

Intro a Manejo de errores con excepciones

Por:
Fernando Dodino
Con aportes de Matías Potel Feola
y Pablo Tesone

Versión 1.3
Marzo 2014

Indice de contenidos

[1 Errores y excepciones](#)

[1.1 Primera solución: Call & return](#)

[1.2 Segunda solución con excepciones](#)

[2 Jerarquía de Excepciones](#)

[3 ¿Cómo generamos una excepción?](#)

[4 ¿Qué sucede cuando un objeto recibe una excepción?](#)

[4.1 No la trata](#)

[4.2 La envuelve en una excepción de más alto nivel](#)

[4.3 Tratar la excepción](#)

[4.4 Finally](#)

[5 Excepciones de negocio y de programa](#)

[6 Control adicional de excepciones \(únicamente para Java\)](#)

[6.1 Excepciones chequeadas](#)

[6.2 Cuándo utilizar excepciones chequeadas](#)

[7 Buenas y malas prácticas](#)

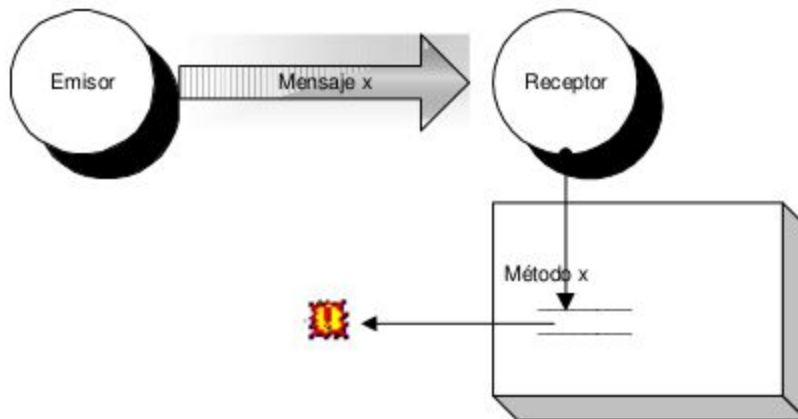
[8 Bibliografía complementaria](#)

1 Errores y excepciones


Recordemos la definición de sistema dentro del paradigma de objetos

Sistema = conjunto de objetos que colaboran para un objetivo común.

Entonces precisamos que haya comunicación entre los objetos. Cuando un objeto le envía un mensaje a otro,

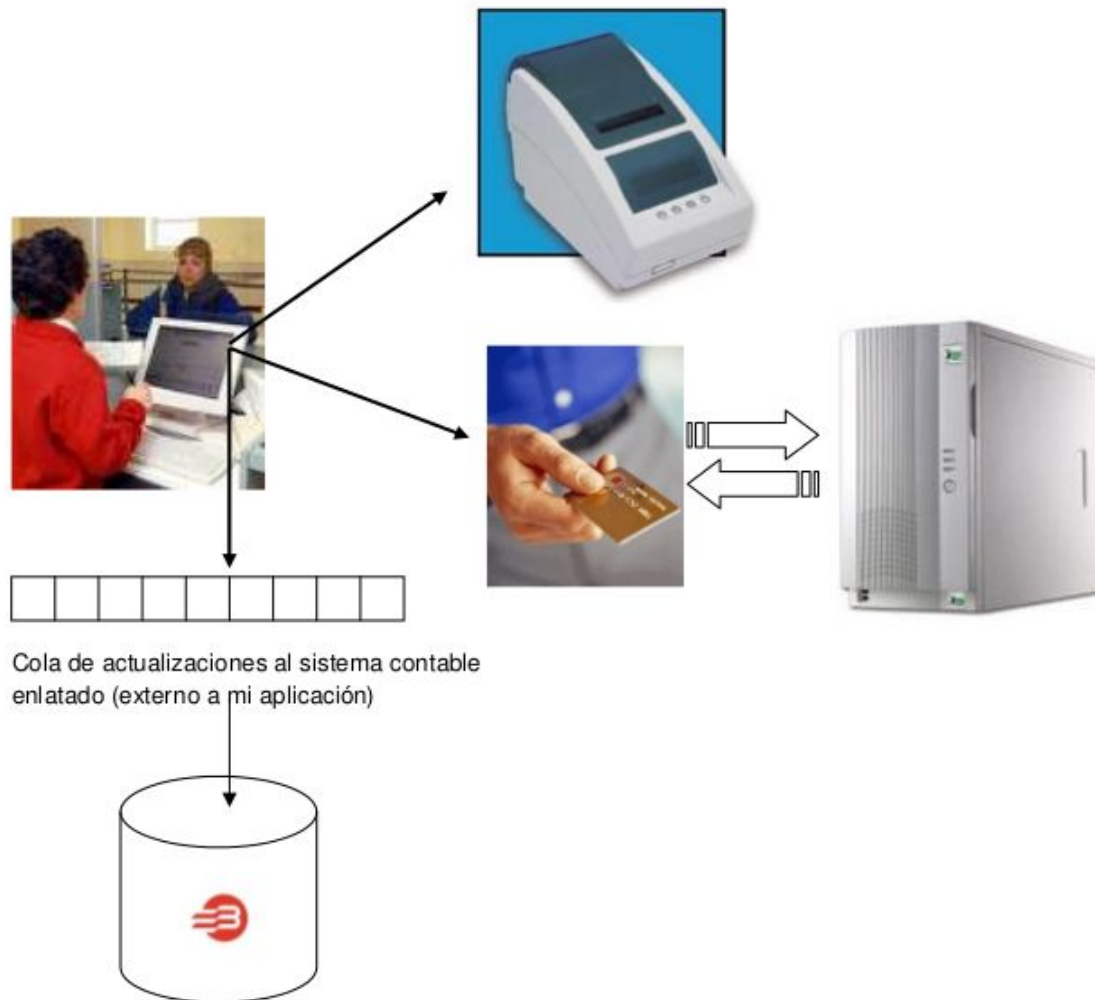


el emisor espera que el receptor pueda realizar esa tarea. Ahora ¿qué sucede si falla algo en el método x?

 Una *excepción* es un evento que altera el flujo normal de mensajes entre objetos.

Muchos tipos de errores pueden causar excepciones, desde errores serios de hardware como la ruptura del disco rígido hasta errores simples de programación como tratar de acceder a un elemento fuera de los límites de un array.

Breve ejemplo



Supongamos que existe una funcionalidad “Pago con Tarjetas de Crédito” para el módulo de Cobranzas de una empresa, que permite a un cajero (humano) realizar una serie de operaciones, tales como imprimir la factura de lo que se está cobrando, enviar información al sistema de Tarjetas de Crédito (que autoriza la operación para dicha tarjeta), enviar el pago a un sistema de Contabilidad enlatado y finalmente actualizar el saldo del cliente en nuestra aplicación descontando de la deuda la factura que acaba de pagar. Nos quedaría algo así:

```
cobrar {  
    imprimir factura en controladora fiscal  
    enviar info tarjeta de credito  
    informar pago a Contabilidad  
    actualizar saldo del cliente  
}
```

A primera vista este método parece bastante simple, pero ignora todos estos errores potenciales:

- ¿Qué pasa si la controladora fiscal de la impresora devuelve error (por ejemplo, por trabarse el papel)?
- ¿Qué pasa si el host de la tarjeta de crédito está caído?
- ¿Qué pasa si el sistema contable está caído y no atiende los pedidos de nuestro sistema?

Sin contar el error posible dentro de nuestra aplicación al querer actualizar el saldo del cliente (por ejemplo, por algún problema de concurrencia).

1.1 Primera solución: Call & return

Para resolver esas cuestiones, se debe agregar bastante código para detectar esos errores, manejarlos e informarlos a quien nos haya invocado. El código pasaría a verse de esta manera:

```
def tipoCodigoError cobrar {
  codigoError = OK
  imprimir factura en controladora fiscal
  if (sePudoImprimir) {
    enviar info a tarjeta de credito
    if (sePudoEnviarInfoATarjeta) {
      informar pago a Contabilidad
      if (sePudoInformarContabilidad) {
        actualizar saldo del cliente
        if (noSePudoActualizarSaldo) {
          codigoError = ERROR_ACTUALIZAR_SALDO
        }
      }
    }
    else {
      codigoError = ERROR_SISTEMA_CONTABLE
    }
  }
  else {
    codigoError = ERROR_TARJETA_CREDITO
  }
}
else {
  codigoError = ERROR_IMPRESION_FACTURA_FISCAL
}
return codigoError
}
```

Con este mecanismo de detección de errores, las 4 líneas originales se convirtieron en 24, y peor aún, la lógica original se perdió entre los múltiples ifs de forma que es bastante difícil entender la lógica de la aplicación o asegurar que funciona correctamente. Muchas veces estos problemas simplemente se ignoran y los errores se “reportan” cuando el sistema cancela.¹

1.2 Segunda solución con excepciones

Las excepciones permiten manejar los casos que salen del flujo principal de envío de mensajes dentro del método. Entonces el código pasa a ser:

```

cobrar {
  Lógica de negocio (flujo normal) { try {
    imprimir factura en controladora fiscal;
    enviar info tarjeta de credito;
    informar pago a Contabilidad;
    actualizar saldo del cliente;
  }
  Manejo de errores (por separado) {
    catch (noSePudoImprimir) {
      hacer algo;
    }
    catch (noSePudoInformarTarjetaCredito) {
      hacer algo;
    }
    catch (noSePudoInformarContabilidad) {
      hacer algo;
    }
    catch (noSePudoActualizarSaldo) {
      hacer algo;
    }
  }
}

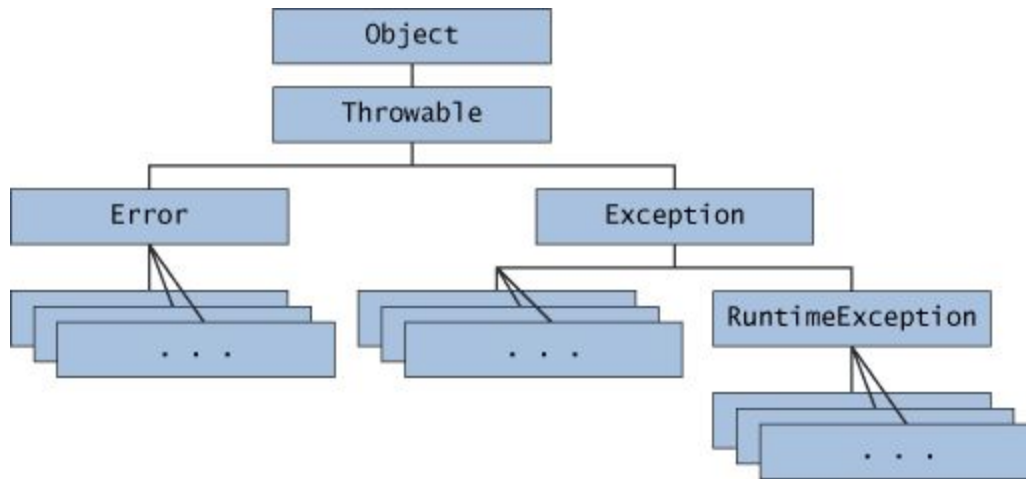
```

Las excepciones no evitan el esfuerzo de detectar, reportar y manejar los errores, pero proveen las herramientas para realizar estas tareas fuera de la lógica principal y generalmente la cantidad de código necesario es menor.

Todo el bloque de código encerrado en el try permite “atrapar” (catch) las diferentes excepciones que puedan surgir. A continuación veremos que las excepciones se subclasifican según un orden jerárquico.

¹ La justificación de algunos seguidores de esta línea es “no penalicemos costo de programación evitando errores que jamás ocurrirán”. Si bien a priori podemos decir que es una idea descabellada penalizar entonces al usuario y exponer a que el sistema se considere poco fiable, es interesante rescatar la idea como contrapartida de la “programación defensiva”.

2 Jerarquía de Excepciones



Throwable es la clase madre de la cual heredan:

- **Error:** ninguna aplicación debería manejarlo, sólo aplica a fallas graves de la Máquina Virtual.
- **Exception:** volviendo a la definición de la página 1, una Exception es el evento genérico que altera el flujo normal de mensajes de la aplicación. Podemos contar entre las subclases a [IOException](#), [SQLException](#), [TimeoutException](#), [ClassNotFoundException](#).
- **RuntimeException:** es un caso particular que redefine el comportamiento estándar de Exception para que sean no chequeadas (en un rato veremos la diferencia entre excepciones chequeadas y no chequeadas). Ejemplos que subclasean a RuntimeException son: [ClassCastException](#), [EmptyStackException](#), [IllegalArgumentException](#), y por supuesto, nuestra querida [NullPointerException](#).

3 ¿Cómo generamos una excepción?

Modelemos la compra de un cliente donde si el cliente tiene saldo suficiente para pagar, realice la compra, en caso contrario tengo que rechazarla:

```

class Cliente {
    var saldo = 102
    ...
    def comprar(int monto) {
        if (saldo < monto) {

```

² Por cuestiones didácticas se prefirió el uso de int en lugar del BigDecimal que sería más apropiado.

```

        throw new SinUnMangoException("Saldo  insuficiente")
    }
    ...
}
...
}

```

Para poder modelar las exceptions lo que tenemos que hacer es crear una clase que extienda de Exception o RuntimeException. En el caso de usar XTend cualquiera de los dos casos se comportará de la misma manera.

```

class SinUnMangoException extends Exception {
    new(String msg) {
        super(msg)
    }
}

```

Es una buena práctica que la excepción venga acompañada de un mensaje de error representativo, que no sólo describa la situación actual, sino también cómo solucionarlo, siempre que sea posible. Entonces en lugar de tener un constructor vacío para SinUnMangoException, vamos a pasar un mensaje que nos ayude. Veamos algunas opciones:

- “Error 808012” => si no tengo saldo suficiente, no hay por qué “esconder” el motivo detrás de un código mágico
- “No puede comprar” => no indica por qué no puedo comprar, no me da mucha información
- “Saldo insuficiente” => esto va mejorando, aún así tengo que conocer el saldo para intentar otra compra
- “El monto de la compra (\$ “ + monto + “) debe ser inferior al saldo que ud.tiene (\$ “ + saldo + “)” => este mensaje de error me permite entender claramente cuál es el problema y me ayuda a resolverlo.

4 ¿Qué sucede cuando un objeto recibe una excepción?

4.1 No la trata

1. Podemos elegir no hacer nada y seguir trabajando como hasta ahora:

```

Cliente chiara = new Cliente
chiara.comprar(25)

```

El resultado es que el usuario recibe la notificación de la excepción junto con la pila de

mensajes que se fueron enviando cada uno de los objetos (Stack Trace):

```
Exception in thread "main" ar.edu.clientes.SinUnMangoException:  
Saldo  insuficiente  
    at ar.edu.clientes.Cliente.comprar(Cliente.java:14)  
    at ar.edu.clientes.Prueba.main(Prueba.java:9)
```

Cuando el error es insalvable para el dominio, es una buena práctica dejarlo pasar hacia otros objetos que puedan hacer algo mejor (por ejemplo, una interfaz de usuario puede capturar el mensaje de error y mostrarle un cartel al usuario). No es una buena práctica tratar de atrapar el error cuando no es responsabilidad de dicho objeto hacerlo.

4.2 La envuelve en una excepción de más alto nivel

Una ligera variante puede ser:

```
static def main(String[] args) {  
    try {  
        var chiara = new Cliente  
        chiara.comprar(25)  
    }  
    catch (SinUnMangoException sme) {  
        throw new ClienteSinSaldoException(sme3)  
    }  
}
```

La idea es capturar una excepción específica del objeto al que nosotros conocemos, y propagar **otro** error más representativo, de más alto nivel⁴. Esta estrategia se aplica especialmente cuando podemos dar un mejor nivel de información a quien recibe la excepción, en lugar de recibir el mensaje de error estándar poco representativo:

```
try {  
    var cliente = conn.buscarCliente(nombre) as Cliente  
    ... enviarle mensajes al cliente ...  
} catch (ClassCastException ccEx) {  
    throw new ConsultaInvalidaException(ccEx)  
}  
catch (SQLException sqlEx) {  
    throw new SinConexionDeDatosException(sqlEx)  
}
```

³ Vemos que al crear la excepción, podemos pasarle una referencia a la excepción original ("wrappeándola").

⁴ Tengamos en cuenta que el cliente de nuestro código puede ser tanto el usuario final como otro desarrollador.

4.3 Tratar la excepción

Puedo tratar la excepción y actuar en consecuencia. Por ejemplo, podríamos tener un servidor alternativo al cual avisar el pago de nuestra factura:

```
def informarPagoSistemaContable(Pago pago) {  
    var conexion = null  
    try {  
        conexion = openConnection(servidorPrincipal)  
        ...  
    }  
    catch (ConnectionException exception) {  
        conexion = openConnection(servidoAlternativo)  
    }  
    ...  
}
```

En este caso el objeto está capacitado para manejarlo, y en consecuencia aplica el try/catch correspondiente.

4.4 Finally

```
abrir archivo  
try {  
    hacer algo  
} catch (AnyKindOfException ex) {  
    manejar exception  
} finally {  
    cerrar archivo  
}
```

El bloque de código encerrado en el **finally** siempre se ejecuta (se haya producido la excepción o no): resulta útil cuando tomamos algún recurso y tenemos que asegurarnos que se libere (archivos, conexiones, transacciones entre objetos, etc.)

Debemos tener en cuenta que el bloque de código finally puede fallar, con lo cual hay que tener cuidado con la estrategia de manejo de errores que adoptemos al definirlo.

5 Excepciones de negocio y de programa

En base al cliente que va a recibir la excepción, podemos clasificar a las excepciones en dos tipos diferentes:

- **Excepciones de negocio o de dominio:** ocurren en el uso de la aplicación y

son entendibles para el usuario final (“no hay saldo en la cuenta corriente”, “no hay stock del producto a facturar”, “no hay precio del producto a facturar”, etc.)

- **Excepciones de programa:** se producen cuando se ejecuta código de la aplicación y las puede analizar un especialista técnico (“falló el acceso a la base de datos”, “hubo división por cero”, “no pude castear este objeto a este tipo”, “el objeto no entiende este mensaje (NullPointerException)”, etc.)

La naturaleza de ambos tipos de excepción son diferentes: en general las excepciones de negocio (o de aplicación) requieren que el usuario corrija la información que quiere ingresar al sistema (y valore el producto, o bien seleccione un producto alternativo para facturar, o trate de sacar plata de otra cuenta bancaria), en tanto que las excepciones de programa requieren una corrección por parte de un usuario técnico (que chequeará la conexión a la base de datos o bien corregirá el código que originó el error).

Por lo tanto las acciones a tomar cuando armamos cada tipo de excepción son diferentes: en las excepciones de negocio intentamos que el usuario vea una pantalla amigable donde le mostramos el problema que hubo al tratar de completar una acción con un mensaje representativo (e incluso proponiéndole soluciones alternativas para que la tarea se realice), mientras que en las excepciones de programa también mostramos una pantalla amigable al usuario, pero reservamos todos los detalles internos al desarrollador. De esa manera, las excepciones terminan siendo una herramienta más que ayuda a que nuestra aplicación se vuelva más robusta y confiable.

Entonces para el componente de UI (interfaz de usuario) quizás sea más conveniente no tener que atrapar distintas excepciones, sino diferenciar las que son de negocio y las que no. En el ejemplo de la compra:

```
def comprar(int monto) {  
    if (saldo < monto) {  
        throw new UserException("Debe comprar por menos de $ " +  
            saldo)  
    }  
    ...  
}
```

Mientras que en la interfaz de usuario el código si tenemos una tecnología similar a la que venimos trabajando en el dominio, podremos hacer:

```
def void comprar() {  
    try {  
        <<cliente>>.comprar(<<monto>>)  
    } catch (UserException e) {  
        showWarning(e.message)  
    } catch (Exception e) {  
        ...  
    }  
}
```

```
        e.printStackTrace()
        showError("Ocurrió un error en la aplicación. Consulte al
administrador")
    }
}
```

6 Control adicional de excepciones (únicamente para Java)

6.1 Excepciones chequeadas

El lenguaje Java diferencia dos tipos de excepciones: las chequeadas y las no chequeadas. Si la clase `SinUnMangoException` hereda de `Exception`

```
public class SinUnMangoException extends Exception {
    ...
}
```

el método `comprar` definido en `Cliente` no compila, ya que el compilador exige que declaremos en el método la excepción que vamos a tirar:

```
public void comprar(int monto) throws SinUnMangoException {
    ...
}
```

Entonces `SinUnMangoException` es una **excepción chequeada**.

Esto no es obligatorio si `SinUnMangoException` hereda directamente de `RuntimeException` (la cláusula `throws` es opcional en estos casos).

```
public class SinUnMangoException extends RuntimeException {
    ...
}
```

En este caso decimos que `SinUnMangoException` es un **excepción no-chequeada**.

Esta diferencia la vamos a tener sólo si trabajamos con Java, mientras que si usamos XTend como lenguaje todas las excepciones se van a comportar como excepciones no chequeadas.

6.2 Cuándo utilizar excepciones chequeadas

Hay dos corrientes bien marcadas a la hora de utilizar excepciones chequeadas o no-chequeadas.

A favor de la primera podemos decir:

- Las excepciones chequeadas quedan documentadas (tanto por las herramientas estándares como javadoc como por el código mismo, donde estoy obligado a decir tal método **throws** tal exception)
- Como el compilador me obliga a trabajarlo con un try/catch me aseguro de que no rompa en tiempo de ejecución

De hecho, se acerca bastante a la postura oficial de Sun, cuya recomendación es: “si un cliente puede recuperarse de dicha excepción, debería ser chequeada. Por el contrario, si el cliente no puede hacer nada al respecto (como al enviar un mensaje desde una referencia a un null), la excepción debería no ser chequeada”.

Por otro lado permitámonos disentir y decir que:

- El hecho de que el compilador me obligue a definir un bloque **catch** es una opción tentadora para hacer cosas como:

```
try {
    hacer algo
}
catch (AnyKindOfException ex) {
    // Ojota!!! no hay que olvidarse de hacer xxxx
}

o

try {
    hacer algo
}
catch (AnyKindOfException ex) {
    System.out.println("Todo anduvo mal!")
    <- ¡nadie va a leer esto!
}
```

Lo que termina siendo más nocivo aún, pues el cliente nunca se entera de que hay algo en la aplicación que anda mal. ⁵

⁵ Una alternativa a esto es configurar el Eclipse para que en cada catch, la wrappee en una RuntimeException, y después modificarlo por un manejo más adecuado

windows -> preferences -> Java -> Code Style -> Code templates -> code

```
catch block body
throw new RuntimeException("${exception_var});
```

```
method body
throw new UnsupportedOperationException("under construction");
```

- La excepción chequeada rompe con la firma de un método

```
public void comprar(int monto) throws SinUnMangoException y  
public void comprar(int monto)
```

son métodos que tienen firmas diferentes (si tengo dos objetos que implementan un método comprar y uno lanza una excepción chequeada, se rompe el polimorfismo).

La mayoría de los lenguajes más modernos que Java se inclinan a usar únicamente excepciones no chequeadas, descartando esta diferenciación.

7 Buenas y malas prácticas

Ya sabemos que no hay recetas, pero sí libros de auto-ayuda, aquí van algunas palabras que pueden servir de puntapié para debates a la hora de trabajar con excepciones:

- Sólo las atrapa el que las sabe tratar.
- Cuidarnos de la “programación paranoica”: demasiadas protecciones también atentan contra la claridad de código (¿qué pasa si me quedo sin memoria? ¿si se rompe el disco rígido? ¿si se quema el edificio y no tengo la llave para salir?). Las excepciones debieran tratarse en pocos lugares.
- Utilizar mensajes claros en las excepciones. El tiempo en elegir el nombre y el mensaje que va a aparecer hace ahorrar tiempo a todo el equipo de trabajo.
- *Corolario del consejo anterior:* Siempre que te encontrás con un mensaje de error que no está claro y tenés que buscar la causa, lo primero que hay que hacer cuando la encontrás es arreglar el mensaje de error y poner uno bien claro, porque si no después se pierde la oportunidad de corregirlo y alguien más va a perder ese tiempo. Entonces sí después se corrige el error.
- No deben ser utilizadas para tomar decisiones de negocio. Ejemplo:

```
def CondicionIVA getCondicionIVA() {  
    if (monotributista) {  
        throw new ClienteMonotributistaException  
    } else {  
        ...  
    }  
}  
  
def metodo() {  
    try {  
        ...  
        condicionIVA = cliente.condicionIVA  
    } catch (ClienteMonotributistaException e) {
```

```
        ...  
    }  
}
```

Representamos como excepciones las condiciones que salen del flujo normal del negocio.

- No hacer catch o throw de un Error (no tiene sentido).
- No debemos nunca atrapar una excepción (“cachearla” por así decirlo) y no hacer nada. No hacer nada incluye hacer `e.printStackTrace()` o cosa similar, donde el error se escribe en un archivo de log que difícilmente se lea, pero que además tiene un problema mucho más grave: **no avisa del error al componente llamador, con lo cual es probable que este error propague errores posteriores en otros componentes de la aplicación y estemos buscando el error en otro lugar durante un buen tiempo.**
- Este ejemplo muestra cómo el uso de excepciones se desaprovecha para trabajar con valores de retorno:

```
def int metodo() {  
    try {  
        var chiara = new Cliente  
        chiara.comprar(25)  
        0 // o return 0  
    }  
    catch (SinUnMangoException sme) {  
        -1 // o return -1  
    }  
}
```

8 Bibliografía complementaria

Tutorial de Excepciones de Sun “Handling Errors with Exceptions”

<http://java.sun.com/docs/books/tutorial/essential/exceptions/>

Discusiones sobre excepciones

<http://c2.com/cgi/wiki?CheckedExceptionsAreOfDubiousValue>

<http://c2.com/cgi/wiki?JavaExceptionsAreParticularlyEvil>