



Manejo de errores

Teoría: Introducción a las ciencias de la computación

Ayudante: Cynthia Lizbeth Sánchez Urbano

del 2025

1. Errores en tiempo de compilación y errores en tiempo de ejecución.

Existen dos tipos de errores a la hora de programar, los primeros son errores que pueden detectarse *en tiempo de compilación*, estos son fáciles de arreglar ya que el compilador de Java es bastante útil al señalarnos la línea del error y datos sobre lo que encontró. Los errores de este tipo están relacionados con una mala sintaxis o mala semántica.

Veamos algunos ejemplos:

Código 1: Error de compilación, falta de ;

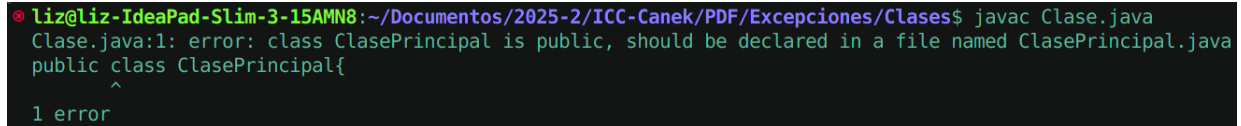
```
1 /**
2  * Método principal imprime la cadena "Hola Mundo" en terminal.
3  */
4 public static void main(String[] args) {
5     System.out.println("Hola_Mundo");
6 }
```

```
liz@liz-IdeaPad-Slim-3-15AMN8:~/Documentos/2025-2/ICC-Canek/PDF/Excepciones/Clases$ javac Clase.java
Clase.java:3: error: ';' expected
    System.out.println("Hola Mundo")
                        ^
1 error
```

Figura 1: Resultado de compilar el código 1.

Código 2: Error de compilación, el nombre de la clase es diferente al del archivo.

```
1 // Nombre del archivo Clase.java
2
3 /**
4  * Clase que imprime "Hola Mundo" en pantalla.
5  */
6 public class ClasePrincipal{
7     public static void main(String[] args) {
8         System.out.println("Hola_Mundo");
9     }
10 }
```



```

liz@liz-IdeaPad-Slim-3-15AMN8:~/Documentos/2025-2/ICC-Canek/PDF/Excepciones/Clases$ javac Clase.java
Clase.java:1: error: class ClasePrincipal is public, should be declared in a file named ClasePrincipal.java
public class ClasePrincipal{
    ^
1 error

```

Figura 2: Resultado de compilar 2.

Por otro lado tenemos errores relacionados con alguna falla *en tiempo de ejecución*, este tipo de errores los podemos dividir en dos, por un lado hay errores que podrían evitarse y ocurren por culpa del programador y por otro existen errores que ocurren aún tomando precauciones puesto que están relacionados con circunstancias bajo las que no tenemos un control directo.

2. Errores prevenibles y no prevenibles.

Los **errores prevenibles** están ligados al funcionamiento y lógica que se maneja al programar, es decir, este tipo de errores suceden cuando no tomamos en cuenta la probabilidad de que ocurra algo que sabemos podría ocurrir, un ejemplo muy fácil de ver es el siguiente:

Código 3: División entre cero.

```

1 /**
2  * Método div, recibe dos enteros y regresa la división del primero entre el segundo.
3  * @param a, el primer entero
4  * @param b, el segundo entero
5  * @return el resultado de a/b en float.
6  */
7 public float div(int a, int b) {
8     return a / b;
9 }

```

Supongamos que estamos creando una clase Calculadora que interactúa con un usuario, si cuando el usuario pide hacer una división introduce el segundo número como cero nuestro programa fallaría, el que alguien coloque cero como un posible número en esta operación en algo que puede prevenirse. Otro caso es cuando intentamos usar un objeto *null* para acceder a algo en particular, puede ser un método, una variable, etc.

Código 4: Acceder con un objeto null.

```

1 /**
2  * Metodo auxiliar de elimina llamado buscaNodo
3  * @param e, el estudiante que estamos buscando
4  * @param n, el nodo inicial desde donde comenzaremos a buscar.
5  * @return el nodo que contiene al estudiante o null en caso de que no lo encontremos.
6  */
7 private Nodo buscaNodo(Estudiante e, Nodo n) {
8     if (n == null)
9         return null;
10    return n.elemento.equals(e) ? n : buscaNodo(e, n.siguiente);
11 }
12
13 /**
14  * Elimina un elemento de la lista. Si el elemento no está contenido en la

```

```

15  * lista, el método no la modifica.
16  *
17  * @param elemento el elemento a eliminar.
18  */
19  public void elimina(Estudiante elemento) {
20      Nodo eliNodo = buscaNodo(elemento, cabeza);
21      // Posible error si el estudiante que se busco no pertenece a la lista.
22      if (eliNodo.elemento.equals(cabeza.elemento)) {
23          eliminaPrimero();
24          return;
25      }
26      // Más código ...
27  }

```

El código anterior podría fallar si en *elimina* al llamar a *buscaNodo* nos regresa un nodo *null*, cuando hacemos *eliNodo.elemento* estamos asumiendo que *eliNodo* es distinto de *null* pero como esta declarado el código no podemos asegurar eso, en este caso lo que deberíamos hacer antes de intentar el operador punto con *eliNodo* es verificar que *eliNodo* sea distinto de *null*, de esta manera ya podríamos garantizar que usar el operador punto no nos provocara un problema.

Los **errores no prevenibles** como bien lo indicamos están fuera de nuestro control, un corte de luz, que la computadora se descomponga, etc. Este tipo de situaciones no son prevenibles y en la gran mayoría no podemos hacer mucho, por otro lado hay situaciones que podemos considerar y manejar un poco mejor como: que un servidor se caiga o que ya no exista, que nos den de baja el Internet, etc.

Los errores prevenibles lanzan una excepción instancia de la clase *RuntimeExcepción* pero entonces ¿Qué es una excepción?

Class RuntimeException

```

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException

```

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AnnotationTypeMismatchException, ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, CompletionException, ConcurrentModificationException, DataBindingException, DateTimeException, DOMException, EmptyStackException, EnumConstantNotPresentException, EventException, FileSystemAlreadyExistsException, FileSystemNotFoundException, IllegalArgumentException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, IllformedLocaleException, ImagingOpException, IncompleteAnnotationException, IndexOutOfBoundsException, JMRuntimeException, LSEException, MalformedParameterizedTypeException, MalformedParametersException, MirroredTypesException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NoSuchMechanismException, NullPointerException, ProfileDataException, ProviderException, ProviderNotFoundException, RasterFormatException, RejectedExecutionException, SecurityException, SystemException, TypeConstraintException, TypeNotPresentException, UncheckedIOException, UndeclaredThrowableException, UnknownEntityException, UnmodifiableSetException, UnsupportedOperationException, WebServiceException, WrongMethodTypeException

Figura 3: Clase RuntimeException y sus clases hijas.

3. Excepciones

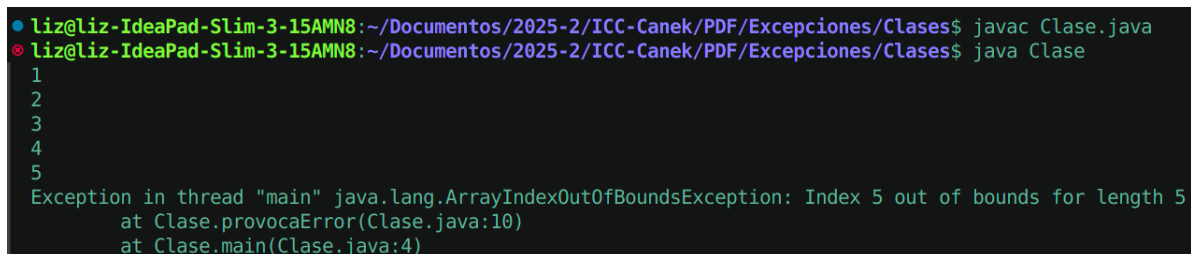
En Java las excepciones son *objetos instancia de la clase Exception* que son lanzadas mediante una estructura de control llamada *throw* las cuales nos permiten manejar errores. Cuando un método lanza una excepción es porque encontró un problema que no sabe manejar, esta acción provoca la interrupción del método y la descarga del registro de activación de la pila. La excepción recorrerá los registros de la pila y se intentará que alguno pueda lidiar con ella hasta llegar al método que inicio todo el despliegue (main) pero si ninguno puede lidiar con ella la maquina virtual la atrapa y muestra el recorrido que hizo la excepción, esto no es recomendable además de que para un usuario común es confuso y hasta preocupante pues la mayoría de usuarios estándar no saben cómo funciona un programa a este nivel.

Código 5: Excepción con recorrido.

```

1 /**
2  * Clase que provoca un error de tipo ArrayIndexOutOfBoundsException
3  */
4 public class Clase {
5
6     /**
7      * Método principal que manda a llamar al método provocaError
8      */
9     public static void main(String[] args) {
10         Clase c = new Clase();
11         c.provocaError();
12     }
13
14     /**
15      * Método que imprime un arreglo de 5 elementos, pero intenta imprimir un
16      * elemento más provocando un error de tipo ArrayIndexOutOfBoundsException
17      */
18     public void provocaError() {
19         int[] arreglo = { 1, 2, 3, 4, 5 };
20         for (int i = 0; i <= 5; i++) {
21             System.out.println(arreglo[i]);
22         }
23     }
24 }

```



```

liz@liz-IdeaPad-Slim-3-15AMN8:~/Documentos/2025-2/ICC-Canek/PDF/Excepciones/Clases$ javac Clase.java
liz@liz-IdeaPad-Slim-3-15AMN8:~/Documentos/2025-2/ICC-Canek/PDF/Excepciones/Clases$ java Clase
1
2
3
4
5
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
    at Clase.provocaError(Clase.java:10)
    at Clase.main(Clase.java:4)

```

Figura 4: Resultado de compilar 7. Podemos ver que muestra los métodos que se llamaron empezando por el método donde ocurrió el error y finalizando con main.

Cuando creamos un método que puede lanzar una excepción que no es instancia de *RuntimeException*

tion¹ tenemos que especificar la excepción para ello haremos uso de la palabra *throws*:

Código 6: Método que puede lanzar una excepción.

```

1 public void leerArchivo(String nombre) throws IOException { // Declaramos que puede
    lanzar una excepción.
2     BufferedReader br = new BufferedReader(new FileReader(nombre));
3     System.out.println(br.readLine());
4     br.close();
5 }

```

Debido a que cuando las excepciones no son manejadas pueden verse en pantalla y eso mostraría el recorrido por la pila nos gustaría prevenir que esto ocurra para ello podemos hacer uso de un bloque *try-catch* es cual es un mecanismo con el cual podemos atrapar la excepción y hacer un manejo distinto de ella, muchas veces este manejo puede brindar un mensaje más simplificado y claro del problema.

Código 7: Cachando una excepción.

```

1 public String leerArchivo(String nombreArchivo) {
2     try {
3         BufferedReader reader = new BufferedReader(new FileReader(nombreArchivo));
4         // más código para leer un archivo
5         reader.close();
6         return contenido.toString();
7     } catch (FileNotFoundException e) {
8         System.err.println("El archivo no se encontró: " + nombreArchivo);
9         return null;
10    }

```

Si queremos crear una excepción propia podemos hacerlo mediante la creación de una clase que extienda a *Exception* o *RuntimeException*

4. Preguntas

- Conceptualmente, ¿Qué es una excepción? Un error inesperado al ejecutar un programa.
- En Java ¿Qué es una excepción? Un objeto instancia de la clase *Exception*.
- ¿En que excepciones no es necesario usar el bloque *try-catch*? ¿Por que? En aquellas que extienden de *RuntimeException*, porque son errores que se pueden prevenir y están relacionadas a un descuido del programador.

¹Los errores prevenibles son instancia de *RuntimeException*, y para estos no es necesario especificar que van a ser lanzados o hacer un bloque *try-catch*.