Metodología de la Programación2

Apuntes hechos por Brianda García, Lorena Ballesteros, Lía de Miguel y Marina Sancho

Excepciones en Programación

1. Introducción a las Excepciones

En programación, una **excepción** es un evento anómalo que interrumpe el flujo normal de ejecución de un programa. Puede ser causada por errores de entrada/salida, acceso a datos incorrectos, operaciones matemáticas inválidas, entre otros.

El manejo adecuado de excepciones es fundamental para evitar que los programas se bloqueen de manera inesperada y para garantizar que los recursos utilizados (archivos, conexiones de red, bases de datos) se liberen correctamente.

1.1. ¿Por qué son importantes las excepciones?

Si no se manejan correctamente, las excepciones pueden dejar conexiones abiertas, archivos sin cerrar o incluso corromper datos. Algunos de los problemas más comunes que pueden surgir debido a una mala gestión de excepciones incluyen:

- Archivos abiertos sin cerrar correctamente.
- Conexiones de bases de datos o redes sin liberar.
- Programas que terminan abruptamente sin devolver mensajes de error informativos.

1.2. Ejemplo de problema sin manejo de excepciones

```
import java.io.File;
import java.util.Scanner;

public class SinManejoExcepciones {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(new File("archivo.txt")); // Error si el archivo no existe
        System.out.println("Archivo leído correctamente");
    }
}
```

Si el archivo "archivo.txt" no existe, el programa se detendrá de inmediato y mostrará una excepción de tipo FileNotFoundException, sin dar oportunidad al usuario de corregirlo.

2. Ventajas del Manejo de Excepciones

El uso de excepciones tiene varias ventajas clave:

2.1. Declaración de excepciones en los métodos

Los métodos pueden especificar qué excepciones pueden lanzar, lo que ayuda a estructurar mejor el código e informar a los desarrolladores qué errores pueden ocurrir.

Ejemplo:

```
public void abrirArchivo(String nombreArchivo) throws FileNotFoundException {
   Scanner scanner = new Scanner(new File(nombreArchivo));
}
```

Aquí, throws FileNotFoundException indica que el método puede generar esa excepción.

2.2. Mejor legibilidad y mantenimiento del código

Separar la lógica principal del manejo de errores mejora la organización del código.

Ejemplo:

```
try {
   realizarOperacion();
   } catch (Exception e) {
   System.out.println("Ocurrió un error: " + e.getMessage());
}
```

Esto permite que realizarOperacion() tenga su propia lógica sin estar saturada con código de manejo de errores.

2.3. Obligatoriedad de manejar excepciones

Lenguajes como Java requieren que ciertas excepciones sean manejadas o declaradas, lo que obliga a escribir código más robusto.

3. Tipos de Excepciones

Las excepciones en Java se dividen en varias categorías.

3.1. Excepciones Verificadas (Checked Exceptions)

Son aquellas que deben ser capturadas con un bloque try-catch o declaradas con throws en la firma del método.

Ejemplos:

- IOException: Error en operaciones de entrada/salida.
- SQLException: Error en acceso a bases de datos.
- FileNotFoundException: Archivo no encontrado.

```
try {
   Scanner scanner = new Scanner(new File("datos.txt"));
} catch (FileNotFoundException e) {
   System.out.println("El archivo no existe");
}
```

3.2. Excepciones No Verificadas (Unchecked Exceptions)

Son errores que derivan de problemas en la lógica del programa y no requieren ser capturados explícitamente.

Ejemplos:

- NullPointerException: Se intenta acceder a un objeto null.
- ArithmeticException: División por cero.
- ArrayIndexOutOfBoundsException: Índice de arreglo fuera de límites.

```
int[] numeros = {1, 2, 3};
System.out.println(numeros[5]); // Error: índice fuera de rango
```

3.3. Errores del Sistema (Errors)

Son fallos críticos en la máquina virtual de Java o el sistema operativo.

Ejemplos:

- StackOverflowError: Llamadas recursivas sin condición de salida.
- OutOfMemoryError: Se agotó la memoria del programa.

4. Manejo de Excepciones

4.1. Uso de try-catch

```
try {
   int resultado = 10 / 0;
} catch (ArithmeticException e) {
   System.out.println("Error: División por cero");
}
```

4.2. Uso de finally

El bloque finally siempre se ejecuta, incluso si ocurre una excepción.

```
try {
   File archivo = new File("datos.txt");
   Scanner lector = new Scanner(archivo);
} catch (FileNotFoundException e) {
   System.out.println("Archivo no encontrado");
} finally {
   System.out.println("Ejecución finalizada");
}
```

4.3. Propagación de Excepciones

Las excepciones pueden propagarse a través de la pila de llamadas.

```
public void metodoA() throws IOException {
   metodoB();
}

public void metodoB() throws IOException {
   metodoC();
}

public void metodoC() throws IOException {
   throw new IOException("Error en método C");
}
```

Si metodoC() lanza una excepción, esta se propagará hasta metodoA().

5. Creación de Excepciones Personalizadas

Se pueden definir nuevas excepciones extendiendo la clase Exception o RuntimeException.

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

Uso de la excepción personalizada:

```
public void validarEdad(int edad) throws MiExcepcion {
   if (edad < 18) {
      throw new MiExcepcion("Debes ser mayor de edad");
   }
}
```

6. Comparación entre Excepciones Verificadas y No Verificadas

CARACTERÍSTICAS	EXCEPCIONES VERIFICADAS	EXCEPCIONES NO VERIFICADAS
Deben ser capturadas o declaradas	Sí	No
Se derivan de	Exception	RuntimeException
Ejemplo	IOException	NullPointerException

7. Mejores Prácticas en el Uso de Excepciones

1. **No atrapar excepciones genéricas como Exception**, a menos que sea necesario.
2. **Usar excepciones específicas** para mejorar la claridad.
3. **Evitar el uso excesivo de excepciones** como control de flujo.
4. **Cerrar recursos en finally o usar try-with-resources**:

```
try (Scanner scanner = new Scanner(new File("datos.txt"))) {
    System.out.println(scanner.nextLine());
} catch (FileNotFoundException e) {
    System.out.println("Archivo no encontrado");
}
```

Esto evita fugas de recursos.

8. Programación de Interfaces y Manejo de Eventos

8.1. Componentes Gráficos

En la programación de interfaces gráficas (GUI), los programas interactúan con el usuario a través de **ventanas** y **componentes** visuales. Estos elementos permiten al usuario introducir datos, recibir información y ejecutar acciones.

Principales Componentes:

- **Ventanas (Window):** Área principal de interacción del usuario.
- **Etiquetas (Label):** Muestran texto estático.
- **Botones (Button):** Ejecutan acciones al ser pulsados.
- **Campos de texto (TextField):** Permiten al usuario introducir texto.
- **Formularios (Forms):** Conjunto estructurado de componentes para entrada/salida.

8.2. Eventos y Manejadores

Los **eventos** son acciones generadas por el usuario (como un clic o movimiento del ratón) o por el sistema. Cada objeto gráfico puede responder a múltiples eventos.

Tipos de Eventos Comunes:

1. **Ventana:**

- **Propiedades:** visible, tamaño, posición, tipo.
- **Eventos:** apertura/cierre, redimensionamiento, minimización.
- **Contenido:** componentes (botones, textos), diálogos emergentes.

2. **Ratón (Mouse):**

- **Atributos:** ícono, color, posición (x, y), DPI, tamaño.
- **Eventos:** clic izquierdo (pulsador 1), clic derecho (pulsador 2), clic central (pulsador 3), doble clic, movimiento.

3. **Teclado (Keyboard):**

- **Eventos:** pulsación de teclas, combinación de teclas (atajos), entrada de texto.

4. **Evento OnClick:**

- Es el más común y se produce cuando el usuario hace clic en un componente.
- Requiere implementar un **manejador de eventos** que defina qué acción realizar.

8.3. Adaptadores de Ventanas

Cuando se gestionan múltiples ventanas o se desea extender el comportamiento de los eventos sin sobrescribir todos, se utilizan **clases adaptadoras**. Estas clases permiten implementar solo los métodos necesarios de un evento complejo (como los de ventana o ratón), sin tener que definir todos.

Ejemplo visual: Se muestra una interfaz con reloj y botones que reaccionan a eventos usando adaptadores personalizados entre ventanas.

8.4. Ejemplo de Clase con Evento

```
class Reloj {
   DateTime d;
   String s;
   void setD(DateTime hora) {
      this.d = hora;
      t.setText(hora.toString()); // Actualiza texto
      c.setTime(hora);          // Actualiza vista del reloj
   }
}
```

Aquí, se actualiza tanto el valor visual (t) como el componente lógico (c) del reloj al recibir una nueva hora.

9. Patrón de Diseño MVC (Modelo-Vista-Controlador)

9.1. ¿Qué es MVC?

Es un patrón arquitectónico que separa la lógica de una aplicación en tres componentes interrelacionados, lo que facilita la escalabilidad, el mantenimiento y la reutilización del código.

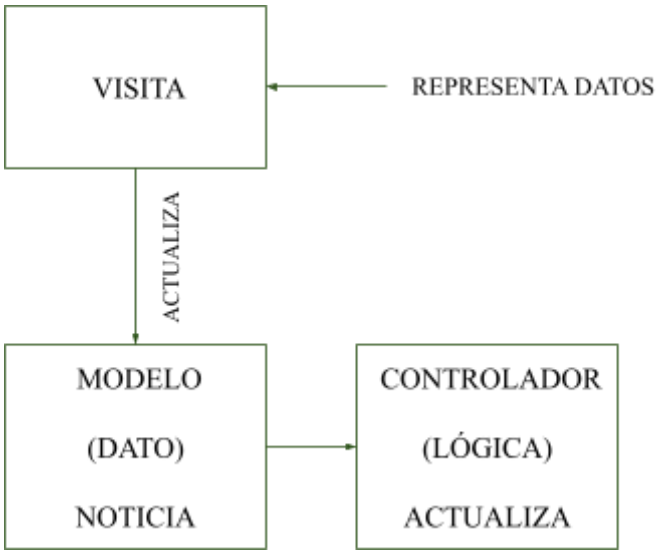
Componentes de MVC:

- **Modelo (Model):** Representa los datos y la lógica de negocio. Notifica los cambios a la vista.
- **Vista (View):** Representa visualmente los datos del modelo. Es la interfaz con el usuario.
- **Controlador (Controller):** Gestiona la entrada del usuario y coordina acciones entre el modelo y la vista.

Funcionamiento Básico:

- El **usuario interactúa con la vista**.
- El **controlador interpreta la acción del usuario** y actualiza el modelo.
- El **modelo notifica a la vista** los cambios.
- La **vista se actualiza** automáticamente.

9.2. Diagrama MVC Básico



9.3. MVC Complejo con Observadores (UML)

En implementaciones avanzadas, MVC se basa en el patrón **Observador**, donde la vista se suscribe al modelo para recibir notificaciones de cambio.

Clases y Relaciones:

- **Model:**
    - **Métodos:** attach(view), detach(view), notify().
- **ConcreteModel:**
    - **Estado:** subjectState.
    - **Métodos:** getState(), setState().
- **View:**
    - **Métodos:** update(), contextInterface().

- **ConcreteView:**
    - **Atributos:** observerState.
    - **Métodos:** update().
- **Controller:**
    - **Métodos:** algorithmInterface().
- **ConcreteController:**
    - Define la lógica de respuesta a eventos del usuario.

10. Persistencia de Datos

10.1. ¿Qué es la persistencia?

La persistencia se refiere a la capacidad de guardar el estado de los objetos de un programa para su uso posterior. Permite que los datos sobrevivan al cierre del programa.

Ejemplo de Clase Persistente:

```
class X {
   String nombre;
   void dimeNombre() {
      System.out.println(nombre);
   }
}
```

10.2. Formatos de Datos Persistentes

Los datos pueden guardarse en dos grandes tipos de archivos:

1. Binarios:

- Almacenan la información en forma de bytes.
- Más eficientes pero menos legibles.
- **Ej:** archivos .dat, registros binarios.

2. Texto:

- Legibles por humanos.
- **Ej:** .txt, .csv, .json, .xml.

10.3. Interoperabilidad

La **interoperabilidad** permite que diferentes sistemas y lenguajes compartan información utilizando formatos comunes.
Formatos populares:

- **JSON**: Ideal para web, APIs.
- **XML**: Muy usado en servicios SOA.
- **YAML**: Utilizado en contenedores y configuración de infraestructuras (ej. Docker).

10.4. Ejemplo JSON

```
{
"Nombre": "Antonio",
"Edad": 48,
"C": [
  { "x": 0, "y": 1, "radio": 7 },
  { "x": 1, "y": 3, "radio": 9 }
]
}
```

JSON organiza los datos en objetos y arreglos con claves y valores.

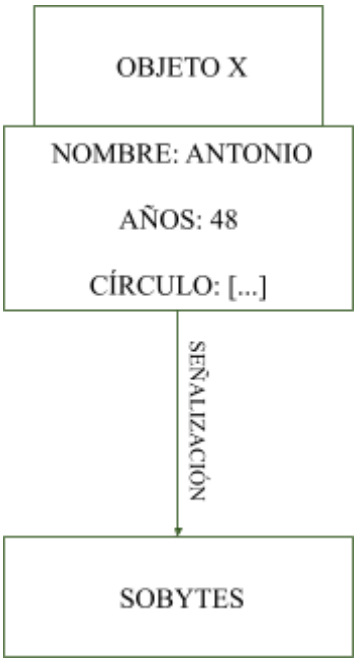
10.5. Ejemplo XML

```
<X>
 <nombre>Antonio</nombre>
 <edad>48</edad>
 <C type="Array" num="2">
  <Círculo>
   <x type="int">0</x>
   <y type="int">1</y>
   <radio type="int">7</radio>
  </Círculo>
  <Círculo>
   <x>1</x>
   <y>3</y>
   <radio>9</radio>
  </Círculo>
 </C>
</X>
```

XML es más detallado, ideal para estructurar jerarquías complejas.

10.6. Esquema de Serialización

Al guardar un objeto se convierte a una secuencia de bytes:



10.7. Lenguajes de Estructura:

- **XKHL**: Adaptado para estructuras jerárquicas.
- **HTML**: Para presentación en la web.
- **MD (Markdown)**: Para documentación sencilla.

10.8. Aplicación práctica en código

```
X variable = new X();
variable.nombre = "Antonio";
manejador.guardar(variable); // Persistencia del objeto
```

Aquí, manejador puede ser un sistema que guarde en JSON, XML o binario.