

TEMA 4

Corrección y Robustez

Facultad de Informática
Universidad de Murcia

Contenido

1. Introducción
 - Fiabilidad = Corrección y Robustez
2. Aertos y Técnica de diseño por contrato
3. Abordando los casos excepcionales

Introducción

- “La reutilización y la extensibilidad no se deben lograr a expensas de la **fiabilidad** (*corrección y robustez*)”.
- **Corrección:**
 - Capacidad de los sistemas software de ajustarse a la especificación.
 - Asegura que el programa hace lo correcto durante la ejecución normal del programa.
 - Los **asertos** permiten especificar la semántica de una clase, establecen las condiciones que se deben cumplir.
- **Robustez:**
 - Capacidad de los sistemas software de reaccionar ante circunstancias inesperadas.
 - El **mecanismo de excepciones** proporciona un mecanismo para manejar estas situaciones excepcionales durante la ejecución de un programa.

Corrección. Asertos

(A) Especificar la **semántica de las rutinas** mediante:

– **PRECONDICIONES:**

Condiciones para que una rutina funcione adecuadamente.

– **POSTCONDICIONES:**

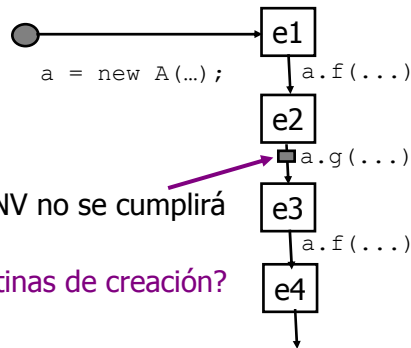
Describen el efecto de una rutina, definiendo el estado final.

(B) Especificar las **propiedades globales de una clase** mediante el **INVARIANTE:**

Aserción que expresa restricciones de integridad que deben ser satisfechas por cada instancia de la clase si se encuentra en una **situación estable**.

Momentos "estables"

- Los "momentos estables" son aquellos en los que una instancia está en un estado observable (e1, e2, ...):
 - Después de la creación
 - Antes y después de la invocación remota de una rutina de la clase



En etapas intermedias el INV no se cumplirá

¿Cuál es el papel de las rutinas de creación?

Ejemplo: Pre y Postcondiciones

```
/**
 * Inserta en el mapa un elemento con
 * clave key
 * @pre count < capacity mapa no lleno
 *       key != "" Clave significativa
 * @post count <= capacity;
 *        item (key) = elemento;
 *        count = oldCount + 1;
 */
public void put (T elemento, String key){
    ...
}
```

Ejemplo invariante

```
/**
 * Clase que representa figuras
 * geométricas 2D
 * @inv numVertices > 3
 */
public class Poligono{
    ...
}
```

- Todas las instancias de polígono tienen que cumplir que el número de vértices sea mayor de tres.

Utilidad de los asertos

- Escribir software correcto:
describir requisitos exactos de cada rutina y las propiedades globales de las clases ayuda a producir software que es correcto desde el principio.
- Ayuda para la documentación:
pre, post condiciones e invariante proporcionan información precisa a los clientes de los módulos.
- Apoyo para la prueba y depuración:
el programador establece como opción del compilador el efecto de las aserciones en tiempo de ejecución.

Asertos en Java

- El mecanismo de aserciones **permite insertar pruebas** durante la depuración y luego eliminarlas automáticamente.
- A partir del JDK1.4 el lenguaje incluye la palabra reservada **assert**
- Expresión booleana que un desarrollador indica explícitamente que se debe cumplir en un punto del programa en tiempo de ejecución.

```
assert expresion_boolean;  
assert expresion_boolean : expresion;
```

- El sistema evalúa la expresión booleana e informa del error en el caso de que el valor sea **false**.
 - Lanza **AssertionError**

Ejemplos

```
assert ref != null;  
assert saldo == (oldSaldo + cantidad);  
assert ref.m1(parametro);  
assert valor>0 : "argumento negativo";  
assert x>0 : x;
```

- La `expresion_booleana` **no debe** tener efectos laterales
- La expresión se pasa como argumento al constructor de la excepción en el caso de que el aserto sea `false`.

Activación de asertos en Java

- Los asertos los soporta sólo a partir de la [versión 1.4](#). Se debe indicar explícitamente al compilador para que reconozca la palabra clave `assertion`

```
javac -source 1.4 MiAplicacion.java
```

- La comprobación de los asertos se puede [desactivar](#) en tiempo de ejecución para incrementar el rendimiento.

- Normalmente se activa durante la fase de depuración y pruebas

```
java [-ea|-da] MiAplicacion
```

```
java -ea:UnaClase -ea:java.modelo -da:OtraClase App
```

- A partir de la versión JDK 5.0 el soporte para aserciones está activado por defecto.

Usos de los asertos

- Invariantes internos:

```
if (i%3 == 0){
...}else if (i%3 == 1){
...} else {
    assert (i%3 ==2);
    ...
}
```

- Invariantes de flujo de control:

```
void met(){
    for (...){
        if(...) return;
    }
    //nunca deberíamos llegar a este punto
    assert false;
}
```

- **Precondiciones, postcondiciones e invariantes:**

Pre y postcondiciones en Java

```
private void setIntervalo(int intervalo){  
    //precondición  
    assert intervalo>0 && intervalo<=MAXIMO;  
  
    ...//establecer el intervalo  
}
```

```
//devuelve  $\text{this}^{-1} \bmod m$   
public BigInteger modInverso (BigInteger m){  
    //comprobaciones de la precondición  
    ...  
    //hacer el cálculo  
    //postcondición  
    assert this.multiply(result).mod(m).equals(UNO);  
    return result;  
}
```

Corrección y Robustez

13

Invariante en Java

```
public class Pila{  
    //método que comprueba el invariante  
    private boolean invariante() {  
        return ((count>=0) && (count <=capacity) &&  
                (capacity == representation.length));  
    }  
    ...  
}
```

- Se debe cumplir antes y después de la terminación de cada método.
- Todo método y constructor debe contener la línea
assert invariante();
antes de empezar e inmediatamente antes de que termine.

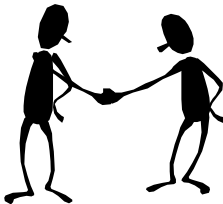
Corrección y Robustez

14

¿Cuándo utilizar assert?

- Localizar **errores internos irrecuperables** durante la fase de pruebas y depuración.
- No utilizar para evaluar condiciones externas al programa (existencia de un fichero, conexión de red, ...)
- Puesto que se pueden desactivar, **la corrección de un programa no puede depender de los asertos.**

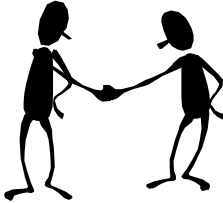
Técnica del Diseño por Contrato



Definir una pre y una postcondición para una rutina es una forma de definir un **contrato** que liga a la rutina con quien la llama.

put	Obligaciones	Beneficios
Cliente	Al invocar a <i>put</i> debe asegurar que la tabla no está llena	Obtiene una tabla en la que <i>elemento</i> está asociado con <i>clave</i>
Servidor	Insertar <i>elemento</i> en la tabla asociándolo a <i>clave</i>	No necesita tratar la situación de tabla llena antes de la inserción

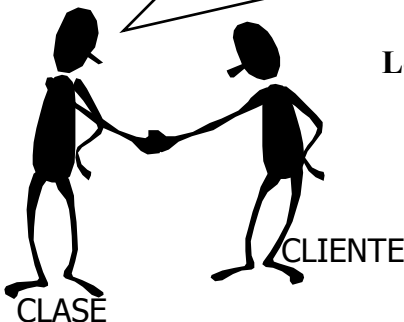
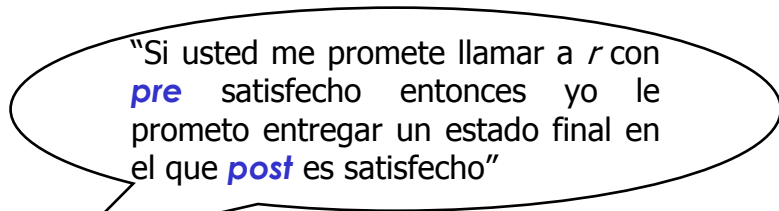
Técnica del Diseño por Contrato



Definir una pre y una postcondición para una rutina es una forma de definir un **contrato** que liga a la rutina con quien la llama.

put	Obligaciones	Beneficios
Cliente	Satisfacer precondición	De la postcondición
Servidor	Satisfacer postcondición	De la precondición

Contrato Software



Lo que es una **obligación** para uno es un **beneficio** para el otro

Rechazo a la "programación defensiva"

```
sqrt(x: REAL): REAL is do
  if x<0 then "Manejar error"
  else "Calcular raíz"
end;
```

No hay que comprobar la precondición en la rutina

```
sqrt(x: REAL): REAL is
  require
    x >= 0;
  do
    "Calcular raíz"
  end;
```

CLIENTE:
if (x>=0)
 Math.sqrt(x);

El cuerpo de la rutina no comprueba el cumplimiento de la precondición

Rechazo a la "programación defensiva"

- Redundancia es perjudicial: **software más complejo.**
iiLa complejidad es el mayor enemigo de la calidad!!
- Mejor identificar condiciones y establecer responsabilidades.
- Tedioso eliminar o ignorar los controles cuando no se necesiten
- Paradoja: "La fiabilidad se mejora chequeando menos"
iiGarantizar mas comprobando menos!!
"El código cliente debe comprobar la precondición"

Diseño por Contrato y assert

```
/** Insertar en la tabla elemento con clave key */  
public void put (Object elemento, String key) {  
    assert count < capacity: "fallo en la pre";  
    int oldCount = count;  
    ... "algoritmo de inserción"  
    assert count <= capacity &&  
        item (key).equals(elemento) &&  
        count == oldCount + 1: "fallo en la post";  
}
```

El control de las precondiciones no se debe dejar en manos de los assert (excepciones Runtime)

3.- Abordando los casos excepcionales

Técnicas de diseño {

- Esquema a priori
- Esquema a posteriori
- Mecanismo de excepciones

• Esquema a priori:

- Se pide al cliente que tome medidas por adelantado para evitar posibles errores
- Los errores en ejecución implican un error del cliente.

<pre>if pre(y) then operacion(y) else --acción alternativa end</pre>	<pre>operacion(x:...) is require pre(x) do ...acción si pre= true end</pre>
----------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------

Problemas del esquema a priori

- **Problemas de eficiencia:**
 - No siempre es posible comprobar primero la precondition.
 - **Ejemplo:** calcular si una matriz es o no singular antes de calcular su inversa.
- **Limitaciones de los lenguajes de asertos:**
 - Algunas aserciones no se pueden expresar.
 - Cuando la precondition es una propiedad global de una estructura de datos y necesita cuantificadores.
 - **Ejemplo:** comprobar que un grafo no tiene ciclos.
- **El éxito depende de eventos externos:**
 - Es imposible comprobar la aplicabilidad sin ejecutarla.
 - **Ejemplo:** una línea de comunicaciones

Esquema a posteriori

- Probar después de la ejecución de la operación.
- Sólo es posible en algunas ocasiones.

<pre>obj:A x:INTEGER obj.operacion(y) if obj.exito then x:= obj.resultado else ...manejar el error end</pre>	<pre>class A feature éxito: BOOLEAN resultado:INTEGER operacion(x:...)is do ...acciones ...actualiza éxito y resultado end end</pre>
----------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Mecanismo de excepciones

- Existen casos en los que no es posible utilizar las técnicas anteriores:
 - errores del hardware o del sistema operativo
 - detección de errores tan pronto como sea posible aunque no se pueda detectar con una precondición
 - tolerancia frente a fallos del software
- En estos casos parece necesarias las técnicas basadas en excepciones.

Cuando se rompe el contrato: tratamiento de excepciones

- Buscar un *equilibrio* entre:
 - **CORRECCIÓN** = comprobar todos los errores
 - **CLARIDAD** = no desordenar el código del flujo normal con excesivas comprobaciones
- Solución elegante => **MECANISMO DE EXCEPCIONES**
 - Separa el código de trabajo del código que maneja el error mediante cláusulas **try-catch**
 - Permite la **propagación de errores** de manera ordenada. Si el método al que se invoca encuentra una situación que no puede manejar, pueda lanzar una excepción y dejar que la trate el método que le llamó.
 - Se **registran las situaciones "excepcionales"** anticipadamente de manera que el compilador puede asegurar que se tratan.
- **Excepción:** suceso inesperado o no deseado

Cláusula try-catch

- Se evalúan en orden
- Sólo se ejecuta una

Una excepción es un objeto!!

```
try{
    //sentencias comportamiento normal
} catch (TipoExcepcion e){
    //tratamiento recuperación o re-throw
} catch (TipoExcepcion2 e){
    //tratamiento
}
...
finally{
    //sentencias que se hacen SIEMPRE
    //salte o no una excepción
}
```

Ej: cerrar ficheros

Cualquier nº. Si no hay se pasa al código que invocó este método

Corrección y Robustez

27

¿Cuándo se lanza una excepción?

- **ORIGEN** de una excepción:
 - “se rompe el contrato”
 - (fallo en la precondition o postcondición)
 - “condición anormal”
- Un método lanza una excepción cuando se encuentra con una **condición anormal** que no puede manejar.
 - Ejemplo: EOF cuando leemos enteros de un `DataInputStream` (no puede ser -1).
 - Ejemplo: invocar `nextToken()` sin ningún token en el `StringTokenizer`
- Evite utilizar excepciones para indicar condiciones que son razonables que ocurran como parte del funcionamiento típico de un método.

Corrección y Robustez

28

Ejemplo: Se rompe el contrato

- En la clase String:

```
public char charAt(int index) {
    if ((index < 0) || (index >= count)) {
        throw new StringIndexOutOfBoundsException(index);
    }
    return value[index + offset];
}
```

- Si el cliente llama `charAt(-1)` rompe el contrato (no cumple la **precondición**) y se le debe informar de ello lanzando una excepción (`StringIndexOutOfBoundsException`).
- Si el método encuentra problemas con los recursos runtime y es incapaz de devolver el carácter en la posición solicitada (no puede cumplir la **postcondición**) debe indicarlo lanzando una excepción.

¿Comportamiento "normal" o excepcional?

```
A. Pasajero getPasajero(){
    try{
        Pasajero p = bd.buscarPasajeroRegistroVuelo("Ana Col");
    }catch (PasajeroNoEncontradoException e){
        //hacer algo
    }
}
```

Que la búsqueda devuelva un conjunto vacío forma parte del procesamiento **normal**

```
B. Pasajero getPasajero(){
    Pasajero p = bd.buscarPasajeroRegistroVuelo("Ana Col");
    if (p==null) //¿Forma parte de la semántica de getPasajero?
        throw new PasajeroNoEncontradoException();
        //situación excepcional
}
```

¿Valores especiales de retorno o excepciones?

Valores especiales

a) ¿Se puede distinguir siempre el resultado especial de un resultado normal?

- `List>>indexOf()` devuelve un -1 cuando no se encuentra el elemento en la lista porque el índice -1 no es un índice válido.
- `DataStream>>getInt()` no puede indicar el EOF como -1 porque es uno de sus valores válidos.

b) ¿Qué ocurre si al que llama se le olvida comprobar el valor especial de retorno?

```
Empleado e = plantilla.buscar(dni);  
e.subirSueldo() → NullPointerException
```

- Las excepciones no tienen ninguno de estos problemas

Tipos de excepciones en Java

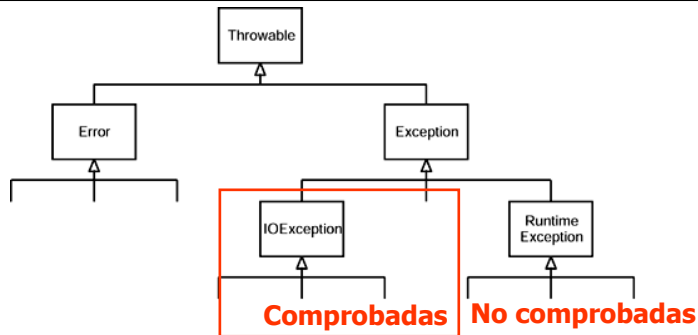
• Comprobadas (**Exception**):

- Indican un fallo del método en cumplir su contrato (falla la **postcondición**).
- Si se lanza una excepción comprobada en un método hay que declararla en la cláusula `throws`.
- Fuerzan a los clientes a tratar la excepción potencial.
- Ejemplo: `IOException`

• No comprobadas (**RuntimeException**):

- Indican un uso inadecuado de la clase (fallo en la **precondición**).
- El cliente decide si capturarla o ignorarla.
- Ejemplo `StringIndexOutOfBoundsException`

Jerarquía de excepciones en Java



- La jerarquía `Error` describe errores internos y agotamiento de recursos del sistema de ejecución de Java.
- El programador no debe lanzar objetos de tipo `Error`.
- El programador debe centrarse en las excepciones de tipo `Exception`.

Manejo de excepciones en Java

- Si se invoca a un método `m1` que tiene una excepción comprobada `e1` en su cláusula `throws`, en un método `m2` existen tres opciones:
 - 1) `m2` **capturar** la excepción **y la gestiona**.
 - 2) `m2` **capturar** la excepción **y la transforma** en una excepción `e2` declarada en su cláusula `throws`
 - 3) `m2` declara la excepción `e1` en su cláusula `throws` y **hace que pase** por el método (aunque puede incluir algún tratamiento en la cláusula `finally`)

Ejemplo caso 1

```
Pasajero getPasajero(){
    try{
        Pasajero p = bd.buscarPasajeroRegistroVuelo("Ana
Col");
    }catch (MalformedURLException mue){
        //hacer algo
    }catch (SQLException sqle){
        //hacer algo
    }
}
```

- Necesita devolver un valor especial para señalar el error al método que llame a `getPasajero()`
- El método que llama debe controlar todos los posibles valores de retorno

Ejemplo caso 3

```
Pasajero getPasajero() throws MalformedURLException,
    SQLException{
    Pasajero p = bd.buscarPasajeroRegistroVuelo("Ana
Col");
}
```

- Se pasa hacia arriba toda la responsabilidad de manejo de las situaciones excepcionales.
- Problema cuando existen múltiples límites entre sistemas.

Ejemplo caso 2

```
Pasajero getPasajero() throws ViajeException{
    try{
        Pasajero p = bd.buscarPasajeroRegistroVuelo("Ana Col");
    }catch (MalformedURLException mue){
        //hacer algo
        throw (new ViajeException("Fallo búsqueda",mue);
    }catch (SQLException sqle){
        //hacer algo
        throw (new ViajeException("Fallo búsqueda",sqle);
    }
}
```

- Transforma una excepción de nivel de sistema en una de nivel de aplicación.
- Solución elegante al problema del caso3.

Definición de nuevas excepciones

```
public class ViajeException extends Exception
{
    private Exception excepcionOcultas;
    public ViajeException(String error, Exception e){
        super(error);
        excepcionOcultas = e;
    }
    public Exception getExcepcionOcultas(){
        return excepcionOcultas;
    }
}
```

- Podemos encapsular la excepción de bajo nivel para conservar la información de la excepción que tuvo lugar.
- Razones: nombre más significativo, añadir información.

Definir un nuevo tipo de excepción

```
public class NuevaExcepcion extends
    [Exception|RuntimeException] {
    public NuevaExcepcion () {
        super();
    }

    public NuevaExcepcion (String s){
        super(s);
    }
}
```

¿Excepción comprobada o no comprobada?

- Se debe definir una excepción **no comprobada** si se espera que el usuario escriba código que asegure que no se lanzará la excepción, porque:
 - Existe una forma adecuada y no costosa de evitar la excepción.
 - El contexto de uso es local.
 - Ejemplo: `EmptyStackException`
- En otro caso, la excepción será **comprobada**.
 - Ejemplo: `FileNotFoundException`

• Utiliza excepciones comprobadas para los resultados especiales y las excepciones no comprobadas para fallos.

Guías

- Si el método encuentra una **situación anormal** que no puede manejar, debe lanzar una excepción.
- Evite utilizar excepciones para indicar condiciones que forman parte del funcionamiento normal del método.
- Si se lanza una excepción por una condición anormal que el **cliente debería tratar**, debe ser una excepción **comprobada**.
- Si el cliente rompe su parte del contrato (**precondición**) lanza una excepción **no comprobada**.
- Si el método es incapaz de cumplir su parte del contrato (**postcondición**) lanza una excepción **comprobada**.

Consejos para el uso de excepciones (1/2)

- El manejo de excepciones no debe sustituir a una sencilla comprobación

<pre>if (!pila.empty) pila.pop();</pre>	<pre>try{ pila.pop(); } catch (EmptyStackException e){ //tratar }</pre>
---------------------------------------------	---------------------------------------------------------------------------------

Intentar 1.000.000 de veces sacar información de una pila vacía

154 milisegundos

10739 milisegundos

Consejos para el uso de excepciones (2/2)

- Toda la ejecución normal en un bloque try en lugar de probar cada sentencia en un bloque try
- Utilizar la jerarquía de excepciones convenientemente
 - No lanzar `RuntimeException` sino la subclase adecuada
 - No capturar `Throwable`, especificar el tipo de excepción que se captura para mejorar el mantenimiento y la legibilidad
- No *silenciar* las excepciones: `catch (Exception e) {}`
- Deja pasar una excepción hasta el manejador más adecuado

```
void imprimirPuntoCorte(){
    try{
        double x = ecuacion2°grado(a,b,c);
        system.out.println(""+x+",0");
    }catch (NotRealException e){
        System.out.println("No corta al eje\n");
    }
}

double ecuacion2°Grado(double a, double b, double c) throws
    NotRealException{
    try{
        return (-b + sqrt( b*b -4*a*c))/(2*a);
    }catch (IllegalArgumentException e){
        throw new NotRealException();
    }
}

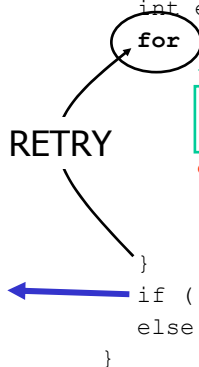
public double sqrt(double x) throws IllegalArgumentException{
    if (x<0) throw new IllegalArgumentException();
    ...
}
```

¿Exception o Runtime?

Ejemplo manejo de excepciones en Java

```
public int LeerEnteroTeclado() throws IOException {  
  
    boolean fin= false;  
    int fallos;  
    int enteroLeido=-1;  
    for (fallos=0; ((fallos<=5) && (fin==false));fallos++){  
        try{  
            enteroLeido=getInt();  
            fin = true;  
        } catch (NumberFormatException e){  
            System.out.println("Un entero!!");  
        };  
    }  
    if (!fin) throw (new IOException());  
    else return enteroLeido;  
}
```

RETRY



Corrección y Robustez

45

Diseño por Contrato y Java

```
/** Insertar en la tabla elemento con clave key */  
public void put (Object elemento, String key) {  
    if (count >= capacity) throw new FullMapException();  
  
    int oldCount = count;  
  
    ... "algoritmo de inserción"  
  
    assert count <= capacity &&  
           item (key).equals(elemento) &&  
           count == oldCount + 1: "fallo en la post";  
}
```

El control de las precondiciones se debe implementar
Utilizando excepciones Runtime

Corrección y Robustez

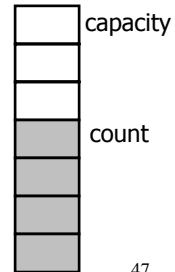
46

Diseño por Contrato y Java

```
public class Stack{
    ...
    /**
     * @pre: Pila no vacía
     * @post: Pila no llena & tiene un elemento menos
     * @throws EmptyStackException si la pila está vacía */
    public void remove() {
        PRE if (this.empty())
            throw (new EmptyStackException());
        count = count -1;
        POST assert !this.full(): "Pila llena";
    }
}
```

- El cliente en ambos casos tendrá que:

```
if (!pila.empty())
    pila.remove();
```



Ejercicio

[Exam Sept-06]

```
/**
 * Representa los números racionales
 * @inv denominador > 0
 */
public class Fraccion{
    private int numerador;
    private int denominador;
    public Fraccion (int n, int d){
        numerador = n;
        denominador = d;
    }
    /** Cambia numerador por denominador
     * @pre numerador != 0
     * @post numerador = old denominador AND
     *         denominador = old numerador
     */
    public void inversa()
    {
        int aux;
        aux = numerador;
        numerador = denominador;
        denominador = aux;
    }
}
```


Herencia y excepciones en Java

- Si redefinimos un método en la subclase, éste no puede lanzar más excepciones comprobadas que el método de la superclase que está redefiniendo.
- El método de la subclase puede lanzar menos.
- Si el método de la superclase no lanza ninguna excepción comprobada, tampoco puede hacerlo el método redefinido de la subclase.