Taller Investigación programación II

PRUEBAS UNITARIAS

Una prueba unitaria consiste en un fragmento de código que evalúa la precisión de otro fragmento más pequeño y aislado de código en una aplicación, típicamente una función o un método. Su propósito es garantizar que este fragmento se comporte como se espera, según la lógica diseñada por el desarrollador. Una característica clave de las pruebas unitarias es que interactúan con el código únicamente a través de entradas y salidas específicas, asegurando que las salidas sean las previstas para ciertas entradas.

Es esencial que las pruebas unitarias se realicen de manera independiente, sin depender de otros elementos del sistema. Esto implica que no deben basarse en datos externos, como bases de datos o conexiones de red. En su lugar, se utilizan datos simulados para recrear estas interacciones externas. Las pruebas unitarias son más fáciles de implementar en fragmentos de código pequeños y simples, ya que simplifican la validación y garantizan un comportamiento consistente.

En las pruebas unitarias, es común aplicar estrategias como verificar que el programa realice los cálculos correctos y siga el flujo adecuado en el código al recibir una entrada, así como garantizar que el sistema responda correctamente a estas entradas. Además, se enfoca en manejar apropiadamente los errores del sistema en circunstancias como problemas con las entradas, bloqueos del software o solicitudes de entradas adicionales al usuario.

Un ejemplo de cómo se ven las pruebas unitarias en Java:

Tenemos el siguiente Código:

```java
public class Rectangulo {

    public double calcularArea(double longitud, double anchura) {
        return longitud * anchura;
    }
}
```

CARLOS LEONARDO VEGA - 02220132011

Ahora, vamos a escribir pruebas unitarias para este método utilizando JUnit:

```java
import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class RectanguloTest {
    @Test
    public void testCalcularArea() {
        Rectangulo rectangulo = new Rectangulo();
        double longitud = 5;
        double anchura = 3;
        double resultadoEsperado = 15;
        double resultado = rectangulo.calcularArea(longitud, anchura);
        assertEquals(resultadoEsperado, resultado, 0.001);

    }
}
```

En este ejemplo:

- Importamos las clases necesarias de JUnit.
- Creamos una clase llamada RectanguloTest que contiene un método de prueba testCalcularArea.
- Dentro del método de prueba, creamos una instancia de la clase Rectangulo.
- Definimos los valores de longitud y anchura.
- Calculamos el área utilizando el método calcularArea y comparamos el resultado con el valor esperado utilizando assertEquals.

¿QUÉ ES UN REPORTE DE COBERTURA O DE COVERAGE?

Un reporte de cobertura, o también conocido como reporte de Coverage, es un documento esencial en el proceso de desarrollo de software que brinda una visión detallada del grado en que las pruebas automatizadas cubren el código fuente de un programa. Este informe se genera durante las fases de prueba y proporciona información crítica sobre qué parte del código ha sido ejecutada y qué parte no durante las pruebas.

En términos simples, el reporte de cobertura muestra qué tan completa y exhaustivamente se ha evaluado el código mediante pruebas automatizadas. Para lograr esto, el reporte presenta una serie de métricas que indican la proporción del código que ha sido ejecutada, generalmente expresada como un porcentaje del total.

Este informe es invaluable para los equipos de desarrollo, ya que les permite identificar áreas específicas del código que no han sido suficientemente probadas. Al proporcionar una visión detallada de la cobertura de código, el reporte de cobertura ayuda a los desarrolladores a enfocar sus esfuerzos de prueba en las áreas que necesitan más atención, lo que a su vez puede ayudar a mejorar la calidad y la confiabilidad del software final.

Los reportes de cobertura suelen presentarse en diferentes formatos, como tablas, gráficos o resúmenes visuales, lo que facilita su comprensión y su uso por parte de los miembros del equipo de desarrollo. En última instancia, el reporte de cobertura es una herramienta fundamental en el proceso de desarrollo de software, ya que proporciona información valiosa sobre la efectividad de las pruebas automatizadas y contribuye a garantizar la calidad del producto final.

CARLOS LEONARDO VEGA - 02220132011

EXCEPCIONES EN JAVA

En Java, una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de ejecución. Puede ser causada por diversas razones, como errores de programación, condiciones inesperadas o problemas en tiempo de ejecución. Las excepciones se manejan utilizando el mecanismo de manejo de excepciones, que implica el uso de bloques try-catch o lanzando excepciones con la palabra clave throw.

En Java, las excepciones se dividen en dos categorías principales: excepciones verificadas y excepciones no verificadas.

1. Excepciones verificadas (Checked Exceptions):

- Estas son excepciones que el compilador obliga a manejar o declarar en la firma del método.
- Son subclases de la clase Exception, pero no de RuntimeException.
- Algunos ejemplos comunes de excepciones verificadas son IOException, SQLException y ClassNotFoundException.

2 Excepciones no verificadas (Unchecked Exceptions):

- También conocidas como excepciones de tiempo de ejecución.
- No se requiere manejar explícitamente en el código, aunque es posible hacerlo.
- Son subclases de RuntimeException.
- Algunos ejemplos comunes de excepciones no verificadas son NullPointerException, ArrayIndexOutOfBoundsException y ArithmeticException.

Ambos tipos de excepciones son importantes en el manejo de errores en Java, pero la diferencia clave radica en cómo se manejan y en si el compilador las obliga a ser manejadas o declaradas.

CARLOS LEONARDO VEGA - 02220132011

1- IOException:

En este ejemplo, intentamos abrir el archivo "archivo.txt" utilizando un BufferedReader. Sin embargo, si el archivo no existe o no se puede abrir por alguna razón (por ejemplo, si no tenemos permiso para acceder al archivo), se lanzará una IOException.

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Main {
    Run | Debug
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(new FileReader(fileName:"archivo.txt"));
            String linea;
            while ((linea = br.readLine()) != null) {
                System.out.println(linea);
            }
            br.close();
        } catch (IOException e) {
            System.err.println("Error al abrir o leer el archivo: " + e.getMessage());
        }
    }
}
```

2- FileNotFoundExeption:

En este ejemplo, hemos agregado un bloque catch adicional para manejar la FileNotFoundException específicamente. Si el archivo "archivo.txt" no se puede encontrar, se lanzará esta excepción y se imprimirá un mensaje de error adecuado.

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.FileNotFoundException;

public class Main {
    Run | Debug
    public static void main(String[] args) throws IOException {
        try {
            BufferedReader br = new BufferedReader(new FileReader(fileName:"archivo.txt"));
            String linea;
            while ((linea = br.readLine()) != null) {
                System.out.println(linea);
            }
            br.close();
        } catch (FileNotFoundException e) {
            System.err.println("El archivo no se pudo encontrar: " + e.getMessage());


    }
    }
}
```

3- SQLException:

En este ejemplo, se intenta ejecutar una consulta SQL incorrecta (seleccionar de una tabla que no existe). Si ocurre un error SQL, se captura la excepción SQLException y se muestra un mensaje de error.

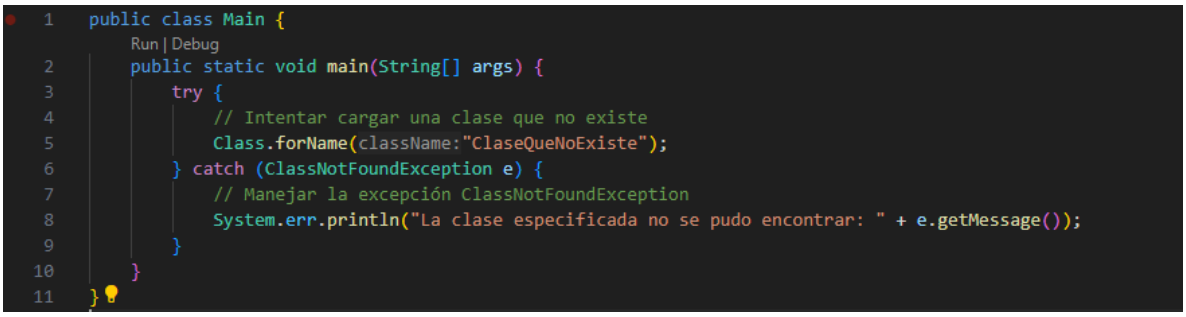
```java
import java.sql.*;

public class Main {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/nombre_base_de_datos";
        String usuario = "usuario";
        String contraseña = "contraseña";

        try (Connection conexion = DriverManager.getConnection(url, usuario, contraseña);
             Statement statement = conexion.createStatement()) {
            String consulta = "SELECT * FROM tabla_no_existente"; // consulta incorrecta a propósito
            statement.executeQuery(consulta);
        } catch (SQLException e) {
            System.err.println("Error de SQL: " + e.getMessage());
        }
    }
}
```

4- ClassNotFoundException:

En este ejemplo, se intenta cargar una clase llamada "ClaseQueNoExiste" utilizando el método Class.forName(). Si la clase no se puede encontrar, se lanzará una ClassNotFoundException, que es capturada y manejada en el bloque catch.



```java
public class Main {
    Run | Debug
    public static void main(String[] args) {
        try {
            // Intentar cargar una clase que no existe
            Class.forName(className:"ClaseQueNoExiste");
        } catch (ClassNotFoundException e) {
            // Manejar la excepción ClassNotFoundException
            System.err.println("La clase especificada no se pudo encontrar: " + e.getMessage());
        }
    }
}
```

5- ParseException:

En este ejemplo, intentamos analizar la cadena "2022-01-32" como una fecha en el formato "yyyy-MM-dd". Sin embargo, el día "32" es inválido para enero, por lo que se producirá una ParseException.

6- NullPointerException:

En este ejemplo, la variable texto se declara pero no se inicializa con ninguna cadena, lo que la hace nula. Al intentar llamar al método length() en la variable texto, se produce una NullPointerException porque no se puede invocar métodos en un objeto nulo.

7- ArrayIndexOutOfBoundsException:

En este ejemplo, intentamos acceder al elemento en la posición 3 del arreglo arreglo, que tiene solo tres elementos (índices 0, 1 y 2). Como resultado, se lanza una ArrayIndexOutOfBoundsException porque estamos intentando acceder a un índice fuera del rango válido.

```java
public class Main {
    Run | Debug
    public static void main(String[] args) {
        int[] arreglo = {1, 2, 3};

        try {
            // Intentamos acceder a un índice fuera del rango del arreglo
            int valor = arreglo[3];
            System.out.println("Valor en la posición 3: " + valor);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println(x:"Error: Índice fuera del rango del arreglo.");
        }
    }
}
```

8- ArithmeticException:

En este ejemplo, intentamos dividir el valor de dividendo por el valor de divisor, que es cero. Esto provocará una ArithmeticException porque la división por cero no está definida en matemáticas.

```java
public class Main {
    public static void main(String[] args) {
        int dividendo = 10;
        int divisor = 0;

        try {
            // Intentamos dividir el dividendo por cero
            int resultado = dividendo / divisor;
            System.out.println("Resultado de la división: " + resultado);
        } catch (ArithmeticException e) {
            System.err.println("Error: División por cero.");
        }
    }
}
```

9- ClassCastException:

En este ejemplo, intentamos convertir el objeto de tipo String a un tipo Integer mediante una conversión de tipo explícita. Sin embargo, como el objeto no es de tipo Integer, se producirá una ClassCastException.

```java
public class Main {
    Run | Debug
    public static void main(String[] args) {
        // Creamos un objeto de tipo String
        Object objeto = "Hola";

        try {
            // Intentamos convertir el objeto a un tipo Integer
            Integer numero = (Integer) objeto;
            System.out.println("Número: " + numero);
        } catch (ClassCastException e) {
            System.err.println(x:"Error: No se puede convertir el objeto a Integer.");
        }
    }
}
```

10- IllegalArgumentException:

En este ejemplo, estamos llamando al método imprimirNumero con un argumento de -5, que es un valor negativo. Dentro del método imprimirNumero, verificamos si el argumento es negativo y lanzamos una IllegalArgumentException si lo es.

```java
public class Main {
    Run | Debug
    public static void main(String[] args) {
        try {
            // Llamamos al método imprimirNumero con un argumento negativo
            imprimirNumero(-5);
        } catch (IllegalArgumentException e) {
            System.err.println(x:"Error: El argumento no puede ser negativo.");
        }
    }

    public static void imprimirNumero(int numero) {
        if (numero < 0) {
            throw new IllegalArgumentException(s:"El número no puede ser negativo.");
        }
        System.out.println("Número: " + numero);
    }
}
```

Java tiene muchas excepciones definidas en su biblioteca estándar y en bibliotecas adicionales. En la biblioteca estándar de Java, hay más de 70 excepciones distintas, cada una diseñada para manejar situaciones específicas que pueden surgir durante la ejecución de un programa.

Estas excepciones están organizadas en una jerarquía de clases, con Exception como la clase base. Las excepciones se dividen en dos categorías principales: excepciones verificadas (checked exceptions) y excepciones no verificadas (unchecked exceptions). Las excepciones verificadas son aquellas que el compilador obliga a manejar o declarar en la firma del método, mientras que las excepciones no verificadas no necesitan ser manejadas explícitamente en el código.

Si bien es difícil proporcionar un número exacto de excepciones en Java debido a que las bibliotecas pueden definir excepciones personalizadas y también debido a las versiones y extensiones de Java, la biblioteca estándar de Java ofrece una amplia gama de excepciones para manejar diferentes situaciones de error.

En resumen, hay muchas excepciones en Java, cada una destinada a manejar un tipo específico de error o situación excepcional que pueda surgir durante la ejecución de un programa.