



Números adyacentes

Matías Hoyuela, Eduardo Vásquez, Francisco Cáceres

8 de mayo de 2025

1. Introducción

El informe presenta los pasos realizados para resolver el problema de buscar el mayor producto de números adyacentes en un arreglo.

2. Descripción del caso

Dado el arreglo:

```
new double [ ] { 1 , -4, 2 , 2 , 5 , -1 }
```

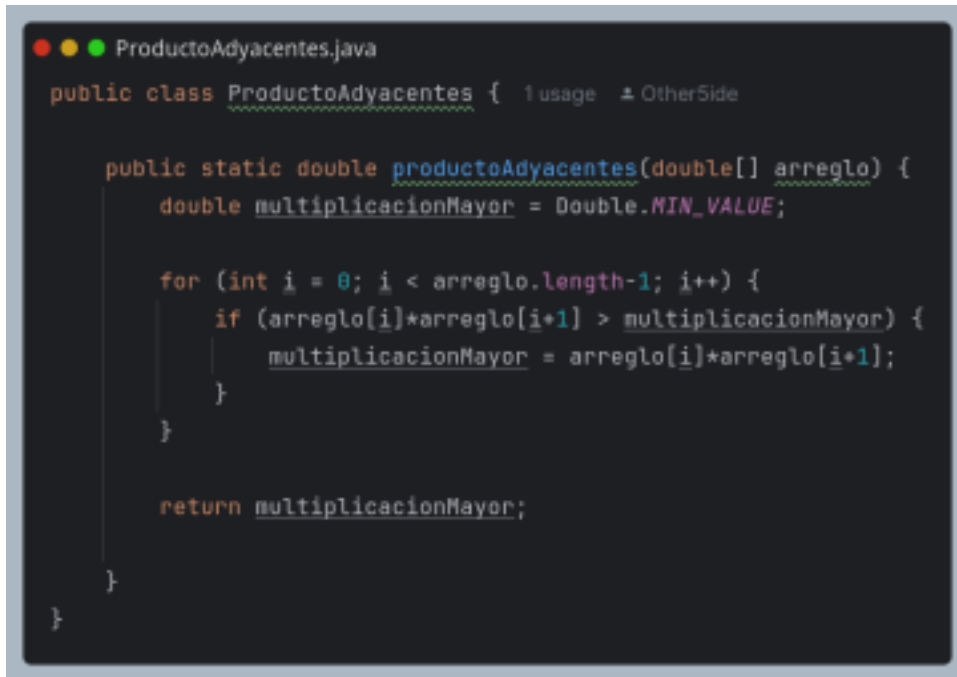
Se debe de implementar un método que retorne el mayor producto de números adyacentes. En este caso, el mayor producto es 10, con los números 2 y 5.

3. Análisis del caso

Se planifica el método:

- Parámetros de entrada: un arreglo de números double[].
- Valor de retorno: el mayor producto de números adyacentes double.
- Instrucciones: iterar sobre el arreglo de entrada e ir actualizando una variable cada vez que se encuentre un producto entre 2 números adyacentes mayor al valor de la variable, retornar el valor de la variable e imprimir en la consola el resultado.

4. Implementación de la solución



```

public class ProductoAdyacentes { 1 usage 1 OtherSide

    public static double productoAdyacentes(double[] arreglo) {
        double multiplicacionMayor = Double.MIN_VALUE;

        for (int i = 0; i < arreglo.length-1; i++) {
            if (arreglo[i]*arreglo[i+1] > multiplicacionMayor) {
                multiplicacionMayor = arreglo[i]*arreglo[i+1];
            }
        }

        return multiplicacionMayor;
    }
}

```

Para la solución, se inicializa una variable double `multiplicacionMayor`, y luego se itera sobre el arreglo. En cada iteración, se compara el producto de `arreglo[i]` y `arreglo[i+1]` con la variable `multiplicacionMayor`, y si el producto es mayor a esta, se actualiza la variable con ese producto. Para evitar un error de `ArrayIndexOutOfBoundsException`, se itera hasta `arreglo.length - 1`, por que si fuera hasta `arreglo.length`, durante la iteración `i = arreglo.length`, `arreglo[i+1]` intentaría acceder a un índice que no existe y lanzaría la excepción. Una vez se termina la iteración, se retorna el valor final de `multiplicacionMayor`.

5. Dise~no-implementación de las pruebas unitarias

Se pensaron las siguientes pruebas:

Arreglo de un solo elemento. El método debe lanzar una excepción `IllegalArgumentException`, puesto que se requiere un arreglo de al menos 2 elementos para calcular el producto de números adyacentes.

Un arreglo vacío. Nuevamente, el método debe lanzar `IllegalArgumentException`, puesto que se requiere un arreglo de al menos 2 elementos.

Arreglo nulo. El método debe lanzar una excepción `IllegalArgumentException`, puesto que el arreglo no puede ser nulo.

Arreglo negativo. El método debe retornar el mayor producto de números adyacentes, considerando los signos durante la multiplicación y la comparación.

Arreglo con un NaN. El método debe lanzar `IllegalArgumentException`, puesto que NaN no es válido para la multiplicación.

Arreglo con un infinito. El método debe lanzar `IllegalArgumentException`, puesto que infinito no es válido para la multiplicación.

Arreglo con un producto infinito. El método debe lanzar `ArithmeticException`, puesto que el producto debe ser finito.

Esto considerando que el input cumple con las siguientes condiciones:

$2 \leq \text{arreglo.length} \leq 1000$

$-1000 \leq \text{arreglo}[i] \leq 1000$

Tras diseñar las pruebas en papel, se implementaron en el programa:

```

●●● ProductoAdyacentesTest.java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class ProductoAdyacentesTest {
    @Test
    public void testCasoNormal() {
        double[] arreglo = {1, -4, 2, 2, 5, -1};
        assertEquals(10.0, ProductoAdyacentes.productoAdyacentes(arreglo));
    }

    @Test
    public void testConNegativos() {
        double[] arreglo = {-2.0, -3.0, 4.0};
        assertEquals(6.0, ProductoAdyacentes.productoAdyacentes(arreglo));
    }

    @Test
    public void testErrorArregloNull() {
        assertThrows(IllegalArgumentException.class, () -> {
            ProductoAdyacentes.productoAdyacentes(arreglo: null);
        });
    }

    @Test
    public void testErrorArregloMuyCorto() {
        double[] arreglo = {5.8};
        assertThrows(IllegalArgumentException.class, () -> {
            ProductoAdyacentes.productoAdyacentes(arreglo);
        });
    }

    @Test
    public void testErrorConNaN() {
        double[] arreglo = {1.0, Double.NaN, 2.0};
        assertThrows(IllegalArgumentException.class, () -> {
            ProductoAdyacentes.productoAdyacentes(arreglo);
        });
    }

    @Test
    public void testErrorConInfinito() {
        double[] arreglo = {1.0, Double.POSITIVE_INFINITY, 2.0};
        assertThrows(IllegalArgumentException.class, () -> {
            ProductoAdyacentes.productoAdyacentes(arreglo);
        });
    }

    @Test
    public void testProductoInfinito() {
        double[] arreglo = {Double.MAX_VALUE, Double.MAX_VALUE, 1.0};
        assertThrows(ArithmeticException.class, () -> {
            ProductoAdyacentes.productoAdyacentes(arreglo);
        });
    }
}

```

Resumen de las pruebas:

testCasoNormal: prueba con el arreglo de ejemplo del ejercicio.

- Entrada: double arreglo[] = {1, -4, 2, 2, 5, -1}

- Salida: 10

testConNegativos: prueba con un arreglo que contiene números negativos.

- Entrada: {-2.0, -3.0, 4.0}

- Salida: 6.0

testErrorArregloNulo: prueba con un null.

- Entrada: null

- Salida: IllegalArgumentException

testErrorArregloMuyCorto: prueba con un arreglo de solo un elemento.

- Entrada: double arreglo[] = {5.0}

- Salida: IllegalArgumentException

testErrorConNan: prueba con un arreglo que contiene un NaN.

- Entrada: double arreglo[] = {1.0, Double.NaN, 2.0}

- Salida: IllegalArgumentException

testErrorConInfinito: prueba con un arreglo que contiene un infinito.

- Entrada: double[] arreglo = {1.0, Double.POSITIVE_INFINITY, 2.0}

- Salida: IllegalArgumentException

testProductoInfinito: prueba con un arreglo que el mayor producto de números adyacentes es infinito.

- Entrada: double[] arreglo = {Double.MAX_VALUE, Double.MAX_VALUE, 1.0}

Salida: ArithmeticException

6. Diseño-implementación y resultados de las excepciones implementadas

Se modificó el programa para que gestione las excepciones y pase las pruebas:

```

●●● ProductoAdyacentes.java
public class ProductoAdyacentes { 8 usages  ▲ Eduardo +1

    public static double productoAdyacentes(double[] arreglo) { 8 usages  ▲ Eduardo +1
        validarArreglo(arreglo);
        validarNumeros(arreglo);

        double multiplicacionMayor = Double.MIN_VALUE;
        if (!Double.isFinite(multiplicacionMayor)) {
            throw new ArithmeticException("Producto fuera de rango.");
        }

        for (int i = 0; i < arreglo.length-1; i++) {
            double producto = arreglo[i] + arreglo[i + 1];
            if (!Double.isFinite(producto)) {
                throw new ArithmeticException("Producto fuera de rango");
            }
            if (producto > multiplicacionMayor) {
                multiplicacionMayor = producto;
            }
        }

        return multiplicacionMayor;
    }

    private static void validarArreglo(double[] arreglo) { 1 usage  ▲ Eduardo
        if (arreglo == null) {
            throw new IllegalArgumentException("El arreglo no puede ser null.");
        }

        if (arreglo.length < 2) {
            throw new IllegalArgumentException("El arreglo debe contener como mínimo dos elementos.");
        }
    }

    private static void validarNumeros(double[] arreglo) { 1 usage  ▲ Eduardo
        for (double valor : arreglo) {
            if (!Double.isFinite(valor)) {
                throw new IllegalArgumentException("El arreglo contiene valores no numéricos");
            }
        }
    }
}

```

Primero, el método validarArreglo verifica que el arreglo no sea null y que contenga al menos 2 elementos. Si no cumple con ninguna de estas condiciones, lanza una IllegalArgumentException.

Luego, el método validarNumeros itera sobre el arreglo para verificar que no contenga valores no numéricos. Si encuentra alguno, lanza una IllegalArgumentException.

Por último, durante la búsqueda del producto mayor, si alguno de los productos es infinito, se lanza una ArithmeticException.

7. Discusión con los resultados y comentarios de la experiencia

Las pruebas unitarias permitieron detectar varios errores en los métodos cuando eran sometidos a casos extremos pero posibles en la ejecución normal del código, el resultado es un código que tiene una tolerancia mayor a fallos y puede manejar excepciones.

8. Conclusiones

Gracias a las pruebas unitarias, se pudo hacer un programa más robusto, que en un inicio, requería que el usuario ingresara un arreglo que cumpliera con ciertas reglas. Sin embargo, como aprendimos en clases, no se puede confiar en el usuario, por tanto, el programa debe de ser capaz de manejar todo tipo de entradas, y no solo las que cumplen con las reglas. Para esto, se implementaron las excepciones, que hacen que el programa, pueda identificar y manejar entradas no válidas, como arreglos de un solo elemento, que contengan elementos no válidos, etc. De esta forma, el programa funciona en cualquier caso, y no depende de que el usuario siga nuestras reglas.