideal para listas en memoria.
- **Métricas exactas**: La variable totalLeidos cuenta todas las entradas válidas procesadas, mientras que **`nombresUnicos.size()`** nos da el total neto final tras el filtro del contenedor.

Clasificador de Nombres con Stream

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class ClasificadorNombresStream {

    public static void main(String[] args) {
        // 1. Simulación de la lista precargada
        // (incluyendo duplicados, minúsculas y el centinela)
        List<String> listaOriginal = Arrays.asList(
            " Willy", "Ana", "pedro", "Ana", "WILLY", "Carlos", "FinDatos", "pedro", "Juan"
        );

        String centinela = "FINDATOS";

        // 2. Procesamiento de la lista con Streams
        List<String> nombresProcesados = listaOriginal.stream()
                // Limpiamos espacios y pasamos a mayúsculas
                .map(nombre -> nombre.trim().toUpperCase())
                // Truncamos el flujo si aparece el centinela
                // (Java 9+ takeWhile, o filtrado en Java 8)
                .filter(nombre -> !nombre.equals(centinela))
                // Eliminamos duplicados/triplicados
                .distinct()
                // Ordenamos alfabéticamente
                .sorted()
                // Recolectamos el resultado en una nueva lista
                .collect(Collectors.toList());

        // 3. Cálculo de las métricas requeridas
        // Para los leídos, contamos cuántos elementos válidos había originalmente
        // antes de eliminar duplicados
        long totalLeidos = listaOriginal.stream()
                .map(nombre -> nombre.trim().toUpperCase())
                .filter(nombre -> !nombre.equals(centinela))
                .count();

        // 4. Salida de resultados
        System.out.println("=== LISTA PROCESADA ALFABÉTICAMENTE ===");
        nombresProcesados.forEach(System.out::println);

        System.out.println("\n========================================");
        System.out.println("Nombres válidos leídos de la lista: "
                           + totalLeidos);
        System.out.println("Nombres impresos (únicos)         : "
                           + nombresProcesados.size());
        System.out.println("========================================");
    }
}
```

Tenga en cuenta utilizado **Stream**:

- **`.map(nombre ->nombre.trim().toUpperCase())`**: Transforma cada elemento del flujo para asegurar la homogeneidad.
- **`.filter(nombre ->!nombre.equals(centinela))`**: Descarta el centinela en caso de que venga embebido en la lista precargada.
- **`.distinct()`**: Es el operador de *cortocircuito* intermedio que elimina cualquier duplicado basándose en el método **`.equals()`** de la cadena (que al estar en mayúsculas funciona de forma exacta).
- **`.sorted()`**: Ordena el flujo según el orden natural de los **String** (alfabético).
- **`.collect(Collectors.toList())`**: Cierra el *pipeline* del **Stream**, materializando los elementos en una colección utilizable.

Clasificador de Nombres con Stream leídos desde el Teclado

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.stream.Collectors;

public class ClasificadorNombresTecladoStream {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Esta será nuestra lista precargada desde el teclado
        List<String> listaOriginal = new ArrayList<>();

        String centinela = "FINDATOS";

        System.out.println("=== Registro de Nombres ===");
        System.out.println("Introduce los nombres uno a uno. Para terminar, escribe: "
                           + centinela);
        System.out.println("------------------------------------------------------------
                            --------");

        // 1. Fase de lectura y carga de datos
        while (true) {
            System.out.print("Ingrese un nombre: ");
            String entrada = scanner.nextLine();

            // Evaluamos el centinela de inmediato (ignorando mayúsculas/minúsculas
            // y espacios)
            if (entrada.trim().toUpperCase().equals(centinela)) {
                break;
            }

            // Ignoramos líneas vacías para no ensuciar la lista
            if (entrada.trim().isEmpty()) {
                continue;
            }

            // Agregamos a la lista tal y como lo ingresó el usuario
            listaOriginal.add(entrada);
        }

        // 2. Fase de procesamiento utilizando la API de Streams
        List<String> nombresProcesados = listaOriginal.stream()
                // Limpieza y estandarización a mayúsculas
                .map(nombre -> nombre.trim().toUpperCase())
```

```
                // Eliminación de duplicados y triplicados
                .distinct()
                // Ordenación alfabética
                .sorted()
                // Recolección en la lista final
                .collect(Collectors.toList());

        // 3. Salida de resultados y métricas
        System.out.println("\n========================================");
        System.out.println("===          LISTA FILTRADA           ===");
        System.out.println("========================================");

        // Impresión analítica
        nombresProcesados.forEach(System.out::println);

        System.out.println("----------------------------------------");
        // Al haber filtrado el centinela y las líneas vacías en la lectura,
        // el tamaño de listaOriginal es exactamente el total de nombres válidos leídos.
        System.out.println("Total de nombres leídos (válidos): " + listaOriginal.size());
        System.out.println("Total de nombres impresos (únicos): " + nombresProcesados.size());
        System.out.println("========================================");

        scanner.close();
    }
}
```

Tenga en cuenta:

- **Separación de responsabilidades**: El bucle **while** se encarga estrictamente de la I/O (Entrada/Salida) y de cortar el flujo con el centinela. La API de Streams se encarga del procesamiento de datos puro.
- **Métricas exactas a costo cero**: Como no se guarda el *centinela* ni los **Enter** vacíos en **`listaOriginal`**, **`listaOriginal.size()`** te da el número exacto de nombres leídos en un tiempo constante $O(1)$, evitando tener que volver a recorrer el **Stream** para contarlos.
- **Evolución del código**: Si el día de mañana la lista ya no viene del teclado sino de un archivo de texto o de una base de datos, el bloque de **Streams** (Fase 2) se mantendrá exactamente igual, cumpliendo con el principio de diseño de bajo acoplamiento.

Una App de Nombres con interfaz GUI

```
import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class AppNombresGUI {

    // Lista en memoria que actuará como nuestro canal de comunicación entre interfaces
    private final List<String> listaPrecaragada = new ArrayList<>();

    public static void main(String[] args) {
        // Aseguramos que la GUI se ejecute en el hilo de despacho de eventos de Swing
        SwingUtilities.invokeLater(() -> new AppNombresGUI().crearVentanaIngreso());
    }
```

```
    /**
     * PRIMERA INTERFAZ: Captura de datos
     */
    private void crearVentanaIngreso() {
        JFrame frameIngreso = new JFrame("Registro de Nombres");
        frameIngreso.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frameIngreso.setSize(400, 250);
        frameIngreso.setLocationRelativeTo(null); // Centrar en pantalla
        frameIngreso.setLayout(new BorderLayout(10, 10));

        // Panel Superior: Instrucciones
        JLabel lblInstrucciones = new JLabel("Ingrese los nombres uno a uno:",
        SwingConstants.CENTER);
        lblInstrucciones.setFont(new Font("Arial", Font.BOLD, 14));
        frameIngreso.add(lblInstrucciones, BorderLayout.NORTH);

        // Panel Central: Input y botón de agregar
        JPanel panelCentral = new JPanel(new FlowLayout());
        JTextField txtNombre = new JTextField(15);
        txtNombre.setFont(new Font("Arial", Font.PLAIN, 14));
        JButton btnAgregar = new JButton("Agregar");

        panelCentral.add(txtNombre);
        panelCentral.add(btnAgregar);
        frameIngreso.add(panelCentral, BorderLayout.CENTER);

        // Panel Inferior: Botón de finalización y contador en vivo
        JPanel panelInferior = new JPanel(new GridLayout(2, 1, 5, 5));
        JLabel lblContador = new JLabel("Nombres registrados: 0", SwingConstants.CENTER);
        JButton btnFinalizar = new JButton("Finalizar Entrada de Datos");
        btnFinalizar.setBackground(new Color(46, 204, 113)); // Verde sutil
        btnFinalizar.setForeground(Color.WHITE);
        btnFinalizar.setFont(new Font("Arial", Font.BOLD, 12));

        panelInferior.add(lblContador);
        panelInferior.add(btnFinalizar);
        frameIngreso.add(panelInferior, BorderLayout.SOUTH);

        // --- LÓGICA DE EVENTOS (Primera Ventana) ---

        // Acción para agregar nombre (Compartida por botón y tecla Enter)
        Runnable agregarAccion = () -> {
            String entrada = txtNombre.getText().trim();
            if (!entrada.isEmpty()) {
                listaPrecaragada.add(entrada);
                lblContador.setText("Nombres registrados: " + listaPrecaragada.size());
                txtNombre.setText(""); // Limpiar campo
                txtNombre.requestFocus();
            }
        };

        btnAgregar.addActionListener(e -> agregarAccion.run());
        txtNombre.addActionListener(e -> agregarAccion.run()); // Soporte para presionar 'Enter'

        // Acción al finalizar: Cerramos esta ventana y abrimos la de resultados
        btnFinalizar.addActionListener(e -> {
            frameIngreso.dispose(); // Destruye la primera ventana liberando recursos
            crearVentanaResultados(); // Invoca la segunda interfaz
        });

        frameIngreso.setVisible(true);
    }

    /**
     * SEGUNDA INTERFAZ: Procesamiento (Streams) y Despliegue
     */
    private void crearVentanaResultados() {
        // 1. Procesamiento exacto mediante la API de Streams sobre la lista precargada
        List<String> nombresProcesados = listaPrecaragada.stream()
                .map(nombre -> nombre.trim().toUpperCase())
                .distinct()
```

```
                .sorted()
                .collect(Collectors.toList());

        // 2. Configuración de la ventana de resultados
        JFrame frameResultados = new JFrame("Resultados del Análisis");
        frameResultados.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frameResultados.setSize(400, 400);
        frameResultados.setLocationRelativeTo(null);
        frameResultados.setLayout(new BorderLayout(10, 10));

        // Panel Superior: Título
        JLabel lblTitulo = new JLabel("Lista Alfabética (Sin Duplicados)",
        SwingConstants.CENTER);
        lblTitulo.setFont(new Font("Arial", Font.BOLD, 14));
        frameResultados.add(lblTitulo, BorderLayout.NORTH);

        // Panel Central: Área de texto con scroll para mostrar los nombres
        DefaultListModel<String> listModel = new DefaultListModel<>();
        nombresProcesados.forEach(listModel::addElement);
        JList<String> jListNombres = new JList<>(listModel);
        jListNombres.setFont(new Font("Monospaced", Font.PLAIN, 14));
        JScrollPane scrollPane = new JScrollPane(jListNombres);
        frameResultados.add(scrollPane, BorderLayout.CENTER);

        // Panel Inferior: Estadísticas y botón de salida
        JPanel panelInferior = new JPanel(new BorderLayout(5, 5));

        // Subpanel para las métricas requeridas
        JPanel panelMetricas = new JPanel(new GridLayout(2, 1));
        JLabel lblLeidos = new JLabel(" Total nombres leídos: "
                                    + listaPrecaragada.size());
        JLabel lblImpresos = new JLabel(" Total nombres únicos impresos: "
                                    + nombresProcesados.size());
        lblLeidos.setFont(new Font("Arial", Font.PLAIN, 12));
        lblImpresos.setFont(new Font("Arial", Font.BOLD, 12));
        panelMetricas.add(lblLeidos);
        panelMetricas.add(lblImpresos);

        JButton btnTerminar = new JButton("Terminar Aplicación");
        btnTerminar.setBackground(new Color(231, 76, 60)); // Rojo sutil
        btnTerminar.setForeground(Color.WHITE);
        btnTerminar.setFont(new Font("Arial", Font.BOLD, 13));

        panelInferior.add(panelMetricas, BorderLayout.NORTH);
        panelInferior.add(btnTerminar, BorderLayout.SOUTH);
        frameResultados.add(panelInferior, BorderLayout.SOUTH);

        // --- LÓGICA DE EVENTOS (Segunda Ventana) ---
        btnTerminar.addActionListener(e -> System.exit(0)); // Cierre total seguro de la JVM

        frameResultados.setVisible(true);
    }
}
```

Tenga en cuenta:

- **Transición Limpia (`.dispose()` vs `System.exit()`)**: Al presionar el primer botón, usamos **`frameIngreso.dispose()`**. Esto destruye la estructura física de la primera ventana en memoria pero mantiene viva la aplicación (y los datos capturados en **`listaPrecaragada`**).

  En la segunda ventana, al presionar el botón rojo, usamos **`System.exit(0)`** para finalizar por completo el proceso de la Máquina Virtual.

- **Arquitectura de Datos Segura**: La **`listaPrecaragada`** funciona como el estado de nuestra aplicación. La segunda ventana no sabe (ni le importa) si los datos se escribieron

uno a uno o si se cargaron desde un archivo; simplemente recibe el estado, ejecuta el *Pipeline* de **Streams** de forma inmediata y renderiza el resultado en un componente visual JList.

- **Experiencia de Usuario (UX) Fluida**: Se ha añadido un mapeo al evento de la caja de texto (**txtNombre.addActionListener**), lo que permite que el usuario pueda escribir un nombre y presionar la tecla Enter en su teclado para agregarlo velozmente, sin tener que arrastrar el ratón hasta el botón "**Agregar**çada vez.

Buenas Prácticas

- **Importación explícita:** Se recomienda importar solo las clases necesarias (ej. `import java.util.Scanner;`) en lugar de usar el comodín (*) para mejorar la claridad y el rendimiento.
- **Jerarquía:** `java.util` no se importa automáticamente; debe declararse explícitamente en cada archivo fuente donde se utilicen sus clases.
- **Subpaquetes:** El paquete contiene subpaquetes como `java.util.regex` (expresiones regulares) o `java.util.concurrent` (concurrentes), los cuales requieren importación individual.

Referencias

- https://cjavaperu.com/2021/09/informacion-practica/
- https://www.programaenlinea.net/lo-nuevo-en-java-8/
- https://learn.microsoft.com/es-es/java/openjdk/reasons-to-move-to-java-11