

Programación Orientada a Objetos

Excepciones y robustez

Lanzar, capturar, propagar · checked y unchecked · recursos y contratos

Gerson Garrido

Semana 9

UNIVERSIDAD NACIONAL DE INGENIERÍA

Esta semana

Sesión A: excepciones

1. Por qué no códigos de error
2. `throw`, `try`, `catch`
3. Propagación y stack traces
4. La jerarquía de `Throwable`
5. Excepciones propias y causa

Sesión B: robustez

6. Checked frente a unchecked
7. `finally` y `try`-with-resources
8. Contratos y fail fast
9. `assert`
10. Probar las excepciones

> **Lectura**
>
> Eck (v. 9): 3.7.1–3.7.2, 8.2.2–8.2.4, 8.3 y 8.4.1. Código: `code/week-09`.

SECCIÓN 01

Por qué excepciones

01

Así se avisa un error en C

transit_card.c · semana 1

```c
int card_pay(struct card *c, int fare)
    {
    if (c->balance < fare)
        return 0;       /* no alcanza
     */
    c->balance -= fare;
    return 1;
}
```

- El error se puede ignorar: `card_pay(&c, 250);` sin `if` compila sin advertencias.
- El código normal se mezcla con los `if (r == -1)`.
- El error sube a mano: cada función del camino tiene que devolverlo.
- ¿Qué devuelve `parse("12a")`? Todo `int` es un resultado válido.

La idea: lanzar el error

jshell

```
jshell> int n = Integer.parseInt("12a")
|  Exception java.lang.NumberFormatException: For input string: "12a"
|        at NumberFormatException.forInputString (NumberFormatException.java
    :67)
|        at Integer.parseInt (Integer.java:565)
|        at Integer.parseInt (Integer.java:662)
|        at (#1:1)
```

- `parseInt` no devuelve nada: lanza *(throw)* un objeto excepción.
- La JVM busca, subiendo por las llamadas, quién la captura *(catch)*. Si nadie: termina el hilo; en nuestros programas, de un solo hilo, el programa.
- El código normal va de corrido; el manejo del error, donde se sepa qué hacer.

SECCIÓN 02

Lanzar, capturar, propagar

02

throw

BankAccount.java

```java
public void deposit(long amount) {
    if (amount <= 0) {
        throw new IllegalArgumentException(
                "monto no positivo: " +
 amount);
    }
    balance += amount;
}
```

- Una excepción es un objeto: se crea con `new`.
- `throw` termina el método, como un `return`...
- ...pero no vuelve normalmente: empieza la búsqueda de un `catch`.
- `balance += amount` no se ejecuta.

`try` y `catch`

```java
try {
    Money price = Money.parse(text);
    IO.println("precio: " + price);
} catch (NumberFormatException e) {
    IO.println("no es un precio: "
            + e.getMessage());
}
IO.println("sigue");
```

- Si `parse` termina bien: el `try` sigue y el `catch` se salta.
- Si lanza: el resto del `try` se salta, y corre el `catch` con `e` apuntando a la excepción.
- En los dos casos, después se imprime `sigue`.
- Un `catch` captura su tipo y sus subtipos: polimorfismo.

Varios `catch`: el orden importa

errores/CatchOrder.java

```java
try (FileReader in = new FileReader("
     precios.txt")) {
    IO.println(in.read());
} catch (Exception e) {
    IO.println("algo falló");
} catch (IOException e) {
    IO.println("no se pudo leer");
}
```

bash

```
$ make errores
...
errores/CatchOrder.java:11:
    error: exception IOException
     has already been caught
```

- Se prueban de arriba abajo: gana el primero que coincide.
- Los específicos van primero.

Multi-catch para tipos que se manejan igual: `catch (NumberFormatException | IllegalStateException e) { ... }`

Propagación

Propagation.java

```java
static void charge(BankAccount account, long amount)
        throws InsufficientFundsException {
    IO.println("  charge: retira " + amount);
    account.withdraw(amount);
    IO.println("  charge: listo");
}
static void processOrder(BankAccount account, long
     total)
        throws InsufficientFundsException {
    IO.println(" processOrder: total " + total);
    charge(account, total);
    IO.println(" processOrder: pedido pagado");
}
```

- Si un método no captura, termina en ese punto…
- …y la excepción aparece en quien lo llamó, en la línea de la llamada.
- Sigue subiendo hasta un `catch` de su tipo.

Propagación: `main` captura

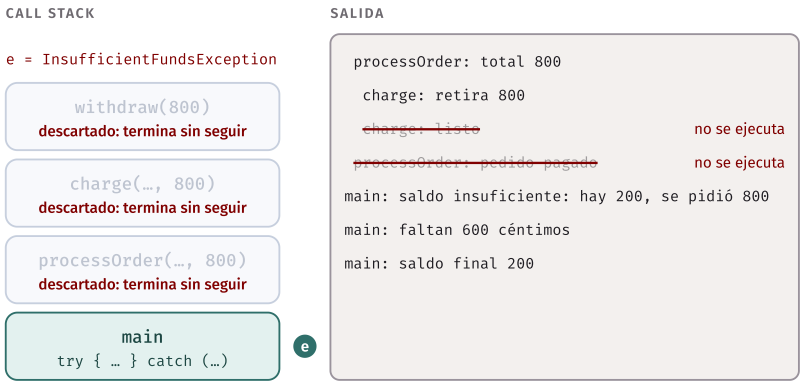
Propagation.java · cuenta con 500

```java
try {
    processOrder(account, 300);
    processOrder(account, 800);
    IO.println("main: no llega aquí");
} catch (InsufficientFundsException e) {
    IO.println("main: " + e.getMessage());
    IO.println("main: faltan "
        + (e.requested() - e.balance())
        + " céntimos");
}
```

bash

```
$ java -cp build Propagation
...
 processOrder: pedido pagado
 processOrder: total 800
  charge: retira 800
main: saldo insuficiente: hay 200,
     se pidió 800
main: faltan 600 céntimos
main: saldo final 200
```

La excepción baja por el call stack



4. el catch de main la captura y el programa sigue

Si nadie la captura: el stack trace

bash

```
$ java -cp build Propagation crash
 processOrder: total 800
  charge: retira 800
Exception in thread "main" InsufficientFundsException: saldo insuficiente: hay
    500, se pidió 800
    at BankAccount.withdraw(BankAccount.java:24)
    at Propagation.charge(Propagation.java:7)
    at Propagation.processOrder(Propagation.java:13)
    at Propagation.main(Propagation.java:21)
```

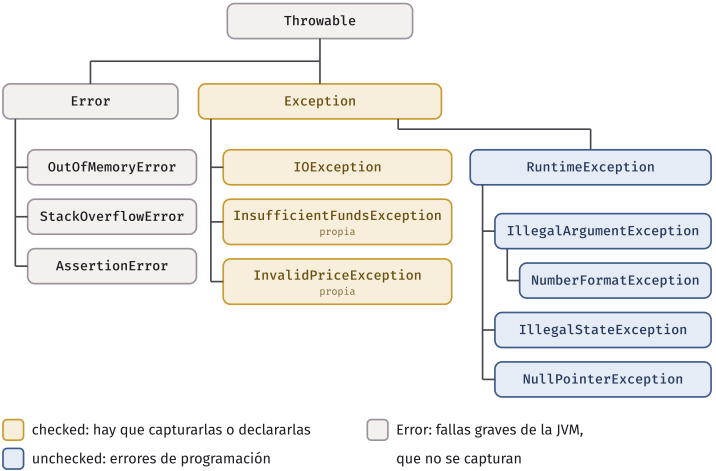
1. Primera línea: qué pasó. Primera línea `at`: dónde se lanzó; las siguientes, desde dónde se llamó.
2. Para depurar: la primera línea `at` que esté en su código. (Código de salida: 1.)

SECCIÓN 03

La jerarquía

03

Todo lo que se lanza es un `Throwable`



Tres ramas

`Error`

Fallas graves de la JVM: sin memoria, recursión sin fin. Un programa normal no las captura.

`RuntimeException`

Errores de programación: `null`, índice malo, argumento inválido. **Unchecked**.

Las demás `Exception`

Fallas que el que llama puede esperar: un archivo que no está, saldo insuficiente. **Checked**.

- `NumberFormatException` es una `IllegalArgumentException`.
- En `Throwable`: `getMessage()`, `getCause()`, `printStackTrace()`.

SECCIÓN 04

Excepciones propias

04

Una excepción del dominio

InsufficientFundsException.java

```java
public class InsufficientFundsException
        extends Exception {
    private final long balance;
    private final long requested;
    public InsufficientFundsException(
            long balance, long requested) {
        super("saldo insuficiente: hay " +
     balance
                + ", se pidió " + requested);
        this.balance = balance;
        this.requested = requested;
    }
}
```

- Un nombre que dice qué pasó.
- `extends Exception`: **checked**.
- El mensaje, una vez, con `super(...)`.
- Guarda los datos del error.
- En el archivo: getters y `serialVersionUID`.

Lanzarla antes de tocar nada

BankAccount.java

```java
public void withdraw(long amount)
        throws InsufficientFundsException {
    // ... rechaza amount <= 0 ...
    if (amount > balance) {
        throw new InsufficientFundsException(
                balance, amount);
    }
    balance -= amount;
    assert balance >= 0 : "invariante roto: "
     + balance;
}
```

- Se verifica primero y se modifica después.
- Si falla, la cuenta queda exactamente como estaba.
- Dos tipos distintos para dos problemas distintos (más en la sesión B).

Encadenar: la causa

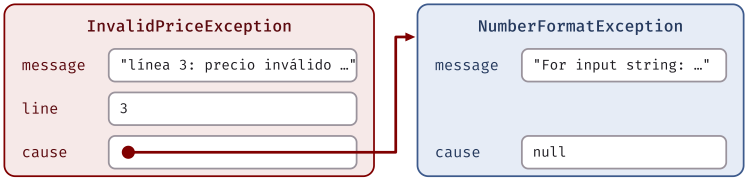
PriceReader.java

```java
for (int i = 0; i < lines.size(); i++) {
    String[] fields = lines.get(i).split(";"
     );
    String price = fields.length == 2
            ? fields[1] : "";
    try {
        prices.add(Money.parse(price));
    } catch (NumberFormatException e) {
        throw new InvalidPriceException(
                i + 1, price, e);
    }
}
```

- `Money.parse("4,90")` lanza `NumberFormatException`: no dice en qué línea.
- Se captura y se lanza una del dominio…
- …que guarda la original como causa: `super(mensaje, causa)`.

La causa en el heap y en el stack trace

HEAP



STACK TRACE IMPRESO (RECORTADO)

```
InvalidPriceException: línea 3: precio inválido "4,90"
    at PriceReader.read(PriceReader.java:16)
    at ChainDemo.main(ChainDemo.java:9)
Caused by: java.lang.NumberFormatException: For input string: "4,90"
    at java.base/java.lang.Long.parseLong(Long.java:594)
    at Money.parse(Money.java:19)
    at PriceReader.read(PriceReader.java:13)
    ... 1 more
```

Error común: capturar para no hacer nada

```java
try {
    account.withdraw(amount);
} catch (Exception e) {
}
```

- Compila. El retiro no se hizo, y nadie se enteró.
- Peor que no capturar: sin el `try`, al menos había un stack trace.
- Pariente cercano: `e.printStackTrace();` y seguir como si nada.

Regla

Capturen solo donde puedan **hacer algo**: reintentar, pedir otro dato, avisar y cancelar. Si no, déjenla pasar.

SECCIÓN 05

Checked y unchecked

05



Checked: capturar o declarar

- Si un método puede lanzar una **checked**, quien lo llama tiene que capturarla o declararla con `throws`.
- **Unchecked** (`RuntimeException`, `Error`): el compilador no exige nada.

errores/Unreported.java

```
1  BankAccount account =
2          new BankAccount("C-01");
3  account.withdraw(100);
```

bash

```
errores/Unreported.java:5: error:
    unreported exception
    InsufficientFundsException; must
    be caught or declared to be thrown
```

En C, nada impedía ignorar el `0` de `card_pay`. Aquí, el compilador no deja olvidar el caso.

¿Cuál usar?

Checked

Fallas que no son culpa del programador y que quien llama puede esperar y manejar:

- un archivo que no existe
- una red que no responde
- saldo insuficiente
- un precio mal escrito en un archivo

Unchecked

Errores de programación: se corrigen, no se capturan.

- un monto negativo como argumento
- un índice fuera del arreglo
- un `null` donde no se permite
- usar un objeto en un estado inválido

`withdraw` usa las dos: monto $\leq 0 \rightarrow$ `IllegalArgumentException`; saldo insuficiente $\rightarrow$ `InsufficientFundsException`.

`throws` y las reglas de override

errores/OverrideThrows.java

```java
class Report {
    String render() {
        return "reporte";
    }
}
class FileReport extends Report {
    @Override
    String render() throws
     IOException {
        throw new IOException("...")
     ;
    }
}
```

bash

```
errores/OverrideThrows.java:12:
    error: render() in FileReport
    cannot override render() in
    Report
  overridden method does not throw
    IOException
```

- Liskov (semana 7): quien usa un `Report` no espera una `IOException`. Se puede declarar menos, no más.
- `throws` no es parte de la firma, sino del contrato declarado.

Error común: `throws Exception` en todo

- Se declara para que el compilador “deje de molestar”.
- Quien llama ya no sabe qué puede fallar.
- Tiene que capturar `Exception`, que también captura los `NullPointerException` y demás errores de programación.

Mejor

Declarar los tipos concretos: `throws InsufficientFundsException`.

SECCIÓN 06

Liberar recursos

06



`finally`: al salir del `try`, de cualquier forma

ResourceDemo.java · caso 4

```java
try {
    IO.println("en el try");
    return;
} finally {
    IO.println("finally se ejecuta"
            + " aunque haya return");
}
```

bash

```
en el try
finally se ejecuta aunque haya return
```

- Archivos, conexiones: hay que cerrarlos.
- Si algo lanza entre abrir y cerrar, el cierre no se ejecuta.
- `finally` corre al salir del `try`: bien, con excepción o con `return`. Salvo que la JVM se detenga antes (`System.exit`).
- Con dos recursos: `try` anidados y `if (x != null)`.

`try`-with-resources

Gate.java · resumido

```java
public class Gate implements AutoCloseable {
    public void use(boolean fail) {
        IO.println("usa " + name);
        if (fail) {
            throw new IllegalStateException(
                    "falla al usar " + name)
     ;
        }
    }
    @Override
    public void close() {
        IO.println("cierra " + name);
    }
}
```

- Los recursos se declaran entre paréntesis después del `try`.
- Al salir del bloque, Java llama a `close()` en cada uno, siempre.
- Sirve cualquier `AutoCloseable`: una interfaz con un solo método.

Se cierran en orden inverso

ResourceDemo.java · caso 2

```java
try (Gate a = new Gate("A", false);
     Gate b = new Gate("B", false)) {
    a.use(false);
    b.use(true);
    IO.println("no llega aquí");
} catch (IllegalStateException e) {
    IO.println("capturada: " + e.getMessage());
}
```

bash

```
abre A
abre B
usa A
usa B
cierra B
cierra A
capturada: falla al usar
    B
```

- Se cierra primero lo que se abrió encima; todo está cerrado cuando corre el `catch`.

Paso a paso

RESOURCEDEMO.JAVA · CASO 2

```
try (Gate a = new Gate("A", false);
     Gate b = new Gate("B", false)) {
    a.use(false);
    b.use(true);
} catch (IllegalStateException e) {
    IO.println("capturada: " + e.getMessage());
}
```

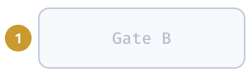
SALIDA

```
abre A          cierra B
abre B          cierra A
usa A           capturada: falla al usar B
usa B
```

EXCEPCIÓN

`IllegalStateException`
"falla al usar B"

RECURSOS ABIERTOS



se abren en orden;
se cierran al revés

4. recién entonces el catch maneja la excepción

Si `close()` también falla: suprimidas

ResourceDemo.java · caso 3

```java
try (Gate a = new Gate("A", true)) {
    a.use(true);
} catch (IllegalStateException e) {
    IO.println("capturada: "
            + e.getMessage());
    for (Throwable s : e.getSuppressed()) {
        IO.println("suprimida: "
                + s.getMessage());
    }
}
```

bash

```
abre A
usa A
cierra A
capturada: falla al usar A
suprimida: falla al cerrar A
```

- Se propaga la del bloque.
- La de `close()` viaja adjunta: `getSuppressed()`.

SECCIÓN 07

Contratos

07

El contrato de un método

Precondición

Lo que vale al entrar. Responsabilidad de quien llama.

`deposit`: `amount > 0`.

Postcondición

Lo que el método garantiza al salir.

El saldo aumentó en `amount`.

Invariante

Lo que vale antes y después de cada método público.

Saldo ≥ 0.

Un método público no sabe quién lo llama: verifica sus precondiciones y lanza una unchecked si no se cumplen.

Tres excepciones estándar para precondiciones

Excepción	Cuándo
`IllegalArgumentException`	Argumento con valor inválido: monto negativo, texto vacío.
`IllegalStateException`	El objeto no está en un estado que lo permita: stack vacío.
`NullPointerException`	Argumento `null`: `Objects.requireNonNull(x, "x")`.

BankAccount.java

```java
public BankAccount(String id) {
    this.id = Objects.requireNonNull(id, "id");   // verifica y asigna
}
```

Fail fast

Sin verificar

- `new BankAccount(null)` guarda el `null` sin quejarse.
- El `NullPointerException` aparece mucho después, en `id().length()`, en otra parte del programa.
- El stack trace no dice de dónde vino el `null`.

Verificando al entrar

- La excepción sale en el constructor.
- El stack trace apunta a la línea que pasó el `null`: el culpable.

> **Idea clave**
>
> Rechazar el dato malo **donde entra**, no tres llamadas después.

`assert`: lo que nunca debería fallar

AssertDemo.java

```java
static long[] split(long total, int parts) {
    if (parts <= 0) {
        throw new IllegalArgumentException(
                "parts debe ser positivo: "
     + parts);
    }
    long[] shares = new long[parts];
    Arrays.fill(shares, total / parts); //
     error
    assert Arrays.stream(shares).sum() ==
     total
            : "el reparto no suma " + total;
    return shares;
}
```

- `parts <= 0`: precondición. Se verifica siempre.
- Que el reparto sume el total: depende solo de este método. `assert`.
- Si falla: `AssertionError`.
- Desactivadas por defecto. Se activan con `java -ea`.

Con y sin `-ea`

bash

```
$ java -cp build AssertDemo
reparto de 1000 en 3: [333, 333, 333]
$ java -ea -cp build AssertDemo
Exception in thread "main" java.lang.AssertionError: el reparto no suma 1000
    at AssertDemo.split(AssertDemo.java:14)
    at AssertDemo.main(AssertDemo.java:19)
```

- Sin `-ea`: se reparten 999 céntimos y nadie se entera. Pruebas con assertions: `make test RUNFLAGS=-ea`.
- Error común: `assert` para validar argumentos públicos, o con efectos (`assert list.remove(x)`).

Error común: excepciones para el flujo normal

```java
int i = 0;
try {
    while (true) {
        total += prices[i++];
    }
} catch (
     ArrayIndexOutOfBoundsException e)
      {
    // se terminó el arreglo
}
```

- El final del bucle queda escondido en el `catch`.
- Más lento.
- Un error de índice de verdad dentro del bucle también quedaría escondido.
- Si se puede preguntar antes (`i < prices.length`), se pregunta.

SECCIÓN 08

Probar las excepciones

08



`assertThrows` devuelve la excepción

BankAccountTest.java

```java
@Test
void withdrawTooMuchKeepsDataAndBalance() {
    BankAccount account = new BankAccount("C-01");
    account.deposit(500);
    InsufficientFundsException e =
            assertThrows(InsufficientFundsException.class, () -> account.withdraw
     (800));
    assertEquals(500, e.balance());
    assertEquals(800, e.requested());
    assertEquals(500, account.balance());   // el saldo no cambió
}
```

- Se prueba el tipo, los datos y que el objeto no quedó a medias.
- La causa: `assertInstanceOf(NumberFormatException.class, e.getCause())`.

SECCIÓN 09

Laboratorio

09

`make test` de la semana

- Las demostraciones.
- `errores/`: tres archivos que no deben compilar.
- `Propagation crash` y `AssertDemo` con `-ea`: deben fallar.
- Nueve pruebas JUnit.

bash

```
$ make test
...
== errores/CatchOrder.java (no debe
    compilar)
...
== Propagation crash (debe terminar con
    error)
...
[         9 tests successful      ]
[         0 tests failed          ]
```

Salida recortada.

Laboratorio: manejo de errores y pruebas de excepciones

1. Leer stack traces reales, con `Caused by:`, y ubicar la línea que originó cada error.
2. Definir una jerarquía pequeña de excepciones del dominio y lanzarlas al validar.
3. Probar cada caso de error con `assertThrows`, incluidos el mensaje y los datos.
4. Una clase propia que implemente `AutoCloseable`, y una prueba de que `close()` se llama aunque el bloque lance.
5. Correr las pruebas con y sin `-ea`.

Proyecto

P2 se entrega. **P3** (Ventas) se publica: stock insuficiente, producto inexistente…

Resumen

- `throw` termina el método; la excepción **se propaga** hasta un `catch` de su tipo. Si nadie la captura, el hilo termina con un **stack trace** (con un solo hilo, el programa).
- `catch` en orden: específicos primero. La causa se ve en `Caused by:`.
- **Checked**: capturar o declarar con `throws`; para fallas esperables. **Unchecked**: errores de programación.
- Excepciones propias: nombre claro, mensaje útil, datos del error, causa.
- `try`-with-resources: cierra siempre, en orden inverso; suprimidas.
- Precondiciones al entrar (**fail fast**); `assert` para lo interno, con `-ea`.
- Nunca un `catch` vacío ni `throws Exception` por comodidad.

Para la próxima clase

- Lecturas de la semana 10: Eck, 10.1.1–10.1.3, 10.1.7, 10.5.1 y 10.5.2.
- Leer el enunciado de **P3** y decidir qué errores del dominio son checked y cuáles unchecked.

Pregunta para pensar

Un `ArrayList` guarda cualquier `Object`. ¿Qué excepción aparece si guardan un `Integer` y lo sacan como `String`? ¿Podría el compilador avisarles antes?