TECNOLÓGICA DEL ORIENTE
INSTITUCIÓN DE EDUCACIÓN SUPERIOR

PROGRAMACIÓN ORIENTADA A OBJETOS

PRUEBAS Y DEPURACIÓN DE EXCEPCIONES

PRUEBAS Y DEPURACIÓN DE EXCEPCIONES

Las pruebas y depuración de excepciones son aspectos cruciales en el desarrollo de software robusto y confiable. Las pruebas permiten verificar que el manejo de excepciones funciona correctamente y que el programa responde apropiadamente a situaciones excepcionales. Por otro lado, la depuración ayuda a identificar y solucionar problemas relacionados con las excepciones.

En este apartado, se explorarán las técnicas básicas de depuración, el uso de herramientas del entorno de desarrollo (IDE), la identificación y corrección de errores comunes, el registro de errores (logging) y la integración de pruebas unitarias para validar el manejo de errores.

Para ampliar la temática de depuración en Java, le invitamos a ver el siguiente vídeo.

Bueno. D. (2021, 19 de marzo). Introducción a la Depuración en Java (usando Eclipse). [Vídeo]. YouTube https://youtu.be/5FTFSQn-I7M

Técnicas básicas de depuración (puntos de ruptura, trazas)

La depuración es el proceso de identificar y corregir errores en el código. Dos técnicas básicas de depuración son: los puntos de ruptura (breakpoints) y las trazas (tracing).

Los puntos de ruptura permiten detener la ejecución del programa en una línea específica de código. Esto permite examinar el estado de las variables, el flujo de ejecución y detectar errores. Los IDEs modernos proporcionan herramientas para establecer puntos de ruptura y examinar el estado del programa durante la depuración.

Las trazas, por otro lado, implican la inserción de sentencias de impresión (como System.out.println() en Java) en puntos estratégicos del código para rastrear el flujo de ejecución y los valores de las variables. Esto ayuda a comprender el comportamiento del programa y detectar errores.

Ceballos Sierra (2018), propone las técnicas de depuración, como los puntos de ruptura y las trazas, son herramientas fundamentales para identificar y solucionar problemas en el código.

En el desarrollo de software, es importante utilizar técnicas de depuración de manera efectiva para localizar y corregir errores de manera eficiente. Los puntos de ruptura y las trazas permiten examinar el estado del programa en puntos específicos y comprender el flujo de ejecución.

Figura 1. Ejemplo de uso de puntos de ruptura y trazas en Java

```java
public void processData(String data) {
    try {
        System.out.println("Procesando datos: " + data); // Traza
        // Punto de ruptura establecido en la siguiente línea
        int result = Integer.parseInt(data);
        System.out.println("Resultado: " + result); // Traza
    } catch (NumberFormatException e) {
        System.out.println("Error de formato numérico: " + e.getMessage()); // Traza
    }
}
```

Uso de herramientas del entorno de desarrollo (IDE)

Los entornos de desarrollo integrado (IDE) son una parte fundamental en el kit de herramientas de cualquier programador. Proporcionan una gran variedad de herramientas y funcionalidades que facilitan enormemente la depuración y prueba de excepciones en el código.

Algunos de los IDE más populares, como Eclipse, IntelliJ IDEA y NetBeans, contienen amplias características avanzadas de depuración que pueden ahorrar horas de sufrimiento tratando de encontrar esos errores esquivos. Imagine tener un detective privado que brinda apoyo en cualquier momento para investigar las entrañas del código.

Estas herramientas permiten hacer funciones como:

Establecer puntos de ruptura	Es como decirle al programa "¡Alto ahí, compañero! Déjame ver lo que llevas en la bolsa." El programa se detiene donde se ha puesto el punto de ruptura y se puede examinar tranquilamente el estado de todas las variables en ese momento. Es como una máquina del tiempo para el código
Examinar variables	Hablando de variables, los IDEs permiten ver lo que contienen en cualquier momento de la ejecución del programa. Es como tener rayos X para inspeccionar las entrañas de la aplicación
Recorrer el código paso a paso	Esto es especialmente útil cuando se trata de entender el flujo de un programa. Se puede avanzar línea a línea y ver cómo cambia el estado en cada paso. Es como ser el director de una película del código y poder gritar "¡Corten!" en cualquier momento
Inspeccionar la pila de llamadas	Cuando un programa lanza una excepción, la pila de llamadas muestra el camino que ha tomado la ejecución hasta llegar a ese punto. Es como tener un GPS que dice exactamente cómo ha llegado a ese desolado paraje de la aplicación

Para Moreno Pérez (2015) el uso efectivo de las herramientas de depuración del IDE puede ahorrar tiempo y esfuerzo en la identificación y corrección de errores, dado que proporcionan una visión detallada del estado del programa en tiempo de ejecución.

Ejemplo. Suponga que está desarrollando una aplicación de gestión de inventario y de repente, al intentar actualizar la cantidad de un producto, salta una excepción. En lugar de entrar en pánico e imprimir un montón de declaraciones de registro a lo loco, se puede establecer tranquilamente un punto de ruptura en la línea donde ocurre la actualización, iniciar el programa en modo de depuración y cuando la ejecución se detenga en ese punto, examinar cuidadosamente los valores de todas las variables relevantes. Quizás se descubra que el problema está en los datos de entrada, o en la lógica de actualización, o tal vez sea una excepción que no se está manejando correctamente. El IDE es el mejor amigo en estos casos.

Para practicar estas herramientas, se propone un pequeño ejercicio:

1. Abra su IDE favorito y crea un nuevo proyecto.
2. Escriba un método que calcule el promedio de un arreglo de números y lance una excepción si el arreglo está vacío.

3. Establezca un punto de ruptura en la línea donde se calcula el promedio.
4. Ejecute el programa en modo de depuración y examine los valores de las variables cuando se alcanza el punto de ruptura.
5. Verifique si la excepción se lanza correctamente cuando el arreglo está vacío.

Jugar con estas herramientas es la mejor manera de aprender a usarlas efectivamente. A medida que las use, comenzará a preguntarse cómo se pudo vivir sin ellas.

Identificación y corrección de errores comunes

Como previamente se ha revisado o practicado la depuración de un IDE, es importante hablar de algunos de los errores más comunes que se encuentran al manejar excepciones y cómo solucionarlos.

1. **No manejar excepciones:** Este es quizás el error más común y el más peligroso. Es como tener un perro guardián que ladra cuando hay un intruso, pero no hacer nada al respecto. Si no se manejan las excepciones, el programa podría terminar abruptamente sin dar ninguna pista de lo que salió mal.
2. **Capturar excepciones demasiado generales:** Esto es como usar un paraguas en una tormenta de nieve. Puede que cubra, pero no es la herramienta adecuada para el trabajo. Cuando se capturan excepciones usando la clase Exception, se está atrapando todo tipo de excepciones, lo que puede dificultar la identificación y manejo de errores específicos.
3. **Ignorar excepciones:** Este es el equivalente a poner una cinta adhesiva sobre la luz de "revisar el motor" en el coche. Puede que ya no se vea la luz, pero el problema sigue ahí. Nunca se debe capturar una excepción y no hacer nada con ella. Como mínimo, se debe registrar el error.
4. **Uso inadecuado de bloques try-catch:** A veces, en el entusiasmo por manejar excepciones, se puede ir un poco overboard y terminar con un código que parece un laberinto de bloques try-catch anidados. Esto no solo hace que el código sea difícil de leer, sino que también puede ocultar errores.
5. **No liberar recursos en el bloque finally:** Imagine que alquila un libro de la biblioteca y nunca lo devuelve. Los recursos, como archivos o conexiones de base de datos, son similares. Si no se cierran apropiadamente, pueden causar problemas de rendimiento e incluso bloqueos en el programa.

López Goytia (2015) recuerda que identificar y corregir errores comunes relacionados con el manejo de excepciones es esencial para crear aplicaciones robustas e igualmente evitar problemas difíciles de depurar.

Ejemplo. Suponga que tiene este código:

Figura 2. Ejemplo de errores comunes.

```java
try {
    int result = divideNumbers(10, 0);
    System.out.println("Resultado: " + result);
} catch (Exception e) {
    System.out.println("Ocurrió un error.");
}
```

Aquí se están cometiendo dos errores. Primero, se está capturando Exception, que es demasiado general. Segundo, el mensaje de error no dice nada útil.

Para solucionar esto, se puede capturar la excepción específica que se espera (en este caso, ArithmeticException) y proporcionar un mensaje de error más descriptivo:

Figura 3. Ejemplo de uso de ArithmeticException.

```java
try {
    int result = divideNumbers(10, 0);
    System.out.println("Resultado: " + result);
} catch (ArithmeticException e) {
    System.out.println("Error: División por cero. " + e.getMessage());
}
```

Ahora practique con este ejercicio:

1. Escriba un método que lea un archivo de texto y cuente el número de líneas.
2. Intencionalmente, no maneje la excepción FileNotFoundException.
3. Ejecute el programa y observe lo que sucede cuando el archivo no existe.
4. Ahora, agregue un bloque try-catch para manejar la excepción FileNotFoundException y proporcione un mensaje de error adecuado.
5. Ejecute el programa de nuevo y observe la diferencia.

Recuerde, el manejo de excepciones es una habilidad que se desarrolla con la práctica. Cuanto más código escriba y depure, será más competente identificando y corrigiendo estos errores comunes.

Además, le invitamos a utilizar la inteligencia artificial para encontrar y resolver los errores en su código, le será de gran ayuda.

a continuación presento algunas de las herramientas de inteligencia artificial (IA) más destacadas para la depuración de código en 2025, junto con sus respectivas descripciones y enlaces para consulta:

GitHub Copilot	• **Descripción:** Desarrollado por GitHub y OpenAI, GitHub Copilot es un asistente de codificación que sugiere líneas completas de código y funciones basadas en el contexto del editor. Aunque su enfoque principal es la autocompletación, también ayuda en la identificación y corrección de errores comunes durante la escritura del código. • **Enlace:** https://github.com/features/copilot

SnykCode (DeepCode)	• **Descripción:** SnykCode, impulsado por DeepCode AI, ofrece análisis de código en tiempo real para detectar errores y vulnerabilidades de seguridad. Utiliza inteligencia artificial para proporcionar sugerencias precisas y automáticas que ayudan a mantener la calidad y seguridad del código. • **Enlace:** https://snyk.io/product/snyk-code
ChatDBG	• **Descripción:** ChatDBG es un asistente de depuración que integra modelos de lenguaje grandes (LLMs) para interactuar con el programador durante el proceso de depuración. Permite realizar análisis de causas raíz, inspeccionar el estado del programa y sugerir soluciones de manera conversacional. • **Enlace:** https://arxiv.org/abs/2403.16354arXiv
Workik AI Debugger	• **Descripción:** Workik ofrece una herramienta de depuración impulsada por IA que permite a los desarrolladores identificar y corregir errores en su código de manera eficiente. La plataforma proporciona sugerencias contextuales y soluciones automatizadas para diversos lenguajes de programación. • **Enlace:** https://workik.com/ai-code-debugger
AI Code Debug	• **Descripción:** AI Code Debug es una herramienta en línea que utiliza inteligencia artificial para ayudar a los desarrolladores a depurar código rápidamente. Los usuarios pueden ingresar fragmentos de código y recibir análisis detallados y sugerencias para corregir errores. • **Enlace:** https://zzzcode.ai/code-debug
Google Bard	• **Descripción:** Google Bard es un chatbot de IA que ha incorporado capacidades de codificación, incluyendo la generación y depuración de código en múltiples lenguajes. Permite a los desarrolladores interactuar de manera conversacional para obtener soluciones y explicaciones sobre su código. • **Enlace:** https://bard.google.com/

Para ampliar los conceptos sobre tester o ingeniero de pruebas, le invitamos a ver el siguiente vídeo.

Software Guru. (2014, 29 de octubre). ¿Cómo convertirse en un tester de verdad?. [Vídeo]. https://youtu.be/UMMMWaP7ejU

Registro de errores (logging)

Hasta ahora, se ha hablado sobre cómo manejar excepciones cuando ocurren. Pero ¿qué pasa después? ¿Cómo se puede aprender de estos errores y evitar que se repitan en el futuro? Aquí es donde entra en juego el registro de errores o logging.

El logging es como tener un diario para la aplicación. Registra eventos importantes, incluyendo errores, para que se pueda revisar lo que sucedió después. Es una herramienta invaluable para entender el comportamiento del programa, especialmente cuando las cosas van mal.

En Java, se tiene la clase Logger en el paquete java.util.logging para hacer esto. Entre sus funcionalidades se encuentran:

- ☑ Configurar diferentes niveles de registro (¿se quiere saber todo o solo los eventos críticos?)
- ☑ Enviar mensajes de registro a diferentes destinos (consola, archivos, bases de datos, etc.)
- ☑ Controlar el formato de los mensajes de registro

Como plantea Oviedo Regino (2015), el registro de errores es una práctica fundamental en el desarrollo de software robusto y mantenible, ya que proporciona una visión detallada de lo que sucede en el sistema durante su ejecución.

Figura 4. Uso de Logger para registrar un mensaje informativo.

```
java
import java.util.logging.*;

public class EjemploLogging {
    private static final Logger logger = Logger.getLogger(EjemploLogging.class.getName());

    public static void main(String[] args) {
        try {
            int result = divideNumbers(10, 0);
            logger.info("Resultado: " + result);
        } catch (ArithmeticException e) {
            logger.severe("Error de división por cero: " + e.getMessage());
        }
    }

    private static int divideNumbers(int a, int b) {
        return a / b;
    }
}
```

En este ejemplo, se usa el Logger para registrar un mensaje informativo si la división tiene éxito y un mensaje severo si se obtiene una ArithmeticException.

El logging es especialmente útil en aplicaciones grandes o en entornos de producción donde no se puede simplemente adjuntar un depurador y empezar a depurar. Los registros pueden ser la única pista sobre lo que salió mal.

Ejercicio:

1. Agregue registro de errores al ejercicio anterior de lectura de archivos.
2. Configure el logger para que envíe los mensajes de registro tanto a la consola como a un archivo de registro.
3. Ejecute el programa y verifique que los mensajes de registro se capturen correctamente en la consola y en el archivo de registro.
4. Simule diferentes escenarios de error (como archivo no encontrado, permisos insuficientes, etc.) y verifique que los errores se registren adecuadamente.

Recuerde, un buen logging puede ahorrar horas de depuración y puede ser la diferencia entre encontrar un error crítico antes o después de que la aplicación se despliegue en producción.

Integración de pruebas unitarias para validación del manejo de errores

Adicional al registro de errores para el manejo de excepciones que permitan detectar dichos errores antes de que el código llegue a producción, existen otras alternativas que facilitan validar el manejo de errores, entre las cuales se encuentran las pruebas unitarias.

Las pruebas unitarias son como un escuadrón de control de calidad para el código. Su trabajo es asegurarse de que cada unidad del código (generalmente una función o un método) se comporta como se espera, incluso cuando las cosas van mal.

En Java, JUnit es el framework más popular para escribir y ejecutar pruebas unitarias. Permite hacer cosas como:

- Verificar que un método devuelve el resultado esperado cuando se le dan entradas válidas.
- Verificar que un método lanza las excepciones esperadas cuando se le dan entradas inválidas.
- Configurar y limpiar el estado antes y después de cada prueba.

Ruiz Rodríguez (2009) destaca que las pruebas unitarias del manejo de errores permiten verificar que las excepciones se lancen en las situaciones esperadas, que se capturen y manejen correctamente, y que el flujo del programa siga el camino adecuado después de una excepción.

Figura 5. Ejemplo de pruebas unitarias.

```java
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class CalculadoraTest {

    @Test
    public void testDivisionPorCero() {
        Calculadora calculadora = new Calculadora();
        Assertions.assertThrows(ArithmeticException.class, () -> {
            calculadora.dividir(10, 0);
        });
    }

    @Test
    public void testDivisionValida() {
        Calculadora calculadora = new Calculadora();
        int resultado = calculadora.dividir(10, 2);
        Assertions.assertEquals(5, resultado);
    }
}
```

En este ejemplo, se tienen dos pruebas unitarias para el método dividir de la clase Calculadora. La primera prueba verifica que se lance una ArithmeticException cuando se intenta dividir por cero. La segunda prueba verifica que una división válida devuelva el resultado correcto.

La ventaja de las pruebas unitarias es que se pueden automatizar. Cada vez que se realice un cambio en el código, se puede ejecutar pruebas unitarias para asegurarse de que no se hayan introducido nuevos errores.

Ejercicio:

1. Escribe pruebas unitarias para el ejercicio anterior de lectura de archivos.
2. Verifique que se lance una excepción FileNotFoundException cuando intentes leer un archivo que no existe.
3. Verifique que el método de lectura de archivos cuente correctamente el número de líneas cuando se le proporciona un archivo válido.
4. Ejecute las pruebas y asegurarse de que pasen.

Las pruebas unitarias son la primera línea de defensa contra los errores. Cuantas más pruebas se tengan y cuanto más cubran, más confianza se tendrá en el código.

BIBLIOGRAFÍA

- Bueno, D. (2021, 19 de marzo). Introducción a la depuración en Java (usando Eclipse) [Vídeo]. YouTube. https://youtu.be/5FTFSQn-I7M
- Ceballos Sierra, F. J. (2018). Programación orientada a objetos con C++ (5ª ed.). RA-MA Editorial. https://elibro.net/es/ereader/tecnologicadeloriente/106519
- López Goytia, J. L. (2015). Programación orientada a objetos C++ y Java: un acercamiento interdisciplinario. Grupo Editorial Patria. https://elibro.net/es/ereader/tecnologicadeloriente/39461
- Moreno Pérez, J. C. (2015). Programación orientada a objetos. RA-MA Editorial. https://elibro.net/es/ereader/tecnologicadeloriente/106461
- Oviedo Regino, E. M. (2015). Lógica de programación orientada a objetos. Ecoe Ediciones. https://elibro.net/es/ereader/tecnologicadeloriente/70431
- Ruiz Rodríguez, R. (2009). Fundamentos de la programación orientada a objetos: una aplicación a las estructuras de datos en Java. El Cid Editor. https://elibro.net/es/ereader/tecnologicadeloriente/34869
- Software Guru. (2014, 29 de octubre). ¿Cómo convertirse en un tester de verdad? [Vídeo]. YouTube. https://youtu.be/UMMMWaP7ejU