TEMA 3. Excepciones

1. Empecemos un tema sobre control de errores en lenguajes de programación, con algo básico. En C, donde no existen las excepciones, pongamos un ejemplo de una raíz que toma número flotante positivo. Queremos controlar el error si la función recibe un número negativo. El usuario debe ser informado pero desde fuera de la función `raiz` ¿Cómo indicamos ese error?. Enumera dos opciones diferentes de diseñar, poniendo un ejemplo de código de cada una.

Aunque lenguajes como Java sí manejan excepciones nativamente, podemos simular este diseño clásico (sin usar bloques `try-catch` ni lanzar errores) para resolver el problema. Para informar al código que llama a la función de que ha ocurrido un problema sin imprimir nada desde dentro, tenemos dos opciones principales de diseño:

Opción 1: Devolver un "valor centinela" (o valor especial)

Consiste en que la función devuelva un valor que normalmente sería imposible como resultado válido de la operación. Como la raíz cuadrada en los números reales nunca puede ser negativa, devolver `-1.0` (o usar la constante `Double.NaN`) es una señal inconfundible de error. El código externo tiene la obligación de comprobar el resultado antes de continuar.

```
public class CalculadoraCentinela {
    // Opción 1: Devuelve -1.0 si hay error (valor centinela)
    public static double raiz(double n) {
        if (n < 0.0) {
            return -1.0; // Señal de error
        }
        return Math.sqrt(n);
    }

    public static void main(String[] args) {
        double num = -4.0;
        double resultado = raiz(num);
```

```
        // El usuario (código externo) comprueba el error desde fuera
        if (resultado < 0.0) {
            System.out.println("Error: No se puede calcular la raiz de un numero negativo.
        } else {
            System.out.println("La raiz es: " + resultado);
        }
    }
}
```

---

2. Brevemente ¿Qué es una "excepción"? ¿Con qué objetivo las usa un programador cuando implementa funciones o cuando las llama?

Una **excepción** es un evento anómalo o inesperado que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de las instrucciones. En lenguajes orientados a objetos como Java, una excepción es literalmente un objeto que empaqueta información sobre un error que acaba de ocurrir (tipo de error, mensaje y en qué línea exacta sucedió).

¿Con qué objetivo las usa un programador?

El objetivo principal de las excepciones es separar el código que realiza la tarea normal (el "camino feliz") del código que gestiona los errores. Su propósito específico depende de lo que esté haciendo el programador:

- **Cuando implementa funciones (Lanzar):** El programador usa excepciones para avisar de que algo ha salido mal y la función no puede terminar su trabajo. En lugar de intentar arreglarlo ahí mismo, "lanza" (con `throw`) la excepción hacia arriba, delegando la responsabilidad. Es una forma de decir: *"Me han pedido hacer esto, pero los datos son inválidos o falló un recurso; que lo resuelva quien me llamó"*.
- **Cuando llama a funciones (Capturar):** El programador las usa para anticiparse a posibles fallos y gestionarlos de forma controlada (usando bloques `try-catch`). El objetivo es evitar que el programa "crashee" o se cierre abruptamente. Si la función llamada falla, el programador captura la excepción y decide qué hacer: mostrar un mensaje amigable al usuario, reintentar la operación o cargar un valor por defecto.

3. Reescribe el mismo ejemplo de raiz, pero en Java, metiendo ese método en una clase `Calculadora` y llama a dicho método desde el método `main`, mostrando cómo se puede controlar desde fuera.

En Java, la forma correcta e idiomática de manejar este tipo de errores es lanzando una excepción. En este caso, como el usuario está pasando un argumento que no es válido matemáticamente (un número negativo), lo ideal es utilizar la excepción estándar `IllegalArgumentException`.

De esta forma, la función `raiz` delega el problema, y es el método `main` (el código externo que hace la llamada) quien lo captura y decide qué hacer con él usando un bloque `try-catch`.

```
public class Calculadora {

    // Método que calcula la raíz y lanza una excepción si el dato es inválido
    public static double raiz(double n) {
        if (n < 0.0) {
            // Lanzamos la excepción interrumpiendo la ejecución normal de este método
            throw new IllegalArgumentException("No se puede calcular la raiz de un numero
        }
        return Math.sqrt(n);
    }

    public static void main(String[] args) {
        double num = -4.0;

        // Intentamos ejecutar el código que podría fallar
        try {
            double resultado = raiz(num);
            System.out.println("La raiz es: " + resultado);
        }
        // Capturamos la excepción si se produce y controlamos el error desde fuera
        catch (IllegalArgumentException e) {
            System.out.println("Error capturado: " + e.getMessage());
            // Aquí el programa no "crashea", el error está completamente controlado
        }
```

```
        System.out.println("El programa continua su ejecucion normalmente...");
    }
}
```

---

4. ¿Qué es "lanzar" una excepción? ¿Qué es "controlar" o "capturar" una excepción? ¿Qué es que se "propague" una excepción? ¿Qué le va ocurriendo a las funciones en la pila de llamadas por donde se va propagando la excepción? ¿Las funciones que no la controlan se reanudan después de alguna forma? Explica con el mismo ejemplo anterior en Java de la raíz cuadrada.

Conceptos clave:

1. **Lanzar (Throw):** Es la acción de indicar explícitamente que ha ocurrido un error anómalo. Se crea un objeto de tipo excepción y se "entrega" al entorno de ejecución, interrumpiendo inmediatamente la secuencia normal del código. En Java se usa la palabra `throw`.
2. **Controlar / Capturar (Catch):** Es el acto de interceptar una excepción que ha sido lanzada, envolviendo el código sospechoso en un bloque `try-catch`. Al capturarla, asumes la responsabilidad del error, ejecutas una solución alternativa y evitas que el programa finalice con un *crash*.
3. **Propagar (Propagate):** Si ocurre una excepción dentro de un método y este no tiene un bloque `catch` para controlarla, el sistema la "pasa" hacia atrás, al método que realizó la llamada. Este viaje inverso continúa hasta que algún método de la cadena la capture (o hasta que llegue al inicio del programa y lo cierre).

¿Qué pasa con la pila de llamadas y las funciones?

A medida que la excepción se propaga hacia arriba, las funciones involucradas en la pila de llamadas (*call stack*) sufren lo que se conoce como **terminación abrupta** (*stack unwinding*). Se destruyen sus variables locales y se eliminan de la memoria.

¿Las funciones que no la controlan se reanudan después de alguna forma? **No, en absoluto.** Una vez que una función es interrumpida por una excepción no controlada, desaparece. Cualquier línea de código que estuviera escrita después de la llamada conflictiva nunca llegará a ejecutarse. El programa solo retomará su flujo normal en el bloque `catch` que logre detener la propagación.

Ejemplo en Java (mostrando la propagación)

```
public class Calculadora {

    // 1. LANZAR: Aquí nace el error
    public static double raiz(double n) {
        if (n < 0.0) {
            // Se interrumpe la función inmediatamente al lanzar la excepción
            throw new IllegalArgumentException("Raíz de número negativo: " + n);
        }
        return Math.sqrt(n);
    }

    // 3. PROPAGAR: Este método hace de puente. No tiene try-catch.
    public static void calcularYMostrar(double n) {
        // Si raiz() lanza la excepción, el código se rompe exactamente aquí.
        double resultado = raiz(n);

        // ¡ATENCIÓN! Si n es negativo, esta línea de abajo NUNCA se ejecuta.
        // La función sufre una terminación abrupta y no se reanuda jamás.
        System.out.println("El resultado es: " + resultado);
    }

    // 2. CAPTURAR: Aquí detenemos la propagación
    public static void main(String[] args) {
        try {
            // El main llama a calcularYMostrar, y este a raiz
            calcularYMostrar(-4.0);
        } catch (IllegalArgumentException e) {
            // Capturamos la excepción que viajó desde raiz() atravesando calcularYMostrar
            System.out.println("Controlando el error desde main: " + e.getMessage());
        }

        System.out.println("El programa sigue vivo.");
```

```
    }
}
```

---

5. ¿Qué ventajas tiene frente a C, la "propagación natural" de las excepciones a través de la pila (*stack*) de llamadas?

La **propagación natural** (automática) de las excepciones ofrece ventajas enormes frente al manejo manual de errores típico de C:

1. **Código mucho más limpio y legible:** En C, debes comprobar el valor de retorno de *cada* función invocada con un bloque `if` (e.g., `if (resultado == -1) return -1;`), mezclando constantemente la lógica principal del programa con la gestión de errores. En lenguajes con excepciones, escribes tu "camino feliz" de forma continua y dejas todo el control de errores centralizado en un único lugar (el `catch`).
2. **Imposibilidad de ignorar errores accidentalmente:** En C, si te olvidas de comprobar si una función devolvió un código de error, el programa continuará su ejecución usando datos corruptos o inválidos, provocando fallos impredecibles más adelante. Las excepciones, por el contrario, no pueden ser ignoradas por accidente; si no las capturas explícitamente, interrumpen el programa, obligándote a lidiar con el problema.
3. **Delegación sencilla:** Permite que la función de bajo nivel que detecta el error simplemente lo lance, sin tener que preocuparse de cómo comunicarlo a través de todas las capas intermedias, ya que el sistema lo propaga automáticamente hasta quien sepa cómo manejarlo.

---

6. En orientación a objetos, ¿las excepciones suelen ser objetos? ¿Qué ventajas tiene esto en términos de encapsulación? ¿Podemos entonces crear excepciones personalizadas?

Sí, absolutamente. En lenguajes orientados a objetos (como Java, C# o Python), las excepciones son objetos instanciados a partir de clases específicas (en Java, todas heredan de `java.lang.Throwable`).

Ventajas en términos de encapsulación:

Al ser un objeto, una excepción no es solo una "señal" vacía. Puede **encapsular en su estado interno** toda la información relevante sobre el fallo en el momento exacto en que ocurrió. En lugar de devolver un simple código numérico, el objeto excepción encapsula un mensaje de texto detallado, el contexto del error, e incluso otras excepciones previas (la "causa"). Todo esto se protege y se expone a través de una interfaz pública (como `getMessage()`).

¿Podemos crear excepciones personalizadas?

Sí. Dado que son simples clases, puedes aprovechar la **herencia** para crear tus propias excepciones que modelen problemas específicos de tu dominio de negocio. Solo tienes que crear una clase que herede de `Exception` o `RuntimeException`.

```
// Ejemplo de excepción personalizada en Java
public class SaldoInsuficienteException extends Exception {
    private final double saldoActual;
    private final double cantidadIntentada;

    public SaldoInsuficienteException(double saldoActual, double cantidadIntentada) {
        super("Operacion denegada: Intento de retirar " + cantidadIntentada + " con saldo
        this.saldoActual = saldoActual;
        this.cantidadIntentada = cantidadIntentada;
    }

    // Encapsulamos información extra útil para quien capture el error
    public double getSaldoActual() { return saldoActual; }
}
```

---

7. En relación con las ventajas de la encapsulación, comparando el ejemplo en C con Java. ¿Qué información

esencial lleva cualquier **objeto excepción** que es muy útil tener cuando se llega a un manejador?

A diferencia de lenguajes como C, donde un error suele ser un simple número (como `-1` o un código almacenado en una variable global como `errno`), un objeto excepción en Java viaja por la pila de llamadas llevando consigo un "expediente" completo del accidente gracias a la encapsulación.

La información esencial más útil que incluye y protege en su estado interno es:

1. **El tipo de error (La Clase):** El simple nombre de la clase ya te indica exactamente qué tipo de fallo ocurrió (por ejemplo, `NullPointerException`, `FileNotFoundException`, `ArithmeticException`).
2. **El mensaje de detalle:** Un texto descriptivo accesible mediante `getMessage()` que explica los pormenores específicos del fallo (ej. *"El archivo 'datos.txt' no existe en la ruta especificada"*).
3. **La Traza de la Pila (*Stack Trace*):** Esta es la ventaja más espectacular. La excepción encapsula una lista exacta de todas las llamadas a métodos que estaban activas en el momento del fallo, incluyendo **el archivo y el número de línea de código exacto** donde se originó. Esto se puede imprimir con `printStackTrace()` y es oro puro para depurar.
4. **La Causa (*Cause*):** Si una excepción provocó otra (por ejemplo, un fallo de red provocó un fallo al cargar la interfaz de usuario), el objeto puede encapsular la excepción original en su interior, manteniendo el historial completo de la reacción en cadena (accesible con `getCause()`).

---

8. En Java, sobre el bloque **"try-catch"**, ¿se pueden tener más de un bloque `catch`? ¿cuántos bloques `catch` se ejecutan?

Sí, se pueden tener múltiples bloques `catch` encadenados inmediatamente después de un único bloque `try`. Esto se hace para manejar diferentes tipos de errores de maneras distintas y específicas.

¿Cuántos bloques se ejecutan?

Solamente se ejecuta UNO (como máximo).

Cuando se lanza una excepción dentro del bloque `try`, el entorno de ejecución evalúa los bloques `catch` secuencialmente de arriba hacia abajo, buscando el primero cuyo tipo coincida con la excepción lanzada (o que sea una superclase de esta).

Una vez que encuentra una coincidencia, entra en ese bloque, lo ejecuta, y automáticamente **salta e ignora todos los demás bloques** `catch` restantes.

> ***Nota importante:*** *Por esta razón de evaluación descendente, si usas múltiples* `catch`*, siempre debes colocar las excepciones más específicas arriba de todo y las más genéricas (como la clase base* `Exception`*) abajo. Si lo haces al revés, el compilador de Java te dará un error indicando que los bloques inferiores son "inalcanzables".*

```
// Ejemplo de múltiples catch en Java (JDK 25)
public void procesarTexto(String texto) {
    try {
        int longitud = texto.length(); // Si texto es null, lanza NullPointerException
        System.out.println("Longitud: " + longitud);
    }
    catch (NullPointerException e) {
        // SI TEXTO ES NULL, SE EJECUTA ESTE BLOQUE
        System.out.println("Error: Se recibió un texto nulo.");
    }
    catch (RuntimeException e) {
        // ESTE SE IGNORA (aunque NullPointerException herede de RuntimeException, el bloq
        System.out.println("Error de ejecución genérico.");
    }
    catch (Exception e) {
        // ESTE SE IGNORA (es el más genérico, se pone al final por si falla algo inespera
        System.out.println("Error catastrófico o inesperado.");
    }
}
```

---

9. Si las excepciones producen rupturas en el código llamador, ¿cómo podemos garantizar que se ejecuta

siempre finalmente un código necesario para cierre de ficheros, liberacion de recursos, antes de que continúe propagándose la excepción? Pon un ejemplo en Java con `finally`, tanto con `catch` como sin él.

Para garantizar que un bloque de código crítico (como el cierre de un recurso) se ejecute siempre, independientemente de si se produce una ruptura por error o no, Java proporciona el bloque `finally`.

Este bloque se coloca al final de una estructura `try`. Su característica principal es que **su ejecución está garantizada en prácticamente cualquier circunstancia**: se ejecutará si el `try` termina con éxito, si ocurre una excepción, e incluso si dentro del `try` o del `catch` se ejecuta una instrucción `return`.

Es el mecanismo clásico para evitar "fugas" o dejar archivos bloqueados cuando una excepción corta de golpe la ejecución de un método.

1. Ejemplo de `finally` CON `catch`

En este caso, si ocurre un error, lo capturamos y lo gestionamos. Pase lo que pase (haya éxito o salte el `catch`), el programa pasará inexorablemente por el `finally` para limpiar antes de continuar.

```
// Ejemplo en Java
public void procesarArchivoConCatch() {
    System.out.println("1. Abriendo conexión al archivo...");
    try {
        System.out.println("2. Leyendo datos...");
        // Simulamos que ocurre un error de lectura
        throw new RuntimeException("El archivo está corrupto");
    }
    catch (RuntimeException e) {
        System.out.println("3. Error capturado: " + e.getMessage());
        // Controlamos el error, pero el archivo sigue abierto
    }
    finally {
        // ESTO SE EJECUTA SIEMPRE (haya saltado el catch o el try haya ido bien)
        System.out.println("4. FINALLY: Cerrando la conexión al archivo de forma segura.")
    }
```

```
    System.out.println("5. El método continúa porque el error fue controlado.");
}
```

Ejemplo de finally SIN catch

Es perfectamente válido usar try seguido únicamente de finally (sin catch). Esto se usa cuando no queremos controlar el error en ese método (queremos que la excepción se propague hacia quien nos llamó), pero aún así necesitamos hacer limpieza local de recursos antes de que la excepción abandone nuestra función y destruya la pila de llamadas.

```
// Ejemplo en Java
public void procesarArchivoSinCatch() {
    System.out.println("1. Abriendo conexión al archivo...");
    try {
        System.out.println("2. Leyendo datos críticos...");
        // Simulamos un error fatal
        throw new RuntimeException("Fallo de hardware inesperado");
    }
    finally {
        // La excepción NO se captura aquí, se propagará hacia arriba inevitablemente.
        // PERO antes de que la excepción logre salir de este método, el finally nos salva
        System.out.println("3. FINALLY: Cerrando la conexión antes de que la excepción se
    }

    // ¡ATENCIÓN! Cualquier código debajo del finally NUNCA se ejecutará
    // porque la excepción no fue controlada y rompió el flujo normal.
    System.out.println("4. Esta línea jamás se imprimirá.");
}
```

---

10. En Java, el bloque `finally` puede ir sin `catch`? ¿Se ejecuta siempre tanto si ocurre como si no ocurre una excepción? ¿Y si hay un `return` en medio del `try`?

Sí, el bloque `finally` puede ir perfectamente sin un `catch`. La única regla de sintaxis en Java es que un bloque `try` debe estar seguido obligatoriamente por al menos un `catch`, por

un `finally`, o por ambos. Usar `try-finally` (sin `catch`) es muy común cuando quieres asegurar la limpieza de un recurso pero no quieres gestionar el error ahí mismo, permitiendo que la excepción siga propagándose hacia el método llamador.

Sí, se ejecuta siempre. Esa es exactamente la razón de existir del `finally`. El código dentro de este bloque se ejecutará indefectiblemente tanto si el bloque `try` llega a su fin con éxito, como si se interrumpe porque se lanza una excepción. *(La única excepción real a esto es si el programa sufre un cuelgue catastrófico, como un fallo de hardware, o si se ejecuta un `System.exit(0)` explícito que apaga la máquina virtual).*

¿Y si hay un `return` en medio del `try`?

El bloque `finally` **también se ejecuta**. Esta es una de las características más potentes (y a veces confusas para los principiantes) de Java.

Si el flujo de ejecución encuentra una instrucción `return` dentro del `try` (o dentro de un `catch`), la salida del método se "pausa" justo antes de devolver el valor al llamador. El control salta inmediatamente al bloque `finally` para ejecutar su contenido, y solo cuando el `finally` termina de ejecutarse, el método hace efectivo el `return` original.

Ejemplo demostrativo:

```
public class EjemploFinally {

    public static String probarReturn() {
        try {
            System.out.println("1. Entrando al try...");
            // Hay un return directo, el flujo normal del método debería acabar aquí
            return "Valor devuelto por el try";
        } finally {
            // Sin embargo, el finally intercepta la salida y se ejecuta ANTES
            System.out.println("2. Ejecutando el finally antes de devolver el valor!");
        }
    }

    public static void main(String[] args) {
        String resultado = probarReturn();
        System.out.println("3. Resultado recibido: " + resultado);
    }
}
```

11. En Java, qué son las excepciones **"controladas"** y las **"no controladas"**? ¿Qué papel juega `RuntimeException`? Pon un ejemplo de excepciones típicas controladas y no controladas que incluso nosotros mismos podríamos usar. Haz dos listas con 3 o 4 ejemplos de situación donde se suele preferir una excepción controlada y donde se suele preferir una excepción no controlada.

En Java, las excepciones se dividen en dos grandes familias según cómo obliga el compilador a tratarlas:

Excepciones Controladas (*Checked Exceptions*)

Son aquellas que **el compilador de Java te obliga a gestionar explícitamente**. Si llamas a un método que lanza una excepción controlada, el código no compilará a menos que la captures con un `try-catch` o declares en la firma de tu propio método que la vas a propagar usando la palabra reservada `throws`. Representan fallos previsibles que escapan al control del programa y de los que una aplicación robusta debería poder recuperarse.

Excepciones No Controladas (*Unchecked Exceptions*)

Son aquellas que **el compilador no te obliga a gestionar**. Puedes poner un `try-catch` si quieres, pero no es obligatorio, ni necesitas usar `throws`. Representan errores de lógica, uso incorrecto de una API o fallos de programación que deberían solucionarse corrigiendo el código fuente, no capturando el error en tiempo de ejecución.

El papel de `RuntimeException`

La clase `RuntimeException` es la pieza central que divide ambos mundos. En la jerarquía de Java, cualquier clase que herede (directa o indirectamente) de `RuntimeException` se considera una excepción **no controlada**. Por el contrario, cualquier clase que herede de `Exception` pero *no* de `RuntimeException`, se considera una excepción **controlada**.

Ejemplos típicos que podemos usar

- **Típicas Controladas:** `IOException` (fallos de lectura/escritura), `FileNotFoundException` (archivo no encontrado), `SQLException` (errores de base de

datos).

- **Típicas No Controladas:** `IllegalArgumentException` (argumento inválido, como nuestra raíz negativa), `NullPointerException` (intentar usar un objeto no instanciado), `IllegalStateException` (el objeto no está en el estado correcto para la operación).

Situaciones donde se prefiere una Excepción CONTROLADA

Se usan cuando el error es provocado por un factor externo al programa y el usuario o el sistema puede hacer algo al respecto para continuar:

1. **Operaciones de entrada/salida (I/O):** Intentar abrir un archivo de configuración que el usuario ha borrado accidentalmente. Se puede capturar y cargar una configuración por defecto.
2. **Comunicaciones por red:** Intentar conectar a una base de datos o a un servidor web remoto que está temporalmente caído (se puede capturar e intentar la reconexión unos segundos después).
3. **Procesamiento de datos externos:** Parsear un archivo XML o JSON subido por el usuario que resulta tener un formato corrupto o mal formado.
4. **Hilos y concurrencia:** Cuando un hilo de ejecución está pausado y es interrumpido por otro proceso (`InterruptedException`).

Situaciones donde se prefiere una Excepción NO CONTROLADA

Se usan cuando el error es culpa del programador (un *bug*) y la mejor solución es que el programa falle estrepitosamente para que el error sea detectado y corregido en el código:

1. **Validación de argumentos (Precondiciones):** Un método que calcula la edad de una persona recibe una fecha de nacimiento en el futuro (`IllegalArgumentException`). No tiene sentido intentar "recuperarse" de esto; el programador que llamó al método lo hizo mal.
2. **Errores lógicos de estado:** Intentar darle al botón de "Pagar" en un carrito de la compra que está vacío (`IllegalStateException`).
3. **Acceso a estructuras de datos:** Intentar acceder a la posición 10 de un array que solo tiene 5 elementos (`IndexOutOfBoundsException`).
4. **Operaciones matemáticas imposibles en enteros:** Intentar dividir un número entero entre cero (`ArithmeticException`).

---

12. ¿Qué es y para qué se usa `throws`? ¿Por qué es alternativa a capturar una excepción controlada?

En Java, `throws` es una palabra reservada que se coloca en la firma o cabecera de un método (justo antes de abrir las llaves `{}`) para declarar formalmente que la ejecución de ese método podría provocar una o varias excepciones concretas.

¿Para qué se usa?

Se utiliza para **delegar la responsabilidad** del manejo del error hacia arriba en la pila de llamadas. Básicamente, el método indica en su "contrato": *"Ojo, ejecutar esta operación es peligroso y podría fallar con este tipo de error. Si me llamas, tú tendrás que hacerte cargo de gestionarlo"*.

¿Por qué es alternativa a capturar una excepción controlada?

En Java existe una regla estricta conocida como *"Catch or Specify"* (Capturar o Especificar) que se aplica a todas las excepciones controladas (*checked exceptions*, como vimos en la pregunta anterior).

Si dentro de tu método utilizas una instrucción que corre el riesgo de lanzar una excepción controlada (como abrir un archivo en disco, que puede lanzar `IOException`), el compilador te bloquea y no te dejará generar el programa a menos que tomes **una de estas dos alternativas**:

1. **Capturarla (Catch):** Envuelves la operación problemática en un bloque `try-catch` y solucionas el error internamente. El problema "muere" ahí, queda controlado y no sale de tu método.
2. **Especificarla (Throws):** Si no sabes cómo solucionar el error en ese punto exacto del programa, o prefieres que lo decida quien llamó al método, añades `throws IOException` a tu firma. Al hacer esto, "esquivas" la obligación de escribir el `try-catch` localmente, pero automáticamente le trasladas el problema (y la obligación del compilador) al método que te haya invocado.

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

public class LectorArchivos {
```

```
    // El método decide NO capturar la excepción y usa throws como alternativa.
    // Delega la responsabilidad de lidiar con el archivo no encontrado al llamador.
    public static void leerConfiguracion(String ruta) throws IOException {

        // El constructor de FileReader puede lanzar una IOException (excepción controlada
        // Como hemos especificado "throws" arriba, el compilador nos perdona no poner try
        FileReader lector = new FileReader(ruta);

        System.out.println("Archivo abierto con éxito.");
    }
}
```

---

13. Pon un ejemplo en Java de firma de método que incluya `throws`, de una función que abre un fichero pero que declara que no le interesa menejar la excepción de si el fichero no existe, sino que se propague hacia arriba. Eso sí, acuérdate del `finally`.

Aquí tienes un ejemplo clásico de cómo se combinan **`throws`** y **`finally`** sin utilizar un bloque **`catch`**.

En este diseño, el método asume que no tiene sentido intentar "arreglar" la falta del archivo internamente. Por ello, declara en su firma que lanza la excepción (**`throws FileNotFoundException`**) para que el método llamador se haga cargo. Sin embargo, utiliza un bloque **`try-finally`** para garantizar de forma absoluta que, si el archivo sí se abrió pero ocurrió un error posterior (o si todo fue bien), el recurso del sistema operativo se cierre correctamente.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class GestorArchivos {

    // La firma del método delega la responsabilidad usando "throws".
```

```
    // Quien llame a procesarArchivo() estará obligado a gestionar estos posibles errores.
    public void procesarArchivo(String ruta) throws FileNotFoundException, IOException {
        FileInputStream archivo = null;

        try {
            // Si el archivo no existe, el constructor lanza FileNotFoundException.
            // Como NO hay bloque catch, la excepción detiene la ejecución aquí mismo
            // y comienza a propagarse hacia arriba.
            archivo = new FileInputStream(ruta);
            System.out.println("Archivo abierto correctamente.");

            // Simulación de lectura u otras operaciones que podrían fallar...

        } finally {
            // El finally nos salva: intercepta la excepción justo antes de que abandone e
            // Garantiza que, pase lo que pase, dejemos la casa limpia.
            System.out.println("Ejecutando bloque finally para liberar recursos...");
            if (archivo != null) {
                archivo.close(); // Cerramos el flujo del archivo si llegó a inicializarse
            }
        }
    }
}
```

---

14. ¿Podemos poner en `throws` excepciones no controladas, como `RuntimeException`? ¿Debería el método llamador entonces poner `try-catch` en ese caso? ¿Qué sentido tendría?

Sí, sintácticamente Java permite poner cualquier excepción en la cláusula `throws`, incluyendo **`RuntimeException`** o cualquiera de sus derivadas (como **`IllegalArgumentException`** o **`NullPointerException`**).

¿Debería el método llamador poner `try-catch`?

No, el compilador no le obligará a hacerlo. La regla de "Capturar o Especificar" de Java ignora por completo las excepciones no controladas. Aunque las declares en la firma con

`throws`, el compilador tratará la llamada exactamente igual que si no estuvieran, por lo que el método llamador no necesita ni `try-catch` ni propagarlas explícitamente.

¿Qué sentido tendría entonces declararlas?

El único sentido real es **puramente informativo y documental**.
Se utiliza para avisar a los programadores que van a consumir tu código de que esa función tiene "trampa" bajo ciertas circunstancias. Al ponerlo en la firma, herramientas como JavaDoc generarán automáticamente en la documentación una advertencia sobre qué dispara ese error, invitando al programador a ser cuidadoso (por ejemplo: `throws IllegalArgumentException si el parámetro es nulo`).

---

15. ¿Cuándo se recomienda usar excepciones controladas, como `IOException`, y cuándo no controladas como `IllegalArgumentException`? ¿Existen en todos los lenguajes ambas opciones? En los que sólo existe una opción, ¿cuál es la más habitual?

Como regla general de diseño en Java:

- **Usa excepciones controladas (Checked - `IOException`):** Cuando el error provenga de una condición ambiental ajena a la lógica de tu programa (el disco duro está lleno, se cortó el internet, el usuario introdujo un archivo inválido) y esperas de forma realista que el programa que te llame **pueda recuperarse** del problema (ej. pidiendo otro archivo).
- **Usa excepciones no controladas (Unchecked - `IllegalArgumentException`):** Cuando el error sea un **defecto de programación** (un *bug*), un fallo lógico o el uso incorrecto de una API. No se espera que el programa se recupere, sino que el desarrollador corrija el código fuente para que la excepción no llegue a producirse nunca.

¿Existen en todos los lenguajes ambas opciones?

No. De hecho, Java es una rareza en la industria del software moderna en este aspecto. Fue un experimento pionero para intentar hacer el software más seguro, pero la división entre controladas y no controladas no cuajó fuera de su ecosistema.

En los que sólo existe una opción, ¿cuál es la más habitual?

La opción universal es la **excepción no controlada**.
Lenguajes modernos y populares como **C#, Python, C++, JavaScript, Ruby o Kotlin** NO tienen excepciones controladas. En todos ellos, cualquier excepción lanzada se comporta exactamente igual que un `RuntimeException` en Java: el compilador nunca te obliga a capturarlas y se propagan libremente por la pila de llamadas hasta que alguien decide atraparlas o el programa falla. La industria se decantó por este modelo porque las excepciones controladas obligaban a escribir firmas de métodos demasiado largas y ensuciaban el código con bloques `catch` vacíos.

---

16. ¿Tiene sentido lanzar excepciones dentro del `catch`? ¿Se puede relanzar la misma excepción capturada? ¿Cuándo tendría sentido hacer esto último? Pon ejemplos de ambos casos.

Sí, tiene mucho sentido y es una práctica extremadamente común en el diseño de software robusto. Existen dos escenarios principales: lanzar una *nueva* excepción o *relanzar* la misma.

Caso 1: Lanzar una nueva excepción (Traducción de Excepciones)

Se utiliza para "ocultar" los detalles técnicos de bajo nivel y lanzar un error que tenga sentido para la lógica de negocio. Por ejemplo, capturas un error técnico de base de datos (`SQLException`) y lanzas una excepción personalizada (`UsuarioNoEncontradoException`). Es fundamental pasar la excepción original como argumento para no perder el *Stack Trace* original (esto se llama encadenamiento o *chaining*).

```
// Ejemplo de lanzar una NUEVA excepción dentro del catch
public Usuario buscarUsuario(String id) throws UsuarioInvalidoException {
    try {
        // Operación de base de datos que falla...
        return baseDeDatos.ejecutarConsulta("SELECT * FROM users WHERE id = " + id);
    } catch (SQLException e) {
        // Capturamos el error técnico de SQL y lanzamos uno de negocio.
        // Pasamos 'e' para que el log mantenga la causa técnica real.
        throw new UsuarioInvalidoException("Fallo al buscar el usuario " + id, e);
```

```
    }
}
```

---

17. ¿En qué consiste que una excepción sea la "causa" de otra excepción? Pon un ejemplo en Java, donde capturemos una excepción de bajo nivel y la encapsulemos en otra personalizada de alto nivel. Cuando una excepción sale por pantalla y tiene una causa, ¿se ve?

Respuesta