TRATAMIENTO DE ERRORES

CÓDIGOS DE ERROR

Ejemplo habitual de tratamiento de errores con **códigos de error** en un lenguaje como C:

```
if (deletePage(page) == E_OK) {
  if (registry.deleteReference(page.name) == E_OK) {
    if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
      logger.log("page deleted");
    } else {
      logger.log("configKey not deleted");
    }
  } else {
    logger.log("deleteReference from registry failed");
  }
} else {
  logger.log("delete failed");
  return E_ERROR;
}
```

Imanes de dependencias

Con esta técnica creamos *imanes de dependencias*, que son una mala práctica.

Ejemplo:

```
public enum Error {
  OK,
  INVALID,
  NO_SUCH,
  LOCKED,
  OUT_OF_RESOURCES,
  WAITING_FOR_EVENT;
}
```

- Los programadores intentan evitar añadir nuevos motivos de error, porque eso significa tener que volver a compilar y desplegar todo el código.

Otros imanes de dependencias: clases con nombres como *Utilidades*, *Tools*, etc.

EXCEPCIONES

Muchos lenguajes usan **excepciones** en lugar de códigos de error:

Queda más claro:

```
try {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
  logger.log(e.getMessage());
}
```

Ventaja...?

Las nuevas excepciones son derivadas de una clase base `Exception`, lo que facilita la definición de nuevos motivos de error.

¿Dónde se produce el error?

Si se eleva una excepción, ¿en cuál de las instrucciones del bloque `try` se ha producido?

Separar la función y el tratamiento de errores

Queda más fácil de comprender, modificar y depurar

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}

private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
  logger.log(e.getMessage());
}
```

Excepciones en Java

```
try {
    /* guarded region that can send
      IOException or Exception */
}
catch (IOException e) {
    /* decide what to do when an IOException
      or a sub-class of IOException occurs */
}
catch (Exception e) {
    // Treats any other exceptions
}
finally {
    // in all cases execute this
}
```

Elevar una excepción `e` $\Rightarrow$ Deshacer (*roll back*) la llamada a un método hasta encontrar un `catch` para el tipo de `e`. Si no se encuentra, se detiene el programa.

Tipos de excepciones:

- **Checked** — Derivadas de `java.lang.Throwable` (menos `RuntimeException`). Deben declararse en el método mediante `throws` y obligan al llamador a tratar la excepción.
- **Unchecked** — Derivadas de `java.lang.RuntimeException`. No se declaran en el método y no obligan al llamador a tratar la excepción.

Recomendaciones sobre excepciones

Incluir el **contexto** de la ejecución:

- Incluir información suficiente con cada excepción para determinar el motivo y la ubicación de un error
- No basta con el *stack trace*
- Escribir mensajes informativos: operación fallida y tipo de fallo

Los beneficios de las excepciones *checked* en Java son mínimos: ¿por qué?

(Hay quien recomienda usar solamente excepciones **unchecked**)

¿Por qué no usar excepciones checked?

Ejemplo: se necesita procesar un archivo CSV con datos de empleados. El código está estructurado en capas:

1. `EmployeeCSVProcessor` (mi aplicación) — Necesita lanzar `IOException` si hay errores
2. `CSVReader` (una librería de terceros) — Itera sobre las líneas del archivo
3. `EmployeeRowHandler` (mi implementación de callback) — Procesa cada fila

```java
// Librería de terceros - NO sabe ni debe saber sobre IOException
public class CSVReader {
    public void processRows(RowHandler handler) {
        List<String> lines = readFile(filePath);
        for (String line : lines) {
            handler.handle(line);  // Llama al handler
        }
    }
}

// Contrato que proporciona CSVReader
public interface RowHandler {
    void handle(String row);  // NO puede lanzar excepciones checked
}

// Mi implementación - ¡PROBLEMA!
public class EmployeeRowHandler implements RowHandler {
    @Override
    public void handle(String row) throws IOException {  // ❌ INCOMPATIBLE
        if (!isValid(row)) {
            throw new IOException("Invalid employee data");
        }
    }
}
```

El dilema:

`EmployeeCSVProcessor` (quiere `IOException`) → `CSVReader` (no declara `IOException`) → `EmployeeRowHandler` (necesita lanzarla)

⇒ ❌ Contrato violado: ¡no compila!

Cómo afectan al diseño las excepciones checked

- Se paga el precio de violar el principio OCP (*Open-Closed Principle*): si lanzamos una excepción *checked* desde un método y el `catch` está tres niveles por encima, hay que declarar la excepción en la signatura de todos los métodos que van entre medias.
- Esto significa que un cambio en un nivel bajo del software puede forzar cambios en niveles altos.
- Imaginemos cuando entre medias hay una biblioteca de terceros que no podemos modificar

Transformación de excepciones

Muchas APIs de Java lanzan excepciones *checked* cuando deberían ser *unchecked*

Ejemplo: Al ejecutar una consulta mediante `executeQuery` en el API de JDBC se lanza una excepción `java.sql.SQLException` (de tipo checked) si la SQL es errónea.

- ¿Le interesa al cliente del API saber que el error es provocado por una sentencia SQL?
- ¿Le interesa al cliente del API conocer el tipo de excepción *checked* que una consulta puede generar?

Solución: ¿transformación en unchecked?

Transformar las excepciones checked en unchecked:

```
try {
  // Codigo que genera la excepcion checked
} catch (Exception ex) {
  throw new RuntimeException("Unchecked exception", ex)
}
```

La solución es un code smell:

```
public class EmployeeRowHandler implements RowHandler {
  @Override
  public void handle(String row) {
    if (!isValid(row)) {
      throw new RuntimeException("Invalid employee data: " + row);
    }
  }
}

public class EmployeeCSVProcessor {
  public void process(String filePath) throws IOException {
    try {
      reader.processRows(new EmployeeRowHandler());
    } catch (RuntimeException e) {
      // ¿Era una IOException o un error real?
      if (e.getMessage().contains("Invalid")) {
        throw new IOException(e);  // Reconvertir
      }
      throw e;  // Relanzar si era otra cosa
    }
  }
}
```

Precio a pagar:

- Pérdida de información: No se sabe si la `RuntimeException` es realmente la excepción esperada
- Violación del tipo: La excepción no comunica claramente el error

Transformación a UncheckedIOException

Java 8 introdujo `UncheckedIOException`, para transformar una `IOException` en unchecked sin perder la información de la causa original

```
import java.io.UncheckedIOException;

public class EmployeeRowHandler implements RowHandler {
    @Override
    public void handle(String row) {
        try {
            if (!isValid(row)) {
                throw new IOException("Invalid employee data");
            }
            // Lógica principal
        } catch (IOException e) {
            // "Envolvemos" el error checked en un wrapper unchecked oficial de Java
            throw new UncheckedIOException(e);
        }
    }
}
```

Excepciones en otros lenguajes

- C#, C++, Python o Ruby no ofrecen excepciones *checked*.
- Scala no usa excepciones *checked* como Java: Scala exception handling

Excepciones encapsuladas

Criticar la siguiente implementación:

```
ACMEPort port = new ACMEPort(12);
try {
  port.open();
} catch (DeviceResponseException e) {
  reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
  reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
  reportPortError(e);
  logger.log("Device response exception");
} finally {
  ...
}
```

Código duplicado: llamada a `reportPortError()` se repite mucho. ¿Cómo evitarlo?

Solución: Excepción encapsulada

```
public class LocalPort {
  private ACMEPort innerPort;
  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }
  public void open() throws PortDeviceFailure {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {
      throw new PortDeviceFailure(e);
    }
  }
  ...
}
```

Sustituir ahora por...

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportPortError(e);
  logger.log(e.getMessage(), e);
} finally {
  ...
}
```

- La encapsulación de excepciones es recomendable cuando se usa un API de terceros, para minimizar las dependencias con respecto al API elegido.
- También facilita la implementación de **mocks** del componente que proporciona el API para construir pruebas.

Las excepciones son excepcionales

- **Recomendación de uso**: Usar excepciones para problemas excepcionales (eventos inesperados)

Ejemplo: excepciones en el tratamiento de ficheros: ¿Usar excepciones cuando se intenta abrir un fichero para leer y el fichero no existe? Depende de si el fichero debe estar ahí

- Caso en que se debe lanzar una excepción:

```
public void open_passwd() throws FileNotFoundException {
  // This may throw FileNotFoundException...
  ipstream = new FileInputStream("/etc/passwd");
  // ...
}
```

- Caso en que no se debe lanzar:

```
public boolean open_user_file(String name)
    throws FileNotFoundException {
  File f = new File(name);
  if (!f.exists())
    return false;
  ipstream = new FileInputStream(f);
  return true;
}
```

ABUSO DE NULL

Obtener un *null* cuando no se espera puede ser un quebradero de cabeza para el tratamiento de errores.

Principio general: no devolver null

Este código puede parecer inofensivo, pero es maligno: ¿Qué pasa si `persistentStore` es null?

```
public void registerItem(Item item) {
  if (item != null) {
    ItemRegistry registry = persistentStore.getItemRegistry();
    if (registry != null) {
      Item existing = registry.getItem(item.getID());
      if (existing.getBillingPeriod().hasRetailOwner()) {
        existing.register(item);
      }
    }
  }
}
```

- Peligro de `NullPointerException`
- ¿Se nos ha olvidado añadir un `if null`?
- El problema no es que se haya olvidado uno, sino que hay demasiados
- En su lugar, elevar una excepción o devolver un objeto *especial*

No devolver null

Evitar esto:

```
List<Employee> employees =
                    getEmployees();

if (employees != null) {
  for(Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

Mejor así:

```
List<Employee> employees =
                      getEmployees();

for (Employee e: employees) {
  totalPay += e.getPay();
}

public List<Employee> getEmployees() {
  if( /* there are no employees */ )
    return Collections.emptyList();
}
```

No pasar valores null

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x - p1.x) * 1.5;
  }
}
```

¿Qué sucede si llamamos a `xProjection()` así...?

```
calculator.xProjection(null, new Point(12, 13))
```

Devolver null es malo, pero ¡pasar un valor null es peor!

¿Es mejor así...?

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    if (p1 == null || p2 == null)
      throw InvalidArgumentException(
                "Invalid argument for MetricsCalculator.xProjection");
    return (p2.x - p1.x) * 1.5;
  }
}
```

¿Qué acción realizar ante un `InvalidArgumentException`? ¿Hay alguna buena?

Alternativa con aserciones

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    assert p1 != null : "p1 should not be null";
    assert p2 != null : "p2 should not be null";
    return (p2.x - p1.x) * 1.5;
  }
}
```

El uso de `assert` es una buena forma de documentar, pero no resuelve el problema.

Pueden usarse **aserciones** o **contratos** para resolver esto.

OPTIONALS

- En la mayoría de lenguajes no hay una forma satisfactoria de tratar con *nulls* pasados como argumento accidentalmente.
- Para eso están los **options** u **optionals**, disponibles actualmente en muchos lenguajes como:
  - Scala `Option`
  - Java 8 `java.util.Optional`
  - C++17 `std::optional`
- TypeScript recomienda usar `undefined` (algo que no se ha inicializado) en lugar de `null` (algo que no está disponible)

Scala `Option`

En Scala, `Option[T]` es un contenedor de un valor opcional de tipo T.

- Si el valor de tipo T está presente, `Option[T]` es una intancia de `Some[T]` que contiene el valor presente de tipo T.
- Si el valor está ausente, `Option[T]` es el objeto `None`.

```
object Demo {
   def main(args: Array[String]) {
      val a: Option[Int] = Some(5)
      val b: Option[Int] = None

      println("a.isEmpty: " + a.isEmpty )  //false
      println("b.isEmpty: " + b.isEmpty )  //true
   }
}
```

Valores vacíos en Scala

Diferencias entre `Null`, `null`, `Nil`, `Nothing`, `None` y `Unit` en Scala:

- `null` es como el de Java
- `Null` es un trait, subconjunto de todos los tipos-referencia, cuya única instancia es `null`
- `Nothing` es un trait sin instancias; sirve para especificar el tipo de retorno en métodos que siempre elevan una excepción
- `Unit` es análogo a `void` en Java
- `Nil` es una lista con cero elementos (su tipo es `List[Nothing]`)
- `None` es uno de los hijos de `Option`

```
object Demo {
   def main(args: Array[String]) {
      val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")

      println("show(capitals.get( \"Japan\")) : " +
        show(capitals.get( "Japan")) )
      println("show(capitals.get( \"India\")) : " +
        show(capitals.get( "India")) )
   }

   def show(x: Option[String]) = x match {
      case Some(s) => s
      case None => "?"
   }
}
```

Java 8 `Optional`

Lecturas recomendadas

- Leer Java 8 Optional in Depth.
- Leer Jugando con Optional en Java 8.

Ejemplo en Java 8 con sintaxis imperativa

```
private static Optional<Double> getDurationOfAlbumWithName(String name) {
    Album album;
    Optional<Album> albumOptional = getAlbum(name);
    if (albumOptional.isPresent()) { // albumOptional == null
        album = albumOptional.get();
        Optional<List<Track>> tracksOptional = getAlbumTracks(album.getName());
        double duration = 0;
        if (tracksOptional.isPresent()) { // tracksOptional == null
            List<Track> tracks = tracksOptional.get();
            for (Track track : tracks) {
                duration += track.getDuration();
            }
            return Optional.of(duration);
        } else {
            return Optional.empty();
        }
    } else {
        return Optional.empty();
    }
}
```

Al ejecutar varias operaciones seguidas que pueden devolver null, el nivel de anidamiento del código aumenta y queda menos claro (se mezcla código funcional con código de gestión de errores). Solución...

Ejemplo en Java 8 con sintaxis *fluent*

```
Optional<Double> getDurationOfAlbumWithName(String name) {
    Optional<Double> duration = getAlbum(name)
            .flatMap((album) -> getAlbumTracks(album.getName()))
            .map((tracks) -> getTracksDuration(tracks));
    return duration;
}
```

La función `map` comprueba si el `Optional` que recibe está vacío. Si lo está devuelve un `Optional` vacío y, si no, aplica la función anónima que le hemos pasado por parámetro, pasándole el valor del `Optional` (es decir, si el `Optional` está vacío, el método `map` no hace nada).
Esto sirve para concatenar operaciones sin necesidad de comprobar en cada momento si el `Optional` está vacío.

Cuando queremos encadenar distintas operaciones que devuelvan `Optional`, es necesario usar `flatMap`, ya que si no acabaríamos teniendo un `Optional<Optional<Double>>`.

Pero... `getDurationOfAlbumWithName()` devuelve un `Optional<Double>`. ¿No debería mejor devolver un `double`?

```
private static double getDurationOfAlbumWithName(String name) {
    return getAlbum(name)
            .flatMap((album) -> getAlbumTracks(album.getName()))
            .map((tracks) -> getTracksDuration(tracks))
            .orElse(0.0);
}
```

Podríamos seguir devolviendo `Optional` por toda la aplicación, pero en algún momento tenemos que decidir qué hacer en caso de que el valor que queremos no estuviera presente.

Para ello se usa `orElse()` para proporcionar un valor alternativo en caso de que el valor no estuviera presente.

Ejemplo: `record` en vez de clases

El compilador genera automáticamente constructor, accessores (sin `get`), `equals`, `hashCode` y `toString`:

```
record ScreenResolution(int width, int height) {}

record DisplayFeatures(String size, ScreenResolution resolution) {}

record Mobile(long id, String brand, String name,
              DisplayFeatures displayFeatures) {}
```

Ejemplo: Programa de prueba

Con `var` se infiere el tipo de cada variable local:

```
public class MobileTester {
  public static void main(String[] args) {
    var resolution1 = new ScreenResolution(750, 1334);
    var dfeatures1  = new DisplayFeatures("4.7", resolution1);
    var mobile1     = new Mobile(2015001, "Apple", "iPhone 6s", dfeatures1);

    var mService = new MobileService();
    int mobileWidth = mService.getMobileScreenWidth(mobile1);
    System.out.println("Apple iPhone 6s Screen Width = " + mobileWidth);

    var resolution2 = new ScreenResolution(0, 0);
    var dfeatures2  = new DisplayFeatures("0", resolution2);
    var mobile2     = new Mobile(2015001, "Apple", "iPhone 6s", dfeatures2);
    int mobileWidth2 = mService.getMobileScreenWidth(mobile2);
    System.out.println("Apple iPhone 16s Screen Width = " + mobileWidth2);
  }
}
```

Reducir *boilerplate* de `if (x!_null)`: expresión `switch` con *record patterns* y *unnamed patterns* `_`:

```
public class MobileService {
  public int getMobileScreenWidth(Mobile mobile) {
    return switch (mobile) {
      case null                                                -> 0;
      case Mobile(_, _, _, null)                               -> 0;
      case Mobile(_, _, _, DisplayFeatures(_, null))           -> 0;
      case Mobile(_, _, _, DisplayFeatures(_, ScreenResolution(int w, _))) -> w;
    };
  }
}
```

Novedades de Java usadas en el ejemplo

Característica	JDK	Qué aporta
`record`	16	Clase inmutable en una línea; accessors sin prefijo `get`
Pattern matching en `switch`**	21	Elimina `if (x != null)` anidados
Record patterns	21	Deconstrucción en los `case`
Unnamed patterns `_`	22	Ignora campos del `record` irrelevantes
`var`	10	Inferencia de tipo local; reduce repetición de tipos

Ejemplo con `Optionals`

Con `var` se infiere el tipo de cada variable local:

```
public class MobileTesterWithOptional {
  public static void main(String[] args) {
    var resolution = new ScreenResolution(750, 1334);
    var dfeatures  = new DisplayFeatures("4.7", Optional.of(resolution));
    var mobile     = new Mobile(2015001, "Apple", "iPhone 13", Optional.of(dfeatures));
    var mService   = new MobileService();

    int width = mService.getMobileScreenWidth(Optional.of(mobile));
    System.out.println("Apple iPhone 13 Screen Width = " + width);

    var mobile2 = new Mobile(2015001, "Apple", "iPhone 13", Optional.empty());
    int width2  = mService.getMobileScreenWidth(Optional.of(mobile2));
    System.out.println("Apple iPhone 13 Screen Width = " + width2);
  }
}
```

```java
record ScreenResolution(int width, int height) {}

record DisplayFeatures(String size, Optional<ScreenResolution> resolution) {}

record Mobile(long id, String brand, String name,
              Optional<DisplayFeatures> displayFeatures) {}
```

Menos *boilerplate* - Sintaxis *fluent* con `flatMap` - Elimina el problema de los nulos:

```java
public class MobileService {
  public int getMobileScreenWidth(Optional<Mobile> mobile) {
    return mobile.flatMap(Mobile::displayFeatures)
      .flatMap(DisplayFeatures::resolution)
      .map(ScreenResolution::width)
      .orElse(0);
  }
}
```

Carencias de Optional

- El tratamiento de errores clásico del lenguaje C (con el que empezábamos este tema) se basa en devolver un **valor especial** que contiene un *código de error* (normalmente negativo) que indica qué ha salido mal.
  Se podrían representar tantos motivos de error como posibles valores devueltos.
- Los `Optional` no ofrecen la posibilidad de que decir qué es lo que ha salido mal (en caso de que no haya valor a devolver).
- Por tanto, no son apropiados para métodos en los que pueden salir varias cosas mal y no solo una (que no exista un valor a devolver)
- Lenguajes como Scala proponen alternativas como `Either` y `Validation`.

Lecturas para ampliación: clase Validation en Scala

Ejemplo: División por cero

```
object EitherLeftRightExample extends App {

  def divideXByY(x: Int, y: Int): Either[String, Int] = {
      if (y == 0) Left("Can't divide by 0")
      else Right(x / y)
  }

  println(divideXByY(1, 0))
  println(divideXByY(1, 1))
  divideXByY(1, 0) match {
      case Left(s) => println("Answer: " + s)
      case Right(i) => println("Answer: " + i)
  }

}
```

Nota: No hay implementaciones de `Either` en el JDK, pero sí en extensiones funcionales a Java (v.g. functionaljλvλ)