En Java, las estructuras `try` y `catch` se usan para manejar excepciones, que son eventos inesperados que pueden ocurrir durante la ejecución de un programa. Cuando se produce una excepción, el flujo normal del programa se detiene y puede causar errores si no se manejan adecuadamente. Para evitar esto, se utiliza el bloque `try-catch`.

Aquí te explico cómo funciona:

Estructura básica:

```
try {
    // Código que puede generar una excepción
} catch (TipoDeExcepcion ex) {
    // Código que maneja la excepción
}
```

1. **`try`**: En este bloque colocamos el código que podría generar una excepción. Si ocurre una excepción, el flujo de ejecución se transfiere al bloque `catch` correspondiente.
2. **`catch`**: En este bloque capturamos y manejamos la excepción. Podemos especificar qué tipo de excepción queremos capturar y qué hacer cuando ocurra esa excepción.

Ejemplo:

```
public class Ejemplo {
    public static void main(String[] args) {
        try {
            // Intentamos dividir por 0, lo que causará una excepción
            int resultado = 10 / 0;
        } catch (ArithmeticException e) {
            // Si ocurre una ArithmeticException, se maneja aquí
            System.out.println("Error: División por cero!");
        }
    }
}
```

Explicación:

- En el bloque `try`, intentamos hacer una operación que sabemos que generará una excepción: `10 / 0`.
- La excepción que ocurre es `ArithmeticException` (división por cero).
- El bloque `catch` captura esa excepción y ejecuta el código dentro de él, en este caso, mostrando un mensaje de error.

Variantes:

1. **Múltiples excepciones**: Puedes usar varios bloques `catch` para manejar diferentes tipos de excepciones.

```
try {
    // Código que puede generar excepciones
} catch (NullPointerException e) {
    // Manejo de NullPointerException
} catch (ArithmeticException e) {
    // Manejo de ArithmeticException
} catch (Exception e) {
    // Manejo de otras excepciones generales
}
```

2. **`finally`**: El bloque `finally` se ejecuta siempre, haya o no ocurrido una excepción. Es útil para cerrar recursos como archivos o conexiones a bases de datos, independientemente de si ocurrió una excepción.

```
try {
    // Código que puede generar una excepción
} catch (Exception e) {
    // Manejo de la excepción
} finally {
    // Este bloque se ejecuta siempre, se haya producido o no una excepción
    System.out.println("Este bloque siempre se ejecuta.");
}
```

Resumen:

- **`try`**: Ejecuta código que podría lanzar una excepción.
- **`catch`**: Captura la excepción y permite manejarla de forma controlada.
- **`finally`**: (Opcional) Ejecuta código que siempre debe ocurrir, independientemente de si hay una excepción o no.

Aquí tienes dos ejercicios similares, siguiendo la misma estructura que el que mencionaste:

Ejercicio 1: Sistema de Gestión de Reservas de Hotel con Excepciones

Implementa un sistema de gestión de reservas de hotel en Java con las siguientes características:

Clases y requisitos:

1. **Clase Reserva**
    - **Atributos**:
        - id (int)
        - noches (int)
        - precioPorNoche (double)
    - **Constructor** con parámetros para inicializar los atributos.
    - **Método calcularTotal()**, que devuelve el total de la reserva (noches * precioPorNoche).
    - **Validaciones**:
        - Si noches es menor o igual a 0, se debe lanzar una excepción personalizada `NochesInvalidaException`.
        - Si precioPorNoche es menor o igual a 0, se debe lanzar una excepción personalizada `PrecioPorNocheInvalidoException`.
2. **Clase GestorReservas**
    - Debe contener un método **procesarReserva(Reserva reserva)** que invoque `calcularTotal()`.
    - En la clase principal, se deben crear varias reservas, probando con valores válidos e inválidos.

- Se deben capturar y manejar correctamente todas las excepciones.

Consideraciones:

- Las excepciones personalizadas deben definirse en clases separadas.
- El alumno debe decidir si las excepciones se capturan en el propio método o se delegan a un nivel superior.
- Se debe utilizar **try-catch** para gestionar los errores y un bloque **finally** para indicar que la operación ha finalizado.

Ejercicio 2: Sistema de Gestión de Compras en una Tienda Online con Excepciones

Implementa un sistema de gestión de compras en una tienda online en Java con las siguientes características:

Clases y requisitos:

1. **Clase Producto**
    - **Atributos**:
        - id (int)
        - cantidadStock (int)
        - precio (double)
    - **Constructor** con parámetros para inicializar los atributos.
    - **Método calcularPrecioTotal()**, que devuelve el total de la compra (cantidadStock * precio).
    - **Validaciones**:
        - Si cantidadStock es menor o igual a 0, se debe lanzar una excepción personalizada `CantidadStockInvalidaException`.
        - Si precio es menor o igual a 0, se debe lanzar una excepción personalizada `PrecioInvalidoException`.
2. **Clase GestorCompras**
    - Debe contener un método **procesarCompra(Producto producto)** que invoque `calcularPrecioTotal()`.
    - En la clase principal, se deben crear varios productos, probando con valores válidos e inválidos.
    - Se deben capturar y manejar correctamente todas las excepciones.

Consideraciones:

- Las excepciones personalizadas deben definirse en clases separadas.
- El alumno debe decidir si las excepciones se capturan en el propio método o se delegan a un nivel superior.
- Se debe utilizar **try-catch** para gestionar los errores y un bloque **finally** para indicar que la operación ha finalizado.

Estos ejercicios siguen la misma lógica, pero con diferentes contextos (reservas de hotel y compras en línea). Puedes usar estos en un entorno educativo para que los estudiantes trabajen con excepciones y validaciones.

¡Claro! Aquí te dejo las soluciones de los dos ejercicios que te planteé.

Ejercicio 1: Sistema de Gestión de Reservas de Hotel con Excepciones

1. Excepciones Personalizadas:

Primero, creamos las excepciones personalizadas `NochesInvalidaException` y `PrecioPorNocheInvalidoException`.

```
// Excepción personalizada para noches inválidas
class NochesInvalidaException extends Exception {
    public NochesInvalidaException(String mensaje) {
        super(mensaje);
    }
}

// Excepción personalizada para precio por noche inválido
class PrecioPorNocheInvalidoException extends Exception {
    public PrecioPorNocheInvalidoException(String mensaje) {
        super(mensaje);
    }
}
```

2. Clase `Reserva`:

```
class Reserva {
    private int id;
    private int noches;
    private double precioPorNoche;

    // Constructor
    public Reserva(int id, int noches, double precioPorNoche) {
        this.id = id;
        this.noches = noches;
        this.precioPorNoche = precioPorNoche;
    }

    // Método para calcular el total
    public double calcularTotal() throws NochesInvalidaException,
PrecioPorNocheInvalidoException {
        if (noches <= 0) {
            throw new NochesInvalidaException("La cantidad de noches no puede
ser menor o igual a 0.");
        }
        if (precioPorNoche <= 0) {
            throw new PrecioPorNocheInvalidoException("El precio por noche no
puede ser menor o igual a 0.");
        }
        return noches * precioPorNoche;
    }
}
```

3. Clase `GestorReservas`:

```
class GestorReservas {
    public void procesarReserva(Reserva reserva) {
        try {
            double total = reserva.calcularTotal();
            System.out.println("El total de la reserva es: " + total);
        } catch (NochesInvalidaException | PrecioPorNocheInvalidoException e) {
            System.out.println("Error al procesar la reserva: " +
e.getMessage());
        } finally {
            System.out.println("Proceso de reserva finalizado.");
        }
    }
}
```

4. Clase Principal:

```
public class Main {
    public static void main(String[] args) {
        Reserva reserva1 = new Reserva(1, 3, 100.0);
        Reserva reserva2 = new Reserva(2, 0, 100.0); // Noches inválidas
        Reserva reserva3 = new Reserva(3, 3, -100.0); // Precio inválido

        GestorReservas gestor = new GestorReservas();

        gestor.procesarReserva(reserva1);
        gestor.procesarReserva(reserva2);
        gestor.procesarReserva(reserva3);
    }
}
```

Ejercicio 2: Sistema de Gestión de Compras en una Tienda Online con Excepciones

1. Excepciones Personalizadas:

```
// Excepción personalizada para cantidad de stock inválida
class CantidadStockInvalidaException extends Exception {
    public CantidadStockInvalidaException(String mensaje) {
        super(mensaje);
    }
}

// Excepción personalizada para precio inválido
class PrecioInvalidoException extends Exception {
    public PrecioInvalidoException(String mensaje) {
        super(mensaje);
    }
}
```

2. Clase `Producto`:

```
class Producto {
    private int id;
    private int cantidadStock;
    private double precio;

    // Constructor
    public Producto(int id, int cantidadStock, double precio) {
        this.id = id;
```

```
        this.cantidadStock = cantidadStock;
        this.precio = precio;
    }

    // Método para calcular el precio total
    public double calcularPrecioTotal() throws CantidadStockInvalidaException,
PrecioInvalidoException {
        if (cantidadStock <= 0) {
            throw new CantidadStockInvalidaException("La cantidad de stock no
puede ser menor o igual a 0.");
        }
        if (precio <= 0) {
            throw new PrecioInvalidoException("El precio no puede ser menor o
igual a 0.");
        }
        return cantidadStock * precio;
    }
}
```

3. Clase `GestorCompras`:

```
class GestorCompras {
    public void procesarCompra(Producto producto) {
        try {
            double total = producto.calcularPrecioTotal();
            System.out.println("El precio total de la compra es: " + total);
        } catch (CantidadStockInvalidaException | PrecioInvalidoException e) {
            System.out.println("Error al procesar la compra: " +
e.getMessage());
        } finally {
            System.out.println("Proceso de compra finalizado.");
        }
    }
}
```

4. Clase Principal:

```
public class Main {
    public static void main(String[] args) {
        Producto producto1 = new Producto(1, 10, 50.0);
        Producto producto2 = new Producto(2, 0, 50.0); // Stock inválido
        Producto producto3 = new Producto(3, 10, -50.0); // Precio inválido

        GestorCompras gestor = new GestorCompras();

        gestor.procesarCompra(producto1);
        gestor.procesarCompra(producto2);
        gestor.procesarCompra(producto3);
    }
}
```

Explicación de las Soluciones:

- **Excepciones Personalizadas**: Se crean excepciones propias para manejar situaciones donde los valores son inválidos. Por ejemplo, si el número de noches o el precio de una reserva son menores o iguales a cero, se lanza una excepción correspondiente.

- **Clases `Reserva` y `Producto`**: Ambas clases tienen un método para calcular el total de la compra/reserva, pero este cálculo está sujeto a validaciones. Si alguna validación falla, se lanza la excepción respectiva.
- **Clase `GestorReservas` y `GestorCompras`**: Estas clases gestionan la ejecución de los pedidos/reservas, llamando al método para calcular el total. Capturan las excepciones y las manejan adecuadamente con `try-catch`. También utilizan el bloque `finally` para indicar que la operación se ha finalizado, independientemente de si hubo errores.
- **Clase Principal**: En la clase principal se crean instancias de los objetos `Reserva` o `Producto` y se pasa a las clases correspondientes para probar los distintos escenarios de validación.

Con estas soluciones, el estudiante puede experimentar con la captura de excepciones y ver cómo manejar diferentes tipos de errores en el sistema.

¡Por supuesto! Aquí te dejo tres ejercicios adicionales que siguen la misma lógica y estructura que los anteriores, junto con sus soluciones correspondientes.

Ejercicio 3: Sistema de Gestión de Cuentas Bancarias con Excepciones

Implementa un sistema de gestión de cuentas bancarias en Java con las siguientes características:

Clases y requisitos:

1. **Clase CuentaBancaria**
    - **Atributos**:
        - id (int)
        - saldo (double)
    - **Constructor** con parámetros para inicializar los atributos.
    - **Método depositar()**, que suma una cantidad al saldo.
    - **Método retirar()**, que resta una cantidad del saldo.
    - **Validaciones**:
        - Si la cantidad a depositar o retirar es menor o igual a 0, se debe lanzar una excepción personalizada `CantidadInvalidaException`.
        - Si el saldo es insuficiente para realizar el retiro, se debe lanzar una excepción personalizada `SaldoInsuficienteException`.
2. **Clase GestorCuentas**
    - Debe contener un método **procesarOperacion(CuentaBancaria cuenta, double cantidad, String operacion)** que invoque los métodos `depositar()` o `retirar()` según corresponda.
    - En la clase principal, se deben crear varias cuentas bancarias y realizar operaciones con valores válidos e inválidos.

- Se deben capturar y manejar correctamente todas las excepciones.

Consideraciones:

- Las excepciones personalizadas deben definirse en clases separadas.
- Se debe utilizar **try-catch** para gestionar los errores y un bloque **finally** para indicar que la operación ha finalizado.

Solución:

```
// Excepciones Personalizadas
class CantidadInvalidaException extends Exception {
    public CantidadInvalidaException(String mensaje) {
        super(mensaje);
    }
}

class SaldoInsuficienteException extends Exception {
    public SaldoInsuficienteException(String mensaje) {
        super(mensaje);
    }
}

// Clase CuentaBancaria
class CuentaBancaria {
    private int id;
    private double saldo;

    public CuentaBancaria(int id, double saldo) {
        this.id = id;
        this.saldo = saldo;
    }

    public void depositar(double cantidad) throws CantidadInvalidaException {
        if (cantidad <= 0) {
            throw new CantidadInvalidaException("La cantidad a depositar debe
ser mayor que 0.");
        }
        saldo += cantidad;
    }

    public void retirar(double cantidad) throws CantidadInvalidaException,
SaldoInsuficienteException {
        if (cantidad <= 0) {
            throw new CantidadInvalidaException("La cantidad a retirar debe ser
mayor que 0.");
        }
        if (cantidad > saldo) {
            throw new SaldoInsuficienteException("No hay suficiente saldo para
realizar el retiro.");
        }
        saldo -= cantidad;
    }

    public double getSaldo() {
        return saldo;
    }
}

// Clase GestorCuentas
class GestorCuentas {
```

```
    public void procesarOperacion(CuentaBancaria cuenta, double cantidad, String
operacion) {
        try {
            if (operacion.equals("depositar")) {
                cuenta.depositar(cantidad);
                System.out.println("Depósito exitoso. Nuevo saldo: " +
cuenta.getSaldo());
            } else if (operacion.equals("retirar")) {
                cuenta.retirar(cantidad);
                System.out.println("Retiro exitoso. Nuevo saldo: " +
cuenta.getSaldo());
            }
        } catch (CantidadInvalidaException | SaldoInsuficienteException e) {
            System.out.println("Error en la operación: " + e.getMessage());
        } finally {
            System.out.println("Operación finalizada.");
        }
    }
}

// Clase Principal
public class Main {
    public static void main(String[] args) {
        CuentaBancaria cuenta1 = new CuentaBancaria(1, 500.0);
        CuentaBancaria cuenta2 = new CuentaBancaria(2, 100.0);

        GestorCuentas gestor = new GestorCuentas();

        gestor.procesarOperacion(cuenta1, 200.0, "depositar");
        gestor.procesarOperacion(cuenta2, 150.0, "retirar");
        gestor.procesarOperacion(cuenta2, -50.0, "depositar");
    }
}
```

Ejercicio 4: Sistema de Gestión de Empleados con Excepciones

Implementa un sistema de gestión de empleados en Java con las siguientes características:

Clases y requisitos:

1. **Clase Empleado**
    - **Atributos**:
        - id (int)
        - nombre (String)
        - salario (double)
    - **Constructor** con parámetros para inicializar los atributos.
    - **Método aumentarSalario()**, que aumenta el salario de un empleado.
    - **Validaciones**:
        - Si el salario es menor o igual a 0, se debe lanzar una excepción personalizada `SalarioInvalidoException`.
        - Si el aumento del salario es negativo, se debe lanzar una excepción personalizada `AumentoInvalidoException`.
2. **Clase GestorEmpleados**

- Debe contener un método **procesarAumento(Empleado empleado, double aumento)** que invoque el método `aumentarSalario()`.
- En la clase principal, se deben crear varios empleados y probar con valores válidos e inválidos.
- Se deben capturar y manejar correctamente todas las excepciones.

Solución:

```
// Excepciones Personalizadas
class SalarioInvalidoException extends Exception {
    public SalarioInvalidoException(String mensaje) {
        super(mensaje);
    }
}

class AumentoInvalidoException extends Exception {
    public AumentoInvalidoException(String mensaje) {
        super(mensaje);
    }
}

// Clase Empleado
class Empleado {
    private int id;
    private String nombre;
    private double salario;

    public Empleado(int id, String nombre, double salario) {
        this.id = id;
        this.nombre = nombre;
        this.salario = salario;
    }

    public void aumentarSalario(double aumento) throws AumentoInvalidoException
{
        if (aumento < 0) {
            throw new AumentoInvalidoException("El aumento de salario no puede
ser negativo.");
        }
        salario += aumento;
    }

    public void validarSalario() throws SalarioInvalidoException {
        if (salario <= 0) {
            throw new SalarioInvalidoException("El salario debe ser mayor a
0.");
        }
    }

    public double getSalario() {
        return salario;
    }
}

// Clase GestorEmpleados
class GestorEmpleados {
    public void procesarAumento(Empleado empleado, double aumento) {
        try {
            empleado.validarSalario();
            empleado.aumentarSalario(aumento);
            System.out.println("Aumento exitoso. Nuevo salario: " +
empleado.getSalario());
```

```
        } catch (SalarioInvalidoException | AumentoInvalidoException e) {
            System.out.println("Error al procesar el aumento: " +
e.getMessage());
        } finally {
            System.out.println("Operación de aumento finalizada.");
        }
    }
}

// Clase Principal
public class Main {
    public static void main(String[] args) {
        Empleado empleado1 = new Empleado(1, "Juan", 1500.0);
        Empleado empleado2 = new Empleado(2, "Ana", -200.0); // Salario inválido

        GestorEmpleados gestor = new GestorEmpleados();

        gestor.procesarAumento(empleado1, 300.0);
        gestor.procesarAumento(empleado2, 100.0); // Salario inválido
    }
}
```

Ejercicio 5: Sistema de Gestión de Proyectos con Excepciones

Implementa un sistema de gestión de proyectos en Java con las siguientes características:

Clases y requisitos:

1. **Clase Proyecto**
    - **Atributos**:
        - id (int)
        - nombre (String)
        - duracion (int) - en meses
        - presupuesto (double)
    - **Constructor** con parámetros para inicializar los atributos.
    - **Método calcularCostoTotal()**, que devuelve el costo total del proyecto (duracion * presupuesto).
    - **Validaciones**:
        - Si la duración es menor o igual a 0, se debe lanzar una excepción personalizada `DuracionInvalidaException`.
        - Si el presupuesto es menor o igual a 0, se debe lanzar una excepción personalizada `PresupuestoInvalidoException`.
2. **Clase GestorProyectos**
    - Debe contener un método **procesarProyecto(Proyecto proyecto)** que invoque el método `calcularCostoTotal()`.
    - En la clase principal, se deben crear varios proyectos y probar con valores válidos e inválidos.
    - Se deben capturar y manejar correctamente todas las excepciones.

Solución:

```
// Excepciones Personalizadas
```

```
class DuracionInvalidaException extends Exception {
    public DuracionInvalidaException(String mensaje) {
        super(mensaje);
    }
}

class PresupuestoInvalidoException extends Exception {
    public PresupuestoInvalidoException(String mensaje) {
        super(mensaje);
    }
}

// Clase Proyecto
class Proyecto {
    private int id;
    private String nombre;
    private int duracion;
    private double presupuesto;

    public Proyecto(int id, String nombre, int duracion, double presupuesto) {
        this.id = id;
        this.nombre = nombre;
        this.duracion = duracion;
        this.presupuesto = presupuesto;
    }

    public double calcularCostoTotal() throws DuracionInvalidaException,
PresupuestoInvalidoException {
        if (duracion <= 0) {
            throw new DuracionInvalidaException("La duración del proyecto debe
ser mayor a 0.");
        }
        if (presupuesto <= 0) {
            throw new PresupuestoInvalidoException("El presupuesto debe ser
mayor a 0.");
        }
        return duracion * presupuesto;
    }
}

// Clase GestorProyectos
class GestorProyectos {
    public void procesarProyecto(Proyecto proyecto) {
        try {
            double costoTotal = proyecto.calcularCostoTotal();
            System.out.println("El costo total del proyecto es: " + costoTotal);
        } catch (DuracionInvalidaException | PresupuestoInvalidoException e) {
            System.out.println("Error al procesar el proyecto: " +
e.getMessage());
        } finally {
            System.out.println("Operación de proyecto finalizada.");
        }
    }
}

// Clase Principal
public class Main {
    public static void main(String[] args) {
        Proyecto proyecto1 = new Proyecto(1, "Proyecto A", 12, 10000.0);
        Proyecto proyecto2 = new Proyecto(2, "Proyecto B", -6, 5000.0); //
Duración inválida
        Proyecto proyecto3 = new Proyecto(3, "Proyecto C", 6, -5000.0); //
Presupuesto inválido
```

```
        GestorProyectos gestor = new GestorProyectos();

        gestor.procesarProyecto(proyecto1);
        gestor.procesarProyecto(proyecto2);
        gestor.procesarProyecto(proyecto3);
    }
}
```

Estos tres ejercicios adicionales siguen el mismo patrón de trabajo con excepciones personalizadas para garantizar que el código sea robusto y maneje adecuadamente los posibles errores. Puedes probarlos de forma similar a los ejercicios anteriores, asegurándote de capturar y gestionar correctamente los casos de error.