

Guia de Repaso

Excepciones y Try-with-Resources

—

Semana 02 - Lunes 16 de Febrero de 2026

Academia Java - Ciudad de Mexico

Temas del dia:

1. Excepciones: conceptos basicos
2. Excepciones Checked vs Unchecked
3. Excepciones personalizadas (Custom)
4. Multi-catch (Java 7+)
5. Excepciones en herencia
6. Try-with-Resources

Instructor: Miguel Rugerio

1. Excepciones: conceptos basicos

Una **excepcion** es un evento que ocurre durante la ejecucion de un programa y que interrumpe el flujo normal de instrucciones. Si no se maneja, el programa se detiene.

Sin manejo de excepciones

Si dividimos entre cero sin try-catch, el programa se detiene abruptamente:

```
public static void main(String[] args) {  
    int x = 9;  
    int y = 0;  
    int resultado = dividir(x, y); // ArithmeticException!  
    System.out.println("Resultado: " + resultado);  
    System.out.println("Fin de Programa"); // Nunca se ejecuta  
}  
  
private static int dividir(int x, int y) {  
    return x / y;  
}
```

El programa lanza `ArithmeticException: / by zero` y se detiene. La linea "Fin de Programa" nunca se ejecuta.

Con try-catch

Envolvemos el codigo riesgoso en un bloque `try` y capturamos la excepcion en `catch`:

```
try {  
    resultado = dividir(x, y);  
} catch (ArithmeticException e) {  
    System.out.println("Exception!!!!");  
}  
System.out.println("Resultado: " + resultado);  
System.out.println("Fin de Programa"); // Si se ejecuta
```

Con try-catch el programa no se detiene. El flujo continua despues del bloque catch.

throw y throws

Palabra clave	Que hace	Donde se usa
throw	Lanza una excepcion	Dentro del metodo
throws	Declara que un metodo puede lanzar una excepcion	En la firma del metodo

```
private static int dividir(int x, int y) throws Exception {  
    if (y == 0)  
        throw new Exception("No se puede dividir entre Cero!!!!");  
    return x / y;  
}
```

Propagacion de excepciones

Si un metodo no captura la excepcion, puede propagarla al metodo que lo llama usando

`throws :`

```
// La excepcion se propaga de dividir() -> main() -> JVM (crash)  
public static void main(String[] args) throws Exception {  
    int resultado = dividir(9, 0);  
}
```

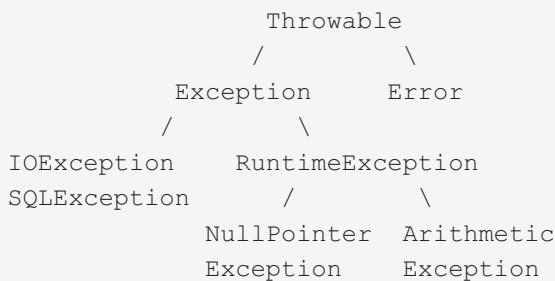
Si nadie captura la excepcion, la JVM la recibe y termina el programa con el stack trace.

2. Excepciones Checked vs Unchecked

Java tiene dos tipos principales de excepciones:

Tipo	Hereda de	Obligatorio capturar?	Ejemplos
Checked	Exception	Si (el compilador te obliga)	IOException, SQLException
Unchecked	RuntimeException	No (opcional)	NullPointerException, ArithmaticException

Jerarquía de excepciones



Checked: el compilador te obliga

```

// Esta clase extiende Exception -> es CHECKED
public class DividirEntreCero extends Exception {
    DividirEntreCero(String msg) {
        super(msg);
    }
}

// OBLIGATORIO: throws en la firma + try-catch en quien lo llama
private static int dividir(int x, int y) throws DividirEntreCero {
    if (y == 0)
        throw new DividirEntreCero("No se puede dividir entre cero!!!");
}

```

```
    return x / y;
}
```

Unchecked: el compilador no te obliga

```
// Esta clase extiende RuntimeException -> es UNCHECKED
public class DividirEntreCero extends RuntimeException {
    DividirEntreCero(String msg) {
        super(msg);
    }
}

// NO necesita throws ni try-catch (pero es buena practica manejarla)
private static int dividir(int x, int y) {
    if (y == 0)
        throw new DividirEntreCero("No se puede dividir entre cero!!!!");
    return x / y;
}
```

Regla para recordar: Si tu excepcion extiende `Exception` = checked (obligatorio). Si extiende `RuntimeException` = unchecked (opcional).

3. Excepciones personalizadas

Puedes crear tus propias excepciones para representar errores específicos de tu aplicación.

Crear una excepción custom

```
public class DividirEntreCero extends Exception {
    DividirEntreCero(String msg) {
        super(msg); // Pasa el mensaje al constructor de Exception
    }
}

public class DivisorNoNegativo extends Exception {
    DivisorNoNegativo(String msg) {
        super(msg);
    }
}
```

Método que lanza múltiples excepciones

```
private static int dividir(int x, int y)
    throws DividirEntreCero, DivisorNoNegativo {

    if (y == 0)
        throw new DividirEntreCero("No se puede dividir entre cero!!!!");
    if (y < 0)
        throw new DivisorNoNegativo("No se puede dividir con negativo");
    return x / y;
}
```

Capturar múltiples excepciones (bloques separados)

```
try {
    resultado = dividir(x, y);
} catch (DividirEntreCero e) {
    e.printStackTrace();
} catch (DivisorNoNegativo e) {
    e.printStackTrace();
}
```

Agregar Exception como catch generico

Se puede agregar un `catch (Exception e)` al final como "red de seguridad":

```
try {
    resultado = dividir(x, y);
} catch (DividirEntreCero e) {
    e.printStackTrace();
} catch (DivisorNoNegativo e) {
    e.printStackTrace();
} catch (UnsupportedOperationException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.printStackTrace(); // Captura cualquier otra excepcion
}
```

Orden importante: Los catch van de mas especifico a mas generico. Si pones `Exception` primero, los demas catch son inalcanzables y el compilador da error.

4. Multi-catch (Java 7+)

Desde Java 7, puedes agrupar multiples excepciones en un solo bloque catch usando el operador pipe | .

Antes de Java 7 (bloques separados)

```
try {
    resultado = dividir(x, y);
} catch (DividirEntreCero e) {
    e.printStackTrace();
} catch (DivisorNoNegativo e) {
    e.printStackTrace();
} catch (UnsupportedOperationException e) {
    e.printStackTrace();
}
```

Con multi-catch (Java 7+)

```
try {
    resultado = dividir(x, y);
} catch (DivisorNoNegativo | DividirEntreCero
        | UnsupportedOperationException e) {
    e.printStackTrace();
}
```

Aspecto	Bloques separados	Multi-catch
Lineas de codigo	Mas codigo, mas repeticion	Menos codigo, mas limpio
Manejo diferente por tipo	Si, cada catch hace algo distinto	No, todos se manejan igual
Cuando usarlo	Cuando cada excepcion requiere manejo diferente	Cuando el manejo es el mismo para varias

Regla del multi-catch: Las excepciones en un multi-catch NO pueden tener relación de herencia entre ellas. Por ejemplo, no puedes poner `Exception | IOException` porque `IOException` ya es hija de `Exception`.

5. Excepciones en herencia

Cuando una subclase sobreescribe (`@Override`) un metodo de la superclase, hay reglas estrictas sobre las excepciones que puede declarar.

Reglas de override con excepciones

```
class SomeException extends Exception {}
class OtherException extends SomeException {}

class A {
    protected void m() throws SomeException {}
}

class B extends A {
    @Override
    public void m() {} // Opcion 1: Eliminar la excepcion (VALIDO)

    // public void m() throws SomeException {} // Opcion 2: Misma excepcion (VALIDO)
    // public void m() throws OtherException {} // Opcion 3: Subclase (VALIDO)
}
```

La subclase puede...	Valido?	Ejemplo
Eliminar la excepcion	Si	<code>void m() {}</code>
Declarar la misma excepcion	Si	<code>void m() throws SomeException</code>
Declarar una subclase de la excepcion	Si	<code>void m() throws OtherException</code>
Declarar una superclase de la excepcion	No	<code>void m() throws Exception</code> (no compila)
Declarar una excepcion nueva no relacionada	No	<code>void m() throws IOException</code> (no compila)

Polimorfismo y excepciones

```
A a = new B();
((B)a).m();           // OK: B.m() no declara excepcion
```

```
A aa = new B();
try {
    aa.m();      // Compilador ve A.m() que throws SomeException
} catch (SomeException e) {
    e.printStackTrace();
}
```

Clave: El compilador decide que excepciones manejar basandose en el *tipo de la referencia*, no en el tipo real del objeto. Si la referencia es tipo `A`, el compilador te obliga a manejar `SomeException`.

Lanzar null como excepcion

```
try {
    RuntimeException re = null;
    throw re; // Lanza NullPointerException, no la excepcion original
} catch (Exception e) {
    System.out.println(e); // java.lang.NullPointerException
}
```

Dato curioso: Si intentas lanzar (`throw`) una referencia null, Java lanza un `NullPointerException` en lugar de la excepcion que esperabas.

6. Try-with-Resources

Cuando trabajamos con recursos (conexiones a BD, archivos, streams), es importante cerrarlos siempre. Java ofrece tres formas de hacerlo, cada una mejor que la anterior.

Problema: fuga de recursos (v0)

```
MongoDb conMongoQA = new MongoDb("MongoDB QA");

try {
    conMongoQA.open();
} catch (Exception e) {
    e.printStackTrace();
}

// PROBLEMA: Si open() falla, NUNCA cerramos la conexion
System.out.println("Program End");
```

Si ocurre una excepcion, el recurso queda abierto. Esto causa fugas de memoria y conexiones huérfanas.

Solucion con finally (v1)

El bloque `finally` se ejecuta **siempre**, haya o no excepcion:

```
MongoDb conMongoQA = new MongoDb("MongoDB QA");

try {
    conMongoQA.open();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        conMongoQA.close(); // SIEMPRE se ejecuta
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Funciona, pero el código es verboso: necesitas un try-catch dentro del finally porque `close()` también puede lanzar excepción.

Solucion moderna: try-with-resources (v2)

Paso 1: La clase debe implementar `AutoCloseable`:

```
public class MongoDb implements AutoCloseable {
    private String name;

    MongoDb(String name) {
        this.name = name;
    }

    boolean open() throws Exception {
        System.out.println("Abrir Conexion MongoDB");
        return true;
    }

    @Override
    public void close() throws Exception {
        System.out.println("Cerrar Conexion MongoDB");
    }
}
```

Paso 2: Usar la sintaxis `try (recurso) { }`:

```
// Try-with-Resources: close() se llama automaticamente
try (MongoDb conMongoQA = new MongoDb("MongoDB QA")) {
    conMongoQA.open();
} catch (Exception e) {
    e.printStackTrace();
}
// close() ya se ejecuto automaticamente aqui
```

Aspecto	Sin finally	Con finally	Try-with-Resources
Cierre garantizado	No	Si	Si
Código limpio	Si (pero inseguro)	No (verboso)	Si
Requiere AutoCloseable	No	No	Si
Versión mínima de Java	1.0	1.0	7+

Regla: Siempre que trabajes con recursos (conexiones, archivos, streams), usa **try-with-resources**. Es la forma moderna, limpia y segura de garantizar que los recursos se cierren.

Resumen de conceptos clave

Concepto	Descripcion
<code>try-catch</code>	Captura excepciones para que el programa no se detenga
<code>throw</code>	Lanza una excepcion manualmente dentro de un metodo
<code>throws</code>	Declara en la firma del metodo que puede lanzar una excepcion
Checked	Extiende <code>Exception</code> . El compilador obliga a manejarla
Unchecked	Extiende <code>RuntimeException</code> . No es obligatorio manejarla
Custom Exception	Crear tu propia clase que extienda <code>Exception</code> o <code>RuntimeException</code>
Multi-catch	<code>catch (A B C e)</code> para agrupar excepciones con el mismo manejo
<code>finally</code>	Bloque que se ejecuta siempre, haya o no excepcion
<code>AutoCloseable</code>	Interface que permite usar try-with-resources
Try-with-Resources	<code>try (recurso) { }</code> cierra automaticamente el recurso

Progresion de los ejercicios del dia

Paquete	Version	Tema
<code>com.curso</code>	v0	Excepciones basicas, try-catch, throw/throws, propagacion
<code>com.curso</code>	v1	Excepcion custom checked (extends <code>Exception</code>)
<code>com.curso</code>	v2	Excepcion custom unchecked (extends <code>RuntimeException</code>)
<code>com.curso</code>	v3	Multiples excepciones custom con catch separados
<code>com.curso</code>	v4	Catch generico con <code>Exception</code> como red de seguridad
<code>com.curso</code>	v5	Multi-catch con operador pipe (<code> </code>)
<code>com.sim</code>	v0	Propagacion de excepciones con <code>Math.random()</code>

com.sim	v1	Excepciones en herencia y reglas de override
com.sim	v2	Lanzar null como excepcion (NullPointerException)
com.tryResource	v0	Problema: fuga de recursos
com.tryResource	v1	Solucion: try-catch-finally
com.tryResource	v2	Solucion moderna: try-with-resources + AutoCloseable