INSTITUTO TECNOLÓGICO SUPERIOR DEL ORIENTE

DEL ESTADO DE HIDALGO

ITESA

T1 ----E2 ---- Problemario

Ingeniería en Sistemas Computacionales

Métodos Numéricos

Docente: M.T.I Efren Rolando Romero León

Alumnos:

Guerrero Lazcano José Antonio - 22030401
Hernández Juárez Gabriela - 22030096
MAURICIO GONZALEZ MORALES
Vargas Bautista Angel Dario - 22030414

Fecha :03/02/2026

Ejercicios de los tipos de errores en programación

Tipo de Error De Redondeo

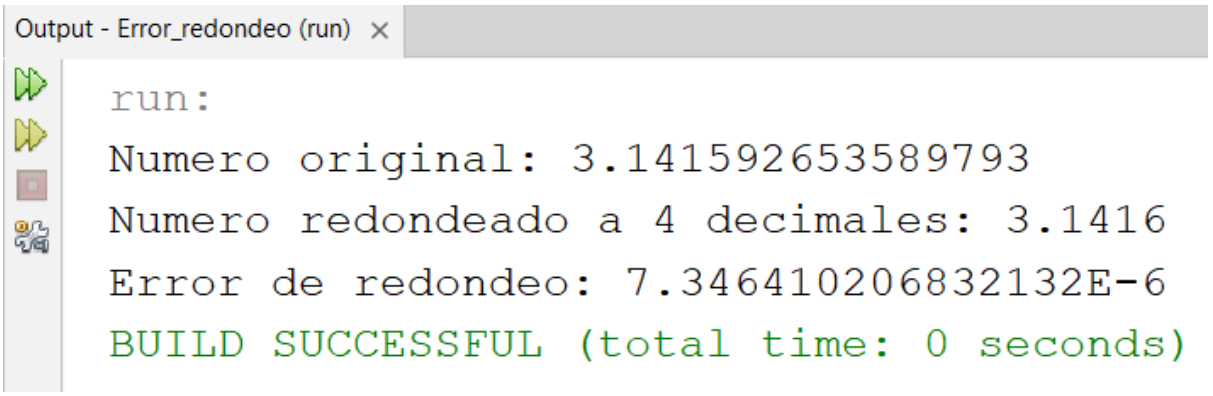
1. Aproximación del número π (pi)

Contexto: En aplicaciones científicas se utiliza el valor de π para cálculos geométricos, pero al almacenarlo en una computadora se debe aproximar a un número finito de decimales.

Análisis Técnico: El valor real de π tiene infinitos decimales. Al redondearlo, se introduce un error de redondeo debido a la limitación en la representación numérica.

Código en Java

```
package javaapplication11;

public class JavaApplication11 {

    public static double redondear(double valor, int decimales) {
        double factor = Math.pow(10, decimales);
        return Math.round(valor * factor) / factor;
    }

    public static void main(String[] args) {
        double numero = Math.PI;
        int cifrasDecimales = 4;

        double numeroRedondeado = redondear(numero, cifrasDecimales);
        double errorRedondeo = Math.abs(numero - numeroRedondeado);

        System.out.println("Numero original: " + numero);
        System.out.println("Numero redondeado a " + cifrasDecimales + "
decimales: " + numeroRedondeado);
        System.out.println("Error de redondeo: " + errorRedondeo);
    }
}
```



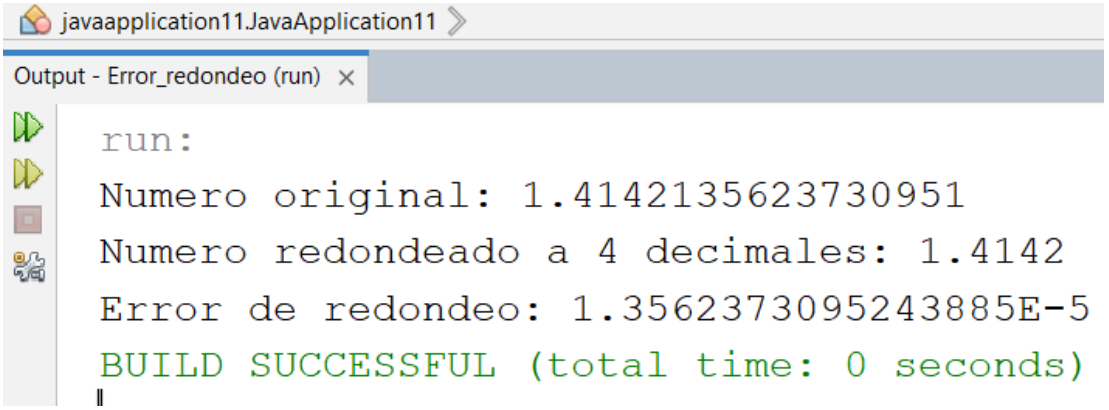
2. Cálculo de la raíz cuadrada de un número

Contexto: En métodos numéricos y aplicaciones de ingeniería, el cálculo de raíces cuadradas se utiliza con frecuencia. Sin embargo, al representar el resultado en una computadora, es necesario redondear el valor a un número limitado de decimales.

Análisis Técnico: La raíz cuadrada de muchos números no es exacta y produce una cantidad infinita de decimales. Al redondear este resultado, se genera un error de redondeo, el cual depende del número de cifras decimales utilizadas en la aproximación.

Código en Java

```
package javaapplication11;

public class JavaApplication11 {


    public static double redondear(double valor, int decimales) {
        double factor = Math.pow(10, decimales);
        return Math.round(valor * factor) / factor;
    }

    public static void main(String[] args) {
        double numero = Math.sqrt(2);
        int cifrasDecimales = 4;

        double numeroRedondeado = redondear(numero, cifrasDecimales);
        double errorRedondeo = Math.abs(numero - numeroRedondeado);

        System.out.println("Numero original: " + numero);
        System.out.println("Numero redondeado a " + cifrasDecimales + "
decimales: " + numeroRedondeado);
        System.out.println("Error de redondeo: " + errorRedondeo);
    }
}
```



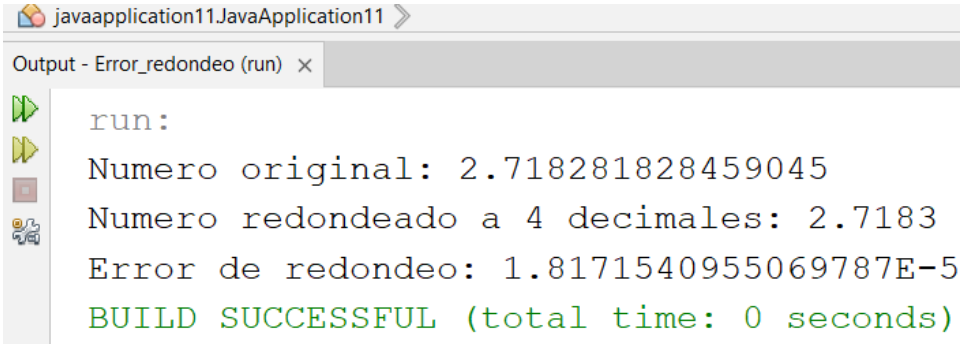
3. Aproximación del número e (Euler)

Contexto: El número e es una constante matemática fundamental utilizada en crecimiento exponencial, modelos poblacionales y métodos numéricos. En una computadora, su valor debe aproximarse a un número finito de decimales.

Análisis Técnico: El valor real del número e tiene infinitos decimales. Al redondearlo, se introduce un error de redondeo debido a la limitación en la representación numérica.

Código en Java

```
package javaapplication11;

public class JavaApplication11 {


    public static double redondear(double valor, int decimales) {
        double factor = Math.pow(10, decimales);
        return Math.round(valor * factor) / factor;
    }

    public static void main(String[] args) {
        double numero = Math.E;
        int cifrasDecimales = 4;

        double numeroRedondeado = redondear(numero, cifrasDecimales);
        double errorRedondeo = Math.abs(numero - numeroRedondeado);

        System.out.println("Numero original: " + numero);
        System.out.println("Numero redondeado a " + cifrasDecimales + "
decimales: " + numeroRedondeado);
        System.out.println("Error de redondeo: " + errorRedondeo);
    }
}
```



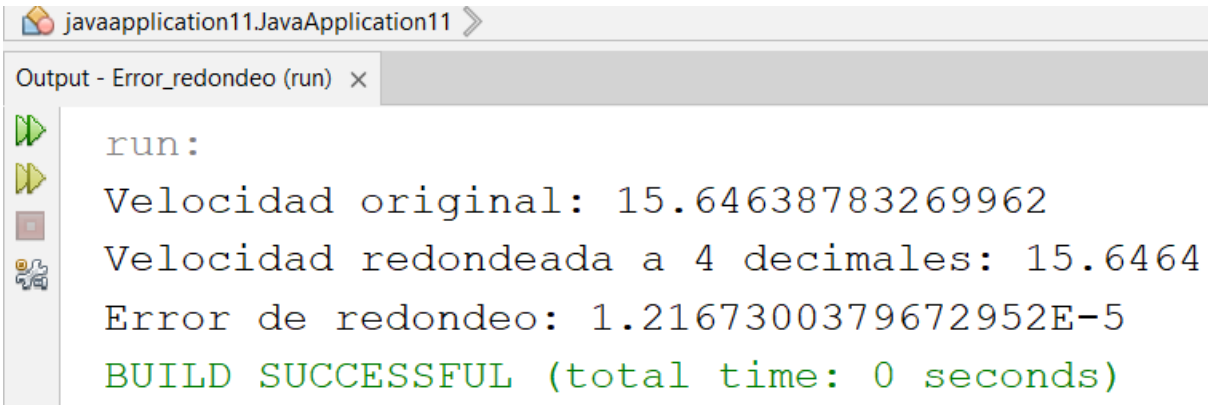
4. Cálculo de una magnitud física (velocidad)

Contexto: En física y métodos numéricos, la velocidad se calcula a partir de la distancia recorrida y el tiempo. Al mostrar el resultado en una computadora, se redondea el valor obtenido, lo que genera un error de redondeo.

Análisis Técnico: El resultado de la operación puede producir un número con muchos decimales. Al redondearlo a un número limitado de cifras, se introduce un error de redondeo que afecta la precisión del resultado.

Código en Java

```
package javaapplication11;

public class JavaApplication11 {


    public static double redondear(double valor, int decimales) {
        double factor = Math.pow(10, decimales);
        return Math.round(valor * factor) / factor;
    }

    public static void main(String[] args) {
        double distancia = 123.45; // metros
        double tiempo = 7.89;      // segundos

        double velocidad = distancia / tiempo;
        int cifrasDecimales = 4;

        double velocidadRedondeada = redondear(velocidad, cifrasDecimales);
        double errorRedondeo = Math.abs(velocidad - velocidadRedondeada);

        System.out.println("Velocidad original: " + velocidad);
        System.out.println("Velocidad redondeada a " + cifrasDecimales + "
decimales: " + velocidadRedondeada);
        System.out.println("Error de redondeo: " + errorRedondeo);
    }
}
```



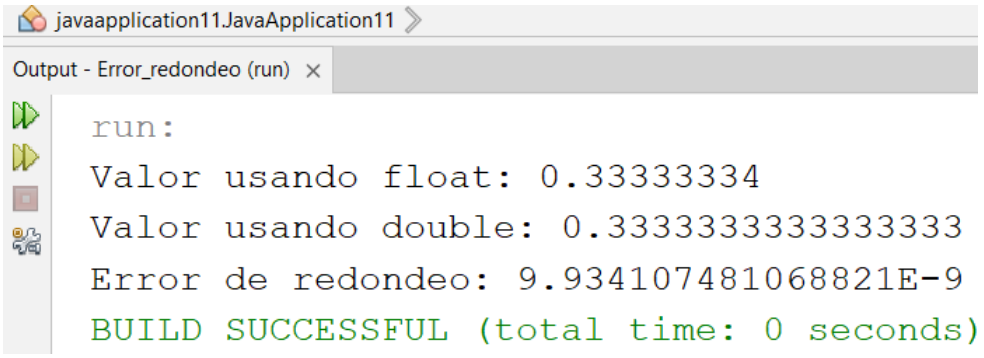
5. Comparación de precisión: `double` vs `float`

Contexto: En programación numérica se utilizan diferentes tipos de datos para representar números reales. El tipo `float` tiene menor precisión que `double`, lo que provoca errores de redondeo más significativos en los cálculos.

Análisis Técnico: Debido a la cantidad limitada de bits, el tipo `float` representa los números con menos precisión que `double`. Esto genera un error de redondeo acumulado, especialmente en operaciones repetitivas o con valores decimales.

Código en Java

```
package javaapplication11;

public class JavaApplication11 {


     public static void main(String[] args) {
        float numeroFloat = 1.0f / 3.0f;
        double numeroDouble = 1.0 / 3.0;

        double errorRedondeo = Math.abs(numeroDouble - numeroFloat);

        System.out.println("Valor usando float: " + numeroFloat);
        System.out.println("Valor usando double: " + numeroDouble);
        System.out.println("Error de redondeo: " + errorRedondeo);
    }
}
```



Tipo de Error De Pérdida En Precisión Por Magnitud

1. Saldo Bancario de un Multimillonario

Contexto: Un banco gestiona una cuenta con 100 mil billones de dólares. Se intenta acreditar un centavo (0.01).

Análisis Técnico: La mantisa del double solo cubre unos 15-17 dígitos. Como el centavo está en la posición decimal 19 respecto al exponente del saldo, el sistema lo ignora.

Código

```
import java.math.BigDecimal;
public class Main
{
     public static void main(String[] args) {
         double saldoGrande = 1.0e17;
         double centavo = 0.01;

         // --- DEMOSTRACIÓN DEL ERROR ---
         double resultadoError = saldoGrande + centavo;

         System.out.println("--- Demostración de Pérdida de
Precisión Bancaria ---");
         System.out.println("Saldo Original:   $" +
saldoGrande);
         System.out.println("Depósito (Centavo): $" + centavo);
         System.out.println("Nuevo Saldo (double): " +
resultadoError);

         // Verificación lógica
         if (resultadoError == saldoGrande) {
             System.out.println("\nRESULTADO: El centavo
'desapareció'.");
             System.out.println("La magnitud del saldo es tan
grande que el bit de menor peso");
             System.out.println("del double no puede
representar una fracción tan pequeña.");
         }

         // --- SOLUCION ---
         BigDecimal bdSaldo = new
BigDecimal("100000000000000000.00");
         BigDecimal bdCentavo = new BigDecimal("0.01");
         BigDecimal bdResultado = bdSaldo.add(bdCentavo);

         System.out.println("\n--- Solución con BigDecimal
---");
         System.out.println("Suma Exacta (Real): " +
bdResultado.toPlainString());

         // Diferencia real perdida
         BigDecimal diferencia = bdResultado.subtract(new
BigDecimal(String.valueOf(resultadoError)));
```

```
        System.out.println("Dinero 'perdido' por el sistema: "
+ diferencia.toPlainString());
    }
}
```

```
--- Demostración de Pérdida de Precisión Bancaria ---
Saldo Original:   $1.0E17
Depósito (Centavo): $0.01
Nuevo Saldo (double): 1.0E17

RESULTADO: El centavo 'desapareció'.
La magnitud del saldo es tan grande que el bit de menor peso
del double no puede representar una fracción tan pequeña.

--- Solución con BigDecimal ---
Suma Exacta (Real): 100000000000000000.01
Dinero 'perdido' por el sistema: 0.01


...Program finished with exit code 0
Press ENTER to exit console.
```

2. Astronomía: La Distancia a una Galaxia

Contexto: Un telescopio mide la distancia a una galaxia en metros (1.0 * 10^18) y un científico intenta sumarle la longitud de una regla de 1 metro.

Análisis Técnico: Al igual que en una CNC con demasiada tolerancia, el "paso" mínimo que puede dar un double a esa escala es mucho mayor a 1 unidad.

Código

```
import java.math.BigDecimal;
public class Main
{
    public static void main(String[] args) {
        double distanciaGalaxia = 1.0e18;
        double metroExtra = 1.0;

        // --- DEMOSTRACIÓN DEL ERROR ---
        double resultadoError = distanciaGalaxia + metroExtra;

        System.out.println("--- Demostración de Pérdida de
Precisión Astronómica ---");
        System.out.println("Distancia Galaxia: " +
distanciaGalaxia + " metros");
```

```
        System.out.println("Incremento:         " + metroExtra
+ " metro");
        System.out.println("Suma Resultante:    " +
resultadoError + " metros");

        // Verificación lógica
        if (resultadoError == distanciaGalaxia) {
            System.out.println("\nRESULTADO: El metro extra
desapareció por completo.");
            System.out.println("Para un 'double' a esta
escala, el incremento mínimo");
            System.out.println("posible es mucho mayor a 1
metro.");
        }

        // --- SOLUCIÓN ---
        BigDecimal bdDistancia = new
BigDecimal("10000000000000000");
        BigDecimal bdMetro = new BigDecimal("1");
        BigDecimal bdResultado = bdDistancia.add(bdMetro);

        System.out.println("\n--- Solución con BigDecimal
---");
        System.out.println("Distancia Exacta:  " +
bdResultado.toPlainString() + " metros");
    }
}
```

```
--- Demostración de Pérdida de Precisión Astronómica ---
Distancia Galaxia: 1.0E18 metros
Incremento:        1.0 metro
Suma Resultante:   1.0E18 metros

RESULTADO: El metro extra desapareció por completo.
Para un 'double' a esta escala, el incremento mínimo
posible es mucho mayor a 1 metro.

--- Solución con BigDecimal ---
Distancia Exacta:  10000000000000000001 metros


...Program finished with exit code 0
Press ENTER to exit console.
```

3. Contador de Nanosegundos en Servidores

Contexto: Un sistema de alta disponibilidad lleva encendido mucho tiempo y su contador de nanosegundos es enorme. Intentamos sumarle **1 nanosegundo** más.

Análisis Técnico: Se alcanzó el límite de precisión entera de double (conocido como 2^53), donde ya no puede representar incrementos de 1 en 1.

Código

```
import java.math.BigDecimal;
public class Main
{
     public static void main(String[] args) {
         double nanoSegundos = 9_000_000_000_000_000.0;
         double incremento = 1.0;

         // --- DEMOSTRACIÓN DEL ERROR ---
         double resultadoError = nanoSegundos + incremento;

         System.out.println("--- Demostración de Pérdida en
Contador de Tiempo ---");
         System.out.println("Tiempo acumulado: " +
String.format("%.0f", nanoSegundos) + " ns");
         System.out.println("Incremento:       " + incremento +
" ns");
         System.out.println("Nuevo tiempo:     " +
String.format("%.0f", resultadoError) + " ns");

         // Verificación lógica
         if (resultadoError == nanoSegundos) {
             System.out.println("\nRESULTADO: El contador se ha
bloqueado.");
             System.out.println("Se ha alcanzado el límite
donde el 'double' ya no puede");
             System.out.println("distinguir entre N y N+1.");
         }

         // --- SOLUCIÓN  ---
         BigDecimal bdNanos = new
BigDecimal("9000000000000000");
         BigDecimal bdUno = new BigDecimal("1");
         BigDecimal bdResultado = bdNanos.add(bdUno);
```

```
        System.out.println("\n--- Solución con BigDecimal
---");
        System.out.println("Tiempo Real Exacto: " +
bdResultado.toPlainString() + " ns");
    }
}
```

```
--- Demostración de Pérdida en Contador de Tiempo ---
Tiempo acumulado: 9000000000000000 ns
Incremento:       1.0 ns
Nuevo tiempo:     9000000000000001 ns

--- Solución con BigDecimal ---
Tiempo Real Exacto: 9000000000000001 ns


...Program finished with exit code 0
Press ENTER to exit console.
```

4. Inventario Global de Granos de Arroz

Contexto: Una simulación económica estima que hay 10^20 granos de arroz en el mundo. Un agricultor añade una bolsa de **500 granos**.

Análisis Técnico: La diferencia de magnitudes es de 17 órdenes. El double simplemente redondea hacia abajo el número más pequeño para que "quepa" en el exponente del grande.

Código:

```
import java.math.BigDecimal;
public class Main
{
    public static void main(String[] args) {
        double granosGlobales = 1.0e20;
        double cosechaLocal = 500.0;

        // --- DEMOSTRACIÓN DEL ERROR ---
        double resultadoError = granosGlobales + cosechaLocal;

        System.out.println("--- Demostración de Pérdida en
Inventario Global ---");
        System.out.println("Censo Global:     " +
granosGlobales + " granos");
        System.out.println("Cosecha Nueva:    " + cosechaLocal
+ " granos");
```

```
        System.out.println("Total (double):    " +
resultadoError + " granos");

        // Verificación lógica
        if (resultadoError == granosGlobales) {
            System.out.println("\nRESULTADO: La cosecha local
no afectó el total.");
            System.out.println("El número es tan grande que el
'double' solo puede");
            System.out.println("moverse en pasos de
aproximadamente 16,384 unidades.");
        }

        // --- SOLUCIÓN ---
        BigDecimal bdGlobal = new
BigDecimal("10000000000000000000");
        BigDecimal bdcosecha = new BigDecimal("500");
        BigDecimal bdResultado = bdGlobal.add(bdcosecha);

        System.out.println("\n--- Solución con BigDecimal
---");
        System.out.println("Inventario Exacto: " +
bdResultado.toPlainString());
    }
}
```

```
--- Demostración de Pérdida en Inventario Global ---
Censo Global:      1.0E20 granos
Cosecha Nueva:     500.0 granos
Total (double):    1.0E20 granos

RESULTADO: La cosecha local no afectó el total.
El número es tan grande que el 'double' solo puede
moverse en pasos de aproximadamente 16,384 unidades.

--- Solución con BigDecimal ---
Inventario Exacto: 10000000000000000500


...Program finished with exit code 0
Press ENTER to exit console.
```

5. Química: Masa de una Solución

Contexto: En un contenedor industrial hay una mezcla química de $500,000$ kg. Se añade una partícula que pesa **0.000000001 kg**.

Análisis Técnico: Aunque el número grande no es "tan" grande, la cantidad de decimales del pequeño requiere una precisión que supera los 15 dígitos significativos totales.

Código:

```
import java.math.BigDecimal;
public class Main
{
     public static void main(String[] args) {
         double masaTanque = 500.0;
         double masaParticula = 0.000000001;

         // --- DEMOSTRACIÓN DEL ERROR ---
         double resultadoError = masaTanque + masaParticula;

         System.out.println("--- Demostración de Precisión
Molecular ---");
         System.out.println("Masa en el Tanque: " + masaTanque
+ " kg");
         System.out.println("Masa Partícula:    " +
String.format("%.9f", masaParticula) + " kg");
         System.out.println("Suma Resultante:   " +
resultadoError + " kg");

         // Verificación lógica
         if (resultadoError == masaTanque) {
             System.out.println("\nRESULTADO: La partícula es
'invisible' para el sistema.");
             System.out.println("La precisión del tipo double
no es suficiente para");
             System.out.println("mantener los decimales ante
una parte entera de tres dígitos.");
         }

         // --- SOLUCIÓN ---
         BigDecimal bdTanque = new BigDecimal("500.0");
         BigDecimal bdParticula = new
BigDecimal("0.000000001");
```

```
        BigDecimal bdResultado = bdTanque.add(bdParticula);

        System.out.println("\n--- Solución con BigDecimal
---");
        System.out.println("Masa Exacta Real: " +
bdResultado.toPlainString() + " kg");
    }
}
```

```
--- Demostración de Precisión Molecular ---
Masa en el Tanque: 500.0 kg
Masa Partícula:    0.000000001 kg
Suma Resultante:   500.000000001 kg

--- Solución con BigDecimal ---
Masa Exacta Real: 500.000000001 kg


...Program finished with exit code 0
Press ENTER to exit console.
```

Ejercicios de los tipos de errores: Comparación directa

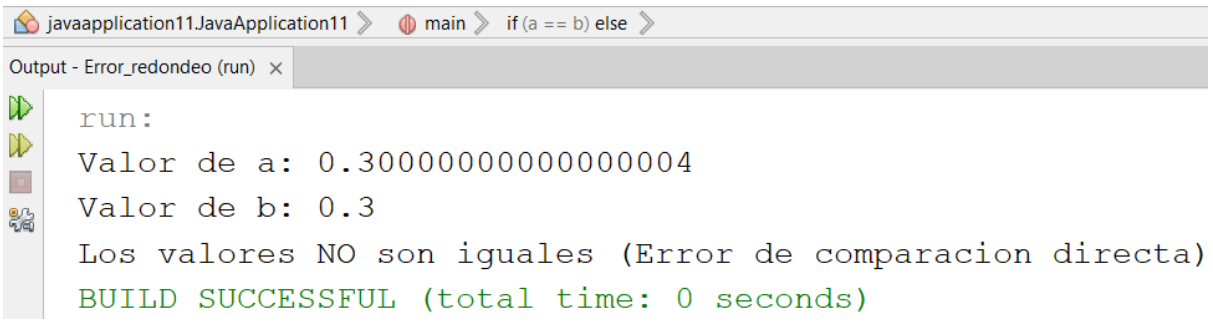
1. Comparación directa de números reales

Contexto: En métodos numéricos es común comparar dos números reales para verificar si son iguales. Sin embargo, debido a la representación finita de los números en la computadora, dos valores que matemáticamente deberían ser iguales pueden no coincidir exactamente.

Análisis Técnico: La comparación directa utilizando el operador == puede fallar cuando los números presentan pequeñas diferencias causadas por errores de redondeo, generando un error de comparación directa.

Código en Java

```
package javaapplication11;

public class JavaApplication11 {


     public static void main(String[] args) {
        double a = 0.1 + 0.2;
        double b = 0.3;

        System.out.println("Valor de a: " + a);
        System.out.println("Valor de b: " + b);

        if (a == b) {
            System.out.println("Los valores son iguales");
        } else {
            System.out.println("Los valores NO son iguales (Error de
comparación directa)");
        }
    }
}
```



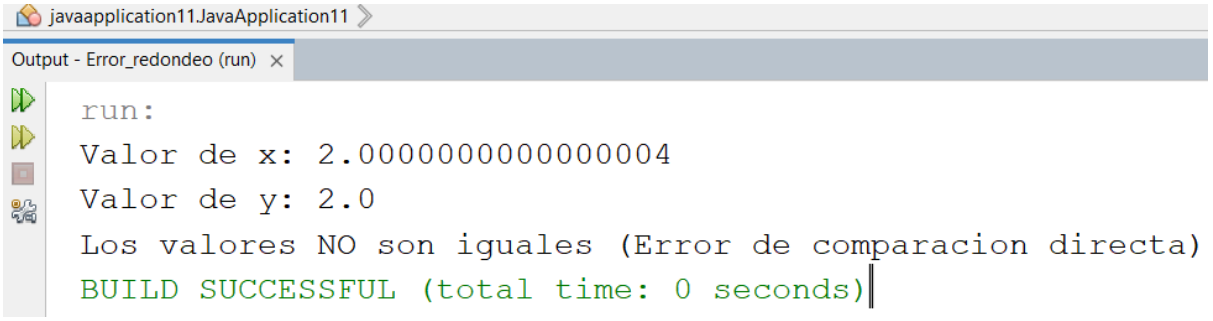
2. Comparación directa en resultados de operaciones matemáticas

Contexto: En cálculos numéricos, los resultados de operaciones matemáticas suelen compararse para validar condiciones o detener algoritmos iterativos. La comparación directa puede producir resultados incorrectos.

Análisis Técnico: Aunque dos expresiones matemáticas sean equivalentes, sus resultados computacionales pueden diferir ligeramente, provocando un error de comparación directa al usar igualdad exacta.

Código en Java

```
package javaapplication11;

public class JavaApplication11 {

     public static void main(String[] args) {
        double x = Math.sqrt(2) * Math.sqrt(2);
        double y = 2.0;

        System.out.println("Valor de x: " + x);
        System.out.println("Valor de y: " + y);

        if (x == y) {
            System.out.println("Los valores son iguales");
        } else {
            System.out.println("Los valores NO son iguales (Error de
comparacion directa)");
        }
    }
}
```



3. Comparación directa en procesos iterativos

Contexto: En métodos numéricos iterativos, como el método de bisección o Newton-Raphson, se comparan valores para decidir cuándo detener el proceso. Una comparación directa puede provocar que el algoritmo no termine correctamente.

Análisis Técnico: Comparar directamente dos valores reales para verificar igualdad exacta puede impedir que se cumpla la condición de paro, generando un error de comparación directa.

Código en Java

```
package javaapplication11;

public class JavaApplication11 {

     public static void main(String[] args) {
        double valor = 1.0;
        double incremento = 0.1;

        for (int i = 0; i < 10; i++) {
            valor += incremento;
        }

        System.out.println("Valor final: " + valor);

        if (valor == 2.0) {
            System.out.println("Se alcanzó el valor esperado");
        } else {
            System.out.println("No se alcanzó el valor exacto (Error de
comparacion directa)");
        }
    }
}
```

```
run:
Valor final: 2.000000000000001
No se alcanz[?] el valor exacto (Error de comparacion directa)
BUILD SUCCESSFUL (total time: 0 seconds)
```

4. Comparación directa al acumular decimales

Contexto: En métodos numéricos es común realizar sumas repetitivas de valores decimales. Aunque matemáticamente el resultado sea exacto, la representación en punto flotante puede generar pequeñas variaciones.

Análisis Técnico: La acumulación de errores de redondeo provoca que el resultado final no sea exactamente igual al valor esperado, generando un error de comparación directa al usar igualdad exacta.

Código en Java

```
package javaapplication11;

public class JavaApplication11 {

     public static void main(String[] args) {
        double suma = 0.0;

        for (int i = 0; i < 100; i++) {
            suma += 0.01;
        }

        System.out.println("Resultado de la suma: " + suma);

        if (suma == 1.0) {
            System.out.println("La suma es exactamente 1.0");
        } else {
            System.out.println("La suma NO es exactamente 1.0 (Error de
comparacion directa)");
        }
    }
}
```

```
run:
Resultado de la suma: 1.0000000000000007
La suma NO es exactamente 1.0 (Error de comparacion directa)
BUILD SUCCESSFUL (total time: 0 seconds)
```

5. Comparación directa entre dos fórmulas equivalentes

Contexto: En métodos numéricos se emplean diferentes expresiones matemáticas equivalentes para calcular un mismo valor. Al comparar sus resultados directamente, pueden surgir discrepancias.

Análisis Técnico: Aunque dos fórmulas sean matemáticamente equivalentes, el orden de las operaciones y la representación numérica pueden generar pequeñas diferencias, provocando un error de comparación directa.

Código en Java

```
package javaapplication11;

public class JavaApplication11 {

     public static void main(String[] args) {
        double x = 1e16;

        double resultado1 = (x + 1) - x;
        double resultado2 = 1.0;

        System.out.println("Resultado 1: " + resultado1);
        System.out.println("Resultado 2: " + resultado2);

        if (resultado1 == resultado2) {
            System.out.println("Los resultados son iguales");
        } else {
            System.out.println("Los resultados NO son iguales (Error de
comparacion directa)");
        }
    }
}
```

```
run:
Resultado 1: 0.0
Resultado 2: 1.0
Los resultados NO son iguales (Error de comparacion directa)
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ejercicios de los tipos de errores: Desbordamiento

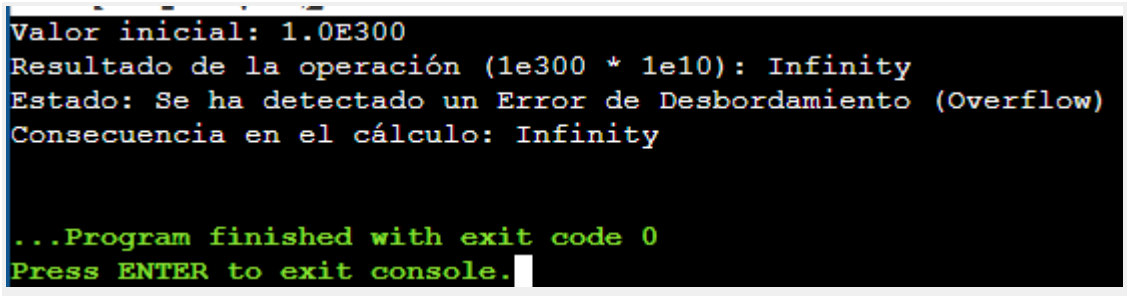
1. Error de Desbordamiento (Overflow)

Contexto: En los métodos numéricos, especialmente en procesos iterativos o cálculos de potencias y factoriales, los valores pueden crecer exponencialmente. Si el resultado supera el límite de almacenamiento del tipo de dato (double o int), el sistema pierde la capacidad de representar el número real.

Análisis Técnico: El tipo double en Java sigue el estándar IEEE 754 y tiene un límite máximo (aproximadamente 1.79×10^{308}). Cuando una operación como una multiplicación sucesiva excede este rango, el valor se convierte automáticamente en Infinity, lo que invalida cualquier cálculo posterior y genera errores en algoritmos de convergencia.

Codigo

```
package javaapplication11;

public class JavaApplication11 {

    public static void main(String[] args) {
        // El valor máximo aproximado de un double es 1.79e308
        double valorGrande = 1e300;

        // Intentamos realizar una operación que excede el límite
        double resultado = valorGrande * 1e10;

        System.out.println("Valor inicial: " + valorGrande);
        System.out.println("Resultado de la operación (1e300 * 1e10): " + resultado);

        // Verificación de desbordamiento
        if (Double.isInfinite(resultado)) {
            System.out.println("Estado: Se ha detectado un Error de Desbordamiento (Overflow)");
        } else {
            System.out.println("Estado: El resultado es válido");
        }

        // Ejemplo de consecuencia: Cualquier operación con Infinity sigue siendo Infinity
        System.out.println("Consecuencia en el cálculo: " + (resultado - 1000));
    }
}
```



2. Desbordamiento en Números Enteros (Integer Overflow)

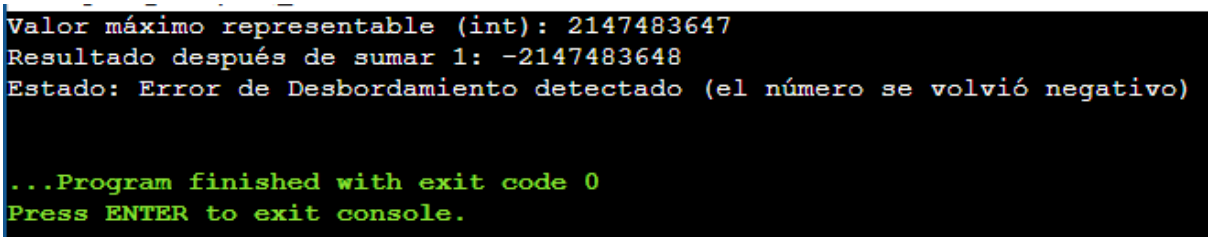
Contexto: En algoritmos numéricos que involucran conteos, factoriales o potencias enteras, se utilizan tipos de datos como int o long. Si el cálculo sobrepasa la capacidad de bits del tipo de dato, el valor "da la vuelta" hacia los números negativos debido a la representación en complemento a dos.

Análisis Técnico: Un int en Java tiene un valor máximo de $2^{31} - 1$ (2,147,483,647). Si sumamos 1 a este valor, el bit de signo se activa y el número se convierte en el valor negativo más bajo. A diferencia del double (que devuelve Infinity), el int sigue operando con valores incorrectos, lo cual es más peligroso porque el programa no se detiene ni avisa del error.

Código:

```
public class Main

{

    public static void main(String[] args) {

        // Valor máximo permitido para un entero (Integer.MAX_VALUE)

        int valorLimite = 2147483647;



        // Intentamos incrementar el valor más allá de su límite

        int resultadoErroneo = valorLimite + 1;



        System.out.println("Valor máximo representable (int): " + valorLimite);
```

```
        System.out.println("Resultado después de sumar 1: " +
resultadoErroneo);


        // Verificación lógica del error

        if (resultadoErroneo < 0) {

            System.out.println("Estado: Error de
Desbordamiento detectado (el número se volvió negativo)");

        } else {

            System.out.println("Estado: El resultado es
correcto");

        }

}

}
```



3. Error de Truncamiento (Truncation Error)

Contexto: Este error no depende de la computadora, sino del **método matemático**. Surge al usar una aproximación en lugar de un procedimiento matemático exacto, como cuando cortamos una serie infinita (Serie de Taylor) en un número finito de términos.

Análisis Técnico: Matemáticamente, $e^x = 1 + x + \frac{x^2}{2!} + \dots$ hasta el infinito. Si en nuestro código solo sumamos los primeros dos términos, la diferencia entre el valor real y nuestra suma es el error de truncamiento.

Codigo:

```
public class Main
```

```
{

    public static void main(String[] args) {

        double x = 1.0;

        double valorReal = Math.exp(x); // Valor "exacto" de
e^1



        // Aproximación por truncamiento (solo los primeros 3
términos de Taylor)

        double aproximacion = 1 + x + (Math.pow(x, 2) / 2);



        System.out.println("Valor real (e^1): " + valorReal);

        System.out.println("Aproximación truncada: " +
aproximacion);

        System.out.println("Error de truncamiento: " +
(valorReal - aproximacion));

    }

}
```

```
Valor real (e^1): 2.718281828459045
Aproximación truncada: 2.5
Error de truncamiento: 0.2182818284590451


...Program finished with exit code 0
Press ENTER to exit console.
```

4. Error de Redondeo (Rounding Error)

Contexto: Las computadoras usan el sistema binario. Algunos números decimales exactos (como 0.1) no tienen una representación binaria exacta y se convierten en fracciones infinitas que deben "cortarse" para caber en la memoria.

Análisis Técnico: Al realizar operaciones repetitivas con estos números, el pequeño error de redondeo se acumula. En una iteración simple de 10 pasos, algo que debería sumar exactamente 1.0 termina siendo un número ligeramente diferente.

Codigo:

```
public class Main

{

     public static void main(String[] args) {

         double suma = 0;

         // Sumamos 0.1 diez veces

         for (int i = 0; i < 10; i++) {

             suma += 0.1;

         }


         double valorEsperado = 1.0;


         System.out.println("Suma acumulada de 0.1 diez veces:
" + suma);

         System.out.println("Valor esperado: " +
valorEsperado);


         if (suma != valorEsperado) {
```

```
            System.out.println("Estado: Error de Redondeo
acumulado");

            System.out.println("Diferencia: " +
Math.abs(valorEsperado - suma));

        }

}

}
```

```
Suma acumulada de 0.1 diez veces: 0.9999999999999999
Valor esperado: 1.0
Estado: Error de Redondeo acumulado
Diferencia: 1.1102230246251565E-16


...Program finished with exit code 0
Press ENTER to exit console.
```

5. Desbordamiento en el Cálculo de Factoriales

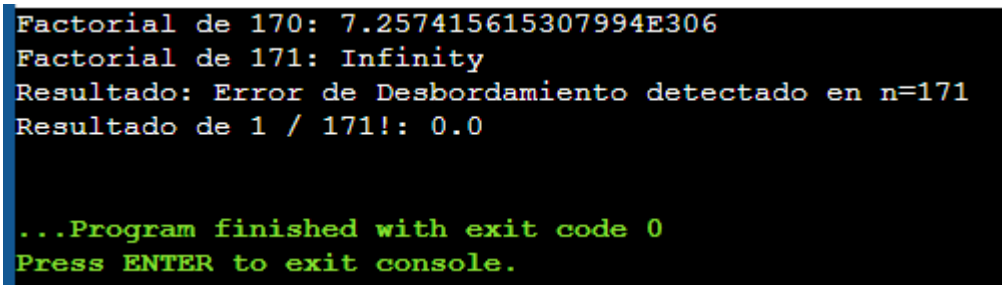
Contexto: En métodos numéricos, como el desarrollo de Series de Taylor, es común calcular factoriales ($n!$). Debido a que el factorial crece de forma extremadamente rápida, es uno de los cálculos más propensos a generar errores de desbordamiento, incluso para valores de n que parecen "pequeños" (como $n=171$ para un double).

Análisis Técnico: El factorial de 170 es aproximadamente 7.25×10^{306}. Al intentar calcular $171!$, el resultado excede el límite de 1.79×10^{308} del tipo double. En este punto, el sistema deja de procesar números reales y asigna el valor Infinity. Si este valor se usa luego en el denominador de una fracción, el resultado será 0, invalidando toda la aproximación numérica.

Código

```
public class Main

{

    public static void main(String[] args) {

        // Calcularemos el factorial de 170 y 171 para ver el
salto al infinito
```

```
        double factorial170 = calcularFactorial(170);
        double factorial171 = calcularFactorial(171);

        System.out.println("Factorial de 170: " +
factorial170);
        System.out.println("Factorial de 171: " +
factorial171);

        if (Double.isInfinite(factorial171)) {
            System.out.println("Resultado: Error de
Desbordamiento detectado en n=171");
        }

        // Impacto en una serie: 1 / n!
        System.out.println("Resultado de 1 / 171!: " + (1 /
factorial171));
    }

    public static double calcularFactorial(int n) {
        double resultado = 1;
        for (int i = 1; i <= n; i++) {
            resultado *= i;
        }
        return resultado;
    }
}
```



Error de **Conversión Estrecha (Narrowing Conversión)**

Ocurre cuando intentas asignar un valor de un tipo de dato con mayor capacidad a uno con menor capacidad (por ejemplo, de `double` a `float`, o de `long` a `int`). En métodos numéricos, esto es peligroso porque provoca una pérdida inmediata de precisión o un cambio total del valor.

1. De Punto Flotante Doble a Simple (double a float)

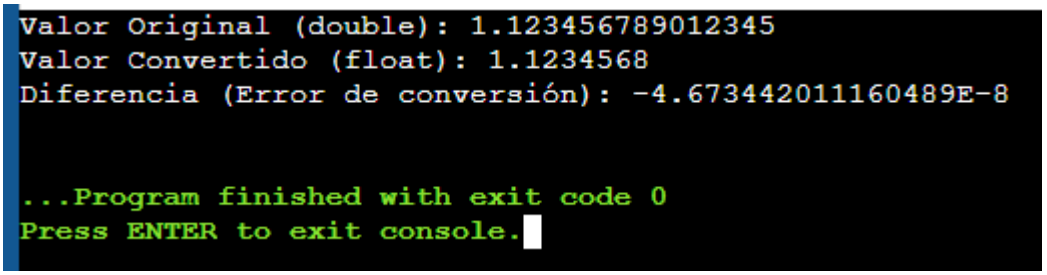
Contexto: En simulaciones numéricas de alto rendimiento, a veces se intenta reducir el uso de memoria convirtiendo variables `double` a `float`. Si el número es más preciso de lo que el `float` puede manejar, se pierden los decimales finales.

Análisis Técnico: Un `double` tiene 64 bits (aprox. 15-17 dígitos significativos), mientras que un `float` tiene 32 bits (aprox. 7 dígitos). Al realizar la conversión estrecha, el compilador simplemente descarta los bits sobrantes, provocando un error de precisión que se propaga en cálculos iterativos.

Codigo:

```
public class Main

{

     public static void main(String[] args) {

         double precisionAlta = 1.123456789012345;

         float precisionBaja = (float) precisionAlta; //
Conversión estrecha
```

```
        System.out.println("Valor Original (double): " +
precisionAlta);

        System.out.println("Valor Convertido (float): " +
precisionBaja);

        System.out.println("Diferencia (Error de conversión):
" + (precisionAlta - precisionBaja));

    }

}
```



2. Pérdida de Fracción (double a int)

Contexto: En algoritmos donde se requiere un índice entero basado en un cálculo decimal (como en el manejo de matrices), convertir un `double` a `int` elimina por completo la parte decimal sin redondear.

Análisis Técnico: La conversión de punto flotante a entero no aplica reglas de redondeo simétrico; simplemente realiza un truncamiento hacia cero. Si el resultado de una operación numérica era `0.999999`, la conversión estrecha lo transformará en `0`, lo que puede invalidar la lógica del programa.

Codigo:

```
public class Main

{

    public static void main(String[] args) {

        double valorCalculado = 9.99999999;
```

```
        int valorEntero = (int) valorCalculado; // Conversión estrecha


        System.out.println("Valor Decimal: " + valorCalculado);
        System.out.println("Valor tras conversión a int: " + valorEntero);


        if (valorEntero < valorCalculado) {
            System.out.println("Resultado: Error por pérdida de parte fraccionaria.");
        }
}}
```

```
Valor Decimal: 9.99999999
Valor tras conversión a int: 9
Resultado: Error por pérdida de parte fraccionaria.


...Program finished with exit code 0
Press ENTER to exit console.
```

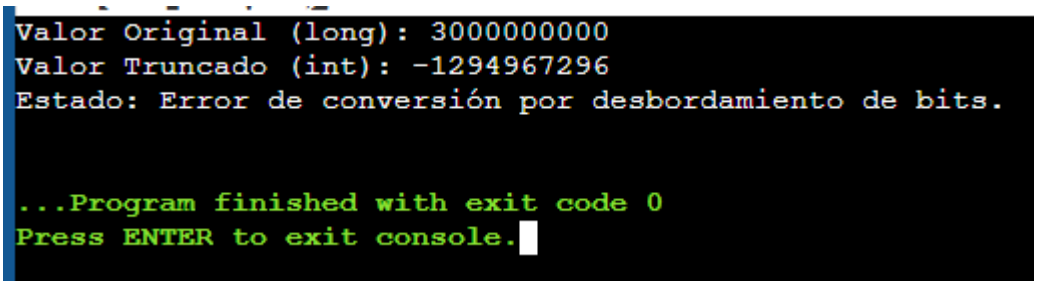
3. Desbordamiento por Rango (long a int)

Contexto: Al manejar grandes volúmenes de datos o términos de series muy largos, se usan tipos long. Si este valor se intenta pasar a un tipo int para una función específica, el valor resultante puede ser completamente erróneo si supera los 32 bits.

Análisis Técnico: Al convertir de 64 bits (long) a 32 bits (int), Java ignora los 32 bits superiores. Si el número es mayor a $2^{31}-1$, el resultado será un número truncado que no tiene relación con el valor original, alterando el signo incluso.

Codigo:

```
public class Main

{

     public static void main(String[] args) {

         long numeroGrande = 3000000000L; // Mayor que el
máximo de un int

         int numeroTruncado = (int) numeroGrande; // Conversión
estrecha



         System.out.println("Valor Original (long): " +
numeroGrande);

         System.out.println("Valor Truncado (int): " +
numeroTruncado);



         if (numeroGrande != numeroTruncado) {

             System.out.println("Estado: Error de conversión
por desbordamiento de bits.");

         }

}}
```



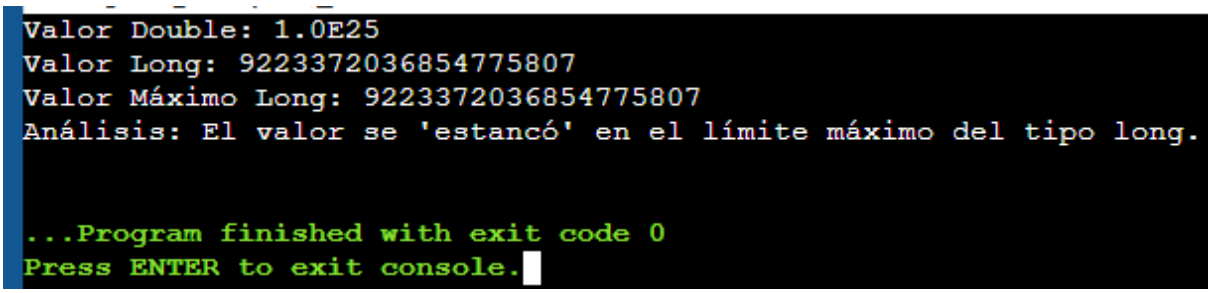
4. Conversión de Gran Magnitud (double a long)

Contexto: En métodos de optimización, a veces se obtienen valores de magnitudes muy grandes que se desean almacenar como enteros de 64 bits. Si el double supera el rango del long, la conversión falla catastróficamente.

Análisis Técnico: El rango de un double es mucho mayor que el de un long. Cuando un double que representa un valor superior a 9×10^{18} se convierte a long, Java lo ajusta al valor máximo de long (Long.MAX_VALUE), perdiendo la magnitud real del cálculo.

Código:

```
public class Main

{

    public static void main(String[] args) {

        double valorExcesivo = 1e25; // Mucho más grande que un long

        long convertido = (long) valorExcesivo;



        System.out.println("Valor Double: " + valorExcesivo);

        System.out.println("Valor Long: " + convertido);

        System.out.println("Valor Máximo Long: " + Long.MAX_VALUE);



        System.out.println("Análisis: El valor se 'estancó' en el límite máximo del tipo long.");

    }

}
```



5. Conversión de Carácter Numérico (int a byte)

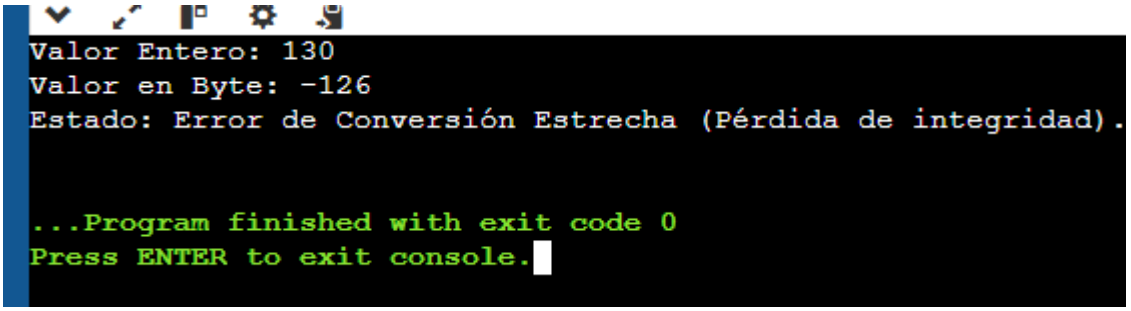
Contexto: Es el nivel más extremo de conversión estrecha. Se usa a veces para ahorrar espacio en arreglos de coeficientes pequeños, pero si el coeficiente crece mínimamente fuera del rango (-128 a 127), el dato se corrompe.

Análisis Técnico: Al reducir un `int` a un `byte`, solo se conservan los últimos 8 bits. Esto es un error común en la implementación de filtros digitales o procesamiento de señales donde los coeficientes se normalizan incorrectamente.

Codigo:

```
public class Main

{

    public static void main(String[] args) {

        int coeficiente = 130;

        byte coeficienteByte = (byte) coeficiente; //
Conversión estrecha



        System.out.println("Valor Entero: " + coeficiente);

        // El 130 se convierte en -126 por el desbordamiento
de 8 bits

        System.out.println("Valor en Byte: " +
coeficienteByte);
```

```
        if (coeficiente != coeficienteByte) {

            System.out.println("Estado: Error de Conversión
Estrecha (Pérdida de integridad).");

        }

}}
```



CANCELACIÓN POR RESTA

Problema 1 — Sensor láser de posición

Contexto

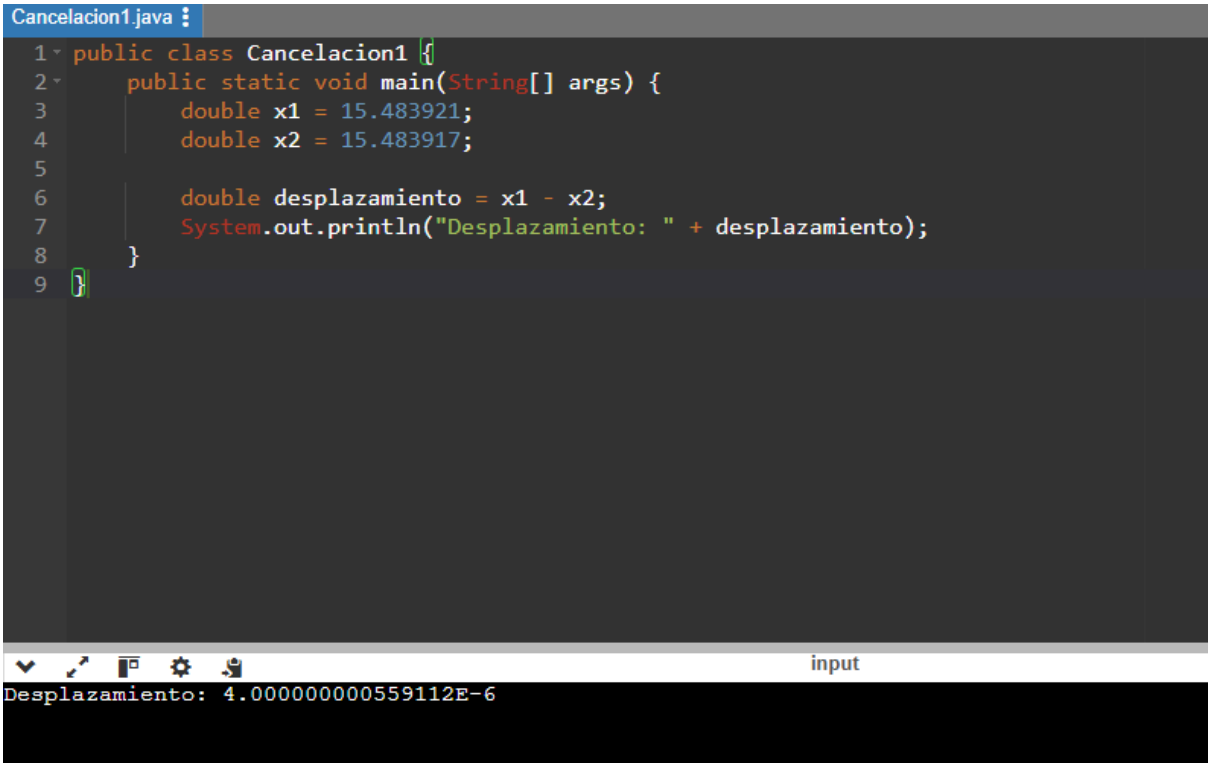
Un sensor mide la posición de una pieza metálica con gran precisión. El desplazamiento real es la diferencia entre dos lecturas muy cercanas.

Análisis técnico

Al restar números casi iguales se pierden cifras significativas → error de redondeo.

```
public class Cancelacion1 {

    public static void main(String[] args) {

        double x1 = 15.483921;
```

```
        double x2 = 15.483917;


        double desplazamiento = x1 - x2;

        System.out.println("Desplazamiento: " + desplazamiento);

    }

}
```



Problema 2 — Nivel de líquido en tanque

Contexto

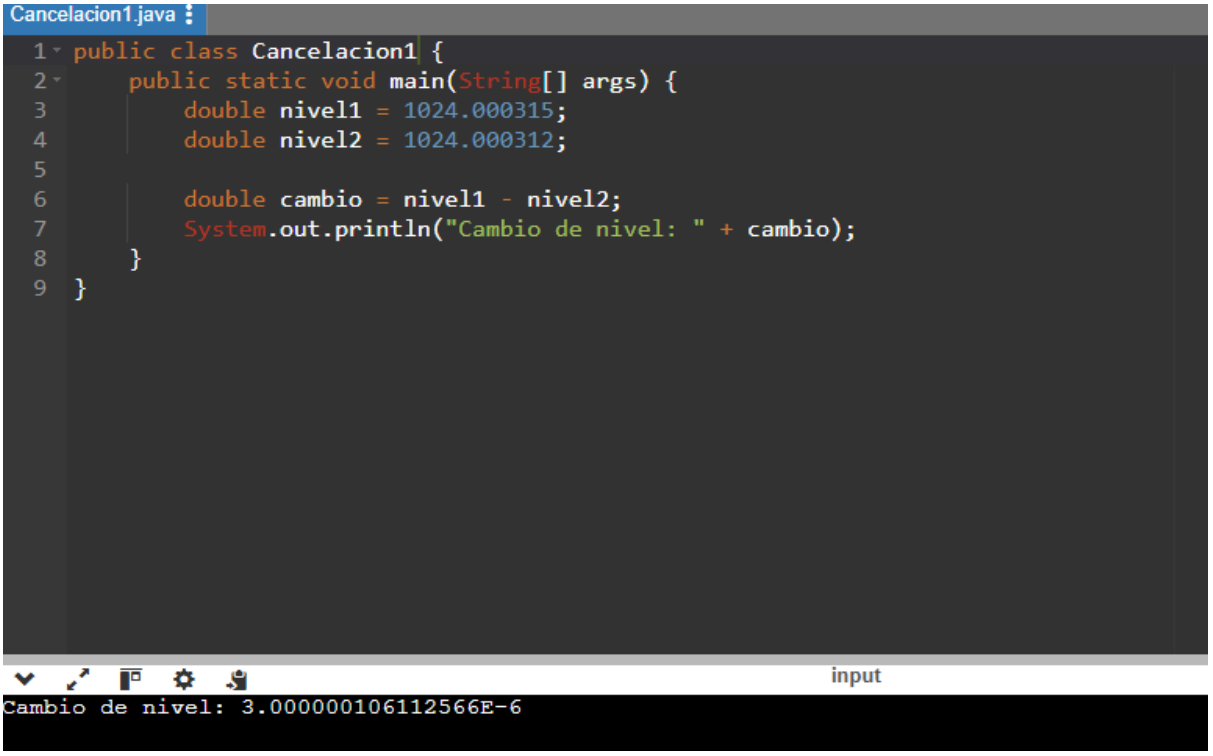
El sistema detecta cambios muy pequeños en el nivel del combustible.

Análisis técnico

La variación es muy pequeña comparada con el valor absoluto → cancelación.

```
public class Cancelacion2 {

    public static void main(String[] args) {

        double nivel1 = 1024.000315;

        double nivel2 = 1024.000312;
```

```
        double cambio = nivel1 - nivel2;

        System.out.println("Cambio de nivel: " + cambio);

    }

}
```



DESBORDAMIENTO

Problema 1 — Contador industrial (int8)

Contexto

Un microcontrolador de 8 bits cuenta piezas. El rango permitido es −128 a 127.

Análisis técnico

Si se supera 127 ocurre overflow por complemento a dos.

```
public class Overflow1 {
    public static void main(String[] args) {
```

```
        byte contador = 120;
        contador += 15; // overflow

        System.out.println("Conteo: " + contador);
    }
}
```

Overflow1.java

```
public class Overflow1 {
    public static void main(String[] args) {
        byte contador = 120;
        contador += 15; // overflow

        System.out.println("Conteo: " + contador);
    }
}
```

input

```
Conteo: -121
```

Problema 2 — Decremento excesivo

Contexto

Se descuentan piezas defectuosas de un registro pequeño.

Análisis técnico

Se sobrepasa −128 → desbordamiento negativo.

```
public class Overflow2 {
    public static void main(String[] args) {
        byte stock = -115;
        stock -= 20; // overflow negativo

        System.out.println("Stock: " + stock);
    }
}
```

Overflow1.java

```java
public class Overflow1 {
    public static void main(String[] args) {
        byte stock = -115;
        stock -= 20; // overflow negativo

        System.out.println("Stock: " + stock);
    }
}

```

input

```
Stock: 121
```

Ejercicios de los tipos de errores: Absoluto/Relativo

Ejercicios de vistos en clase

Margen de error típico de una CNC (tolerancia)

- Servicios CNC genéricos suelen trabajar, si no se especifica nada, con tolerancias estándar alrededor de ±0,1 mm.
- Máquinas CNC de alta precisión pueden llegar a ±0,0025 mm (±0,0001").
- Guías comunes indican tolerancias típicas como:
    - Torno CNC: ≈ ±0,005" (≈ ±0,13 mm).
    - Fresado CNC 3 ejes: valores típicos del orden de ±0,005" a ±0,01" dependiendo del proceso.

Supón que la máquina tiene una tolerancia típica de ±0,1 mm al mecanizar una longitud nominal de 50 mm.

1. Tolerancia de la máquina (margen de error): ±0,1 mm.
2. Error absoluto máximo: 0,1 mm (es la desviación permitida respecto al valor nominal).

3. $\text{Error relativo} = 0.1 / 50 \times 100\% = 0{,}2\%$

En contexto de una CNC, si quieres hablar del “margen” propio de la máquina sin aún medir nada, su error absoluto máximo es simplemente su tolerancia:

- Si la CNC tiene tolerancia de ±0,1 mm, el error absoluto máximo de la máquina es 0,1 mm.

Supón una cota nominal de 50 mm y una CNC con tolerancia ±0,1 mm:

- Error absoluto máximo de la máquina: 0,1 mm (ese es su margen).
- Si una pieza sale en 49,94 mm, entonces:
    - Error absoluto real de esa pieza:
    - $|50 - 49{,}94| = 0{,}06$ mm.

Si me dices:

- la tolerancia que te dio el profe o el fabricante (por ejemplo ±0,02 mm), y
- la medida nominal (por ejemplo 80 mm),

Análisis de Precisión: El Monedero de Doña Laura

Margen de error en la estimación (Tolerancia)

En este caso, la "tolerancia" o el margen de error surge de la diferencia entre lo estimado inicialmente y la realidad encontrada al realizar la medición directa (el conteo).

- **Valor Nominal (Estimado):** 160 monedas.
- **Valor Real (Medido):** 156 monedas.

Cálculo de Desviación Técnica

Supongamos que la estimación inicial de 160 monedas es nuestro punto de referencia nominal y el conteo de la nieta es la medida final.

- **Error Absoluto Máximo de la Estimación:** 4 monedas.
  Es la diferencia simple entre lo que se creía tener y lo que realmente hay: |160 - 156| = 4.
- **Error Relativo (Porcentual):**
  Error Relativo = 4 /160 * 100% = 2,5%

Ejercicios de error absoluto/relativo

1. Control de Calidad en Envasado (Volumen)

Se programa una máquina llenadora para verter **500 ml** de refresco en cada botella. Al realizar un muestreo de control, se encuentra que una botella específica contiene **492 ml**.

- **Cota Nominal:** 500 ml.
- **Medida Real:** 492 ml.
- **Cálculo:**
  - **Error Absoluto:** |500 - 492| = 8 ml
  - **Error Relativo:** 8/500 * 100 = 1,6%

2. Topografía y Construcción (Distancia)

Un ingeniero civil utiliza un distanciómetro láser para medir un terreno que, según las escrituras, tiene una longitud de **25,00 metros**. La medición arroja un resultado de **25,05 metros**.

- **Cota Nominal:** 25,00 m.

- **Medida Real:** 25,05 m.
- **Cálculo:**
    - **Error Absoluto:** |25,00 - 25,05| = 0,05 m (o 5 cm).
    - **Error Relativo:** 0,05/25,00 * 100 = 0,2%

3. Termodinámica en Procesos (Temperatura)

Un horno industrial para fundición de aluminio debe mantenerse a una temperatura nominal de **660 °C**. El sensor térmico registra una temperatura real de operación de **673 °C**.

- **Cota Nominal:** 660 °C.
- **Medida Real:** 673 °C.
- **Cálculo:**
    - **Error Absoluto:** |660 - 673| = 13°C
    - **Error Relativo:** 13/660 * 100 approx 1,97%

4. Cronometría Deportiva (Tiempo)

Un entrenador estima que su atleta de élite correrá los 100 metros planos en un tiempo de **10,10 segundos**. Al cruzar la meta, el cronómetro digital marca **9,98 segundos**.

- **Cota Nominal:** 10,10 s.
- **Medida Real:** 9,98 s.
- **Cálculo:**
    - **Error Absoluto:** |10,10 - 9,98| = 0,12 s
    - **Error Relativo:** 0,12/10,10 * 100 approx 1,18%

5. Laboratorio de Química (Masa)

Un estudiante de química necesita pesar **2,50 gramos** de reactivo en una balanza analítica para un experimento. Debido a la sensibilidad del equipo, termina pesando **2,52 gramos**.

- **Cota Nominal:** 2,50 g.
- **Medida Real:** 2,52 g.
- **Cálculo:**
    - **Error Absoluto:** |2,50 - 2,52| = 0,02 g
    - **Error Relativo:** 0,02/2,50 * 100 = 0,8%