

DEPARTAMENTO:	Ciencias de la Computacion	CARRERA:	ITIN		
ASIGNATURA:	POO	NIVEL:	segundo	FECHA:	17/5/2025
DOCENTE:	Ing. J	PRÁCTICA N°:	4	CALIFICACIÓN:	

## Manejo de excepciones en Java

Angel Steven Rodriguez Chavez

### RESUMEN

En esta práctica se implementó un sistema de gestión de inventario en Java, enfocado en el manejo de excepciones personalizadas para garantizar la integridad de los datos. El programa permite registrar productos, validar entradas (como códigos únicos, precios positivos y campos no vacíos) y persistir los datos en un archivo CSV. Se desarrollaron excepciones específicas como CampoVacioException, ValorNegativoException y DatosDuplicadosException, integradas en las clases de validación. Los resultados demostraron la eficacia del manejo de errores para evitar datos corruptos y mejorar la experiencia del usuario.

**Palabras Claves:** manejo de excepciones, validación de datos, persistencia en CSV.

### 1. INTRODUCCIÓN:

La gestión de excepciones es crucial en la creación de software para manejar fallos imprevistos y preservar la estabilidad de las aplicaciones. El propósito de esta práctica fue establecer un sistema de inventario que emplee excepciones a medida para gestionar entradas no válidas, códigos duplicados y formatos equivocados. A través de un menú interactivo, se verificaron datos en tiempo real y se garantizó la conservación de los datos en archivos CSV, implementando principios de programación orientada a objetos y buenas prácticas de diseño.

### 2. OBJETIVO(S):

- 2.1 Implementar excepciones personalizadas para gestionar errores específicos en un sistema de inventario.
- 2.2 Validar entradas de usuario (campos vacíos, valores negativos, formatos incorrectos).
- 2.3 Garantizar la integridad de los datos mediante la detección de códigos duplicados.

2.4 Integrar la persistencia de datos en archivos CSV.

### 3. MARCO TEÓRICO:

#### 1. ¿Qué es una excepción en Java?

Una excepción en Java es un evento que interrumpe el flujo normal de un programa cuando ocurre un error o una condición inesperada. Por ejemplo, al intentar dividir entre cero, acceder a un archivo inexistente o ingresar un dato inválido. Java maneja estos eventos mediante un mecanismo estructurado que permite:

- Detectar el error (mediante bloques `try`).
- Manejarlo (con bloques `catch`).
- Limpiar recursos (usando `finally`).
- Las excepciones son objetos que heredan de la clase `Throwable`, como `Exception` (errores recuperables) o `Error` (fallos graves, como falta de memoria).

#### 2. ¿Cuál es la diferencia entre checked y unchecked exceptions?

- Checked Exceptions (Verificadas):
  - Son verificadas en tiempo de compilación. El compilador obliga a manejarlas (con `try-catch` o declarándolas con `throws`).
  - Representan errores recuperables o externos al programa, como `IOException` (fallo al leer un archivo) o `SQLException`.

- Ejemplo:

```
public void leerArchivo() throws IOException { ... }
```

- **Unchecked Exceptions (No verificadas):**

- No son verificadas en tiempo de compilación. Surgen durante la ejecución, generalmente por errores lógicos.
- Heredan de `RuntimeException` (ej. `NullPointerException`, `ArrayIndexOutOfBoundsException`).

Ejemplo:

```
int[] arr = {1, 2};
```

```
System.out.println(arr[3]); // Lanza ArrayIndexOutOfBoundsException
```

### 3. ¿Qué ventajas aporta usar excepciones personalizadas?

- Claridad en el manejo de errores: Permiten definir mensajes específicos para cada situación, como `DatosDuplicadosException``, facilitando la depuración.
- Control granular: Se adaptan a reglas de negocio únicas. Por ejemplo, validar que un código de producto no esté repetido.
- Mejor legibilidad: Al nombrar excepciones según el contexto (ej. `CampoVacioException``), el código se vuelve más intuitivo.
- Flexibilidad: Permiten agregar campos o métodos adicionales para incluir detalles del error (ej. timestamp, ID del recurso afectado).
- Separación de preocupaciones: Centralizan la lógica de manejo de errores, evitando repetir código en múltiples clases.

#### Ejemplo de excepción personalizada:

```
public class SaldoInsuficienteException extends Exception {  
    public SaldoInsuficienteException(double saldo) {  
        super("Saldo insuficiente: $" + saldo);  
    }  
}
```

**Excepciones en Java:** Mecanismos para manejar errores durante la ejecución.

**Excepciones personalizadas:** Clases que extienden `Exception` para representar errores específicos de la aplicación (ej. `DatosDuplicadosException`).

**Validación de datos:** Técnicas para asegurar que las entradas cumplan con reglas predefinidas (ej. valores positivos, formatos correctos).

**Persistencia en CSV:** Almacenamiento estructurado de datos en archivos de texto, utilizando bibliotecas como FileWriter y BufferedReader.

#### 4. DESCRIPCIÓN DEL PROCEDIMIENTO:

Creación de excepciones personalizadas en la clase Excepciones.java.

Validación de entradas mediante la clase Validaciones.java, usando métodos como ValidacionIntPositivo() y capturando excepciones.

Menú interactivo en Menu.java para registrar productos, listarlos y guardar en CSV.

Persistencia de datos con CSVManager.java, manejando lectura/escritura de archivos y escapado de caracteres.

Detección de duplicados mediante comparación con datos existentes en el CSV y el inventario en memoria.

#### 5. ANÁLISIS DE RESULTADOS:

Validaciones exitosas:

- Campos vacíos: Se lanza CampoVacioException y se solicita reinserción.
- Códigos duplicados: DatosDuplicadosException evita registros repetidos.
- Precios inválidos: ValorNegativoException bloquea valores  $\leq 0$ .

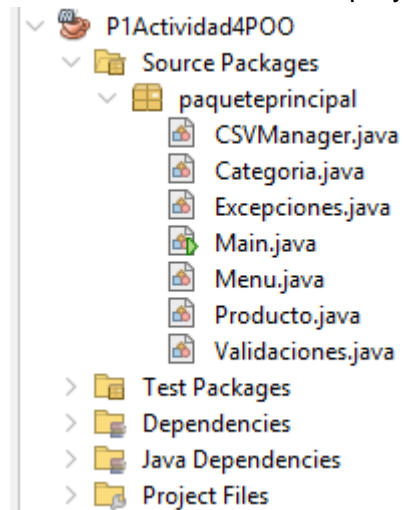
**Persistencia en CSV:** Los datos se guardaron correctamente con el formato

Codigo,Nombre,Precio,ID\_Categoria,Categoria.

**Mejoras identificadas:** Implementar excepciones para categorías no existentes y optimizar la carga inicial del CSV.

## 6. GRÁFICOS O FOTOGRAFÍAS:

Gestión de archivos en el proyecto java



### Main.java

Clase principal que inicia la aplicación. Ejecuta el método main(), instanciando el menú principal (Menu.java).

```
1 package paqueteprincipal;
2 public class Main {
3     public static void main(String[] args) {
4         Menu menu=new Menu();
5         menu.mostrarMenuPrincipal();
6     }
7 }
8
```

### Producto.java

- Modela un producto con atributos: *nombre*, *categoría*, *código*, *precio*.
- Los setters (setNombre(), setPrecio()) usan validaciones automáticas.
- Incluye método MostrarResumen() para visualizar detalles.

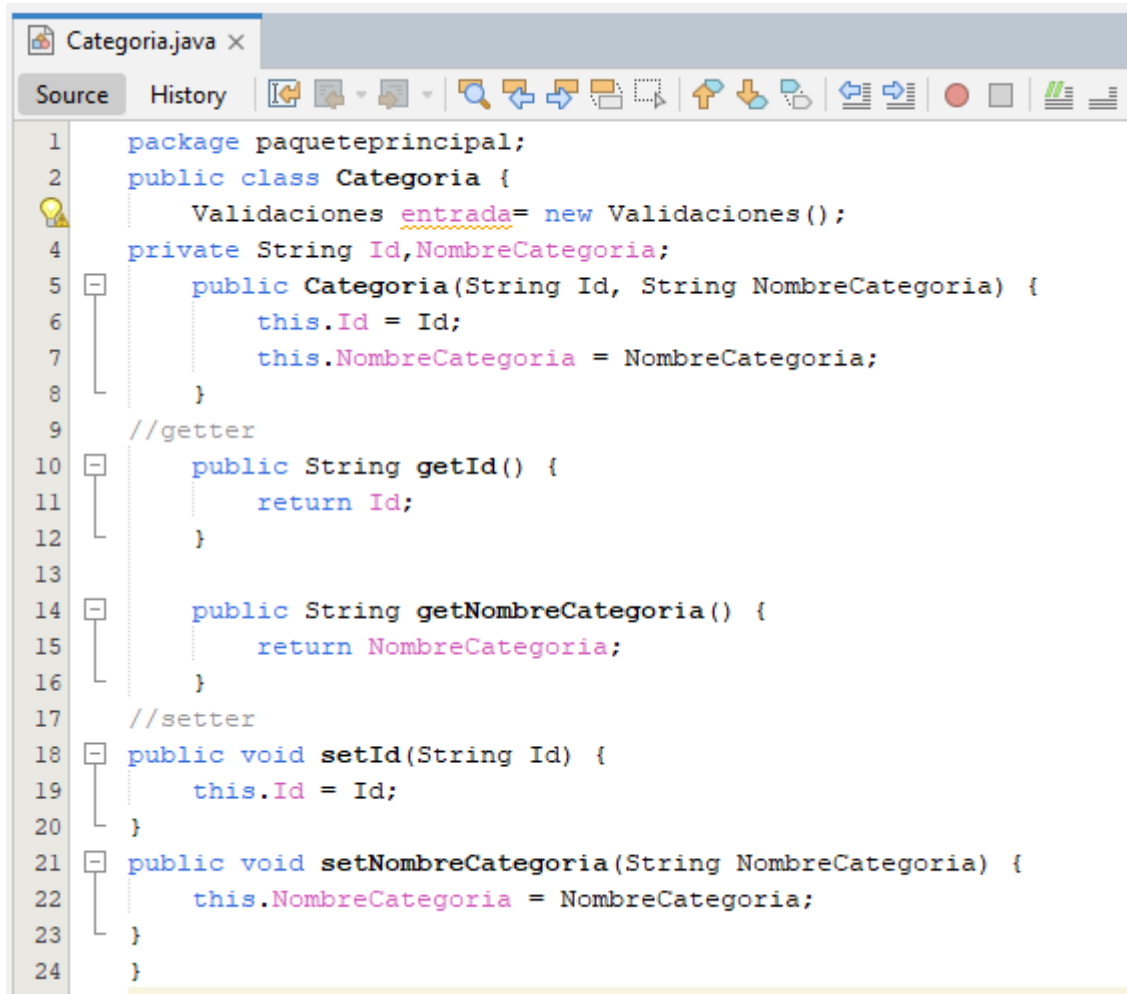
```
Producto.java x
Source History
1 package paqueteprincipal;
2 public class Producto {
3     Validaciones entrada= new Validaciones();
4     private String nombre;
5     private Categoria categoria;
6     private int codigo;
7     private double precio;
8
9     public Producto(String nombre, Categoria categoria, int codigo, double precio) {
10         this.nombre = nombre;
11         this.categoria = categoria;
12         this.codigo = codigo;
13         this.precio = precio;
14     }
15
16     //getter
17     public String getNombre() {
18         return nombre;
19     }
20
21     public Categoria getCategoria() {
22         return categoria;
23     }
24
25     public int getCodigo() {
26         return codigo;
27     }
28
29     public double getPrecio() {
30         return precio;
31     }
32 }
```

```
31 //setter
32 public void setCodigo(int codigo) {
33     this.codigo = codigo;
34 }
35 public void setPrecio(double precio) {
36     this.precio = precio;
37 }
38 public void setNombre(String nombre) {
39     this.nombre = nombre;
40 }
41
42 public void setCategoria(Categoria categoria) {
43     this.categoria = categoria;
44 }
45
46
47
48 public void MostrarResumen() {
49     System.out.println("\n Producto:");
50     System.out.println("Codigo: " + getCodigo());
51     System.out.println("Nombre: " + getNombre());
52     System.out.println("Precio: " + getPrecio());
53     System.out.println("Categoria: " + categoria.getNombreCategoria());
54     System.out.println("ID categoria: " + categoria.getId());
55 }
56 }
```

### Categoria.java

Representa una categoría de producto con *ID* y *nombre*.

- Setters (setId(), setNombreCategoria()) validan entradas no vacías mediante Validaciones.java.



```
1 package paqueteprincipal;
2 public class Categoria {
3     Validaciones entrada= new Validaciones();
4     private String Id,NombreCategoria;
5     public Categoria(String Id, String NombreCategoria) {
6         this.Id = Id;
7         this.NombreCategoria = NombreCategoria;
8     }
9     //getter
10    public String getId() {
11        return Id;
12    }
13
14    public String getNombreCategoria() {
15        return NombreCategoria;
16    }
17    //setter
18    public void setId(String Id) {
19        this.Id = Id;
20    }
21    public void setNombreCategoria(String NombreCategoria) {
22        this.NombreCategoria = NombreCategoria;
23    }
24 }
```

### Validaciones.java

Clase dedicada a la validación de entradas del usuario. Incluye métodos para:

- Verificar campos vacíos (ValidacionString()).
- Validar números positivos (ValidacionDoublePositivo(), ValidacionIntPositivo()).
- Detectar códigos duplicados en el inventario (validarCodigoDuplicado()).

Uso de excepciones personalizadas: CampoVacioException, ValorNegativoException.



```
Validaciones.java x
Source History

1 package paqueteprincipal;
2 import java.util.*;
3 import paqueteprincipal.Excepciones.*;
4 public class Validaciones {
5     Scanner sc = new Scanner(System.in);
6     public String ValidacionString(String mensaje) {
7         while (true) {
8             try {
9                 System.out.println(mensaje);
10                String input = sc.nextLine().trim();
11                if (input.isEmpty()) {
12                    throw new CampoVacioException("ERROR: campo vacio, ingrese de nuevo");
13                }
14                return input;
15            } catch (CampoVacioException e) {
16                System.out.println(e.getMessage());
17            }
18        }
19    }

20    public double ValidacionDoublePositivo(String mensaje) {
21        while (true) {
22            try {
23                System.out.println(mensaje);
24                String entrada = sc.nextLine();
25                double valor;
26                try {
27                    valor = Double.parseDouble(entrada);
28                } catch (NumberFormatException e) {
29                    throw new FormatoIncorrectoException("ERROR: Ingrese solo numeros"+ e.getMessage());
30                }
31                if (valor <= 0) {
32                    throw new ValorNegativoException("ERROR: debe ser mayor que 0");
33                }
34                return valor;
35            } catch (FormatoIncorrectoException | ValorNegativoException e) {
36                System.out.println(e.getMessage());
37            }
38        }
39    }
}
```

```

40 public int ValidacionIntPositivo(String mensaje) {
41     while (true) {
42         try {
43             System.out.println(mensaje);
44             String entrada = sc.nextLine();
45             int valor = Integer.parseInt(entrada);
46             if (valor <= 0) {
47                 throw new IllegalArgumentException("ERROR: debe ser mayor que 0");
48             }
49             return valor;
50         } catch (NumberFormatException e) {
51             System.out.println("ERROR: Ingrese solo numeros"+e.getMessage());
52         } catch (IllegalArgumentException e) {
53             System.out.println(e.getMessage());
54         }
55     }
56 }
57
58 public void validarCodigoDuplicado(int codigo, ArrayList<Producto> inventario) throws DatosDuplicadosException {
59     for (Producto p : inventario) {
60         if (p.getCodigo() == codigo) {
61             throw new DatosDuplicadosException("ERROR: Código duplicado, ya existe un producto con ese código.");
62         }
63     }
64 }
65 }

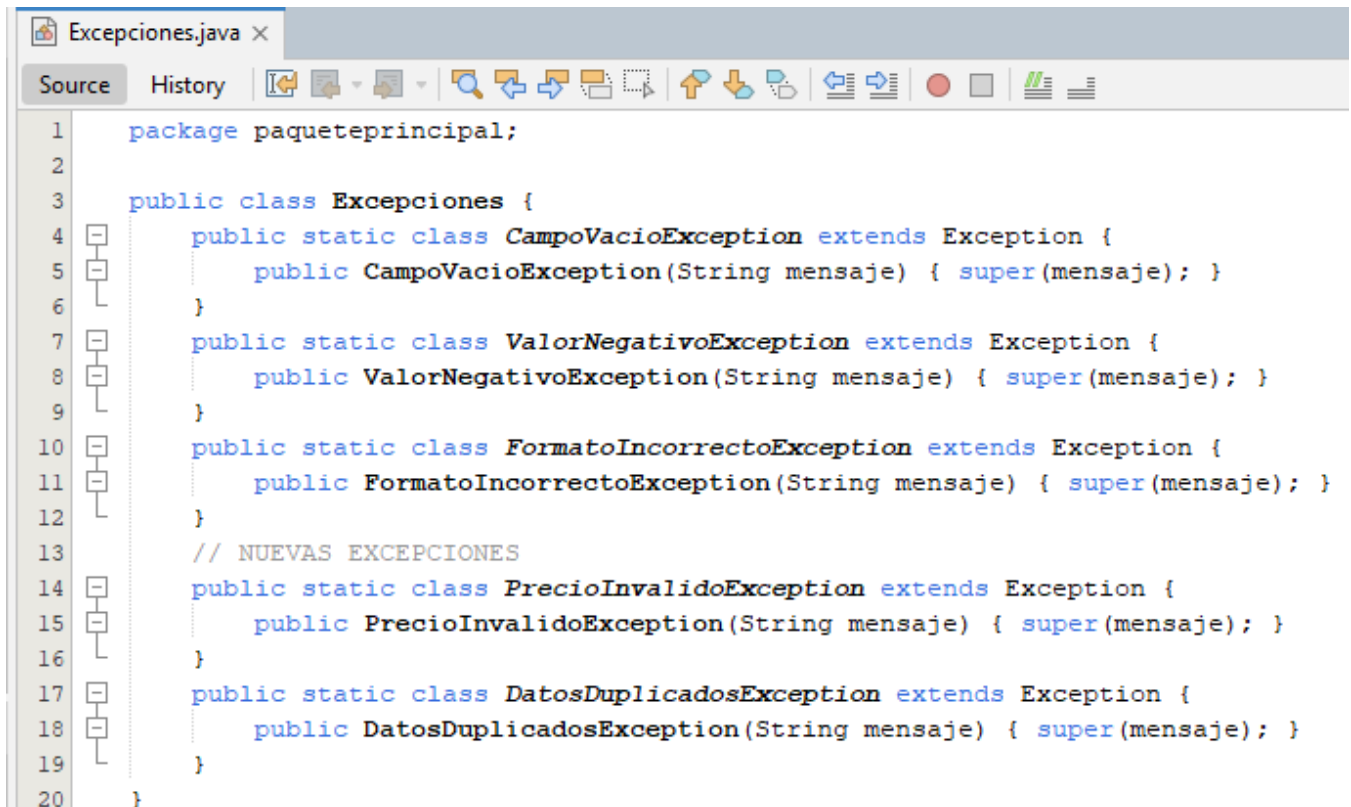
```

## Excepciones.java

Contiene excepciones personalizadas para el sistema:

- DatosDuplicadosException: Códigos repetidos.
- PrecioInvalidoException: Precios  $\leq 0$ .
- CampoVacioException: Entradas vacías.

*Organización:* Clases anidadas estáticas para agrupación lógica.



```

1 package paqueteprincipal;
2
3 public class Excepciones {
4     public static class CampoVacioException extends Exception {
5         public CampoVacioException(String mensaje) { super(mensaje); }
6     }
7     public static class ValorNegativoException extends Exception {
8         public ValorNegativoException(String mensaje) { super(mensaje); }
9     }
10    public static class FormatoIncorrectoException extends Exception {
11        public FormatoIncorrectoException(String mensaje) { super(mensaje); }
12    }
13    // NUEVAS EXCEPCIONES
14    public static class PrecioInvalidoException extends Exception {
15        public PrecioInvalidoException(String mensaje) { super(mensaje); }
16    }
17    public static class DatosDuplicadosException extends Exception {
18        public DatosDuplicadosException(String mensaje) { super(mensaje); }
19    }
20 }

```

## CSVManager.java

Gestiona la persistencia de datos en archivos CSV. Métodos principales:

- guardarEnCSV(): Escribe productos en formato CSV con encabezados.
- cargarDesdeCSV(): Recupera datos y reconstruye objetos Producto.
- existeCodigoEnCSV(): Verifica duplicados en el archivo.

```
11 public void guardarEnCSV(ArrayList<Producto> inventario, String nombreArchivo) {
12     try (PrintWriter writer = new PrintWriter(new FileWriter(nombreArchivo))) {
13         writer.println("Codigo,Nombre,Precio,ID_Categoria,Categoria");
14
15         for (Producto producto : inventario) {
16             writer.println(
17                 producto.getCodigo() + "," +
18                 "\"" + producto.getNombre().replace("\"", "\\\"") + "\",\" +
19                 producto.getPrecio() + "," +
20                 "\"" + producto.getCategoria().getId() + "\",\" +
21                 "\"" + producto.getCategoria().getNombreCategoria() + "\""
22             );
23         }
24         System.out.println("Datos guardados correctamente en: " + nombreArchivo);
25     } catch (IOException e) {
26         System.out.println("Error crítico al guardar: " + e.getMessage());
27     }
28 }
```

```

30 public ArrayList<Producto> cargarDesdeCSV(String nombreArchivo) {
31     ArrayList<Producto> inventario = new ArrayList<>();
32     File archivo = new File(nombreArchivo);
33     if (!archivo.exists()) return inventario;
34     try (BufferedReader br = new BufferedReader(new FileReader(archivo))) {
35         br.readLine(); // Saltar encabezado
36
37         String linea;
38         while ((linea = br.readLine()) != null) {
39             String[] datos = linea.split(", (?=(?:[^\"]*"|\"\"|\"'\")*$)");
40
41             if (datos.length == 5) {
42                 int codigo = Integer.parseInt(datos[0]);
43                 String nombre = datos[1].replaceAll("^\\\"|\\\"$", "");
44                 double precio = Double.parseDouble(datos[2]);
45                 String idCat = datos[3].replaceAll("^\\\"|\\\"$", "");
46                 String nombreCat = datos[4].replaceAll("^\\\"|\\\"$", "");
47                 inventario.add(new Producto(
48                     nombre,
49                     new Categoria(idCat, nombreCat),
50                     codigo,
51                     precio
52                 ));
53             }
54         }
55     } catch (Exception e) {
56         System.out.println("Error leyendo archivo: " + e.getMessage());
57     }
58     return inventario;
59 }
60
61 public boolean existeCodigoEnCSV(int codigo, String nombreArchivo) {
62     try (BufferedReader br = new BufferedReader(new FileReader(nombreArchivo))) {
63         String linea;
64         boolean primeraLinea = true;
65         while ((linea = br.readLine()) != null) {
66             if (primeraLinea) {
67                 primeraLinea = false;
68                 continue; // Saltar encabezado
69             }
70             String[] datos = linea.split(",");
71             if (datos.length > 0 && Integer.parseInt(datos[0]) == codigo) {
72                 return true;
73             }
74         }
75     } catch (Exception e) {
76         // Si el archivo no existe, no hay duplicados
77     }
78     return false;
79 }
80 }

```

## Menu.java

Controla la interfaz de usuario mediante un menú interactivo. Funcionalidades clave:

- Registrar productos (valida datos y evita duplicados).
- Listar productos cargados desde CSV.
- Guardar el inventario en un archivo CSV.
- Integra Validaciones.java y CSVManager.java.
- 

```
1 package paqueteprincipal;
2 import java.util.*;
3 import paqueteprincipal.Excepciones.*;
4 public class Menu {
5     Scanner sc=new Scanner(System.in);
6     CSVManager csvManager = new CSVManager();
7     ArrayList<Producto> inventario=new ArrayList<>();
8     Validaciones validaciones = new Validaciones();
9     public void mostrarMenuPrincipal(){
10         char opc;
11         do{
12             System.out.println("\n--- MENU PRINCIPAL ---");
13             System.out.println("1. Registrar producto");
14             System.out.println("2. Listar productos");
15             System.out.println("3. Guardar en CSV");
16             System.out.println("4. Salir");
17             opc=sc.nextLine().trim().charAt(0);
```

```
19 switch(opc) {
20     case '1':
21         // Registrar producto
22         RegistrarProdcutos();
23         break;
24     case '2':
25         // Cargar productos desde CSV al inventario
26         ListarProductos();
27         break;
28     case '3':
29         // Guardar inventario en CSV
30         csvManager.guardarEnCSV(inventario, "inventario.csv");
31         break;
32     case '4':
33         // Salir
34         System.out.println("BYE BYE");
35         break;
36     default:
37         System.out.println("ERROR: Opcion no valida");
38         break;
39 }
40 while(opc != '4');
41 // Guardar automáticamente al salir
42 csvManager.guardarEnCSV(inventario, "inventario.csv");
43 }
```

```
45 public void RegistrarProdcutos() {
46     try {
47         System.out.println("\n--- REGISTRO DE PRODUCTO ---");
48
49         // 1. Validar nombre
50         String nombre = validaciones.ValidacionString("Ingrese nombre: ");
51
52         // 2. Validar código único
53         int codigo = validaciones.ValidacionIntPositivo("Ingrese código: ");
54         if (csvManager.existeCodigoEnCSV(codigo, "inventario.csv")) {
55             throw new Excepciones.DatosDuplicadosException("Código ya existe en CSV");
56         }
57         validaciones.validarCodigoDuplicado(codigo, inventario);
58
59         // 3. Validar precio
60         double precio = validaciones.ValidacionDoublePositivo("Ingrese precio: ");
61
62         // 4. Validar categoría
63         String idCategoria = validaciones.ValidacionString("ID categoría: ");
64         String nombreCategoria = validaciones.ValidacionString("Nombre categoría: ");
65
66         // Crear objeto Producto
67         Producto nuevo = new Producto(
68             nombre,
69             new Categoria(idCategoria, nombreCategoria),
70             codigo,
71             precio
72         );
73
74         inventario.add(nuevo);
75         System.out.println("\n;Producto registrado!");
76     } catch (Excepciones.DatosDuplicadosException e) {
77         System.out.println(e.getMessage());
78     } catch (Exception e) {
79         System.out.println("Error: " + e.getMessage());
80     }
81 }
```

```
84 public void ListarProductos() {
85     ArrayList<Producto> productosCSV = csvManager.cargarDesdeCSV("inventario.csv");
86
87     if (productosCSV.isEmpty()) {
88         System.out.println("\nNo hay productos en el archivo CSV");
89         return;
90     }
91
92     System.out.println("\n--- PRODUCTOS DESDE CSV ---");
93     productosCSV.forEach(producto -> {
94         System.out.println("\nCodigo: " + producto.getCodigo());
95         System.out.println("Nombre: " + producto.getNombre());
96         System.out.println("Precio: $" + producto.getPrecio());
97         System.out.println("Categoria: " + producto.getCategoria().getNombreCategoria());
98         System.out.println("ID Categoria: " + producto.getCategoria().getId());
99     });
100 }
101 }
102 }
```

- Ejecución Datos inválidos:

Primera Excepción del menú (no permite caracteres diferente a los que se encuentre disponibles en las opciones)

Excepción en caso de no existir el archivo para guardar datos, el programa genera un archivo nuevo al seleccionar la opción 3 pero sigue su funcionamiento

```
--- MENU PRINCIPAL ---
1. Registrar producto
2. Listar productos
3. Guardar en CSV
4. Salir
a
ERROR: Opcion no valida

--- MENU PRINCIPAL ---
1. Registrar producto
2. Listar productos
3. Guardar en CSV
4. Salir
2
Error leyendo CSV: inventario.csv (El sistema no puede encontrar el archivo especificado)

No hay productos en el archivo CSV
```



Uso de la opción 1 pero ingreso de datos incorrectos (recordar que: nombre, id\_categoria y nombre de categoría son de tipo string) por lo tanto solo se validaría que no ingrese datos vacíos:

```
--- REGISTRO DE PRODUCTO ---
Ingrese nombre:
laptop
Ingrese codigo:
s
ERROR: Ingrese solo numerosFor input string: "s"
Ingrese codigo:
9784
Ingrese precio:
e
ERROR: Ingrese solo numerosFor input string: "e"
Ingrese precio:
21.2
ID categoria:
3e32
Nombre categoria:
electronico

Producto registrado
```

Creacion del archivo

```
--- MENU PRINCIPAL ---
1. Registrar producto
2. Listar productos
3. Guardar en CSV
4. Salir
3

Datos guardados exitosamente en: inventario.csv
```

Ejecución Datos válidos:

```
Output - Run (P1Actividad4POO) x
3. Guardar en CSV
4. Salir
1
--- REGISTRO DE PRODUCTO ---
Ingrese nombre:
moto
Ingrese codigo:
23421
Ingrese precio:
2000
ID categoria:
34t5e
Nombre categoria:
vehiculo

Producto registrado
```

Uso correcto de la opción 2:

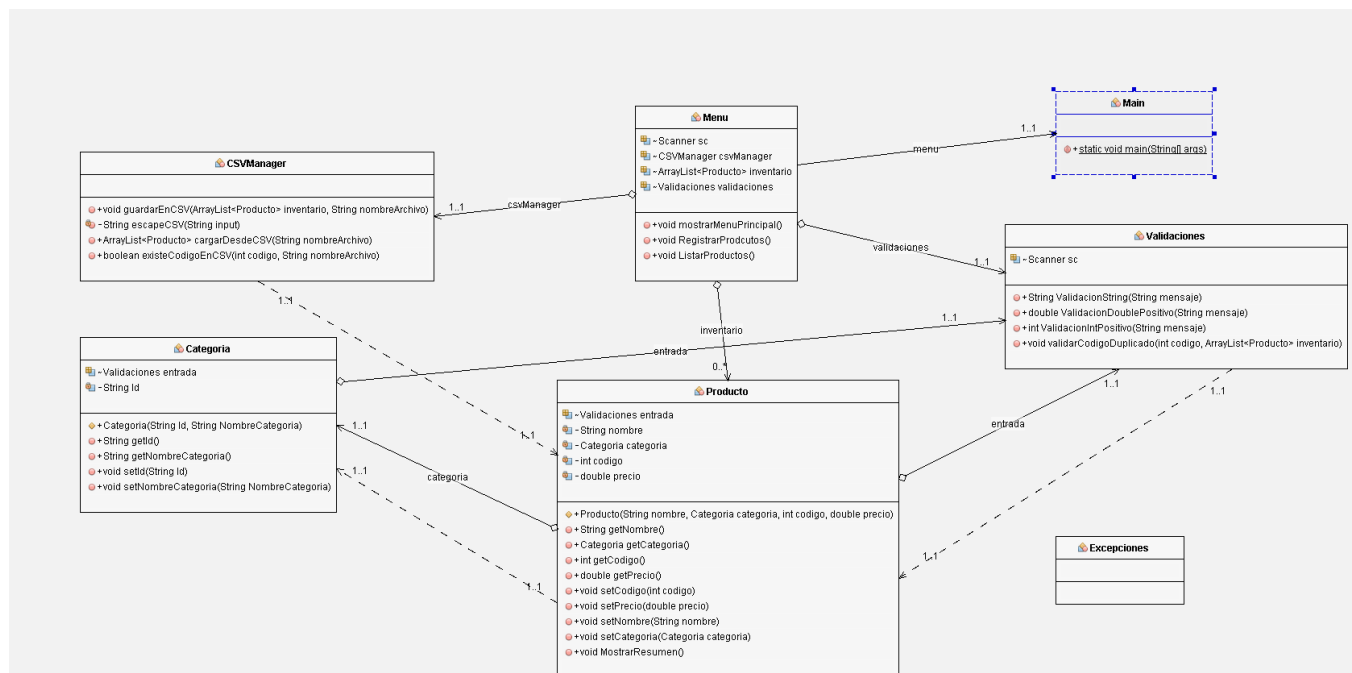
```

--- MENU PRINCIPAL ---
1. Registrar producto
2. Listar productos
3. Guardar en CSV
4. Salir
2

--- PRODUCTOS DESDE CSV ---

Codigo: 2894
Nombre: moto
Precio: $2000.0
Categoria: vehiculo
ID Categoria: 3e9t6
  
```

Diagrama Clases:



Link Github: [angeldev7/Actividad5POO](https://github.com/angeldev7/Actividad5POO)

## 7. DISCUSIÓN:

El uso de excepciones personalizadas permitió un control granular de errores, superando las limitaciones de las excepciones genéricas de Java. Sin embargo, la validación de categorías quedó incompleta, ya que no se verifica si existen previamente. La persistencia en CSV demostró ser eficiente para pequeñas escalas, aunque en sistemas mayores sería recomendable usar bases de datos.

**CONCLUSIONES:**

- Las excepciones personalizadas mejoran la claridad del manejo de errores.
- La validación en tiempo real evita la corrupción de datos.
- La integración con CSV es viable para aplicaciones sencillas.
- Se recomienda extender las validaciones a campos como categorías y nombres.

**8. BIBLIOGRAFÍA:**

- Oracle. 2023. Exceptions in Java. Documentación oficial de Java.
- Freeman, A. 2020. Pro Java EE 7. Apress.
- CSV File Handling in Java. GeeksforGeeks. Consultado el 17/5/2025.